В. Н. Бондарев
Ф. Г. Аде

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ



В. Н. БОНДАРЕВ
Ф.Г. АДЕ

ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ

Рекомендовано Министерством образования и науки Украины
в качестве учебного пособия для студентов высших учебных заведений

2002

ББК 32.813
Б811
УДК 004.8(075)+004.93(075)

Рецензенты:
Мохор В. В., д-р техн. наук, заведующий отделом специализированных средств моделирования Института проблем моделирования в энергетике им. Г.Е. Пухова НАН Украины;
Суворов А. М., д-р физ.-мат. наук, заведующий отделом морских информационных систем и технологий Морского гидрофизического института НАН Украины;
Теленик С.Ф., д-р техн. наук, профессор, заведующий кафедрой автоматики и управления в технических системах Национального технического университета Украины "Киевский политехнический институт"

Решение МОН Украины о присвоении грифа "Рекомендовано в качестве учебного пособия для студентов высших учебных заведений" № 14/18.2-1313 от 17.06.2002

Бондарев В.Н.
Б811 **Искусственный интеллект:**Учеб. пособие для вузов/ В.Н. Бондарев, Ф.Г. Аде. - Севастополь: Изд-во СевНТУ, 2002. – 615с.: ил.

ISBN 966-7473-45-7

В пособии рассматриваются универсальные модели и методы, применяемые при создании компьютерных систем, способных выполнять функции, традиционно считающиеся интеллектуальными: понимание естественного языка и изображений, приобретение знаний и обучение, логический вывод, планирование действий. Приводятся многочисленные примеры и программы, поясняющие и детализирующие общие принципы. Подробно рассматриваются языки Коммон Лисп и Пролог, широко используемые при создании интеллектуальных систем. Большое внимание уделяется основным направлениям искусственного интеллекта – экспертным системам, нейронным сетям, обработке естественного языка, машинному зрению. Пособие написано в соответствии с международной программой подготовки специалистов в области информационных систем.

Книга предназначена для студентов вузов, обучающихся по направлению "Компьютерные науки". Она будет полезна студентам, обучающимся по направлениям "Компьютерная инженерия", "Компьютеризированные системы автоматики и управления" и др., а также широкому кругу специалистов, интересующихся вопросами построения интеллектуальных систем.

ISBN 966-7473-45-7 © Бондарев В.Н., Аде Ф.Г., 2002

СОДЕРЖАНИЕ

ПРЕДИСЛОВИЕ……... 9
СПИСОК СОКРАЩЕНИЙ…….. 12
Глава 1. ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ…..... 13
1.1. Понятие “искусственный интеллект”……. 13
1.2. Этапы развития искусственного интеллекта …….. 15
1.3. Основные направления исследований…… 22
1.3.1. Представление задач и поиск решений…….. 22
1.3.2. Доказательство теорем…… 23
1.3.3. Представление знаний…….. 23
1.3.4. Экспертные системы……. 24
1.3.5. Обучение и выявление закономерностей…… 25
1.3.6. Общение на естественном языке…….. 26
1.3.7. Распознавание образов…….. 27
1.3.8. Компьютерное зрение…… 27
1.3.9. Языки программирования……. 28
1.4. Структура систем с искусственным интеллектом…… 28
Вопросы для самопроверки…… 33

Глава 2. СПОСОБЫ ПРЕДСТАВЛЕНИЯ ЗАДАЧ И ПОИСК РЕШЕНИЙ…… 34
2.1. Общая характеристика способов представления задач…… 35
2.1.1. Представление задач в пространстве состояний……. 35
2.1.2. Сведение задач к подзадачам…… 39
2.1.3. Представление задач в виде теорем……. 44
2.2. Поиск решений задач в пространстве состояний……. 45
2.2.1. Методы “слепого” поиска…….. 46
2.2.1.1. Случайный поиск…….. 46
2.2.1.2. Поиск “в глубину и ширину”…….. 46
2.2.1.3. Алгоритм равных цен…… 47
2.2.2. Эвристический поиск……. 49
2.2.2.1. Алгоритм “подъема на гору”……..…….. 49
2.2.2.2. Глобальный учет соответствия цели…… 50

2.2.2.3. А-алгоритм……. 52
2.2.3. Поиск с распространением ограничений…….. 58
2.3. Поиск решений при сведении задач к подзадачам…….. 60
2.3.1. Алгоритм поиска в глубину ……..…….. 61
2.3.2. Алгоритм эвристического поиска в И-ИЛИ графе…. 64
2.3.3. Поиск решений в игровых программах…… 68
2.3.3.1. Минимаксный метод…… 69
2.3.3.2. Альфа-бета поиск……. 70
Вопросы для самопроверки……. 72

Глава 3. ПРЕДСТАВЛЕНИЕ ЗНАНИЙ…….. 73
3.1. Знания и их представление в СИИ…….. 74
3.2. Логические модели…… 77
3.2.1. Формальные системы…….. 77
3.2.2. Исчисление высказываний…….. 79
3.2.3. Исчисление предикатов……... 85
3.2.4. Логики высокого порядка и
псевдофизические логики……... 91
3.3. Продукционные модели…… 93
3.3.1. Продукционные системы……. 93
3.3.2. Управление выводом в продукционных системах…… 98
3.4. Семантические сети…….. 104
3.4.1. Общее понятие о семантических сетях……. 104
3.4.2. Способы описания семантических сетей
и логический вывод…….…… 110
3.5. Фреймы…….. 117
3.5.1. Структура фрейма…… 117
3.5.2. Управление выводом…….. 120
Вопросы для самопроверки …… 128

Глава 4. ОСНОВНЫЕ МОДЕЛИ ВЫВОДА……. 130
4.1. Дедуктивный вывод в исчислении предикатов……. 131
4.1.1. Формулировка задачи дедуктивного вывода…… 131
4.1.2. Стандартизация предикатных формул…….. 132
4.1.3. Метод Эрбрана……. 134
4.1.4. Принцип резолюции……. 136
4.1.5. Стратегии поиска……. 141
4.1.6. Применения принципа резолюции……. 143
4.1.6.1. Информационный поиск…… 143
4.1.6.2. Планирование перемещений робота……. 146
4.1.6.3.Автоматическое написание программ…… 148

4.1.7. Принцип резолюции и система STRIPS……. 152
4.1.8. Принцип резолюции и язык Пролог……... 158
4.2. Выводы в условиях ненадежных или неполных знаний…… 163
4.2.1. Виды неопределенности…… 163
4.2.2. Ненадежные знания и выводы…….. 164
4.2.2.1. Байесовский метод…… 164
4.2.2.2. Метод коэффициентов уверенности…… 170
4.2.2.3. Теория свидетельств Демпстера-Шефера…… 172
4.2.2.4. Нечеткие множества и нечеткая логика…….. 175
4.2.3. Неполнота знаний и немонотонный вывод…….. 185
4.3. Индуктивный вывод……. 192
4.3.1. Основные понятия индуктивного обобщения…….. 193
4.3.2. Алгоритм Уинстона……. 195
4.3.3. Алгоритм Митчелла……. 199
4.3.4. Алгоритм обобщения ID3…… 204
Вопросы для самопроверки……. 208

Глава 5. ЯЗЫК ЛИСП……. 210
5.1. Общая характеристика языка…… 210
5.2. Основные понятия языка…….. 212
5.3. Интерпретация Лисп-программ…… 216
5.4. Арифметические функции…… 218
5.5. Основные функции обработки списков…….. 220
5.6. Присваивание значений…… 223
5.7. Предикаты…….. 224
5.8. Определение функций…….. 226
5.9. Связывание и область действия переменных……. 230
5.10. Последовательные и условные вычисления…… 234
5.11. Итерационные циклические вычисления…… 237
5.12. Рекурсивные функции……... 240
5.13. Структурно-разрушающие функции…… 244
5.14. Ассоциативные списки и списки свойств…… 247
5.15. Типы данных…….. 249
5.16. Ввод-вывод…… 257
5.17. Функционалы……. 260
5.18. Макролитеры……. 265
5.19. Замыкания…….. 267
5.20. Структуры…….. 268
5.21. Объектно-ориентированное программирование…… 271
5.22. Макросы……. 279
5.23. Пакеты…… 282

5.24. Множественные значения…… 285
5.25. Программирование задач ИИ на языке ЛИСП…….. 286
5.25.1. Случайный поиск…… 286
5.25.2. Поиск в глубину и ширину…… 288
5.25.3. Эвристический поиск на основе А-алгоритма……. 290
5.25.4. Сопоставление с образцом……. 294
5.25.5. Поиск в семантических сетях……. 305
Вопросы для самопроверки …….. 307

Глава 6. ЯЗЫК ПРОЛОГ……. 309
6.1. Общая характеристика языка……. 309
6.2. Основные понятия языка…… 310
6.3. Операторы…… 318
6.4. Арифметические выражения……. 320
6.5. Списки и рекурсия…….. 322
6.6. Управление возвратом (отсечение)…… 325
6.7. Отрицание в языке Пролог……. 329
6.8. Метаусловия……. 330
6.9. Организация циклов……. 331
6.10. Встроенные предикаты…… 335
6.10.1. Предикаты ввода-вывода…….. 336
6.10.2. Предикаты обработки термов…… 340
6.10.3. Предикаты работы с базой данных…… 343
6.11. Решение задач ИИ на языке Пролог…… 345
6.11.1. Создание и обработка бинарных деревьев…… 345
6.11.2. Поиск решений в пространстве состояний…… 346
6.11.2.1. Поиск в глубину …… 348
6.11.2.2. Поиск в ширину……. 354
6.11.2.3. Эвристический поиск……. 356
6.11.3. Поиск решений в И-ИЛИ графах…….. 360
6.11.4. Поиск в пространстве версий……. 362
6.11.5. Поиск в сети фреймов……. 367
Вопросы для самопроверки …….. 369

Глава 7. ЭКСПЕРТНЫЕ СИСТЕМЫ …… 371
7.1. Основные функции и компоненты экспертных систем……. 372
7.2. Разработка экспертных систем……. 375
7.3. Приобретение знаний…… 379
7.4. Поиск и объяснение решений…….. 382
7.5. Реализация экспертной системы на языке Лисп…… 389
7.6. Реализация экспертной системы на языке Пролог…… 396
7.6.1. Представление знаний правилами…… 396

7.6.2. Представление знаний фреймами……. 406
Вопросы для самопроверки……. 412

Глава 8. РАСПОЗНАВАНИЕ ОБРАЗОВ И ОБУЧЕНИЕ..... 413
8.1. Основные сведения о распознавании образов…… 414
8.2. Геометрический метод распознавания…… 415
8.3. Байесовский метод распознавания…….. 421
8.4. Рекуррентные алгоритмы обучения распознаванию образов……. 426
8.5. Распознавание и обучение на основе моделей нейронных сетей……. 429
8.5.1. Модели нейронных элементов…….. 429
8.5.2. Структуры нейронных сетей……. 434
8.5.3. Виды обучения ИНС…….. 437
8.5.4. Обучение с учителем в ИНС с прямыми связями…… 438
8.5.4.1. Простой персептрон……. 438
8.5.4.2. Адаптивный линейный элемент…… 441
8.5.4.3. Многослойный персептрон…… 443
8.5.5. Обучение без учителя……. 451
8.5.5.1. ИНС, основанные на правиле обучения Хебба……. 451
8.5.5.2. Состязательные ИНС……. 453
8.5.5.3. Неокогнитрон……. 463
8.5.6. ИНС с фиксированными весами связей и ассоциативная память…… 467
8.5.6.1. Ассоциативная память и сеть Хэмминга…. 467
8.5.6.2. Сеть Хопфилда…… 471
8.5.7. Коннекционистские экспертные системы…… 477
8.6. Синтаксический метод распознавания…… 482
Вопросы для самопроверки ……. 487

Глава 9. ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА…… 489
9.1. Компоненты ЕЯ-системы…… 489
9.2. Понимание ЕЯ-высказываний…… 493
9.2.1. Рациональный подход…….. 494
9.2.2. Эмпирический подход…….. 504
9.3. ЕЯ-интерфейс доступа к базам данных……. 508
9.4. Распознавание речи…….. 517
9.4.1. Основные понятия…….. 518
9.4.2. Предварительная обработка и распознавание звуков. 521
9.4.3. Статистический подход к распознаванию речи…….. 524
9.4.4. Модель языка……. 526

9.4.5. Акустическая модель…….. 528
9.4.6. Композиция моделей……. 534
9.4.7. Алгоритмы поиска……. 536
9.4.8. Оценивание параметров СММ…….. 540
9.4.9. СММ с непрерывными параметрами…… 541
9.5. Синтез речи по тексту…… 543
9.5.1. Основные понятия…… 543
9.5.2. Методы синтеза речевых сигналов…… 545
9.5.3. Определение управляющих параметров синтезаторов речи…… 549
Вопросы для самопроверки…….. 554

Глава 10. КОМПЬЮТЕРНОЕ ЗРЕНИЕ…….. 557
10.1. Основные понятия……. 557
10.2. Система зрения человека…….. 561
10.3. Системы компьютерного зрения…….. 571
10.4. Выделение граничных элементов…….. 574
10.5. От граничных элементов к граничным сегментам……. 577
10.5.1. Преобразование Хафа…….. 578
10.5.2. Поиск при выделении контурных сегментов……. 581
10.5.3. Активные контурные модели…….. 583
10.6. Выделение областей изображения…….…… 584
10.7. Интерпретация контурных рисунков…..…… 585
10.8. Распознавание объектов в системах зрения роботов…… 589
10.8.1. Системы зрения роботов и компьютерное зрение… 589
10.8.2. 2D-модели……. 590
10.8.3. 3D-методы……. 591
10.9. Примеры методов распознавания изображений……. 591
10.9.1. 2D-методы, использующие глобальные признаки… 591
10.9.2. 2D-методы, использующие локальные признаки….. 593
10.9.3. 2D-метод, использующий граф отношений…….. 594
10.9.4. Пример 3D-метода……. 598
Вопросы для самопроверки……. 602

БИБЛИОГРАФИЧЕСКИЙ СПИСОК…….. 604
ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ…….. 609

ПРЕДИСЛОВИЕ

Дисциплина “Искусственный интеллект” играет фундаментальную роль в подготовке специалистов в области информатики и вычислительной техники. Различные учебные курсы, относящиеся к проблематике искусственного интеллекта (ИИ), включены в учебные планы многих университетов. Ассоциациями ACM, AIS и AITP разработана программа подготовки специалистов в области информационных систем, которая рекомендуется для международного использования[1)]. Программой предусматривается в рамках дисциплины “Искусственный интеллект” рассмотрение следующих тем: история, основные понятия и направления развития ИИ; представления задач и пространства поиска решений; основные стратегии управления выводом (поиск в глубину и в ширину, прямой и обратный вывод); эвристический поиск; представление знаний; экспертные системы и оболочки; нечеткая логика и нечеткий вывод; машинное обучение; распознавание образов; нейронные сети; обработка естественного языка; распознавание речи и машинное зрение.

Несмотря на обилие книг, посвященных вопросам ИИ, учебное пособие, в котором бы систематически рассматривались все указанные выше темы, на территории СНГ не издавалось. Большинство книг было издано в 70-е и 80-е годы, т.е. почти четверть века назад. Многие из них являются научными, справочными или научно-популярными изданиями и не подходят для использования в учебном процессе. Исключением является изданный недавно учебник Т. А. Гавриловой и В. Ф. Хорошевского (см. библиографический список). Однако он преимущественно посвящен вопросам инженерии знаний и системам ИИ, основанным на знаниях.

Цель издания учебного пособия – обеспечить систематическое рассмотрение всего спектра тем, рекомендованных для изучения программой подготовки специалистов в области информационных систем, включая как изложение общих подходов и принципов ИИ, так и алгоритмических основ

[1)] Davis G. B. IS’97 Model Curriculum and Guidelines for undergraduate Degree Programs in Information Systems / G.B. Davis, J.T. Gorgone, J.D. Couger, D.J. Feinstein , H.E. Longenecker – Park Ridge: AITP inc., 1997. – 94 p.

их реализации. Последнее обстоятельство представляется весьма важным. Включение в учебное пособие большого числа примеров конкретных алгоритмов, применяемых в системах ИИ, и иллюстрация их с помощью языков Лисп и Пролог позволяет глубже понять излагаемые в книге универсальные модели и методы ИИ. Наличие таких примеров создает необходимую базу для экспериментального изучения ИИ и разработки на этой основе собственных программ ИИ.

Для изучения учебного пособия требуются знания в области программирования и ЭВМ, высшей математики, теории информации и математической статистики. Материал учебного пособия логически состоит из двух частей и рассчитан на двухсеместровый курс с общим объемом 216 часов. В первой части рассматриваются общие принципы ИИ, языки Лисп и Пролог (главы 1 – 6). Включение этих языков в учебный курс обусловлено несколькими обстоятельствами. Во-первых, требованиями упомянутой выше программы, в соответствии с которой студенты должны обладать необходимыми знаниями и умениями в области языков функционального и логического программирования, к которым относят Лисп и Пролог. Во-вторых, особенностями учебного плана по направлению “Компьютерные науки”, которым не предусмотрено отдельное изучение указанных языков. В-третьих, широким использованием этих языков при создании прототипов интеллектуальных систем. В-четвертых, стремлением авторов наполнить пособие примерами реализации соответствующих принципов ИИ, и сделать его удобным для чтения широким кругом читателей. Во второй части рассматриваются различные направления ИИ (главы 7 - 10). При этом в зависимости от специализации студентов, слушающих курс, основные акценты могут быть сделаны на различных направлениях ИИ. Авторы рекомендуют изучать первую часть курса в рамках программы подготовки бакалавров, а вторую – в рамках программ подготовки специалистов или магистров. При чтении первой части рекомендуется следующий порядок изучения глав и параграфов: 1, 2, 5, 3, 4.1, 6.1 – 6.10, 4.2 – 4.3, 6.11. Это позволяет параллельно с лекционным курсом проводить занятия лабораторного практикума на языках Лисп и Пролог.

Учебное пособие основано на материалах лекций, прочитанных авторами для студентов старших курсов Севастопольского национального технического университета и Швейцарского федерального технологического университета (ETH, г. Цюрих). Книга предназначена для студентов, обучающихся по направлению “Компьютерные науки”, специальность “Информационные управляющие системы и технологии”. Она будет полезна студентам, обучающимся по направлениям “Компьютерная инженерия” и “Компьютеризированные системы автоматики и управления”, а также разработчикам интеллектуальных систем.

В заключение авторы благодарят профессора Г. Сцекели и администрацию ЕТН за помощь в издании книги, а также рецензентов – д-ра техн. наук В. В. Мохора, д-ра физ.-мат. наук А.М. Суворова и д-ра техн. наук, профессора С.Ф. Теленика за доброжелательную критику и полезные замечания, которые способствовали улучшению книги. Особую благодарность авторы выражают д-ру техн. наук, профессору В.К. Маригодову за его большой труд по редактированию рукописи, а также канд-ту техн. наук Карлусову В.Ю. за внимательное первое чтение рукописи и многочисленные замечания. Авторы признательны доктору Дж. Кризу, любезно предоставившему материалы, использованные в 6 и 7 главах книги.

Введение, главы 1,3,4,6,7,8,9 написаны В.Н. Бондаревым, глава 10 написана Ф.Г. Аде, главы 2 и 5 написаны авторами совместно.

Отзывы и предложения просьба направлять по адресу: Украина, 99053, г. Севастополь, Студгородок, Севастопольский национальный технический университет, НМЦ.

СПИСОК СОКРАЩЕНИЙ

АМ – акустическая модель
АРР – автоматическое распознавание речи
АРТ – теория адаптивного резонанса
АСУП – автоматизированная система управления производством
БД – база данных
БЗ – база знаний
ЕЯ – естественный язык
ЗКР – список закрытых вершин
ИИ – искусственный интеллект
ИНС – искусственная нейронная сеть
ИР – интеллектуальный робот
ИС – интеллектуальная система
ЛНЭ – линейный непрерывный элемент
ЛПК – линейное предиктивное кодирование
ЛПР – лицо, принимающее решение
ЛПЭ – линейный пороговый элемент
МР – магнитный резонанс
НОУ – наиболее общий унификатор
НЭ – нейроподобный элемент
ОТК – список открытых вершин
ПЗС – приборы с зарядовой связью
ПО – программное обеспечение
ППФ – правильно построенная формула
ПрО – предметная область
ПСИИ – производственная система с искусственным интеллектом
ПЭ – процессорный элемент
РСП – расширенная сеть переходов
СИИ – система искусственного интеллекта
СММ – скрытая Марковская модель
СПЗИ – система поддержания значений истинности
СУБД – система управления базой данных
ТДШ – теория Демпстера-Шефера
УК – правило универсальной конкретизации
ФС – формальная система
ЦФ – цифровой фильтр
ЭК – правило экзистенциональной конкретизации
ЭС – экспертная система
ЯПЗ – язык представления знаний

ГЛАВА 1
ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ

Искусственный интеллект (ИИ) – быстро развивающееся направление современной информатики. Основной целью исследований, выполняемых в ИИ, является разработка компьютерных систем, способных выполнять функции, которые свойственны интеллекту человека. Попытки создания таких систем начали предприниматься с конца 40-х годов предыдущего столетия, когда были разработаны первые ЭВМ. Поэтому ИИ – относительно молодое научное направление, которое лежит на стыке разнообразных дисциплин, таких, как дискретная математика, лингвистика, психология, программирование, математическая статистика и др.

Диапазон исследований, выполняемых в рамках ИИ, весьма широкий. Он включает решение как общих проблем, например, представление знаний и логический вывод, так и специальных – игра в шахматы, доказательство теорем, сочинение стихов и т.п.

В главе рассматриваются наиболее распространенные определения понятия “искусственный интеллект”, описываются этапы развития ИИ, анализируются основные направления исследований.

1.1. Понятие “искусственный интеллект”

Несмотря на все попытки дать точное определение понятию “искусственный интеллект”, строгого определения не существует. Поэтому будет полезным, по крайней мере, обозначить границы этого понятия. С этой целью рассмотрим наиболее распространенные определения. Так, И. Рич полагает, что *ИИ – это область исследований, направленная на создание компьютеров, которые выполняют такие функции, которые в настоящий момент человек выполняет лучше* [84]. Речь идет о функциях, связанных с умением воспринимать, анализировать, рассуждать, использовать знания, планировать свои действия и т.д., в которых естественно проявление интеллекта. Характерной особенностью этого определения является указание на то, что эти функции, в настоящий момент времени, хорошо выполняют-

ся человеком и плохо компьютером. Это означает, что проблематика ИИ меняется со временем. В этом смысле, близким к рассмотренному определению является определение, предложенное Дж. Аленом: "*ИИ – это наука о создании машин, решающих задачи, которые могут решать люди…*"[57]. Можно отметить, что данное определение охватывает многие исследования, выполняемые в области компьютерных наук. Например, сложение чисел – задача, которую могут решать люди. Тем не менее, мы сейчас вряд ли назовем эту задачу интеллектуальной. Однако если суммирующее устройство было бы предъявлено человечеству в средние века, то оно, безусловно, было бы отнесено к разряду интеллектуальных. Это говорит о том, что наши представления о "думающей машине" эволюционируют во времени. Поэтому в фокусе искусственного интеллекта оказываются те задачи, которые успешно решаются человеком и плохо – компьютером. Сегодня к проблематике ИИ относятся задачи, связанные со зрительным восприятием, пониманием изображений, пониманием естественного языка (текста или речи), умением рассуждать, выявлением закономерностей, планированием действий и др.

Введенные выше определения ИИ являются сравнительными. Они сопоставляют возможности человека и машин. В 1950 году был предложен эмпирический *тест А. Тьюринга* для определения уровня интеллектуальности машин. В соответствии с тестом эксперт с помощью телетайпа мог вступать в диалог либо с компьютером, либо с человеком. Тьюринг предложил считать поведение компьютера интеллектуальным, если в диалоге участвовал компьютер, а эксперт был не в состоянии определить, с кем он ведет диалог. Тест Тьюринга не рассматривает механизмы формирования ответов на вопросы эксперта, и, следовательно, позволяет исключить все вопросы об "истинной" природе интеллекта. Несмотря на ряд недостатков, тест Тьюринга создал основу для оценки многих современных систем ИИ. Например, качество функционирования экспертных систем определяется по сравнению с решениями, предлагаемыми человеком. Такая оценка является простой разновидностью теста Тьюринга. Часто тест Тьюринга критикуют за *антропоморфизм* (уподобление человеку, наделение человеческими свойствами предметов неживой природы), так как он предписывает машине имитировать поведение человека. В то же время машинный интеллект отличается от интеллекта человека, и, вероятно, попытка уподобления его естественному интеллекту ошибочна. Например, нежелательно, чтобы машины выполняли вычисления подобно человеку, т.е. слишком медленно и неточно. Поэтому многие исследователи полагают, что тест Тьюринга отвлекает силы от решения основной задачи ИИ – разработки общей теории машинного интеллекта, и использования этой теории для разработки интеллектуальных систем, решающих практические задачи. Тем не менее,

отметим важность теста Тьюринга для оценивания качества современных программ ИИ.

Под воздействием указанной критики были предложены иные определения ИИ, например:

- *ИИ – ветвь информатики, которая связана с автоматизацией интеллектуального поведения* [77];
- *ИИ – это наука о вычислениях, которые делают возможными восприятие, логический вывод и действие* [95];
- *ИИ – это информационная технология, связанная с процессами логического вывода, обучения и восприятия* [91].

Синтезируя подобные определения из различных источников, введем следующее рабочее определение.

Искусственный интеллект – это одно из направлений информатики, целью которого является разработка компьютерных систем, способных выполнять функции, традиционно считающиеся интеллектуальными, - понимание языка, логический вывод, использование накопленных знаний, обучение, планирование действий и т.д.

Работы в области ИИ в значительной степени имеют эмпирическую и инженерную ориентацию. Создаваемые системы искусственного интеллекта (СИИ) совершенствуются в экспериментальных условиях. В результате разрабатываются некоторые универсальные процедуры, применимые к широкому классу задач. Именно такие процедуры, составляющие основу большинства СИИ, и рассматриваются в следующих главах книги.

ИИ прошел в своем развитии несколько этапов. Каждый из этапов можно охарактеризовать господствующей научной идеей (парадигмой). Чтобы глубже понять предмет ИИ полезно рассмотреть смену парадигм ИИ на разных этапах его развития. Если на предшествующих этапах развития ИИ создавались сосредоточенные, централизованные интеллектуальные системы, то нынешний этап развития ИИ характеризуется переходом к коллективистским представлениям о природе интеллекта. В этой связи основными тенденциями развития ИИ на нынешнем этапе является интеграция и децентрализация.

Рассмотрим подробнее этапы развития ИИ.

1.2. Этапы развития искусственного интеллекта

Исследования в области ИИ начались сразу же после появления первых электронных компьютеров. За 50 лет ИИ прошел в своем развитии несколько этапов. Опишем их в хронологическом порядке.

Становление искусственного интеллекта (1943 - 1956 гг.). Работы в области искусственного интеллекта начались с зарождения нейрокибернетики. Связано это было с тем, что основу мозга образуют нервные клетки – нейроны. Поэтому исследователи в ранний период развития ИИ пытались строить разумные машины, имитируя поведение коллектива нейронов. В 1943 г. У. Маккаллох и У. Питтс предложили модель формального логического нейрона, который мог находиться в двух устойчивых состояниях. Д. Хебб (1949 г.) разработал простое правило, позволяющее изменять веса связей между нейронами с целью их обучения. В 1951 году М. Минский и Д. Эдмондс разработали нейрокомпьютер, который содержал 40 нейронов. Эти первые работы в области ИИ обозначили два основных подхода к разработке СИИ – логический и коннекционистский.

Термин “искусственный интеллект” был предложен на семинаре с аналогичным названием, который проходил в Дартсмутском колледже (США) летом 1956 года. Семинар был организован М. Минским и Дж. Маккарти. Среди участников семинара были Г. Саймон и А. Ньюэлл, которые разработали программу “Логик-теоретик”, предназначенную для доказательства теорем в исчислении высказываний. Именно этих четырех исследователей считают “отцами” ИИ. М. Минский и Дж. Маккарти основали лабораторию ИИ в Массачусетском технологическом институте, а Г. Саймон и А. Ньюэлл в университете Карнеги-Меллона. Позже Дж. Маккарти перешел в Стэнфордский университет. В этих трех университетах, а также в Эдинбургском университете выполнялись первые работы, обеспечившие становление ИИ.

Эвристический поиск и доказательство теорем (1956 – 1969 гг.). Работы Г. Саймона и А. Ньюэлла по созданию программы “Логик-теоретик” были завершены в 1957 г. С помощью этой программы был успешно доказан ряд теорем из книги Уайтхеда и Рассела “Principia mathematica”. Один из недостатков программы состоял в том, что она неэффективно осуществляла поиск решений. А. Ньюэлл и Г. Саймон заинтересовались общими методами поиска решений, приступив к созданию следующей программы, которую назвали “Универсальный решатель задач”. Эту программу обычно называют GPS по первым буквам английского названия General Problem Solver. GPS разрабатывалась с целью имитации процесса решения задач человеком и базировалась на идеях эвристического поиска. *Эвристика* – правило, полученное опытным путем, которое позволяет сократить количество переборов в пространстве поиска решений.

Программа GPS могла настраиваться на предметную область. Для этого необходимо было задавать структуру состояний задачи и операторы, преобразующие эти состояния. Решение задачи осуществлялось на основе поисковых алгоритмов в пространстве возможных решений по эвристиче-

ским правилам, которые направляли поиск к искомой цели. GPS была, вероятно, первой программой, которая использовала способы рассуждений, свойственные человеку.

В этот же период Дж. Маккарти создает язык Лисп, который становится доминирующим языком программирования систем ИИ. В 1958 году Маккарти публикует статью “Программы и здравый смысл”, в которой описывает гипотетическую программу “Advice Taker”, центральной идеей которой было представление знаний и логический вывод.

Идея Маккарти использовать логику для построения СИИ была развита в работах Дж. Робинсона. В 1965 г. Робинсон разрабатывает *метод резолюций*, который позволяет автоматизировать процесс доказательства теорем при наличии исходных аксиом. Логические представления кладутся в основу вопросно-ответной системы, разработанной К. Грином (1969 г.), и в основу робота “Шейки”, который был создан в 1969 г. в Стэнфордском исследовательском институте.

В начале 60-х годов интенсивно развивались исследования в области нейронных сетей. Правило адаптации весов связей нейронных элементов, предложенное Д. Хеббом, было усовершенствовано в работах Б. Уидроу (1962 г.), который разработал нейронную сеть под названием Adaline, а также в работах Ф. Розенблатта, создавшего персептрон (1962 г.). Персептрон Розенблатта представлял собой устройство, предназначенное для распознавания зрительных образов. В частности, с помощью персептрона можно было различать буквы алфавита. Розенблатт доказал теорему сходимости персептрона, обеспечившую теоретическую базу алгоритмам обучения персептрона. Однако возможности персептрона Розенблатта оказались весьма ограниченными. Он не мог распознавать частично закрытые буквы, а также буквы иного размера, нежели те, которые предъявлялись ему в процессе обучения. Вскоре М. Минский в соавторстве с С. Пейпертом (1969 г.) написал книгу, в которой доказал, что персептрон Розенблатта не в состоянии выполнять многие функции, предсказанные Розенблаттом. Это привело к сокращению ассигнований, выделяемых на развитие нейроинформатики, о чем позже Минский высказывал сожаление.

Таким образом, к концу 60-х годов основное внимание в области ИИ стало уделяться методам представления задач и поиску решений, в частности, представлению задач в логической форме и автоматическому доказательству теорем на основе метода резолюций.

Представление знаний (1969 – 1979 гг.). К концу 60-х годов было обнаружено, что для решения практически важных задач недостаточно одних знаний общего характера (общих стратегий поиска решений). Успешное решение прикладных задач возможно только при наличии хорошо структурированных специальных знаний.

Программа DENDRAL, разработанная в 1969 году Э. Фейгенбаумом, Б. Букхененом и Э. Ледербергом, содержала детальные сведения из области органической химии и помогала химикам определять молекулярную структуру неизвестных органических соединений по данным, получаемым с помощью масс-спектрометра. Масс-спектрометр, разделяя молекулы на фрагменты, измеряет массу и электрический заряд каждого из фрагментов. Чтобы определять множество форм молекул, которые могут состоять из таких фрагментов, в программе DENDRAL использовались эмпирические знания химиков, представленные в форме правил “если-то”. Это позволило резко сократить число предлагаемых вариантов решений.

DENDRAL была первой успешно реализованной программой, аккумулирующей знания экспертов. Такие программы получили название “*экспертные системы*”. Они содержали большой объем практических знаний, что позволяло им давать ответы на специальные вопросы в той или иной области.

Вдохновленные успехом программы DENDRAL Э. Фейгенбаум, Б. Букхенен и Э. Шортлифф разрабатывают экспертную систему MYCIN. Система MYCIN содержала около 450 правил, позволяющих диагностировать инфекционные заболевания крови. Важным отличием системы MYCIN от DENDRAL было то, что MYCIN позволяла осуществлять правдоподобные заключения на основе ненадежных знаний. С этой целью факты и правила в системе MYCIN характеризовались коэффициентами уверенности.

Авторы системы MYCIN пошли дальше, когда удалили из нее базу знаний, т.е. всю медицинскую информацию. Оставшаяся часть системы содержала только логику управления правилами и была названа EMYCIN (пустая MYCIN). EMYCIN представляла собой оболочку экспертной системы, которую можно было наполнять знаниями. Подобные системы стали применять в самых разных областях – от определения конфигурации компьютерных систем до геологической разведки.

Искусственный интеллект перестал быть чистой наукой и начал приносить практическую пользу. Большой успех имела экспертная система PROSPECTOR (1979 г.), с помощью которой были открыты залежи молибдена. В системе PROSPECTOR база знаний представлялась в виде семантической сети. Система обеспечивала взаимодействие с пользователем на естественном языке.

Семантические сети были предложены в 1967 году М. Куиллианом. В это время важность предметных знаний стала осознаваться при решении задачи понимания естественного языка. Существенный шаг в этом направлении сделал Виноград, который разработал программу SHRDLU (1968 г.). Программа содержала знания о некотором игрушечном мире, в котором

можно было перемещать с помощью естественно-языковых команд кубики и пирамиды.

Дальнейшее совершенствование систем понимания естественного языка связано с именами Р. Шенка и В. Вудса. Шенк разработал программу, преобразующую входные естественно-языковые высказывания к элементарным концептам, которые можно было представлять в памяти ЭВМ. В дальнейшем в сотрудничестве с Р. Абельсоном он разработал модель представления знаний, называемую “скриптами”, или *сценариями*. Шенк использовал скрипты в программе SAM, которая могла интерпретировать рассказы. В 1973 году В. Вудс создает систему LUNAR, которая позволяла геологам задавать вопросы на естественном языке относительно образцов пород, доставленных с Луны. LUNAR была первой используемой на практике системой с естественно-языковым интерфейсом доступа к базе данных.

Расширение приложений СИИ требовало развития моделей представления знаний. В рассматриваемый период были разработаны различные языки представления знаний. Некоторые из них базировались на логике. Так, в 1973 году А. Колмероэ создает язык логического программирования Пролог, ставший популярным в Европе. В США создается язык PLANNER, поддерживающий предикатный уровень представления знаний. Часть языков использовала теорию фреймов, предложенную М. Минским (1975 г.). *Фрейм* представляет информационную структуру, состоящую из слотов, в которых собраны характеристики и соответствующие значения, обеспечивающие описание объектов или понятий. Фреймы организуются в иерархические сети, что позволяет экономно представлять знания в памяти ЭВМ. Для работы с фреймовыми моделями представления знаний были разработаны языки FRL, KRL, GUS и др.

Коммерческий успех (1979 – 1988 гг.). Первой интеллектуальной системой, нашедшей применение в промышленности, стала экспертная система К1, разработанная Мак-Дермоттом (1982 г.). Система К1 применялась для определения конфигурации компьютерных систем семейства VAX. Коммерческая версия системы, разработанная корпорацией Digital Equipment совместно с университетом Карнеги-Меллона (США), получила название XCON. К 1986 году система XCON позволяла корпорации Digital Equipment экономить 70 млн. долларов ежегодно. Кроме этого, применение системы сократило число ошибок с 30% до 1%.

Экспертные системы начинают широко применяться и в других корпорациях и фирмах, например, American Express сократила свои затраты на 27 млн. долларов в год благодаря экспертной системе, управляющей выдачей кредитов.

В 1981 году Япония объявляет о начале проекта машин V поколения, базирующихся на принципах ИИ. Проект был рассчитан на 10 лет и предусматривал, среди многих целей, создание Пролог-машины с возможностями общения на естественном языке. Этот проект способствовал активизации исследований в области ИИ во многих странах. К разработке интеллектуальных систем подключились многие компании – Texas Instruments, Xerox, Symbolics и др. Объемы финансирования проектов в области ИИ за 1980-1988 гг. выросли с нескольких миллионов до 2 млрд. долларов в 1988 году.

Возрождение нейроинформатики (с 1986 г. по настоящее время). После доказательства М. Минским ограниченных возможностей первых нейронных сетей исследования в этом направлении выполнялись учеными других областей знаний. В 1982 г. физик Дж. Хопфилд опубликовал работу, в которой проанализировал методами статистической механики свойства однослойной нейронной сети с симметричными связями. Хопфилд обнаружил возможность применения такой сети в задаче распознавания образов. Психологи Д. Румельхарт и Г. Хинтон, изучавшие модели нейросетевой памяти, заново открыли в 1986 году алгоритм обучения многослойного персептрона, названный алгоритмом обратного распространения ошибки. Ранее этот алгоритм был открыт Брайсоном и Хо в 1969 году [84].

Указанные две работы послужили толчком к лавинообразному росту применений моделей нейронных сетей для решения практических задач. Помимо теоретических разработок, этому способствовало и появление мощных аппаратно-программных средств, позволяющих моделировать нейронные сети достаточного уровня сложности.

В настоящее время модели нейронных сетей применяются в самых различных областях.

Коллективистские модели ИИ (с 1996 г. по настоящее время). Рассмотренные выше этапы развития ИИ характеризуются разрозненными подходами, основанными на выделении частного свойства понятия “интеллект”. Например, принцип знаниецентризма обусловил развитие и господство в течение определенного периода когнитивистских моделей ИИ, в частности, логических. В соответствии с такими моделями на первый план выдвигается способность интеллектуальной системы рассуждать, а действия (поведение) рассматриваются как нечто вторичное. Основное внимание при этом уделяется логическому выводу. Выделение в определении ИИ функции обучения (адаптации) способствовало развитию коннекционистских (нейронные сети) и эволюционных моделей (генетические алгоритмы).

Трактовка ИИ с позиций способности к восприятию и коммуникации привела к развитию моделей понимания изображений и естественного языка.

Выделение и гиперболизация отдельных свойств интеллекта способствовали противопоставлению и расхождению различных школ ИИ. К началу 90-х годов исследователи начали осознавать, что подобные трактовки ИИ носят весьма ограниченный характер и не учитывают реалии развития интеллекта человека. Во-первых, естественный интеллект интегрирует в себе весь спектр свойств интеллекта. Во-вторых, интеллектуальное развитие человека происходит в обществе, в процессе выполнения им определенной деятельности. В отрыве от взаимодействия с себе подобными развитие естественного интеллекта проблематично. Поэтому, начиная с начала 90-х годов, в ИИ стали преобладать две основные тенденции – интеграция и децентрализация [41,77,84,95].

Интеграционные процессы проявились в разработке гибридных систем искусственного интеллекта, объединяющих в себе преимущества разнородных моделей, например, нечеткие экспертные системы и нейронные сети. В таких интегрированных системах могут поддерживаться различные модели представления знаний, разные типы рассуждений, модели восприятия и распознавания образов.

Процессы децентрализации связаны с рассмотрением ИИ с позиций коллективного поведения большого числа взаимодействующих между собой интеллектуальных агентов. При этом интеллект агента рассматривается как подсистема управления деятельностью в процессе взаимодействия агента с другими агентами.

В основе распределенного (децентрализованного) интеллекта лежит функционально-структурная единица – *агент*, способная [41]:

- воздействовать на других агентов и самих себя;
- образовывать свои собственные цели;
- общаться с другими агентами;
- функционировать без прямого вмешательства со стороны любых средств и осуществлять самоконтроль (автономность);
- воспринимать часть среды своего функционирования;
- строить локальное представление среды;
- выполнять обязанности и оказывать услуги;
- самовоспроизводиться.

Важно отметить, что при решении конкретной задачи агенты образуют структурированное сообщество, в котором наблюдается определенная кооперация между агентами. Например, можно говорить об агентах, выполняющих простую передачу сообщений, о координирующих агентах, которые организуют взаимодействие в сообществе (группе) агентов, о поис-

ковых агентах, которые анализируют большие объемы информации и возвращают результат поиска, об агентах, формирующих понятия, и др.

Таким образом, согласно коллективистской модели основным объектом исследований ИИ является сообщество неоднородных, взаимодействующих агентов, а основное содержание разработок связано с созданием интеллектуальных агентов, обладающих заданными свойствами, и вычислительных структур, поддерживающих взаимодействие агентов.

В дальнейшем рассматриваются классические модели и процедуры ИИ, которые могут быть положены в основу построения агентов различных видов. Вопросы взаимодействия агентов, определяющие архитектуру многоагентных систем, выходят за рамки книги. Подробнее см. [63,84,89].

1.3. Основные направления исследований

К области ИИ относятся многие задачи, которые хорошо решаются человеком и плохо вычислительной машиной. Например, понимание естественного языка, манипулирование знаниями, восприятие и распознавание образов, выявление закономерностей и обучение и др. Рассмотрим подробнее некоторые из направлений ИИ.

1.3.1. Представление задач и поиск решений

Интеллектуальная деятельность человека связана с поиском решений задач. При этом задача считается интеллектуальной, если точный (алгоритмический) метод ее решения не известен. В рамках направления рассматриваются различные способы представления задач, подходящие для решения их поисковыми методами на компьютере. Выделяют несколько способов представления задач:

- в пространстве состояний;
- в пространстве подзадач;
- в виде доказательства теорем.

Если задача представляется в пространстве состояний, то задают начальное и конечное состояние задачи, множество операторов, обеспечивающих перевод задачи из одного состояния в другое. Решение задачи сводится к поиску последовательности операторов, осуществляющих перевод задачи из начального состояния в конечное. Такой поиск осуществляется различными методами. Наиболее интересными являются методы эвристического поиска, в частности, A*- алгоритм, применяемый при решении многих задач: восстановление контуров изображений, распознавание речи, игровые задачи, планирование поведения роботов. Подробнее вопросы представления задач и поиска решений рассмотрены в главе 2.

1.3.2. Доказательство теорем

Доказательство теорем является чрезвычайно важной областью ИИ. Рождение этого направления началось с создания программы “Логик-теоретик”, которая доказала ряд теорем математической логики. Однако ранние подходы к автоматизации доказательства теорем были не эффективны. Связано это было с экспоненциальным ростом числа логических выражений в процессе доказательства. Для ограничения количества выражений предлагались различные методы, но наиболее эффективным оказался метод резолюций, предложенный Робинсоном.

К доказательству теорем могут быть сведены многие практические задачи. В случае представления задачи в форме доказательства теорем возможные состояния задачи, включая начальные и целевые, рассматриваются как правильно построенные формулы исчисления предикатов. Операторы, отображающие одно состояние в другое, рассматриваются как правила, которые выводят одно правильно построенное выражение из другого. Процесс поиска решения заключается в переписывании формул с помощью правил, пока не будет получены целевые утверждения. Основные процедуры логического вывода рассматриваются в главе 4.

1.3.3. Представление знаний

Это один из центральных вопросов ИИ. В системах искусственного интеллекта знания не могут быть представлены непосредственно в том виде, в котором они используются человеком. Необходим поиск таких форм отображения знаний, которые были бы подходящими для хранения в памяти компьютера. Эти формы знаний, которые называют *моделями представления знаний*, должны обладать достаточной выразительностью, удобством использования, эффективностью манипулирования. Выразительность означает способность модели одинаково хорошо представлять как обобщенные, так и конкретные знания, как из универсальных, так и из специальных предметных областей. Она также предполагает способность модели отражать неполные знания. Удобство использования означает, что модель представления знаний должна базироваться на понятиях, которые используют эксперты и пользователи. Это упрощает приобретение знаний и оценку их корректности. Эффективность манипулирования связана с возможностью быстрого извлечения необходимых порций знаний и выполнения на их основе соответствующих выводов. Поэтому под знаниями в СИИ понимают формализованные сведения, данные, на которые ссылаются в процессе логического вывода [37].

Знания, представляемые в СИИ в виде самостоятельных информационных компонент, хранятся в базе знаний.

Имеются четыре основные модели представления знаний:

- логическая модель;
- продукционная модель (модель, основанная на правилах);
- модель семантической сети;
- фреймовая модель.

Языки, используемые для разработки систем ИИ на основе указанных моделей, называют языками представления знаний (ЯПЗ). Наибольшее распространение получила продукционная модель представления знаний. При использовании продукционной модели база знаний состоит из набора правил, а программа, управляющая перебором правил, называется *машиной вывода*.

В настоящее время основные усилия исследователей, работающих в области представления знаний, направлены на разработку языков представления знаний, поддерживающих модели рассуждений, характерные для псевдофизических логик (временная логика, пространственная логика, каузальная логика), а также нечеткие и немонотонные рассуждения (см. главы 3 и 4)

1.3.4. Экспертные системы

Уже при разработке первых программ, осуществляющих поиск решений в пространстве состояний, исследователи осознали важность проблемно-ориентированных знаний. Так, врач успешно выполняет свои функции, поскольку он обладает большим запасом медицинских знаний. Геолог открывает новые месторождения тех или иных природных ресурсов, руководствуясь теоретическими и экспериментальными знаниями из области геологии. Экспертные знания представляют собой комбинацию теоретико-аналитических методов и эвристических правил решения задач, которые показали свою эффективность на практике. Программные системы, которые аккумулируют знания эксперта в определенной области и способны вырабатывать решения и рекомендации на уровне эксперта, называют *экспертными системами* (ЭС). ЭС – наиболее широко применяемые на практике интеллектуальные системы. ЭС создаются в процессе непрерывного взаимодействия эксперта в той или иной области со специалистами в области информационных технологий. В ходе этого процесса осуществляется извлечение знаний эксперта и перенос их в базу знаний системы. Специалисты, осуществляющие структурирование базы знаний и ее заполнение, называются инженерами по знаниям или *когнитологами*.

Перечень типовых задач, решаемых ЭС, включает: интерпретацию, диагностику, прогнозирование, мониторинг и др.

Узким местом в разработке ЭС является получение знаний. Направление ИИ, занимающееся проблемами получения и структурирования знания специалистов с целью последующей разработки баз знаний, называется *инженерией знаний*. Подробнее проблемы разработки ЭС рассмотрены в главе 7.

1.3.5. Обучение и выявление закономерностей

В процессе испытаний экспертных систем выяснилось, что их возможности ограничены объемом информации, хранимой в базе знаний. Возникла задача пополнения и приобретения знаний в процессе функционирования системы. Данная задача связана с построением обучающихся систем, которые могут самосовершенствоваться.

Проблема создания обучающихся систем интересовала исследователей с момента зарождения ИИ. Так, уже персептрон Ф. Розенблата использовал простейшие правила обучения. А. Сэмюэль (1962 г.) создал программу игры в шашки, которая могла накапливать опыт игры и действовать, исходя из него. Идеи Сэмюэля были восприняты другими исследователями. В 1973 г. П. Уинстон увлекся проблемой обучения на примерах и составил программу “Арки”, которая могла на примерах обучаться простейшим понятиям. Крупным шагом вперед в машинном обучении стала программа AQ11, разработанная Р. Михальским (1976 г.). Программа AQ11 могла по данным выводить правила “если-то” для определения заболеваний растений.

Существенный вклад в решение проблемы обучения внес Д. Ленат, создав программу “Автоматический математик” (AM). Эта программа была снабжена определенными знаниями из области чисел и набором эвристических правил, позволяющих устанавливать закономерности. Программа AM повторно открыла около 200 важных понятий теории чисел. Позже Д. Ленат создал обучающуюся программу EURISKO, которая открыла новый способ построения логического элемента для сверхбольших интегральных схем – трехмерный элемент типа И/ИЛИ.

Другими важными исследованиями, выполненными в этой области, стали работы Т. Митчелла и Дж. Квинлана.

Характерной особенностью обучающихся систем является использование индуктивных умозаключений. *Индукция* – способ рассуждения от частного к общему. Индукция противоположна *дедукции* – суждению от общего к частному. Применение индуктивных схем рассуждений позволяет

программам обучаться на примерах путем их обобщения. Данные вопросы обсуждаются в главе 4.

1.3.6. Общение на естественном языке

Одним из основных направлений исследований в области ИИ является разработка программно-аппаратных средств, обеспечивающих общение с компьютером на естественном языке. Решение данной задачи предусматривает как разработку подсистем понимания естественно-языковых (ЕЯ) высказываний, так и генерацию текстов на естественном языке. К обработке естественного языка относится и задача машинного перевода.

Сложность проблемы ЕЯ–общения можно проиллюстрировать на задаче машинного перевода. Первые программы машинного перевода, базирующиеся на использовании словарей и грамматик, оказались неэффективными. Обусловлено это было, прежде всего, многозначностью естественного языка – одни и те же слова в предложении имеют разный смысл в зависимости от контекста и знаний о предмете общения. Такие знания выходят за рамки знаний морфологии и синтаксиса языка. Для правильной интерпретации ЕЯ–высказываний компьютер должен обладать теми же знаниями, что и пользователь. Это требует создания больших баз знаний, что, само по себе, является сложной задачей. Проблема здесь состоит не столько в том, как разместить эти знания в памяти компьютера, а в том, как их получить. Поэтому в настоящее время общение пользователя с компьютером выполняется на проблемно-ориентированных подмножествах естественного языка.

Традиционно в системах, допускающих общение на естественном языке, выполняется несколько фаз анализа входных сообщений:

- морфологический анализ – поиск основ слов с отнесением их к определенным грамматическим категориям, обнаружение и исправление орфографических ошибок;
- синтаксический анализ – грамматический разбор предложений и определение их лингвистической структуры;
- семантический анализ – формирование смысла предложений на внутреннем языке системы;
- прагматический анализ – выявление взаимосвязи между предложениями текста.

В последнее время в задачах обработки ЕЯ все чаще применяют эмпирический подход, базирующийся на использовании методов статистики и машинного обучения. В этом случае подсистема понимания высказываний обучается на больших объемах реальных языковых данных. Это позво-

ляет автоматизировать процесс формирования необходимых баз знаний (см. главу 9).

1.3.7. Распознавание образов

В рамках этого направления ИИ ведутся исследования по восприятию зрительной, слуховой и других видов информации, методам ее обработки с целью выделения существенных признаков воспринимаемых объектов и осуществления их классификации. Примерами задач распознавания образов могут служить: автоматическое чтение рукописных текстов, распознавание речи, анализ изображений, диагностика болезней, дистанционная идентификация объектов и др. Если множество признаков, разделяющих объекты на классы заранее не известно, то возникает задача обучения распознаванию образов или самообучения распознаванию образов. Поэтому данное направление ИИ близко к машинному обучению и неразрывно связано с нейросетями.

Решение задачи распознавания образов сводится к построению решающей функции, которая по значениям наблюдаемых признаков позволяет отнести объект к тому или иному классу. При построении решающей функции часто используют статистический подход. Однако его применение требует задания многомерных плотностей распределения вероятностей, определение которых может составлять самостоятельную задачу. Часто применяют иной подход, основанный на обучении распознаванию образов. Предложено много алгоритмов обучения распознаванию образов. Одним из наиболее простых является алгоритм обучения персептрона, который представляет собой простейшую нейронную сеть. Возможности простого персептрона по распознаванию образов весьма ограничены. Поэтому при решении многих практических задач применяют более сложные модели. Детально вопросы распознавания образов и обучения нейронных сетей рассматриваются в главе 8.

1.3.8. Компьютерное зрение

Более 85% информации человек получает с помощью зрения. Для современного компьютера этот канал ввода информации также играет важную роль. Но ввести изображение в память ЭВМ нетрудно. Для машины намного сложнее понять изображение. Поэтому главная задача, которую приходится решать при построении систем машинного зрения, – это не создание устройств ввода изображений, а разработка методов и алгоритмов анализа визуальной информации. Проблема создания систем компьютер-

ного зрения тесно связана с такими направлениями ИИ, как представление знаний, обучение, распознавание образов (см. главу 10).

1.3.9. Языки программирования систем искусственного интеллекта

В рамках этого направления разрабатываются специальные языки для решения задач ИИ. Важной особенностью этих языков является использование более высоких уровней абстракции данных, а также ориентация на обработку символьной информации. Наиболее популярными языками, применяемыми при создании исследовательских прототипов СИИ, являются языки Лисп и Пролог.

Кроме этого, создаются специальные инструментальные средства, ориентированные на промышленную разработку СИИ, например, G2, ART, KEE, Knowledge Craft и др. [12,63]. В книге подробно рассматриваются языки Лисп и Пролог (см. главы 5 и 6).

Выше перечислены лишь те направления ИИ, которые нашли отражение в книге. Вместе с тем исследования в области ИИ охватывают многие другие направления, например:

- синтез изображений по моделям;
- интеллектуальные роботы;
- генетические алгоритмы;
- многоагентные системы;
- когнитивное моделирование;
- игры и машинное творчество, и многое другое.

1.4. Структура систем с искусственным интеллектом

Рассмотрим в качестве примера СИИ обобщенную структурную схему производственной системы с искусственным интеллектом (ПСИИ) и структурную схему интеллектуального робота (ИР). ПСИИ представляет собой органичное звено в структуре автоматизированных систем управления производством (АСУП). Основной характеристикой ИР является целенаправленное поведение в изменяющейся среде. ИР является звеном гибких автоматизированных производств.

Рассмотрение обобщенных схем указанных интеллектуальных систем (ИС) позволит точнее выяснить место и роль различных универсальных процедур и методов ИИ, обсуждаемых в последующих главах книги.

Структурная схема ПСИИ изображена на рисунке 1.1 [2]. Такая ПСИИ может решать несколько характерных задач: управление, прогнозирование, мониторинг, диагностика и др. Основным элементом ПСИИ явля-

ется база знаний (БЗ), в которой хранятся знания о предметной области, представленные в компьютерно интерпретируемой форме.

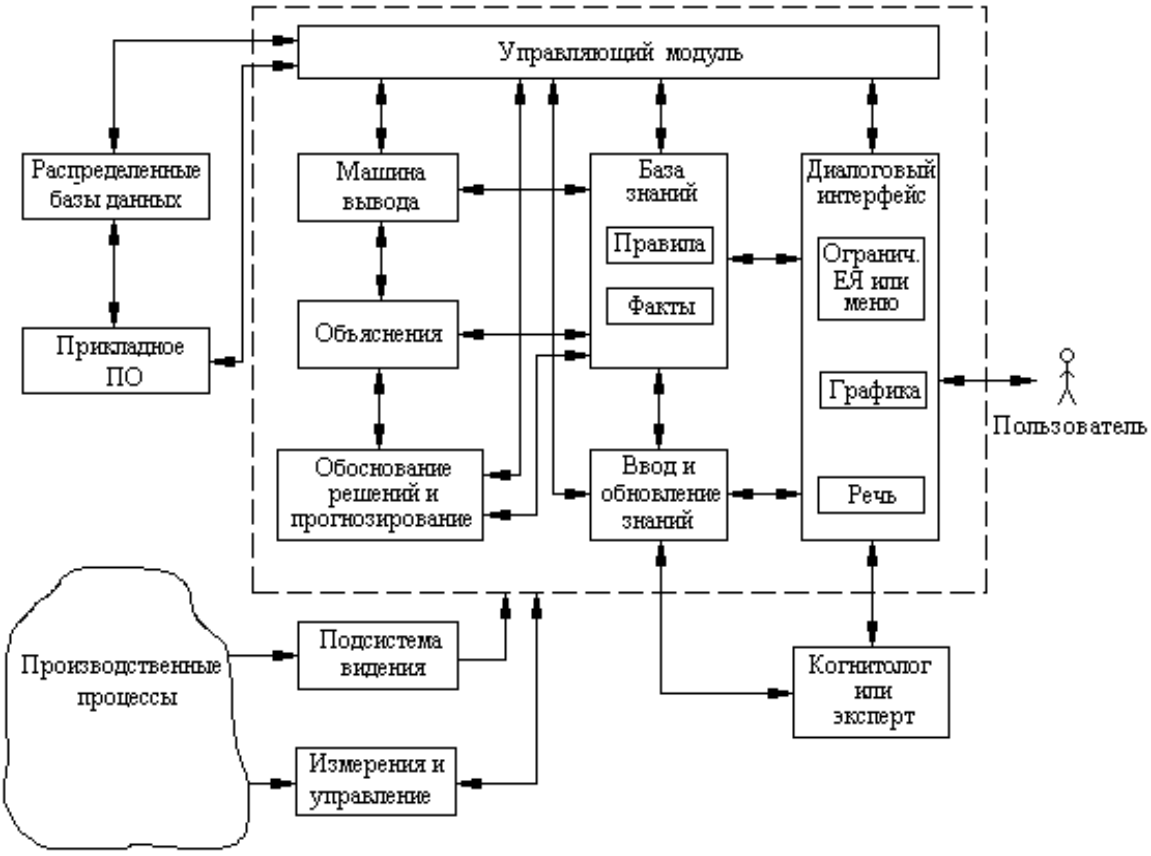


Рисунок 1.1 – Производственная система с искусственным интеллектом

Знания в БЗ представляются фактами и правилами. Факты – данные, характеризующие конкретные объекты, явления, ситуации предметной области. Применяя правила к фактам, можно получать новые факты, которых не было в системе. Для того чтобы из известных фактов получать новые факты, правила необходимо применять в соответствии с некоторыми принципами управления, которые реализуются машиной вывода.

Важной чертой рассматриваемой ПСИИ, основанной на знаниях, является наличие подсистемы объяснения. Подсистема объяснения позволяет объяснить пользователю, как система пришла к тому или иному выводы, и почему требуются те или иные сведения для продолжения вывода. С одной стороны, это повышает степень доверия пользователя к результатам вывода, а с другой – позволяет когнитологу (инженеру по знаниям) или эксперту оценивать правильность функционирования системы.

Диалоговый интерфейс предусматривает общение с системой на ограниченном естественном языке, графическое отображение информации, а

также ввод-вывод речевых сообщений. В качестве пользователя системы может выступать либо оператор, либо производственный процесс. В последнем случае в состав системы дополнительно включают подсистемы видения, измерения и управления. Когнитолог или эксперт осуществляют ввод знаний и их обновление с помощью специальных редакторов. При этом ввод знаний может осуществляться как с помощью указанных средств, так и посредством диалогового интерфейса.

Взаимодействие ПСИИ с распределенными базами данных позволяет получать информацию из других информационных систем, находящихся на различных уровнях иерархии управления производством. В тех случаях, когда для решения задачи требуется выполнять специфические вычисления, основанные на стандартных операциях обработки данных, осуществляется доступ к прикладному программному обеспечению (ПО).

Подсистема обоснования и прогнозирования решений позволяет моделировать последствия выполнения решений, рекомендованных системой, прогнозировать критические ситуации. Используя результаты прогноза, можно корректировать процесс логического вывода.

Большое значение при создании ПСИИ имеют знания. Знания вводятся в систему когнитологом, который получает их в процессе системной работы с экспертом.

Рассмотренная форма представления знаний – в виде набора правил и фактов – одна из возможных моделей представления знаний. В процессе общения с экспертом когнитолог выбирает наиболее адекватную для данного приложения модель представления знаний и соответствующий механизм логического вывода.

Структурная схема рассмотренной ПСИИ является обобщенной. В схемах реальных ПСИИ могут отсутствовать некоторые из рассмотренных блоков. Наличие тех или иных связей в конкретной системе в значительной степени определяется её назначением.

Рассмотренная ПСИИ относится к группе организационно-управляющих систем, используемых при оперативном управлении, контроле качества, планировании и т.д. В этой группе систем выделяют три типа ИС [2]: вопросно-ответные системы, позволяющие взаимодействовать с базами данных на ограниченном ЕЯ; расчетно-логические системы, позволяющие пользователю, не являющемуся специалистом в области вычислительной техники, автоматически выбрать метод решения задачи и формировать программу ее решения в диалоге; экспертные системы – системы-консультанты, аккумулирующие знания и опыт экспертов в той или иной области и позволяющие менее квалифицированным пользователям получать консультации.

Рассмотренная ПСИИ представляет собой гибридную СИИ, вобравшую в себя характерные признаки всех трех типов указанных выше систем.

Интеллектуальные роботы (ИР) относятся к группе робототехнических систем. ИР представляет собой систему, обладающую способностью накапливать и корректировать знания на основе активного восприятия информации о внешнем мире, а также осуществлять целенаправленное поведение на основе накопленных знаний [35]. Обобщенная структурная схема интеллектуального робота изображена на рисунке 1.2 [2,29,35]. В ней указаны три основные подсистемы: восприятия внешней информации, представления знаний, планирования и исполнения действий. Рассмотрим каждую из этих подсистем.

Подсистема представления знаний обеспечивает накопление и корректировку знаний робота о внешней среде (мире). Форма представления знаний выбирается с учетом конкретного класса задач, решаемых роботом.

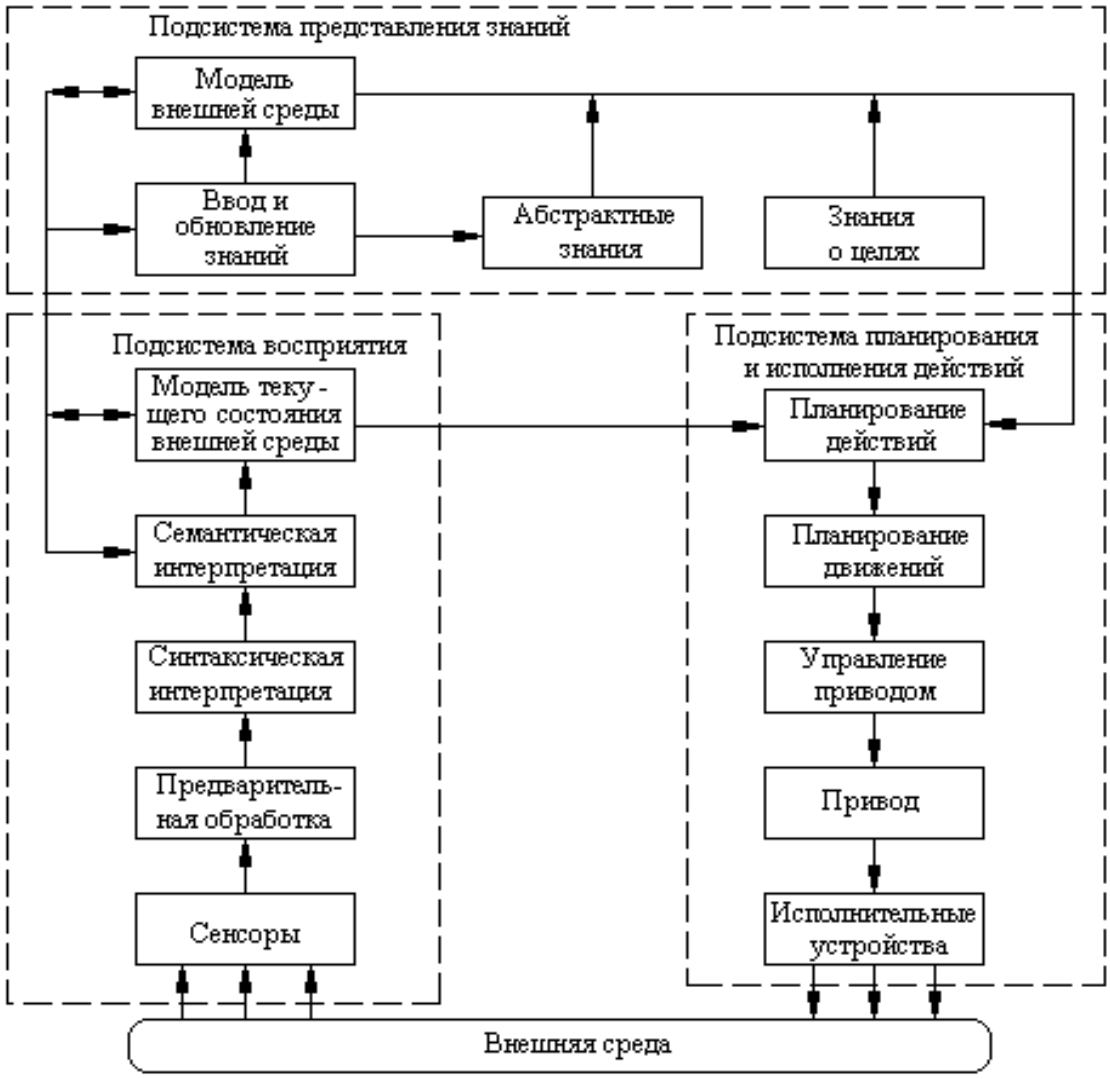


Рисунок 1.2 – Структурная схема робота с искусственным интеллектом

В системе знаний робота выделяют абстрактные знания, знания о целях, модель внешней среды. Абстрактные знания – это знания об общих законах, действующих как во внешней среде, так и во внутренней среде робота. Абстрактные знания, как правило, не меняются во времени. Знания о целях – это локальные и глобальные цели функционирования ИР. Модель внешней среды – формализованное описание внешней среды, в которой функционирует робот. Модель может пополняться и корректироваться в процессе функционирования робота с помощью блока ввода и обновления знаний.

Подсистема восприятия внешней информации осуществляет связь ИР с внешней средой. Основная задача этой подсистемы – построение модели текущего состояния внешней среды. Формирование ситуационной модели среды происходит в результате предварительной обработки информации, воспринимаемой сенсорами, ее синтаксической и семантической интерпретации. Источниками информации о внешней среде являются сенсоры различного типа: приемники визуальной и звуковой информации, тактильные, локационные и др. В ходе синтаксической интерпретации формируется описание текущей ситуации во внешней среде на некотором внутреннем языке ИР. Однако смысл воспринимаемых при этом сведений остается нераскрытым. На этапе семантической интерпретации устанавливается смысл поступившей информации, что позволяет построить модель текущего состояния внешнего мира. Отметим, что в ходе построения такой модели используется модель внешней среды, хранящаяся в подсистеме представления знаний ИР.

Подсистема планирования и исполнения действий обеспечивает реализацию программы воздействия на внешнюю среду для достижения целей функционирования робота. Такая программа формируется с учетом сложившейся ситуации и согласно глобальным целям. Планирование действий робота обычно представляется как процесс решения задач. При этом задача и ее решение понимаются в широком смысле. Цель решения задачи состоит в приведении текущего состояния внешней среды к желаемому состоянию. Решение задачи сводится к поиску последовательности действий, обеспечивающих преобразования указанных состояний. Воздействие на внешнюю среду выполняется посредством устройства управления приводом и исполнительными устройствами.

Рассмотренные примеры интегрируют многие функции, выполняемые СИИ различного типа. В следующих главах подробно рассматриваются принципы организации, возможности и характеристики основных функциональных частей СИИ.

Вопросы для самопроверки

1. Проанализируйте существующие определения понятия “искусственный интеллект”. Составьте собственное определение.
2. Объясните причину критики теста Тьюринга.
3. Назовите основные этапы развития ИИ и укажите основные результаты, полученные на каждом из этапов.
4. Перечислите основные направления ИИ, дайте их краткую характеристику.
5. Приведите обобщенную структурную схему системы ИИ и объясните функции всех блоков, входящих в её состав.
6. Дайте определение понятию “интеллектуальный робот” и приведите его обобщенную структурную схему.
7. Может ли компьютер в настоящее время решать следующие задачи:
   а) делать открытия и доказывать теоремы;
   б) сочинять забавные рассказы;
   в) давать компетентные советы в узких областях знаний;
   г) осуществлять перевод устной русской речи на английский язык;
   д) управлять движением автомобиля в центре крупного города?

ГЛАВА 2
СПОСОБЫ ПРЕДСТАВЛЕНИЯ ЗАДАЧ И ПОИСК РЕШЕНИЙ

Термин “решение задач” (problem solving) употребляется в искусственном интеллекте в весьма ограниченном смысле. Речь идет о хорошо определенных задачах, решаемых на основе поисковых алгоритмов.

Задача считается хорошо определенной, если для неё имеется возможность задать пространство возможных решений (состояний), а также способ просмотра этого пространства с целью поиска конечного (целевого) состояния, соответствующего решенной задаче. Поиск конечного состояния задачи заключается в применении к каждому состоянию алгоритмической процедуры с целью проверки, не является ли это состояние решением задачи. Данная поисковая процедура продолжается до тех пор, пока не будет найдено решение. Примерами хорошо определенных задач являются: доказательство теорем, поиск маршрута на карте, планирование действий робота в среде с препятствиями, различные игры с полной информацией и др.

Человек обычно не решает задачу в той форме, в которой она изначально формулируется. Он стремится представить задачу таким образом, чтобы ему было удобно ее решать. Для этого он выполняет преобразование исходного представления задачи с целью сокращения пространства, в котором необходимо выполнять поиск решения задачи. Этап выбора подходящей формы представления задачи настолько обыден, что мы часто не осознаем его важность. В то же время форма или способ представления задачи в значительной мере определяет успех ее решения. При выборе способа представления задачи обычно учитывают два обстоятельства: представление задачи должно достаточно точно моделировать реальность; способ представления должен быть таким, чтобы решателю задач было удобно с ним работать.

Поскольку под решателем задач в искусственном интеллекте понимается компьютер, то в настоящей главе рассматриваются способы представления задач, удобные для их решения на ЭВМ. К ним относятся следующие наиболее часто используемые способы:

- представление задач в пространстве состояний;
- представление, сводящее задачу к подзадачам;
- представление задач в виде теорем.

Данные представления и соответствующие универсальные методы поиска решений разрабатывались преимущественно на начальных этапах развития искусственного интеллекта. Позже было замечено, что для решения многих практических задач одних универсальных стратегий недостаточно. Необходим также большой объем знаний и наличие практического опыта. Исследования в области ИИ сосредоточились на представлении и приобретении знаний. Однако это не снизило значимости разработанных стратегий поиска решений, так как они представляют некоторые общие схемы управления механизмом вывода систем, основанных на знаниях. Рассматриваемые ниже способы представления задач и методы поиска их решений играют важную роль во многих системах ИИ, включая экспертные системы, понимание естественного языка, доказательство теорем и обучение.

2.1. Общая характеристика способов представления задач

2.1.1. Представление задач в пространстве состояний

Подход, использующий пространство состояний, весьма распространен в решении задач. Он включает в себя:

- задание начальных состояний задачи;
- задание конечных (целевых) состояний задачи;
- задание операторов, преобразующих одни состояния задачи в другие.

Решение задачи состоит в том, чтобы найти последовательность операторов, преобразующих начальное состояние задачи в конечное, которое соответствует решенной задаче.

Существуют многочисленные задачи, решение которых интерпретируется как поиск в пространстве состояний. Рассмотрим примеры некоторых задач.

Поиск пути в сети улиц. Задача состоит в том, чтобы на схеме улиц найти путь из начального пункта в конечный пункт. Задание только топологии сети улиц не полностью описывает задачу. Необходимо указать, в чем должно состоять решение задачи, и какие ограничения накладываются на допустимые переходы между состояниями задачи.

Для данной задачи состояние определяется положением перемещающегося объекта на схеме улиц. Непосредственный переход между

двумя состояниями допустим, если между соответствующими позициями на схеме имеется прямая связь (т.е. улица). Начальное и конечное (целевое) состояния представляются определенными позициями объекта. Задача считается решенной, если проложен путь из начального пункта в конечный пункт за конечное число элементарных переходов.

Менее тривиальным является поиск пути минимальной стоимости между начальным и целевым пунктом. Затратами, учитываемыми при поиске такого пути, могут быть: расстояние, время, потребление бензина и т.п. В этом случае не достаточно обозначать состояние задачи только позицией перемещающегося объекта на схеме, так как в данную позицию можно попасть разными маршрутами. Необходимо связывать с каждым состоянием текущие затраты и уже пройденные пункты маршрута (например, посредством списковых структур). Тогда начальное состояние можно представить в виде списка, содержащего только исходный пункт, а конечное состояние, получаемое в ходе поиска решения, – в виде упорядоченного списка всех пунктов, расположенных на оптимальном пути. Для целенаправленного движения в направлении конечного пункта желательно перемещаться каждый раз в тот пункт, который более всего сокращает расстояние (затраты) до цели.

Игра-головоломка “8”. Игра-головоломка “8” – это одна из показательных задач поиска решений в пространстве состояний, используемая многими авторами [30,77]. В игре используется восемь фишек, пронумерованных от 1 до 8 (рисунок 2.1). Фишки располагаются в девяти ячейках, образующих матрицу размером 3×3. Одна из ячеек всегда пустая. Любая фишка, смежная с пустой ячейкой, может быть передвинута в позицию, соответствующую пустой ячейке.

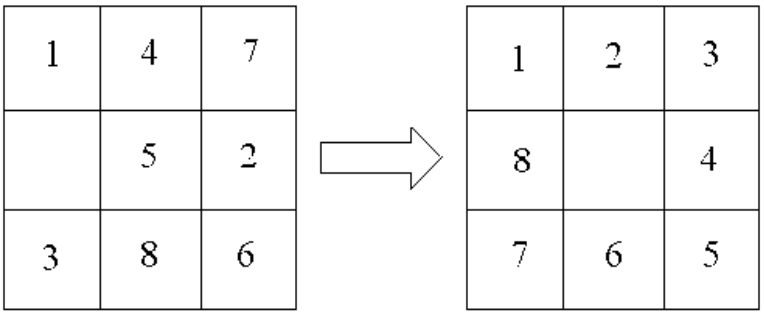


Рисунок 2.1 – Игра-головоломка “8”

В головоломке ищут минимальную последовательность ходов, переводящих игру из начального состояния в конечное состояние, определяемое заранее заданной конфигурацией фишек. Для целенаправленного движения к целевому состоянию может учитываться количество неправильно расположенных фишек для каждой возможной позиции свободной ячейки.

Задача о N ферзях. На шахматной доске необходимо расставить N ферзей таким образом, чтобы они не угрожали друг другу. Расстановка ферзей начинается с пустой доски (начальное состояние) и выполняется последовательно линия за линией. Конечное состояние достигается, когда расставлены все ферзи. Смежные состояния задачи определяются условием отсутствия угрозы определенному ферзю. Для целенаправленного движения к целевому состоянию необходимо ставить очередного ферзя на соответствующую линию так, чтобы оставалось как можно больше не атакованных полей.

Выделение контуров на изображении. Важным этапом анализа изображений является обнаружение контуров. На первом шаге для каждого пикселя изображения вычисляют функцию градиента. Затем отдельные пиксели изображения объединяют в линии, представляющие контуры или ребра. Это выполняется с помощью методов поиска решений в пространстве состояний. В качестве исходного пикселя выбирается пиксель с достаточно высоким значением градиента. Поиск продолжается в направлении перпендикулярном по отношению к направлению вектора градиента. При этом предпочтение отдается пикселям с большими значениями градиента, лежащими вдоль выбранного направления поиска, т.е. ожидается, что контуры относительно гладкие. Цель считается достигнутой, если выполняется одно из следующих условий: значение градиента стало меньше некоторого порогового значения; достигнута граница изображения; контур попадает на уже ранее найденный контур.

В общем случае задача, поставленная как задача поиска в пространстве состояний, определяется совокупностью четырех составляющих

$$(S_0, S, F, G),$$

где S – множество состояний;

S_0 – множество начальных состояний, $S_0 \in S$;

F – множество операторов, преобразующих одни состояния в другие;

G – множество целевых состояний, $G \in S$.

Каждый оператор $f \in F$ является функцией, отображающей одно состояние в другое – $s_j = f(s_i)$, где $s_i, s_j \in S$. Решением задачи является последовательность операторов $f_i \in F$, преобразующих начальные состояния в конечные, т.е. $f_n(f_{n-1}(\ldots(f_2(f_1(S_0)))\ldots)) \in G$. Если такая последовательность не одна и задан критерий оптимальности, то возможен поиск оптимальной последовательности.

Удобно поиск решений в пространстве состояний представлять в виде *графа состояний*. Множество вершин графа соответствует состояниям задачи, а множество дуг (ребер) – операторам. В этом случае поиск решения задачи может интерпретироваться как поиск пути на графе.

Уточним сказанное на примере поиска пути в лабиринте (рисунок 2.2,а) [86]. Если обозначить меткой X путешественника, то начальное и конечное состояния задачи можно задать с помощью схем, изображенных на рисунке 2.2,б.

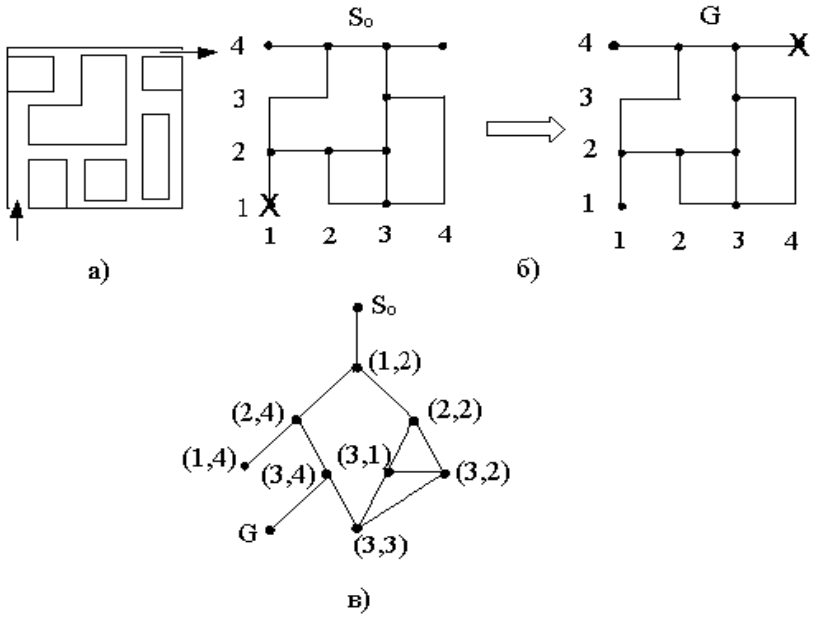


Рисунок 2.2 – Поиск пути в лабиринте

Хотя путешественник может находиться в любой точке лабиринта, интерес представляют только пересечения различных путей, обозначенные точками. Попадая на пересечения путей, путешественник должен принять решение о дальнейшем направлении движения. Поэтому возможные состояния задачи определяются нахождением путешественника в указанных точках. Перемещения путешественника из одной точки в другую соответствует переходам задачи из одного состояния в другое. Следовательно, множество операторов задачи представляется только одним оператором перемещения, а решение задачи сводится к нахождению последовательности операторов перемещений, преобразующих задачу из начального состояния S_0 в конечное состояние G.

Пространство состояний рассматриваемой задачи можно представить в виде графа, изображенного на рисунке 2.2,в. Вершины графа, обозначенные координатами точек пересечения, соответствуют состояниям задачи, а дуги – операторам перемещений. Так как при решении данной задачи используется только один оператор, то дуги не имеют меток. Решение задачи состоит в нахождении пути из начальной вершины S_0 в целевую вершину G.

В рассматриваемой задаче к целевой вершине ведут пять различных путей. Если количество элементарных отрезков между точками пересечений использовать в качестве оценки стоимости пути, то суммарные стоимости каждого из пяти путей будут равны: 6, 6, 8, 10, 10. В качестве опти-

мального пути можно выбрать один из двух путей с минимальной стоимостью.

Иногда рассматривают поиск с ограничениями. *Ограничения* представляют некоторые условия, которые должны выполняться в ходе достижения целевого состояния. Например, при поиске пути в лабиринте может быть поставлено условие, чтобы длина пройденного пути не превышала некоторого значения L, тогда состояние задачи будет определяться позицией путешественника в лабиринте и пройденным расстоянием.

Граф состояний задачи может быть задан неявно. В этом случае задается множество начальных состояний S_0 и множество операторов F, которые, будучи применимы к вершине графа, дают все ее дочерние вершины. Процесс применения операторов к некоторой вершине с целью получения всех ее дочерних вершин называется *раскрытием вершины*. При этом могут задаваться *условия применимости* оператора к вершине (состоянию). Например, оператор перемещения в задаче поиска пути в лабиринте может быть выполнен в том случае, если существует путь между соответствующими вершинами. Если рассматривается несколько операторов, то каждый из них может иметь собственные условия применимости.

2.1.2. Сведение задач к подзадачам

Иной способ представления задач основан на сведении их к подзадачам. Основная идея такого представления заключается в разбиении задачи на подзадачи, которые, в свою очередь, тоже разбиваются на подзадачи и так до тех пор, пока исходная задача не будет сведена к элементарным подзадачам, имеющим решения, или пока не будет доказано, что задача не имеет решения. Процесс разбиения (декомпозиции) задачи на подзадачи называется *редукцией*. Поэтому представление задачи в виде множества подзадач – это представление в системе редукций, а поиск решения – это поиск в пространстве описаний задач. Разбиение на подзадачи дает преимущество в том случае, когда подзадачи взаимно независимы, и, следовательно, решать их можно независимо друг от друга.

Пространство задач представляется в виде направленного графа, который называется графом редукции задачи. С каждой вершиной этого графа связывается определенная подзадача, а дуги соответствуют операторам порождения подзадач. В графе редукции задачи существуют два типа вершин: конъюнктивные и дизъюнктивные. *Конъюнктивные* вершины, или вершины типа “И”, указывают, что для решения задачи требуется решить все ее подзадачи. *Дизъюнктивные* вершины – вершины типа “ИЛИ” – соответствуют альтернативным подзадачам. Для решения задачи, представляемой такой вершиной, достаточно решить только одну из ее подзадач.

Граф редукции задачи является графом типа “И-ИЛИ”. С каждой вершиной такого графа можно связать высказывание в виде булевой функции, и поэтому его называют также *пропозициональным графом*.

На рисунке 2.3 показан частный случай графа редукции задачи – дерево редукции. Здесь *A* – исходная задача, для решения которой необходимо решить подзадачи *B*, *C* и *D*. Вершина *A* является конъюнктивной. На рисунке 2.3 это обозначено дужкой, связывающей ветви, ведущие к подзадачам *B*, *C* и *D*. Для решения задачи *B* необходимо решить одну из ее подзадач *E* или *F*. Вершина *B* является дизъюнктивной.

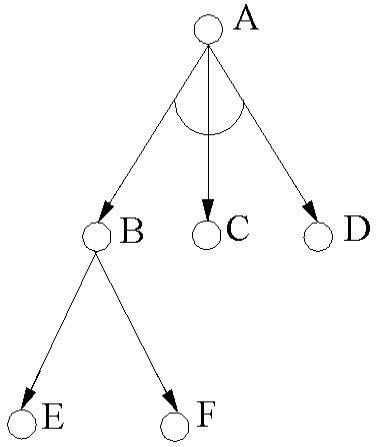


Рисунок 2.3 - Дерево редукции

Граф редукции задачи отличается от графа состояния только тем, что в нем имеются И-вершины. В частном случае, когда на графе редукции отсутствуют И-вершины, И-ИЛИ граф трансформируется в граф пространства состояний. Тогда решение может быть найдено с помощью методов поиска решений в пространстве состояний.

Начальная вершина (корень И-ИЛИ дерева) представляет исходную задачу, которую следует решить. Вершины, соответствующие задачам, решение которых известно, называются *заключительными*. Задачи и соответствующие вершины, не имеющие решений, называются *тупиковыми*. Заключительные вершины являются разрешимыми, а тупиковые – неразрешимыми. Разрешимыми являются также И-вершины, у которых разрешимы все дочерние вершины, и ИЛИ-вершины, у которых разрешима хотя бы одна дочерняя вершина. Цель поиска на И-ИЛИ графе – показать, что начальная вершина разрешима. Для этого выполняется построение специального графа, который называется графом решения. *Граф решения* – это подграф графа редукции задачи с начальной вершиной, соответствующей исходной задаче, содержащий разрешимые вершины. Просмотр графа решения в направлении от заключительных вершин к начальной вершине определяет множество подзадач, которые необходимо решить, и порядок их решения. Рассмотрим примеры представления задач с помощью И-ИЛИ графов.

Решение неопределенных интегралов. Пусть требуется найти неопределенный интеграл

$$\int \sin^2 x \cos^4 x dx .$$

Граф решения данной задачи показан на рисунке 2.4.

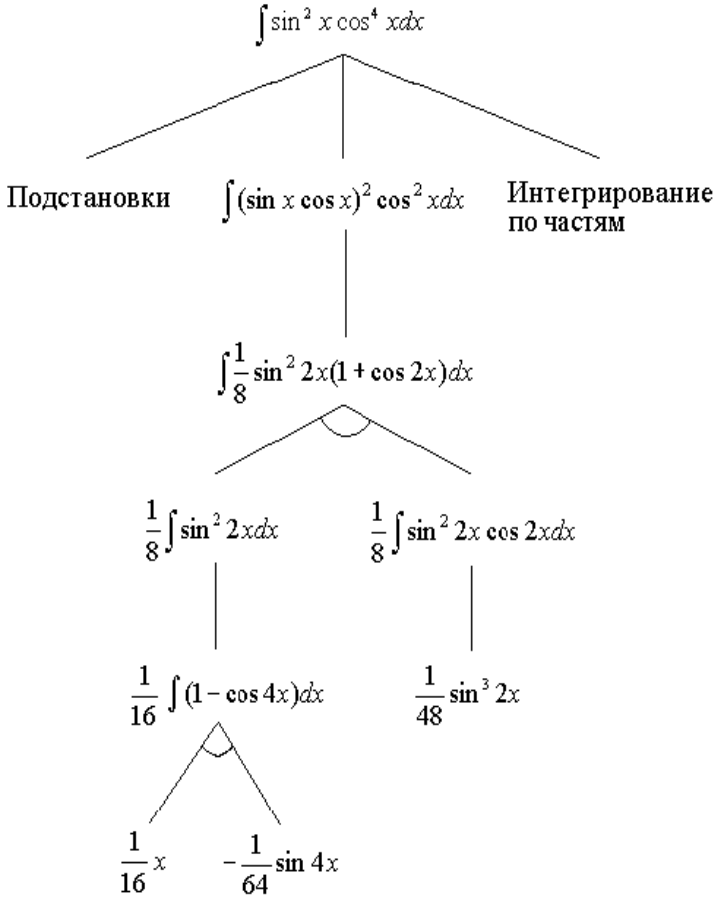


Рисунок 2.4 – Вычисление неопределенного интеграла

При нахождении неопределенных интегралов часто используют следующие операторы: всевозможные подстановки (замены), интегрирование по частям, алгебраические преобразования, тригонометрические преобразования и т.п. Исходная задача может быть сведена к множеству подзадач, указанных на рисунке 2.4, с помощью упомянутых выше операторов. Декомпозиция исходной задачи выполняется до тех пор, пока не будут найдены элементарные подзадачи, решение которых известно. Когда такие задачи получены, они помечаются как решенные. Затем выполняется специальная процедура, которая вводит полученные решения во все возможные подзадачи верхних уровней, отмечая, если нужно, каждую подзадачу более высокого уровня как решенную. Если будет отмечена исходная задача, то она считается решенной.

Формирование предложений с помощью формальных грамматик. Интересно отметить, что конечный И-ИЛИ граф может представлять порождающую *контекстно-свободную грамматику.* В этом случае И-вершины соответствуют правилам-продукциям, ИЛИ-вершины – вспомогательным символам, заключительные вершины – терминальным символам (см. §8.6).

Рассмотрим процесс генерации синтаксически правильных предложений английского языка как пример задачи, которая сводится к решению подзадач. Задача состоит в том, чтобы формировать предложения в соответствии с грамматикой, заданной совокупностью правил-продукций:

1) S → SUB PRED;
2) SUB → PRON;
3) SUB → NP;
4) PRED → V NP;
5) NP → DET N;
6) PRON → he;
7) PRON → it;
8) DET → a;
9) DET → the;
10) V → saw;
11) V → carried;
12) N → dog;
13) N → man.

Здесь строчными буквами обозначены терминальные элементы, а прописными буквами – нетерминальные. Символ S обозначает начальный символ. Начиная с S, нужно построить последовательность из терминальных символов, представляющую синтаксически правильные предложения. Применяя к S правило 1, сведем задачу формирования предложения S к задаче формирования символов SUB и PRED. Теперь задача состоит в том, чтобы преобразовать каждый из символов SUB и PRED в последовательность терминальных символов. Эти подзадачи проще исходной. Они также могут быть сведены к более простым подзадачам путем применения правил 2, 3 и 4. Продолжая указанный процесс, получим дерево редукции задачи, изображенное на рисунке 2.5 .

Штриховыми линиями на рисунке показано возможное дерево решения, в соответствии с которым формируется последовательность из терминальных элементов, образующая английское предложение “he carried a dog” (он нес собаку). С помощью рассматриваемой грамматики можно сформировать 48 правильных предложений: “the man saw a dog”, “the dog carried the man” и т.п. Безусловно, данные предложения корректны только в отношении синтаксиса. Семантика предложения здесь не рассматривается. Таким образом, дерево решения, представленное на рисунке 2.5, соответствует дереву вывода некоторого высказывания в контекстно-свободной грамматике. Поиск деревьев решений может выполняться с привлечением методов теории формальных грамматик.

В дереве, изображенном на рисунке 2.5, имеются повторяющиеся вершины, например NP. Такие вершины соответствуют одним и тем же подзадачам. Если совместить две вершины NP, то дерево редукции транс-

формируется в граф редукции. Методы поиска решений в пространстве редукции задачи должны исключать повторное рассмотрение уже решенных подзадач.

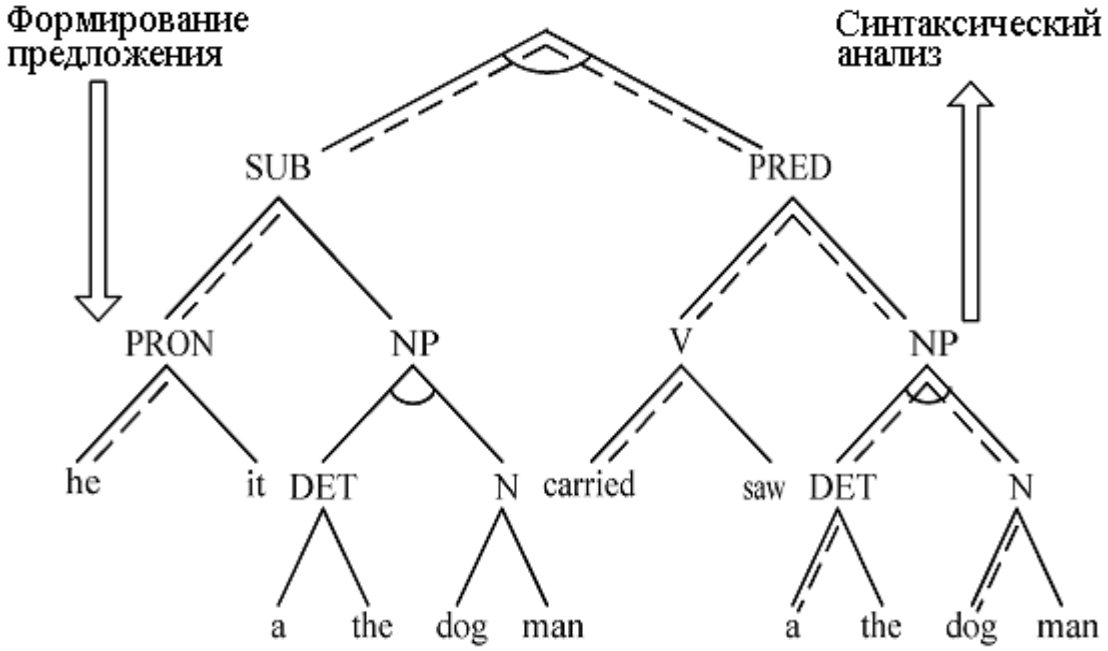


Рисунок 2.5 – Граф редукции задачи формирования предложений

Игры для двух игроков-противников. Имеется большой класс игр с нулевой суммой и полной информацией для двух лиц, например: шахматы, шашки, игра “го” и др. В этих играх игрок обладает полной информацией о текущей игровой ситуации. Причем, если один игрок что-либо выигрывает, то другой столько же проигрывает. Возможны два исхода игры: выигрыш и проигрыш.

Представим игру в виде И-ИЛИ графа. Так как ходы игроков выполняются по очереди, то на графе выделяют два вида позиций – МАКС и МИН (рисунок 2.6). На рисунке позиции, соответствующие игроку МАКС обозначены квадратом, а игроку МИН – кругом. Игрок, в пользу которого решается игра, обычно называется МАКС. Пусть в начальной позиции S игрок МАКС может сделать любой ход. Каждый ход игрока МАКС приводит к одной из возможных позиций $Q_1,Q_2,\ldots,Q_N$ игрока МИН, а ответный ход МИН (противника) в позиции Q_1 приводит к новым возможным позициям игрока МАКС (т.е. $R_{11},R_{12},\ldots,R_{1k}$).

Для того чтобы игрок МАКС выиграл в позиции S, необходимо найти такой ход из позиции S, который создает выигрышную позицию Q_i. Таким образом, МАКС выигрывает в позиции S, если он выигрывает или в позиции Q_1, или Q_2, или Q_3 и т.д. Следовательно, S – это ИЛИ вершина. Так как МАКС выигрывает в позиции $Q_{i,}$, то он должен выигрывать после

любого ответного хода противника, т.е. и в позиции R_{i1}, и в R_{i2}, и в R_{i3} и т.д. Таким образом, все позиции, в которых выбор должен сделать игрок МИН, – это И-вершины.

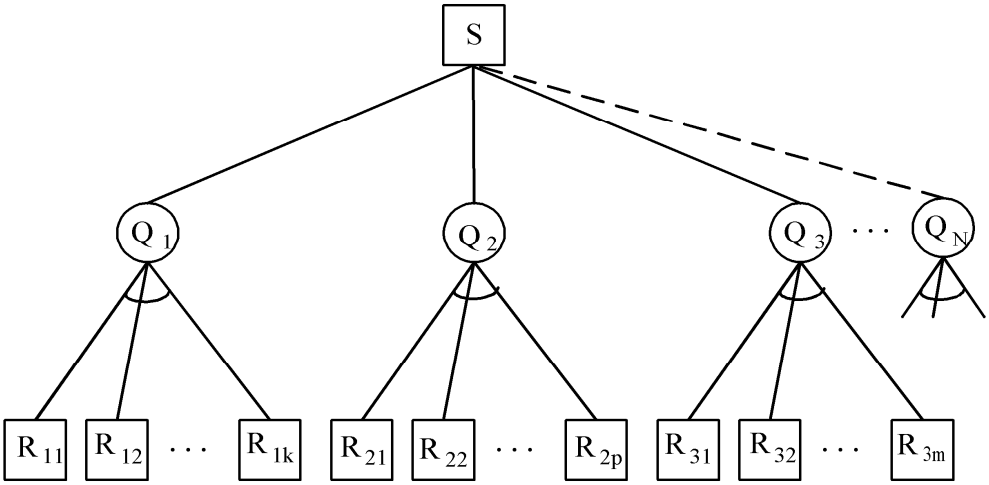


Рисунок 2.6 – И-ИЛИ дерево игры для двух игроков

Чтобы решить задачу, необходимо построить дерево игры, обеспечивающее победу игрока МАКС независимо от ответных ходов игрока МИН. Поскольку поиск дерева игры имеет свои особенности, ниже посвятим этому отдельный параграф.

2.1.3. Представление задач в виде теорем

При представлении задач в форме доказательства теорем возможные состояния задачи, включая начальные и целевые состояния, рассматриваются как правильно построенные формулы исчисления предикатов, которое будет рассмотрено позже. Операторы, отображающие одно состояние в другое, рассматриваются как правила, которые выводят одно правильно построенное выражение из другого. Процесс решения задачи в этом случае протекает следующим образом:

а) формируется множество исходных истинных утверждений (аксиом) относительно условия задачи, записываемых в виде формул языка исчисления предикатов;

б) строится гипотеза относительно результата решения задачи, которая то же записывается на языке исчисления предикатов и называется целевым утверждением;

в) аксиомы комбинируются между собой на основе допустимых правил вывода с целью получения множества новых утверждений;

г) проверяется, не содержит ли множество новых утверждений целевое утверждение или его отрицание; если содержит, то теорема считается доказанной, и процесс решения задачи на этом заканчивается;

если целевое утверждение или его отрицание не входят в множество вновь сформированных утверждений, то полученное множество утверждений объединяется с множеством исходных аксиом и процесс повторяется.

Рассмотренный процесс поиска решений представляет собой логическую формализацию хода решения задачи в пространстве состояний методами полного перебора. Такой метод гарантирует нахождение решения, если оно существует. Однако непосредственное его применение ограничено, так как полный перебор может потребовать больших ресурсов времени и памяти.

Для управления поиском используют различные стратегии выбора пар аксиом или промежуточных утверждений, участвующих в выводе, что позволяет внести систематичность в процесс поиска и сократить число возможных комбинаций различных утверждений в процессе вывода.

Основные процедуры логического вывода рассматриваются в главе 4 после изложения основ исчисления предикатов.

2.2. Поиск решений задач в пространстве состояний

Методы поиска в пространстве состояний подразделяются на две группы: методы “слепого” и упорядоченного (эвристического) поиска. Достоинством методов “слепого” поиска является простота алгоритмической реализации и гарантированность получения решения, если оно существует. Методы “слепого” поиска называют также “неинформированными” методами, так как в процессе поиска пути на графе не учитывается информация о степени близости текущей и целевой вершин. Это приводит к тому, что в методах “слепого” поиска выполняется полный просмотр всего пространства состояний. Для нетривиальных задач такой просмотр невозможен из-за слишком большого размера пространства состояний. В этом случае возникает проблема *комбинаторного взрыва*, когда размер пространства решений возрастает чрезвычайно быстро с ростом числа рассматриваемых альтернатив. Эффективным средством борьбы с комбинаторной сложностью являются методы эвристического поиска, позволяющие существенно снизить объем поиска в больших пространствах решений.

В рассматриваемых ниже алгоритмах поиска используются два списка: список открытых вершин и список закрытых вершин. В *списке открытых вершин* (OPEN) находятся идентификаторы (имена) вершин, подлежащих раскрытию; *в списке закрытых вершин* (CLOSED) находятся имена вершин, раскрытых к данному моменту, и имя вершины, раскрываемой на текущем шаге работы алгоритма.

2.2.1. Методы “слепого” поиска

2.2.1.1. Случайный поиск. Случайный поиск реализуется простым алгоритмом. Информированность при случайном поиске ограничивается топологией графа пространства состояний. Стратегия поиска состоит в том, что, начиная с начального состояния, случайно выбирают оператор, осуществляющий переход к следующему состоянию задачи. Поиск продолжается до тех пор, пока не будет выполнен переход в целевое состояние.

Ниже приведена процедура случайного поиска на псевдоязыке:

```
Procedure Random_Search;
Begin
    n:=‘начальная вершина’;
    While n <>‘целевая вершина’ Do
      Begin
        Выбрать случайно оператор, применимый к ‘n’;
        Применить данный оператор к ‘n’ и получить новую вершину
        ‘n_New’;
        n:=n_New;
 End.
```

Случайный выбор оператора не гарантирует сходимость алгоритма. Тем не менее, такой метод поиска обеспечивает решение простых задач, с ограниченным пространством поиска, при наличии пути между начальной и целевой вершиной.

Так как граф состояний обследуется случайным образом, без запоминания пройденного маршрута, возможен многократный выбор одних и тех же состояний. Улучшить процедуру поиска можно введением определенной систематичности выбора переходов состояния.

2.2.1.2. Поиск “в глубину и ширину”. Данные методы исключают бесполезный повторный выбор одних и тех же вершин. Для этого с помощью списков CLOSED и OPEN ведется соответственно учет уже раскрытых вершин и вершин, подлежащих раскрытию. Кроме этого, задается определенный порядок (систематичность) обследования графа состояния, что позволяет последовательно пройти все маршруты по одному разу, без пропусков и повторений. Рассмотрим процедуру поиска в глубину:

```
Procedure Depth_First;
Begin
  Поместить начальную вершину в список OPEN;
  CLOSED:=‘пустой список’;
  While OPEN<>‘пустой список’ Do
    Begin
```

```
        n:=first(OPEN);
        If n='целевая вершина' Then Выход(УСПЕХ);
        Переместить n из списка OPEN в CLOSED;
        Раскрыть вершину n и поместить все ее дочерние вершины, отсут-
        ствующие в списках OPEN и CLOSED, в начало списка OPEN, свя-
        зав с каждой дочерней вершиной указатель на n;
      End;
      Выход(НЕУДАЧА);
End.
```

В начале поиска список CLOSED пустой, а OPEN содержит только начальную вершину. Каждый раз из списка OPEN выбирается для раскрытия первая вершина (вызов функции first(OPEN)). Раскрытая вершина перемещается в список CLOSED, а ее дочерние вершины помещаются в начало списка OPEN. Таким образом, на следующем шаге будут раскрываться дочерние вершины текущей вершины n. Принцип формирования списка OPEN в этом случае соответствует стеку, т.е. вершина, добавленная в список последней, обрабатывается первой (LIFO – Last In First Out). Если проследить направление поиска по дереву состояний, то фронт поиска растет в глубину. Для построения обратного пути (из целевой вершины в начальную вершину) все дочерние вершины снабжаются указателями на соответствующие родительские вершины.

Процедура поиска в ширину отличается от рассмотренной процедуры поиска в глубину тем, что дочерние вершины, получаемые при раскрытии вершины *n*, помещаются в конец списка OPEN. Следовательно, при поиске в ширину принцип формирования списка OPEN соответствует очереди, т.е. вершина, внесенная в список первой, обрабатывается первой (FIFO – First In First Out). Благодаря этому фронт поиска растет в ширину, что гарантирует нахождение пути с минимальным количеством дуг (ребер) между исходной и целевой вершинами (своего рода оптимальность).

2.2.1.3. Алгоритм равных цен. Информированность о решаемой задаче здесь выше, чем в случае поиска в глубину и ширину. Каждому оператору, преобразующему состояние n_i в состояние n_j, ставится в соответствие некоторая функция стоимости $c(n_i,n_j)$, характеризуемая положительными значениями. В ходе поиска пути на графе учитываются суммарные затраты для достижения той или иной вершины, т.е. стоимость пути к этой вершине. Стоимость пути к вершине n_j может быть определена по следующей формуле

$$g(n_j)=g(n_i)+c(n_i, n_j),$$

где $g(n_i)$ – стоимость пути из начальной вершины в вершину n_i .

В ходе поиска вершины в списке OPEN сортируются в порядке увеличения стоимости. Каждый раз раскрывается вершина с наименьшей стои-

мостью. Это позволяет найти путь минимальной стоимости из начальной вершины в целевую вершину. Следует отметить, что в процессе поиска стоимость пути к вершине может меняться, если обнаруживается более дешевый путь. Поэтому будем обозначать стоимость оптимального пути из начальной вершины в вершину *n* через *g(n)*, а стоимость текущего пути, найденного алгоритмом, через $\hat{g}(n)$, т.е. $\hat{g}(n)$ – это оценка стоимости пути *g(n)*. В конце поиска $\hat{g}(n) = g(n)$.

Ниже представлена процедура поиска оптимального пути (минимальной стоимости), записанная на псевдоязыке:

```
Procedure Optimal_Search;
Begin
    Поместить начальную вершину в список OPEN;
    CLOSED:='пустой список';
    While OPEN<>'пустой список' Do
        n:=first(OPEN);
        If n='целевая вершина' Then Выход(УСПЕХ);
        Переместить вершину n из списка OPEN в CLOSED;
        Раскрыть вершину n, для каждой дочерней вершины i вычислить
        стоимость ĝ(n, n_i);
        Поместить дочерние вершины, которых нет в списках OPEN и
        CLOSED, в список OPEN, связав с каждой вершиной указатель на
        вершину n и положить ĝ(n_i) = ĝ(n, n_i);
        Для каждой из дочерних вершин, которые уже содержатся в списке
        OPEN, сравнить текущую стоимость ĝ(n, n_i) с ранее вычислен-
        ным значением стоимости ĝ(n_i), хранящемся в списке OPEN, если
        ĝ(n, n_i) < ĝ(n_i), то установить ĝ(n_i) = ĝ(n, n_i). Снабдить указанные
        дочерние вершины указателями на вершину n;
        Упорядочить список OPEN по возрастанию стоимости;
    End;
    Выход(НЕУДАЧА);
End.
```

В тот момент, когда раскрывается некоторая вершина *n*, оптимальный путь к этой вершине уже найден, т.е. $\hat{g}(n)=g(n)$ [86]. Иными словами, если вершина уже находится в начале списка OPEN, то для нее невозможно найти более дешевого пути. Следовательно, перевод вершины в список CLOSED является окончательным.

Рассмотренная процедура поиска гарантирует нахождение пути минимальной стоимости, если граф конечен и существует путь из начальной вершины в целевую вершину. Однако сам процесс поиска не является оптимальным, так как поиск выполняется без учета положения целевых вершин.

2.2.2. Эвристический поиск

Основная идея эвристического поиска состоит в использовании дополнительной информации для управления процессом поиска. Эта дополнительная информация формируется на основе эмпирического опыта, догадок и интуиции исследователя, т.е. *эвристик*. Использование эвристик позволяет сократить количество просматриваемых вариантов при поиске решения задачи, что ведет к более быстрому достижению цели.

В алгоритмах эвристического поиска список открытых вершин упорядочивается по возрастанию некоторой оценочной функции, формируемой на основе эвристических правил. Оценочная функция может включать две составляющие, одна из которых называется эвристической и характеризует близость текущей и целевой вершин. Чем меньше значение эвристической составляющей оценочной функции, тем ближе рассматриваемая вершина к целевой вершине. В зависимости от способа формирования оценочной функции выделяют следующие алгоритмы эвристического поиска: алгоритм "подъема на гору", алгоритм глобального учета соответствия цели, А-алгоритм.

2.2.2.1. Алгоритм "подъема на гору". Алгоритм осуществляется целенаправленный поиск в направлении наибольшего убывания *эвристической оценочной функции* $\hat{h}(n)$. Данная функция обеспечивает оценку (прогноз) стоимости кратчайшего пути от текущей вершины *n* до ближайшей целевой вершины, т.е. является мерой стоимости оставшегося пути. Чем меньше значение $\hat{h}(n)$, тем более "перспективен" путь, на котором находится вершина *n*.

Подобный алгоритм используется при поиске экстремумов функции. Известно, что положение экстремума функции многих переменных определяется направлением вектора градиента функции. Поэтому поиск экстремума осуществляют в направлении наибольшего возрастания (убывания) функции, т.е. в направлении, совпадающем (или противоположном) с направлением градиента. Поиск максимума функции в этом случае напоминает восхождение на пик по наиболее крутому маршруту. Поэтому рассматриваемый алгоритм и называют алгоритмом "подъема на гору" (hill – climbing).

На очередном шаге алгоритма для каждой из дочерних вершин n_1, $n_2,\ldots,n_m$ вершины *n* вычисляется значение оценочной функции $\hat{h}(n_i)$, и для продолжения пути выбирается вершина с наименьшим значением $\hat{h}(n_i)$. Процедура поиска терпит неудачу, если для всех дочерних вершин $\hat{h}(n_i) \geq \hat{h}(n)$:

```
Procedure Hill_Climbing;
Begin
    n:='начальная вершина';
    While n<>'целевая вершина' Do
    Begin
        Раскрыть вершину n и для всех дочерних вершин n_i вычислить
        оценки ĥ(n_i);
        Выбрать дочернюю вершину n_i с минимальным значением ĥ(n_i);
        If ĥ(n_i) ≥ ĥ(n) Then Выход(НЕУДАЧА);
        n:=n_i;
    End;
    Выход(УСПЕХ);
End.
```

Существуют различные способы построения эвристической оценочной функции, однако готовых рецептов нет. При решении каждой конкретной задачи используют ранее накопленный опыт решения подобных задач. Рассмотрим пример поиска пути в лабиринте (рисунок 2.2). В этом случае оценка $\hat{h}(n_i)$ может вычисляться с помощью формулы [86]

$$\hat{h}(n_i)=|x_g - x_n| + |y_g - y_n|,$$

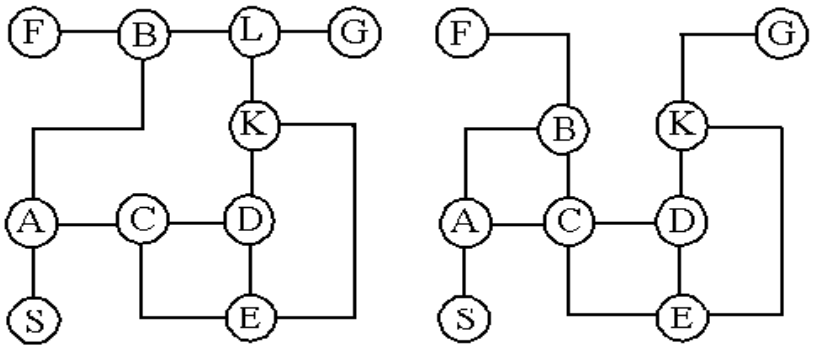
где x_g, y_g – координаты целевой вершины; x_n, y_n – координаты текущей вершины.

Данная оценка соответствует расстоянию от вершины n до целевой вершины. При поиске пути в лабиринте, изображенном на рисунке 2.7,а, последовательность раскрываемых вершин будет следующей: A, B, L, G. Поиск пути в лабиринте, показанном на рисунке 2.7,б, завершится неудачей. При этом последовательность раскрытых вершин будет следующей: A, B. Оценочная функция $\hat{h}(n_i)$ в вершине B получит значение 3 Дальнейшие шаги оказываются безуспешными, так как $\hat{h}(F)$=3 и $\hat{h}(C)$=4, а для продолжения поиска необходимо, чтобы выполнялось условие $\hat{h}(n_i) < \hat{h}(n)$.

Таким образом, хотя данный алгоритм обеспечивает ускорение поиска, он не гарантирует достижение целевой вершины. Преждевременное завершение алгоритма происходит в следующих случаях: если в процессе поиска встречаются локальные минимумы оценочной функции $\hat{h}(n)$; если для группы соседних вершин оценки $\hat{h}(n)$ равны между собой (проблема "плато").

2.2.2.2. Глобальный учет соответствия цели. Этот алгоритм поиска решения задачи похож на предыдущий. Здесь также оценивается "перспективность" того или иного пути с помощью эвристической оценочной функ-

ции $\hat{h}(n)$. Отличие состоит в том, что на каждом этапе выполняется не локальное сравнение вершин, полученных при раскрытии текущей вершины, а глобальное сравнение всех вершин-кандидатов, находящихся в списке OPEN. Для этого список открытых вершин сортируется в порядке возрастания значений $\hat{h}(n)$. Наилучшая вершина для продолжения пути будет находиться на первом месте в списке OPEN. Поэтому данный алгоритм называют также алгоритмом выбора первой наилучшей вершины (Best-First).



а) б)

Рисунок 2.7 – Графы состояний

Процедура поиска, основанная на данном алгоритме, похожа на процедуру Optimal_Search. Отличие состоит в том, что вместо функции стоимости $\hat{g}(n)$ применяется эвристическая функция $\hat{h}(n)$, т.е. учитываются только предстоящие затраты:

```
Procedure Best_First_Search;
Begin
    Поместить начальную вершину в OPEN;
    CLOSED:=‘пустой список’;
    While OPEN<>‘пустой список’ Do
    Begin
        n:=first(OPEN);
        If n=‘целевая вершина’ Then Выход(УСПЕХ);
        Переместить вершину n из списка OPEN в список CLOSED;
        Раскрыть вершину n, для каждой дочерней вершины вычислить
        ĥ(n_i);
        Поместить дочерние вершины, которых нет в списках OPEN и
        CLOSED, в список OPEN, связав с каждой дочерней вершиной указатель на вершину n;
        Упорядочить список OPEN по возрастанию значений ĥ(n)
    End;
    Выход(НЕУДАЧА);
End.
```

Если данную процедуру применить к задаче поиска пути в лабиринте, изображенном на рисунке 2.7,а, то результат будет таким же, как и в случае алгоритма подъема на гору. Если процедуру Best_First_Search применить к лабиринту, представленному на рисунке 2.7,б, то состояние списка открытых вершин будет следующим:

(S(6))→(A(5)) → (B(3) C(4)) → (F(3) C(4)) → (C(4)) → (D(3) E(4)) →
→ (K(2) E(4)) → (G(0) E(4)).

Таким образом, алгоритм позволяет успешно справляться с проблемой локальных минимумов. Однако при этом снижается эффективность самого процесса поиска. Эффективность процесса поиска будет оставаться высокой, если оценка стоимости кратчайшего пути $\hat{h}(n)$ из вершины n в целевую вершину будет как можно ближе к реальной стоимости *h(n)*[86].

2.2.2.3. А-алгоритм. Достоинство алгоритма равных цен состоит в том, что он гарантирует нахождение оптимального пути. Достоинством алгоритмов, использующих оценку предстоящих затрат $\hat{h}(n)$, является возможность усечения дерева поиска, что повышает эффективность поиска. Естественным является стремление комбинировать эти алгоритмы и учитывать как сделанные, так и предстоящие затраты. Этого можно добиться, если воспользоваться оценочной функцией

$$\hat{f}(n) = \hat{g}(n) + \hat{h}(n), \qquad (2.1)$$

где $\hat{g}(n)$ – оценка стоимости пути из начальной вершины в вершину n; $\hat{h}(n)$ – оценка стоимости кратчайшего пути из вершины n в целевую вершину.

Алгоритм, базирующийся на использовании оценочной функции $\hat{f}(n)$, называется А-алгоритмом. Оценки $\hat{h}(n)$ в ходе поиска не меняют своих значений. Оценки $\hat{g}(n)$ также не меняют своих значений и всегда равны реальным затратам *g(n)*, если поиск пути выполняется на дереве состояний. Однако при поиске пути на графе состояний оценки $\hat{g}(n)$ могут меняться. Следовательно, в общем случае оценки $\hat{f}(n)$ меняют свои значения в процессе поиска. Это приводит к тому, что вершины из списка CLOSED могут перемещаться обратно в список OPEN:

```
Procedure A_Search;
Begin
    Поместить начальную вершину s в список OPEN: f̂(s) = ĥ(s);
    CLOSED:=‘пустой список’;
```

While OPEN<>‘пустой список’ Do
Begin
n:=first(OPEN);
If n=‘целевая вершина’ Then Выход(УСПЕХ);
Переместить n из OPEN в CLOSED;
Раскрыть n и для всех ее дочерних вершин n_i вычислить оценку $\hat{f}(n, n_i) = \hat{g}(n, n_i) + \hat{h}(n_i)$;
Поместить дочерние вершины, которых нет в списках OPEN и CLOSED, в список OPEN, связав с каждой вершиной указатель на вершину n;
Для дочерних вершин n_i, которые уже содержатся в OPEN, сравнить оценки $\hat{f}(n,n_i)$ и $\hat{f}(n_i)$, если $\hat{f}(n,n_i) < \hat{f}(n_i)$, то связать с вершиной n_i новую оценку $\hat{f}(n,n_i)$ и указатель на вершину n;
Если вершина n_i содержится в списке CLOSED и $\hat{f}(n,n_i) < \hat{f}(n_i)$, то связать с вершиной n_i новую оценку $\hat{f}(n,n_i)$, переместить её в список OPEN и установить указатель на n;
Упорядочить список OPEN по возрастанию $\hat{f}(n_i)$;
End;
Выход(НЕУДАЧА);
End.

Укажем свойства А-алгоритма. Если для всех вершин $\hat{h}(n)=0$, то А-алгоритм будет соответствовать алгоритму равных цен. В этом случае гарантируется нахождения оптимального пути, но эффективность поиска будет низкой, и если пространство поиска будет бесконечно, то поиск просто не осуществим.

А-алгоритм не даёт гарантии, что найденный путь будет оптимальным. Путь может быть не оптимальным в случае, если оценка затрат на пути из вершины *n* в целевую вершину будет преувеличена (переоценена), т.е. если $\hat{h}(n) > h(n)$. Сказанное можно пояснить с помощью рисунка 2.8.

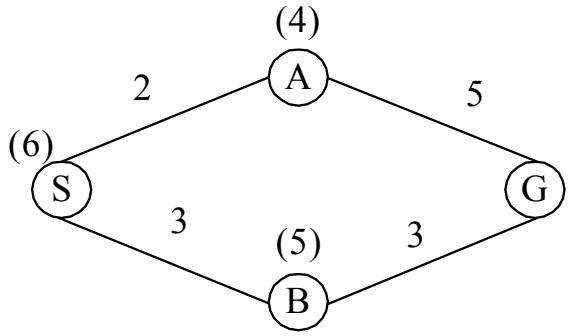


Рисунок 2.8 – Граф состояний с переоценкой затрат *h(n)*

Числа в скобках при вершинах соответствуют оценкам $\hat{h}(n)$, а рядом с ребрами – стоимости перехода $c(n_i, n_j)$. При раскрытии вершины S получаем дочерние вершины A и B. Оценки $\hat{f}(n)$ для этих вершин, вычисленные с помощью выражения (2.1.), равны: $f(A)$=2+4, $f(B)$=3+5. На следующем шаге раскрывается вершина A, и получается $f(G)$=7+0. Решением будет путь S→A→G, который для данного графа не является оптимальным.

A-алгоритм обеспечит поиск оптимального пути, если выполняется условие

$$\hat{h}(n) \le h(n). \qquad (2.2)$$

Алгоритм, учитывающий условие (2.2), называют *A*- алгоритмом или гарантирующим алгоритмом*. A*-алгоритм недооценивает затраты на пути из промежуточной вершины в целевую вершину или оценивает их правильно, но никогда не переоценивает.

Пусть A_1* и A_2* – произвольные гарантирующие алгоритмы и на всех вершинах $\hat{h}_1(n) > \hat{h}_2(n)$. Тогда информированность A_1* выше, чем A_2*. Говорят, что в этом случае A_1* имеет большую эвристическую силу, чем A_2*. Эвристически более сильный алгоритм A_1* в большей степени сокращает пространство поиска.

Эффективность поиска с помощью A*-алгоритма может снижаться из-за того, что вершина, находящаяся в списке CLOSED, может попадать обратно в список OPEN и затем повторно раскрываться. Следующий пример демонстрирует случай многократного раскрытия вершин (рисунок 2.9).

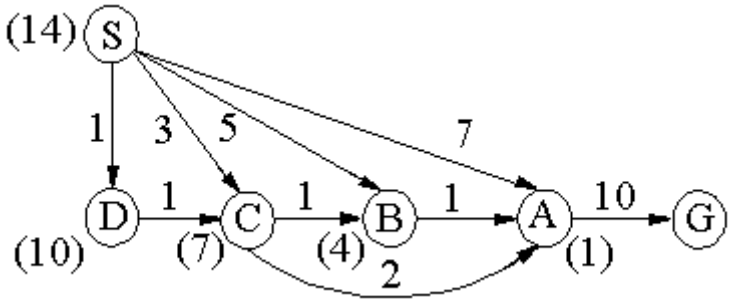


Рисунок 2.9 – Граф состояний с многократно раскрываемыми вершинами

Состояния списков OPEN и CLOSED показаны в таблице 2.1. Каждая строка таблицы соответствует очередному шагу работы алгоритма. Числа рядом с именами вершин соответствует значениям оценочной функции $\hat{f}(n)$. Из таблицы 2.1 следует, что вершина A раскрывается четыре раза, вершина B – 3 раза, вершина C – 2 раза.

Таблица 2.1—Выполнение A*-алгоритма

№	OPEN	CLOSED	Комментарий
1	S(14)	–	
2	A(8) B(9) C(10) D(11)	S(14)	
3	B(9) C(10) D(11) G(17)	A(8) S(14)	
4	A(7) C(10) D(11) G(17)	B(9) S(14)	Возврат: A
5	C(10) D(11) G(16)	A(7) B(9) S(14)	
6	A(6) B(8) D(11) G(16)	C(10) S(14)	Возврат: A, B
7	B(8) D(11) G(15)	A(6) C(10) S(14)	
8	D(11) G(15)	B(8) A(6) C(10) S(14)	
9	C(9) G(15)	D(11) B(8) A(6) S(14)	Возврат: C
10	A(5) B(7) G(15)	C(9) D(11) S(14)	Возврат: A, B
11	B(7) G(14)	A(5) C(9) D(11) S(14)	
12	G(14)	B(7) A(5) C(9) D(11) S(14)	

Чтобы A*-алгоритм не раскрывал несколько раз одну и ту же вершину, необходимо учитывать *условие монотонности*

$$\hat{h}(n_i) \le \hat{h}(n_j) + c(n_i, n_j), \qquad (2.3)$$

где n_i – родительская вершина; n_j – дочерняя вершина; $c(n_i, n_j)$ – стоимость пути между вершинами n_i и n_j.

Если условие монотонности соблюдается для всех дочерних вершин, то можно доказать, что в тот момент, когда раскрывается некоторая вершина n, оптимальный путь к ней уже найден. Следовательно, оценочная функция $\hat{f}(n)$ для данной вершины в дальнейшем не меняет своих значений, и никакие вершины из списка CLOSED в список OPEN не возвращаются.

Ранее отмечалось, что A*-алгоритм недооценивает затраты $h(n)$. Условие монотонности означает, что недооценка на пути к цели становится все меньше или, в крайнем случае, не изменяется. Действительно, если обозначить недооценку через $\hat{u}(n)$, то

$$\hat{u}(n) = h(n) - \hat{h}(n), \quad \hat{u}(n) \ge 0.$$

Из условия монотонности следует, что $\hat{u}(n_j) \le \hat{u}(n_i)$, т.е. недооценка $h(n)$ в родительских вершинах больше или равна недооценкам $h(n)$ в дочерних вершинах. На графе, изображенном на рисунке 2.9, условие монотонности не выполняется для начальной вершины. Например, $h(S) > h(D) + c(S,D)$, т.е. 14>10+1.

Если $\hat{h}(n)=h(n)$, то информированность является полной. В этом случае никакого поиска не происходит и приближение к цели идет по оптимальному пути.

Таким образом, свойства А-алгоритма существенно зависят от условий, которым удовлетворяет или не удовлетворяет эвристическая часть оценочной функции $\hat{f}(n)$:

1) А-алгоритм соответствует алгоритму равных цен, если $h(n)=0$;
2) А-алгоритм гарантирует оптимальное решение, если $\hat{h}(n) \le h(n)$; в этом случае он называется А* – алгоритмом;
3) А-алгоритм обеспечивает однократное раскрытие вершин, если выполняется условие монотонности $\hat{h}(n_i) \le \hat{h}(n_j) + c(n_i, n_j)$;
4) алгоритм A_1^* эвристически более сильный, чем алгоритм A_2^* при $\hat{h}_1(n) > \hat{h}_2(n)$;
5) А*-алгоритм полностью информирован, если $\hat{h}(n)=h(n)$;
6) при $\hat{h}(n)>h(n)$ А-алгоритм не гарантирует получение оптимального решения, однако часто решение получается быстро.

Рассмотренные алгоритмы поиска в пространстве состояний можно систематизировать с помощью таблицы 2.2.

Таблица 2.2 – Алгоритмы поиска в пространстве состояний

<table>
<tr><td>Несистематические</td><td colspan="8">Систематические</td></tr>
<tr><td colspan="2">Неинформированные</td><td colspan="7">Информированные</td></tr>
<tr><td colspan="3">Без оценки затрат</td><td colspan="6">С оценками затрат</td></tr>
<tr><td></td><td>список OPEN – механизм “LIFO”</td><td>список OPEN– механизм “FIFO”</td><td>Стоимость от старта $\hat{g}(n)$</td><td colspan="2">Стоимость до цели $\hat{h}(n)$</td><td colspan="3">Стоимость от старта к цели $\hat{f}(n)=\hat{g}(n)+\hat{h}(n)$</td></tr>
<tr><td rowspan="2">Случайный поиск (Random)</td><td rowspan="2">Поиск в глубину (Depth)</td><td rowspan="2">Поиск в ширину (Breadth)</td><td rowspan="2">Алгоритм равных цен (Optimal search)</td><td rowspan="2">Алгоритм подъема на гору (Hill climbing)</td><td rowspan="2">Глобальный учет соответствия цели (Best-First)</td><td colspan="3">$\hat{h}(n) \le h(n)$
А*-алгоритм</td></tr>
<tr><td colspan="3">$\hat{h}(n_i) \le \hat{h}(n_j) + c(n_i, n_j)$
А* с ограничением монотонности</td></tr>
</table>

Поиск решений в пространстве состояний может выполняться в нескольких направлениях: из начального состояния в целевое состояние (*поиск, управляемый данными*); из целевого состояния в начальное состояние (*поиск, управляемый целью*, или обратный поиск); одновременно в двух направлениях (комбинированный поиск). Так как поисковые графы (деревья) для практических задач велики, то двунаправленный поиск может быть

эффективнее однонаправленного [35].

Поиск решений в пространстве состояний – комбинаторная задача. Решение получается на основе перебора всех состояний. Однако для многих практических задач полный перебор невозможен. Были предложены различные меры для оценки сложности решаемой задачи. Одна из них – это *степень разветвления* вершин графа состояния. Она равна среднему числу дочерних вершин, приходящихся на одно состояние задачи. Если рассматривать дерево состояний, то можно вывести формулу, устанавливающую зависимость общего числа состояний N от степени разветвления B и глубины дерева L [77]:

$$N = B(B^L - 1)/(B - 1).$$

Пусть, например, необходимо подсчитать общее число состояний для игры-головоломки “8”.Укажем возможные перемещения фишек:

- по два перемещения из каждого угла, т.е. для четырех углов восемь перемещений;
- по три перемещения из центральных позиций вдоль каждой из сторон, т.е. всего двенадцать перемещений;
- четыре перемещения из центральной позиции.

Общее число возможных перемещений 24. Разделив это число на 9 (число различных положений для пустой ячейки), получим степень разветвления 2,67. Теперь можно определить общее число состояний при просмотре дерева состояний глубины L (таблица 2.3).

Таблица 2.3 – Число состояний при игре “в восемь”

L	2	3	4	5	6	7	8	9	10
N	10	29	80	215	577	1545	4127	11024	29436

Как видно из таблицы 2.2, даже для такой простой задачи количество просматриваемых состояний резко возрастает при увеличении глубины поиска.

Сложность решаемой задачи можно также оценивать размерами списков OPEN и CLOSED. Для того чтобы эти списки не разрастались, можно сохранять в списке OPEN только несколько наиболее “перспективных” состояний. Это, конечно, сокращает пространство поиска, но возникает опасность пропуска решения. Такие стратегии поиска, накладывающие ограничения на размер списка OPEN, соответствуют *лучевому поиску* (beam search). Более безопасны стратегии эвристического поиска, рассмотренные выше. Чем больше информированность о решаемой задаче, тем больший вес имеет эвристическая часть оценочной функции и тем больше сокращается пространство поиска. Следует обратить внимание на то, что вычисление

эвристик также требует определенных затрат. В этом случае возникает вопрос минимизации общего процессорного времени, затрачиваемого на поиск решения [29].

2.2.3. Поиск с распространением ограничений

Существуют задачи, где необходимо принимать во внимание действующие ограничения на переходы состояний, и благодаря этому целенаправленно идти к её решению. В качестве примера можно привести задачу распознавания речи или задачу интерпретации трехмерных контурных рисунков.

Одна из технологий распознавания речи основана на разбиении речевого сигнала на элементарные участки и присвоении каждому участку метки в виде фонемы. Последовательность фонем, получаемая в результате такой разметки, должна быть согласованной и не противоречить правилам произнесения слов. Ограничения, действующие между возможными взаимными сочетаниями фонем, позволяют исключить из рассмотрения варианты, которые не формируют акустические аналоги слов. Таким образом, элементарные участки речи, представленные фонемами, обеспечивают формирование правильного слова, а последовательность слов – грамматически корректную фразу. В этом смысле накладываемые ограничения позволяют по элементам восстанавливать целое и их часто называют *ограничениями целостности*.

Если одна и та же переменная участвует одновременно в нескольких ограничениях, то образуются сети (системы) ограничений. Решение задачи в этом случае состоит в определении всех сочетаний значений переменных, которые удовлетворяют ограничениям целостности.

Рассмотрим задачу согласованной разметки (consistent labeling), при решении которой анализируется m элементов, пронумерованных $n_1, n_2, \ldots, n_m$. Предположим, что каждый элемент n_i может быть помечен (интерпретирован) несколькими способами. Количество возможных интерпретаций каждого элемента задается числами $b_1, b_2, \ldots, b_m$. Если рассматривать возможные интерпретации всей совокупности элементов и проверять их на соблюдение ограничений целостности (т.е. выяснять отображают ли они допустимые решения), то необходимо проанализировать $b_1 \times b_2 \times b_3 \times \ldots \times b_m$ случаев. Хотя данная задача легко представляется в виде графа состояний, целесообразнее все же перейти к другому представлению – к графу, вершины которого есть элементы n_i, а ребра соответствуют бинарным отношениям между соответствующими элементами.

Пусть для каждого из элементов $n_i \in \{n_1, n_2, ..., n_m\}$ имеется конечное множество интерпретации (меток) L_i мощностью G_i. Возможность или невозможность согласованной интерпретации двух элементов n_i и n_j с помощью меток $l_i \in L_i$ и $l_j \in L_j$ задается бинарным отношением b_{ij}. Если определенная метка l_i для элемента n_i не согласуется с соответствующей меткой l_j для элемента n_j (т.е. не удовлетворяет отношению b_{ij}), то данная метка удаляется из множества L_i. Аналогичным образом исследуются все метки, входящие во множество L_j. В ходе указанного процесса с каждой вершиной графа связывается метка (или множество меток). При этом метки на концах ребер графа должны удовлетворять заданным отношениям.

Рассмотрим пример согласованной разметки вершин графа, изображенного на рисунке 2.10 [86]. Около вершин графа указаны множества возможных меток, а ограничения, которым должны удовлетворять соответствующие метки, записаны рядом с ребрами. Основная идея состоит в сужении множества возможных меток для каждой из вершин, опираясь на заданные ограничения.

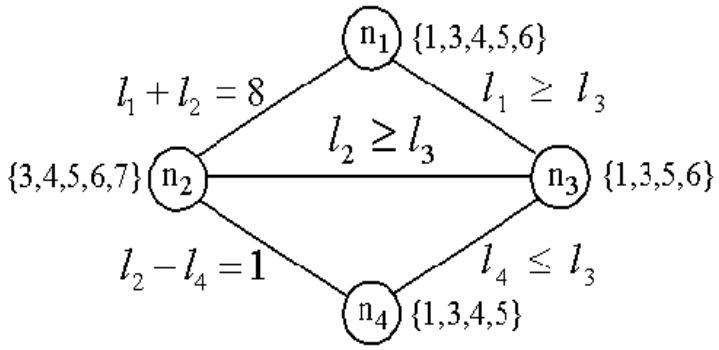


Рисунок 2.10 – Согласованная разметка узлов графа

Рассмотрим множества совместимых меток для вершин n_1 и n_2 при ограничении $l_1+l_2=8$. Сначала при заданном множестве $L_1=\{1,3,4,5,6\}$ удалим из L_2 все метки, не удовлетворяющие указанному условию. В данном случае отношение b_{12} выполняется для всех меток из L_2, за исключением метки 6. Следовательно, данная метка удаляется, и $L_2=\{3,4,5,7\}$. Затем отношение b_{12} используется для проверки множества меток L_1. В этом случае из L_1 также удаляется метка 6, и множество L_1 будет состоять из меток {1,3,4,5}. Таким образом, мы сначала распространили ограничение b_{12} из вершины n_1 в вершину n_2, а затем обратно из n_2 в n_1, удаляя соответствующие метки из множеств L_2 и L_1. Метки, входящие во множества L_2 и L_1, теперь согласованы.

На следующем шаге для анализа выбирается вершина, которая связана с наибольшим числом вершин, уже рассмотренных к настоящему моменту. В данном случае это вершина n_3. Распространив ограничения от n_1 к

n_3 и от n_2 к n_3, получим L_3={1,3,5}. Затем соответствующие ограничения распространяются в обратную сторону. При этом множества L_2 и L_1 не изменяются. Наконец, аналогичные операции выполняются с вершиной n_4. В итоге L_4={3,4}, L_2={4,5}, L_3={3,5}. Затем ограничения распространяются между n_2 и n_3 (это дает L_3={3}), а также от n_2 и n_3 к n_1. Получаем L_1={3,4}. Распространяя ограничения обратно от n_1 к n_2, n_3 и n_4, получаем окончательный результат: l_1=4, l_2=4, l_3=3, l_3=2. Распространение ограничений продолжается до тех пор, пока на множествах меток происходят изменения.

В общем случае вершина может иметь несколько меток. При этом произвольно комбинировать метки для разных вершин нельзя. Необходимо учитывать существующие ограничения. Очевидно, в этом случае задача имеет несколько решений.

В главе 10 рассмотрено применение метода распространения ограничений для решения задачи интерпретации контурных рисунков трехмерных тел (алгоритм Хаффмана-Клоуза).

2.3. Поиск решений при сведении задач к подзадачам

При таком представлении задачи поиск решений выполняется с использованием И-ИЛИ графов. Рассмотрим особенности поиска в И-ИЛИ графах в сравнении с поиском в пространстве состояний.

В пространстве состояний решение представляет путь из начальной вершины в целевую вершину. Решение в И-ИЛИ графах соответствует подграфу или поддереву, так как при раскрытии И-вершины необходимо определить, являются ли разрешимыми все ее дочерние вершины. Наличие ИЛИ-вершины приводит к тому, что в ходе поиска решения рассматривают несколько возможных подграфов решений. Если необходимо найти оптимальный граф решений, то вводятся оценочные функции. Однако, в отличие от поиска в пространстве состояний, эти функции представляют не стоимости путей, а стоимости графов, в частности, графа решения или его подграфов.

В процессе поиска строятся подграфы, которые выступают в роли претендентов на участие в формировании полного графа решения. Будем называть такие подграфы *потенциальными подграфами решений*. Обычно в ходе поиска строятся несколько потенциальных подграфов, которые определенным образом представляются в памяти ЭВМ. Построение указанных подграфов выполняют повторением двух обобщенных действий:

а) определяют, являются ли вершины потенциального подграфа решения разрешимыми, неразрешимыми (тупиковыми) или нераскрытыми; если необходимо, то вычисляют стоимости соответствующих подграфов;

б) расширяют потенциальный подграф решений.

В ходе выполнения первого действия проверяют все концевые вершины потенциального подграфа. Если эти вершины соответствуют элементарным задачам, решение которых известно, то они считаются заключительными и помечаются меткой SOLVED (разрешимые). Если концевые вершины являются тупиковыми, то они помечаются как неразрешимые (метка UNSOLVED). Затем анализируются их родительские вершины. Родительская И-вершина помечается как разрешимая, если разрешимыми являются все ее дочерние вершины, иначе она помечается как неразрешимая. Родительская ИЛИ-вершина получает метку, совпадающую с меткой дочерней вершины, рассматриваемой в данный момент.

Если начальная вершина потенциального подграфа решений помечается как разрешимая, то данный потенциальный подграф становится возможным подграфом решения.

В случае поиска оптимального графа решения оценивается стоимость всех вершин потенциальных подграфов решений, начиная с концевых вершин. Правила указанных вычислений будут рассмотрены ниже.

При выполнении второго действия раскрывают концевые вершины потенциального подграфа решений, которые не имеют меток. Если раскрываемая концевая вершина является И-вершиной, то генерируется новый потенциальный подграф путем добавления всех дочерних вершин к расширяемому потенциальному подграфу. Если раскрываемая концевая вершина – это ИЛИ-вершина, то к расширяемому потенциальному подграфу добавляется только одна дочерняя вершина. При этом формируется столько потенциальных подграфов, сколько имеется вершин.

Аналогично поиску в пространстве состояний при поиске в пространстве редукции задачи выделяют алгоритмы слепого и упорядоченного (эвристического) поиска.

2.3.1. Алгоритм поиска в глубину

В рассматриваемом ниже алгоритме элементами списка OPEN являются потенциальные подграфы решений, т.е. подграфы, начальная вершина которых не имеет метки или имеет метку SOLVED (подграф решения). В процессе поиска вновь формируемые потенциальные подграфы решений помещаются в начало списка OPEN.

```
Procedure Depth_First_AO1;
Begin
        Поместить в список OPEN потенциальный подграф, состоящий
        только из начальной вершины S;
        Выполнить разметку данного подграфа;
```

```
While OPEN <> 'пустой список' Do
Begin
    P:= first (OPEN); {выбрать из OPEN первый подграф}

    {применить к подграфу P функцию проверки разрешимости
    solved(P) }
    IF solved(P) THEN выход ('УСПЕХ');

    Удалить подграф P из списка OPEN;
    Расширить подграф P и выполнить разметку вновь сформиро-
    ванных потенциальных подграфов решений;
    Поместить все новые потенциальные подграфы решений в на-
    чало списка OPEN;
End;
Выход ('НЕУДАЧА')
End.
```

Рассмотрим применение процедуры к графу редукции, изображенному на рисунке 2.11. Здесь концевые вершины, отмеченные прописными буквами, соответствуют неразрешимым задачам, а вершины, отмеченные строчными буквами, являются заключительными (терминальными).

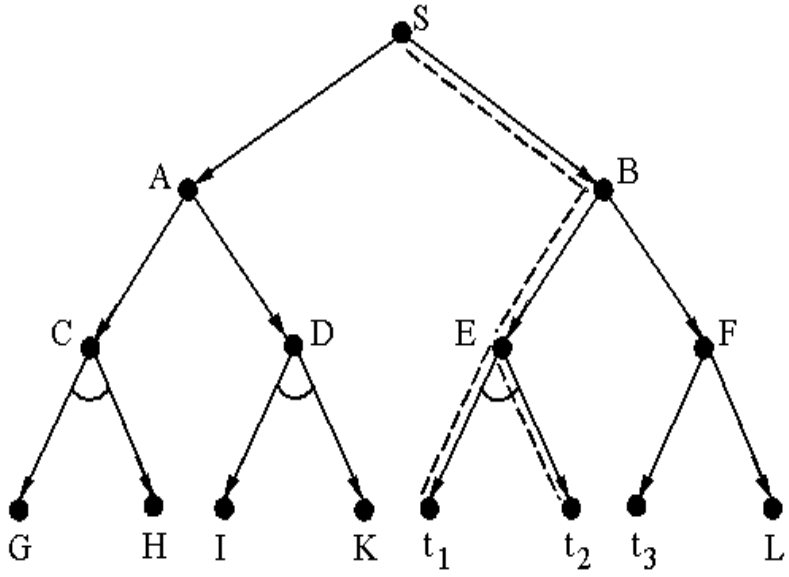


Рисунок 2.11—Граф редукции задачи

На рисунке 2.12 представлены потенциальные подграфы решений, помещаемые в список OPEN на разных шагах выполняемой процедуры. На шаге 4 расширяемый потенциальный подграф решений помечается как неразрешимый и из дальнейшего рассмотрения исключается. Аналогичная ситуация имеет место на шаге 5. В дальнейшем расширяется правый подграф вершины S, который и приводит к графу решения, показанному на рисунке 2.11 штриховой линией.

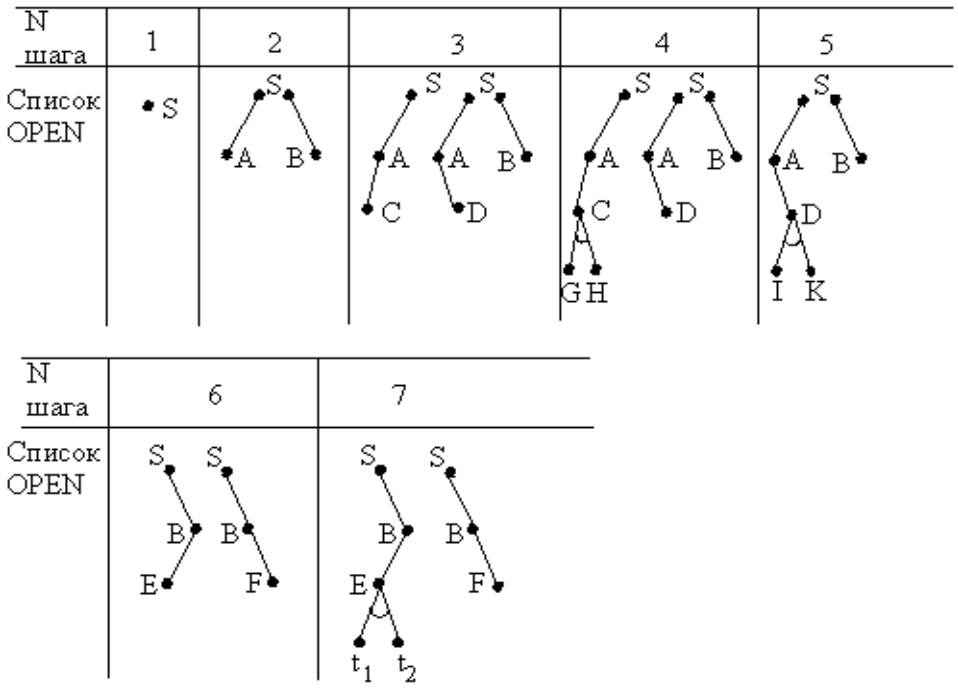


Рисунок 2.12 – Потенциальные графы решений при поиске в глубину

В рассматриваемом примере в списке OPEN хранятся несколько потенциальных подграфов решений. Если граф редукции задачи является сложным, то это приводит к неэффективному использованию памяти ЭВМ. Для устранения указанного недостатка, необходимо при раскрытии очередной вершины потенциального подграфа решения рассматривать не все дочерние вершины, а только одну. С этой целью в процедуру поиска вводится функция *NextChild* [86].

Пусть у вершины n имеются дочерние вершины $n_1,n_2,\ldots,n_i,\ldots,n_m$ и вершина n_i уже включена в потенциальный граф решения. Тогда функция *NextChild* определится следующим образом

$$NextChild\ (n) = \begin{cases} n_{i+1}, 0 \le i < m, \\ nil, i = m, \end{cases}$$

где i=0 до применения функции к вершине n. В этом случае процедуру поиска в глубину можно переписать в виде:

```
Procedure Depth_first_AO2;
Begin
    Создать потенциальный подграф P, состоящий только из одной
    начальной вершины S;
    n:=S;
    Выполнить разметку подграфа P;
```

```
While True Do
Begin
    If unsolved (P) Then выход ('НЕУДАЧА');
    If solved (P) Then выход ('УСПЕХ');
    If n не имеет метки Then
      Begin
        n:=NextChild(n);
        Построить новый потенциальный подграф решения,
        добавив в P вершину n;
        Выполнить разметку нового потенциального подграфа
        и обозначить его через P;
      End
    Else
      Begin
        If n имеет метку UNSOLVED Then
        Удалить вершину n из подграфа P;
        n:=родительская вершина (n)
      End
End
End.
```

Если данную процедуру применить к дереву редукции задачи, изображенному на рисунке 2.11, то порядок обработки вершин будет следующим: S, A, C, G(UNSOLVED), C(UNSOLVED), A, D, I(UNSOLVED), D(UNSOLVED), A(UNSOLVED), S, B, E, t1(SOLVED), E, t2(SOLVED), E(SOLVED), B(SOLVED), S(SOLVED).

Когда вершина помечается меткой UNSOLVED, то она исключается из потенциального подграфа решения. В итоге в подграф решения включаются только вершины, помеченные меткой SOLVED.

Каждый раз, когда обнаружена неразрешимая вершина, процедура выполняет возврат к родительской вершине, обеспечивая обследование других альтернативных ветвей графа редукции задачи. Если обнаружена разрешимая вершина и при этом еще не законченно построение подграфа решения, то возврат выполняется с целью обследования оставшихся конъюнктивных ветвей. Благодаря возврату нет необходимости раскрывать все дочерние вершины одновременно и запоминать альтернативные потенциальные подграфы решений.

2.3.2. Алгоритм эвристического поиска в И-ИЛИ графе

Алгоритмы эвристического поиска предполагают упорядочение списка OPEN в соответствии с некоторой эвристической оценочной функцией. При поиске в пространстве состояний эвристическая функция представляла собой оценку стоимости пути между текущей и целевой верши-

нами. В случае поиска в И-ИЛИ графе эвристическая функция соответствует оценке стоимости подграфа решения.

Обозначим через *h(n)* стоимость подграфа решения, начинающегося в вершине *n*. Стоимость подграфа решения определяется рекурсивно:

а) если *n* соответствует заключительной вершине, то *h(n)*=0;

б) для любой другой вершины

$$h(n) = \sum_{i=1}^{k} [h(n_i) + c(n, n_i)],$$

где n_i, i=1,2,…,k – дочерние вершины для вершины n; $c(n,n_i)$ – стоимость дуги между вершинами n и n_i. Определение стоимости подграфа решения начинается с заключительных вершин и заканчивается начальной вершиной.

В ходе поиска подграфа решения рассматриваются потенциальные подграфы решений, для которых концевые вершины – это, как правило, еще нераскрытые вершины. Для определения стоимости потенциальных подграфов решений вводятся оценки *h(n)*, обозначаемые $\hat{h}(n)$. При этом $\hat{h}(n) = 0$, если n – заключительная вершина, и

$$\hat{h}(n) = \sum_{i=1}^{k} [\hat{h}(n_i) + c(n, n_i)], \qquad (2.3)$$

где n – рассматриваемая вершина. Оценка $\hat{h}(n)$ строится на основе эвристической информации.

Рассмотрим пример. На рисунке 2.13 изображен потенциальный граф решения. Здесь пунктирными линиями показаны не построенные ветви. Цифры в скобках рядом с вершинами C и D обозначают эвристические оценки подграфов решений, начинающихся в этих вершинах. Цифры без скобок (вершины A, S) обозначают оценки, вычисляемые в соответствии с формулой (2.3). Стоимость дуги полагается равной единице.

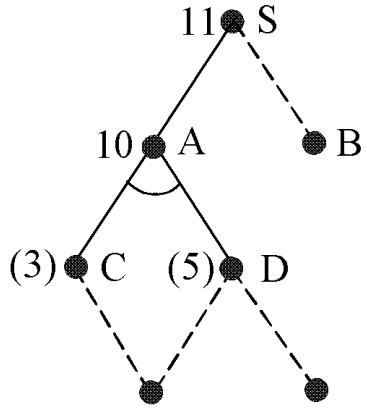


Рисунок 2.13 – Потенциальный граф решения

При известных эвристических оценках подграфов решений, начинающихся в вершинах C и D, оценка подграфа для вершины A равна

$$\hat{h}(A) = \hat{h}(C) + c(A,C) + \hat{h}(D) + c(A,D) = 10,$$

а полная оценка стоимости $\hat{h}(S)$ изображенного потенциального графа равна 11.

Эвристический алгоритм упорядоченного поиска можно представить в виде процедуры:

```
Procedure AO_Search;
Begin
    Поместить в список OPEN потенциальный граф P, состоящий
    только из начальной вершины S;
    Выполнить разметку графа P;
    Вычислить оценку ĥ(S);
    While OPEN <> ' пустой список' Do
    Begin
        P:=first(OPEN);{выбрать из OPEN первый граф}
        If solved(P) Then выход ('УСПЕХ');
        Удалить граф P из списка OPEN;
        Расширить граф P, выполнить разметку всех вновь сформи-
        рованных потенциальных графов P_i, определить оценки
        ĥ(P_i);
        Внести потенциальные графы P_i в список OPEN с учетом
        оценок ĥ(P_i)
    End
    Выход ('НЕУДАЧА')
End.
```

В рассматриваемой процедуре используется определенный порядок раскрытия вершин. Если раскрываемая вершина *n* является дизъюнктивной, то в новый потенциальный подграф решения включается только одна ее дочерняя вершина. Если раскрываемая вершина конъюнктивная, то в новый потенциальный подграф решения включаются все ее дочерние вершины. В дальнейшем, в первую очередь, раскрывается та из дочерних вершин, которая имеет максимальную оценку $\hat{h}(n_i)$. Основанием такого выбора является то, что необходимо как можно скорее выяснить, разрешима или не разрешима родительская И-вершина *n*. И-вершина неразрешима, если неразрешима хотя бы одна из ее дочерних вершин. Если этот факт будет установлен, то подграф потенциального графа решений, начинающихся в вершине *n*, из дальнейшего рассмотрения исключается. Вероятность обнаружения неразрешимости И-вершины повышается, если двигаться в направлении максимальной оценки $\hat{h}(n_i)$.

Рассмотрим применение процедуры эвристического поиска к графу редукции задачи, изображенному на рисунке 2.14. На рисунке 2.15 показаны потенциальные графы решений, хранящиеся в списке OPEN. На шаге 3 раскрывается вершина B, имеющая оценку $\hat{h}(B) = 5$. Раскрытие вершины B

приводит к подзадачам D и I с оценками 2 и 5. В результате этого рассматриваемый потенциальный подграф получает оценку 10 и переводится на второе место списка OPEN. В дальнейшем строится левый подграф вершины S и на шаге 7 потенциальный граф помечается как граф решения.

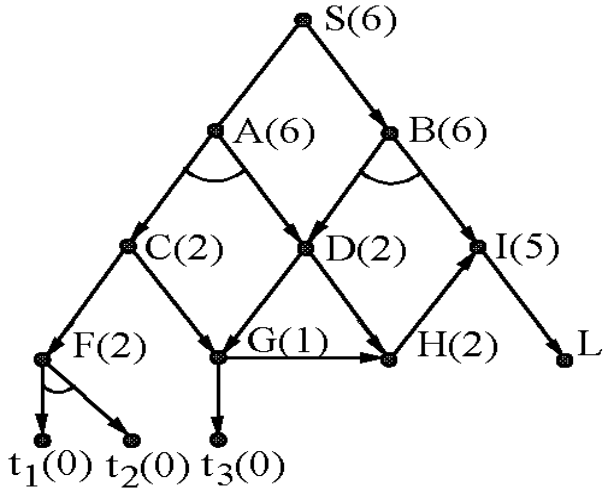
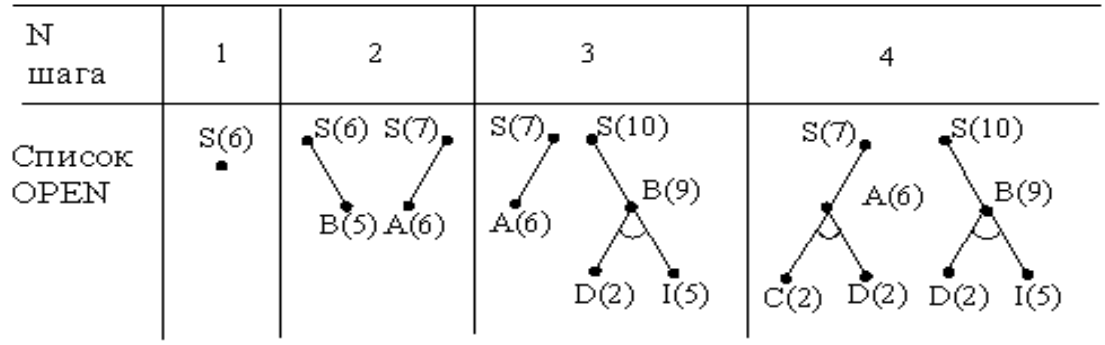


Рисунок 2.14 – Граф редукции задачи с оценками $\hat{h}(n)$



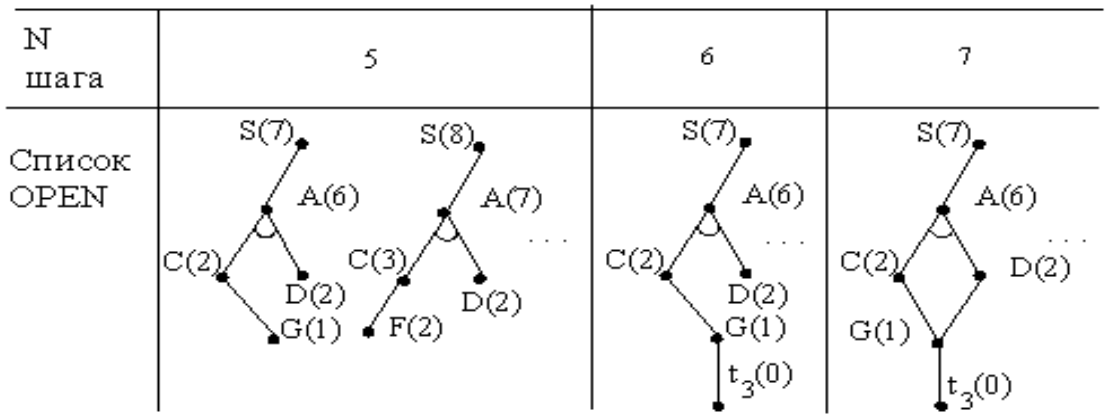


Рисунок 2.15 – Потенциальные подграфы решений

Аналогично поиску в глубину в рассматриваемую процедуру можно включить функцию NextChild(n). Это позволит повысить эффективность

использования памяти ЭВМ при поиске в графах редукции задач с большим числом вершин.

Если алгоритм поиска находит оптимальный граф решения (такой граф имеет минимально возможную оценку $\hat{h}(s)$), то он называется гарантирующим. Можно показать, что алгоритм поиска в И-ИЛИ графе будет гарантирующим, если $\hat{h}(n) \le h(n)$ и стоимости всех дуг $c(n, n_i)$ больше некоторой положительной величины [86].

Если положить стоимость всех дуг равной нулю, то будут минимизироваться только предстоящие затраты.

2.3.3. Поиск решений в игровых программах

В сложных играх, например шахматах, лучшие игроки (люди) в большинстве случаев все еще превосходят компьютерные программы. Пока не существует принципов построения программ, позволяющих компьютерам выиграть шахматную партию наверняка. В то же время игра в шахматы с помощью компьютера представляет собой проблему, которой в ИИ уделяется большое внимание. Прогресс в этой области оценивается как прогресс в ИИ. Помимо шахмат, большое внимание в ИИ уделялось шашкам, игре “го”, различным вариантам игры в крестики-нолики и др. [55]

Все эти игры характеризуются как игры с полной информацией о текущей игровой ситуации (имеются игры с неполной информацией), где два игрока-противника по очереди делают ходы. Успех одного игрока – такая же по величине потеря для другого игрока (игры с нулевыми суммами). Допустимые ходы игры определяются конечным набором правил, которые исключают случайность. Игра начинается из специфического начального положения и завершается победой того или иного игрока, либо ничьей. При рассмотрении игровых программ игроком считается компьютер, а противником – человек .

Сложность поиска решений в игровых деревьях весьма высока, так как вершины имеют высокую степень разветвления. Например, в середине шахматной партии игрок может сделать около 35 ходов, разрешенных правилами. Общее количество вершин дерева игры в шахматы оценивается значением 10^{120}. Это число сравнимо с числом молекул во Вселенной или количеством наносекунд, прошедших от момента “Большого взрыва”. Если предположить, что можно обследовать 10^9 вершин в секунду, то построение полного дерева игры в шахматы заняло бы 10^{103} лет [77]. Из этого следует, что полный просмотр такого дерева невозможен. Поэтому при поиске в игровых деревьях необходимо выполнять следующее:

1) прогнозировать граничную глубину поиска;

2) оценивать перспективность позиций игры с помощью эвристических функций.

Для этого в каждой позиции игры формируется дерево возможных продолжений игры, имеющее определенную глубину, и с помощью эвристической оценочной функции вычисляются оценки концевых вершин такого дерева. Затем полученные оценки распространяются вверх по дереву, и корневая вершина, соответствующая текущей позиции, получает оценку, позволяющую выяснить силу игрока в данной позиции. При игре в шахматы современные компьютерные программы просматривают возможные продолжения игры примерно на 10 ходов вперед. Основная идея такого оценивания заключается в том, что просмотр возможных продолжений игры из заданной позиции на определенную глубину позволяет точнее оценить перспективность того или хода.

2.3.3.1. Минимаксный метод. В соответствии с минимаксным методом вместо полного просмотра дерева игры обследуется лишь его небольшая часть. В этом случае говорят, что дерево подвергается *подрезке*. Простейший способ подрезки — это просмотр дерева игры на определенную глубину. Это означает, что процесс обследования заканчивается в вершинах, которые не являются терминальными (концевыми). Поэтому *дерево поиска* – это только верхняя часть дерева игры. Для оценки силы игрока в различных позициях поискового дерева применяются оценочные функции. Причем, чем выше оценка в соответствующей позиции, тем больше шансов на выигрыш у игрока, а чем ниже оценка, тем выигрышнее позиция противника. Поэтому игрок стремится максимально увеличить выигрыш, а противник – минимизировать проигрыш. Обычно в алгоритмах позиции игрока обозначают через МАКС (MAX), а позиции противника – через МИН (MIN).

Различают два вида оценок: статические и динамические. *Статические оценки* приписываются концевым вершинам дерева поиска и вычисляются на основе эвристических правил. При игре в шахматы эти правила учитывают перевес фигур, контроль центра, подвижность фигур и пр. *Динамические оценки* получаются при распространении статических оценок вверх по дереву. Метод, которым это достигается, называется *минимаксным.* Заключается он в следующем. Для вершины, в которой ход выполняет игрок (МАКС), выбирается наибольшая из оценок вершин нижнего уровня (т.е. уровня МИН'а). Это абсолютно оправдано, так как в этом случае игрок сделает для себя наиболее выгодный ход. Для вершины, в которой ход выполняет противник, выбирается наименьшая из оценок дочерних вершин, так как противник стремится сделать ход наименее выгодный игроку. На рисунке 2.16 показан пример распространения статических оценок вверх по поисковому дереву в соответствии с указанным минимаксным

принципом. Из рисунка следует, что лучшим ходом МАКС'а в вершине *S* будет ход *S-D*, а лучшим ходом МИН'а в вершине *D* будет ход *D-L*.

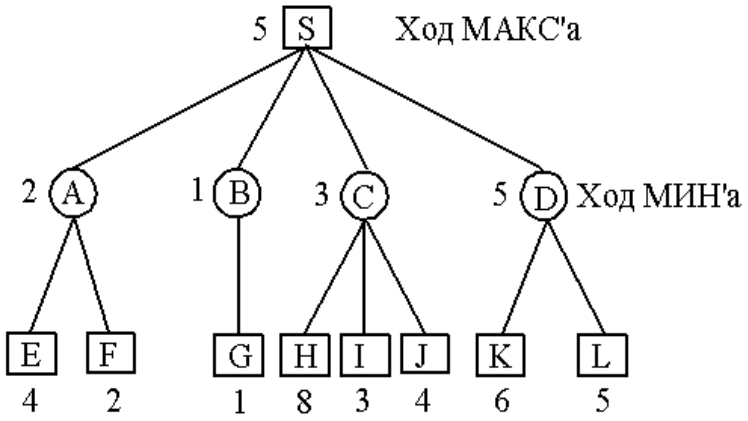


Рисунок 2.16 – Распространение статических оценок

Если обозначить статическую оценку в вершине *P* через *h(P)*, а динамическую – через *H(P),* то формально оценки, приписываемые вершинам в соответствии с минимаксным принципом, можно записать так [7]:

$$H(P)=h(P),$$

если *P* – терминальная вершина дерева поиска;

$$H(P)=\max_i H(P_i),$$

если *P* – вершина с ходом МАКС'а;

$$H(P)=\min_i H(P_i),$$

если *P* – вершина с ходом МИН'а. Здесь P_i дочерние вершины для вершины *P*.

Многое в минимаксном методе зависит от оценочной функции. Если бы имелась абсолютно точная оценочная функция, то можно было гарантировать выигрыш. На практике для приближения минимаксной стратегии игры к гарантирующей приходится увеличивать глубину поиска. Поэтому разработаны подходы, позволяющие сократить объем перебора при обследовании дерева поиска. Один из них базируется на методе альфа-бета поиска.

2.3.3.2. Альфа-бета поиск. Минимаксный метод поиска предполагает полное обследование дерева поиска. Чтобы получить оценку корневой вершины, совсем не обязательно выполнять систематический обход дерева поиска. Рассмотрим пример (рисунок 2.17) альфа-бета поиска для случая, когда обход дерева выполняется сверху вниз и слева направо.

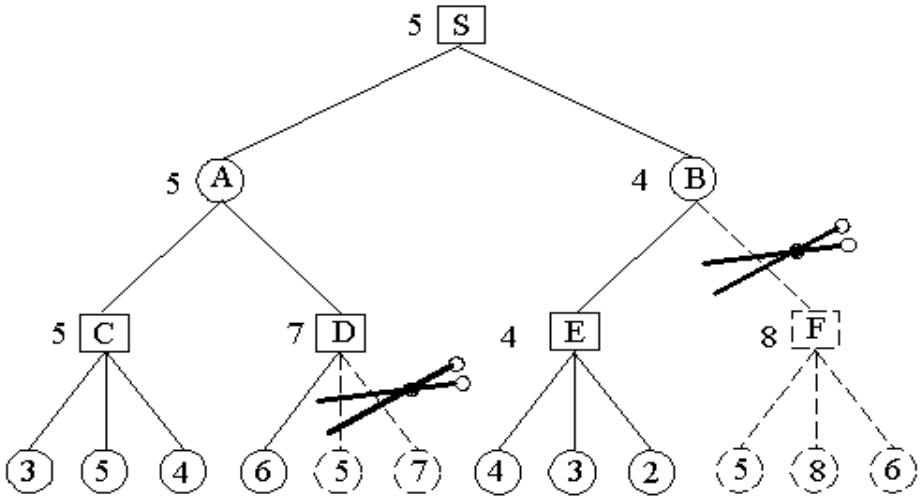


Рисунок 2.17 – Альфа-бета поиск

Процесс поиска начинается в позиции *S* и идет в направлении вершины *C*, для которой выбирается максимальная из оценок вершин-приемников позиции *C*, т.е. *H(C)=5*. Затем происходит возврат к вершине *A* и оценивание вершины *D*. Для этого рассматривается первая вершина-приемник позиции *D* с оценкой 6. В этот момент обнаруживается, что игроку в вершине *D* гарантирована оценка не меньшая, чем 6. Этого достаточно чтобы понять, что для противника позиция *D* хуже, чем *C*. Поэтому можно пренебречь вторым и третьим приемником позиции *D* и связать с вершиной *D* приближенную оценку 6. Точно также при обследовании правого поддерева вершины *S* игроку в позиции *E* гарантируется оценка не меньшая, чем 4. Следовательно, противнику в позиции *B* гарантируется оценка не выше 4. Отсюда следует, что из позиции *S* лучше сделать ход *S-A*. Поэтому правое поддерево вершины *B* можно не анализировать. Таким образом, сложность поиска уменьшилась за счет отсечения части дерева поиска.

В альфа-бета методе статическая оценка каждой терминальной вершины вычисляется сразу, как только такая вершина будет построена. Затем полученная оценка распространяется вверх по дереву и с каждой из родительских вершин связывается предположительно возвращаемая оценка (ПВО). При этом гарантируется, что для родительской вершины, в которой ход выполняет игрок (ее также называют *альфа-вершиной*), уточненная оценка, вычисляемая на последующих этапах, будет не ниже ПВО. Если же в родительской вершине ход выполняет противник (*бета-вершина*), то гарантируется, что последующие оценки будут не выше ПВО. Это позволяет отказаться от построения некоторых вершин и сократить объем поиска. При этом используются следующие правила:

1) если ПВО для бета-вершины становится меньше или равной ПВО родительской вершины, вычисленной на предыдущем шаге, то нет необходимости строить дальше поддерево, начинающееся ниже этой бета-вершины;

2) если ПВО для альфа-вершины становится больше или равной ПВО родительской вершины, вычисленной на предыдущем шаге, то нет необходимости строить дальше поддерево, начинающееся ниже этой альфа-вершины;

Первое правило соответствует *альфа-отсечению*, а второе – *бета-отсечению*. Для дерева поиска, изображенного на рисунке 2.17, ситуация альфа-отсечения имеет место при вычислении ПВО вершины *D* ([*H(D)*=6] > *H(A)*=5]), а ситуация бета-отсечения – для вершины *B* ([*H(B)*=4] < [*H(S)*=5]).

Результаты применения альфа-бета поиска зависят от порядка, в котором строятся дочерние вершины. В худшем случае альфа-бета поиск будет соответствовать минимаксному поиску.

Всегда первыми лучше рассматривать самые сильные ходы каждой из сторон. В этом случае альфа-бета алгоритм вычисляет статические оценки только в N вершинах, число которых равно числу терминальных вершин поискового дерева. Альфа-бета поиск позволяет увеличить глубину дерева поиска примерно в два раза по сравнению минимаксным алгоритмом, что приводит к более сильной игре [7].

Вопросы для самопроверки

1. Какие способы представления задач Вы знаете?
2. Приведите пример представления задачи в пространстве состояний.
3. Приведите пример представления задачи в пространстве ее подзадач.
4. Сформулируйте общую идею представления задач в виде доказательства теорем.
5. Напишите на псевдоязыке процедуры поиска в глубину и ширину, объясните их различие с алгоритмической точки зрения.
6. Напишите на псевдоязыке процедуру оптимального поиска в соответствии с алгоритмом равных цен.
7. Объясните основной принцип построения процедур эвристического поиска.
8. Запишите на псевдоязыке А-алгоритм.
9. Сформулируйте свойства А-алгоритма.
10. Сравните алгоритмы “слепого” и упорядоченного поиска по критерию гарантированности получения результата и эффективности поиска.
11. Что представляет собой решение при поиске в И-ИЛИ графах?
12. Объясните алгоритм поиска в глубину при сведении задачи к подзадачам.
13. Объясните алгоритм эвристического поиска в И-ИЛИ графах.
14. Разработайте пример дерева игры в крестики-нолики и предложите способ вычисления статических оценок.
15. Объясните на примере принцип минимаксной игры.

ГЛАВА 3
ПРЕДСТАВЛЕНИЕ ЗНАНИЙ

Представление знаний и поиск решений образуют ядро искусственного интеллекта. В предыдущей главе были рассмотрены универсальные методы поиска решений. Сосредоточим свое внимание на основных моделях представления знаний.

Понятие “знание” трактуется отдельными авторами по-разному и во многом носит дискуссионный характер. В главе рассматривается наиболее распространенное толкование этого понятия.

Модель представления знаний должна отражать существенные характеристики решаемой задачи и обеспечивать соответствующей информацией процедуры, выполняющие поиск решений. При этом она должна обладать необходимой выразительной способностью, чтобы отражать все интересующие детали предметной области задачи, а также быть достаточно эффективной с точки зрения поиска решений, который рассматривают как вывод на знаниях. Выразительная сила и вычислительная эффективность – две основные характеристики моделей представления знаний. Многие модели представления знаний, обладающие большими выразительными возможностями, не поддаются эффективной реализации. Поэтому поиск оптимального соотношения между выразительной силой используемой модели представления знаний и эффективностью ее реализации – основная задача разработчика СИИ.

Изучение моделей представления знаний начинается с логической модели, базирующейся на исчислении предикатов. Такая модель характеризуется хорошо определенной семантикой и формально обоснованными правилами вывода. Используя логику, вводятся основные понятия представления знаний, и демонстрируется взаимосвязь представления знаний с поиском решений (выводом).

Кроме логических моделей, в СИИ широко применяют такие модели представления знаний, как продукционные системы, семантические сети, фреймы.

3.1. Знания и их представление в СИИ

Что же представляют собой знания? Четкий ответ на этот вопрос дан в энциклопедическом словаре. *Знание* – это “проверенный практикой результат познания действительности, верное её отражение в мышлении человека”. Отражение существующего мира в мышлении человека связано с процессом абстрагирования, который заключается в выделении наиболее существенных свойств и признаков явлений или объектов, наблюдаемых в окружающей действительности, и представлении их в такой упрощенной форме, которая необходима для логических умозаключений и принятия эффективных решений в самых различных ситуациях. Указанное упрощенное представление действительности называют моделью. Человек, познавая окружающий мир, строит модели явлений, процессов, предметов и т.п. – объектов реального мира. *Модель* является информационным эквивалентом части реального мира (предметной области) и лежит в основе процесса познания. В инженерной практике именно модели ассоциируются со знанием. Это знания, на основе которых можно эффективно использовать объекты реального мира для достижения определенных целей. Однако отсюда не следует, что эти термины эквивалентны. Понятие “знание” шире понятия “модель”. Хотя, безусловно, знания в основном состоят из моделей. Знания включают в себя не только те или иные сведения об объектах реального мира, но и информацию о механизмах вывода новых знаний на основании имеющихся.

В настоящее время в искусственном интеллекте нет формального единого определения понятия “знание”. Однако многими исследователями признается, что *знания* – сложноорганизованные данные, хранимые в памяти СИИ и включающие в себя сведения об объектах и отношениях предметной области, процессах взаимодействия объектов во времени и в пространстве, правилах осуществления логического вывода [2,16,38]. Важным элементом этого определения является указание на то, что *знание* – это информация, на основе которой выполняется логический вывод.

Знания характеризуются рядом свойств, отличающих их от традиционных моделей данных. Перечислим эти свойства [21].

Внутренняя интерпретируемость. При хранении знаний в памяти СИИ, наряду с традиционными элементами данных, хранятся и информационные структуры, позволяющие интерпретировать содержимое соответствующих ячеек памяти.

Структурированность. Знания состоят из отдельных информационных единиц, между которыми можно установить классифицирующие отношения: род – вид, класс – элемент, тип – подтип, часть – целое и т.п.

Связность. Между информационными единицами предусматриваются связи различного типа: причина – следствие, одновременно, быть рядом и др. Данные связи определяют семантику и прагматику предметной области.

Семантическая метрика. На множестве информационных единиц, хранимых в памяти, вводятся некоторые шкалы, позволяющие оценить их семантическую близость. Это позволяет находить в информационной базе знания, близкие к уже найденным.

Активность. С помощью данного свойства подчеркивается принципиальное отличие знаний от данных. Выполнение тех или иных действий в СИИ инициируется состоянием базы знаний. При этом предполагается, что появление новых фактов и связей может активизировать систему.

Центральным вопросом построения систем, основанных на знаниях, является выбор формы представления знаний. *Представление знаний* – это способ формального выражения знаний о предметной области в компьютерно-интерпретируемой форме. Соответствующие формализмы, обеспечивающие указанное представление, называют *моделями представления знаний*.

Модели представления знаний можно условно разделить на *декларативные* и *процедурные.* В декларативных моделях знания представляются в виде описаний объектов и отношений между объектами без указания в явном виде, как эти знания обрабатывать. Такие модели предполагают отделение описаний (деклараций) информационных структур от механизма вывода, оперирующего этими структурами. В процедурных моделях знания представляются алгоритмами (процедурами), которые содержат необходимые описания информационных элементов и одновременно определяют способы их обработки.

Конкретные модели, применяемые на практике, представляют собой комбинацию декларативных и процедурных представлений. Наиболее распространенными являются следующие модели представления знаний:

- логические модели;
- продукционные модели;
- сетевые модели;
- фреймовые модели.

Логические модели реализуются средствами логики предикатов. В этом случае знания о предметной области представляются в виде совокупности логических формул. Тождественные преобразования формул позволяют получать новые знания. Достоинством логических моделей представления знаний является наличие четкого синтаксиса и широко принятой формальной семантики, а также теоретически обоснованных процедур автоматического вывода. Основным недостатком данных моделей является невозможность получения заключений в областях, где требуются правдо-

подобные выводы, когда результат получается с определенной оценкой уверенности в его истинности. Кроме этого, такие модели характеризуются монотонным характером вывода, т.е. в базу знаний добавляются только истинные утверждения, что исключает возможность противоречий. На практике часто встречаются немонотонные рассуждения, которые трудно реализовать в рамках логической модели.

Тем не менее, логические модели выступают в качестве теоретической основы описания самой системы представления знаний и постепенно расширяют свои возможности. Поэтому в дальнейшем этим моделям уделяется значительное внимание.

В *продукционных моделях* знания представляются набором правил вида "Если *A*, то *B*", где условие правила *A* является утверждением о содержимом базы фактов, а следствие *B* говорит о том, что надо делать, когда данное продукционное правило активизировано.

Продукционные модели представления знаний благодаря естественной модульности правил, наглядности и простоте их создания широко применяются в интеллектуальных системах.

Семантические сети являются частным случаем сетевых моделей представления знаний. Формально *сетевые модели* задаются в виде [21]

$$H = < I, C_1, C_2, \ldots, C_n, Q >,$$

где I – множество информационных элементов, хранящихся в узлах сети; $C_1, C_2, \ldots, C_n$ – типы связей между информационными элементами; Q – отображение, которое устанавливает соответствие между множеством типов связей и множеством информационных элементов сети.

Сетевые модели представления знаний различаются между собой типами используемых связей (отношений). Если в сети используются иерархические связи (класс-подкласс, род-вид и т.п.), то сеть называется *классифицирующей*. Если связи между информационными элементами представляются функциональными отношениями, позволяющими вычислять значения одних информационных элементов по значениям других, то сети называют *функциональными* (вычислительными). Если в сети допускаются связи различного типа, то ее называют семантической сетью.

Семантическая сеть представляет собой направленный граф, в котором вершинам соответствуют объекты (сущности) предметной области, а дугам – отношения, в которых находятся эти объекты. Вывод в семантических сетях может выполняться на основе алгоритмов сопоставления, путем выделения подграфов с определенными свойствами.

К достоинствам семантических сетей относят: большую выразительную способность, наглядность графического представления, близость структуры сети к семантической структуре фраз естественного языка. Не-

достатком представления знаний в виде семантических сетей является отсутствие единой терминологии. Данная модель представления знаний находит различное воплощение у разных исследователей.

Фреймовые модели представления знаний используют теорию организации памяти, понимания и обучения, предложенную М. Минским. Фрейм (от англ. frame – рамка, каркас, остов) – структура данных, предназначенная для представления стереотипных ситуаций. Фрейм состоит из слотов (slot - гнездо, щелка, паз). Значением слота могут быть числа, выражения, тексты, программы, ссылки на другие фреймы. Совокупности фреймов образуют иерархические структуры, построенные по родовидовым признакам, что позволяет наследовать значения слотов. Такое свойство фреймов обеспечивает экономное размещение базы знаний в памяти. Кроме этого, значения слотов могут вычисляться с помощью различных процедур, т.е. фреймы комбинируют в себе декларативные и процедурные представления знаний. Фреймовые модели можно понимать как сетевые модели представления знаний, когда фрагмент сети представляется фреймом с соответствующими слотами и значениями. С фреймовыми моделями связаны модели представления знаний на основе *сценариев и объектов*.

Знания, которыми оперирует человек, часто имеют качественную природу, характеризуются неполнотой, неточностью, нечеткостью. Для формализации обработки знаний такого вида используются подходы, которые рассматриваются в параграфе 4.2.

Рассмотрим подробнее упомянутые выше модели представления знаний.

3.2. Логические модели

Логические модели представления знаний основаны на формализме языков математической логики. В настоящем параграфе сконцентрируем свое внимание на логике первого порядка, так как она является методологической основой многих моделей представления знаний в СИИ. Развитие навыков работы с логической символикой является весьма важным при анализе принципов работы существующих и разрабатываемых СИИ.

3.2.1. Формальные системы

В основе логических моделей представления знаний лежит понятие формальной системы. *Формальная система* (ФС) задается четверкой

$$M=\{T,P,A,R\},$$

где T – множество базовых элементов; P – множество синтаксических правил; A – множество аксиом; R – множество правил вывода. Выясним суть элементов, образующих ФС.

Множество T состоит из конечного или бесконечного числа элементов различной природы. Элементы множества T – алфавит ФС, на основе которого строятся все остальные составные части ФС. На множество T никаких ограничений не накладывается. Важно только, чтобы для T существовала процедура проверки принадлежности некоторого элемента множеству T.

Множество синтаксических правил P позволяет строить из элементов T синтаксически правильные совокупности базовых элементов. На множество синтаксических правил также не накладывается особых ограничений. Требуется только, чтобы существовала конструктивная процедура, которая позволяла бы за конечное число шагов дать однозначный ответ на вопрос, является ли данная совокупность элементов из T синтаксически правильной. Такие совокупности называют *правильно построенными формулами* (ППФ).

Среди всех ППФ выделяют некоторое подмножество аксиом A. При этом должна существовать процедура, позволяющая для любой ППФ решить вопрос, является ли она аксиомой данной ФС.

Наконец, множество R – это конечное множество отношений между ППФ, называемых *правилами вывода*. Правило вывода – это отношение на множестве формул. Если из формул $F_1, F_2, ...,F_n$ непосредственно выводится формула F, то это часто записывается в виде

$$\frac{F_1, F_2, ..., F_n}{F},$$

где формулы $F_1, F_2, ...,F_n$ называются *посылками правила*, а F – его *следствием (заключением)*.

Применяя правила вывода к множеству аксиом A, можно получать новые ППФ, к которым можно опять применить правила из множества R. Это позволяет осуществлять вывод новых ППФ. Множество R называют также *множеством семантических правил*. А множество ППФ, получаемое после применения правил к аксиомам, называется *множеством семантически правильных совокупностей.*

Выводом формулы B из формул $A_1,A_2,..,A_n$ называется последовательность ППФ $F_1, F_2, ...,F_m$ такая, что $F_m=B$, а для любого i ($1 \le i \le m$) формула F_i есть либо аксиома ФС, либо одна из исходных формул $A_1,A_2,..,A_n$, либо непосредственное следствие формул $F_1, F_2, ...,F_{i-1}$, полученное с помощью правил вывода. Некоторая ППФ F является *выводимой* в

ФС (является *теоремой* ФС), если существует вывод, в котором последней формулой является F. Сокращенно вывод F из $F_1, F_2, ..., F_n$ будем записывать в виде $F_1, F_2, ..., F_n \vdash F$. Если имеется эффективная процедура, позволяющая по данной ППФ F устанавливать, существует ли ее вывод в ФС, то данная ФС называется *разрешимой,* в противном случае – *неразрешимой.*

Притягательной стороной ФС с точки зрения представления знаний является то, что ее можно рассматривать как генератор новых знаний. В этом случае из множества аксиом A, представляющих собой знания, изначально хранящиеся в базе знаний СИИ, выводятся с помощью правил вывода производные знания [21]. Для этого машина вывода анализирует внутреннее представление знаний в памяти ЭВМ. Выделяемые в памяти ЭВМ цепочки символов, представляющие конкретные знания о предметной области (ПрО), называют *образами*, а цепочки символов, соответствующие посылкам правил – *образцами*. Если образ и образец сопоставимы, то в базу знаний добавляется новый элемент знаний – заключение правила.

Рассмотрим два класса ФС, широко используемых в системах искусственного интеллекта: исчисление высказываний и исчисление предикатов. Исчисление предикатов представляет собой развитие исчисления высказываний и включает его полностью как составную часть. Данные системы используют модель *дедуктивного вывода*, т.е. вывода, при котором из заданной системы посылок с помощью фиксированного набора правил формируются частные заключения.

3.2.2. Исчисление высказываний

Высказыванием называется предложение, содержание которого можно оценить как истинное или ложное. В естественных языках высказывания выражаются повествовательными предложениями. Например, “Сегодня ясная погода”, “Пять меньше трех” и т. п. Будем обозначать высказывания прописными латинскими буквами *A,B,C,..,X,Y,Z* и называть их *пропозициональными* (лат. propositio – предложение, высказывание). Пропозициональные буквы могут принимать два значения: истина (*И*) и ложь (*Л*). Значения *И* и *Л* называют *истинностными значениями*.

На основе заданных высказываний с помощью логических связок (союзов) образуются сложные высказывания. Указанные связки выражаются в естественном языке словами “и”, “или”, “если…, то…”, “неверно, что A” и называются: конъюнкция, дизъюнкция, импликация и отрицание. Для обозначения данных связок используются специальные символы, соответственно: $\wedge, \vee, \rightarrow, \overline{A}$ (или $\neg A$). Каждую логическую связку можно рас-

сматривать как операцию, которая из простых высказываний образует сложное высказывание.

Результаты применения логических операций к простым высказываниям определяются с помощью *таблиц истинности*. Таблица 3.1 определяет простейшие составные высказывания, образованные с помощью логических операций.

Таблица 3.1 – Таблица истинности простейших высказываний

A	B	$\overline{A}$	$A \wedge B$	$A \vee B$	$A \to B$	$A \leftrightarrow B$
Л	*Л*	*И*	*Л*	*Л*	*И*	*И*
Л	*И*	*И*	*Л*	*И*	*И*	*Л*
И	*Л*	*Л*	*Л*	*И*	*Л*	*Л*
И	*И*	*Л*	*И*	*И*	*И*	*И*

Здесь высказывание $A \to B$ читается "если *A,* то *B*" (другое обозначение $A \supset B$). Например, импликацией будет высказывание: "Если я пойду на стадион (*A*), то встречу друга (*B*)". В импликации первый элемент *A* называется *антецедентом* (лат. antecedens – предшествующий), а второй элемент *B* – *консеквентом* (consequens – последующий). Высказывание $A \leftrightarrow B$ называется *двойной импликацией (эквиваленцией)* и читается "если и только если *A*, то *B*". Это высказывание истинно тогда и только тогда, когда *A* и *B* имеют одно и то же истинностное значение. Эквиваленцией будет следующее высказывание: "Я пойду на стадион (*A*), тогда и только тогда, когда встречу друга (*B*)". Для обозначения эквиваленции употребляется также знак $\equiv$.

Сложные высказывания, составленные из исходных высказываний (обозначаемых пропозициональными буквами) и знаков логических операций, называют *формулой*. Так высказывание "Если 30 делится на 2 и на 3, то 30 делится на 6 " можно записать в виде формулы $A \wedge B \to C$. Данная формула будет соответствовать не только конкретному высказыванию, упомянутому выше, но и множеству всех других высказываний, имеющих подобную структуру. Поэтому исчисление высказываний не рассматривает конкретное содержание высказываний, а занимается анализом и синтезом формул и изучением отношений между формулами.

Множество базовых элементов *T* исчисления высказываний состоит из:

1) пропозициональных переменных – *A, B, C,…*;
2) логических констант – "истина"(*И*) и "ложь"(*Л*);
3) символов логических операций – $\wedge, \vee, \to, \leftrightarrow, \neg$;
4) скобок – ( ).

ППФ исчисления высказываний определяются с помощью следующих правил:

1) всякая пропозициональная буква есть ППФ;
2) логические константы *И* и *Л* – это ППФ;
3) если F_1 и F_2 ППФ, то $\overline{F}_1, F_1 \wedge F_2, F_1 \vee F_2, F_1 \to F_2, F_1 \leftrightarrow F_2$ – также ППФ;
4) других правил образования ППФ нет.

Правила 1) и 2) определяют *элементарные формулы* (*атомы*), а правило 3) указывает, как из элементарных формул строить новые формулы.

Пусть формула F состоит из атомов $A_1, A_2, \ldots, A_n$. Под *интерпретацией* формулы будем понимать приписывание атомам $A_1, A_2, \ldots, A_n$ истинностных значений. Если в формуле имеется n различных атомов, то можно задать 2^n интерпретаций формулы.

Среди формул исчисления высказываний выделяют формулы, которые истины во всех интерпретациях. Данные формулы называются *тавтологиями* или *общезначимыми* формулами. Примерами тавтологий являются: $A \vee \overline{A}$, $\overline{A \wedge \overline{A}}$. Чтобы в этом убедиться, необходимо составить таблицы истинности. Для указания того, что формула является тавтологией, используется знак $\models$. Например, $\models (A \to B) \wedge (B \to C) \wedge A \to C$.

Формула исчисления высказываний называется *противоречием*, если она принимает значение “ложь” во всех интерпретациях. Примером противоречия является $A \wedge \overline{A}$.

Тавтологии играют особую роль в исчислении высказываний. Различные подстановки в тавтологию, независимо от их конкретного содержания, всегда дают истинные высказывания. Поэтому тавтологии рассматривают как логически истинные схемы рассуждений, которые играют роль *законов* исчисления высказываний. Ясно, что всегда можно установить, является ли данная формула тавтологией, построив ее таблицу истинности. С другой стороны, формула не является тавтологией, если она принимает значение “ложь” хотя бы на одном наборе значений переменных. Это обстоятельство используется для распознавания тавтологий методом обратного рассуждения, заключающемся в поиске таких значений переменных, при которых формула оказывается ложной.

До сих пор вводились понятия, касающиеся только отдельных формул. В исчислении высказываний также рассматриваются *отношения* между двумя и более формулами. Определим отношения равносильности и логического следования.

Две формулы называют *равносильными* (эквивалентными), если они принимают одинаковые значения на всех наборах входящих в них переменных (интерпретациях). Для обозначения равносильности применяется знак $\Leftrightarrow$ (или знак тождества $\equiv$, или знак равенства =), например, $F_1 \Leftrightarrow F_2$.

Две формулы равносильны, если их двойная импликация (эквиваленция) $F_1 \leftrightarrow F_2$ – тавтология. Сокращенно это записывается в виде $\models F_1 \leftrightarrow F_2$. Например, следующая двойная импликация $(A \to B) \leftrightarrow (\overline{A} \vee B)$ имеет всегда истинное значение, т.е. $\models (A \to B) \leftrightarrow (\overline{A} \vee B)$. Следовательно, импликация может быть выражена через дизъюнкцию и отрицание:

$$(A \to B) \Leftrightarrow (\overline{A} \vee B).$$

Отношение равносильности позволяет выражать одни логические операции через другие. Например, эквивалентны следующие формулы (*законы де Моргана*):

$$\overline{A \vee B} \Leftrightarrow \overline{A} \wedge \overline{B},$$
$$\overline{A \wedge B} \Leftrightarrow \overline{A} \vee \overline{B}.$$

Отметим, что данные записи не являются формулами исчисления высказываний, так как знак $\Leftrightarrow$ не входит в множество *T*. Эти записи представляют собой высказывания о формулах логики, выраженные на метаязыке, на котором изучается логика.

Процесс дедуктивного вывода базируется на понятии логического следования. Формула *B* является *логическим следствием формулы A* (пишут $A \Rightarrow B$ или $A \models B$), если *B* истинна на всех наборах значений переменных (интерпретациях), на которых истинна *A*. Можно показать, что $A \Rightarrow B$, если и только если импликация $A \to B$ является тавтологией, т.е. $\models A \to B$. Иными словами, из истинности *A* всегда следует истинность *B*, но если *A* ложно, то относительно *B* ничего нельзя утверждать. Понятие логического следствия можно обобщить на совокупность высказываний

$$A_1 \wedge A_2 \wedge ... \wedge A_n \Rightarrow B,$$

т.е. *B* логически следует из истинных высказываний $A_1, A_2, ..., A_n$. Иными словами, *B* выводимо из $A_1, A_2, ..., A_n$, т.е. $A_1, A_2, ..., A_n \vdash B$.

Можно также показать, что формула *B* является логическим следствием последовательности формул $A_1, A_2, ..., A_n$ тогда и только тогда, когда формула

$$A_1 \wedge A_2 \wedge ... \wedge A_n \wedge \overline{B} \qquad (3.1)$$

является противоречием. Таким образом, доказательство того, что формула *B* является логическим следствием конечной последовательности формул $A_1, A_2, ..., A_n$, сводится к доказательству общезначимости формулы

$$A_1 \wedge A_2 \wedge ... \wedge A_n \rightarrow B \qquad (3.2)$$

или доказательству противоречивости формулы (3.1).

Для доказательства общезначимости необходимо показать, что в любой интерпретации I, в которой истинны $A_1, A_2, ..., A_n$, B также истинно. Если в формуле имеется n переменных, то необходимо будет построить таблицу истинности на 2^n входов, что приведет к экспоненциальному росту времени вычисления в зависимости от n. Поэтому на практике чаще применяют второй способ, соответствующий методу обратного рассуждения.

Задачей вывода является образование из некоторой совокупности исходных тавтологий новых формул, которые также являются тавтологиями. Существует бесконечное множество тавтологий, а значит и законов логики высказываний. Тавтологии формируют систему аксиом исчисления высказываний (множество A в определении ФС). В общем, не все общезначимые формулы могут быть выведены из произвольного множества тавтологий. В то же время доказано, что можно выбрать такую систему исходных тавтологий (аксиом исчисления высказываний), из которых выводимы все общезначимые формулы. Наиболее часто используют систему аксиом П.С. Новикова [25]:

1) $A \rightarrow (B \rightarrow A)$;
2) $(A \rightarrow (B \rightarrow C)) \rightarrow ((A \rightarrow B) \rightarrow (A \rightarrow C))$;
3) $(A \wedge B) \rightarrow A$;
4) $(A \wedge B) \rightarrow B$;
5) $(A \rightarrow (B \rightarrow (A \wedge B))$;
6) $A \rightarrow (A \vee B)$;
7) $B \rightarrow (A \vee B)$;
8) $(A \rightarrow C) \rightarrow ((B \rightarrow C) \rightarrow ((A \vee B) \rightarrow C))$;
9) $(A \rightarrow B) \rightarrow ((A \rightarrow \overline{B}) \rightarrow \overline{A})$;
10) $\overline{\overline{A} \rightarrow A}$.

Данная система аксиом обладает следующими важными свойствами: полнотой, непротиворечивостью и независимостью. *Теорема о полноте* исчисления высказываний утверждает, что если ППФ A общезначима, то она является выводимой. *Непротиворечивость* означает, что в исчислении высказываний не выводимы никакие две ППФ, одна из которых является отрицанием другой. И наконец, *независимость* системы аксиом означает, что ни одна аксиома не выводима из остальных аксиом.

Множество правил вывода R исчисления высказываний задается двумя правилами: правилом подстановки и правилом заключения.

Правило подстановки. Если Φ – выводимая формула (тавтология), содержащая букву A, то, заменив в ней всюду букву A на произвольную ППФ B, получим также выводимую формулу (тавтологию). Данное правило можно записать в виде:

$$\frac{\Phi(A)}{\Phi(B)}.$$

Правило заключения (modus ponens). Если A и $A \to B$ – выводимые формулы (тавтологии), то B – также тавтология. Данное правило записывается в виде:

$$\frac{A, A \to B}{B}.$$

Отметим, что данное правило вывода представляет собой некоторую схему, в которой символы A и B рассматриваются как метапеременные, допускающие замену другими ППФ.

В исчислении высказываний часто применяют дополнительные правила вывода, которые значительно сокращают путь доказательства теорем. Например, правило силлогизма:

$$\frac{(A \to B),(B \to C)}{A \to C}.$$

В ходе доказательства широко используется определенный класс ППФ, называемых *предложениями Хорна*. Предложения Хорна имеют вид:

$$A_1 \wedge A_2 \wedge \ldots \wedge A_n \to B.$$

Имеются две важные частные формы предложений Хорна:

1) если B ложно, то для общезначимого предложения Хорна верно $\overline{A}_1 \vee \overline{A}_2 \vee \ldots \vee \overline{A}_n$ (данное предложение содержит только отрицательные атомы);
2) когда n=1 и A_1=*true*, получаем предложение $true \to B$, которое эквивалентно атомарному истинному высказыванию B (данное предложение состоит только из одного положительного атома).

Не каждая БЗ может быть представлена предложениями Хорна. Но в тех случаях, когда это возможно, можно использовать простую процедуру вывода, основанную на применении правила modus ponens. Данная процедура должна применять modus ponens, пока не перестанут образовываться новые предложения.

Применение правил вывода для получения следствий опирается на свойство *монотонности* исчисления высказываний. Предположим, что в БЗ накоплено определенное множество истинных высказываний M_1 (формул), из которого выводимо высказывание (формула) A. Монотонность означает, что добавление в БЗ нового множества выводимых высказываний (формул) M_2 не повлияет на выводимость A, т.е. если $M_1 \vdash A$, то и $\{M_1 \cup M_2\} \vdash A$. Иными словами, при таком логическом выводе все истинные высказывания остаются истинными, а ложные – ложными. Вывод нового высказывания не требует возврата к предыдущим шагам вывода и изменения истинностных значений ранее полученных высказываний.

В заключение отметим, что исчисление высказываний – это разрешимая формальная система. Это следует из теоремы о полноте исчисления высказываний, которая утверждает, что в исчислении высказываний выводима любая общезначимая ППФ. Значит, если рассматриваемая формула общезначима, то она выводима в исчислении высказываний. Если это не так, то формула не принадлежит множеству семантически правильных формул.

3.2.3. Исчисление предикатов

Исчисление высказываний характеризуется ограниченными возможностями для представления знаний. Объясняется это тем, что в исчислении высказываний рассматриваются логические связи только между утверждениями, а сама структура утверждений не анализируется. Поэтому в исчислении высказываний нельзя установить связь между тем, о чем идет речь (объект) и тем, что сообщается о данном объекте (предикат). Исчисление предикатов рассматривает логические связи между различными элементами утверждений.

Предикатом называется неоднородная двузначная логическая функция от любого числа аргументов. Ее записывают в виде $P(x_1,x_2,..,x_n)$ и называют *n-местным предикатом*. Здесь аргументы $x_1,x_2,..,x_n$ – принадлежат одному и тому же или различным множествам, представляющим объекты предметной области. Аргументы $x_1,x_2,..,x_n$ называют *предметными переменными*, а их конкретные значения – *предметными постоянными*. Функциональную букву P называют предикатной буквой.

Предметные переменные и предметные постоянные называются *термами*. При подстановке вместо предметной переменной x_n константы a n-местный предикат $P(x_1,x_2,\ldots,a)$ от x_n уже не зависит. Если все переменные предиката заменить соответствующими предметными постоянными, то получается 0-местный предикат или высказывание.

Например, трехместный предикат $P(x_1,x_2,x_3)$ = "x_1 есть произведение x_2 на x_3" переходит в высказывание при постановке x_1=6, x_2=2, x_3=3.

В общем случае n-местный предикат $P(x_1,x_2,..,x_n)$ задает отношение между элементами $x_1,x_2,..,x_n$, которое означает, что " $x_1,x_2,..,x_n$ находятся между собой в отношении P". При n=1 предикат выражает свойство предметной переменной, например, "x – студент отличник ".

Вместо предметных переменных в предикаты могут подставляться n-местные функции $f(x_1,x_2,..,x_n)$, определенные на множестве объектов предметной области.

Множество базовых элементов (алфавит) исчисления предикатов включает:

1) знаки логических операций $\wedge,\vee,\rightarrow,\leftrightarrow,\neg$;
2) знаки кванторов $\forall,\exists$;
3) знаки пунктуации ( );
4) предметные переменные $x, y, z\ldots$;
5) предметные постоянные $a, b, c\ldots$;
6) функциональные символы $f, g, h\ldots$;
7) предикатные буквы P, Q, R, S, T, V, N.

Множество синтаксических правил, определяющих понятие ППФ исчисления предикатов, строится на основе понятий терм и элементарная формула. *Терм* определяется рекурсивно следующим образом:

a) каждая предметная переменная или предметная постоянная есть терм;
b) если $t_1, t_2, .., t_n$ – термы и f – n-местная функциональная буква, то $f(t_1, t_2, .., t_n)$ – терм;
c) никакие другие выражения термами не являются.

Элементарная формула (атом) вводится следующим образом: если P – предикатная буква, а $t_1, t_2, .., t_n$ – термы, то $P(t_1, t_2, .., t_n)$ – элементарная формула (атом). Частным случаем элементарной формулы является высказывание.

ППФ исчисления предикатов определяется следующим образом:

a) всякая элементарная формула есть формула;
b) если P и Q – формулы, и x – предметная переменная, то каждое из выражений

$$\overline{P}(x), P(x)\vee Q(x), P(x)\wedge Q(x), P(x)\rightarrow Q(x), \forall x P(x), \exists x P(x)$$

есть формула;

c) никакое другое выражение формулой не является.

Здесь знаки $\forall$ и $\exists$ обозначают соответственно кванторы существования и общности. *Кванторы* – это знаки, которые в сочетании с переменными дают возможность утверждать, что для всех или некоторых объектов

предметной области выполняется определенное свойство или отношение. Пусть *P(x)* – предикат, где предметная переменная определена на множестве *X*, т.е. $x \in X$. Утверждение, что все $x \in X$ обладают свойством *P*, записывается в виде формулы $\forall xP(x)$, которая читается "для всех *x*, *P* от *x*". Утверждение о том, что существует хотя бы один объект *x* предметной области *X*, обладающий свойством *P*, записывают в виде формулы $\exists xP(x)$ и читают: "существует такое *x*, что *P* от *x"*. Выражения $\forall xP(x)$ и $\exists xP(x)$ не зависят от переменной *x*. Приписывая квантор впереди предметной переменной, переходим к высказываниям. Например, если *P(x)* означает "*x* – четное число m", то $\forall xP(x)$ – ложное высказывание, а $\exists xP(x)$ – истинное высказывание.

Кванторы $\forall x$ и $\exists x$ связывают переменную в области своего действия. Поэтому переменная *x* в формулах $\forall xP(x)$ и $\exists xP(x)$ является *связанной*. Если область действия квантора распространяется на несколько предикатов, связанных логическими операциями, то используются скобки. Например, $\forall x \forall y(P(x,y) \to R(x))$.

Для преобразования многоместного предиката в высказывание нужно связать квантором каждую переменную. Если какая-либо переменная не связана квантором, то ее называют *свободной переменной*. Наличие свободной переменной говорит о том, что данная формула представляет не высказывание, а логическую функцию (предикат). Формула без свободных переменных называется *замкнутой формулой*.

Порядок следования одноименных кванторов не имеет значения, но разноименные кванторы переставлять нельзя. Так, $\forall x \exists y P(x,y)$ и $\exists y \forall x P(x,y)$ – разные высказывания.

Для кванторов справедливы следующие *законы де Моргана:*

$$\overline{\forall xP(x)} = \exists x\overline{P}(x),$$
$$\overline{\exists xP(X)} = \forall x\overline{P}(x).$$

Знак квантора общности соответствует перевернутой букве *A* и возник как сокращение английского слова *all* (все). Знак квантора существования соответствует перевернутой букве *E* (от англ. exists – существовать). Квантор общности можно понимать как обобщение конъюнкции, а квантор существования – как обобщение дизъюнкции. Так, если область определения предиката *P(x)* конечна и $X=\{a,b,c\}$, то высказывание $\forall xP(x)$ эквивалентно конъюнкции $P(a) \wedge P(b) \wedge P(c)$, а высказывание $\exists xP(x)$ – дизъюнкции $P(a) \vee P(b) \vee P(c)$. Когда предикатная формула определена на бесконечном множестве, то кванторы общности и существования можно рассматривать как "бесконечные" конъюнкции и дизъюнкции.

Формулы исчисления предикатов имеют смысл, если имеется какая-либо интерпретация входящих в них знаков. Один из способов *интерпретации* состоит в том, что задается непустое множество M объектов предметной области, называемое *областью интерпретации*, и определяется функция, сопоставляющая выражениям языка объекты, отношения между объектами из области интерпретации. Функция сопоставления I соотносит:

- предметной переменной x – элемент множества M, т.е. $I(x) \in M$;
- n-местной функциональной букве f в формуле – n-местную функцию на M;
- n-местной предикатной букве P в формуле – определение отношения между элементами на M.

На основе интерпретации определяется выполнимость и общезначимость формул.

Рассмотрим элементарную формулу $P(f(x_1,x_2),a)$. Пусть областью интерпретации является множество целых чисел и $f(x_1,x_2)=x_1^2 + x_2^2$, $a=9$, а $P(x,y)$ соответствует отношению $x>y$. Тогда рассматриваемая формула обозначает отношение $x_1^2 + x_2^2>9$, которое выполнимо для всех x_1 и x_2, если $|x_1|+|x_2|>5$. Для упрощения интерпретации предикатных символов их часто записывают словами, которые являются названиями определяемых отношений. Например, отец(x,y) – означает, что x приходится отцом y.

Формула называется *выполнимой* (*непротиворечивой*), если она истинна, по крайней мере, в одной интерпретации. Формула называется *опровержимой*, если она ложна хотя бы в одной интерпретации.

Важными понятиями исчисления предикатов являются понятия отношений равносильности и логического следования. Эти понятия определяются аналогично исчислению высказываний. Напомним их.

Две формулы P и Q называются *равносильными*, если формула $P \leftrightarrow Q$ является общезначимой, т.е. $P \Leftrightarrow Q$, если $\models P \leftrightarrow Q$. Например, $\overline{P \vee Q} \Leftrightarrow \overline{P} \wedge \overline{Q}$.

Формула Q *логически следует* из формул $P_1, P_2, ..., P_n$, тогда и только тогда, когда всякая интерпретация I, удовлетворяющая $P_1, P_2, ..., P_n$, удовлетворяет также и Q. Отношение логического следования $P_1, P_2, ..., P_n \Rightarrow Q$ (или $P_1,P_2,...,P_n \models Q$) соответствует общезначимости формулы $\models P_1 \wedge P_2 \wedge ... \wedge P_n \rightarrow Q$ или противоречивости формулы $P_1 \wedge P_2 \wedge ... \wedge P_n \wedge \overline{Q}$. Таким образом, выяснение логического следования сводится к установлению общезначимости или противоречивости определенных формул.

Для доказательства общезначимости формул исчисления предикатов используется аппарат логики исчисления высказываний, дополненный аксиомами и правилами вывода выражений, содержащих кванторы. Рассмат-

ривая исчисление предикатов как формальную систему, отметим, что к системе аксиом исчисления высказываний добавляются еще две аксиомы:

1) $\forall x P(x) \to P(t)$, где t – терм;
2) $P(t) \to \exists x P(x)$.

В этих аксиомах *P(x)* – любая формула, содержащая свободное вхождение аргумента *x*, причем ни одно из них не находится в области действия квантора по *t*. В этом случае говорят, что *P(x)* свободна для *t*. Формула *P(t)* получается из *P(x)* заменой всех свободных вхождений *x* на *t*.

Что касается правил вывода, то к правилам пропозициональной логики добавляются следующие правила, построенные на основе рассмотренных аксиом.

Правило универсальной конкретизации (УК)

$$\frac{\forall\ xP\ (\ x\ )}{P\ (\ t\ )},$$

где *P(t)* получается заменой в *P(x)* переменной *x* на *фундаментальный терм t* (константа или функция от фундаментального терма). Например, $\forall x$ предпочитает(*x*, мороженое), подставив вместо *x* константу “Иван”, получим: предпочитает(Иван, мороженое).

Правило экзистенциональной конкретизации (ЭК):

$$\frac{\exists\, xP\ (\ x\ )}{P\ (\ a\ )}.$$

Данное правило справедливо для любой формулы *P(x)* и предметной постоянной *a*, которой нет в базе знаний. Например, из $\forall x$ нравится(*x*, спорт) можно вывести нравится(спортсмен,спорт), если константа “спортсмен” отсутствует в базе знаний.

Правило экзистенциального обобщения (введение квантора $\exists$):

$$\frac{P\ (\ a\ )}{\exists xP\ (\ x\ )}.$$

Оно позволяет перейти от *P(a)* к $\exists x P(x)$, если *P(a)* истинно.

Правило универсального обобщения (правило введения квантора $\forall$):

$$\frac{Q \to P(x)}{Q \to \forall x P(x)}.$$

Если *P(x)* – следствие посылки *Q*, которая не имеет свободных вхождений *x*, то из нее выводится $\forall x P(x)$.

Применение рассмотренных правил вывода состоит в простом сопоставлении посылок правил с утверждениями, содержащимися в базе знаний, и добавлении в базу знаний получаемых заключений.

Рассмотрим пример. Пусть имеются формулы:

$$\exists x(P(x) \wedge \forall y(Q(y) \to R(x,y))), \tag{3.3}$$

$$\forall x(P(x) \to \forall y(S(y) \to \overline{R}(x,y))). \tag{3.4}$$

Требуется вывести следствие этих формул $\forall x(Q(x) \to \overline{S}(x))$. Процесс вывода изображен в виде диаграммы на рисунке 3.1.

Данную диаграмму можно рассматривать как схему поиска решения в БЗ. При этом начальное состояние БЗ соответствует формулам (3.2) и (3.3). Операторы, преобразующие одно состояние БЗ в другое, задаются множеством правил вывода. Условие окончания поиска представлено формулой $\forall x(Q(x) \to \overline{S}(x))$.

Процесс поиска состоит в сопоставлении посылок правил вывода с утверждениями, содержащимися в БЗ, и если сопоставление успешно, то соответствующее заключение добавляется в БЗ. Процесс повторяется, пока в БЗ не будет сформировано искомое заключение.

Безусловно, описанный процесс поиска является весьма схематичным и для полной автоматизации требует дальнейшей детализации. Вопросы автоматического поиска решений при доказательстве теорем рассматриваются в главе 4.

Отметим, что в исчислении предикатов доказательство выводимости формулы из совокупности посылок сводится к установлению общезначимости формулы (3.2) или противоречивости (3.1). Верно и обратное утверждение: в исчислении предикатов первого порядка всякая общезначимая формула является теоремой, т.е. для нее существует вывод, в котором эта формула является последней. Данное утверждение представляет собой *теорему о полноте* в широком смысле, которую в 1930г. предложил выдающийся немецкий математик К. Гёдель. Кроме понятия полноты в широком смысле, есть еще понятие полноты исчисления в узком смысле. Исчисление называется *полным в узком смысле*, если к системе его аксиом нельзя присоединить без противоречия не выводимую в нем формулу. Исчисление предикатов не является полным в узком смысле, так как к его системе аксиом можно добавить без противоречия недоказуемую в нем формулу $\exists x P(x) \to \forall x P(x)$ [8].

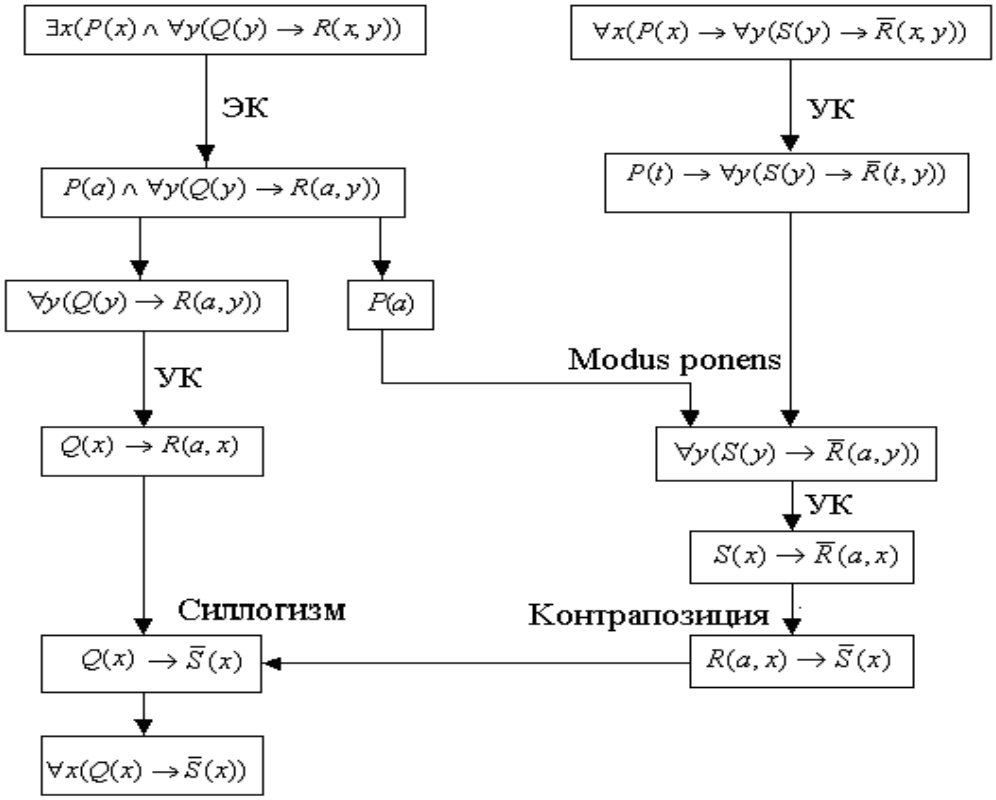


Рисунок 3.1 – Диаграмма вывода $\forall x(Q(x) \to \overline{S}(x))$

Исчисление предикатов – неразрешимая формальная система, т.е. не существует эффективной процедуры, позволяющей узнать по данной формуле, возможен ли ее вывод в исчислении предикатов или нет (теорема А. Черча, 1936). Формула выводима, если она общезначима. Следовательно, исчисление предикатов первого порядка неразрешимо относительно общезначимости. Иными словами, не существует общего метода для установления общезначимости любых формул исчисления предикатов первого порядка. Тем не менее, если, на самом деле, некоторая формула общезначима, то для нее существует процедура проверки общезначимости. Поэтому исчисление предикатов – *полуразрешимая* ФС.

3.2.4. Логики высокого порядка и псевдофизические логики

Логика предикатов первого порядка характеризуется весьма простыми синтаксическими правилами, которые определяют границу описательных возможностей такой логики. В частности, основными элементами исчисления предикатов первого порядка являются термы и предикаты. Термы представляют собой сущности описываемой предметной области, а предикаты выражают отношения между сущностями. При этом предикаты не могут представляться в виде переменных. Это не позволяет на языке ис-

числения предикатов записать некоторые выражения естественного языка. Например, “В каких отношениях находятся объекты x и y?” или “Два объекта x и y равны, если и только если все их свойства попарно совпадают”. Описание указанных предложений можно выполнить с помощью следующих формул: $\exists P P(x, y)$ и $\forall x \forall y \ ((x = y) \leftrightarrow (\forall P \ \ P(x) \leftrightarrow P(y)))$.

Данные формулы используют кванторы по предикатным буквам и поэтому не относятся к исчислению высказываний первого порядка. Если расширить синтаксические правила так, чтобы они допускали квантификацию по предикатным буквам, то расширенная формальная система будет включать в себя все предложения исчисления предикатов первого порядка. Расширенная формальная система называется *логикой второго порядка*. Такая логика разрешает использование вместо предметных переменных предложений исчисления предикатов первого порядка. Обычно исчисление предикатов второго и более высоких порядков называют *логикой высокого порядка.*

Безусловно, логика высокого порядка обладает большими выразительными возможностями. Однако вопросы построения эффективных процедур вывода в логике высокого порядка остаются открытыми.

Методы вывода логики первого порядка могут быть адаптированы к логике высокого порядка. Однако здесь нет прямого переноса. В логике высокого порядка разрешаются многократные возвраты к точкам выбора альтернативных подстановок (унификаторов), что приводит к большой разветвленности дерева вывода. Применение логики высокого порядка для автоматического доказательства теорем стало возможным благодаря разработке алгоритма унификации высокого порядка [70]. Известны соответствующие системы доказательства теорем логики высокого порядка, например, TPS [58], HOL [66]. Тем не менее, многие вопросы построения конструктивных процедур вывода для логики высокого порядка остаются открытыми.

Важным элементом логических моделей представления знаний являются *псевдофизические логики* [36]. Данные логики учитывают особенности восприятия окружающей действительности человеком. Выделяют следующие виды псевдофизических логик:

- временная логика – изучает взаимосвязь временных отношений;
- пространственная логика – изучает пространственные отношения;
- каузальная логика – изучает взаимосвязь отношений “причина-следствие”;
- логика действий – рассматривает отношения типа субъект-действие или действие-место.

Псевдофизические логики в качестве аксиом используют утверждения, вытекающие из особенностей восприятия человеком окружающей действительности, подтверждаемые результатами психологических экспериментов. Для установления ряда отношений в указанных логиках используются *метрические* и *топологические* шкалы. Метрические шкалы позволяют установить точные соотношения между положением фактов предметной области на абсолютной или относительной шкале. Топологические шкалы позволяют установить отношения нестрогого порядка между фактами, проецируемыми на такую шкалу, т.е. топологические шкалы связаны с качественными словесными оценками, активно используемыми человеком при описании различных ситуаций.

Система псевдофизических логик характеризуется ассоциативными связями между отдельными логиками. Например, существуют связи между пространственной и временной логикой, каузальной и логикой действий и др. Псевдофизические логики в СИИ играют основную роль при пополнении знаний и используют не только правила вывода характерные для дедуктивных систем, но и правила правдоподобных рассуждений.

3.3. Продукционные модели

3.3.1. Продукционные системы

Продукционная модель представления знаний является одной из самых распространенных. Представление знаний с помощью правил-продукций имеет в некоторых отношениях сходство с правилами вывода логических моделей. Это позволяет с помощью продукций выполнять эффективный вывод и, кроме того, благодаря естественной аналогии процессу рассуждений человека данные модели нагляднее отражают знания.

В системах продукций знания представляются с помощью наборов правил вида: “если A, то B“. Здесь A и B могут пониматься как “ситуация – действие”, “причина – следствие”, “условие – заключение” и т.п. Часто правило-продукцию записывают с использованием знака логического следования:

$$A \Rightarrow B .$$

Однако не следует отождествлять правило-продукцию и отношение логического следования. Дело в том, что интерпретация продукции зависит от того, что стоит слева и справа от знака логического следования. Часто под A понимается некоторая информационная структура (например, фрейм), а под B – некоторое действие, заключающееся в ее трансформации (преобразовании). Логическая интерпретация рассматриваемого выра-

жения накладывает жесткие ограничения на смысл символов *A* и *B*, которые должны рассматриваться как ППФ. Поэтому понятие продукции шире логического следования.

В качестве примера рассмотрим правило, взятое из базы знаний экспертной системы MYCIN [53], предназначенной для диагностики инфекционных заболеваний.

Если
1) место выделения культуры – кровь, и
2) реакция микроорганизма – грамотрицательная, и
3) форма микроорганизма – палочка, и
4) пациент относится к группе риска,
то
5) с уверенностью (0,6) название микроорганизма – pseudomonia aeruginosa.

Условие правила состоит из четырех фактов, соединенных союзом “и”. Если все четыре факта имеют место, то верным будет следствие правила. С каждым правилом связывается некоторое число, принимающее значение в диапазоне от -1 до 1, выражающее степень достоверности следствия и называемое коэффициентом уверенности. Это означает, что система работает с неточными и ненадежными фактами. Некоторые методы работы с неточными знаниями рассматриваются в параграфе 4.2 .

В базе знаний системы MYCIN факты представляются с помощью триплета: *объект–атрибут–значение*. Так, рассмотренное выше правило содержит пять фактов, которые можно представить в виде таблицы 3.1

Таблица 3.1 – Представление фактов

№ факта	Объект	Атрибут	Значение
1)	культура	место	кровь
2)	микроорганизм	реакция	грамотрицательная
3)	микроорганизм	форма	палочка
4)	пациент	принадлежит к группе риска	истина
5)	микроорганизм	название	pseudomonia aeruginosa

Если в процессе активизации правила первые четыре факта окажутся истинными, то в базу знаний будет помещен новый факт, представляемый триплетом объект–атрибут–значение. Одним из преимуществ хранения фактов в виде триплетов является повышение эффективности процедур поиска в базе знаний, так как появляется возможность упорядочения фактов по объектам и атрибутам.

Рассмотренная форма записи правила-продукции является упрощенной и представляет лишь ядро продукции. Во многих случаях правило-

продукцию записывают в обобщенной форме [21,4,5]

$$Rnj : (Pr, Bc, A \Rightarrow B, Ac), \tag{3.4}$$

где Rnj – идентификатор *j*-продукции в *n*-наборе продукций; Pr – приоритет правила продукции; Bc – предусловие применимости ядра продукции, представляющее предикат, при выполнении которого активизируется ядро продукции; Ac – постусловия продукции, определяющие действия и процедуры, которые необходимо выполнить после выполнения ядра продукции.

В общем случае продукционная система включает следующие компоненты:

- базу продукционных правил;
- базу данных (рабочую память);
- интерпретатор.

Множество продукционных правил образует базу правил, каждое из которых представляет обособленный фрагмент знаний о решаемой проблеме. Психологи называют такие фрагменты чанками (от англ. chunk). Считается, что *чанк* – это объективно существующая единица знаний, выделяемая человеком в процессе познания окружающего мира.

Предпосылка правила часто рассматривается как образец. *Образец* – это некоторая информационная структура, определяющая обобщенную ситуацию (условие, состояние и т.п.) окружающей действительности, при которой активизируется правило. Рабочая память (глобальная база данных) отражает конкретные ситуации (состояния, условия), возникающие во внешней среде. Информационная структура, представляющая конкретную ситуацию внешней среды в рабочей памяти, называется *образом.*

Интерпретатор реализует логический вывод. Процесс вывода является циклическим и называется *поиском по образцу*. Рассмотрим его в упрощенной форме. Текущее состояние моделируемой предметной области отражается в рабочей памяти в виде совокупности образов, каждый из которых представляется посредством фактов. Рабочая память инициализируется фактами, описывающими задачу. Затем выбираются те правила, для которых образцы, представляемые предпосылками правил, сопоставимы с образами в рабочей памяти. Данные правила образуют *конфликтное* множество. Все правила, входящие в конфликтное множество могут быть активизированы. В соответствии с выбранным механизмом разрешения конфликта активизируется одно из правил. Выполнение действия, содержащегося в заключении правила, приводит к изменению состояния рабочей памяти. В дальнейшем цикл управления выводом повторяется. Указанный процесс завершается, когда не окажется правил, предпосылки которых сопоставимы с образами рабочей памяти (рисунок 3.2).

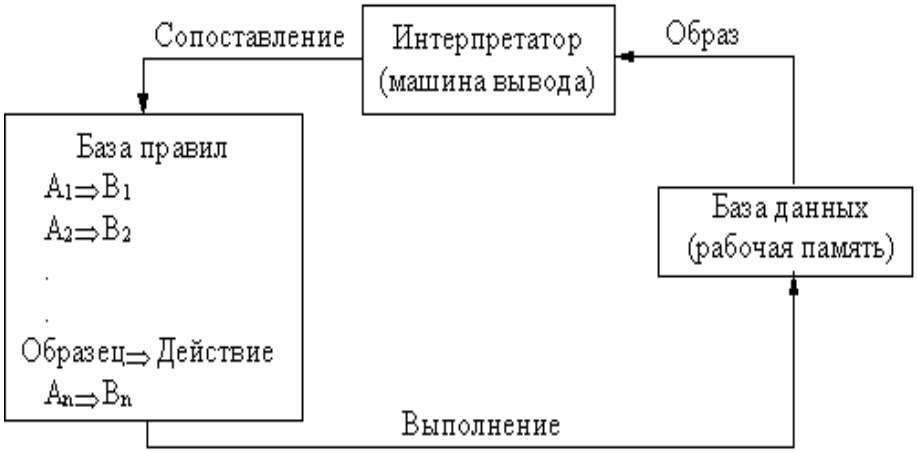


Рисунок 3.2 – Продукционная система

Таким образом, процесс вывода, основанный на поиске по образцу, состоит из четырех шагов:

- выбор образа;
- сопоставление образа с образцом и формирование конфликтного набора правил;
- разрешение конфликтов;
- выполнение правила.

Поясним процесс функционирования продукционной системы на простом примере сортировки строки, состоящей из букв *a*, *b* и *c* [77]. Правила сортировки представлены на рисунке 3.3. Если образец, заданный предпосылкой правила, сопоставим с частью сортируемой строки, то правило активизируется. В результате этого подстрока, которая совпала с условием правила, замещается подстрокой из заключительной части правила.

Рассматривая исходную и отсортированную строки как начальное и конечное состояние задачи, а продукционные правила как операторы, преобразующие одно состояние в другое, приходим к выводу, что поиск решения в продукционных системах соответствует поиску в пространстве состояний. На рисунке 3.3 изображено дерево состояний задачи. Здесь вершины дерева соответствуют промежуточным состояниям сортируемой строки, а ребра – правилам-продукциям.

Продукционные системы были предложены американским математиком Э. Постом (1943) и рассматривались как модель организации вычислительного процесса. При этом правило-продукция трактовалось как оператор замены одной цепочки литер в некотором слове на другую цепочку. Систему продукций можно рассматривать как формальную систему с алфавитом $A = \{ a_1, a_2, \ldots, a_n \}$, конечным множеством аксиом и конечным множеством правил вывода, имеющих вид

$$P_i : \alpha_i s \Rightarrow s\beta_i, i = 1,2,..., k ,$$

где α_i, β_i, s – слова алфавита *A*. Применение данного правила к некоторой цепочке литер, состоящей из слов α_i и s, означает вычеркивание α_i и приписывание β_i. Указанная замена соответствует подстановке, являющейся основным оператором нормальных алгоритмов Маркова [25] .

Множество продукций

1. $ba \Rightarrow ab$ 2. $ca \Rightarrow ac$ 3. $cb \Rightarrow bc$

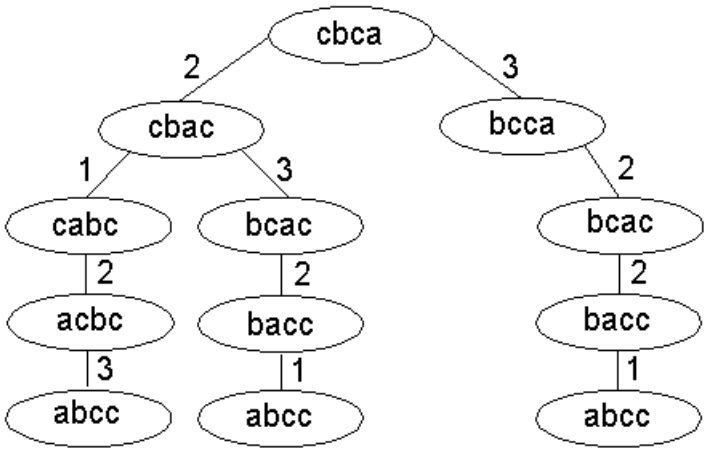


Рисунок 3.3 – Сортировка строки с помощью правил-продукций

Интересные результаты были получены А.Ньюэллом и Г.Саймоном при разработке системы GPS. Они установили, что правила-продукции соответствуют элементам знаний, накапливаемым в долговременной памяти человека. Такие элементы знаний активизируются, если возникает соответствующая ситуация (по образцу). Новые элементы знаний могут накапливаться в памяти без необходимости перезаписывания уже существующих элементов. Рабочая память продукционных систем аналогична кратковременной памяти человека, которая удерживает в центре внимания текущую ситуацию. Содержимое рабочей памяти, как правило, не сохраняется после решения задачи. Благодаря указанной аналогии, продукционные модели представления знаний получили широкое распространение в экспертных системах, моделирующих решение задач человеком-экспертом в той или иной области.

Широкое применение продукционных моделей определяется следующими основными достоинствами:

– универсальностью, практически любая область знаний может быть представлена в продукционной форме;

- модульностью, каждая продукция представляет собой элемент знаний о предметной области, удаление одних и добавление других продукций выполняется независимо;
- декларативностью, продукции определяют ситуации предметной области, а не механизм управления;
- естественностью процесса вывода заключений, который во многом аналогичен процессу рассуждений эксперта;
- асинхронностью и естественным параллелизмом, который делает их весьма перспективными для реализации на параллельных ЭВМ.

Однако продукционные модели не свободны от недостатков:

- процесс вывода имеет низкую эффективность, так как при большом числе продукций значительная часть времени затрачивается на непроизводительную проверку условий применения правил;
- проверка непротиворечивости системы продукций становится весьма сложной из-за недетерминированности выбора выполняемой продукции из конфликтного множества.

3.3.2. Управление выводом в продукционных системах

В ходе решения проблемы машина вывода (интерпретатор) выполняет две задачи – собственно логический вывод и управление выводом. Логический вывод в продукционных системах не отличается особой сложностью и реализуется на основе процедуры поиска по образцу, описанной выше.

Управление выводом в продукционных системах предполагает решение двух вопросов [53]:

1) с чего следует начинать процесс вывода;
2) как поступить, если на некотором шаге вывода возможен выбор различных вариантов его продолжения.

Ответ на первый вопрос приводит к прямой и обратной цепочке рассуждений, а на второй вопрос – к механизмам разрешения конфликтов в продукционных системах.

Как было отмечено выше, применение процедуры поиска по образцу в продукционных системах соответствует поиску решения задач в пространстве состояний или пространстве редукции задачи. Поэтому для управления выводам в продукционных системах могут применяться стратегии слепого и упорядоченного поиска, рассмотренные в главе 2.

В настоящем параграфе ограничимся рассмотрением стратегий прямого и обратного вывода (соответственно поиска, управляемого данными и целью), а также основных методов разрешения конфликтов.

Прямой и обратный вывод. Прямой вывод начинается с задания исходных данных решаемой задачи, которые фиксируются в виде фактов в рабочей памяти системы. Правила, применяемые к исходным данным, обеспечивают генерацию новых фактов, добавляемых в рабочую память. Процесс продолжается, пока не будет получено целевое состояние рабочей памяти.

На рисунке 3.4 представлен пример поиска, управляемого данными, на множестве продукций, записанных в виде формул пропозициональной логики. Здесь на каждом шаге работы машины вывода применяется простая стратегия разрешения конфликтов: активизируется последнее из успешно сопоставленных правил. Порядок сопоставления правил соответствует их номерам.

Множество правил-продукций

1) $G \wedge H \rightarrow C$ 2) $I \wedge K \rightarrow D$ 3) $L \wedge M \rightarrow E$ 4) $N \rightarrow F$ 5) $O \rightarrow F$
6) $C \rightarrow A$ 7) $D \rightarrow A$ 8) $E \rightarrow B$ 9) $F \rightarrow B$
10) $A \rightarrow goal$ 11) $B \rightarrow goal$

Исходные данные:
START={*L, M, N*}

№ шага	Рабочая память	Конфликтное множество	Активизируемое правило
0	*L, M, N*	-	-
1	*L, M, N*	3,4	4
2	*L, M, N, F*	3,9	9
3	*L, M, N, F, B*	3,11	11
4	*L, M, N, F, B,Goal*	3	Остановка

Рисунок 3.4 – Прямой вывод в продукционных системах

На рисунке 3.5,а изображен граф решения. Вершины графа соответствуют высказываниям, а ребра – соответствующим правилам ввода. Процесс вывода начинается с размещения множества исходных фактов *L, M* и *N* в рабочей памяти и заканчивается, когда в рабочую память будет помещена целевая вершина *Goal*. Направление вывода обозначено рядом с графом и соответствует движению от стартовой к целевой вершине, т.е. от исходных данных (высказывания *L,M* и *N*). Поэтому такой вывод называют выводом, управляемым данными.

Отметим, что выбранная стратегия активизации последнего правила, добавленного в конфликтное множество, соответствует поиску в глубину. Именно эта стратегия наиболее часто применяется в экспертных системах продукционного типа, так как она соответствует процессу рассуждений эксперта при решении той или иной задачи. Поведение системы для поль-

зователя выглядит довольно логичным: он видит, что система стремится свести каждую цель к подцели.

Если активизировать на каждом шаге работы машины вывода первое правило конфликтного множества, то граф решения будет строиться методом поиска в ширину (рисунок 3.5,б). В этом случае все альтернативные цепочки рассуждений, имеющиеся на графе решений, продолжаются параллельно, и ни одна не обгоняет другую.

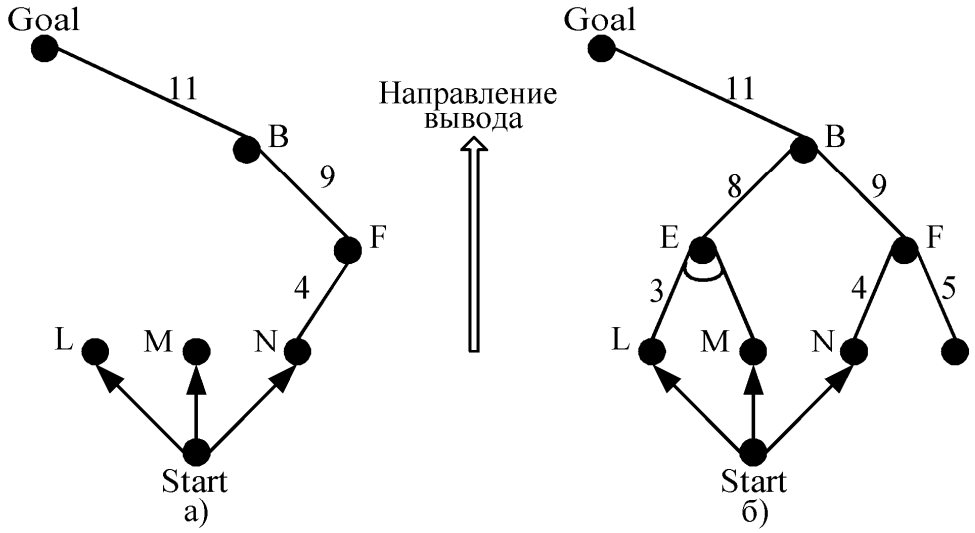


Рисунок 3.5 – Прямой вывод: поиск в глубину (а) и поиск в ширину (б).

Безусловно, возможны и смешанные стратегии вывода, когда объединяются оба вида поиска – поиск в ширину и поиск в глубину, а также различные эвристические стратегии, рассмотренные в главе 2. Поскольку эти стратегии в продукционных системах связаны с правилами разрешения конфликтов, то ниже мы рассмотрим их подробнее.

Кроме прямого вывода, в продукционных системах широко применяется и обратный вывод, т.е. вывод управляемый целевыми условиями. Такой вывод начинается с целевого утверждения, которое фиксируется в рабочей памяти. Затем отыскивается правило-продукция, заключение которого сопоставимо с целью. Условия данного правила помещаются в рабочую память и становятся новой подцелью. Процесс повторяется до тех пор, пока в рабочей памяти не будут найдены факты, подтверждающие целевое утверждение. Проиллюстрируем обратный вывод на множестве продукций предыдущего примера.

Процесс вывода начинается с того, что в рабочую память помещается целевое утверждение *Goal*, истинность которого необходимо подтвердить или опровергнуть, а также множество исходных фактов {*L, M, N*}, которые считаются истинными утверждениями.

Удобно рассматривать переменные правил как многозначные объекты, характеризуемые тремя возможными значениями: “не определено”, “ложь”, “истина”. В этом случае начальное значение переменной Goal – “не определено”, а факты *L, M,* и *N* имеют значение “истина”.

Обратная цепочка рассуждений для рассматриваемого примера изображена на рисунке 3.6 . Здесь факты, имеющие значение “истина”, выделены курсивом. Вывод начинается с утверждения Goal, которое рассматривается как текущая подцель. Все продукционные правила, заключение которых сопоставимо с текущей подцелью, добавляются в конфликтное множество правил. На каждом шаге активизируется первое правило конфликтного множества. Посылки данного правила добавляются в рабочую память и на следующем шаге выступают в качестве новой подцели системы. Такая стратегия разрешения конфликтов соответствует поиску в ширину.

№ шага	Рабочая память	Конфликтное множество	Активизируемое правило
0	Goal, ***L, M, N***	10,11	10
1	Goal, A, ***L, M, N***	11,6,7	11
2	Goal, A, B, ***L, M, N***	6,7,8,9	6
3	Goal, A, B, C, ***L, M, N***	7,8,9,1	7
4	Goal, A, B, C, D, ***L, M, N***	8,9,1,2	8
5	Goal, A, B, C, D, E, ***L, M, N***	9,1,2,3	9
6	Goal, A, B, C, D, E, F, ***L, M, N***	1,2,3,4,5	1
7	Goal, A, B, C, D, E, F,G, H, ***L, M, N***	2,3,4,5	2
8	Goal, A, B, C, D, E, F, G, H, I, K, ***L, M, N***	3,4,5	3
9	***Goal***, A, ***B***, C, D, ***E***, F, G, H, I, K, ***L, M, N***	4,5	Остановка

1) $G \wedge H \rightarrow C$ 4) $N \rightarrow F$ 7) $D \rightarrow A$ 10) $A \rightarrow goal$

2) $I \wedge K \rightarrow D$ 5) $O \rightarrow F$ 8) $E \rightarrow B$ 11) $B \rightarrow goal$

3) $L \wedge M \rightarrow E$ 6) $C \rightarrow A$ 9) $F \rightarrow B$

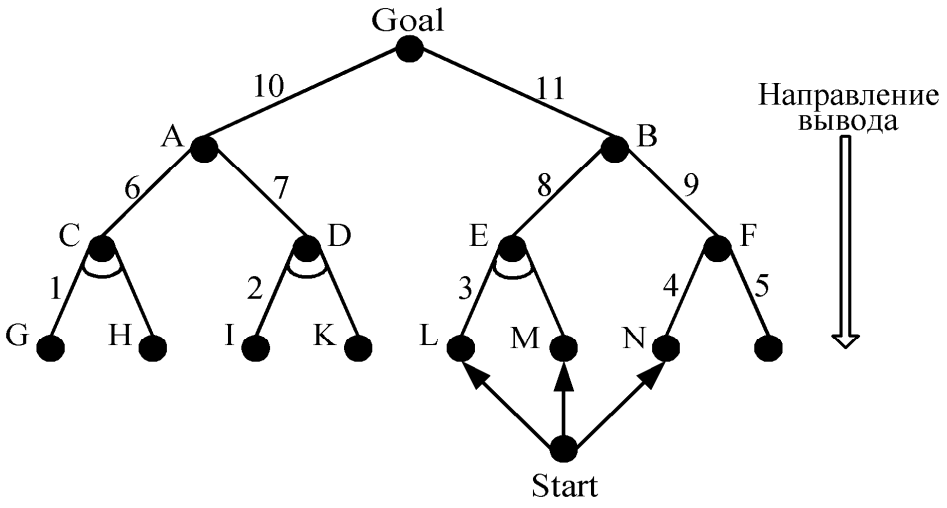


Рисунок 3.6 – Обратный вывод в продукционных системах

Указанный процесс повторяется до тех пор, пока все условия (посылки) некоторого правила не станут истинными, т.е. совпадут с фактами, имеющимися в рабочей памяти. В этом случае заключение правила тоже получает значение “истина”. Данное истинностное значение распространяется вверх по цепи активизированных правил. Если целевая вершина получает значение “истина”, то процесс поиска на этом заканчивается. Если этого не происходит, то вновь выбирается первое правило конфликтного множества и т.д.

На рисунке 3.6 при активизации правила 3 устанавливается, что его посылки *L* и *M* являются фактами. Следовательно, фактом является и заключение *E*. Далее, если верно *E*, то верно и *B* (правило 8). И наконец, если верно *B*, то справедливо и целевое утверждение *Goal* (правило 11).

Рассмотренные примеры показывают, что вывод в продукционных системах в общем случае соответствует поиску решений в И-ИЛИ графах. В частном случае, когда правила-продукции не содержат в условной части конъюнкций, вывод в продукционных системах будет соответствовать поиску решений в пространстве состояний. Поэтому рассмотренные ранее методы поиска решений в пространстве состояний и в И-ИЛИ графах успешно применяются в продукционных системах. Часто в продукционных системах применяют комбинированные методы поиска решений. Например, применяют двунаправленный поиск. В этом случае сначала поиск ведут в прямом направлении до тех пор, пока число состояний не станет большим. Затем выполняется обратный поиск уже достигнутых состояний (подцелей). При такой организации поиска имеется одна опасность. Если применяются методы эвристического поиска, то, возможно, что подграфы, обследуемые в прямом и обратном направлении, окажутся разными и не “встретятся”. Это приведет к увеличению времени поиска, по сравнению с однонаправленным поиском. Этой опасности нет, если применяются методы “слепого” поиска. Тогда комбинированный метод значительно сокращает время поиска [77].

Отметим, что ситуация, когда в рабочую память заранее вносятся все известные факты, встречается редко. Обычно в рабочей памяти изначально содержится только часть фактов, необходимых для вывода. Остальные сведения продукционная система выясняет сама, задавая вопросы пользователю, опрашивая измерительные или иные подсистемы, сопряженные с ней. Например, следуя цепочке обратных рассуждений, изображенной на рисунке 3.6, система направляет поиск от цели к фактам. Когда будет активизировано правило 1, то система пытается установить, являются ли высказывания *G* и *H* фактами, т.е. приписано ли им значение “истина”. Так как значения высказываний *G* и *H* неизвестны, то система может задать пользователю вопрос, относительно истинности высказываний *G* и *H*. Получен-

ный ответ вносится в рабочую память. Если высказывания *G* и *H* получат значения “истина”, то это изменит ход рассуждений системы и произойдет отказ от анализа других цепочек правил.

Сравнивая прямой и обратный вывод, следует отметить, что прямой вывод имеет более широкую область применения. Объясняется это тем, что обратный вывод будет эффективен в том случае, если решаемая проблема характеризуется немногими, четко определенными целевыми состояниями. Когда количество целевых состояний велико, или их определение является частью самой проблемы, то обратный вывод мало пригоден. Подобные ситуации встречаются, например, при решении задач планирования, где план является одновременно и целевым состоянием, и искомым решением, в медицинских экспертных системах при назначении методики лечения и др. [53].

Управление разрешением конфликтов. Как отмечалось выше, разрешение конфликтов – важная проблема, связанная с управлением порядком применения правил, образующих конфликтное множество.

Порядок активизации правил конфликтного множества определяется выбранной стратегией разрешения конфликтов. Ранее, в примерах, конфликтное множество правил представлялось в виде упорядоченного списка. При этом конфликтные правила дописывались в конец этого списка. Простые стратегии разрешения конфликтов основаны на том, что выбирается либо первое, либо последнее правило, входящие в список. Выбор первого правила соответствует поиску в ширину, а выбор последнего правила (т.е. только что добавленного) – поиску в глубину. Во многих продукционных системах чаще всего применяют второй способ.

Другими принципами, используемыми при разрешении конфликтов, являются [21]:

- принцип “стопки книг”;
- принцип наиболее длинного условия;
- принцип метапродукций;
- принцип приоритетного выбора.

Принцип “стопки книг” заключается в том, что список конфликтующих правил упорядочивается в соответствии с частотой использования продукций в прошлом. В первую очередь выбирается продукция, которая использовалась чаще всех.

Принцип наиболее длинного условия отдает приоритет той продукции, ядро которой имеет наиболее длинное условие. Такие продукции соответствуют специфичным (узким) ситуациям. Данный принцип опирается на тот факт, что продукции с длинными условиями учитывают больше информации о текущей ситуации, и это должно приводить к ускорению поиска решения.

Принцип метапродукций основан на введении в базу знаний специальных метаправил, упорядочивающих процесс разрешения конфликтов. На основании метаправил осуществляется анализ множества конфликтных правил и, в определенных ситуациях, активизируются те или иные правила из этого множества.

В случае приоритетного выбора с каждой продукцией связывается статический или динамический приоритет *Pr* (см. 3.4), определяющий порядок ее активизации.

Иногда в постусловиях продукции может указываться имя следующей продукции, которую необходимо выполнить. Это превращает систему продукций в обычный алгоритм.

3.4. Семантические сети

3.4.1. Общее понятие о семантических сетях

Семантические сети не являются однородным классом моделей представления знаний. Часто общей основой отнесения схемы представления знаний к семантической сети является то, что она представляется в виде направленного графа, вершины которого соответствуют объектам (понятиям, сущностям) предметной области, а дуги – отношениям (связям) между ними. И узлы, и дуги, как правило, имеют метки (имена). Имена вершин и дуг обычно совпадают с именами соответствующих объектов и отношений предметной области.

Объекты предметной области, отображаемые в семантической сети, можно условно разделить на три группы: обобщенные, индивидные (конкретные) и агрегатные объекты.

Обобщенный объект соответствует некоторой собирательной абстракции реально существующего объекта, процесса или явления предметной области. Например, “изделие”, “предприятие”, “сотрудник” и т.д. Обобщенные объекты фактически представляют определенные классы предметной области.

Индивидный объект – это каким-то образом выделенный единичный представитель (экземпляр) класса. Например, “сотрудник Петров И.Н.”

Агрегатным называется составной объект, образованный из других объектов, которые рассматриваются как его составные части. Например, изделие состоит из совокупности деталей, предприятие состоит из совокупности отделов, служб, цехов.

Введенная классификация объектов является относительной. В зависимости от решаемой задачи один и тот же объект может рассматриваться как обобщенный или индивидный, как агрегатный или неагрегатный.

Типы связей между объектами семантических сетей могут быть любыми. Но чаще всего применяются следующие основные связи (отношения): “род-вид”, “является представителем”, “является частью”. Наличие связи типа “род-вид” между обобщенными объектами *A* и *B* означает, что понятие *A* более общее, чем понятие *B*. Любой объект, отображаемый понятием *B*, отображается и понятием *A*, но не наоборот. Например, понятие “животное” – это родовое понятие для объекта “птица”. Все свойства родового объекта *A*, как правило, присущи и видовому объекту *B*. Иными словами, объект *B* наследует свойства объекта *A*.

Связь “является представителем” существует обычно между обобщенным и индивидным объектом, когда индивидный объект выступает в роли представителя некоторого класса. Так, индивидный объект “овчарка Альма” является представителем (экземпляром) обобщенного объекта “овчарка”. Экземпляр может быть представителем нескольких обобщенных объектов. В этом случае ему присущи свойства нескольких обобщенных объектов, что соответствует множественному наследованию.

В ряде случаев между связями “род-вид” и “является представителем” не делают различий, отмечая, что эти связи задают отношение “общее-частное” (рисунок 3.7). Иногда это приводит к недоразумениям. Поэтому для формализации таких связей будем использовать отношения ako (от англ. a-kind-of – разновидность) и is_a (от англ. is a member of the class – быть представителем класса) [94].

Не менее важно отношение “является частью” (англ. part of). Данное отношение связывает агрегатный объект с его составными частями. Оно позволяет отражать в базе знаний структуру объектов предметной области. Иногда данное отношение обозначают меткой “имеет” (англ. has).

В качестве примера на рисунке изображена семантическая сеть, представляющая часть знаний о животном мире.

Данная сеть была разработана А.М. Коллинзом и М.Р. Куиллианом (1969) и использовалась для моделирования механизмов памяти человека. Благодаря отношениям ako сеть представляет собой некоторую иерархическую структуру и позволяет отвечать на такие вопросы, как: “Является ли канарейка птицей?”, “Может ли канарейка летать?” и т.д. В ходе экспериментов с представленной моделью было установлено, что человек стремится запоминать информацию, соответствующую наиболее абстрактному уровню. Например, вместо того, чтобы непосредственно запоминать факт “канарейка умеет летать”, человек представляет этот факт в виде свойства и связывает его с понятием “птица”. Аналогично свойства: “дышать”, “принимать пищу”, “перемещаться” – связываются с понятием “животное”. Такой способ запоминания информации позволяет исключить её дублирование в базе знаний благодаря *наследованию* свойств в соответствии с

ako иерархией. Например, свойство “иметь крылья” присуще птицам, следовательно, оно присуще и канарейке.



Рисунок 3.7 – Мир животных в виде семантической сети

Впервые семантические сети появились в области машинной лингвистики как средство анализа смысла (семантики) естественного языка. В 1967 году М.Р. Куиллианом была разработана программа, фиксирующая смысл английских слов, построенная по принципу толкового словаря. В этой программе некоторое слово английского языка определяется в терминах, выражаемых другими словами. Каждое слово определяется множеством указателей на другие слова. Совокупность указателей образует сеть, “путешествуя” в которой можно выяснить смысл того или иного слова. В семантической сети Куиллиана вершины соответствуют определяемому понятию и снабжаются указателями на другие слова, раскрывающие данное понятие. База знаний организованна в виде страниц (плоскостей), на каждой из которых представляется граф, определяющий одно слово. На рисунке 3.8 изображены три различных определения английского слова “plant”. Указатели, обозначенные на рисунке 3.8 сплошными стрелками, формируют следующие определения слова “plant”:

1) живая (live) структура (structure), которая не относится к животным (animal), часто имеет листья (leaf), получает (get) питание (food) из (from) воздуха (air), воды (water) и почвы (earth);
2) оборудование (аппаратура – apparatus), используемое (use) для (for) реализации любого процесса (process) в (in) промышленности (industry);

3) помещать (put) нечто (seed – семя, plant – растение и т.п.) в почву (earth) для (for) выращивания (grow).

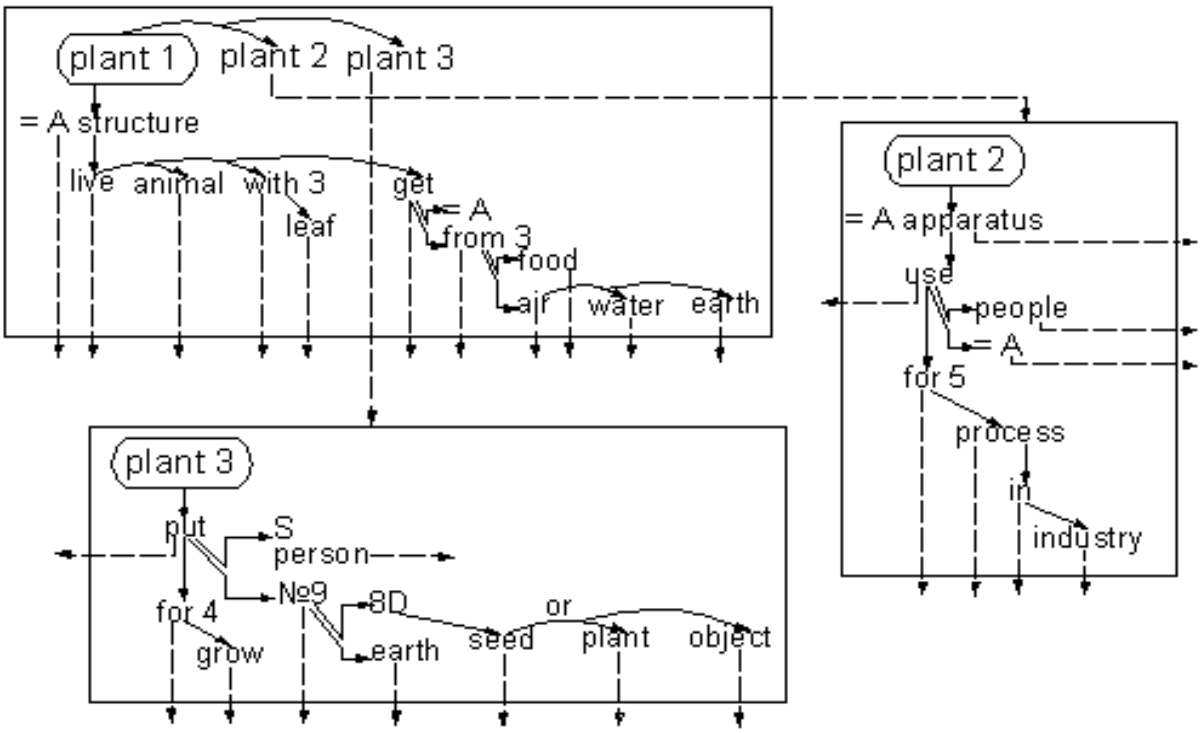


Рисунок 3.8 – Определение слова “plant”

Первое определение (plant 1) соответствует существительному “растение”, второе (plant 2) – существительному “завод”, третье (plant 3) – глаголу “садить”. Указатели, обозначенные на рисунке штриховыми линиями, ссылаются на слова-понятия, представленные на других страницах.

Один из механизмов вывода, применяемый в рассматриваемой модели семантической сети, основан на *поиске по пересечению*. Он позволяет устанавливать отношения между двумя словами. Поиск выполняется методом в ширину по внешним указателям из двух исходных страниц до тех пор, пока не будет найдено общее понятие, соответствующее вершине пересечения двух направлений поиска. Найденные в результате поиска пути от исходных вершин к вершине пересечения и будут представлять отношение между исходными словами-понятиями. Например, слова-понятия “cry” (плачь, плакать) и “comfort” (утешение, утешать) приводят к пересечению путей поиска в вершине “sad” (печальный, унылый). В результате анализа путей поиска модель позволяет сделать следующее заключение:

> “Плачь – это один из способов создания печальных звуков. Утешать – делать нечто менее печальным”.

Таким образом, модель позволяет формировать отношения, которые представлены в ней неявно и, следовательно, на ее основе можно получать

новые знания. Это позволяет строить системы понимания естественного языка, обладающие следующими возможностями [77]:

1) определять основной смысл текста путем поиска множества вершин пересечения;
2) осуществлять выбор необходимого значения многозначного слова на основе кратчайшего пути от этого слова к другим словам рассматриваемого предложения (своего рода семантическая метрика);
3) формировать ответы на различные запросы путем установления взаимосвязей между словами-понятиями запроса и словами- понятиями, хранящимися в памяти системы.

Рассмотренный подход получил некоторое развитие в системе TLC (Teachable Language Comprehender: обучаемая система понимания языка), разработанной Квиллианом. Однако практический успех TLC был ограничен. Это объяснялось тем, что набор используемых типов связок (отношений) между понятиями был недостаточен. Использовались только такие связки, как: “класс – экземпляр класса”, “атрибут – значение”. Кроме того, сам поиск не учитывал смысл связок. Представление знаний требует введения в рассмотрение более богатого набора отношений между понятиями (объектами, сущностями) предметной области. Дальнейшее развитие сетевых моделей представления знаний осуществлялось именно в этом направлении.

Работы Р.Симмонса и К.Филмора указали на важную роль глаголов при анализе смысла предложений естественного языка. Предложение можно представить вершиной-глаголом и различными падежными связками (отношениями). Такую структуру называют *падежной рамкой*. Среди *падежных отношений* выделяют следующие: *агент* – отношение между событием и тем кто (что) его совершает; *объект* – отношение между событием и тем, над чем выполняется действие; *инструмент* – объект, с помощью которого совершается действие; *место* – место совершения события; *время* – время совершения события. При анализе предложения программа находит глагол и устанавливает соответствующие падежные отношения между частями предложения. В качестве примера на рисунке 3.9 изображена падежная рамка предложения: “Иван закрепил деталь клеем”. Падежная рамка фиксирует знания падежной (лингвистической) структуры естественных языков в виде сетевого формализма.

В соответствии с введенным ранее определением семантической сети, отсутствуют какие-либо ограничения как на типы отношений, так и на типы объектов, отображаемых в сети. В большинстве случаев многообразие объектов сети можно подразделить на три группы [29,16]:

1) объекты-понятия – сведения о физических и абстрактных объектах, предметной области;

2) объекты-события – абстрактные или конкретные действия, которые могут привести к изменению состояния предметной области;
3) объекты-свойства – уточняют понятия и события, например, указывают характеристики понятий (цвет, форму, размеры и т.п.), фиксируют параметры событий (место, время, продолжительность).

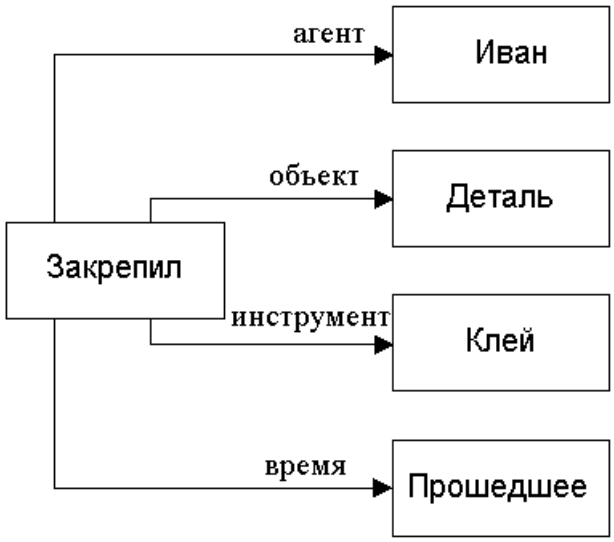


Рисунок 3.9 – Падежная рамка

Многообразие отношений, используемых в семантических сетях, подразделяется на следующие группы:

1) лингвистические отношения, которые, в свою очередь, подразделяются на падежные (агент, объект, инструмент, время, место), глагольные (наклонение, время, вид, число, залог) и атрибутивные (цвет, размер, форма и т.п.);
2) логические отношения (конъюнкция, дизъюнкция, отрицание, импликация);
3) теоретико-множественные отношения включают в себя отношения типа “множество-подмножество” (“род-вид”, “класс-подкласс”), “целое-часть”, “элемент множества” и др.
4) квантифицированные отношения, которые подразделяются на логические кванторы общности и существования, нечеткие кванторы (много, несколько, часто т.д.).

Таким образом, рассмотренные ранее отношения (“род-вид”, “быть представителем”, “быть частью”, падежные отношения) далеко не исчерпывают всего набора отношений, применяемых в семантических сетях. Но они образуют хорошую основу для построения прикладных баз знаний. Особое место при этом занимают теоретико-множественные отношения, обладающие транзитивными свойствами. Отношение *R* называется *транзитивным*, если для любых объектов α, β, γ таких, что α находится в от-

ношении R с β ($\alpha R\beta$) и β – в отношении R с γ ($\beta R\gamma$), следует справедливость утверждения "α находится в отношении R с γ", т.е.

$$(\alpha R\beta \wedge \beta R\gamma) \Rightarrow \alpha R\gamma .$$

Например, транзитивность родовидового отношения обеспечивает возможность наследования свойств от родового объекта к виду и подвиду. Механизм наследования свойств обеспечивает проведение простейших дедуктивных заключений типа: "Все люди смертны. Сократ – человек. Следовательно, Сократ смертен" [34].

В базах знаний выделяют интенсиональные и экстенсиональные знания. Если имеется конечное множество атрибутов $A=\{A_1,A_2,...,A_n\}$ и конечное множество отношений $R=\{R_1,R_2,...,R_m\}$, то *схемой* или *интенсионалом* отношения $R_i(i=1,2,...,m)$ называется набор пар вида

$$INT(R_i) = \{...,[A_j, DOM(A_j)],...\},$$

где $DOM(A_j)$ – домен атрибута A_j, т.е. множество возможных значений атрибута. *Экстенсионал* отношения R_i – это множество фактов

$$EXT(R_i) = \{F_1, F_2, ..., F_p\},$$

где $F_1,F_2,...,F_p$ – факты отношения R_i, обычно задаваемые в виде совокупности пар "атрибут-значение". Факт конкретизирует определенное отношение. В графической интерпретации он представляет собой подграф, имеющий звездообразную структуру. Корнем подграфа является вершина предикатного типа с меткой, соответствующей имени отношения. Ребра подграфа отмечены именами атрибутов.

Интенсиональная семантическая сеть представляет моделируемую предметную область на обобщенном, концептуальном уровне, а экстенсиональная – наполняет ее конкретными, фактическими данными. Таким образом, семантическую базу знаний можно рассматривать как совокупность объектов и отношений, часть из которых определена интенсионально, а часть – экстенсионально.

3.4.2. Способы описания семантических сетей и логический вывод

После знакомства с общими понятиями семантических сетей рассмотрим способы описания семантических сетей. Наиболее часто для этих целей используют концептуальные графы Дж. Соува и блочные структуры

Г.Хендрикса. Блочные структуры подробно описаны в [8,35]. Поэтому рассмотрим описание семантических сетей в виде концептуальных графов.

Вершинами концептуального графа являются либо объекты (понятия, сущности) предметной области, либо концептуальные отношения. Ребра концептуального графа связывают между собой вершины-понятия и вершины-отношения. При этом ребра могут исходить из вершины-понятия и заканчиваться в вершине-отношении, и наоборот. На рисунке 3.10 вершины “мяч” и “красный” соответствуют вершинам-понятиям, а вершина “цвет” – бинарному отношению. Для того чтобы различать указанные типы вершин, на концептуальных графах их изображают прямоугольником и эллипсом, соответственно.

Одно из преимуществ выделения отношений в самостоятельные вершины концептуального графа заключается в упрощении представления *n*-арных отношений. Такое отношение представляется вершиной-отношением с *n*-ребрами (рисунок 3.10).

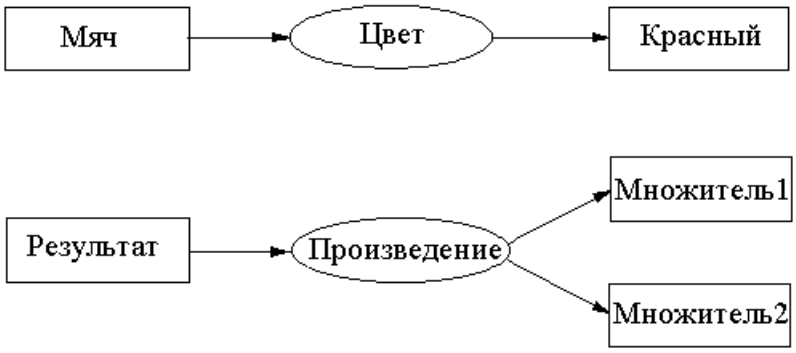


Рисунок 3.10 – Примеры концептуальных графов

Обычно каждый концептуальный граф фиксирует одно предложение. Тогда база знаний будет представляться в виде совокупности таких графов. Например, верхний граф на рисунке 3.10 соответствует предложению “Мяч имеет красный цвет”. Граф, изображенный на рисунке 3.11, отражает более сложное предложение: “Иван закрепил деталь стула клеем”.

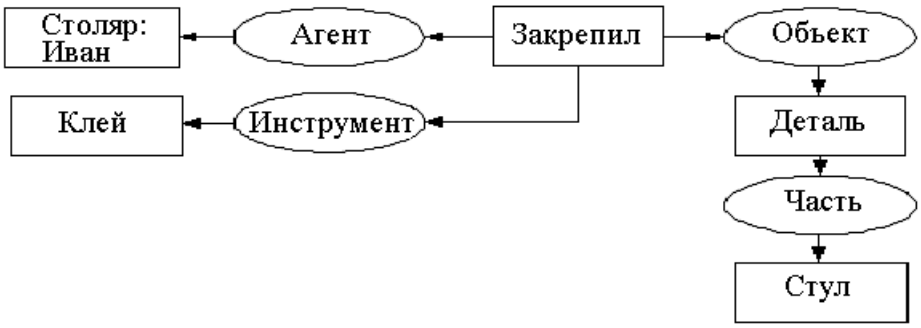


Рисунок 3.11 – Концептуальный граф предложения

Этот граф использует падежные отношения глагола “закрепить” и демонстрирует возможности концептуальных графов для представления предложений естественного языка.

Каждая вершина-понятие концептуального графа может иметь метку типа. Тип обозначает класс принадлежности вершины. На концептуальном графе метку типа вершины отделяют двоеточием от конкретного имени вершины. На рисунке 3.11 с помощью метки типа показано, что Иван является столяром. На концептуальных графах можно также вводить индивидные вершины-понятия с одинаковыми именами. Чтобы различать экземпляры объектов (понятий) с одинаковыми именами, используется специальный числовой маркер, перед которым записан символ #. Граф на рисунке 3.12 указывает, что конкретный "автомобиль" с маркером #1235 имеет зеленый цвет.

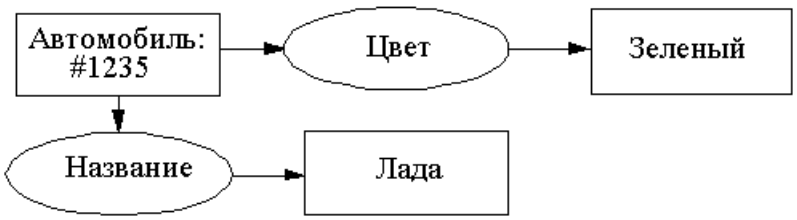


Рисунок 3.12 – Концептуальный граф с числовым маркером

Кроме числового маркера, на концептуальных графах могут применяться обобщенные маркеры, обозначаемые знаком *. В сочетании с переменной, записываемой после этого знака, обобщенный маркер оказывается полезным в ситуациях, когда две различные вершины графа представляют один и тот же объект. В отличие от числовых маркеров, которые применяются для выделения в семантической сети индивидных объектов, обобщенные маркеры позволяют выделять один и тот же обобщенный объект заданного типа (рисунок 3.13). В качестве примера на рисунке изображен концептуальный граф, соответствующий предложению: “Маляр испачкал свою руку краской”.

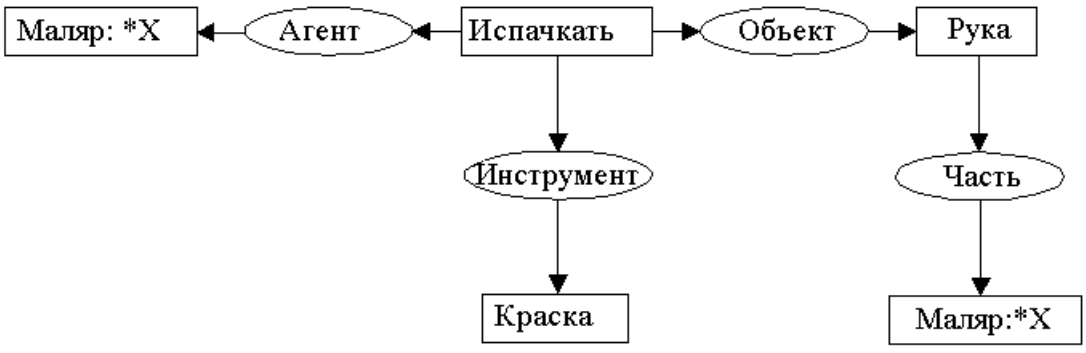


Рисунок 3.13 – Концептуальный граф с обобщенным маркером

На концептуальных графах вводятся операции, которые позволяют выполнять их преобразование. Новые графы получаются с помощью четырех операций: копирования, специализации, объединения и упрощения. Рассмотрим их подробнее.

Операция копирования позволяет сформировать из исходного графа G его копию, например, G_1.

Операция специализации заключается в замене обобщенной вершины ее индивидным (конкретным) вариантом. При этом может быть два случая:

1) если вершина отмечена обобщенным маркером, то обобщенный маркер заменяют конкретным индивидным маркером (именным или числовым).
2) тип вершины можно заменить соответствующими подтипами, обладающими необходимыми свойствами.

Операция объединения позволяет получать из двух графов один. Если на графе G_1 имеется вершина-понятие V_1, которая аналогична вершине-понятию V_2 графа G_2, то можно сформировать новый граф G_3, удалив V_2. При этом все связи V_2 необходимо переориентировать с V_2 на V_1.

Операция упрощения позволяет исключить на графах дубликаты отношений. При этом исключаются вершина-отношение и связанные с ней дуги. Дублирование отношений часто возникает в результате выполнения операции объединения.

Операция специализации позволяет сопоставлять две вершины концептуального графа и, если сопоставление успешно, выполнять объединение. Совместное использование операций специализации и объединения обеспечивает реализацию механизма наследования. Например, заменив обобщенный маркер индивидным, мы распространяем свойства типа на конкретный индивидный объект. Если, в ходе выполнения операции ограничения, происходит замена типа на его подтип, то имеет место наследование “класс-подкласс”.

Операции специализации и объединения конкретизируют исходный граф. Если граф G_1 – это конкретизация графа G_2, то можно сказать, что верно и обратное, т.е. G_2 – есть обобщение G_1. Обобщение широко используется при обучении СИИ, позволяя выводить правдоподобные умозаключения по частным примерам (см. §4.3).

Рассмотренные выше простейшие операции не гарантируют получения правильных утверждений на основе преобразованных графов. Конкретизация вершин-понятий, выполняемая в ходе операций специализации и объединения, не всегда соответствует действительности. Например, если в ходе выполнения операции объединения соединяются два графа, агенты которых обозначены одним и тем же понятием, то это вовсе не означает,

что не может случиться так, что эти понятия соответствуют двум различным индивидным объектам. Например, предложения “Собака грызет кость” и “Собака находится в парке” можно объединить в одно, если речь идет об одном и том же животном, а не о двух разных животных. Тем не менее, рассмотренные операции не приводят к утверждениям, лишенным смысла, и позволяют выполнять правдоподобные рассуждения.

Концептуальные графы могут содержать вершины, которые являются предложениями. Такие вершины изображаются в виде прямоугольного блока, содержащего подграф, соответствующий предложению. В качестве примера на рисунке 3.14 изображен граф утверждения: “Алексей полагает, что Татьяне нравится игрушка”.

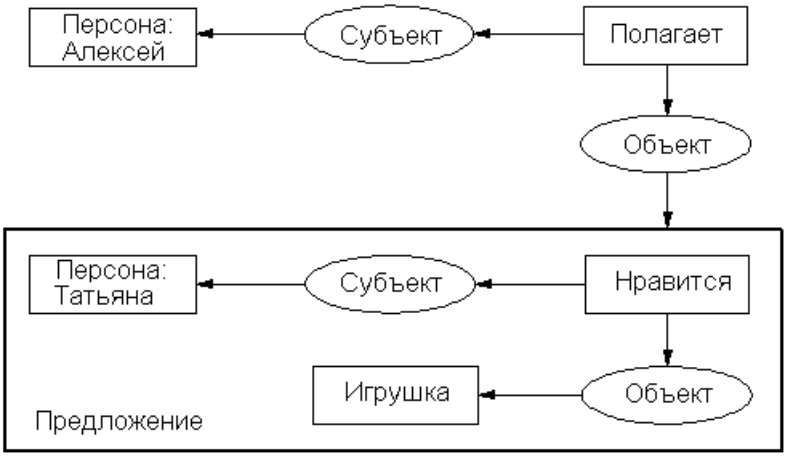


Рисунок 3.14 – Концептуальный граф, содержащий предложение

Введение отдельных блоков для обозначения предложений позволяет фиксировать отношения между предложениями. В приведенном примере глагол “полагает” представляет не конкретное физическое действие, а выражает ощущения Алексея. Поэтому на графе введено новое отношение “субъект”, представляющее лицо, испытывающее те или иные ощущения. Утверждения, содержащие подобные лексические единицы (доверять, верить, намериваться и т.п.) являются объектом исследований *модальной* логики.

Концептуальный граф позволяет естественным образом выражать конъюнкцию. Например, с помощью концептуального графа мы легко можем представить предложение “Собака находилась в саду и грызла кость”. Сложнее обстоит дело с другими логическими операциями: дизъюнкцией, логическим отрицанием, кванторами.

Для представления отрицания на концептуальных графах применяется унарный логический оператор “не”. Оператор применяют к предложению. Утверждение, фиксируемое предложением, считается в этом случае ложным. Граф, изображенный на рисунке 3.15, соответствует предложению: “Нет игрушек, которые нравятся Татьяне”.

Используя отрицание и конъюнкцию, можно представить на концептуальных графах дизъюнкцию. Однако для упрощения концептуальных схем дизъюнкцию представляют в виде специального бинарного отношения “или”, аргументами которого являются вершины-предложения.

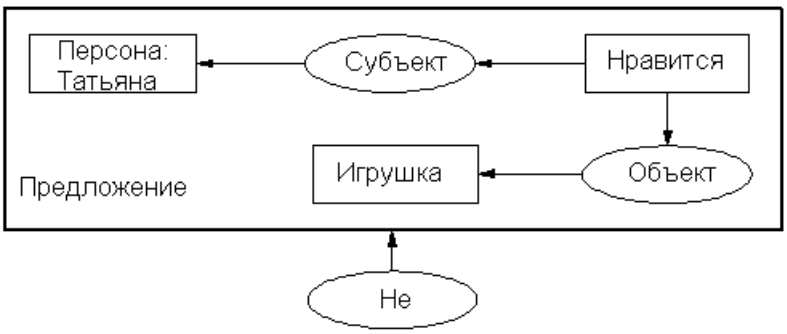


Рисунок 3.15 – Концептуальный граф с логическим отрицанием

Обобщенные объекты-понятия, отображаемые на концептуальных графах, связаны квантором существования. Так, граф, изображенный на рисунке 3.10, можно описать формулой

$$\exists x \exists y (\textit{мяч}\ (x) \wedge \textit{цвет}\ (x, y) \wedge \textit{красный}\ \ (y)).$$

Квантор общности получается при отрицании предложений, связанных квантором существования. Например, применив отрицание к графу, изображенному на рисунке 3.10, получим

$$\forall x \forall y \neg (\textit{мяч}\ (x) \wedge \textit{цвет}\ (x, y) \wedge \textit{красный}\ \ (y)).$$

По своим выразительным возможностям концептуальные графы эквивалентны логике исчисления предикатов. Существует возможность взаимного перехода от описаний предметной области на языке исчисления предикатов к концептуальным графам, и наоборот. Выбор тех или иных средств определяется характером решаемых задач и предпочтениями разработчиков СИИ.

Методы вывода на семантических сетях используют ассоциативные и сопоставляющие алгоритмы, которые сводятся к нахождению путей на графе, построению транзитивных замыканий, выделению подграфов с определенными свойствами. Основные подходы к построению процедур вывода на семантических сетях изложены в [8,29,34]. Во многих случаях они базируются на простых операциях над графами, как-то: удаление и добавление новых вершин и ребер, специализация, поиск вершины или ребра по имени, переход от одной вершины к другой по связям, объединение подграфов и др.

Одним из простых средств вывода на семантических сетях является

поиск по пересечению, который был рассмотрен ранее.

Другим мощным средством вывода является *сопоставление с образцом*. В данном случае происходит сопоставление отдельных фрагментов семантической сети. При этом запрос к базе знаний представляется в виде автономного подграфа, который строится по тем же правилам, что и семантическая сеть. Поиск ответа на запрос реализуется сопоставлением подграфа запроса с фрагментами семантической сети. Для этого осуществляют наложение подграфа запроса на соответствующий фрагмент сети. Успешным будет то наложение, в результате которого фрагмент сети оказывается идентичным подграфу запроса. При этом допускается использование в запросе переменных. Переменная запроса сопоставляется с константой фрагмента сети. Пусть, например, имеется запрос: “Существует ли такая птица X, которая умеет петь и имеет желтый цвет?” (рисунок 3.16).

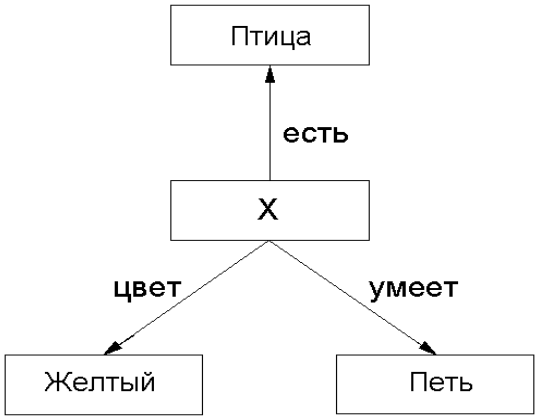


Рисунок 3.16 – Подграф запроса

Изоморфное вложение подграфа запроса в семантическую сеть, изображенную на рисунке 3.7, позволяет дать ответ X=“канарейка”.

В общем случае в подграфе запроса могут быть заданы объекты, атрибуты, имена отношений, не представленные в базе знаний явно. Это требует выполнения предварительных преобразований фрагментов сопоставляемых семантических сетей. В результате таких преобразований могут быть получены новые связи. Указанные преобразования выполняются на основе простейших базовых операций рассмотренных ранее.

Методы вывода на семантических сетях, использующие идею пересечения путей или сопоставления фрагментов сетей, имеют существенный недостаток. Он связан с комбинаторным ростом числа сопоставлений или пересечений в сетях достаточно большой размерности.

В ряде систем, основанных на семантических сетях, используются специализированные правила вывода. В [8] рассматриваются методы дедуктивного вывода на семантических сетях, основанные на понятиях раскраски графов и операторах преобразования сетей.

3.5. Фреймы

3.5.1. Структура фрейма

Фрейм можно рассматривать как фрагмент семантической сети, предназначенный для описания некоторого объекта (события, ситуации) предметной области со всей совокупностью присущих ему (ей) свойств. В этом смысле фреймы – это подграфы семантических сетей. Они представляют блоки знаний, каждым из которых можно манипулировать, как единым целым, что позволяет лучше структурировать базу знаний.

Фреймы впервые были предложены для представления знаний в 1975г. М. Минским. Согласно его определению *фреймы* – это структуры данных, предназначенные для представления стереотипных ситуаций. Когда человек оказывается в новой ситуации, он извлекает из памяти ранее накопленные блоки знаний, имеющие отношение к текущей ситуации, и пытается применить их. Эти блоки знаний и представляют собой фреймы. Вероятно, знания человека организованны в виде сети фреймов, отражающих его прошлый опыт. Например, мы легко можем представить типовой одноместный номер в гостинице. Обычно он имеет кровать, ванную комнату, шкаф для одежды, телефон, план эвакуации в случае пожара и т.п. Детали каждого конкретного номера могут отличаться от приведенного описания. Но они легко уточняются, когда человек оказывается в конкретном номере: цвет обоев и занавесок, положение выключателей света и др.

В простейшем случае под фреймом понимают следующую структуру

$$\{n,(ns_1,vs_1,ps_1),(ns_2,vs_2,ps_2),...,(ns_k,vs_k,ps_k)\},$$

где n – имя фрейма, ns_1 – имя слота, vs_1 – значение слота, ps_1 – имя присоединенной процедуры. Фрейм состоит из слотов. *Слоты* – это некоторые незаполненные подструктуры фрейма. После заполнения слотов конкретными данными, фрейм будет представлять ту или иную ситуацию, явление или объект предметной области. Процедура является необязательным элементом слота. В качестве значений слотов могут выступать имена других фреймов, что обеспечивает построение сети фреймов. В качестве примера на рисунке 3.17 изображен фрагмент сети фреймов, представляющих типовой одноместный номер в гостинице. Здесь “мебель”, “инвентарь”, “санузел”, “оплата” – это имена слотов. Значениями слотов “мебель”, “инвентарь”, “санузел” является список имен других фреймов, которые соответственно описывают предметы и оборудование, находящиеся в номере.

Во фреймовых системах часть фреймов представляет индивидные объекты предметной области. Такие фреймы называют *экземплярами*

фреймов или *фреймами-примерами*. Другие фреймы, представляющие обобщенные объекты, называют *классами* или *фреймами-прототипами*. Фрейм-прототип соответствует интенсиональному описанию множества фреймов-примеров.

Фрейм может содержать специальный слот, с помощью которого задается отношение данного фрейма с другими фреймам сети. Наиболее часто в сети фреймов используют теоретико-множественные отношения, которые позволяют строить иерархические структуры. Специальный слот, называемый is_a слотом, фиксирует отношение “экземпляр класса”. Слот ako указывает, что данный фрейм является прямым *подклассом* вышестоящего по иерархии фрейма. Вышестоящий фрейм называют родительским фреймом или *суперклассом*.

Например, кровать “Эмилия”, установленная в гостиничном номере, является конкретизацией обобщенного объекта “кровать” (рисунок 3.17). Это обозначено с помощью слота is_a, указывающего, что кровать “Эмилия” – экземпляр класса (фрейма-прототипа) “кровать”. Слот ako указывает, что одноместные номера являются подклассом гостиничных номеров.

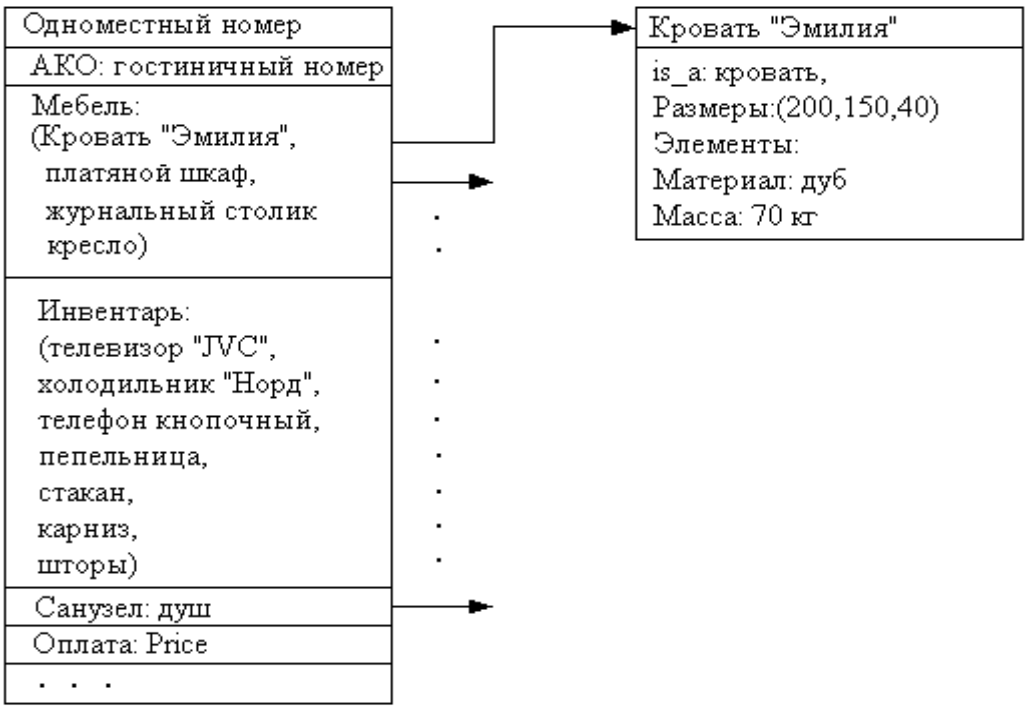


Рисунок 3.17 – Фрейм, представляющий гостиничный номер

Важной характеристикой фреймов является возможность включения в слоты присоединенных процедур. Выделяют два вида процедур: процедуры-демоны и процедуры-слуги. *Процедуры-демоны* активизируются автоматически при каждой попытке добавления или удаления данных из слота. Чаще всего используются три типа процедур-демонов: “если-добавлено”, “если-удалено” и “если-нужно”. Процедура “*если-добавлено*”

выполняется, когда новые данные помещаются в слот. Процедура "*если-удалено*" выполняется, когда данные удаляются из слота. Процедура "*если-нужно*" автоматически запускается, когда запрашивается информация из слота, а с ним не связаны никакие значения. С помощью указанных процедур обычно выполняются все рутинные операции, обеспечивающие поддержание базы знаний в актуальном состоянии. Процедуры-слуги активизируются только по запросу при обращении к слоту. Например, при обращении к слоту "оплата" будет активизирована процедура "Price", которая определит стоимость проживания в номере за сутки.

Фреймовые системы представляют собой иерархически организованные структуры, реализующие принцип наследования информации. Наследование значений происходит в направлении "суперкласс – подкласс", "класс – экземпляр класса". Например, кровать "Эмилия" является экземпляром класса "кровать". Поэтому кровать "Эмилия" наследует все свойства своего суперкласса. В частности, значение слота "элементы" может быть унаследовано от суперкласса.

В ряде фреймовых систем в состав слотов включают различные указатели наследования [37]. Указатель наследования фиксирует ту информацию, которая передается от слотов фрейма верхнего уровня к соответствующим слотам фрейма нижнего уровня. Рассмотрим в качестве примера структуру фрейма системы FMS [37]. Здесь выделяют следующие типы указателей наследования: unique, same, range, override и др.

Unique (уникальный) означает, что слоты с одними и теми же именами во фреймах, находящихся на разных уровнях иерархии, могут иметь различные значения. Same (такой же) показывает, что слоты должны иметь одинаковые значения. Range (диапазон) устанавливает некоторые границы, в которых может находится значение слота фрейма нижнего уровня. Override комбинирует свойства указателей unique и same. При описании слотов фрейма в системе FMS указывается также тип данных слота. Типы данных слотов соответствуют типам данных, применяемых в языках программирования высокого уровня. Например, INTEGER (целый), REAL (вещественный), BOOL (логический), STRING (строковый). Однако применяются также и другие типы данных, повышающие эффективность работы с системой. Например, TEXT (текстовый), TABLE (табличный), LIST (списочный), LISP (присоединенная процедура), FRAME (указатель на другой фрейм) и др.

Обобщая сказанное, фреймовую структуру данных системы FMS можно упрощенно описать в виде формул:

<фрейм>::=<имя фрейма> <ссылка на суперкласс> <слот> {<слот>}

<слот>::=<имя> <указатель наследования> <тип данных>
<значение> [<процедуры-демоны>]

<указатель наследования>::= unique| same| range| override

<тип данных>::= INTEGER| REAL| BOOL| STRING| TEXT| TABLE|
LISP| LIST| FRAME

<процедуры-демоны>::=<если-добавлено>| <если-удалено>|
<если-нужно>|

Другим примером фреймовой системы является язык FRL (**F**rame **R**epresentation **L**anguage), представляющий расширение языка Лисп. Объекты предметной области в языке FRL называются фреймами, а их свойства – слотами. Слот содержит несколько "фацет" (англ. facet – аспект, грань, сторона), характеризуемых значением. Так, предложение "Лошадь серого цвета" можно представить на языке FRL в виде: frame Лошадь, slot цвет, facet $value, value серый. В состав слота входит шесть фацет:

$value – текущее значение слота;
$require – допустимые значения фацета value;
$default – значение по умолчанию для фацета value;
$if-added – процедура-демон "если-добавлено";
$if-removed – процедура-демон "если-удалено";
$if-needed – процедура-демон "если-нужно".

Каждый фрейм, создаваемый на языке FRL, содержит предварительно определенный слот ako, позволяющий наследовать значения слотов в соответствии с иерархией фреймов.

В моделях представления знаний фреймами объединяются преимущества декларативного и процедурного подходов представления знаний. Фреймовый подход широко используется на практике, о чем свидетельствует наличие развитых фреймовых языков, например, GUS, KRL, CONSUL и др. Однако во фреймовых системах отсутствует специальный механизм управления выводом. Поэтому вывод обычно реализуется с помощью процедур, встроенных во фреймы.

3.5.2. Управление выводом

Во фреймовых системах используется три способа управления выводом [37]: с помощью механизма наследования; с помощью процедур-демонов; с помощью присоединенных процедур.

Механизм наследования является основным встроенным средством вывода, которым оснащаются фреймовые системы. Он обеспечивает значительную экономию памяти и автоматическое определение значений для слотов фреймов нижних уровней.

Фреймовые системы также оснащаются набором специальных процедур, к которым относят процедуры: конструирования класса, конструирования экземпляра класса, записи значения в слот, чтения слота. Процедура конструирования класса формирует фрейм-прототип с необходимым набором слотов и соответствующими ссылками на суперклассы. Фрейм может быть связан с несколькими суперклассами. Процедура конструирования экземпляра класса позволяет формировать фреймы-примеры. Она автоматически устанавливает связь всех таких фреймов с соответствующим классом посредством is_a слота. Процедуры записи и чтения значений слотов осуществляют доступ к слотам соответствующих фреймов и позволяют пользователю ввести или определить значение соответствующего слота. Для этого, при их вызове, им передаётся имя фрейма и имя соответствующего слота. В случае если значение какого-либо слота при вызове соответствующей процедуры конструирования не задается, то автоматически вызывается процедура, позволяющая установить значения слота по умолчанию. Это часто выполняется с помощью механизма наследования. Рассмотрим некоторые особенности функционирования механизма наследования.

На рисунке 3.18 изображена простейшая иерархическая структура, в которой каждый фрейм имеет только один суперкласс.

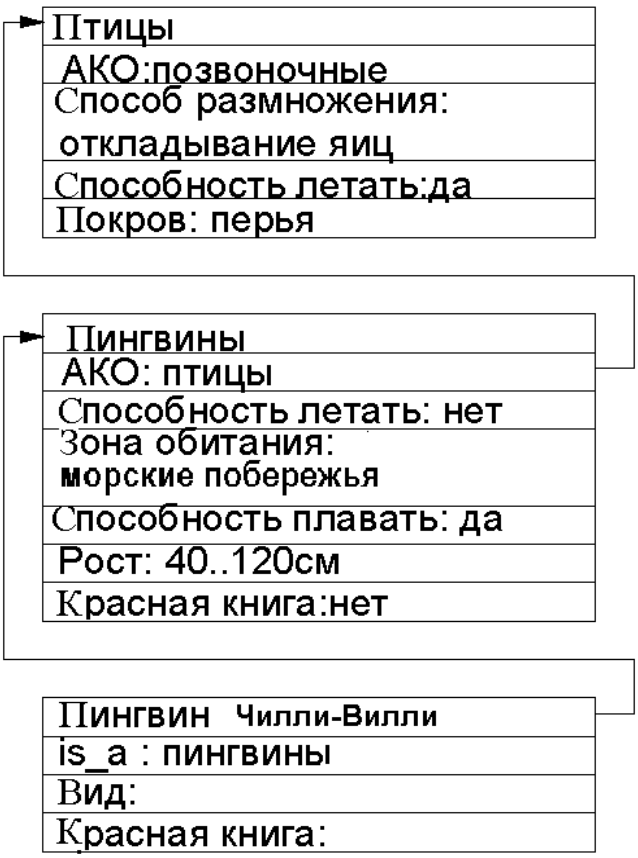


Рисунок 3.18 – Простейшая иерархия фреймов

Каждый подкласс или экземпляр класса наследует слоты своего суперкласса. Если подкласс (экземпляр класса) и суперкласс имеют слоты, с совпадающими именами, то определения значений слотов, сделанные внутри подкласса (экземпляра класса), перекрывают определения суперкласса. Например, ответ на вопрос: “Способен ли пингвин Чилли-Вилли летать?” будет отрицательным. При поиске ответа на этот вопрос фрейм-экземпляр “пингвин Чилли-Вилли” наследует все слоты фрейма “пингвины”. Значение слота “способность летать” фрейма “пингвины” перекрывает значение одноименного слота “птицы”.

В общем случае порядок наследования определяется с помощью *списка предшествований*. В случае линейной иерархической схемы этот список формируется в соответствии с направлением is_a и ako связей. Для рассматриваемого случая его можно записать в виде:

(“пингвин Чилли-Вилли”, “пингвины”, “птицы”).

Более сложная ситуация возникает, если фрейм имеет несколько is_a или ako связей. В этом случае говорят о множественном наследовании. Правила формирования списка предшествований в этом случае будут сложнее. Предположим, что пингвин “Чилли-Вилли” – это персонаж мультфильма. Тогда фрейм-экземпляр “пингвин Чилли-Вилли” будет иметь две is_a связи (рисунок 3.19).



Рисунок 3.19 – Множественное наследование

При попытке установить зону обитания этого пингвина возникает двусмысленность, которая разрешается с помощью списка предшествований, формируемого с учетом иерархии фреймов. Просмотр иерархии

фреймов выполняется сначала в глубину, а затем слева – направо. Тогда список предшествований будет иметь вид:

(“пингвин Чилли-Вилли”, “пингвины”, “птицы”, “персонаж мультфильма”).

При таком способе разрешения неопределенности зона обитания пингвина по имени Чилли-Вилли – это морские побережья. Если пользователя не удовлетворяет результат наследования, то можно ввести дополнительный класс, явно указав значение соответствующего слота. Например, можно ввести класс “пингвины-мультяшки” (рисунок 3.20) и явно указать значение слота “зона обитания”. Однако это исключает возможность передачи значения слота по умолчанию и требует дополнительной памяти[1)].

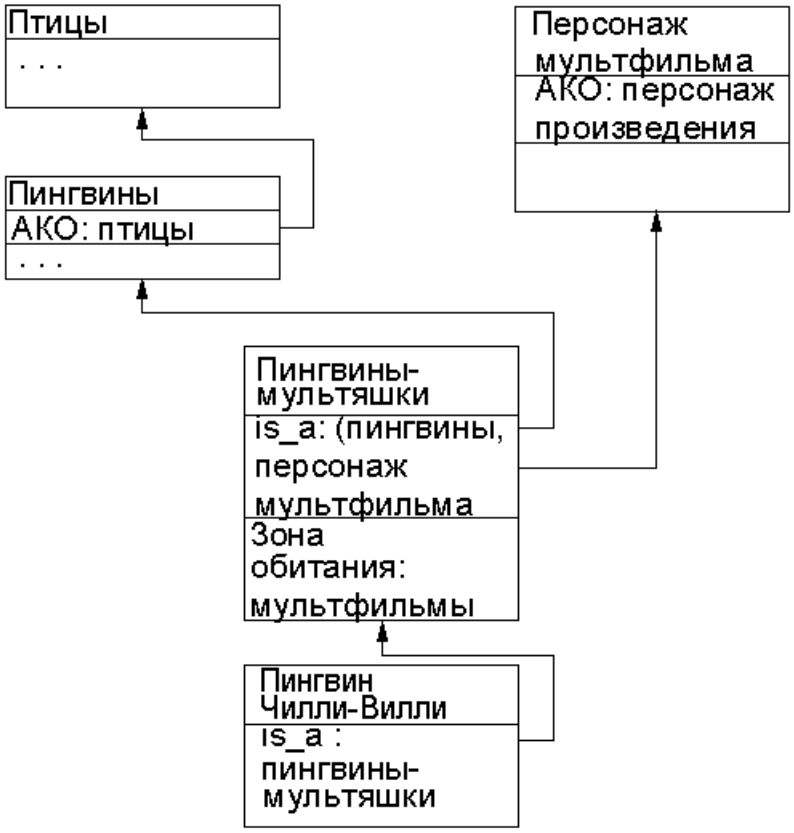


Рисунок 3.20 – Введение дополнительного класса

В реальных ситуациях взаимосвязь фреймов оказывается значительно сложнее, чем в рассмотренных примерах. Поэтому правила формирования

[1)] Конечно, в данном примере требуемый результат можно получить проще. Необходимо лишь изменить порядок следования имен фреймов в слоте is_a фрейма “пингвин Чилли-Вилли”. Т.е. порядок слева направо соответствует порядку следования имен фреймов в слоте is_a или ako.

списка предшествований усложняются. В этом случае, наряду с поиском в глубину и слева направо, применяется принцип исключения из списка предшествований повторяющихся элементов. В списке всегда остается только та повторяющаяся вершина, которая встретилась последней. Например, для рисунка 3.21 список предшествований можно представить в виде:

(экземпляр, суперкласс 2, суперкласс 4, суперкласс 3, суперкласс 5, суперкласс 1, универсальный класс).

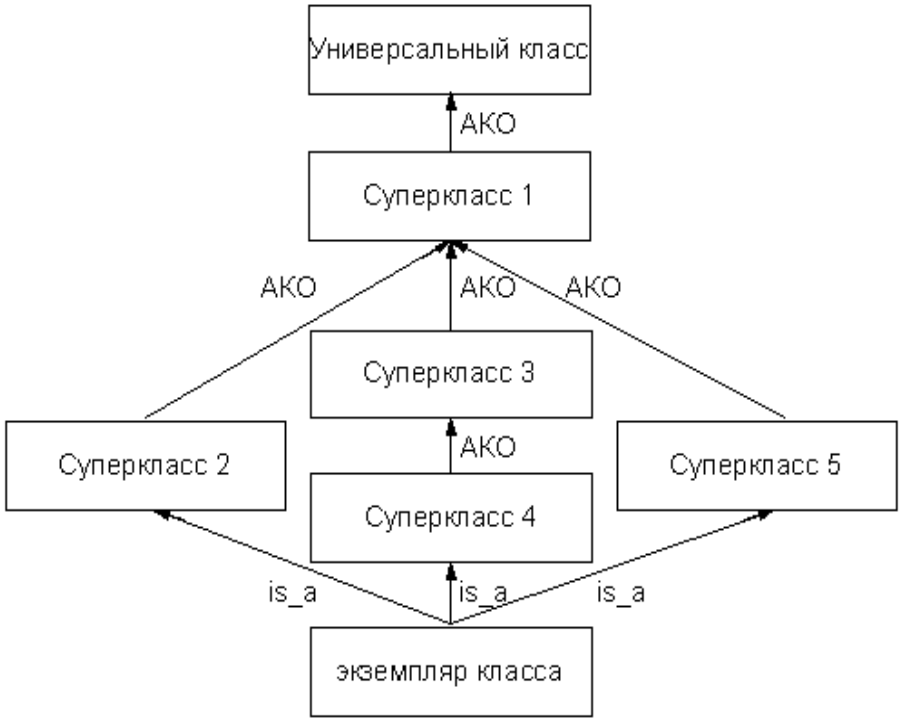


Рисунок 3.21 – Иерархия фреймов с несколькими is_a отношениями

Данный принцип позволяет реализовать требование, заключающееся в том, что любой класс должен появляться в списке предшествований раньше, чем его непосредственный суперкласс [94]. Реализация этого требования обеспечивает передачу по умолчанию наиболее специфичных данных, что соответствует нашему интуитивному представлению о наследовании свойств. Вместе с тем применение рассмотренной процедуры наследования к иерархии фреймов, изображенной на рисунке 3.22, приведет к формированию следующего списка предшествований:

(экземпляр, суперкласс 5, суперкласс 3, суперкласс 6, суперкласс 2, суперкласс 4, суперкласс 1).

В этом списке нарушено требование, состоящее в том, что любой

непосредственный суперкласс некоторого класса должен появиться в списке предшествований раньше, чем другой непосредственный суперкласс этого же класса, расположенный правее (суперкласс 2 следует после суперкласса 3). Чтобы удовлетворить указанное требование, необходимо использовать процедуру топологической сортировки [94].

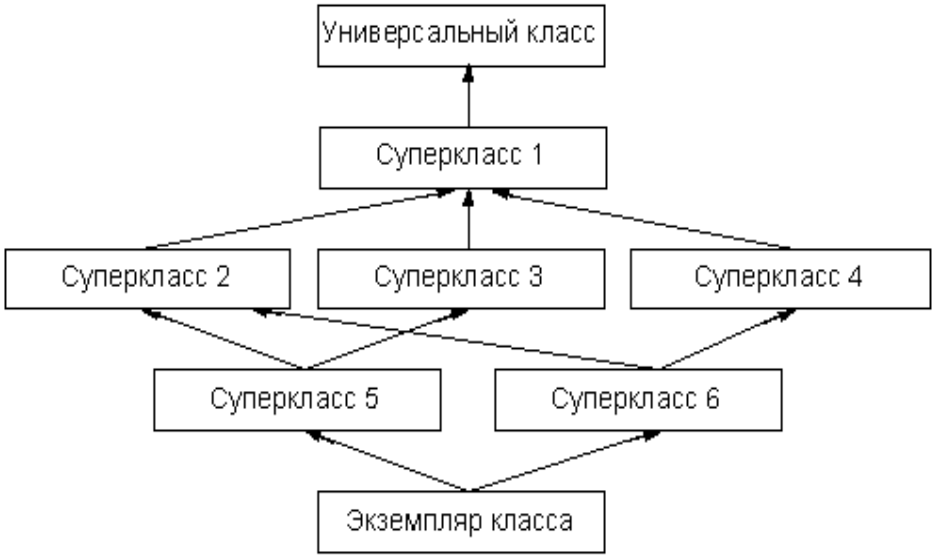


Рисунок 3.22 – Иерархия фреймов с несколькими ako отношениями

Идея такой сортировки состоит в следующем. На первом шаге составляется список экземпляров и суперклассов, которые (для рассматриваемого фрейма) достижимы посредством is_a и ako связей. Для рассматриваемого примера (рисунок 3.22) получим

(экземпляр, суперкласс 5, суперкласс 2, суперкласс 1, суперкласс 3, суперкласс 6, суперкласс 4, универсальный класс).

На втором шаге для каждого элемента полученного списка формируется список пар в соответствии с принципом, изображенным на рисунке 3.23. Данный принцип называют принципом “рыболовецкого крючка”, так как схема объединения суперклассов и экземпляров в соответствующие пары напоминает нанизывание наживки на рыболовецкий крючок. Следуя указанному принципу, для каждого элемента списка получим список соответствующих пар (таблица 3.2).

На третьем шаге просматривают элементы таблицы 3.2 и выделяют элемент, который встречается только с левой стороны пары и никогда не встречается с правой стороны. Таким элементом является “экземпляр”. Будем называть этот элемент текущим. Текущий элемент добавляется в конец списка предшествований, и все пары, в которых он встречается, вычер-

киваются. В рассматриваемом случае это пара “экземпляр–суперкласс 5”. Затем действия повторяются. Текущим элементом становится суперкласс 5, и вычеркиваются пары “суперкласс 5 – суперкласс 6” и “суперкласс 5 – суперкласс 2” и т.д. В итоге формируется следующий список предшествований:

(экземпляр, суперкласс 5, суперкласс 6, суперкласс 2, суперкласс 4, суперкласс 3, суперкласс 1, универсальный класс).

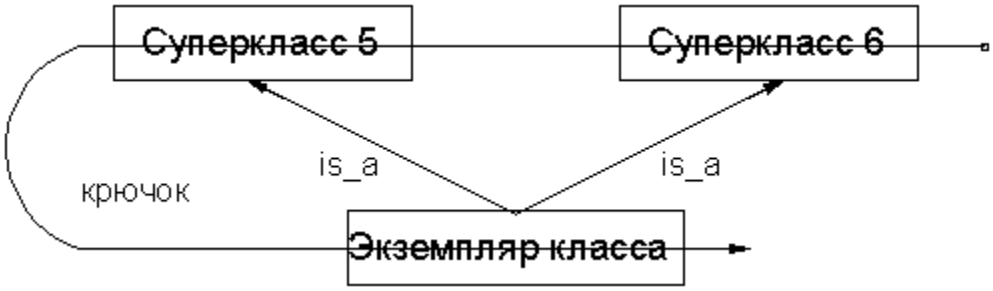


Рисунок 3.23 – Принцип “рыболовецкого крючка”

Таблица 3.2 – Список пар

Элемент	Пары
экземпляр	экземпляр – суперкласс 5, суперкласс 5 – суперкласс 6
суперкласс 5	суперкласс 5 – суперкласс 2, суперкласс 2 – суперкласс 3
суперкласс 2	суперкласс 2 – суперкласс 1
суперкласс 1	суперкласс 1 – универсальный класс
суперкласс 3	суперкласс 3 – суперкласс 1
суперкласс 6	суперкласс 6 – суперкласс 2, суперкласс 2 – суперкласс 4
суперкласс 4	суперкласс 4 – суперкласс 1
универсальный класс	универсальный класс

Если на некотором шаге описанной процедуры на роль текущего элемента претендуют сразу несколько суперклассов, то конфликт разрешается в пользу того суперкласса, который является непосредственным суперклассом для самого правого суперкласса, уже находящегося в списке предшествований. В рассматриваемом примере конфликт возникает между суперклассом 3 и суперклассом 4 после того, как в список предшествований будет помещен суперкласс 2. Предпочтение отдается суперклассу 4, так как он является суперклассом самого правого суперкласса, уже находящегося в списке предшествований, т.е. суперкласса 6.

Вывод значений слотов на основе наследования является, по своей сути, немонотонным, так как перекрытие значений в процессе наследования может менять истинность ранее установленных фактов. Например, из рисунка 3.18 следует, что пингвин Чилли-Вилли покрыт перьями. Однако

если мы добавим слот “покров” со значением ”пух” во фрейм “пингвины”, то исходное утверждение уже будет неправильным.

Другая возможность вывода во фреймовых системах основана на использовании процедур-демонов: “если-нужно”, “если-добавлено”, “если-удалено”.

Процедура “если-нужно” вызывается, когда поступает запрос, требующий установления значений соответствующего слота. Данная процедура, как и значение слота, наследуется подклассами и экземплярами классов. Поэтому связывание ее с одним из суперклассов может оказать влияние на значение соответствующих слотов фреймов, расположенных ниже по иерархии. При этом процедура непосредственно не меняет значений слота, а каждый раз вычисляет его заново, учитывая определенные условия. В частности, для слота “рост” экземпляра фрейма “пингвин” (рисунок 3.18) процедура может устанавливать рост пингвина в зависимости от значения слота “вид”. Например, если значением слота “вид” является “императорский пингвин”, то вернуть значение 100 – 120см, иначе вернуть значение 40 – 50см.

Процедуры “если-добавлено” и “если-удалено” активизируются при записи и удалении значений из слота. Например, процедура “если-добавлено” может активизироваться, когда слот “вид” получит значение (рисунок 3.18). Если новое значение – “галапагосский пингвин”, то записать в слот “красная книга” значение “да”. Очевидно, что данная процедура устанавливает ограничивающие отношения между значениями слотов “вид” и “красная книга”. Можно сказать, что процедуры “если-добавлено” и “если-удалено” устанавливают зависимость значений одного слота от значений другого и играют роль хранителя отношений ограничения во фреймовых системах.

Третья возможность вывода во фреймовых системах основана на применении присоединенных (служебных) процедур. Имя присоединенной процедуры выступает в качестве значения слота. Присоединенная процедура запускается по сообщению, переданному из другого фрейма. Для этого может использоваться специальная функция MSG, формат вызова которой имеет вид:

MSG(<имя фрейма>,<имя слота>,<параметры>).

Данная функция обеспечивает запуск присоединенной процедуры, находящейся в соответствующем слоте соответствующего фрейма. При этом присоединенная процедура получает необходимые параметры. В свою очередь, вызванная присоединенная процедура может также передать сообщение в другой фрейм и вызвать другую присоединенную процедуру и т.д. Возникает цепочка передач сообщений от одного фрейма к другому и

возврат ответов. В итоге ответ возвращается в точку первоначального вызова функции MSG и сообщается пользователю. Таким образом, во фреймовых системах на каждый из фреймов посредством присоединенных процедур возлагаются определенные обязанности, и обеспечивается их согласованное исполнение.

В заключение отметим, что если во фреймовых системах не использовать процедуры-демоны и присоединенные процедуры, то фреймовая модель представления знаний будет соответствовать семантической сети. Использование процедурных знаний во фреймовых системах повышает их гибкость, но усложняет управление. Свое дальнейшее развитие фреймовые системы получили в системах объектно-ориентированного программирования. В главе 5 мы рассмотрим систему объектно-ориентированного программирования языка Коммон Лисп.

Вопросы для самопроверки

1. Приведите определение понятия “знание”. Объясните отличие знаний от данных.
2. Перечислите основные модели представления знаний и дайте их краткую характеристику.
3. Что понимают под формальной системой?
4. Охарактеризуйте исчисление высказываний как формальную систему.
5. Какие формулы исчисления высказываний называют общезначимыми?
6. Объясните понятие “логическое следствие” теории исчисления высказываний.
7. Сформулируйте правила вывода исчисления высказываний.
8. Запишите предложения Хорна в общем виде.
9. Охарактеризуйте исчисление предикатов как формальную систему.
10. Дайте определения следующим понятиям исчисления предикатов: терм, элементарная формула, формула.
11. Какие формулы исчисления предикатов называют выполнимыми, опровержимыми?
12. Сформулируйте правила вывода исчисления предикатов.
13. Приведите пример продукционного правила.
14. Назовите основные компоненты продукционной системы и объясните процесс вывода на основе сопоставления с образцом.
15. Что понимают под конфликтным множеством?
16. Объясните на примере схему прямого вывода на множестве продукционных правил.
17. Объясните на примере схему обратного вывода на множестве продукционных правил.
18. Объясните, что понимают под семантической сетью.
19. Какие виды отношений используют в семантических сетях?
20. Что такое “падежная рамка”?
21. Приведите пример концептуального графа.
22. Какие операции вводятся над концептуальными графами?
23. Как учитываются на концептуальных графах логические операции и кванторы?

24. Как осуществляют вывод в семантических сетях?
25. Что понимают под “фреймом”? Приведите пример фрейма.
26. Какие отношения используются в сети фреймов?
27. Объясните структуру фрейма на примере системы FMS или FRL.
28. Какие способы управления выводом применяются во фреймовых системах.
29. Объясните механизм множественного наследования.
30. Объясните принцип топологической сортировки.

ГЛАВА 4

ОСНОВНЫЕ МОДЕЛИ ВЫВОДА

Под выводом понимают процесс получения заключений из предпосылок. Существуют различные модели вывода. В главе 3 было введено понятие дедуктивного вывода в логике исчисления предикатов, который базируется на применении определенных правил вывода к множеству исходных истинных утверждений, и только лишь обозначено возможное направление автоматизации этого процесса. В данной главе рассмотрим общие процедуры, позволяющие автоматизировать дедуктивный вывод в исчислении предикатов, и, прежде всего, процедуру резолюции. С этой целью введем понятие предложения для формул исчисления предикатов, затем рассмотрим понятие универсума Эрбрана и, наконец, опишем процедуру резолюции и ее применения при решении различных задач.

Хотя исчисление предикатов является весьма мощным инструментом, подходящим для описания самых различных предметных областей, однако оно не лишено недостатков. В частности, теория исчисления предикатов позволяет получать правильные заключения, опираясь на общезначимые утверждения и обоснованные правила вывода. Вместе с тем часто требуется осуществлять вывод в условиях, когда исходные данные не являются абсолютно достоверными и полными, а правила вывода носят эвристический характер и ненадежны. В этом случае применяют процедуры, обеспечивающие вывод заключений с определенной степенью уверенности (процедуры вывода на ненадежных знаниях), а также процедуры немонотонного вывода, допускающие получение в ходе вывода заключений, которые могут логически противоречить утверждениям, имеющимся в базе знаний.

Кроме дедуктивного вывода, в системах искусственного интеллекта широко используют индуктивные схемы вывода. Такие схемы вывода позволяют порождать обобщения имеющихся частных утверждений. Способность к обобщению является важной функцией естественного интеллекта и используется им для получения новых знаний. Поэтому индуктивные схемы вывода являются элементом обучающихся подсистем искусственного интеллекта. Так как при обучении из совокупности имеющихся фактов

формируются путем обобщения новые понятия и правила, то индуктивные схемы вывода находят широкое применение при автоматизации процесса приобретения знаний.

Рассмотрим указанные модели вывода подробнее.

4.1. Дедуктивный вывод в исчислении предикатов

4.1.1. Формулировка задачи дедуктивного вывода

Вернемся к представлению задач в виде теорем (см. §2.1.3). Пусть задача описывается на языке исчисления предикатов. Если обозначить через Φ_0 множество исходных утверждений и через F_g целевое утверждение, то задача, представленная в виде доказательства теоремы, формально может быть записана в виде

$$\Phi_0 \Rightarrow F_g .$$

Иными словами, необходимо доказать, что формула F_g логически следует из множества формул Φ_0.

Формула F_g логически следует из Φ_0, если каждая интерпретация, удовлетворяющая Φ_0, удовлетворяет и F_g. Когда множество Φ_0 представлено формулами $F_1,F_2,\ldots,F_n$, то задачей дедуктивного вывода исчисления предикатов является выяснение общезначимости формулы

$$F_1 \wedge F_2 \wedge \ldots \wedge F_n \rightarrow F_g .$$

В ходе вывода часто используют метод доказательства от обратного, т.е. устанавливают не общезначимость приведенной выше формулы, а невыполнимость формулы

$$\overline{F_1 \wedge F_2 \wedge \ldots \wedge F_n \rightarrow F_g}$$

или

$$F_1 \wedge F_2 \wedge \ldots \wedge F_n \wedge \overline{F_g} .$$

Иными словами, доказывают, что объединение $\Phi_0 \cup \overline{F_g}$ невыполнимо. Такое доказательство может быть существенно проще прямого вывода. Процесс установления невыполнимости некоторого множества формул называют *опровержением*.

Были разработаны эффективные методы автоматизации дедуктивного вывода. История их создания берет свое начало от основополагающих

работ Ж. Эрбрана, который предложил механическую процедуру дедуктивного вывода (1930). В 1965г. Дж. Робинсон разработал принцип резолюции, позволяющий автоматизировать процесс опровержения и являющийся теоретической базой для построения многих систем автоматического доказательства теорем. Именно эти два подхода будут рассмотрены в дальнейшем при описании дедуктивного вывода.

4.1.2. Стандартизация предикатных формул

Для осуществления дедуктивного вывода методом Эрбрана или на основе принципа резолюции, все формулы множества $\Phi_0 \cup \overline{F}_g$ должны быть представлены в виде дизъюнкций литералов.

Литералом называют элементарную формулу или ее отрицание. Дизъюнкция литералов называется *предложением* или *клозом* (от англ. clause). Например, формула $P(x) \vee Q(x,y) \vee \overline{R}(x,y)$ является предложением. Таким образом, формулы множества $\Phi_0 \cup \overline{F}_g$ должны быть записаны в форме предложений. Так как предложение содержит только знаки операций дизъюнкции и отрицания, причем отрицание должно распространяться не более чем на одну предикатную букву, то для приведения формулы к виду предложения необходимо исключить из неё все остальные операции и сократить область действия операции отрицания.

Приведение формул множества $\Phi_0 \cup \overline{F}_g$ к виду предложений выполняют в процессе указанных ниже тождественных преобразований.

Исключение знаков импликации и эквиваленции. Для этого используют равенства:

$$(P \to Q) = \overline{P} \vee Q\,,$$
$$(P \leftrightarrow Q) = (\overline{P} \vee Q) \wedge (\overline{Q} \vee P)\,.$$

Уменьшение области действия операции отрицания. При этом используют законы де Моргана:

$$\overline{(P \vee Q)} = (\overline{P} \wedge \overline{Q})\,,\ \overline{(P \wedge Q)} = (\overline{P} \vee \overline{Q}),$$
$$\overline{\forall x P(x)} = \exists x \overline{P}(x),\ \ \overline{\exists x P(x)} = \forall x \overline{P}(x)\,.$$

Например,

$$\overline{\forall x (P(x) \wedge (Q(x) \vee R(x)))} = \exists x (\overline{P}(x) \vee (\overline{Q}(x) \wedge \overline{R}(x)))\,.$$

Стандартизация переменных. В этом случае выполняют переименование переменных, связанных кванторами общности или существования. Переменные переименовываются так, чтобы каждый квантор связывал отдельную переменную, не встречающуюся в других кванторах. Например,

$$\forall x P(x) \vee \exists x Q(x) = \forall x P(x) \vee \exists z Q(z).$$

Исключение кванторов существования. В простейшем случае это соответствует правилу экзистенциальной конкретизации, которое позволяет перейти от $\exists x P(x)$ к *P(a)*, где *a* – константа (см. §3.2.3).

Если квантор существования находится в области действия квантора общности, задача становится сложнее. Рассмотрим, например, формулу $\forall x \exists y (Q(y) \wedge R(x, y))$, которая может означать, что для каждого *x* существует такой конкретный *y*, что верно *Q(y)* и *R(x,y)*. В этой интерпретации значение переменной *y* зависит от значения переменной *x*. Такая зависимость представляется функцией *y=f(x),* которую называют функцией Сколема. Функция Сколема позволяет исключить квантор существования и переписать рассматриваемую формулу в виде $\forall x (Q(f(x)) \wedge R(x, f(x)))$.

Если квантор существования находится в области действия двух и более кванторов общности, то функция Сколема будет зависеть от двух и более аргументов. Для обозначения функции Сколема не должны применяться функциональные буквы, которые уже имеются в формулах.

Перемещение кванторов общности. Все кванторы общности записываются в начале формулы. Например, формула $\forall x (P(x) \vee \forall y \overline{Q}(y))$ преобразуется к виду $\forall x \forall y (P(x) \vee \overline{Q}(y))$.

Приведение к конъюнктивной нормальной форме. Для этого используется дистрибутивный закон: $(A \wedge B) \vee C = (A \vee C) \wedge (B \vee C)$. Например, выражение $(A \wedge B) \vee (B \wedge C)$ приводится к конъюнктивной нормальной форме следующим образом:

$$(A \wedge B) \vee (B \wedge C) = [A \vee (B \wedge C)] \wedge [B \vee (B \wedge C)] = (A \vee B) \wedge (A \vee C) \wedge B \wedge (B \vee C).$$

Таким образом, любая формула из множества $\Phi_0 \cup \overline{F_g}$ записывается в виде $\forall x_1 ... \forall x_n (K_1 \wedge K_2 \wedge ... \wedge K_n)$. Здесь каждый член K_i представляет собой предложение (дизъюнкт), т.е. имеет вид $(L_1 \vee L_2 \vee ... \vee ... L_k)$, где L_i – литерал.

Исключение кванторов общности. Так как все переменные в формулах связаны кванторами общности, а порядок следования кванторов общности не имеет значения, то их можно не указывать в явном виде, то есть исключить.

Исключение конъюнкций. Любая интерпретация удовлетворяет формуле $K_1 \wedge K_2 \wedge ... \wedge K_n$ в том случае, когда она удовлетворяет формулам $K_1, K_2, ..., K_n$. Поэтому каждая формула, приведенная к конъюнктивной нормальной форме, может быть заменена множеством формул

$$\{K_1, K_2, ..., K_n\}.$$

Рассмотрим пример преобразования формулы к множеству предложений. Пусть дана формула

$$\forall x\{P(x) \to \{\overline{\forall y[Q(x,y) \to R(y)]}\}.$$

Исключим знаки импликации:

$$\forall x\{\overline{P}(x) \vee \{\overline{\forall y[\overline{Q}(x,y) \vee R(y)]}.$$

Уменьшим область действия операции отрицания:

$$\forall x\{\overline{P}(x) \vee \{\exists y[Q(x,y) \vee \overline{R}(y)]\}.$$

Исключим квантор существования:

$$\forall x\{\overline{P}(x) \vee [Q(x, f(x)) \vee \overline{R}(f(x))].$$

Приведем формулу к конъюнктивной нормальной форме:

$$\forall x\{(\overline{P}(x) \vee Q(x, f(x)) \wedge (\overline{P}(x) \vee \overline{R}(f(x)))\}.$$

Исключив квантор общности и конъюнкцию, получим множество предложений:

$$\{\overline{P}(x) \vee Q(x, f(x)), \overline{P}(x) \vee \overline{R}(f(x))\}.$$

4.1.3. Метод Эрбрана

Как отмечалось выше, доказательство теоремы $\Phi_0 \Rightarrow F_g$ состоит в том, чтобы показать, что расширенное множество формул, образованное путем объединения множества аксиом Φ_0 и отрицания целевого утверждения F_g, невыполнимо. Обозначим расширенное множество формул, приведенных предварительно к виду предложений (дизъюнктов), через S. Множество дизъюнктов S невыполнимо тогда, когда не существует интерпретации, которая ему удовлетворяет. Для того чтобы это выяснить, необходимо перечислить все интерпретации на всех возможных областях. Очевидно, рассмотреть все интерпретации на всех областях невозможно. Но,

тем не менее, решение задачи может быть найдено. Оно основывается на понятии универсума Эрбрана. *Универсум Эрбрана* представляет такую область интерпретации *H*, что если не существует интерпретации множества предложений *S* в этой области, то ее не существует вообще. Универсум Эрбрана строится из константных термов и определяется следующим образом:

а) множество всех предметных констант, встречающихся в *S*, принадлежит *H*; если в *S* нет констант, то в *H* включается произвольная константа *a*;

б) множество всех функций, встречающихся в *S* и определенных на множестве термов из *H*, принадлежит *H*.

Проиллюстрируем построение универсума Эрбрана на примере. Пусть

$$S = \{\overline{P}(x, \quad f(x,a)) \vee Q(x,a) \vee R(x,b)\}.$$

В множество *S* входят предметные константы *a* и *b*, функциональная буква *f*. В этом случае область *H* представляет бесконечное счетное множество:

$$H = \{a, b, f(a,a), f(a,b), f(b,a), f(b,b), f(a, f(a,a)), f(a, f(a,b)),...\},$$

которое формируют последовательно в виде уровней:

$$H_0 = \{a,b\},$$
$$H_1 = \{a,b,f(a,a),f(a,b),f(b,a),f(b,b)\},$$
$$H_2 = \{a,b,f(a,a),...,f(b,b),f(a,f(a,a)),f(a,f(a,b)),...\},...$$

Здесь H_0, H_1, H_2 и т.д. – уровни универсума Эрбрана и $H = H_\infty$.

Дизъюнкт, который не содержит переменных, называется *фундаментальным*. Он получается заменой предметных переменных константными термами. Если термы являются элементами универсума Эрбрана, то фундаментальный дизъюнкт называют *фундаментальным примером дизъюнкта*. Множество всех фундаментальных атомов вида $P(t_1, t_2, \ldots t_n)$, где *P* – предикатные буквы, встречающиеся в *S*, а $t_1, t_2, \ldots t_n$ – элементы универсума Эрбрана, называется *эрбрановской базой* для *S*. Например, эрбрановской базой для рассматриваемого выше примера является множество:

$$A = \{\overline{P}(a, f(a,a)), Q(a,a), R(a,b), \overline{P}(b, f(b,a)),...\}.$$

Задание интерпретации множества *S* в области *H* заканчивают, когда с каждой атомарной формулой эрбрановской базы связано соответствующее значение истинности.

Введенные определения позволяют сформулировать *теорему Эрбрана*: множество дизъюнктов S невыполнимо тогда и только тогда, когда существует конечное невыполнимое подмножество фундаментальных примеров дизъюнктов в S. Данная теорема позволяет формально построить процедуру опровержения. Для установления невыполнимости множества дизъюнктов S необходимо: 1) образовывать множества $S_1, S_2, \ldots, S_i, \ldots$ фундаментальных примеров дизъюнктов для каждого уровня i множества H; 2) последовательно проверять их на ложность. Согласно теореме Эрбрана, если S невыполнимо, то процедура обнаружит такой уровень i, что S_i будет ложным.

На указанном принципе работали первые программы доказательства теорем. Основным недостатком процедуры Эрбрана является экспоненциальный рост множества фундаментальных примеров S_i при увеличении уровня i. Поэтому на практике удается с помощью процедуры Эрбрана доказывать только простые теоремы.

4.1.4. Принцип резолюции

Принцип резолюции представляет собой процедуру вывода, с помощью которой порождаются новые дизъюнкты из S. Если в ходе применения этой процедуры выводится пустое предложение, то S невыполнимо.

Правило вывода, лежащее в основе принципа резолюции, по существу, представляет собой расширение правила modus ponens и правила силлогизма. Данные правила вывода можно записать в форме:

$$\frac{A,\ \overline{A} \vee B}{B} \quad \text{(modus ponens)}, \qquad \frac{\overline{A} \vee B,\quad \overline{B} \vee C}{\overline{A} \vee C} \quad \text{(силлогизм)}.$$

В каждом из этих правил имеются дизъюнкты-посылки, содержащие *дополнительные (контрарные) пары литер* (A *и* $\overline{A}$, B *и* $\overline{B}$). Можно заметить, что следствие правил формально получается вычеркиванием в дизъюнктах-посылках контрарных пар литер и объединением оставшихся частей дизъюнктов.

Рассмотренные дизъюнкты-посылки содержали одну или две литеры. Дж. Робинсон распространил указанный прием на случай произвольных дизъюнктов с любым числом литер и сформулировал *правило резолюции*: если любые два дизъюнкта D_1 и D_2 содержат дополнительную пару литер, например L и $\overline{L}$, то, вычеркивая их, формируем новый дизъюнкт из оставшихся частей дизъюнктов D_1 и D_2. Вновь сформированный дизъюнкт называется *резольвентой* (следствием) исходных дизъюнктов. Резольвента, полученная из двух дизъюнктов, является логическим следствием этих

дизъюнктов. Правило резолюции позволяет получать резольвенты множества дизъюнктов *S*. Если в процессе вывода резольвент получаются два одно-литеральных дизъюнкта, образующих контрарную пару, то резольвентой этих двух дизъюнктов будет пустой дизъюнкт, обозначаемый символом *NIL*.

Объединение исходного множества дизъюнктов с множеством всех резольвент, которые могут быть образованы из дизъюнктов, входящих в *S*, будем обозначать *R(S)*. Применяя принцип резолюции к *R(S),* получим $R(R(S)) = R_2(S)$. В общем случае $R_{n+1}(S) = R(R_n(S))$. При этом $R_0(S)=S$. Легко заметить, что если множество *S* невыполнимо, то $R_i(S)$ при любых $i \geq 1$ также невыполнимо, и наоборот, если невыполнимо $R_i(S)$, то невыполнимо и *S*. Это свойство называется *полнотой принципа резолюций*. Оно устанавливается следующей теоремой, доказанной Робинсоном: если *S* – произвольное конечное множество дизъюнктов, то *S* невыполнимо тогда и только тогда, когда $R_i(S)$ содержит пустой дизъюнкт. Данная теорема позволяет построить процедуру опровержения, которая для исходного множества *S* последовательно строит множества $R_1(S), R_2(S), \ldots R_i(S)$ и проверяет, содержит ли множество $R_i(S)$ пустой дизъюнкт.

Вывод с помощью резолюции наглядно можно проиллюстрировать с помощью *графа опровержения*. Вершинами графа являются исходные дизъюнкты и резольвенты, получаемые в процессе вывода. Дизъюнкты исходного множества *S* представляют вершины графа, располагаемые в его верхней части. Если два дизъюнкта, находящиеся в каких-либо вершинах графа, образуют резольвенту (разрешаются), то вершина, соответствующая резольвенте, изображается ниже данных дизъюнктов и соединяется с ними ребрами.

В качестве примера на рисунке 4.1 построен граф опровержения множества исходных дизъюнктов:

$$S=\left\{A,\ \ \overline{A}\vee B,\ \ \overline{B}\vee C,\ \ \overline{C}\vee D,\ \ \overline{D}\right\}.$$

В логике высказываний нахождение контрарных пар не вызывает трудностей. Сложнее обстоит дело в логике предикатов. Например, если имеются дизъюнкты $P(x, f(x))$, $\overline{P}(a, y)\vee Q(z)$, $\overline{Q}(b)$, то резольвента может быть получена только после выполнения следующих подстановок: *a* вместо *x*, *f(x)* вместо *y*, *b* вместо *z*. Затем по правилу резолюции получается пустое предложение.

Подстановкой называется конечное множество

$$\alpha = \left\{t_1 / x_1,\ \ t_2 / x_2, \ldots, t_n / x_n\right\},$$

где x_i - переменная; t_i - терм, отличный от x_i, $1 \le i \le n$. Применение подстановки α к некоторой формуле F означает замену всех вхождений переменной x_i на терм t_i. Формулу, получившуюся в результате подстановки, обозначают F_α и называют *примером F*. Для предыдущего примера подстановку можно записать в форме $\alpha = \{a/x, \quad f(x)/y, \quad b/z\}$. Тогда пример формулы $P(x, f(x))$ будет иметь вид $P(x, f(x))_\alpha = P(a, f(a))$.

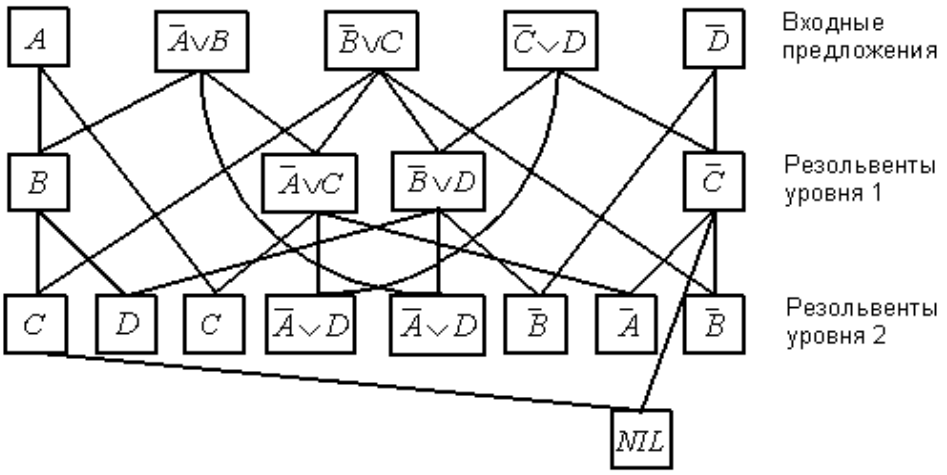


Рисунок 4.1 – Граф опровержения

Композицией двух подстановок α и β называется подстановка $\alpha\beta$, получаемая применением подстановки β к термам подстановки α с добавлением из β всех пар t_i/x_i, содержащих переменные, отсутствующие в α. Например, если $\alpha = \{f(x)/z\}$, $\beta = \{a/x, b/y, c/z\}$, то композиция подстановок будет соответствовать множеству $\alpha\beta = \{f(a)/z, \quad a/x, \quad b/y\}$.

Подстановки выполняются с целью приведения литералов к некоторой общей унифицированной форме, чтобы затем исключить образовавшиеся контрарные пары литералов. Подстановка α называется *унификатором* для множества литералов $\{L_1, L_2, \ldots, L_k\}$, если $L_{1\alpha} = L_{2\alpha} = \ldots = L_{k\alpha}$.

Например, литералы $\{P(y, f(x)), \quad P(a, z)\}$ с помощью подстановки $\alpha_1 = \{a/y, \quad f(x)/z\}$ преобразуются к виду *P(a,f(x))*. Следовательно, α_1 является унификатором указанного множества литералов. Если ввести подстановку $\alpha_2 = \{b/x\}$, то можно убедиться в том, что композиция подстановок $\alpha = \alpha_1\alpha_2$ также унифицирует рассматриваемое множество литералов и преобразует их к виду *P(a, f(b))*. Поэтому подстановка α – тоже унификатор. Однако подстановка α получается из подстановки α_1 с помощью α_2.

Унификатор, который позволяет получить все другие унификаторы с помощью подходящих подстановок, называется *наиболее общим унификатором* (НОУ). Чтобы получить резольвенту, необходимо найти НОУ.

Рассмотрим алгоритм построения НОУ на примере множества литералов:

$$\{P(g(x), x, f(x)), \quad P(y, a, x)\}.$$

Если два литерала имеют общую переменную, то необходимо ее заменить новой переменной в одном из них. Заменив переменную *x* на *z* во втором литерале, получим *P(x,a,z)*. Далее, рассматриваем слева направо каждый из литералов и находим позиции, в которых обнаруживается не совпадение термов. В данном случае не совпадают термы *g(x)* и *y*. Эти термы образуют множество рассогласования {*g(x),y*}, где *y* – переменная. Поэтому выполняется подстановка $\alpha_1 = \{g(x)/y\}$, которая преобразует множество исходных литералов к виду {*P(g(x)), x, f(x)), P(g(x)), a, z)*}. Продолжая процесс сопоставления термов, обнаруживаем несовпадение *x* и *a*, которые формируют множество рассогласования {*x,a*}, соответствующее подстановке $\alpha_2 = \{a/x\}$. Применив α_2 к множеству унифицируемых литералов, получим {*P(g(a)),a,f(a)),P(g(a)),a,z)*}. Наконец, обнаружив несовпадение литералов *f(a)* и *z*, выполняем подстановку $\alpha_3 = \{f(a)/z\}$. Таким образом, НОУ представляет композицию подстановок $\alpha_3 = \alpha_1\alpha_2\alpha_3$.

Алгоритм поиска НОУ для двух литералов L_1 и L_2, не содержащих общих переменных, можно представить в виде процедуры [86]:

```
Procedure Unify(L1,L2);
  Begin
   α:=[ ]; {пустая подстановка}
   Q1:=L1; Q2:=L2;
   While True Do
   Begin
      Построить множество рассогласования D для
      цепочек символов, образующих Q1 и Q2;
      If  D=[ ] Then Возврат(α);
      t1=first(D); t2=last(D);
      If var(t1) and non_contain(t1,t2) Then α1:={ t2/t1}
      Else If var(t2) and non_contain(t2,t1) Then α1={ t1/t2}
            Else Выход('Неудача');
      Q1:=Q1α1;
      Q2:=Q2α1;
      α :=αα1
  End
End.
```

В этой процедуре полагается, что множество рассогласования D состоит из двух термов t_1 и t_2 , т.е. D=$\{ t_1,t_2\}$. Тогда вызов функции first(D) соответствует выделению t_1 – первого элемента множества, а вызов функции last(D) – выделению t_2. Функция var(t) возвращает значение true, если аргумент t – переменная. Функция non_contain(x,y) – предикат, возвращающий значение true, если x не содержится в y. Если x входит в y, то унификация не выполнима. Например, нельзя унифицировать x и f(x).

Обобщая сказанное, отметим, что в логике предикатов нахождение резольвенты двух предложений (дизъюнктов) сводится к следующим действиям:

1) переменные в предложениях переименовываются так, чтобы они не совпадали;
2) находится подстановка (НОУ), при которой литерал одного предложения становится дополнительным к какому-либо литералу другого предложения;
3) дополнительные литералы вычеркиваются;
4) если получены одинаковые литералы, то все они, за исключением одного в каком-либо предложении, вычеркиваются;
5) дизъюнкция оставшихся литералов каждого из предложений и есть резольвента.

На рисунке 4.2 изображено дерево опровержения невыполнимого множества дизъюнктов $S=\{R(x)\vee Q(x),\ \overline{Q}(f(z)),\ \overline{R}(w)\vee P(b),\ \overline{P}(y)\}$, являющихся формулами исчисления предикатов. На дереве показаны дизъюнкты, получающиеся в результате соответствующих подстановок.

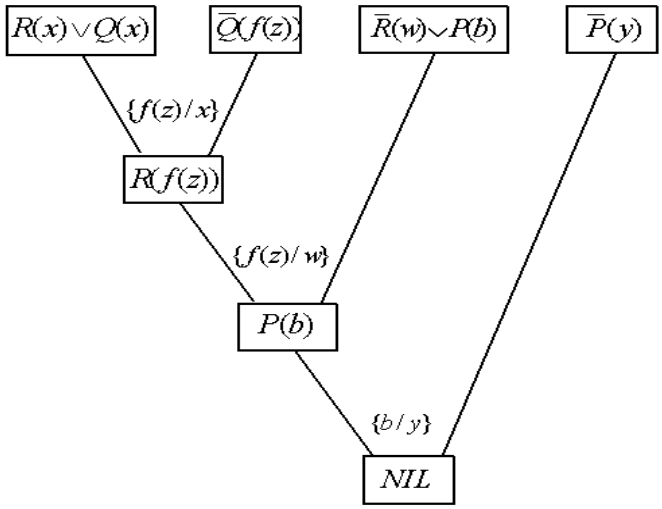


Рисунок 4.2 – Дерево опровержения

В процессе унификации могут получаться факторы дизъюнктов. Если несколько литералов дизъюнкта D имеют НОУ α, то D_α называют *факто-*

ром. Например, пусть $D = Q(x) \vee Q(f(y)) \vee \overline{P}(x)$. Тогда НОУ α={f(y)/x} и фактор $D_\alpha = Q(f(y)) \vee \overline{P}(f(y))$. Факторы представляют собой подстановочные частные случаи, которые позволяют уменьшить дизъюнкты по длине. Принцип резолюции предполагает использование не только резольвент, но и факторов.

В общем, процесс опровержения не протекает столь эффективно, как изображено на рисунке 4.2. Во многих случаях множества $R_1(S)$, $R_2(S), \ldots R_i(S)$ очень быстро разрастаются, так как непосредственное применение принципа резолюций соответствует "слепому" поиску.

Существует большое число модификаций принципа резолюций, направленных на нахождение эффективных стратегий поиска наиболее предпочтительных дизъюнктов для бинарной резолюции. Ниже рассмотрим кратко только некоторые из стратегий поиска, нашедших наиболее широкое применение.

4.1.5. Стратегии поиска

Для решения проблемы "комбинаторного взрыва", возникающей при непосредственном применении принципа резолюций, необходимо осуществлять целенаправленный подбор предложений, участвующих в процессе унификации. Выбор должен выполняться так, чтобы пустое предложение достигалось как можно скорее. Конечно, на каждом этапе опровержения выбор зависит от текущей ситуации. Однако имеются общие стратегии поиска, которые, вне зависимости от контекста задачи, позволяют сократить число резолюций. Ниже рассмотрим следующие четыре стратегии: стратегию унитарности, стратегию опорного множества, стратегию исходных данных, линейную стратегию.

Стратегия унитарности. Согласно этой стратегии, предпочтение отдается тем бинарным резолюциям, где одно из предложений содержит единственный литерал. Такая стратегия гарантирует уменьшение длины результирующих предложений. Например, следствием одно-литерального предложения P и предложения $\overline{P} \vee \overline{Q} \vee R$ будет $\overline{Q} \vee R$, которое короче исходного предложения. Стратегия унитарности упорядочивает выбор пар предложений, упрощая последующие этапы доказательства невыполнимости исходного множества дизъюнктов.

Стратегия опорного множества. Более эффективной может оказаться попытка исключить из рассмотрения сразу несколько предложений на основе понятия опорного множества. Невыполнимое множество предложений S можно разбить на два подмножества S_1 и S_2. Подмножество S_1 состоит из аксиом, а S_2 – из отрицаний тех предложений, которые необходимо

доказать. Очевидно, что подмножество S_1 не содержит противоречий. Поэтому нахождение резольвент предложений из S_1 не допускается. Опорным называют множество предложений, состоящее из дизъюнктов, входящих в S_2, и резольвент тех пар предложений, из которых хотя бы одно принадлежит опорному множеству. Стратегия опорного множества допускает выполнение тех резолюций, в которых, по крайней мере, одно из предложений входит в опорное множество. Если опорное множество относительно небольшое, то это значительно сокращает пространство поиска. Достоинство стратегии опорного множества также состоит в том, что формируемое дерево опровержения легко интерпретируется человеком, так как оно управляется целью.

Если множество исходных предложений невыполнимо, то рассмотренная стратегия приводит к пустому предложению, т.е. она является полной. Стратегия опорного множества является специальным случаем более общей семантической резолюции [21].

Линейная резолюция. Сначала находится следствие двух предложений исходного множества *S*. На следующих этапах в резолюции участвует резольвента C_i, полученная на предыдущем шаге (называемая центральным дизъюнктом), и дизъюнкт *B* (называемый боковым), являющийся либо одним из дизъюнктов исходного множества *S*, либо центральным дизъюнктом C_j, который предшествует в выводе дизъюнкту C_i, т.е. $j < i$. Линейная стратегия является полной и одной из наиболее эффективных в реализации.

Стратегия входных данных. Такая стратегия соответствует линейной резолюции с одним ограничением: в качестве боковых дизъюнктов выбираются только дизъюнкты исходного множества *S*. Дерево опровержения, изображенное на рисунке 4.2, построено в соответствии со стратегией входных данных.

Входная резолюция является полной только, если предложения множества *S* являются дизъюнктами Хорна.

Рассмотренные стратегии поиска можно комбинировать. Например, стратегии унитарности и опорного множества. Кроме этого, можно упрощать множество исходных предложений *S*. Для этого исключаются тавтологии, предложения, содержащие уникальные литералы, подслучаи [26]. Например, если множество *S* содержит дизъюнкт *P(x)*, то не имеет смысла включать в состав *S* дизъюнкты вида *P(a)* или $P(a) \vee Q(b)$, так как они представляют подслучаи *P(x)*.

Разработано большое число стратегий поиска резольвент при доказательстве теорем. Основной способ решения проблемы “комбинаторного взрыва” заключается в использовании семантики и встраивании в правила вывода специфики конкретной предметной области [21].

4.1.6. Применения принципа резолюции

Применение принципа резолюции предполагает перевод исходной постановки задачи на язык исчисления предикатов. При этом рассматриваются два случая. В первом случае требуется выяснить, следует ли логически некоторая формула F_g из множества формул Φ_0, во втором случае, который встречается гораздо чаще, ставится задача нахождения значения аргумента x, при котором данная формула F_g, содержащая x, следует из множества Φ_0. Иными словами, во втором случае требуется сначала установить справедливость утверждения

$$\Phi_0 \Rightarrow \exists x F_g(x),$$

а затем найти значение переменной x, при котором указанное утверждение выполняется. Выясним, как извлечь из доказанной теоремы ответ на поставленный вопрос. Процесс формирования ответа рассмотрим на примерах задач информационного поиска, планирования перемещения робота, автоматического написания программ.

4.1.6.1 Информационный поиск. Пусть имеется база данных, содержащая сведения о химических соединениях. Например, факты, указывающие какие соединения считаются оксидами, и какой цвет имеет то или иное соединение. Допустим, что среди многих фактов, хранящихся в базе данных, имеется утверждение о том, что соединение *MgO* – оксид, и цвет данного оксида – белый. Эти факты можно представить на языке исчисления предикатов в виде двух одноместных предикатов: *oxide(MgO)* и *white(MgO)*. Здесь в качестве имен предикатов выбраны слова английского языка *oxide* (оксид) и *white* (белый).

Пусть требуется установить, существует ли оксид белого цвета. Данный вопрос на языке исчисления предикатов запишется в виде

$$F_g(x) = \exists x(oxide(x) \wedge white(x)).$$

Формулу $F_g(x)$, которую необходимо доказать, называют предположением. Найдем отрицание предположения

$$\overline{F_g}(x) = \forall x(\overline{oxide}(x) \vee \overline{white}(x))$$

и докажем, что множество $S = \{oxide(MgO), white(MgO), \overline{oxide}(x) \vee \overline{white}(x)\}$ невыполнимо. Для этого построим дерево опровержения (рисунок 4.3).

Затем необходимо извлечь то частное значение переменной, относящееся к квантору существования, которое и служит ответом на поставлен-

ный вопрос. Для этого строят *модифицированное дерево доказательства*. Процесс его построения состоит в следующем:

1) к отрицанию предположения дописывают через дизъюнкцию само предположение, т.е. вместо отрицания предположения записывают тавтологию;
2) находят резольвенты тех же предложений, которые участвовали в построении дерева опровержения.

В корне модифицированного дерева доказательства получается предложение, которое и используется в качестве развернутого ответа на поставленный вопрос. Модифицированное дерево доказательства изображено на рисунке 4.4.

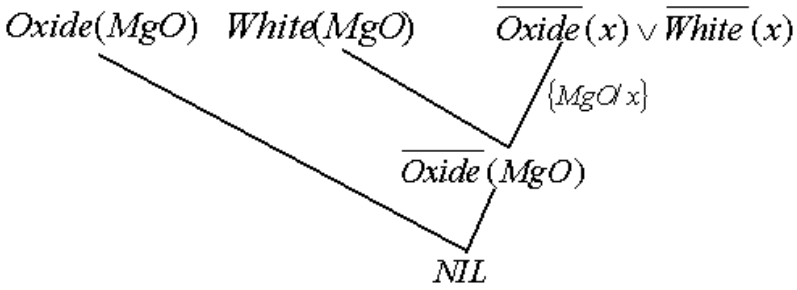


Рисунок 4.3 – Дерево опровержения

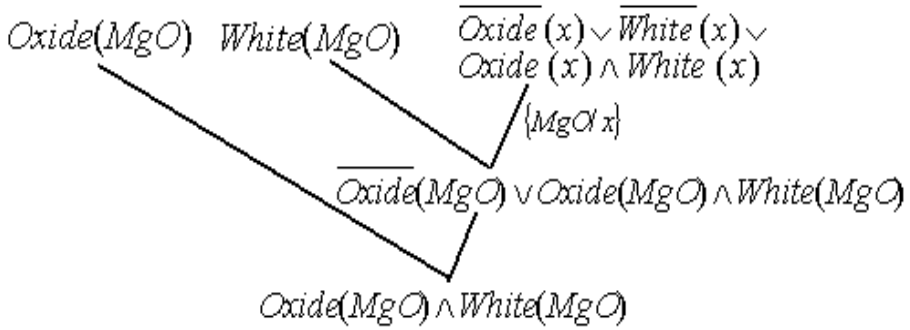


Рисунок 4.4 – Модифицированное дерево доказательства

Интерпретируя предложение, находящееся в корне модифицированного дерева доказательства, заключаем, что существует оксид *MgO*, который имеет белый цвет.

Рассмотренный пример демонстрирует ответ на вопрос типа “да” или “нет”. Вопросы, относящиеся к типу “сколько”, также можно сформулировать в виде доказательства теорем. Пусть имеется правило [46]: “если в *x* есть *m* элементов типа *y*, а в *y* есть *n* элементов типа *z*, то в *x* есть $m \times n$ элементов типа *z*”. Данное правило можно записать в виде формулы, используя предикат *HP(a,b,c)*, который интерпретируется следующим образом: “*a*” содержит “*c*” элементов типа “*b*”. Тогда правило выражается в виде

$$\forall x \forall y \{HP(x, y, m) \wedge HP(y, z, n) \rightarrow HP(x, z, times(m, n))\},$$

где функция *times(m,n)* обозначает произведение *m* на *n*. Предположим, что под объектом *x* понимается человек. Системе известно, что человек имеет две руки, и на каждой руке – пять пальцев. Требуется выяснить, сколько пальцев на руках у человека? Безусловно, с точки зрения естественного интеллекта, это не вопрос. Однако отметим, что система должна формально вывести ответ на поставленный вопрос на основе известных ей фактов и общего правила.

Факты, известные системе, и вопрос можно записать в следующем виде:

HP(man, hand, 2);
HP(hand, fingers, 5);
$\exists k(HP(man, fingers, k))$,

где *man* – человек, *hand* – рука, *fingers* – пальцы. Вопрос интерпретируется так: существует ли такое k, что справедливо *HP(man, fingers, k)*.

Построим множество *S*. Для этого приведем правило к виду предложения

$$\overline{HP}(x, y, m) \vee \overline{HP}(y, z, n) \vee HP(x, z, times(m, n))$$

и найдем отрицание предположения

$$\overline{F}_g(k) = \forall k \overline{HP}(man, fingers, k).$$

Тогда

$$S = \{HP(man, hand, 2),$$
$$HP(hand, fingers, 5),$$
$$\overline{HP}(x, y, m) \vee \overline{HP}(y, z, n) \vee HP(x, z, times(m, n)),$$
$$\overline{HP}(man, fingers, k)\}.$$

Модифицированное дерево доказательства, построенное по принципу входной резолюции, изображено на рисунке 4.5. На первом шаге находится следствие предложения, представляющего правило, и отрицания целевого утверждения. На втором и третьем шагах используются факты $HP(man, hand, 2)$, $HP(hand, fingers, 5)$. В корне дерева формируется ответ на поставленный вопрос *HP(man, fingers, times(2,5))*, т.е. количество пальцев на руках у человека можно вычислить с помощью вызова функции *times(2,5)*.

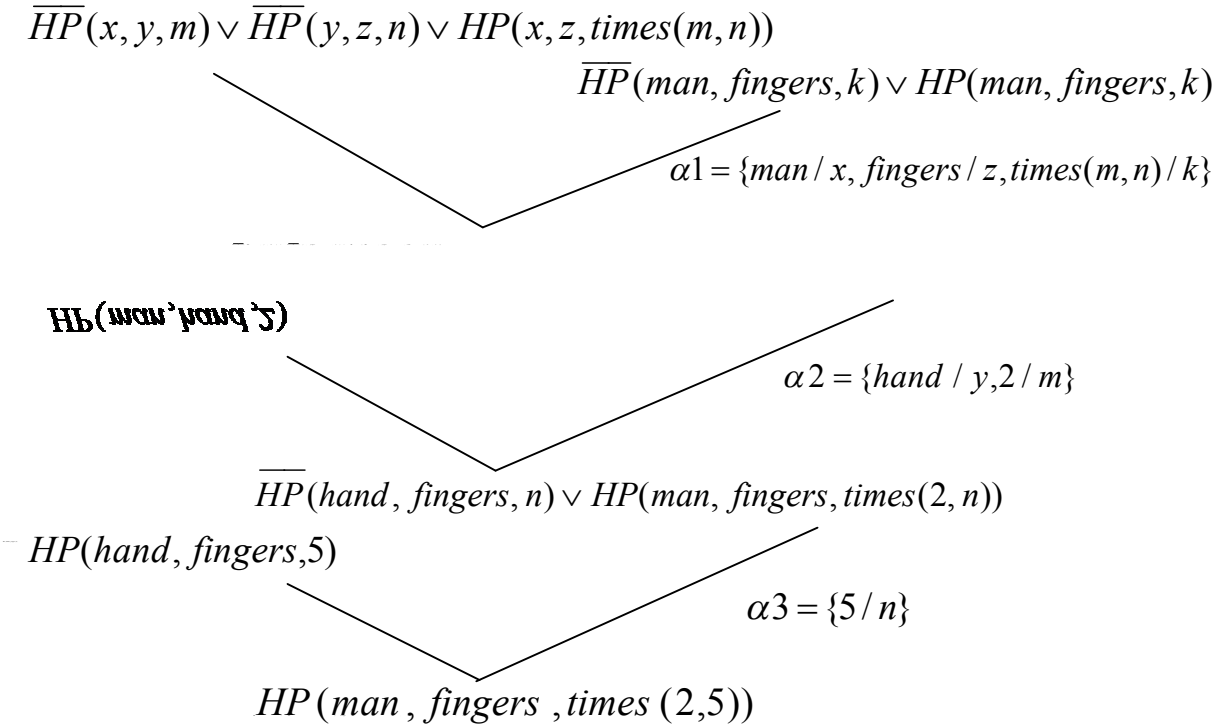


Рисунок 4.5 – Модифицированное дерево доказательства

В рассмотренных выше примерах предположение содержало квантор существования. В тех случаях, когда предположение содержит квантор общности, возникают дополнительные трудности. Дело в том, что при отрицании квантора общности он переходит в квантор существования, который в процессе стандартизации предикатных формул исключается путем введения функций Сколема. Трудности состоят в интерпретации этих функций, если они появляются в ответном утверждении. В [30] рекомендуется такие функции заменять новыми переменными. При доказательстве в эти переменные не делают никаких подстановок.

4.1.6.2. Планирование перемещений робота. Данная задача формулируется как задача поиска пути на графе состояний (рисунок 4.6) [26,46]. В исходном состоянии задачи робот находится в позиции "*a*". Необходимо найти последовательность операторов, обеспечивающих перемещение робота в позицию "*c*", которая соответствует решенному состоянию задачи. Для решения задачи необходимо установить соответствие между описанием задачи в пространстве состояний и описанием ее на языке исчисления предикатов.

Операторам пространства состояний ставятся в соответствие функции языка исчисления предикатов. При этом преобразование состояния s_1 в состояние s_2 задается функцией $s_2=f(s_1)$. Для участка графа состояний можно записать аксиому $P(s_1) \to Q(s_2)$ или $P(s_1) \to Q(f(s_1))$, где $P(s_1)$ –

предикат, описывающий свойство начального состояния участка графа; $Q(s_2)$- предикат, задающий свойство конечного состояния участка графа. Иными словами, если состояние s_1 обладает свойством P, то состояние $s_2=f(s_1)$ обладает свойством Q.

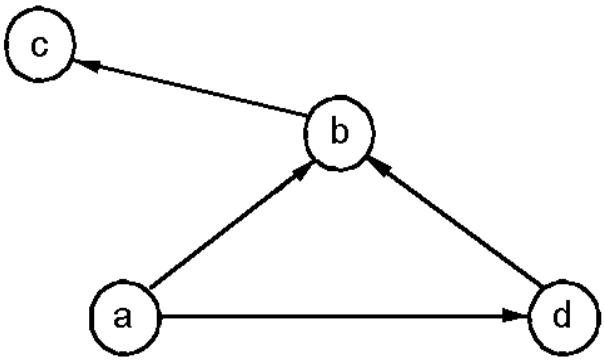


Рисунок 4.6 – Граф состояний задачи перемещения робота

Введем функцию *move(x,y,s),* которая соответствует перемещению робота из точки x в точку y в состоянии s. Для описания состояния задачи будем использовать предикат *AT(x, s)*, обозначающий, что для состояния задачи s робот находится в точке x. В этом случае можно записать следующие аксиомы:

$$AT(a,s_0),$$
$$\forall s_1(AT(a,s_1)\rightarrow AT(b,move(a,b,s_1))),$$
$$\forall s_2(AT(a,s_2)\rightarrow AT(d,move(a,d,s_2))),$$
$$\forall s_3(AT(b,s_3)\rightarrow AT(c,move(b,c,s_3))),$$
$$\forall s_4(AT(d,s_4)\rightarrow AT(b,move(d,b,s_4))).$$

Целевое утверждение формулируется следующим образом: существует ли состояние задачи s, при котором робот будет находиться в точке "*c"?* На языке исчисление предикатов это можно записать в виде

$$\exists sAT(c,s).$$

Дерево опровержения для данной задачи изображено на рисунке 4.7. Результирующая функция, обеспечивающая переход робота из точки "*a"* в точку "*c"*, получается в результате серии подстановок в переменную *s:*

$$s= move(b, c, move(a, b, s_0)).$$

Выполнение действий в соответствии с этой функцией состоит в том, что робот сначала должен переместиться из точки "*a"* в точку "*b"*, а затем

из точки “*b”* в точку “*c”*. Рассмотренный подход использовался Грином в вопросно-ответной системе QA3 [46].

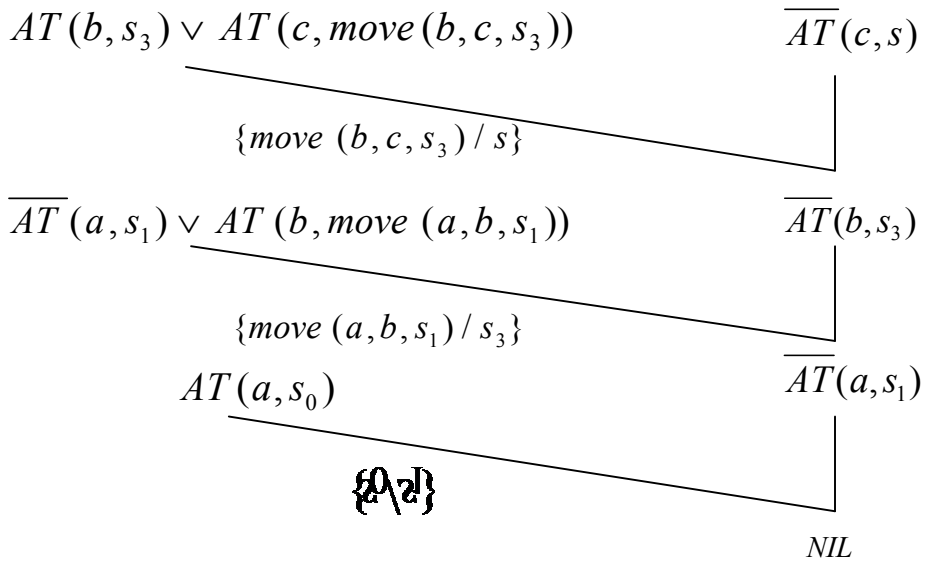


Рисунок 4.7 – Дерево опровержения для задачи поиска пути на графе

4.1.6.3. Автоматическое написание программ [30]. В этом примере продемонстрируем возможность автоматического написания простейшей программы с помощью принципа резолюции.

Рассмотрим задачу сортировки списка чисел *x*. Программа должна формировать упорядоченный по убыванию список чисел *y*. В этом случае можно сказать, что множество входных и выходных чисел удовлетворяет некоторому предикату (отношению) *R(x, y)*. Если интерпретацию предикатной буквы *R* задать некоторым множеством аксиом и воспользоваться элементарными функциями, известными системе, то автоматическое написание программы сводится к тому, чтобы доказать, что предположение $\forall x \exists y R(x, y)$ логически следует из указанных аксиом. В ходе доказательства в переменную *y* будут сделаны необходимые подстановки элементарных функций, композиция которых и будет соответствовать решению задачи.

Для написания программы воспользуемся элементарными функциями языка Лисп *car, cdr, cons, merge*. Вызовы функции *car(x)* и *cdr(x)* обеспечивают выделение соответственно первого элемента списка *x* и оставшейся части списка (хвоста). Функция *cons(x,y)* добавляет элемент *x* в начало списка *y* [1)]. Функция *merge(x,y)* выполняет вставку элемента *x* в упорядоченный список *y* [2)]. Отметим, что

[1)] Здесь мы используем правила записи функций языка исчисления предикатов. Синтаксис языка Лисп требует записи функций в несколько иной форме. Например, *(car x).*См. главу 5.

[2)] Функция *merge* языка Коммон Лисп имеет иной синтаксис вызова

cons(car(x),cdr(x))=x.

Сформулируем аксиомы, формализующие определение предиката *R(x, y)*. Введем дополнительно предикат *S(y)*, который означает, что "список *y* уже упорядочен", и предикат *I(x, y)*, означающий, что "списки *x* и *y* состоят из одних и тех же элементов". Тогда будет верно следующее утверждение

$$\forall x \forall y \{\{R(x,y) \to [S(y) \wedge I(x,y)]\} \wedge \{[S(y) \wedge I(x,y)] \to R(x,y)\}\}.$$

Введем рекурсивные определения предикатов *S(y)* и *I(x, y):*

S(nil) (*nil* обозначает пустой список),
$\forall x \forall y \{S(y) \to S(merge(x,y))\}$,
I(nil, nil),
$\forall x \forall y \forall u \{I(x,y) \to I(cons(u,x), merge(u,y))\}$.

Преобразуем приведенные выше определения к множеству предложений, выполнив тождественные преобразования. Получим

$$\Phi_0 = \{\overline{R}(x,y) \vee S(y), \overline{R}(x,y) \vee I(x,y), \overline{S}(y) \vee \overline{I}(x,y) \vee R(x,y),$$
$$S(nil), \overline{S}(y) \vee S(merge(x,y)), I(nil,nil), \overline{I}(x,y) \vee I(cons(u,x), merge(u,y))\}.$$

Запишем отрицание предположения:

$$\overline{F}_g(x,y) = \exists x \forall y \overline{R}(x,y).$$

Исключив кванторы, получим $\overline{R}(a,y)$, где a – функция Сколема. Доказательство невыполнимости множества формул $\{\Phi_0 \cup \overline{R}(a,y)\}$ выполним по индукции: сначала для списков нулевой длины, а затем, предположив, что оно не выполняется для списка длины $n \geq 0$, проверим доказательство для списка длины *n+1*.

Такая схема доказательства позволяет построить рекурсивное определение функции сортировки списков *sort(x)* и устраняет необходимость введения в рассмотрение предикатов с равенством.

Для списка *x* длины *n*=0 предположение можно записать в виде $\exists y R(nil, y)$. Отрицание предложения запишется в виде $\overline{R}(nil, y)$. Модифицированное дерево доказательства для рассматриваемого случая изображено на рисунке 4.8 Ответное утверждение *R(nil, nil)* соответствует тривиальному результату: если список *x* пустой, то и результирующий список *y* тоже пустой

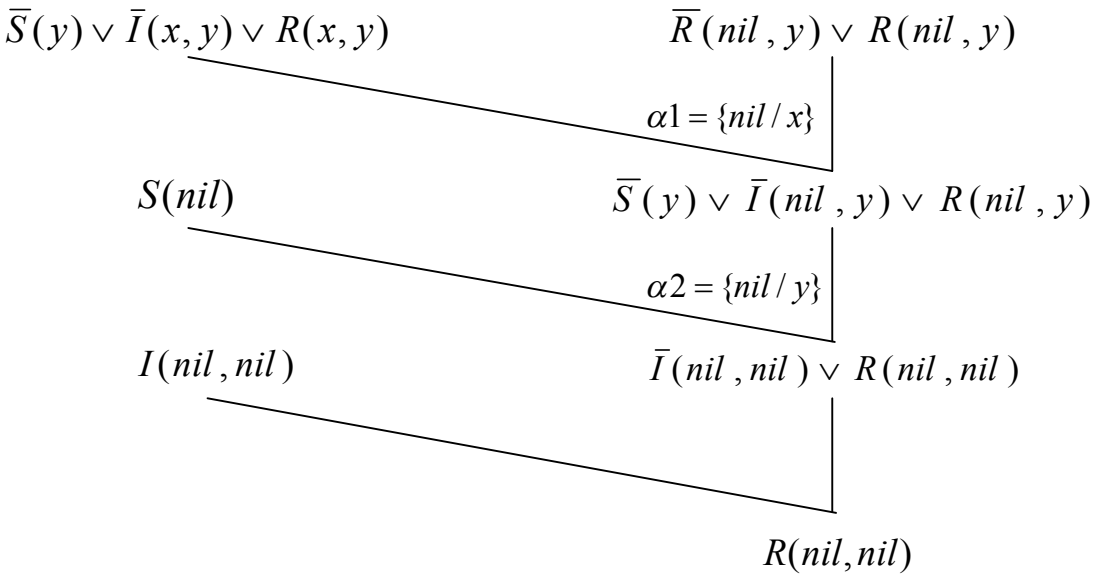


Рисунок 4.8 – Модифицированное дерево доказательства для $\exists y R(nil, y)$

Чтобы выполнить доказательство для случая $n>0$, сформулируем новое дополнительное индуктивное утверждение: “для любого непустого списка x его хвост, получаемый с помощью функции *cdr(x)*, можно отсортировать“. Если допустить, что у нас уже есть функция *sort*, которая может выполнить сортировку списка, то указанное утверждение можно записать в виде дизъюнкта *R(cdr(x),sort(x))*. Добавим его к множеству аксиом Φ_0. Также добавим к множеству Φ_0 очевидное предложение

$$R(cons(car(x), cdr(x)), y) \rightarrow R(x, y),$$

которое после исключения знака импликации примет вид

$$\overline{R}(cons(car(x), cdr(x)), y) \vee R(x, y).$$

Если учесть, что *cons(car(x), cdr (x))=x*, то последнее предложение представляет собой тавтологию, и ее добавление не создает противоречия.

Модифицированное дерево доказательства для предположения $\forall x \exists y R(x, y)$ изображено на рисунке 4.9.

Заменив в ответном утверждении функцию *a* на переменную *x*, получим

R(x, merge(car(x),sort(cdr(x)))).

Объединяя результаты доказательства для $n=0$ и $n>0$, получаем рекурсивную программу упорядочения списка чисел:

$$sort(x) = \begin{cases} nil, & если \quad x = nil, \\ merge(car(x), sort(cdr(x))) & в \quad ином \quad случае. \end{cases}$$

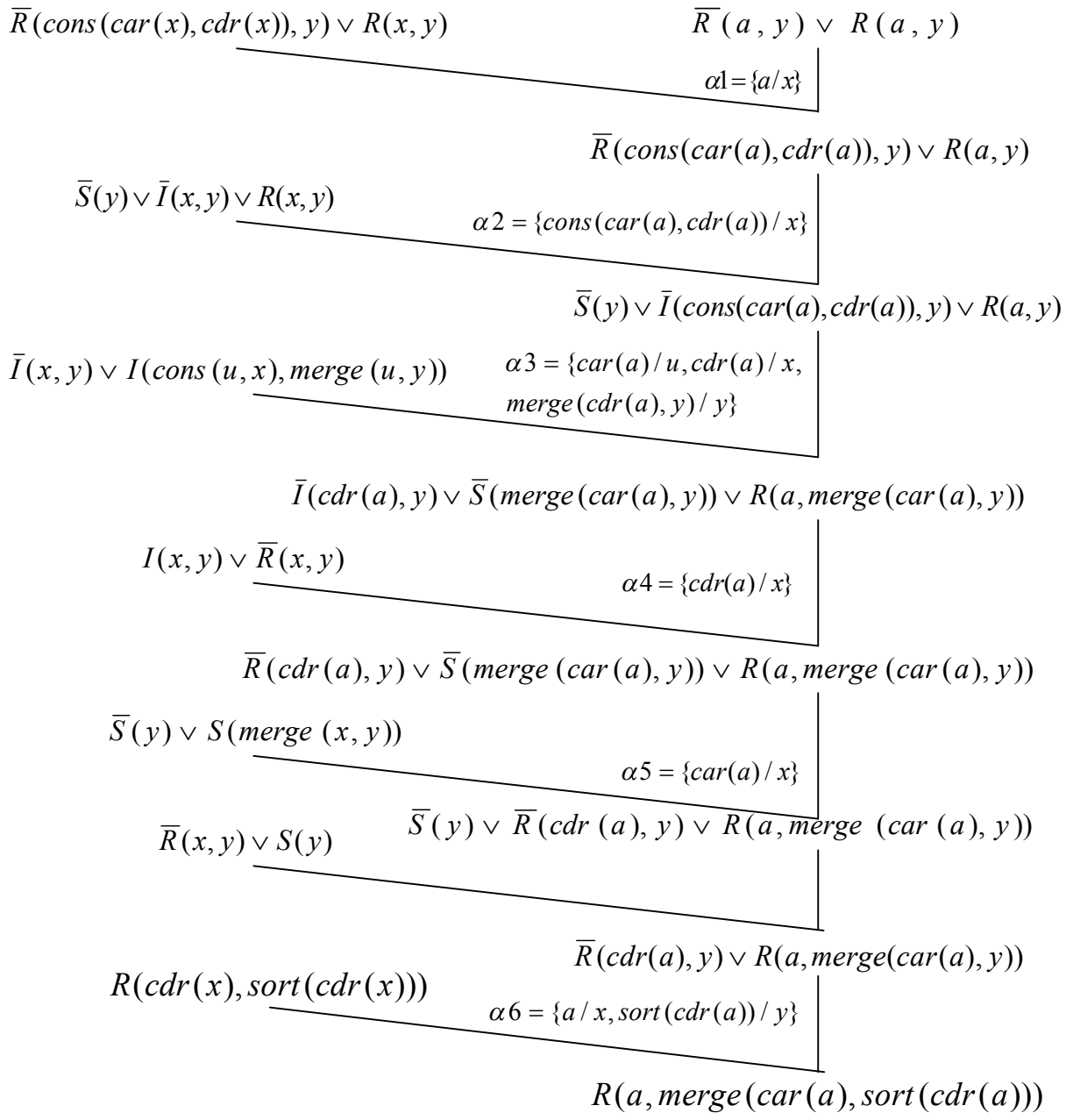


Рисунок 4.9 – Модифицированное дерево доказательства для $\forall x \exists y R(x, y)$

4.1.7. Принцип резолюции и система STRIPS

Система STRIPS (от англ. Stanford Research Institute Problem Solver – решатель задач Стэнфордского исследовательского института) относится к классу решателей задач и представляет собой программу управления роботом, который способен целенаправленно передвигаться в среде с подвижными препятствиями [44]. Задачу перемещения (навигации) робота можно рассматривать как задачу поиска в пространстве состояний. Описание задачи состоит из:

- описания состояния (модели) среды, в которой находится робот;
- описания набора имеющихся операторов, каждый из которых преобразует модель среды;
- описания целевого состояния.

Пространство состояний описывается множеством ППФ. Например, состояние, при котором робот находится в точке *a*, а контейнеры *k1* и *k2* -- в точках *b, c, d* , можно описать с помощью следующих ППФ:

ATR(a);
AT(k1,b);
AT(k2,c);
AT(k3,d).

Операторы представляются в виде операторных схем, которые характеризуются именем, списком параметров и описаниями. Описания операторных схем выполняются на языке исчисления предикатов и включают в себя условия применимости оператора, список вычеркиваний и список добавлений. Например, операторная схема *goto(m,n)* определяет обобщенное действие, связанное с переходом робота из точки *m* в точку *n*. Переменные *m* и *n* рассматриваются как параметры схемы, в которые подставляются константы, задающие возможные позиции робота. Операторная схема порождает конкретные операторы, которые преобразуют среду. Операторная схема *goto(m,n)* может быть применена только в том случае, если робот находится в точке *m*. Поэтому условия ее применимости записываются в виде *ATR(m)*. В результате применения данной операторной схемы робот перейдет из точки *m* в точку *n* . Это приведет к изменению состояния среды. Теперь, во множество ППФ, описывающих состояние среды, необходимо добавить утверждение *ATR(n)* и исключить из него утверждение *ATR(m)*. Следовательно, список добавлений должен содержать предикат *ATR(n)* , а список вычеркиваний – предикат *ATR(m)*. Таким образом, описание рассматриваемой операторной схемы будет следующим:

goto(m,n);
условия применимости: *ATR(m)*;
список вычеркиваний: *ATR(m*);
список добавлений: *ATR(n*).

Необходимо различать параметры операторных схем и обычные квантифицированные переменные, которые также могут входить в операторную схему.

Цель в системе STRIPS также представляется в виде ППФ. Например, если требуется собрать контейнеры *k1, k2* и *k3* в одной точке, то цель может быть выражена в виде следующей ППФ

$$G_0 = \exists x\{AT(k1,x), AT(k2,x), AT(k3,x)\}. \qquad (4.1)$$

Стратегия поиска решения, используемая в системе STRIPS, основана на *методе анализа конечных значений*. Метод позволяет осуществлять целенаправленный поиск решений путём анализа различий текущего и целевого состояний среды. Если различие установлено, то выполняется поиск таких операторов, которые могут уменьшить это различие, т.е. приблизить текущее состояние к целевому состоянию. Впервые метод анализа конечных значений был применен в универсальном решателе задач GPS (General Problem Solver) , предназначенном для решения задач интегрального исчисления, логического вывода, игровых задач [18].

Метод анализа конечных значений используется в системе STRIPS совместно с принципом резолюций. Последний позволяет устанавливать выводимость целевых ППФ и выяснять возможность применения операторов к тому или иному состоянию. При выводе целевых ППФ на основе принципа резолюций образуется незавершенное доказательство, которое и выступает в роли различия между текущим и целевым состояниями. Таким образом, в системе STRIPS процесс доказательства теорем ограничен рамками текущего состояния и отделен от процесса поиска в пространстве состояний, который выполняется в соответствии со стратегией, принятой в GPS.

Представление операторов в системе STRIPS в виде операторных схем привело к необходимости модификации правил унификации. В алгоритме унификации системы STRIPS разрешены следующие виды подстановок:

- вместо переменных могут быть подставлены переменные, константы, параметры и функциональные термы, не содержащие переменных;
- вместо параметров допускается подставлять константы, параметры и функциональные термы, не содержащие функций Сколема, переменных или параметров.

Кроме этого, потребовалось внести дополнение в процедуру доказательства. Пусть *D1* и *D2* два дизъюнкта, которые образуют резольвенту *D*, и $\alpha=\{t/p\}$ некоторая подстановка, используемая при получении *D*, где *t* – терм, поставляемый вместо параметра *p*. Дополнение заключаются в том, что *p* заменяется на *t* во всех дизъюнктах, являющихся производными от *D*.

В системе STRIPS поиск решения начинается с попытки доказательства того, что целевая ППФ G_0 следует из множества формул M_0, определяющих исходное состояние (модель) среды. Для этого система доказывает невыполнимость множества $\{M_0 \cup \overline{G}_0\}$. Если это доказано, то исходное состояние удовлетворяет цели, и задача решена. Если же невыполнимость множества $\{M_0 \cup \overline{G}_0\}$ не установлена, то образуется незавершенный вывод. Этот незавершенный вывод рассматривается как различие между M_0 и G_0. Далее выполняется поиск операторов-кандидатов, способных уменьшить различие. Для этого выбираются операторы, которые так воздействуют на модель среды, что становится возможным продолжение незавершенного доказательства. Продолжить доказательство позволяют те операторы, в списке добавлений которых имеются литералы, дополнительные к литералам незавершенного доказательства. При этом проверяется, может ли процедура доказательства теорем образовывать резольвенты между дизъюнктами незавершенного доказательства и дизъюнктами из списка добавлений оператора. Если этот факт устанавливается, то оператор с соответствующими значениями параметров, рассматривается как подходящий для уменьшения различия. Когда подходящий оператор найден, то условия его применимости выступают в качестве новой подцели системы.

Подцель в системе STRIPS обрабатывается аналогично цели. Иными словами, система теперь пытается установить невыполнимость множества $\{M_0 \cup \overline{G}_1\}$, где G_1 – новая подцель, соответствующая условиям применимости выбранного оператора.

Если на данном шаге устанавливается выводимость ППФ G_1 из M_0, то это означает, что выбранный оператор, например OP_1, может быть применен к состоянию M_0. Применив частный вид оператора OP_1, полученного в ходе доказательства теоремы $M_0 \Rightarrow G_1$, система преобразует состояние M_0 в новое состояние M_1. После этого система вновь попытается установить выводимость G_0, но теперь уже из множества формул, описывающих состояние M_1. Если же в ходе доказательства теоремы $M_0 \Rightarrow G_1$ будет получено незавершенное доказательство, то вновь будет выполняться поиск оператора, обеспечивающего такое изменение M_0, чтобы можно было продолжить доказательство. При этом формируется новая подцель, например G_2 , а подцели G_1 и G_0 отодвигаются на второй план. Указанный процесс повторяется, пока не будет доказана цель G_0.

Описанный процесс показывает, что, до момента получения окончательного решения, неизвестно какое место в этом решении займет тот или иной оператор. Например, OP_1 может быть применен к исходной модели среды M_0 или его применение может быть отложено до момента, когда другой оператор, рассматриваемый после OP_1, преобразует модель к необходимому виду. Такая стратегия сочетает в себе преимущества прямого и обратного поиска. Она дает возможность решить основные подзадачи методом поиска вперед. Затем, чтобы решить вспомогательные подзадачи, возникающие при соединении вместе отдельных частей решения, осуществляется обратный поиск.

Рассмотрим описанные этапы работы системы SRTIPS на примере. Пусть требуется, чтобы робот переместил три контейнера *k1*,*k2* и *k3* в одну точку. Исходная модель среды задана множеством формул:

$$M_0 = \{ATR(a), AT(k1, b), AT(k2, c), AT(k3, d)\}.$$

Целевая формула в этом случае запишется в виде (4.1). Ее отрицание будет иметь вид:

$$\overline{G}0 = \overline{AT}(k1, x) \vee \overline{AT}(k2, x) \vee \overline{AT}(k3, x).$$

Системе доступны следующие операторы: *goto(m,n)* и *push(k,m,n)*. Оператор *goto(m,n)* описан выше, а оператор *push(k,m,n)* обеспечивает перемещение объекта *k* из позиции *m* в позицию *n*. Описание данного оператора задается следующим образом:

push(k,m,n);
условия применимости: $AT(k,m) \wedge ATR(m)$;
список вычеркиваний: *AT(k,m), ATR(m);*
список добавлений: *AT(k,n), ATR(n).*

Решение задачи будем сопровождать построением дерева поиска (рисунок 4.10).

Сначала предпринимается попытка найти противоречие для множества $\{M_0 \cup \overline{G}_0\}$ (вершина (M_0,(G_0)) на рисунке 4.10). Необходимые построения для данного шага изображены на рисунке 4.11.

Образовавшееся неполное доказательство представляет собой различие текущего и целевого состояний среды. Единственным оператором, способным уменьшить данное различие, является *push(k,m,n)*. В его списке добавлений имеется дизъюнкт *AT(k,n)*, который позволяет продолжить доказательство. В зависимости от подстановок, выполняемых в параметры *k* и *n,* возможны шесть вариантов продолжения доказательства. Каждая из подстановок порождает частный вид оператора. Например, подстановка $\alpha_1 = \{k2/k, b/n\}$ дает частный оператор OP_1: *push(k2,m,b)*.

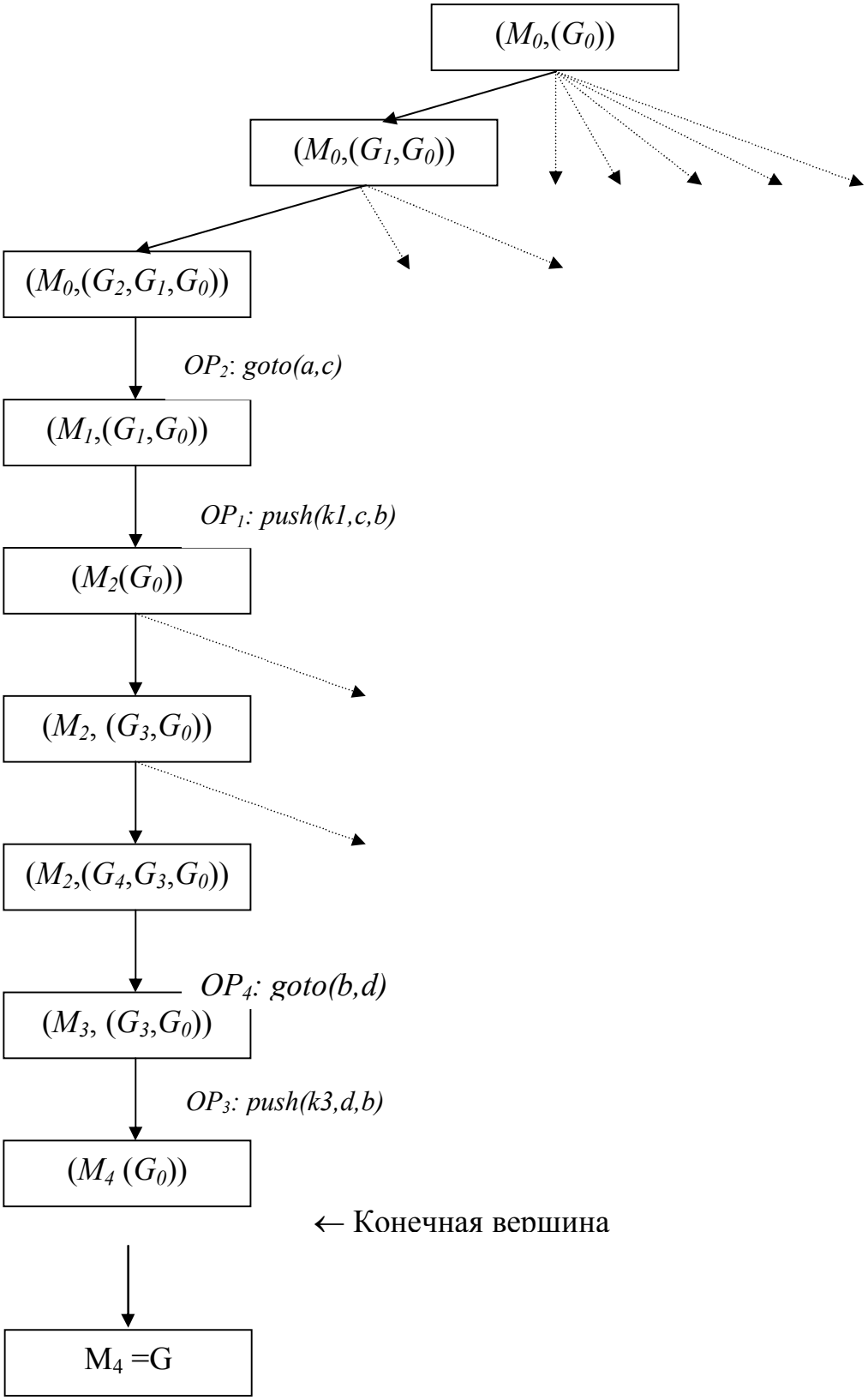


Рисунок 4.10 – Дерево поиска решений в системе STRIPS

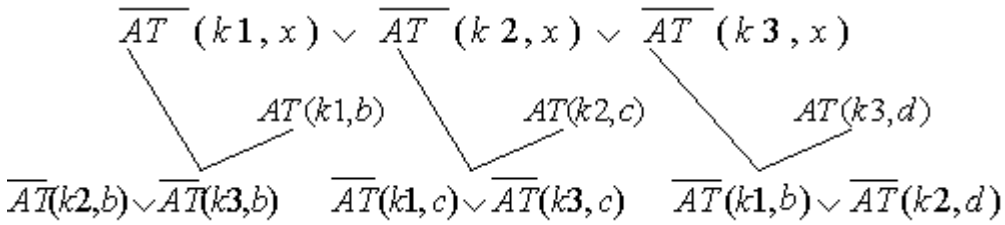


Рисунок 4.11 – Неполное доказательство теоремы $M_0 \Rightarrow G_0$

Оператору OP_1 соответствуют следующее отрицание условий применимости

$$\overline{G}_1 = \overline{AT}(k2,m) \vee \overline{ATR}(m).$$

Если для продолжения решения будет выбран оператор OP_1, то необходимо доказать невыполнимость множества $\{M_0 \cup \overline{G}_1\}$ (если в дальнейшем окажется, что OP_1 не дает решения задачи, то может быть выбран другой частный вид оператора *push*). Соответствующее неполное доказательство представлено на рисунке 4.12.

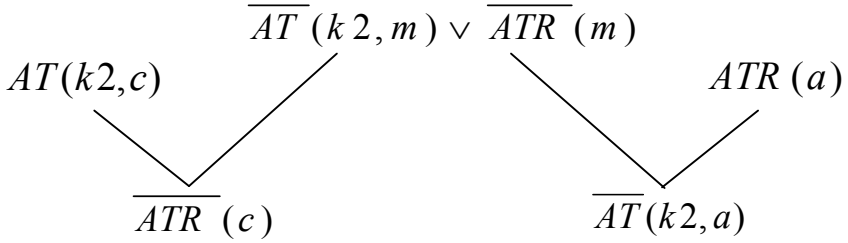


Рисунок 4.12 – Неполное доказательство применимости оператора OP_1

Таким образом, опять образовалось незавершенное доказательство. Одним из подходящих операторов для уменьшения различия будет оператор *goto(m,n)* , так как в его списке добавлений есть предикат *ATR(n)*, который в случае подстановки $\alpha 2=\{c/n\}$ дает резольвенту с предикатом $\overline{ATR}(c)$. Следовательно, частный вид данного оператора можно записать в форме

$$OP_2: goto(m,c).$$

Условие применимости данного оператора выступает в качестве новой подцели. Отрицание подцели G_2 запишется в виде

$$\overline{G}_2 = \overline{ATR}(m).$$

Теперь система пытается установить противоречие для множества $\{M_0 \cup \overline{G}_2\}$. Данная попытка оказывается успешной, так как в M_0 имеется

литерал *ATR(a)* , который при подстановке $\alpha 3=\{a/m\}$ образует с литералом $\overline{ATR}(m)$ контрарную пару. Следовательно, система STRIPS может применить оператор OP_2, имеющий конкретный вид *goto(a,c)*, к состоянию M_0. В результате получается новое состояние

$$M_1 = \{ATR(c), AT(k1,b), AT(k2,c), AT(k3,b)\},$$

и на дереве поиска формируется новая вершина $(M_1,(G_1,G_0))$ (рисунок 4.10). Далее система устанавливает, что множество $\{M_1 \cup \overline{G_1}\}$ является противоречием, если выполняется подстановка $\alpha 4=\{c/m\}$. Поэтому она применяет оператор OP_1 с учетом подстановки $\alpha 4$, т. е. *push(k2,c,b)*. В итоге, получается новое состояние:

$$M_2 = \{ATR(b), AT(k1,b), AT(k2,b), AT(k3,d)\}.$$

На дереве поиска формируется новая вершина *(M2,(G0))*.

Затем система STRIPS вновь пытается вывести целевую формулу G_0, но теперь уже из множества M_2. Дальнейшее решение задачи будет протекать так же, как описано выше. Окончательное решение получается в форме:

{goto(a,c), push(k2,c,b), goto(b,d), push(k3,d,b)}.

В заключение отметим, что в системе STRIPS имеется большее число операторов, и она умеет работать с более сложной моделью среды.

4.1.8. Принцип резолюции и язык Пролог

Рассмотренные ранее принципы доказательства теорем нашли свое воплощение в системе (языке) логического программирования Пролог. Собственно говоря, название языка Пролог составлено из двух слов: *про*граммирование *лог*ическое. В Прологе процесс доказательства теорем рассматривается как процесс вычисления или выполнения программы. Соответствие указанных процессов было установлено Р. Ковальским.

Прежде чем рассмотреть процедуру доказательства теорем, используемую в Прологе, оговорим некоторые элементы синтаксиса языка. Более детальное описание синтаксиса будет приведено позже.

Имена предикатов и констант в Прологе записывают строчными буквами. Имена переменных начинаются с большой буквы или знака подчеркивания. Конкретный список, состоящий из элементов a, b, c, d , записывается на языке Пролог в следующем виде [a,b,c,d]. Список в Прологе может задаваться с помощью переменных, например, [H|T], где H – голова списка, T – хвост списка.

Доказательство теорем в Прологе основано на представлении предложений в форме дизъюнктов Хорна. *Дизъюнкт Хорна* – это предложение, содержащее не более одного положительного литерала. Например, $\overline{P} \vee \overline{Q} \vee \overline{R} \vee S$. В Прологе используется три типа дизъюнктов Хорна:

1) дизъюнкт, состоящий только из одного положительного литерала, например, Q;
2) дизъюнкт, содержащий один положительный и произвольное число отрицательных литералов, например, $Q \vee \overline{P}1 \vee \overline{P}2 \vee ... \vee \overline{P}n$;
3) дизъюнкт, состоящий только из отрицательных литералов, например, $\overline{P}1 \vee \overline{P}2 \vee ... \vee \overline{P}n$.

Дизъюнкты первого типа соответствуют фактам пролог–программы, которые представляют истинные утверждения, например, четное_число(2).

Дизъюнктам второго типа соответствуют правила пролог-программы. Правила описывают утверждения, которые могут быть истинными, если выполняются некоторые условия. Отрицательные литералы дизъюнктов Хорна второго типа формируют условие, а положительный литерал – заключение. Например, приведенный выше дизъюнкт Хорна второго типа соответствует импликации $(P1 \wedge P2 \wedge ... \wedge Pn) \rightarrow Q$, которая на языке Пролог выражается правилом:

```
q:-p1,p2,…,pn.
```

В этом правиле импликации соответствует знак “:-”, а конъюнкции соответствует запятая. Положительный дизъюнкт правила в Прологе называют головой правила, а отрицательные дизъюнкты – телом правила.

Дизъюнкт, содержащий только одни отрицательные литералы, т. е. дизъюнкт Хорна третьего типа, соответствует отрицанию целевого утверждения. Использование дизъюнктов Хорна в системе Пролог позволяет применить линейную входную стратегию выбора дизъюнктов, участвующих в решении задачи.

Рассмотрим применение принципа резолюции в системе Пролог на примере. Для этого обратимся к пролог-программе, проверяющей вхождение элемента H в список:

```
member(H,[H|T]).
member(H,[X|T]):-member(H,T).
```

Здесь member(X,Y) представляет собой предикат, имеющий значение “истина”, если элемент X входит в список Y. Программа состоит из факта member(H,[H|T]) и правила member(H,[X|T]):-member(H,T). С помощью факта задается истинное утверждение о том, что элемент H и список [H|T], голов-

ной элемент которого есть H, находятся в отношении member. Правило member(H,[X|T]):-member(H,T) означает, что элемент H и список [X|T] будут находиться в отношении member , если указанное отношение имеет место между элементом H и хвостом списка T. Иными словами, элемент H содержится в списке [X|T], если он входит в хвост T этого списка.

Рассмотрим процесс выполнения программы на примере целевого утверждения member(c,[a,b,c]), которое на Прологе записывается в виде:

```
?-member(c,[a,b,c]).
```
(4.2)

В этом случае пролог-система будет доказывать невыполнимость множества:

$$S = \{member\ (H,[H\mid T]),$$
$$member\ (H,[X\mid T]\vee \overline{member}\ (H,T]),$$
$$\overline{member}\ (c,[a,b,c])\}.$$

Соответствующее дерево опровержения, реализующее линейную входную резолюцию, изображено на рисунке 4.13.

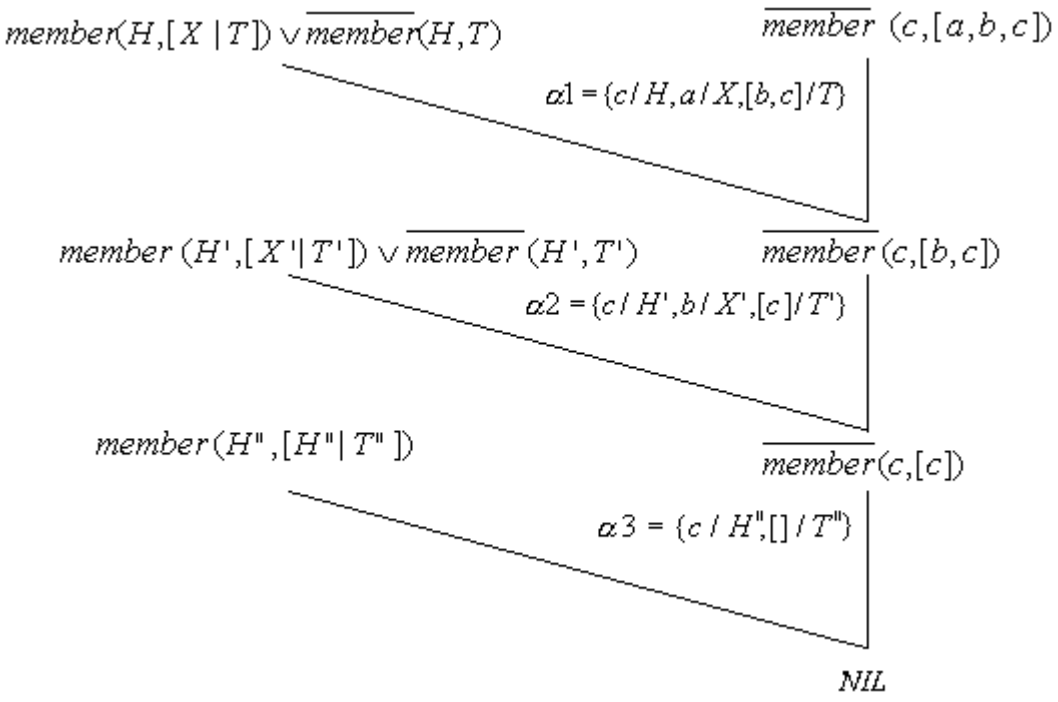


Рисунок 4.13 – Дерево опровержения для отношения member

При унификации литералов $\overline{member}(c,[a,b,c])$ и $member(H,[H\mid T])$ пролог-система обнаруживает, что унифицировать их невозможно. Поэто-

му предпринимается попытка унифицировать предикат $\overline{member}(c,[a,b,c])$ с головой правила

$$member(H,[X \mid T] \vee \overline{member}(H,[H \mid T]).$$

Сопоставление окажется успешным, если выполнить подстановку $\alpha 1 = \{c/H,\ a/X,\ [b,c]/T\}$. Полученная резольвента $\overline{member}(c,[b,c])$ будет новой подцелью. Указанный процесс повторяется до тех пор, пока не будет получено пустое предложение. Необходимо отметить два обстоятельства:

1) на каждом шаге резолюции очередная подцель сопоставляется с положительным литералом либо правила, либо факта;
2) область действия переменных, в которые выполняются подстановки, ограничена одним шагом резолюции (для этого на рисунке использованы имена переменных со штрихом).

Результат выполнения запроса (4.2) возвращается в виде ответа “Да” (Yes).

Рассмотрим еще один пример выполнения пролог-программы, в котором необходимо получить развернутый ответ на поставленный вопрос.

Пусть требуется написать программу, выполняющую соединение двух списков и возвращающую в качестве результата третий список. Определим для этого предикат (отношение) append(X,Y,Z), где X и Y – исходные списки, а Z– результирующий список. При описании отношения append необходимо учесть два случая:

1) если X представляет пустой список, то второй и третий аргумент (т. е. Y и Z) отношения представляют собой один и тот же список, что выражается в виде факта append([ ],L,L);
2) если X не пустой список, то он имеет голову и хвост и может быть записан в виде [H|T]; в результате соединения такого списка со списком Y, получим новый список [H|W], где между хвостом T первого списка, списком Y и хвостом W результирующего должно существовать отношение append(T,Y,W). На Прологе это записывается в виде правила:

```
append([H|T],Y,[H|W]):-append(T,Y,W).
```

Иными словами, списки [H|T], Y, и [H|W] находятся в отношении append, если в этом же отношении находятся списки T,Y и W. Пролог-программа, решающая поставленную задачу, запишется в виде:

```
append([ ],L,L).
append([H|T],Y,[H|W]):-append(T,Y,W).
```

Рассмотрим процесс выполнения этой программы на примере целевого утверждения

$$\exists X append([a,b],[c],X),$$

которое на Прологе запишется в виде:

```
?-append([a,b],[c],X).
```

Результатом выполнения программы будет значение переменной X. Соответствующее дерево опровержения, реализуемое пролог-системой, изображено на рисунке 4.14.

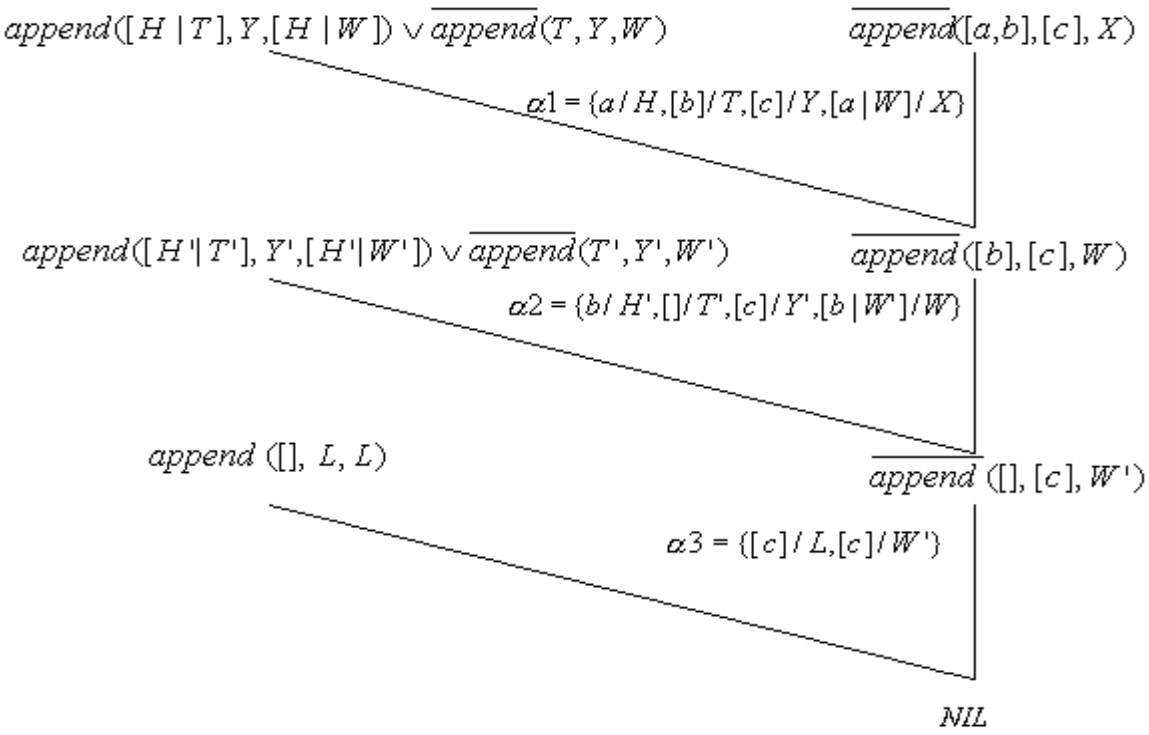


Рисунок 4.14 – Дерево опровержения для отношения append

Значение переменной X определяется композицией подстановок:

$$X \leftarrow [H \mid W] = [a \mid W],$$
$$W \leftarrow [H' \mid W'] = [b \mid W'],$$
$$W' \leftarrow L = [c],$$

т.е.

```
X=[a | [b | [c] ] ]=[a | [b,c] ]=[a,b,c].
```

Обратим внимание на тот факт, что правила унификации, принятые в Прологе, сложнее правил, рассмотренных нами ранее.

4.2. Выводы в условиях ненадежных или неполных знаний

4.2.1. Виды неопределенности

В предыдущих параграфах рассматривался дедуктивный вывод, основанный на исчислении предикатов. Теория исчисления предикатов позволяет получать правильные заключения, опираясь на общезначимые утверждения и обоснованные правила вывода. Однако во многих случаях требуется осуществлять вывод в условиях, когда исходные данные не являются абсолютно точными и достоверными, а правила вывода носят эвристический характер и ненадежны. Человек делает необходимые умозаключения в подобных условиях ежедневно: ставит медицинские диагнозы и назначает лечение, руководствуясь симптомами; выясняет причины плохой работы двигателя по акустическому шуму; правильно понимает обрывки фраз естественного языка; узнает друзей по их голосам и т.п. Знаниям, которыми оперирует человек в указанных случаях, присуща неопределенность. Эта неопределенность имеет самую различную природу. Она может порождаться неполнотой описания ситуации, вероятностным характером наблюдаемых событий, неточностью представления данных, многозначностью слов естественного языка, использованием эвристических правил вывода и др. Наиболее важные виды неопределенности можно представить в виде дерева, изображенного на рисунке 4.15[31].



Рисунок 4.15 – Виды неопределенности

На первом уровне дерева изображены термины, представляющие качественную оценку характера неопределенности. Неопределенность может быть связана либо с неполнотой знаний, либо с их неоднозначностью. *Неполнота знаний* возникает, когда собрана не вся информация, необходимая для построения выводов. *Неоднозначность* означает, что истинность тех

или иных высказываний не может быть установлена с абсолютной достоверностью. Она порождается либо физическими причинами (физическая неопределенность), либо лингвистическими (лингвистическая неопределенность). *Физическая неопределенность* может быть связана со случайностью событий, ситуаций, состояний объекта или неточностью представления данных. *Лингвистическая неопределенность* связана с использованием естественного языка для представления знаний, имеющих качественный характер, и возникает из-за множественности значений слов (полисемия) и смысла фраз. Например, “Двигатель часто перегревается” или “Ганс уже большой”. В этих примерах неоднозначность обусловлена нечеткостью понятий “часто” и “большой”. Или, например, “Он встретил ее на поляне с цветами” [31]. Как он ее встретил: с цветами или без цветов? Такого рода неопределенности присутствуют в системах, на поведение которых в значительной степени влияют суждения человека. При анализе неопределенности смысла фраз выделяют синтаксическую, семантическую и прагматическую неопределенности [31].

Рассмотренная схема видов неопределенности в определенной степени условна. В реальных системах указанные виды неопределенности могут накладываться один на другой. Например, физическая неопределенность может усложняться лингвистической неопределенностью.

Часто указанные выше виды неоднозначности не выделяют в отдельные группы, а рассматривают в рамках одного термина “ненадежные знания”[37]. Основополагающим понятием, используемым при построении моделей вывода на таких знаниях, является понятие достоверности. Достоверность выводов на основе ненадежных знаний может быть определена с помощью различных подходов. Наиболее часто используются: вероятностная байесовская логика, коэффициенты уверенности, нечеткая логика, теория Демпстера-Шефера.

Неполнота знаний приводит к необходимости осуществления немонотонных выводов. Эта проблема упоминалась в §3.2 и §3.5. Для формальной обработки знаний, характеризуемых неполнотой, используются логика умолчания Рейтера, немонотонная логика Мак-Дермотта и Дойла, системы поддержания значений истинности.

4.2.2. Ненадежные знания и выводы

4.2.2.1. Байесовский метод. При байесовском подходе степень достоверности каждого из фактов базы знаний оценивается вероятностью, которая принимает значения в диапазоне от 0 до 1. Вероятности исходных фактов определяют либо методом статистических испытаний, либо опросом экспертов. Вероятности заключений определяют на основе правила Байеса

для вычисления апостериорной условной вероятности $p(H|E)$ события (гипотезы) H при условии, что произошло событие (свидетельство) E

$$p(H \mid E) = \frac{p(E,H)}{p(E)} \quad ,$$

где $p(E)$ – безусловная (априорная) вероятность свидетельства E; $p(E,H)$ – совместная вероятность свидетельства E и гипотезы H. Совместная вероятность $p(E,H)$ равна произведению безусловной вероятности гипотезы $p(H)$ на условную вероятность того, что свидетельство (факт) E имеет место, если наблюдается гипотеза H:

$$p(E,H) = p(H) \cdot p(E \mid H) .$$

Отсюда следует, что апостериорную вероятность $p(H|E)$ можно вычислить с помощью формулы

$$p(H \mid E) = \frac{p(H) \cdot p(E \mid H)}{p(E)} \quad . \tag{4.3}$$

Уточним практическое применение формулы (4.3) на простом примере. Пусть H обозначает некоторое заболевание, а E – симптом. Тогда априорная вероятность заболевания H может быть определена по формуле

$$p(H) = N_H / N,$$

где N_H – количество жителей некоторого региона, имеющих заболевание H; N – количество всех жителей региона. Вероятность $p(E)$ определяется аналогично

$$p(E) = N_E / N,$$

где N_E – количество жителей, у которых наблюдается симптом E. Обычно значения вероятностей $p(H)$ и $p(E)$ выясняют у экспертов.

Вероятность $p(E|H)$ соответствует наличию симптома E у больного с заболеванием H. Ее значение также определяют методом опроса экспертов.

Формула (4.3) справедлива для случая одного свидетельства E и одной гипотезы H. Она позволяет пересчитывать значение вероятности гипотезы H в том случае, когда обнаружено свидетельство E в ее пользу, т.е. получать на основе априорной вероятности $p(H)$ значение $p(H|E)$ апостериорной вероятности. Если рассматривается полная группа несовместных гипотез

$H_1, H_2, \ldots, H_n$ с априорными вероятностями $p(H_1),p(H_2),\ldots,p(H_n)$, то апостериорную вероятность каждой из гипотез при реализации свидетельства E вычисляют с помощью формулы

$$p(H_i \mid E) = \frac{p(E \mid H_i) \cdot p(H_i)}{\sum_{k=1}^{n} p(E \mid H_k) \cdot p(H_k)} \quad , \qquad (4.4)$$

которую называют теоремой гипотез Байеса.

Формула (4.4) позволяет упростить вычисление вероятности гипотезы H при реализации свидетельства E. Так, если рассматривать две несовместные гипотезы H и "$\neg H$" ("*не H*"), то

$$p(H \mid E) = \frac{p(E \mid H) \cdot p(H)}{p(E \mid H) \cdot p(H) + p(E \mid \neg H) \cdot (1 - p(H))} \quad , \qquad (4.5)$$

где $p(E|\neg H)$ – вероятность свидетельства E при условии, что гипотеза H не наблюдается; $1\text{-}p(H)=p(\neg H)$ – априорная вероятность невыполнения гипотезы H. В формуле (4.5), в отличие от (4.3), отсутствует априорная вероятность $p(E)$, которая неудобна с точки зрения экспертной оценки [53].

Введем в рассмотрение отношение

$$D_E = \frac{p(E \mid H)}{p(E \mid \neg H)} \quad ,$$

которое называют *отношением правдоподобия*. Оно характеризует отношение вероятности получения свидетельства E при условии, что гипотеза H верна, к вероятности получения этого же свидетельства при условии, что гипотеза H не верна. Поделив числитель и знаменатель (4.5) на $p(E|\neg H)$, получим

$$\frac{p(H \mid E)}{1 - p(H \mid E)} = D_E \frac{p(H)}{1 - p(H)} \quad . \qquad (4.6)$$

Отношения, записанные в выражении (4.6), называют шансами. *Шансы* – это иная шкала для представления вероятности. Так, отношение

$$O(H) = \frac{p(H)}{1 - p(H)}$$

представляет априорные шансы в пользу гипотезы H, а отношение

$$O(H \mid E) = \frac{p(H \mid E)}{(1 - p(H \mid E))}$$

– апостериорные шансы. Например, если *p(H)=0,3*, то *O(H)=3/7*, т.е. 3 случая “за” и 7 “против”.

Из (4.6) следует, что

$$O(H \mid E) = D_E \cdot O(H). \qquad (4.7)$$

Таким образом, апостериорные шансы весьма просто вычисляются через априорные шансы. Для этого необходимо знать отношение правдоподобия свидетельства *E*. Считается, что использование шансов в соответствии с формулой (4.7) для оценки правдоподобности гипотез проще, чем непосредственное вычисление вероятностей по формуле (4.6). Это объясняется двумя причинами [53]:

1) для многих пользователей использование шкалы шансов выглядит проще;
2) шансы обозначают целыми числами, а их проще вводить при ответах на вопросы системы.

Отметим, что можно получить формулу, симметричную (4.7), позволяющую вычислять апостериорные шансы в пользу гипотезы *H*, если задано, что *E* заведомо ложно

$$O(H \mid \neg E) = N_E \cdot O(H),$$

где N_E – отношение правдоподобия $p(\neg E|H)/p(\neg E|\neg H)$. Следовательно, шансы в пользу некоторой гипотезы *H* можно вычислять, опираясь либо на истинность свидетельства *E*, либо на его ложность. Такой подход используется в экспертной системе PROSPECTOR, применяемой в геологии. При этом отношения правдоподобия D_E и N_E называют соответственно *фактором достаточности* и *фактором необходимости*. Факторы D_E и N_E связаны друг с другом простым соотношением

$$N_E = \frac{1 - D_E \cdot p(E \mid H)}{1 - p(E \mid \neg H)},$$

которое позволяет установить диапазон изменения их значений.

Можно показать, что значения D_E принадлежат диапазону $[1, \infty)$, а значения N_E – диапазону [0, 1]. С каждым правилом в системе PROSPECTOR связывают факторы D_E и N_E, значения которым могут назначаться независимо. Это иногда приводит к противоречиям. Чтобы раз-

решить такие противоречия, фактор D_E используется, когда свидетельство E истинно, а фактор N_E – когда ложно.

На практике гипотеза H может подтверждаться не одним свидетельством, а несколькими. Формула (4.7) легко обобщается на случай нескольких свидетельств. Например, если реализовались два свидетельства E_1 и E_2 в пользу гипотезы H, то ее апостериорные шансы можно вычислить по формуле

$$O(H \mid E_1, E_2) = D_{E_2} D_{E_1} \cdot O(H)$$

или

$$O(H \mid E_1, E_2) = D_{E_1} \cdot O(H \mid E_1).$$

Так как обычно $D_E > 1$, то получение новых свидетельств в пользу гипотезы H позволяет увеличить ее правдоподобие.

В рассмотренных выше формулах предполагалось, что свидетельства $E_1, E_2, \ldots, E_n$ полностью определенны, т.е. подтверждаются с вероятностью 1. Однако на практике факты, используемые в системе, могут подтверждаться с меньшей вероятностью. Это может происходить из-за неопределенных ответов пользователя, а также в результате логического вывода, когда заключения одних правил выступают в роли фактов (свидетельств) других правил. В этом случае необходимо при вычислении правдоподобия той или иной гипотезы учитывать ненадежность фактов. Пусть известно (например, из предыдущих выводов), что свидетельство E подтверждается с вероятностью $p(E|E')$, где через E' обозначено некоторое свидетельство, подтверждающее E. Тогда апостериорную вероятность гипотезы H можно вычислить по формуле [53,54]:

$$p(H \mid E') = p(E \mid E') \cdot p(H \mid E) + (1 - p(E \mid E')) \cdot p(H \mid \neg E).$$

Из формулы следует:

1) если свидетельство E подтверждается с вероятностью $p(E|E')=1$, то $p(H|E')=p(H|E)$;
2) если $p(E|E')=0$ (т.е. свидетельство E не подтверждается), то $p(H|E')=p(H|\neg E)$;

Кроме того, если $p(E|E')=p(E)$ (т.е. свидетельство подтверждается с априорной вероятностью), то значение вероятности не должно измениться: $p(H|E')=p(H)$.

Таким образом, значения $p(H|E')$ лежат в диапазоне от $p(H|\neg E)$ до $p(H)$, если $0 \le p(E \mid E') \le p(E)$ и в диапазоне от $p(H)$ до $p(H|E)$, если $p(E) \le p(E \mid E') \le 1$. Выполнив кусочно-линейную аппроксимацию зависимости $p(H|E')$ от $p(E|E')$, получают формулы [54]:

$$p(H \mid E') = p(H \mid \neg E) + \frac{p(H) - p(H \mid \neg E)}{p(E)} \cdot p(E \mid E'),$$

$$\textit{если } 0 \le p(E \mid E') \le p(E);$$

$$p(H \mid E') = p(H) + \frac{p(H \mid E) - p(H)}{1 - p(H)} \cdot (p(E \mid E') - p(E)),$$

$$\textit{если } p(E) \le p(E \mid E') \le 1.$$

Данные формулы позволяют выполнить коррекцию значения *p(H|E)* в зависимости от вероятности подтверждения свидетельства *E*. В экспертной системе PROSPECTOR при заполнении базы знаний пользователям разрешается вместо вероятности *p(E|E')* задавать коэффициенты (факторы) уверенности, лежащие в диапазоне от –5 до +5. В дальнейшем значения этих коэффициентов пересчитываются в вероятности *p(E|E')* [53,54].

Значение апостериорной вероятности *p(H|E')*, вычисленное с учетом вероятности подтверждения свидетельства *E*, определяет эффективное отношение правдоподобия $D_{E'}$

$$D_{E'} = \frac{p(H \mid E')}{1 - p(H \mid E')} \cdot \frac{1 - p(H)}{p(H)}.$$

Данное отношение позволяет выполнять коррекцию апостериорных шансов в пользу гипотезы *H* [37]:

$$O(H \mid E') = D_{E'} \cdot O(H).$$

Правила логического вывода допускают объединение свидетельств $E_1, E_2, \ldots, E_n$ с помощью логических связок И, ИЛИ, НЕ. В такой ситуации обработка каждого правила выполняется с учетом принципов нечеткой логики. Для свидетельств, связанных логической операцией И, выбирается минимальная из вероятностей. Для логической операции ИЛИ берется максимальная из вероятностей. Операция логического отрицания НЕ приводит к вычислению обратной вероятности.

При использовании байесовского подхода для обработки неточных знаний возникают две основные проблемы. Во-первых, должны быть известны все априорные условные вероятности свидетельств, а также априорные вероятности гипотез (или соответствующие отношения правдоподобия и шансы). Во-вторых, из условий теоремы Байеса следует, что все гипотезы, рассматриваемые системой, должны быть несовместны, а вероятности $p(E|H_i)$ и *p(E)* – независимы. В ряде областей (например, в медицине) последнее требование не выполняется.

Кроме этого, при введении новой гипотезы, необходимо заново пересчитывать таблицы вероятностей, что также ограничивает применимость байесовского подхода. Тем не менее, он достаточно широко применяется в экспертных системах, благодаря хорошему теоретическому фундаменту. Важно, что после многократного применения теоремы Байеса влияние всех исходных допущений на результат становится минимальным. Поэтому, хотя априорные вероятности могут определяться приближенно, их соотношение вычисляется достаточно надежно. Свое дальнейшее развитие рассматриваемый подход получил в байесовских сетях доверия [77].

4.2.2.2. Метод коэффициентов уверенности. Метод коэффициентов уверенности был впервые применен в экспертной системе MYCIN. В отличие от байесовского подхода, который использует теорию вероятностей для подтверждения гипотез, метод системы MYCIN базируется на эвристических соображениях, которые были позаимствованы из практического опыта работы экспертов. Когда эксперт оценивает степень достоверности некоторого вывода, он использует такие понятия, как “точно”, “весьма вероятно”, “возможно”, “ничего нельзя сказать” и т.п. Разработчики системы MYCIN решили отобразить эти размытые понятия на шкалу коэффициентов уверенности, изменяющихся в диапазоне от –1 до +1. Для этого были введены две оценки – *MB* и *MD*. Оценка *MB* отражает степень истинности некоторого факта (свидетельства) и принимает значения от 0 до +1. Оценка *MD* соответствует степени ложности некоторого факта и принимает значения в диапазоне от –1 до 0. Коэффициент уверенности факта, обозначаемый *CF*, определяется как разность оценок *MB* и *MD*:

$$CF = MB - MD\ .$$

Здесь $0<MB<1$ (при $MD=0$) и $-1<MD<0$ (при $MB=0$). Если коэффициент уверенности принимает значение, равное +1, то факт считается истинным. Если $CF= -1$, то факт ложный. Шкала изменения значений коэффициентов уверенности изображена на рисунке 4.16 [53].

В ходе логического вывода над фактами, составляющими предпосылки правил, выполняются логические операции. В результате этого образуются сложные высказывания, коэффициенты уверенности которых вычисляются по следующим правилам:

1) при логической связи И между фактами P_1 и P_2

$$CF(P_1 \wedge P_2) = MIN(CF(P_1), CF(P_2));$$

2) при логической связи ИЛИ между фактами P_1 и P_2

$$CF(P_1 \vee P_2) = MAX(CF(P_1), CF(P_2)).$$

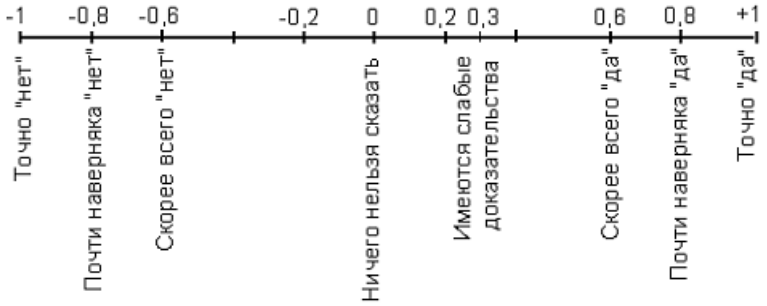


Рисунок 4.16 – Шкала коэффициентов уверенности

В системе MYCIN коэффициенты уверенности приписываются не только фактам, но и правилам. Таким способом обеспечивается учет ненадежности правил, которые часто формируются на основе эвристических соображений. Обозначим коэффициент уверенности правила через CF_R. Коэффициент CF_R соответствует степени истинности заключения правила при истинных предпосылках. Если предпосылки характеризуются коэффициентом уверенности $CF_{\text{пред}} \neq 1$, то коэффициент уверенности заключения $CF_{\text{закл}}$ вычисляют по формуле

$$CF_{закл} = CF_{пред} \cdot CF_R .$$

Рассмотрим пример правила $(P_1 \wedge P_2) \vee P_3 \Rightarrow C_1(0{,}9)$, где P_1, P_2, P_3 – факты, образующие предпосылки правила; C_1 – заключение правила; $0{,}9$ – коэффициент уверенности правила. Пусть факты характеризуются следующими коэффициентами уверенности: $CF(P_1)=0{,}8$; $CF(P_2)=0{,}6$; $CF(P_3)=0{,}7$. Определим коэффициент уверенности заключения $CF(C_1)$:

$$CF(P_1 \wedge P_2) = \min(0{,}8;0{,}6) = 0{,}6;$$
$$CF((P_1 \wedge P_2) \vee P_3) = \max(0{,}6;0{,}7) = 0{,}7;$$
$$CF(C_1) = 0{,}7 \cdot 0{,}9 = 0{,}62.$$

В процессе логического вывода одно и то же заключение может подтверждаться различными правилами, каждое из которых приписывает заключению свой коэффициент уверенности. Очевидно, что коэффициент уверенности заключения, подтверждаемого несколькими правилами должен увеличиться. Чем больше будет иметься подтверждений в пользу некоторого заключения, тем ближе к единице должен быть его коэффициент уверенности. В общем случае комбинация свидетельств в поддержку некоторого заключения выполняется по следующим формулам, используемым в системе EMYCIN [37,54]:

$$CF = CF_1 + CF_2 - CF_1 \cdot CF_2, \text{если} \qquad CF_1 > 0, CF_2 > 0;$$

$$CF = CF_1 + CF_2 + CF_1 \cdot CF_2, \text{если} \qquad CF_1 < 0, CF_2 < 0;$$

$$CF = \frac{CF_1 + CF_2}{1 - \min(|CF_1|, |CF_2|)}, \text{если} \qquad CF_1 \cdot CF_2 \le 0, CF_1 \neq \pm 1, CF_2 \neq \pm 1;$$

$$\text{Если} \qquad CF_1 = \pm 1 \quad \text{и} \quad CF_2 = \mp 1, \quad \text{то} \quad CF = 1.$$

Коэффициенты уверенности заключений, формируемых тремя или более правилами, можно вывести последовательно, применяя указанные выше формулы.

В заключение отметим, что метод коэффициентов уверенности благодаря своей простоте находит широкое применение во многих системах, поддерживающих вывод на ненадежных знаниях. Недостатки метода связаны с отсутствием теоретического фундамента, а также сложностью подбора значений коэффициентов уверенности.

4.2.2.3. Теория свидетельств Демпстера-Шефера. Подход, принятый в теории Демпстера-Шефера (ТДШ), отличается от байесовского подхода и метода коэффициентов уверенности тем, что, во-первых, здесь используется не точечная оценка уверенности (вероятность, коэффициент уверенности), а интервальная оценка. Такая оценка характеризуется нижней и верхней границей, что более надежно. Во-вторых, ТДШ позволяет исключить взаимосвязь между неопределенностью (неполнотой знаний) и недоверием, которая свойственна байесовскому подходу.

В рамках ТДШ множеству высказываний *A* приписывается диапазон значений [*Bl(A),Pl(A)*], в котором находятся степени доверия каждого из высказываний. Здесь *Bl(A)* – степень доверия к множеству высказываний, изменяющая свои значения от 0 (нет *свидетельств* в пользу *A*) до 1 (множество высказываний *A* истинно); *Pl(A)* – степень правдоподобия множества высказываний *A*, определяемая с помощью формулы

$$Pl(A) = 1 - Bl(\overline{A}).$$

Значения степени правдоподобия также лежат в диапазоне от 0 до 1. Степень правдоподобия множества высказываний *A* – это величина обратная степени доверия к множеству противоположных высказываний $\overline{A}$. Если имеются четкие свидетельства в пользу $\overline{A}$, то $Bl(\overline{A})$=1, и степень правдоподобия *A* равна нулю, т.е. *Pl(A)=0*. При *Pl(A)=0* степень доверия *Bl(A)* тоже равна нулю.

В случае отсутствия информации в поддержку той или иной гипотезы полагается, что диапазон изменения значений степеней *Bl* и *Pl* равен [0, 1]. По мере получения свидетельств указанный диапазон сужается. В теории

Демпстера-Шефера неопределенность знаний представляется с помощью некоторого множества *X*. Элементы этого множества соответствуют возможным фактам или заключениям. Неопределенность состоит в том, что заранее неизвестно, какое из возможных значений примет факт или заключение $x \in X$. Для характеристики степени определенности в ТДШ вводится некоторая единичная мера уверенности (ее называют также единичной массой уверенности), которая распределяется между элементами *X*. При этом, если вся масса (степень) уверенности приходится на один элемент $x \in X$, то никакой неопределенности нет. Неопределенность возникает, когда масса уверенности распределяется между несколькими элементами $x \in X$. На рисунке 4.17 изображено распределение масс уверенности между элементами множества *X*, представленного в виде точек. Здесь $X=\{x_1, x_2, x_3\}$.

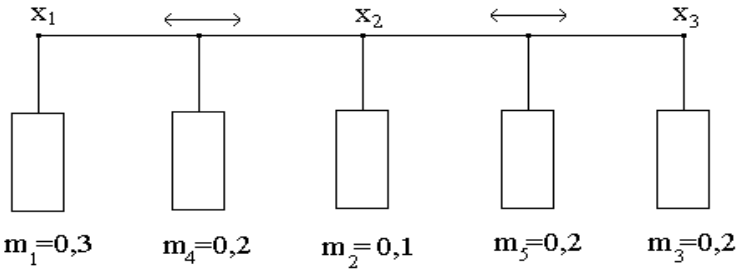


Рисунок 4.17 – Распределение масс уверенности

С каждым элементом множества *X* жестко связана соответствующая масса уверенности. Так, x_1 соответствует m_1=0,3 , x_2 – m_2=0,1, x_3 – m_3=0,2. Имеются также свободные массы уверенности m_4=0,2 и m_5=0,2, которые относятся сразу к нескольким элементам. Масса m_4 свободно перемещается между элементами x_1 и x_2 , а масса m_5 – между элементами x_2 и x_3, т.е. m_4 закреплена за подмножеством $\{x_1, x_2\}$, а m_5 – за подмножеством $\{x_2, x_3\}$. Массы выражают степень уверенности в возможных значениях фактов или заключений. Так, степень уверенности в значении x_1 может изменяться от 0,3 до 0,5. Таким образом, степень незнания соответствует массе, местоположение которой не определено.

В общем случае распределение масс уверенности задается функцией *m(A)*, обладающей следующими свойствами:

$$m(\varnothing) = 0,$$
$$\sum_{A \subseteq X} m(A) = 1.$$

Здесь *A* – множество, образованное из подмножеств *X*, которым назначены соответствующие массы (степени) уверенности; *m(A)* – функция,

которая задает отображение A на интервал [0, 1]. Для примера, изображенного на рисунке 4.17

$$A = \{\varnothing, \{x_1\}, \{x_2\}, \{x_3\}, \{x_1, x_2\}, \{x_2, x_3\}, \{x_1, x_3\}\},$$

а распределение масс уверенности задаётся функцией *m(A),* характеризуемой множеством значений

$$m(A) = \{0;\ 0{,}3;\ 0{,}1;\ 0{,}2;\ 0{,}2;\ 0{,}2;\ 0\}\ .$$

Обратим внимание, что A состоит из подмножеств. Обозначим каждое такое подмножество через A_i. Степень доверия к высказываниям, соответствующим подмножеству A_i, может быть вычислена по формуле

$$Bl(A_i) = \sum_{A_j \subseteq A_i} m(A_j). \qquad (4.8)$$

Здесь суммирование выполняется по всем подмножествам A_j, входящим в A_i. Например,

$$Bl(\{x_1, x_2\}) = m(A_1) + m(A_2) + m(A_4) = m(\{x_1\}) + m(\{x_2\}) + m(\{x_1, x_2\}) =$$
$$= 0{,}3 + 0{,}1 + 0{,}2 = 0{,}6.$$

Соответствующие значения степеней доверия для всех остальных подмножеств A_i представлены в таблице 4.1

Таблица 4.1 – Значения $Bl(A_i)$ и $Pl(A_i)$

A_i	$\varnothing$	$\{x_1\}$	$\{x_2\}$	$\{x_3\}$	$\{x_1,x_2\}$	$\{x_2,x_3\}$	$\{x_1,x_3\}$	X
$Bl(A_i)$	0	0,3	0,1	0,2	0,6	0,5	0,5	1
$Pl(A_i)$	0	0,5	0,5	0,4	0,8	0,7	0,9	1

Степень правдоподобия подмножества A_i определяется по формуле

$$Pl(A_i) = 1 - Bl(\overline{A_i}) = 1 - \sum_{A_j \subseteq \overline{A_i}} m(A_j)\ . \qquad (4.9)$$

Результаты вычислений степеней правдоподобия приведены в таблице 4.1.

Величины $Bl(A_i)$ и $Pl(A_i)$ имеют простую интерпретацию: $Bl(A_i)$ представляет общую массу уверенности, которая остается, если из X удалить все элементы, не ассоциируемые с A_i ; $Pl(A_i)$ представляет максимальную массу уверенности, которую можно получить, если сдвинуть свободные массы к элементам множества A_i. Причем $Bl(A_i) \le Pl(A_i)$. Иными словами, $Bl(A_i)$ представляет нижнюю границу доверия к A_i, а $Pl(A_i)$ – верхнюю.

Важнейшим элементом ТДШ является правило комбинации свидетельств:

$$m(A_k) = \frac{\sum\limits_{A_{1i} \cap A_{2j}} m_1(A_{1i}) \cdot m_2(A_{2j})}{1 - \sum\limits_{A_{1i} \cap A_{2j} = \varnothing} m_1(A_{1i}) \cdot m_2(A_{2j})} \cdot \qquad (4.10)$$

Сумма в числителе правила распространяется на множество $A_k = A_{1i} \cap A_{2j}$. Правило является эвристическим и позволяет осуществлять распределение степеней доверия в ходе вывода [37].

4.2.2.4. Нечеткие множества и нечеткая логика. Для формализации нечетких знаний, характеризуемых лингвистической неопределенностью, применяется теория нечетких или расплывчатых множеств. Основы теории нечетких множеств были созданы в 1965г. американским математиком Л.А.Заде. Пусть имеется множество X и его собственное подмножество A, т.е. $A \subset X$. Тогда подмножество A можно представить в виде совокупности упорядоченных пар

$$A = \{(x, \mu_A(x))\}, \forall x \in X,$$

где $\mu_A(x)$ – функция принадлежности. Причем

$$\mu_A(x) = \begin{cases} 1, если & x \in A, \\ 0, если & x \notin A. \end{cases}$$

Рассмотрим в качестве примера множество, представляющее возможные значения температуры воздуха. Тогда введение функции принадлежности, условно изображенной на рисунке 4.18 сплошной линией, позволит выделить подмножество отрицательных температур, а изображенной пунктирной линией – подмножество положительных температур. При этом нулевая температура относится ко второму подмножеству.

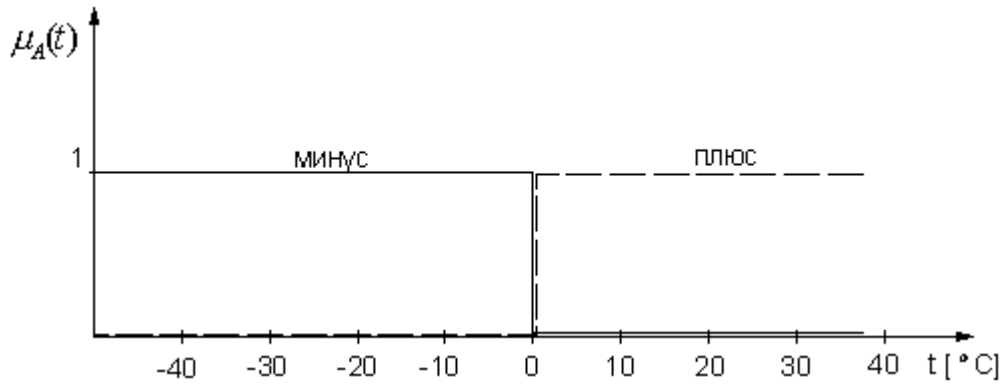


Рисунок 4.18 – Функция принадлежности для четкого множества

Приведенное определение подмножества A и пример функции принадлежности, принимающей всего два возможных значения 0 и 1, соответствует четкому (обычному) подмножеству. Определение нечеткого подмножества получается как обобщение этого определения. *Нечетким подмножеством A* множества X будем называть совокупность упорядоченных пар

$$A = \{(x, \mu_A(x))\}, \forall x \in X, \qquad (4.11)$$

где функция принадлежности $\mu_A(x)$ каждому элементу x ставит в соответствие действительное число из интервала [0,1], указывающие степень принадлежности элемента x подмножеству A. Часто в выражении (4.11) вместо запятой, отделяющей x от $\mu_A(x)$, используется вертикальная черта, т.е. пишут $(x \mid \mu_A(x))$. Это позволяет исключить возможную путаницу, когда значения функции принадлежности задаются числами, в которых дробная часть отделяется от целой запятой.

Математическая структура, определяемая выражением

$$A = \{(x_1 \mid 0{,}8), (x_2 \mid 0{,}3), (x_3 \mid 0), (x_4 \mid 0{,}5)\},$$

где x_1, x_2, x_3, x_4 – элементы универсального множества X, представляет собой пример нечеткого подмножества. Здесь степени принадлежности элементов x_1, x_2, x_3, x_4 подмножеству A заданы числами после вертикальной черты. Наивысшую степень принадлежности имеет элемент x_1. Элемент x_3 подмножеству A не принадлежит[1)]. Элементы x_2 и x_4 принадлежат A в меньшей степени, чем x_1. Таким образом, используя понятие нечеткого подмножества, можно представлять объекты (сущности) предметной области, характеризуемые размытыми границами описаний.

Нечеткое подмножество A называется *нормальным*, если

$$\max_{x \in X} \mu_A(x) = 1.$$

Если это свойство не выполняется, то A называется *субнормальным*. Непустое подмножество A всегда можно нормализовать делением на максимальное значение функции принадлежности. *Носителем* нечеткого подмножества A называется подмножество элементов X, для которых $\mu_A(x) > 0$. Переменная x в выражении (4.11) называется *базовой*. В дальнейшем, говоря о нечетких подмножествах, будем опускать приставку “под”.

При практическом применении нечетких множеств важными являются понятия нечеткой и лингвистической переменной. *Нечеткая переменная*

[1)] Обычно элементы с принадлежностью 0 не записываются

определяется набором $< \alpha, X, R >$, где α – название нечеткой переменной; $X=\{x\}$ – область ее определения; R – нечеткое подмножество множества X, включающее те возможные значения x, которые обусловлены названием α. *Лингвистической* называют переменную, значениями которой являются слова и фразы естественного языка (термы). Формально лингвистическая переменная определяется набором [31]

$$< \beta, T, X, G, M >,$$

где β – название переменной; T – терм-множество переменной β, т.е. множество ее значений, причем каждое из таких значений (терм) есть название нечеткой переменной α, областью определения которой является множество X; G – синтаксическая процедура, порождающая на множестве T значения переменной β; M – семантическая процедура, которая отображает каждое новое значение нечеткой переменной α в нечеткое подмножество $M(\alpha)$ множества X.

Рассмотрим пример. Пусть определяется температура окружающего воздуха с помощью понятий-значений: “очень холодно”, “холодно”, “свежо”, “не холодно”, “тепло”, “жарко”. Каждому понятию-значению соответствует определенная температура (рисунок 4.19). При этом минимальная и максимальная температуры соответственно равны – 40°С и +40°С. Тогда формально лингвистическая переменная “температура” представляется набором

< ТЕМПЕРАТУРА, {ОЧЕНЬ ХОЛОДНО, ХОЛОДНО, СВЕЖО,
НЕ ХОЛОДНО, ТЕПЛО, ЖАРКО}, [-40,+40], *G,M* >,

где G – процедура перебора элементов терм-множества T; M – процедура, которая ставит в соответствие нечетким переменным (“жарко”, “тепло” и т.д.) температуру окружающего воздуха. Иными словами, процедура M определяет взаимосвязь нечетких переменных с соответствующими функциями принадлежности.

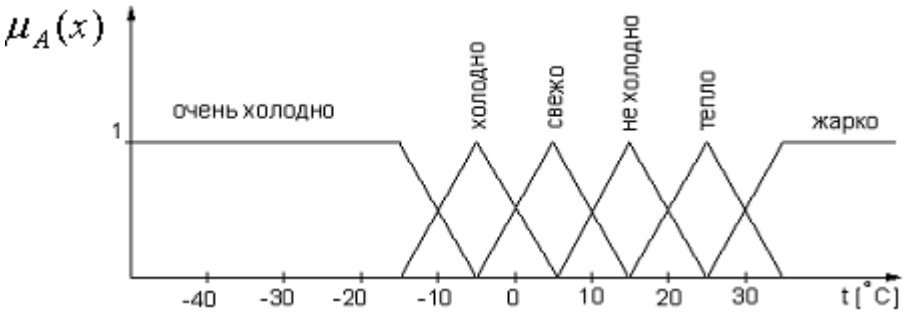


Рисунок 4.19 – Лингвистическая переменная “температура”

В зависимости от характера множества X лингвистические переменные подразделяются на числовые и нечисловые. Числовой называется такая лингвистическая переменная, для которой базовая переменная определенна на числовом множестве. В рассмотренном выше примере лингвистическая переменная “ТЕМПЕРАТУРА” является числовой, а ее значения (нечеткие переменные) – нечеткими числами. Каждой нечеткой переменной на рисунке 4.19 условно соответствует треугольная или трапецеидальная функция принадлежности, которая определяет степень принадлежности той или иной температуры окружающего воздуха к соответствующему нечеткому понятию. Так, руководствуясь рисунком, нечеткую переменную “не холодно” можно определить с помощью следующего нечеткого множества

$$\textit{“не холодно”}=\{(5|0),(15|1,0),(25|0)\}.$$

В общем случае построение функции принадлежности осуществляют на основе экспертных оценок. Соответствующие процедуры построения функций принадлежности описаны в [31,36].

Нечеткие и лингвистические переменные позволят формулировать нечеткие утверждения. Например, “Сегодня не холодно” или “Если тепло и влажно, то овощи растут быстро” и т.п. Подобные утверждения являются объектом исследований нечеткой логики. Важно отметить, что в отличие от формальной логики, которая использует таблицы истинности, нечеткая логика опирается на операции, выполняемые над нечеткими множествами. Поэтому рассмотрим основные операции нечетких множеств.

Будем рассматривать нечеткие подмножества A и B множества X. Нечеткие множества A и B *равны* (обозначается $A=B$), если $\forall x \in X$ $\mu_A(x) = \mu_B(x)$. Множество B *содержится* в множестве A, если $\forall x \in X$ $\mu_B(x) \le \mu_A(x)$. Множества A и B *дополняют* друг друга, если $\forall x \in X$ $\mu_B(x) = 1 - \mu_A(x)$. Множество с функцией принадлежности $1 - \mu_A(x)$ есть *дополнение* к множеству A (обозначается $\overline{A}$).

Пересечение двух нечетких множеств A и B (обозначается $A \cap B$) – это нечеткое множество, функция принадлежности которого определяется выражением

$$\mu_{A\cap B}(x) = \min\{\mu_A(x), \mu_B(x)\}, \forall x \in X.$$

Объединением нечетких множеств A и B (обозначается $A \cup B$) есть нечеткое множество с функцией принадлежности

$$\mu_{A\cup B} = \max\{\mu_A(x), \mu_B(x)\}, \forall x \in X.$$

Операции пересечения и объединения нечетких множеств ассоциативны и дистрибутивны. В теории нечетких множеств часто используют обозначения $\mu_{A\cap B}(x) = \mu_A(x) \wedge \mu_B(x)$ и $\mu_{A\cup B}(x) = \mu_A(x) \vee \mu_B(x)$. Здесь $\wedge$ и $\vee$ соответствуют *min* и *max*.

Алгебраическое произведение нечетких множеств *A* и *B* (обозначается *AB*) есть нечеткое множество, для которого

$$\mu_{AB}(x) = \mu_A(x) \cdot \mu_B(x), \forall x \in X .$$

Алгебраическая сумма нечетких множеств *A* и *B* (обозначается $A \hat{+} B$) есть нечеткое множество с функцией принадлежности

$$\mu_{A\hat{+}B}(x) = \mu_A(x) + \mu_B(x) - \mu_A(x) \cdot \mu_B(x).$$

Дизъюнктивная сумма нечетких множеств *A* и *B* определяется через операции объединения и пересечения

$$A \Delta B = (A \cap \overline{B}) \cup (\overline{A} \cap B) = (A \setminus B) \cup (B \setminus A),$$

где выражения $A \setminus B = A \cap \overline{B}$ и $B \setminus A = B \cap \overline{A}$ представляют соответствующие разности множеств *A* и *B*. В общем случае $A \setminus B \neq B \setminus A$.

Важным понятием теории нечетких множеств является *нечеткое отношение*. Нечетким бинарным отношением $P : X_1 \to X_2$ называется подмножество декартового произведения двух множеств X_1 и X_2

$$P = \{(x_1, x_2) \mid \mu_P(x_1, x_2)\}, \forall x_1 \in X_1, \forall x_2 \in X_2,$$

где $\mu_P(x_1, x_2)$ – функция принадлежности пары элементов (x_1, x_2) к *P*.

В общем случае n-арное нечеткое отношение *P* определяется на прямом произведении множеств $X_1 \times X_2 \times \ldots \times X_n$ с помощью формулы

$$P = \{(x_1, x_2, \ldots, x_n) \mid (\mu_P(x_1, x_2, \ldots, x_n)\},$$

где $x_1, x_2, \ldots, x_n$ – элементы множеств $X_1, X_2, \ldots, X_n$. Сопоставляя определение нечеткого множества и нечеткого отношения, можно заметить, что нечеткое отношение – это нечеткое множество, базовая переменная которого есть вектор. В качестве примера приведем бинарное нечеткое отношение. Пусть множества *X* и *Y* состоят из элементов $X=\{1, 2\}$, $Y=\{1{,}1\ ,\ 2{,}1\ ,\ 5\}$. То-

гда нечеткое бинарное отношение можно задать матрицей, элементами которой будут значения $\mu_P(x_i, y_i)$:

$$P = \begin{bmatrix} 0.9 & 0.1 & 0 \\ 0.1 & 0.9 & 0 \end{bmatrix} \begin{matrix} x_1 = 1 \\ x_2 = 2 \end{matrix}$$
$$y_1 = 1{,}1 \quad y_2 = 2{,}1 \quad y_3 = 5$$

В частности, данная матрица может представлять нечеткое отношение "*X* примерно равно *Y*".

Часто нечеткие отношения используют для представления правил типа "если *A*, то *B*", где *A* и *B* нечеткие подмножества ($A \subset X_1, B \subset X_2$). Такое правило будем обозначать $A \Rightarrow B$. Один из способов задания нечеткого отношения, соответствующего правилу $A \Rightarrow B$, состоит в использовании формулы *декартового произведения* множеств *A* и *B* (обозначаемого $A \times B$):

$$P = A \times B = \{(x_1, x_2) \mid \mu_A(x_1) \wedge \mu_B(x_2)\}, \forall x_1 \in X_1, \forall x_2 \in X_2.$$

Здесь

$$\mu_A(x_1) \wedge \mu_B(x_2) = \min\{\mu_A(x_1), \mu_B(x_2)\} = \mu_P(x_1, x_2),$$

причем $(x_1, x_2) \in X_1 \times X_2$.

Рассмотрим пример [37]. Пусть $X_1 = X_2 = \{1, 2, 3, 4\}$. Определим нечеткое множество *A*, соответствующее малым числам, и нечеткое множество *B*, соответствующее большим числам:

$$A = \{(1 \mid 1), (2 \mid 0{,}6), (3 \mid 0{,}1), (4 \mid 0)\};$$
$$B = \{(1 \mid 0), (2 \mid 0{,}1), (3 \mid 0{,}6), (4 \mid 1)\}.$$

Тогда правилу "если x_1 малое, то x_2 большое" будет соответствовать нечеткое отношение

$$P = A \times B = \begin{bmatrix} 0 & 0{,}1 & 0{,}6 & 1 \\ 0 & 0{,}1 & 0{,}6 & 0{,}6 \\ 0 & 0{,}1 & 0{,}1 & 0{,}1 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} x_1$$
$$\begin{matrix} 1 & 2 & 3 & 4 \end{matrix}$$
$$x_2$$

Так как нечеткие отношения – это нечеткие множества, то к ним можно применять рассмотренные выше операции нечетких множеств. Кроме этого, над нечеткими отношениями выполняют ряд дополнительных операций. Наиболее важной является операция композиции.

Пусть заданы два нечетких отношения: $P \subset X \times Y$ и $R \subset Y \times Z$, где X,Y,Z – четкие множества. Тогда *композиция* отношений P и R (обозначается $P \circ R$) определится с помощью выражения

$$P \circ R = \{((x,z) \mid \max_{y \in Y} \min\{\mu_P(x,y), \mu_R(y,z)\})\} \quad (x,z) \in X \times Z$$

или

$$P \circ R = \{(x,z) \mid \bigvee_y [\mu_P(x,y) \wedge \mu_R(y,z)]\} \quad (x,z) \in X \times Z ,$$

где символы $\bigwedge_y$ и $\bigvee_y$ используются соответственно для обозначения максимума и минимума относительно переменной y. Операцию композиции называют также *max-min сверткой*. Max-min свертка позволяет установить нечеткое отношение элементов множеств X и Z, минуя отношение с элементами из множества Y. Она реализует правило: “если элементы из X находятся в отношении P к элементам из Y, а элементы из Y находятся в отношении R к элементам из Z, то элементы из X находятся в отношении $P \circ R$ к элементам из Z”. Данное правило является составным и, по сути, состоит из двух правил, реализуемых в ходе нечеткого вывода. Поэтому композиция нечетких отношений соответствует цепочке правил, реализуемых в ходе нечеткого вывода.

В нечетких выводах участвуют нечеткие утверждения, над которыми выполняются операции нечеткой логики. В рамках *нечеткой логики* существует два подхода к определению степени истинности нечетких утверждений. В первом случае степень истинности нечеткого утверждения (нечеткого предиката) представляется числом из диапазона [0, 1]. Во втором случае степень истинности характеризуется лингвистическими значениями типа: “истинно”, “ложно”, “почти истинно”, “почти ложно” и т.п. Каждое такое значение, по сути, есть нечеткая переменная, представляемая нечетким подмножеством единичного интервала. Логика с лингвистическими значениями истинности была предложена Заде и называется *лингвистической*. Заметим также, что иногда логику первого типа называют *многозначной*, а лингвистическую логику – нечеткой [31].

Основным объектом многозначной логики является нечеткое логическое выражение, в состав которого входят нечеткие предикаты. Нечеткий предикат $P(x_1,x_2,\ldots,x_n)$ ставит в соответствие конкретному набору нечетких переменных $x_1 \in X_1, x_2 \in X_2, \ldots, x_n \in X_n$, превращающих предикат в выска-

зывание, степень истинности $\mu(P)$ из диапазона [0, 1]. Крайние значения этого диапазона соответствуют понятиям “ложь” и “истина” четких предикатов. Степень истинности сложного нечеткого высказывания, образованного из предикатов P_1 и P_2 с помощью операций конъюнкции, дизъюнкции и отрицания, может быть определена с помощью формул

$$\mu(P_1 \wedge P_2) = \min\{\mu(P_1), \mu(P_2)\}, \tag{4.12}$$

$$\mu(P_1 \vee P_2) = \max\{\mu(P_1), \mu(P_2)\}, \tag{4.13}$$

$$\mu(\overline{P_1}) = 1 - \mu(P_1), \tag{4.14}$$

где P_1, P_2 – нечеткие предикаты. Степень истинности более сложных нечетких утверждений получается последовательным применением формул (4.12), (4.13), (4.14).

Нечеткие выводы в рамках многозначной логики могут выполняться с помощью нечетких условных правил “если *A*, то *B*”. В этом случае истинностное значение заключения определяется по формуле

$$\mu(B) = \min\{\mu(A), \mu_{правила}\},$$

где $\mu_{правила}$ – истинностное значение, приписанное правилу. Если некоторое заключение подтверждается несколькими правилами, то степень истинности может быть определенна на основе формулы для нечеткой операции ИЛИ. В многозначной логике обобщен метод резолюций, что позволяет автоматизировать рассмотренный процесс вывода.

Нечеткий вывод, выполняемый в рамках лингвистической логики, сложнее. Операции лингвистической логики, по сути, есть операции над нечеткими множествами. Так, нечеткая конъюнкция и дизъюнкция определяются в виде, соответственно: $A \cap B$ и $A \cup B$. Нечеткое отрицание соответствует отрицанию $\overline{A}$, нечеткая импликация есть $\overline{A} \cup B$, нечеткая эквиваленция определяется с помощью формулы $(A \cup \overline{B}) \cap (\overline{A} \cup B)$ [24].

В некоторых случаях полезной может оказаться операция “компенсирующее И”, объединяющая в себе свойства конъюнкции и дизъюнкции. Функция принадлежности для этой операции задается следующей формулой

$$\mu_{A \cap B}(x) = (\mu_A(x) \cdot \mu_B(x))^{1-\gamma} \cdot (1 - (1 - \mu_A(x)) \cdot (1 - \mu_B(x)))^{\gamma},$$

где γ – параметр, заданный на интервале [0, 1]. При малых значениях γ “компенсирующее И” соответствует конъюнкции, при значениях близких к единице – дизъюнкции. Операция “компенсирующее И” не обладает ассо-

циативным свойством. Мотивировку введения данной операции поясним на примере. Пусть имеется правило:

Если Температура $\geq 870^{o}$C и Давление ≥ 40 бар,
То Подача_газа=0,3.

При возникновении опасности подача газа должна сокращаться. Так, если измеренные значения температуры и давления будут равны 871 ^{o}C и 41 бар, то правило будет активизировано, и подача газа сократится. Если же температура будет равна 868 ^{o}C, а давление составит 62 бара, то ничего не произойдет. В то же время здравый смысл подсказывает, что значительное повышение давления компенсирует пониженную температуру и, следовательно, подачу газа необходимо снижать. С этой целью разумно операцию конъюнкции заменить операцией "компенсирующее И", связав с параметром γ температуру. При понижении температуры конъюнкция может автоматически переходить в дизъюнкцию.

В повседневной жизни часто используются приближенные выводы на основе нечетких исходных высказываний. Например,

Если температура *низкая*, то время нагрева *большое*.
Температура *слишком низкая.*

Время нагрева слишком *большое*. (4.15)

Для формализации такого вывода Заде предложил *обобщенное* правило modus ponens в форме:

посылка 1: если *x* есть *A*, то *y* есть *B,* иначе *y* есть *C*
посылка 2: *x* есть *A′*

следствие: *y* есть *D*,

где *A, A′, B, C, D* – нечеткие переменные, представленные нечеткими множествами, заданными на следующих областях определения U,U,V,V,V; x,y – лингвистические переменные. Это правило имеет существенную особенность по сравнению с традиционным правилом modus ponens

$$\frac{A, A \to B}{B}.$$

В обобщенном правиле множества A и A' не совпадают. Если A и A' более или менее сопоставимы, то можно получить следствие D, в некоторой степени подобное B, иначе D в некоторой степени подобно C. Ранее было отмечено, что для формализации правила $A \Rightarrow B$ можно использовать нечеткое отношение, определяемое с помощью декартового произве-

дения $A \times B$. По аналогии с этим, для формализации первой посылки обобщенного правила вывода, Заде предложил два композиционных отношения [2]. Одно из них называется максиминным и имеет вид

$$R_m = (A \times B) \cup (\overline{A} \times C).$$

Здесь нечеткое отношение $A \times B$ соответствует первой части правила, (т.е. "если *A*, то *B*"), а нечеткое отношение $\overline{A} \times C$ – второй части правила (т.е. "если не *A*, то *C*"). В частном случае, когда альтернативная ветвь правила отсутствует, отношение R_m сводится к отношению $R = A \times B$, которое называют отношением минимума.

Следствие *D* обобщенного правила вывода можно вывести из посылки 1 и посылки 2 на основе max-min свертки нечеткого множества A' и R_m [2]:

$$D = A' \circ R_m = A' \circ [(A \times B) \cup (\overline{A} \times C)] =$$
$$\{v \mid \vee_u \{\mu_{A'}(u) \wedge [(\mu_A(u) \wedge \mu_B(v)) \vee ((1 - \mu_A(u)) \wedge \mu_C(v))]\}\}. \quad (4.16)$$

Рассмотрим пример. Найдем следствие правила (4.15). Пусть множество *U* соответствует температурам, а множество *V* – времени нагрева: U={30 40 50 60} [^{o}C], V={30 60 120 240} [мин].

Введем нечеткие переменные *A* (низкая температура), A' (слишком низкая температура) и *B* (большое время нагрева):

$$A = \{(30 \mid 1{,}0), (40 \mid 0{,}8), (50 \mid 0{,}2), (60 \mid 0)\},$$
$$A' = \{(30 \mid 1{,}0), (40 \mid 0{,}2), (50 \mid 0), (60 \mid 0)\},$$
$$B = \{(30 \mid 0), (60 \mid 0{,}1), (120 \mid 0{,}8), (240 \mid 1{,}0)\}.$$

Для правила (4.15) следствие *D* определяется с помощью формулы

$$D = A' \circ [A \times B].$$

Определим произведение $A \times B$:

$$A \times B = \begin{bmatrix} 0 & 0{,}1 & 0{,}8 & 1{,}0 \\ 0 & 0{,}1 & 0{,}8 & 0{,}8 \\ 0 & 0{,}1 & 0{,}2 & 0{,}2 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{matrix} u_1 \\ u_2 \\ u_3 \\ u_4 \end{matrix}.$$
$$v_1 \quad v_2 \quad v_3 \quad v_4$$

Отсюда

$$\mu_D(v) = [1{,}0 \quad 0{,}2 \quad 0 \quad 0] \circ \begin{bmatrix} 0 & 0{,}1 & 0{,}8 & 1{,}0 \\ 0 & 0{,}1 & 0{,}8 & 0{,}8 \\ 0 & 0{,}1 & 0{,}2 & 0{,}2 \\ 0 & 0 & 0 & 0 \end{bmatrix} = [0 \quad 0{,}1 \quad 0{,}8 \quad 1{,}0].$$

Таким образом, D={(30|0),(60|0,1),(120|0,8),(240|1,0)}, т.е. время нагрева действительно большое.

В заключение отметим, что в лингвистической логике существует несколько вариантов построения нечетких выводов. Рассмотренный вариант, основанный на применении максиминного отношения (4.16), не всегда совпадает с нашей интуицией [2].

4.2.3. Неполнота знаний и немонотонный вывод

Относительно рассмотренных выше методов оценки достоверности заключений высказываются критические замечания. Во-первых, все рассмотренные методы используют количественные меры неопределенности. При этом приходится довольствоваться субъективными оценками функций принадлежности, априорных вероятностей, коэффициентов уверенности. Возникает проблема определения достоверности самих этих оценок. Во-вторых, многие исследователи полагают, что количественные оценки неопределенности несвойственны рассуждениям человека. Если спросить эксперта, почему данное заключение характеризуется неопределенностью, то, как правило, ответ формулируется в терминах качественных взаимосвязей, существующих в предметной области. В-третьих, количественные оценки неопределенности не обеспечивают выполнение правдоподобных рассуждений в условиях неполноты (отсутствия) знаний. Безусловно, с отсутствующими элементами знаний можно связать низкую оценку их достоверности. Но что делать, если в ходе дальнейшего вывода те или иные элементы знаний, на основе которых строился вывод, будут подкреплены достаточно надежными свидетельствами, или, наоборот, признаны ложными? Иными словами, как быть, если вновь полученные утверждения противоречат исходным посылкам логического вывода? Ответ на этот вопрос дают методы немонотонной логики, применяемой в условиях неполноты знаний.

Системы логического вывода, основанные на классической логике, ограничиваются формализацией общезначимых рассуждений. При этом должны выполняться следующие требования. Во-первых, должно быть заданно множество аксиом и правил вывода, которые позволяют делать за-

ключения из предпосылок. Во-вторых, полученные результаты не опровергаются дальнейшими выводами. В-третьих, количество общезначимых формул (теорем) монотонно возрастает.

Объектом немонотонной логики являются модифицируемые утверждения, которые не являются общезначимыми в классическом смысле, т.е. такие утверждения подтверждаются не во всех интерпретациях. Это означает, что утверждения, формируемые правилами вывода немонотонных логик, всего лишь *выполнимы* (подтверждаются хотя бы в одной интерпретации) при некоторых предположениях, оговариваемых в посылках правил. Такие утверждения называют *правдоподобными*. При поступлении в систему новых сведений правдоподобные утверждения могут стать невыполнимыми и будут отвергнуты. В этом случае количество утверждений в ходе вывода может не только возрастать, но и уменьшаться. Поэтому система, формализующая такую схему вывода, становится *немонотонной*. Немонотонные системы логического вывода позволяют получать различные несовместные множества правдоподобных утверждений. При этом такие множества выводимых утверждений характеризуют с помощью “неподвижных точек”. *Неподвижная точка* – это множество предположений, на основе которых нельзя вывести никакое новое правдоподобное утверждение.

Предположения, выполняемые в ходе немонотонного вывода, позволяют устранить неполноту знаний. В базах знаний основанных на классической логике, определяются исключительно верные факты. Неопределенные факты считаются заведомо ложными. Данное предположение называют *гипотезой умолчания замкнутого мира*. Например, подобный подход используется при определении предиката not языка Пролог. Человек часто использует иной подход к оценке достоверности недостающих фактов. Он обычно считает некоторое предположение верным, пока не будет доказано обратное. Иными словами, человек выполняет выводы, опираясь на согласованность принятого предположения с уже выведенными утверждениями. Рассмотрим, например, правило “если *x* птица, то *x* летает”. Здесь по умолчанию принято предположение, что обычно птицы летают. Отметим, что заключение “*x* летает” будет лишь выполнимым, а не общезначимым. Оно верно не при всех интерпретациях *x*, а только при тех, когда выполнимо принятое предположение.

Для формализации указанных рассуждений в немонотонной логике вводятся специальные правила, дополненные условиями применения. Условия применения (или предусловия) фиксируют принятые предположения и позволяют проверить выполнимость (согласованность, непротиворечивость) предположений (до применения правила) совместно с уже выведенными утверждениями. Данные правила называются *правилами немонотонного вывода*.

В немонотонных логиках предусловия в правилах вывода записывают с помощью модальных операторов[1]. Одним из таких операторов является оператор unless (англ. “пока не”). Рассмотрим правило

$$P(x)\ unless\ Q(x) \Rightarrow R(x) \quad . \tag{4.17}$$

Правило позволяет из посылки *P(x)* вывести заключение *R(x)*, пока не доказано предположение *Q(x)*. Если будут получены сведения о выполнимости *Q(x)*, то необходимо будет исключить *R(x)* из базы знаний. С этой целью системы немонотонного вывода должны сохранять протокол вывода.

Идея введения в правила вывода предположений получила дальнейшее развитие в логике умолчаний Рейтера. При этом предположения, используемые в процессе вывода, берутся такими, какими они бывают в обычных ситуациях, т.е. их значения принимаются по умолчанию. Правила умолчаний записываются в форме

$$\frac{\alpha : M\beta}{\gamma} . \tag{4.18}$$

Здесь *M* обозначает модальный оператор “согласуется с” (выполнимо). Двоеточие отделяет посылку правила от предположения. Смысл правила следующий: если верно α и если β согласуется со всеми значениями, полученными к данному моменту, то выполнимо γ. Рассмотренное ранее правило “если *x* птица, то *x* летает” можно записать в виде следующего правила умолчания

$$\frac{птица\ (x) : M\ \ летает\ (x)}{летает\ (x)} .$$

Оно означает, что если *x* – птица и выполним предикат *летает(x)*, то выводимо “*x* летает”.

Использование правил умолчания позволяет получать различные несовместные множества значений. Например, пусть верно *A* и

$$\frac{A : MB}{C}, \quad \frac{A : MC}{B} .$$

[1] Модальные операторы были введены в логику для формализации конструкций естественного языка: “возможно”, “необходимо”, “разрешено”, “обязательно” и др. Соответствующая логика называется модальной логикой.

Тогда либо *B*, либо *C* выводимы из *A*, но не оба сразу. Следовательно, *B* и *C* представляют собой множество правдоподобных *расширений* исходных утверждений *A*. Каждое такое расширение рассматривается как один из "возможных миров", который можно представить, исходя из принятых предположений.

В логике умолчаний совокупность правил умолчания и формул называют *теорией с умолчаниями*. В общем случае правило умолчания (или просто умолчание) – это выражение вида [27]

$$d = \frac{\alpha(\mathbf{x}) : \mathrm{M}\beta_1(\mathbf{x}), \ldots, \mathrm{M}\beta_m(\mathbf{x})}{\gamma(\mathbf{x})},$$

где $\alpha(\mathbf{x}), \beta_1(\mathbf{x}), \ldots, \beta_m(\mathbf{x}), \gamma(\mathbf{x})$ – формулы языка исчисления предикатов первого порядка; $\mathbf{x}=(x_1, \ldots, x_n)$ – предметные переменные; $\alpha(\mathbf{x})$ – требование умолчания; $\beta_i(\mathbf{x})$ – обоснование умолчания; $\gamma(\mathbf{x})$ – следствие умолчания. Умолчание *d* называется *замкнутым*, когда формулы $\alpha(\mathbf{x})$, $\beta_1(\mathbf{x}), \ldots, \beta_m(\mathbf{x})$, $\gamma(\mathbf{x})$ не содержат свободных переменных. Свободные переменные формул в умолчании считаются связанными квантором общности. Если в формулах имеются свободные переменные, то умолчание называется *открытым*. Теория с умолчаниями задается парой

$$\Delta = (D, F),$$

где *D* – множество умолчаний; *F* – множество замкнутых формул исчисления предикатов.

Теория с умолчаниями позволяет получить некоторое число расширений. Расширения существуют не для всех теорий с умолчаниями. Расширения возможны только для замкнутых теорий, которые получаются путем конкретизации открытых умолчаний и формул. Кроме этого, существование расширений гарантируется, если умолчания теории являются нормальными. *Нормальными* называются умолчания, у которых обоснование и следствие совпадают. Общая формула нормального умолчания

$$\frac{\alpha(x) : M\beta(x)}{\beta(x)}.$$

Нормальное умолчание гарантирует получение, по меньшей мере, одного расширения. При этом нормальные умолчания не применимы, когда ложность их следствий уже доказана. Поэтому они не могут опровергать вы-

полнимость следствий ранее примененных нормальных умолчаний. В силу этого, нормальные теории относятся к полумонотонным.

Рейтер построил процедуру доказательств для замкнутых нормальных теорий. Процедура позволяет для заданной замкнутой формулы выяснить, входит ли эта формула в одно из расширений замкнутой нормальной теории [27,37]. В общем случае проблема проверки существования расширения с заданной замкнутой формулой не является полуразрешимой.

В ряде случаев применение нормальных умолчаний может приводить к нежелательным заключениям. Например, опишем ситуацию, изображенную на рисунке 3.18, нормальной теорией $\Delta = (D, F)$, где D содержит два нормальных умолчания: “если x птица, то x летает”, или в символьном виде

$$\frac{птица(x) : M\ летает(x)}{летает(x)};$$

“если x пингвин, то x не летает”

$$\frac{пингвин(x) : M\ \overline{летает}(x)}{\overline{летает}(x)}.$$

Множество формул F запишется в виде:

$$\{птица(Чилли - Вилли), пингвин(Чилли - Вилли)\}.$$

В зависимости от порядка применения правил из D можно получить два следствия: *летает*(*Чилли-Вилли*) и $\overline{летает}$*(Чилли-Вилли)*. Очевидно, первое следствие не соответствует действительности, чтобы исключить возможность применения первого правила разумно потребовать его блокирования, если выполняется предикат *пингвин(x)*. Перепишем первое правило в виде:

$$\frac{птица(x) : M\ летает(x) \wedge \overline{пингвин}(x)}{летает(x)}.$$

Тогда теория Δ дает только одно следствие $\overline{летает}$*(Чилли-Вилли)*.

Последнее правило называется *полунормальным*. В общем случае оно записывается в виде:

$$\frac{\alpha(x) : M\beta(x) \wedge \gamma(x)}{\beta(x)}.$$

Такое умолчание явно управляется условием, задаваемым формулой $\gamma(x)$. Теории с полунормальными умолчаниями не всегда обладают расширени-

ем и, следовательно, им не свойственна полумонотонность. Однако полунормальные умолчания позволяют отображать в базах знаний наследование свойств с исключениями, что характерно для сетевого и фреймового представлений знаний.

Иной подход предложили Д. Мак-Дермотт и Дж. Дойл. Они создали формальную систему немонотонного вывода, расширив логику исчисления предикатов. Для этого они переопределили понятие формула, включив в него выражение вида *Mp*, где *M* – модальный оператор "выполнимо", *p* – формула языка исчисления предикатов. Добавили к системе аксиом исчисления предикатов схемы модальных аксиом и, кроме этого, расширили множество правил вывода специфическим правилом немонотонного вывода, т.е. правилом вывода всего лишь выполнимых утверждений. Выбирая различные подмножества из списка схем модальных аксиом, можно получать различные немонотонные логики. Эти логики позволяют получать различные множества заключений (неподвижных точек) путем изменения порядка вывода. Например, пусть дано множество формул [27,37]

$$S = \{MA \to \overline{B}, MB \to \overline{A}\},$$

где A и B – пропозициональные буквы. Тогда возможны два решения (две неподвижных точки). Одна неподвижная точка содержит $\overline{A}$, но не включает $\overline{B}$. Другая содержит $\overline{B}$, но не включает $\overline{A}$. Действительно, если первая точка не содержит $\overline{B}$, то она содержит *MB*. Тогда из правила modus ponens $MB(MB \to \overline{A})$ следует $\overline{A}$. Рассуждая аналогично, получаем для второй точки $\overline{B}$. Таким образом, решение в данном случае определяется неоднозначно.

Ценность немонотонных логик Мак-Дермотта и Дойла заключается в распространении модальной логики на формализацию модифицируемых утверждений. Однако при этом остаются серьезные проблемы, связанные с обоснованием выбора системы модальных аксиом [27].

Одной из важных задач, возникающих в системах немонотонного вывода, является управление выводом в условиях постоянного изменения степени достоверности заключений. Например, если на основе утверждения *R* получено заключение *S*, а в дальнейшем установлено, что *R* более не выполнимо, то необходимо не только исключить из базы знаний *R* , но также и *S* , если не имеется дополнительных утверждений в поддержку *S*. Для поддержания непротиворечивости базы знаний применяются специальные механизмы, реализуемые системами поддержания значений истинности (англ. Truth Maintenance System – TMS).

Системы поддержания значений истинности (СПЗИ) применяются для обеспечения логической целостности выводов, формируемых на основе неполных знаний. Один из простых способов решений этой задачи заключается в запоминании протокола вывода. В протоколе вывода с каждым заключением связано соответствующее обоснование. Обоснование включает в себя все факты, правила и предположения, сделанные при выводе заключения. Если то или иное обоснование предположения становится невыполнимым, то, используя процедуру возврата к точкам, в которых принимались альтернативные решения, можно заново оценить истинность тех или иных заключений. В простейшем случае процедура осуществляет слепой *хронологический поиск* точек возврата. Однако, если пространство поиска большое, то хронологический поиск становится неэффективным. Для повышения эффективности применяют *поиск, управляемый причиной противоречия.* В этом случае обеспечивается немедленный возврат к предложению, являющемуся источником противоречия. Перемещаясь в прямом направлении от невыполнимого предположения к последующим заключениям, сделанным на его основе, удаляют эти заключения. При этом имеется возможность для каждого заключения, подлежащего удалению, выполнить проверку его выполнимости в новом окружении. Если имеются предположения в поддержку того или иного заключения, то оно не удаляется из базы знаний.

Известны два метода построения систем поиска, управляемого причиной противоречия, которые соответственно реализуются в системах типа JTMS (justification based TMS – система поддержания значений истинности, использующая обоснования) и ATMS (assumption based TMS – система поддержания значений истинности, использующая предположения). Система JTMS была предложена Дж. Дойлом. Дойл был первым исследователем, которому удалось отделить СПЗИ от механизма вывода. При этом СПЗИ взаимодействует с механизмом вывода: получает вновь сформированные следствия и их обоснования и возвращает сведения об утверждениях, которые достоверны на данном шаге работы механизма вывода.

Все утверждения в JTMS делятся на достоверные и недостоверные. Достоверные утверждения отмечаются меткой IN, а недостоверные – меткой OUT. При этом утверждения представляются в виде сети обоснования достоверности. Такая сеть содержит узлы двух типов. Узлы первого типа соответствуют утверждениям, а узлы второго типа – логической связке “И”. Связка “И” позволяет отобразить в сети взаимосвязь предпосылки и предположения правила умолчания (4.18). Анализируя сеть, можно выяснить достоверность того или иного утверждения и установить все предпосылки и предположения, которые подтверждают (поддерживают) его дос-

товерность. Если при добавлении новых знаний возникает противоречие, то происходит пересмотр меток утверждений и модификация сети.

В системе ATMS не применяют бинарные метки IN (достоверно) и OUT (недостоверно). Здесь метка достоверности утверждения представляется в виде множества предположений (предпосылок), при которых данное утверждение верно. Это делает системы типа ATMS более гибкими по сравнению с системами типа JTMS [77].

В заключение отметим, что процесс доказательства в рамках неклассических логик значительно сложнее поддается автоматизации. Главным образом это связано с комбинаторным взрывом. Основной подход к решению этой проблемы состоит в сокращении пространства поиска. Так, во многих модальных логиках ограничиваются анализом “высказывательных фрагментов”, которые формируются с помощью модальных операторов.

Другой подход к автоматизации доказательства в неклассических логиках основан на сведении их к классическим логикам. Это выполняется путем формализации семантики неклассических логик терминами традиционной логики. Например, семантика модальных логик выражается посредством системы “возможных миров”. Для их формализации каждый предикат классической логики снабжается дополнительным аргументом, который специфицирует “возможный мир”. Дальнейшая квантификация таких аргументов с помощью кванторов общности и существования позволяет представить соответствующие модальные операторы. Связи между “возможными мирами” выражаются посредством отношений между ними и учитываются в алгоритмах унификации. Это позволяет автоматизировать процесс доказательства с помощью процессора резолюций.

4.3. Индуктивный вывод

Кроме дедуктивных рассуждений, рассмотренных в § 4.1, в системах искусственного интеллекта широко используют индуктивные схемы рассуждений. Такие схемы рассуждений позволяют порождать обобщения имеющихся частных утверждений. Способность к обобщению является важной функцией естественного интеллекта. В результате обобщения формируются новые знания. Поэтому индуктивные схемы рассуждений находят применение в подсистемах автоматического приобретения знаний. При этом процесс получения новых знаний рассматривается как обучение ИС. В ходе обучения, благодаря обобщению совокупности имеющихся фактов, формируются понятия и выводятся (восстанавливаются) общие правила, обеспечивающие отнесение объектов, событий, ситуаций к соответствующим классам. Рассмотрим основные понятия индуктивного вывода и процедуры индуктивного формирования понятий по примерам.

4.3.1. Основные понятия индуктивного обобщения

Индукцией называют умозаключение, которое представляет знание о классе предметов, полученное в результате исследования отдельных представителей этого класса [13]. Если дедуктивный вывод строится по принципу движения мысли от общего к частному, то индуктивный вывод определяется как логический переход от частного к общему, т.е. как обобщение.

Обобщение здесь понимается как процесс получения знаний, которые объясняют имеющиеся факты. Целью обобщения является формирование понятий.

Понятие – обобщенная информация о множестве объектов (ситуаций, событий), представленных наборами значений признаков, которая:

а) отражает характерные для этого множества логические отношения между отдельными значениями признаков;

б) является достаточной для различения с помощью некоторого правила распознавания (классификации) объектов, принадлежащих множеству, от объектов, не принадлежащих ему [14].

Таким образом, понятие включает лишь существенные признаки объектов, ситуаций, событий, объединяющих их в один класс, и не включает те признаки, которые индивидуализируют их.

Задачу индуктивного формирования понятий можно определить следующим образом. Дано множество фактов (наблюдений, данных) F, представляющих специфические знания о некоторых объектах (ситуациях, событиях и т.п.); множество ограничений, которым должно удовлетворять искомое понятие. Требуется сформировать понятие H (описание, гипотезу), из которого выводимы все факты, т.е. $H \Rightarrow F$, где отношение $\Rightarrow$ понимается шире, чем отношение логического следования в исчислении предикатов.

Пусть D_i – множество значений i – го признака, $i = 1, 2, \ldots, n$; n – число признаков. Признаковому описанию понятий соответствуют точки n -мерного пространства признаков $D = \{D_1 \times D_2 \times \ldots \times D_n\}$. Если каждому из значений признаков поставить в соответствие переменную, принимающую значение 1 или 0, в зависимости от того, наблюдается или не наблюдается данное значение признака, то можно описывать понятия выражениями булевой алгебры. *Конъюнктивным* называется понятие, описываемое конъюнкцией значений признаков. Запись понятия в виде логического выражения представляет собой правило классификации. Если логическое выражение, представляющее понятие будет иметь значение 1, то наблюдаемый объект (ситуация, событие) обобщается данным понятием. Поэто-

му задачу индуктивного формирования понятий на основе анализа конкретных фактов часто рассматривают как задачу восстановления правил.

Наблюдаемые факты называют *положительными* по отношению к некоторому понятию, если они описываются данным понятием, иначе их называют *отрицательными* фактами данного понятия.

Задачу индуктивного формирования понятий можно рассматривать как задачу обучения. Выделяют обучение с учителем (обучение по примерам) и обучение без учителя (обучение на основе наблюдения).

В первом случае совокупность фактов F имеет вид обучающей выборки, которая представляет собой множество примеров объектов (ситуаций, событий) с заданной принадлежностью к тому или иному классу (понятию) K_1, K_2, …, K_m. В основе обучения лежит упорядоченный отбор конъюнкций значений признаков, характеризующих группы положительных примеров понятия и не определяющих ни одного из отрицательных примеров (контрпримеров). Требуется построить решающее правило, позволяющее определить принадлежность произвольного объекта, ситуации, события, заданному классу K_i.

В случае обучения без учителя априорное разделение фактов по классам отсутствует. Требуется по тем или иным критериям выделить совокупность классов $\{K_i\}$ и построить для них решающие правила. Такая задача сложнее задачи обучения с учителем. Методы формирования понятий по фактам без их априорного разбиения на классы рассматриваются в кластерном анализе. Один из алгоритмов группирования объектов, характеризуемых числовыми признаками, в класс (кластер) приведен в § 8.2. Здесь же ограничимся рассмотрением обучения по примерам.

Центральная роль в процессе обучения отводится индуктивному обобщению. При этом различают полную и неполную индукцию [14]. Полной индукцией называют общее заключение, которое выводится на основании изучения всех имеющихся фактов. Это возможно лишь в тех случаях, когда количество возможных фактов конечно и невелико. Если заключения выводятся на основании изучения лишь части фактов, то индукция называется неполной. Заключения, выполненные с помощью неполной индукции, являются правдоподобными, так как из истинности частного не обязательно следует истинность общего.

В основе индуктивных выводов лежит правило индуктивного обобщения

$$\frac{F, H \Rightarrow F}{H}, \tag{4.19}$$

где F – множество известных фактов; H – понятие (гипотеза). Смысл этого правила заключается в следующем. Пусть имеется множество фактов F.

Если ввести некоторую гипотезу *H* и показать, что из *H* выводим любой факт *F*, то гипотеза *H* верна. Особенностью правила индуктивного обобщения (4.19) является то, что множество объектов, описываемых гипотезой *H,* шире, чем множество, соответствующее фактам *F*. Это может приводить к тому, что из гипотезы *H* будут выводиться и другие факты. Следовательно, есть риск совершить ошибку. Поэтому при выборе гипотезы *H* необходимо стремиться к минимальному обобщению.

В заключение отметим, что задача индуктивного формирования понятий близка к задаче обучению распознаванию образов. И в том и в другом случае формируется модель класса объектов, ситуаций, событий. Однако в случае формирования понятий, полученная модель должна обеспечивать не только распознавание, но и возможность генерации описаний конкретных объектов, ситуаций и т.п. [13].

Это связано с пониманием процесса формирования понятий как процесса выделения закономерностей, приобретения новых знаний, где должны отражаться признаковые, структурные и логические характеристики объектов, ситуаций, событий реального мира. Учитывая указанную специфику, ниже рассмотрим процедуры индуктивного обобщения, применяемые в подсистемах приобретения знаний. Процедуры обучения, ориентированные на применение в задачах распознавания образов, рассматриваются в главе 8.

4.3.2. Алгоритм Уинстона

В 1975 году П. Уинстоном была разработана программа, формирующая описания структурных понятий. Программе последовательно предъявляются описания положительных и отрицательных примеров (контрпримеров). В качестве контрпримеров используются примеры, в которые специально внесены ошибки. На рисунке 4.20 изображены положительные и отрицательные примеры для понятия “арка”.

Программа строит описание понятия в виде семантической сети, структура которой уточняется по мере предъявления примеров. При этом “соседние” примеры должны минимально отличаться друг от друга. Обучение программы выполняется на основе операций обобщения и специализации.

В случае предъявления программе описания арки, состоящей из трех прямоугольных блоков, формируется семантическая сеть, представленная на рисунке 4.21,а . Следующий пример арки, в котором верхний блок заменен пирамидой, описывается семантической сетью, изображенной на рисунке 4.21,б.

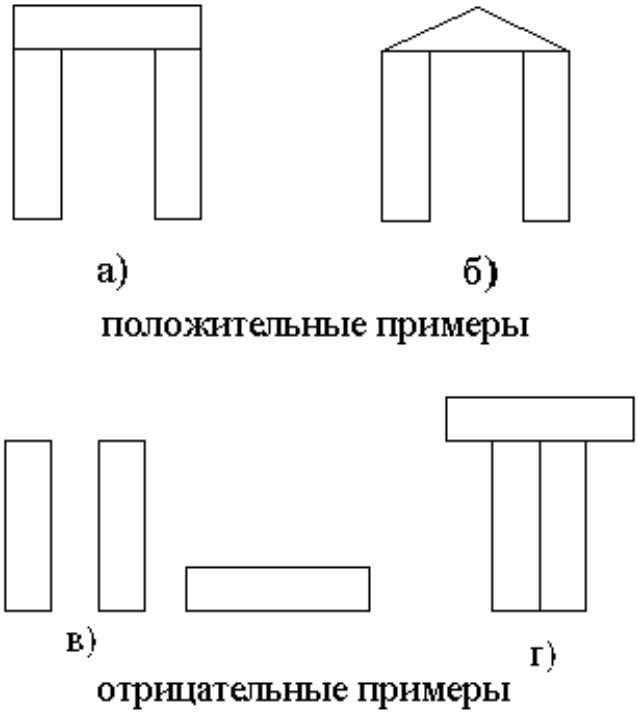
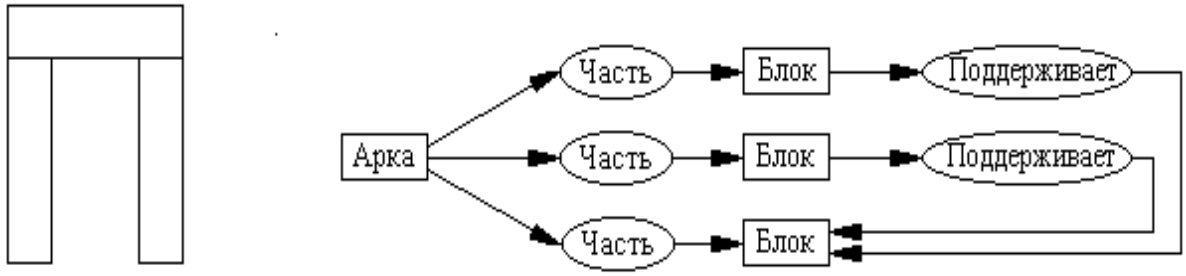
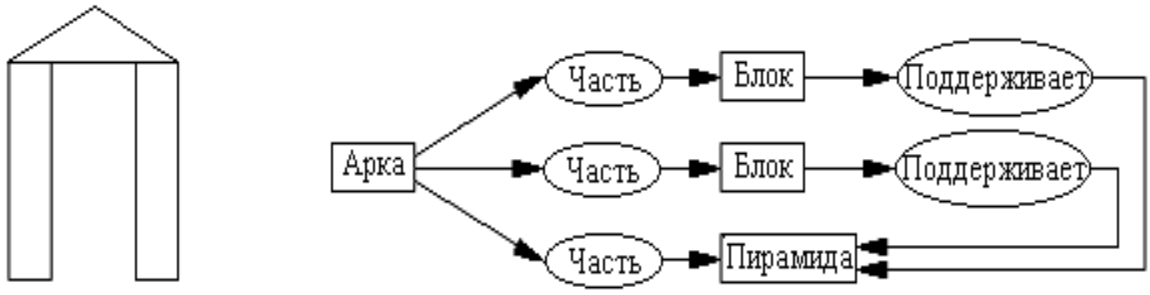


Рисунок 4.20 – Положительные и отрицательные примеры понятия “арка”



а)



б)

Рисунок 4.21 –Описание арки семантической сетью

Сопоставляя семантические сети, изображенные на рисунках 4.21,а и 4.21,б программа обнаруживает отличия между арками. Отличие заключается в том, что верхним элементом первой арки является блок, а второй – пирамида. Используя семантическую сеть, отражающую иерархию, указанных “строительных элементов” (рисунок 4.22), программа обобщает описание понятия “арка”. При этом вместо понятий блок и пирамида используется их ближайший супертип – многоугольник, т.е. выполняется минимальное обобщение. В результате формируется обобщенное описание арки, представляемое семантической сетью, изображенной на рисунке 4.23

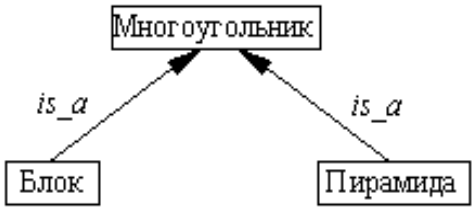


Рисунок 4.22 – Иерархия “строительных блоков”

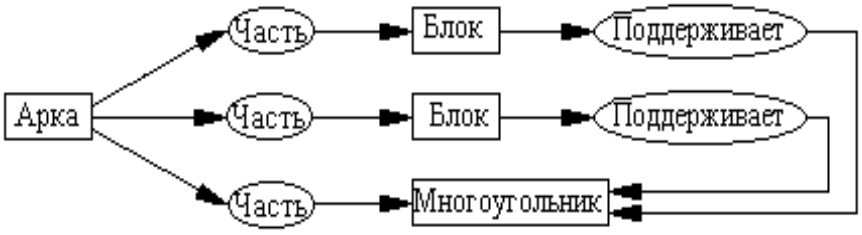


Рисунок 4.23 – Обобщенное описание арки

Данная семантическая сеть обобщает положительные примеры, изображенные на рисунке 4.21,а и 4.21,б. В случае предъявления отрицательных примеров выполняется ограничение общности (специализация) формируемого понятия. Если предъявляется контрпример, в котором поддерживающие блоки соприкасаются, то формируется семантическая сеть контрпримера, изображенная на рисунке 4.24. Сеть контрпримера отличается наличием отношения “соприкасается”.

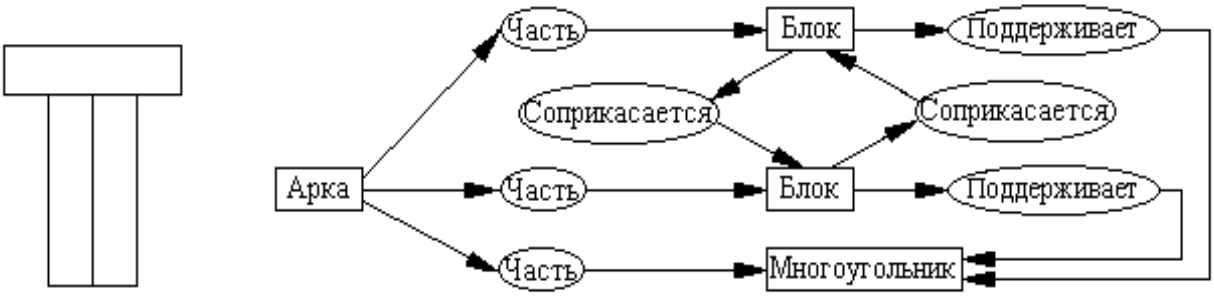


Рисунок 4.24 – Семантическая сеть контрпримера

Поэтому программа выполняет специализацию сети, изображенной на рисунке 4.24 , добавляя отношение “не должен соприкасаться” (рисунок 4.25). Целью специализации является исключение из формируемого описания понятия контрпримера. Отметим, что контрпримеры должны быть максимально близки к формируемому описанию целевого понятия. Рассматриваемый контрпример понятия “арка” отличается от положительного примера только одним дополнительным отношением. Это упрощает выполнение операции специализации.

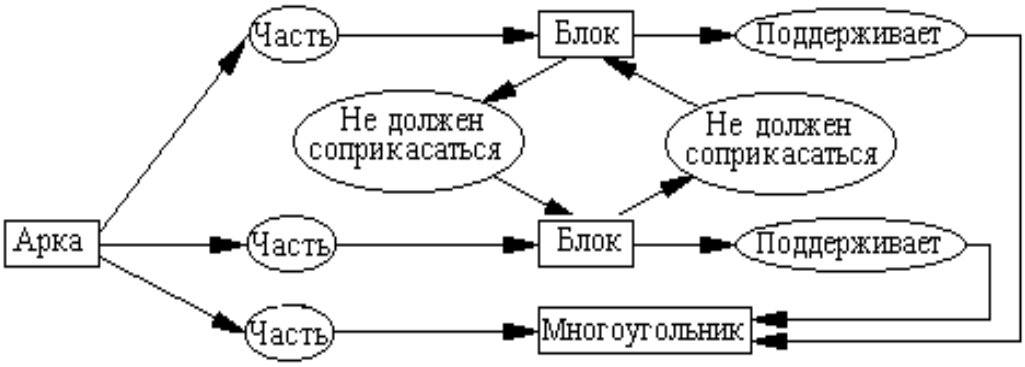


Рисунок 4.25 – Специализация описания

Таким образом, в ходе обучения выполняются две основные операции: *обобщение и специализация*. Обобщение заключается в замене узлов сети более общими понятиями. Специализация предполагает добавление в сеть новых связей. Если представить понятия, формируемые на основе операций специализации и обобщения, в виде некоторых состояний процесса обучения, то можно заметить, что программа П.Уинстона выполняет поиск в пространстве понятий. При этом поиск управляется обучающими примерами (данными) и соответствует алгоритму подъема на гору (см. § 2.2.2.1) [77]. Так как в этом случае не предусматривается возврат к предыдущим состояниям, то алгоритм П. Уинстона весьма чувствителен к порядку, в котором предъявляются обучающие примеры. Примеры должны предъявляться в “хорошем педагогическом порядке”, который исключает попадание в тупиковые вершины на графе поиска. Кроме этого, соседние примеры и контрпримеры должны минимально отличаться друг от друга, что облегчает выполнение операций обобщения и специализации, а также упрощает процедуру сопоставления подграфов семантической сети.

Программа П. Уинстона была одной из первых программ, реализующей идеи индуктивного обучения. Она обозначила важные моменты, свойственные многим программам машинного обучения:

- применение операций обобщения и специализация для формирования пространства понятий;

- применение методов поиска, управляемых данными (примерами);
- зависимость результатов обучения от качества обучающих данных.

4.3.3. Алгоритм Митчелла

Алгоритм Т.М.Митчелла, называемый алгоритмом исключения понятий–кандидатов, аналогично алгоритму П.Уинстона предназначен для формирования одного понятия по предъявляемым примерам. Алгоритм основывается на двунаправленном поиске в *пространстве версий*, которое представляется множеством всех понятий, охватывающих положительные примеры и не выполняющихся на отрицательных примерах. В процессе выполнения алгоритма поисковое пространство сокращается как в направлении от частных понятий к общим, так и в обратном направлении. Если предъявляется положительный пример, то из поисковой области исключаются все понятия-кандидаты, не охватывающие его. Если предъявляется отрицательный пример, то, наоборот, исключаются все охватывающие его понятия-кандидаты. Когда в поисковой области остается одно понятие, алгоритм прекращает работу.

Термин “охватывающие понятие” связан с операцией обобщения. При этом обобщение может выполняться следующими способами:
а) заменой константы на переменную, например, высказывание, заданное атомарной формулой

цвет(мяч, синий),

обобщается предикатом цвет(X, синий);
б) исключением конъюнктивных членов из формул, например, формула

$$\text{цвет}(X,\text{синий}) \wedge \text{вес}(X,\text{большой}) \wedge \text{форма}(X,\text{прямоугольная})$$

является частным случаем формулы

$$\text{цвет}(X,\text{синий}) \wedge \text{форма}(X,\text{прямоугольная});$$

в) добавлением дизъюнктивных элементов, например, утверждение

$$\text{цвет}(X, \text{синий}) \wedge \text{форма}(X, \text{прямоугольная})$$

обобщается формулой

$$\text{цвет}(X,\text{синий}) \wedge (\text{форма}(X, \text{прямоугольная}) \vee \text{форма}(X,\text{треугольная}));$$

г) обобщением некоторого свойства в соответствии с иерархией обобщений, например, предикаты

форма(X, прямоугольная),
форма(X, треугольная)

обобщаются с помощью формулы

форма(X, многоугольная)

для иерархии обобщений, изображенной на рисунке 4.26.

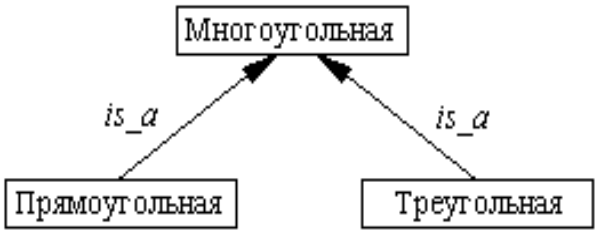


Рисунок 4.26 – Иерархия обобщений

Если понятие *p* является более общим, чем понятие *q*, то *p – охватывающее понятие* для *q*.

В алгоритме Митчелла примеры представляются при помощи атомарных формул исчисления предикатов, и в ходе поиска строятся (восстанавливаются) продукционные правила, посылки которых являются конъюнкциями указанных атомов. Восстановленные правила должны обеспечивать правильную классификацию предъявляемых примеров, т.е. отнесение их к тому или иному классу (понятию) по минимальной совокупности отличительных признаков.

Рассмотрим пример. Пусть имеется множество объектов, характеризуемых следующими свойствами и их возможными значениями:

цвет = {желтый, зеленый, красный, синий};
форма = {круг, квадрат, треугольник};
размер = {маленький, средний, большой}.

Каждый из объектов можно представить предикатом

объект(Цвет, Форма, Размер),

которому соответствует пространство понятий, изображенное на рисунке 4.26. Требуется на основании предъявляемых примеров построить правило, позволяющие обнаруживать круг красного цвета.

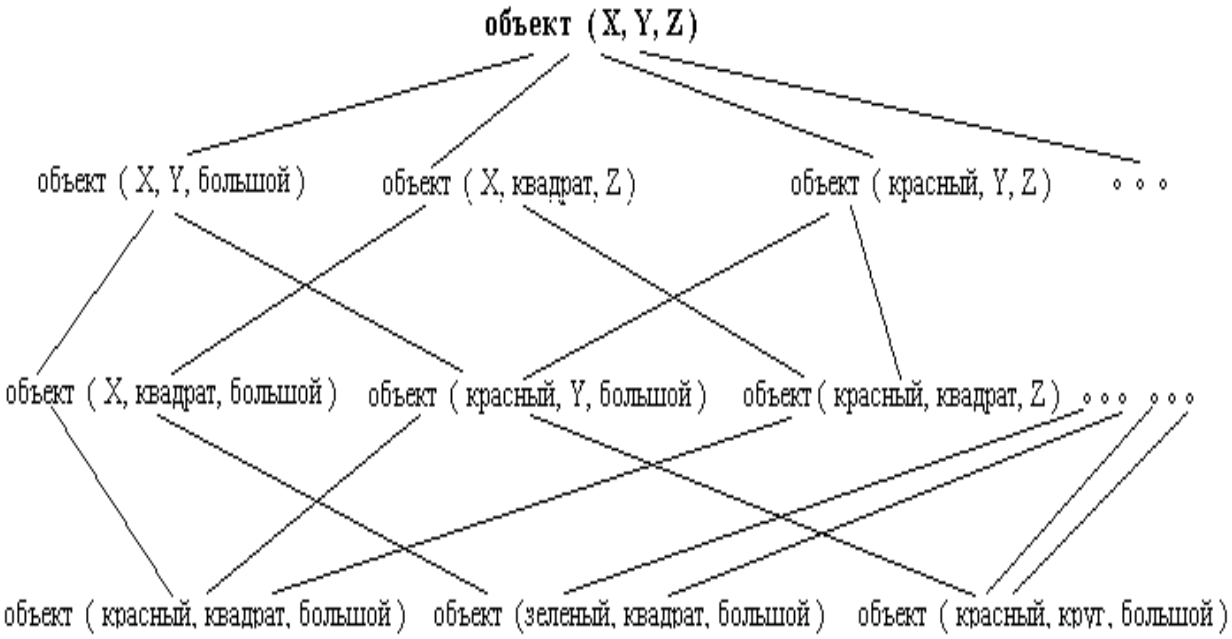


Рисунок 4.26 – Пространство понятий

Будем использовать следующие обозначения: G – множество наиболее общих понятий-кандидатов g, $g \in G$; S – множество наименее общих (наиболее специализированных) понятий-кандидатов s, $s \in S$. Алгоритм исключения понятий кандидатов формулируется следующим образом:

1) создать множество G, включив в него наиболее общее понятие пространства поиска;
2) создать множество S, включив в него первый положительный пример;
3) для каждого положительного примера p выполнить следующее:
- исключить из G все понятия-кандидаты, не охватывающие p;
- для каждого $s \in S$, если s не сопоставимо с p, заменить s более общим понятием, сопоставимым с p, выполнив минимально возможное обобщение;
- исключить из S все понятия-кандидаты более общие, чем любое другое понятие, входящее в S;
- исключить из S все понятия-кандидаты, охватывающие некоторые понятия, входящие в G;
4) для каждого отрицательного примера n выполнять следующее;
- исключить из S все понятия-кандидаты, охватывающие n;
- для каждого $g \in G$, если g сопоставимо с n, заменить g менее общим понятием, не сопоставимым с n, выполнив минимально возможную специализацию;

- исключить из G все понятия-кандидаты менее общие, чем любое другое понятие, входящее G;
- исключить из G все понятия-кандидаты, не охватывающие некоторые понятия, входящие в S;

5) если $G = S$ и оба множества являются одноэлементным, то найденное понятие согласуется со всеми данными, и поиск заканчивается;
6) если G и S – пустые множества, то не существует понятия, которое охватывает все положительные примеры и не охватывает ни одного отрицательного примера.

Процесс обучения понятию “красный круг” с помощью рассмотренного алгоритма представлен в таблице 4.3

Таблица 4.3 – Обучение понятию “красный круг”

№	Положительные (p) и отрицательные (n) примеры	Множества G и S
0	-	G: объект(X, Y, Z). S: { }.
1	p: объект(средний, круг, красный)	G: объект(X, Y, Z). S: объект(средний, круг, красный).
2	n: объект(средний, квадрат, зеленый)	G: объект(X, Y, красный), объект(X, круг, Z). S: объект(средний, круг, красный).
3	p: объект(большой, круг, красный)	G: объект(X, Y, красный), объект(X, круг, Z). S: объект(X, круг, красный).
4	n: объект(средний, треугольник, красный)	G: объект(X, круг, красный). S: объект(X, круг, красный).

Отметим, что множества G и S, представленные в таблице 4.3, содержат только окончательные результаты, соответствующие третьему и четвертому пунктам рассматриваемого алгоритма. Так, в случае предъявления отрицательного примера

объект(средний, квадрат, зеленый)

множество G на первом этапе расширяется при выполнении операции специализации за счет понятий-кандидатов, не сопоставимых с примером:

объект(X, Y, желтый),
объект(X, Y, красный),
объект(X, Y, синий),
объект(X, круг, Z),
объект(X, треугольник, Z),
объект(маленький, Y, Z),
объект(большой, Y, Z).

Затем из множества *G* исключаются все понятия-кандидаты, не охватывающие понятий, содержащихся во множестве *S*. В итоге в *G* остаются два понятия

объект(X, Y, красный),
объект(X, круг, Z),

что отражено в таблице 4.3 . Такое свойство алгоритма объясняется тем, что *S* – множество наименее общих понятий, которые охватывают положительные примеры, поэтому любое новое понятие, включаемое в *G* в ходе операции специализации и не сопоставимое с любым из понятий, входящих в *S*, не может охватывать положительных примеров и должно быть исключено.

Имеет место симметричная ситуация, когда выполняется операция обобщения *S*. Так как *G* – множество наиболее общих понятий, которые не охватывают ни одного отрицательного примера, то любое более общее понятие, включаемое в *S*, будет сопоставимо с некоторым отрицательным примером. Следовательно, оно должно быть исключено.

Условно поиск понятий в пространстве версий можно представить в виде рисунка 4.27 [77]. Здесь знаком “+” отмечены положительные обучающие примеры, а знаком “-” – отрицательные примеры.



Рисунок 4.27 – Обучение в соответствии с алгоритмом Митчелла

Множество *S* включает наименее общие понятия, которые охватывает все положительные примеры. Множество *G* включает наиболее общие понятия, которые не охватывают ни одного отрицательного примера. Любое понятие более общее, чем понятие из *G,* будет охватывать некоторые отрицательные примеры; любое понятие менее общее, чем понятие из *S,*

может не охватывать некоторые положительные примеры. Поэтому в процессе обучения граница множества G сужается, чтобы исключить понятия, охватывающие отрицательные примеры, а граница множества S расширяется за счет включения понятий, охватывающих положительные примеры. Процесс специализации понятий, входящих в G, и обобщения понятий, входящих в S, заканчивается успешно, когда множества G и S будут содержать одно и то же понятие.

Таким образом, алгоритм Митчелла относится к группе поисковых алгоритмов обучения. При этом выполняется поиск в ширину, что не эффективно, если множества G и S быстро расширяются. В этом случае могут применяться методы эвристического поиска.

В алгоритме Митчелла предполагается, что каждый класс описывается одним конъюнктивным правилом, и предъявляемые примеры не искажены шумом. На практике часто необходимо восстанавливать правила в условиях зашумленных данных. Алгоритм Митчелла весьма чувствителен к предъявляемым примерам. Даже один ошибочный пример может привести к потере сходимости.

4.3.4. Алгоритм обобщения ID3

Алгоритм ID3 (Iterative Dichotomizer 3) был предложен Дж. Квинланом в 1983 г. Также как и алгоритм Митчелла, он обеспечивает индуктивное формирование понятий по примерам. Однако понятия в этом случае представляются с помощью деревьев решений. Каждый нетерминальный узел дерева решений соответствует некоторому признаку классификации, а ветви, выходящие из узла, характеризуют альтернативные значения признака. Терминальные узлы дерева решений определяют класс (понятие), к которому относится предъявленный пример. На рисунке 4.28 изображен фрагмент дерева решений при определении степени риска сердечно-сосудистых заболеваний.

Алгоритм ID3 обеспечивает построение простейшего дерева решений, которое охватывает все предъявленные примеры. Построение дерева в соответствии с алгоритмом ID3 выполняется в направлении сверху вниз. В этом случае каждый признак разделяет множество обучающих примеров на подмножества, содержащие примеры с одинаковым значением того или иного признака. Затем алгоритм рекурсивно применяется к подмножествам и т.д. Процесс продолжается, пока предъявленный пример не будет отнесен к заданному классу.

В общем случае различный порядок анализа признаков приводит к построению разных деревьев решений. В алгоритме ID3 для определения по-

рядка, в котором анализируются признаки, используется информационный критерий. Сформулируем алгоритм индуктивного построения деревьев решений, а затем рассмотрим критерий отбора признаков.

Представим алгоритм в виде рекурсивной функции на псевдоязыке [77]. При этом будем использовать следующие обозначения: *MP* – множество примеров; *SP* – список признаков классификации; *V*(*P*) – множество значений признака *P*; *v* – значение признака *P*, $v \in V(P)$.

```
Function Induce_Tree(MP, SP);
Begin
If  Все элементы множества MP принадлежат одному классу
   Then
      Вернуть лист дерева, снабдив его меткой класса
Else If  список SP пустой Then
       Вернуть лист дерева, представив его в виде дизъюнкции
       всех классов, формирующих множество MP
       Else Begin
              Выбрать в качестве корня нового поддерева
              очередной признак P;
              Удалить признак P из списка SP;
              For v∈V(P) Do
                Begin
                  Создать ветвь дерева, снабдив ее меткой v;
                  Сформировать из MP подмножество примеров MPv,
                  признак P для которых имеет значение v;
                  Осуществить вызов Induce_Tree (MPv, SP) и
                  присоединить возвращаемое поддерево к ветви v
                End
            End
  End.
```



Рисунок 4.28 – Фрагмент дерева решений

Рассмотрим процесс построения дерева решений, изображенного на рисунке 4.28. Множество примеров *MP* задано таблицей 4.4

Таблица 4.4– Примеры оценки риска сердечно-сосудистых заболеваний

№	Риск	Курение	Потребление жиров
1	Высокий	да	излишек
2	Средний	да	норма
3	Средний	нет	излишек
4	Низкий	нет	норма

При первом вызове функции Induce_Tree в качестве корня дерева выбирается признак “курение”, который характеризуется двумя значениями “да” и “нет”. Эти значения разбивают множество примеров MP на два подмножества MP_1 и MP_2. Элементами первого подмножества являются примеры 1 и 2. Элементами второго подмножества – примеры 3 и 4. Для каждого из подмножеств MP_1 и MP_2 функция Induce_Tree вызывается рекурсивно. При этом в ходе каждого из вызовов происходит дальнейшее разбиение подмножеств MP_1 и MP_2, но уже относительно признака потребления жиров. В этом случае образуются подмножества, составленные из элементов одного класса, и рекурсия возвращает листья дерева (рисунок 4.28), снабженные метками соответствующих классов. Листья связываются с соответствующими ветвями, отмеченными возможными значениями признака “потребление жиров”, т.е. “норма” и “излишек”. В итоге формируются два поддерева, соответствующие подмножествам MP_1 и MP_2. Указанные поддеревья, в свою очередь, связываются с ветвями, отмеченными возможными значениями признака “курение”, и процесс формирования дерева решений завершается.

Каждый вызов функции Induce_Tree формирует очередное поддерево решений, корень которого представляется признаком *P*. В качестве критерия выбора очередного признака используется количество информации, обусловленное включением того или иного признака в дерево решений.

Пусть имеются независимые и несовместные сообщения $x_1, x_2,\ldots, x_n$, вероятности получения которых равны $p(x_1), p(x_2),\ldots, p(x_n)$. Тогда количество информации для всей совокупности сообщений можно оценить с помощью меры К. Шеннона:

$$I(x) = -\sum_{i=1}^{n} p(x_i)\log_2 p(x_i)\,. \qquad (4.20)$$

Если рассматривать обучающие примеры как сообщения, то с помощью (4.20) можно оценить количество информации, содержащееся в обучающей выборке и, следовательно, в дереве решений, которое охватывает

данную обучающую выборку. Так, для обучающей выборки, заданной таблицей 4.4 и соответствующей дереву решений, изображенному на рисунке 4.28 , получим p(высокий) = $1/4$; p(средний) = $2/4$; p(низкий) = $1/4$, и

$$I(x) = -\frac{1}{4}\log_2(\frac{1}{4}) - \frac{2}{4}\log_2\left(\frac{2}{4}\right) - \frac{1}{4}\log_2(\frac{1}{4}) = -\frac{1}{2}\log_2(\frac{1}{8}) = -\frac{1}{2}\cdot(-3) = \frac{3}{2} = 1{,}5 \quad \textit{бит}.$$

Количество информации, обусловленное включением того или иного признака в дерево решений, определяется как разность между количеством информации, соответствующим всему дереву решений, и количеством информации, соответствующих поддеревьев, расположенных ниже узла, определяемого выбранным признаком классификации. Количество информации, содержащееся во всех указанных поддеревьях, представляет ожидаемый объем информации, который может быть получен после их построения.

Пусть имеется множество примеров MP. Если на очередном шаге классификации выбирается признак P, характеризуемый n значениями, то множество MP разбивается на n подмножеств MP_1, MP_2, …, MP_n. Тогда ожидаемый объем информации, связанный с построением поддеревьев, соответствующих подмножествам MP_1, MP_2, …, MP_n, будет равен

$$E(P) = \sum_{i=1}^{n} \frac{|MP_i|}{|MP|} I(MP_i) ,$$

где $|MP|$, $|MP_i|$ - соответственно мощности множеств MP и MP_i; $I(MP_i)$ - количество информации, соответствующее поддереву множества MP_i.

Количество информации, получаемое при выборе признака P, определится из выражения

$$G(P) = I(MP) - E(P) .$$

На каждом шаге выполнения алгоритма ID3 выбирается признак P, обеспечивающий максимальное количество информации $G(P)$.

Алгоритм ID3 можно рассматривать как поиск, соответствующий алгоритму подъема на гору в пространстве возможных деревьев решений. На каждом шаге поиска анализируются все признаки, которые могут использоваться для расширения текущего дерева решений. Выбирается тот признак, который обеспечивает получение максимального количества информации.

Рассмотренный метод индуктивного построения деревьев решений применяется при решении многих задач классификации. Например, алгоритм ID3 применялся при обучении классификации окончаний шахматных

партий (король и ладья против короля и пешки [28]), при классификации заболеваний щитовидной железы. Это один из наиболее широко используемых методов индуктивного восстановления правил по примерам, обеспечивающий автоматическое построение баз знаний диагностических экспертных систем.

Вопросы для самопроверки

1. Приведите формулировку задачи дедуктивного вывода в терминах логики исчисления предикатов.
2. Сформулируйте этапы стандартизации предикатных формул.
3. Преобразуйте формулу $\forall x \forall y\{[P(x,y) \wedge Q(y)] \rightarrow \exists z R(x,y,z)\}$ к виду предложения.
4. Найдите НОУ для множества литералов $\{P(x,f(x),a),\ P(b,y,x)\}$.
5. Получите следствие $\forall x L(x)$ из посылок $\forall x(S(x) \rightarrow L(x))$ и $\neg(\exists x)(\neg S(x))$, используя принцип резолюции.
6. Назовите и объясните основные стратегии поиска, применяемые в ходе выполнения процедуры резолюции.
7. Приведите пример применения принципа резолюции в ходе информационного поиска.
8. Сформулируйте и приведите пример решения задачи планирования перемещения робота на основе принципа резолюции.
9. Объясните общую схему поиска решения в системе STRIPS.
10. Какие три вида дизъюнктов лежат в основе языка Пролог?
11. Постройте дерево опровержения для целевого утверждения ?-member(1,[2,1,3]).
12. Назовите и объясните виды неопределенности.
13. Запишите и объясните формулу Байеса для вычисления апостериорной условной вероятности гипотезы H
14. Объясните метод обработки неопределенности, используемый в системе PROSPECTOR.
15. Объясните метод обработки неопределенности, используемый в системе MYCIN (EMYCIN).
16. Как вычисляются степени доверия и правдоподобия в теории свидетельств Демпстера-Шефера.
17. Приведите и объясните определения нечеткой и лингвистической переменной.
18. Сформулируйте основные операции нечетких множеств.
19. Объясните понятие нечеткого отношения.
20. Представьте в виде нечеткого отношения правило “ если A, то B”, где A и B – нечеткие множества.
21. Что понимают под композицией нечетких отношений?
22. Сформулируйте правила вычисления функции принадлежности многозначной логики.
23. Приведите примеры нечетких правил вывода, предложенных Л. Заде в рамках лингвистической логики.
24. Какие выводы называют правдоподобными?

25. Что понимают под немонотонным выводом?
26. Запишите правило умолчания в общем виде. Что называют нормальным умолчанием?
27. Что понимают под системой поддержания значений истинности?
28. В чем отличие между дедукцией и индукцией?
29. Сформулируйте определение для термина “понятие”.
30. Какое понятие называют конъюнктивным?
31. Сформулируйте постановку задачи индуктивного формирования понятий.
32. Что понимают под обучением с учителем и без учителя?
33. Сформулируйте правило индуктивного обобщения.
34. Объясните суть алгоритма индуктивного обучения Уинстона.
35. Сформулируйте алгоритм исключения кандидатов Митчелла.
36. Объясните основную идею алгоритма построения деревьев решений ID3.
37. Сформулируйте критерий выбора очередного признака классификации в алгоритме ID3.

ГЛАВА 5
ЯЗЫК ЛИСП

В этой и следующей главе рассматриваются две важные парадигмы программирования, используемые при построении СИИ, – функциональное и логическое программирование. Функциональное программирование реализуется средствами языка Лисп, а логическое – языка Пролог. Важная характеристика этих языков – ориентация не на числовую обработку, а на обработку символов. Поэтому структуры данных, поддерживаемые этими языками, не ограничиваются массивами. Списки – основная структура данных, реализованная как в языке Лисп, так и в языке Пролог.

В языке Лисп списки используются и для представления обрабатываемых данных, и для представления самих программ. Пролог базируется на языке исчисления предикатов и также имеет удобные средства для обработки списков. Поддержка этими языками более мощных абстракций данных значительно повышает эффективность труда программиста. Согласно данным, приведенным в [7], в десять и более раз.

По этим причинам значительная часть прототипов программ с элементами ИИ реализуется на языках Лисп или Пролог. После выяснения функциональной работоспособности таких программ, они могут быть переписаны с помощью других языков для повышения эффективности реализации.

5.1. Общая характеристика языка

Язык Лисп был разработан в конце 50-х годов Дж. Маккарти. Название языка Лисп происходит от английских слов LISt Processing – обработка списков. Существовавшие в то время языки программирования были плохо приспособлены для решения задач ИИ, так как предназначались для обработки числовой информации. Цель, поставленная Маккарти, состояла в том, чтобы создать язык, ориентированный, прежде всего, на символьную обработку, а не на числовые вычисления, а также реализовать модель вычислений, основанную на *теории рекурсивных функций* А. Черча [26].

Первая версия языка Лисп появилась в 1960 г. Основная особенность языка состояла в том, что как данные, так и программы записывались в виде списков. При этом список в языке Лисп представлялся в форме множества из одного или более элементов, заключенных в круглые скобки. Это обеспечивало с одной стороны очень простой синтаксис языка, а с другой – достаточно высокий уровень абстракции данных, необходимый для решения задач ИИ.

Представление данных и программ в одной и той же форме позволяет, при необходимости, интерпретировать данные как программы. Это дает возможность одним программам на языке Лисп генерировать другие программы, что характеризует Лисп как весьма гибкий язык.

Говоря о Лиспе, нельзя не отметить другую его характеристику. Лисп – это язык функционального программирования. Функции являются основным инструментом для манипуляций со списками. В состав Лиспа входит большое число разнообразных функций обработки списков. Это позволяет относительно легко писать и отлаживать сложные программы.

Еще одна важная характеристика языка Лисп – это рекурсивный стиль написания многих функций. Рекурсивные определения имеют мощный математический фундамент, что позволяет выполнять аналитические исследования корректности функций, записывать их в ясной и лаконичной форме.

Сразу после появления первой версии языка начались работы по его совершенствованию. В 1961 г. была разработана новая версия языка под названием Lisp 1.5, которая претендовала на роль квазистандарта в период с 1961 по 1965 г. К началу 70-х годов появляются два новых диалекта языка Лисп – MacLisp и InterLisp. В диалекте MacLisp были введены функции с переменным числом аргументов и возможности работы с макросами. В диалекте InterLisp появились итерационные циклы, которые позже вошли во многие другие диалекты языка. Необходимо отметить, что к этому времени уже существовало значительное число различных диалектов языка Лисп, и встал вопрос о его стандартизации. Первые усилия по стандартизации языка были предприняты в 1965 г., когда на основе Lisp 1.5 был разработан язык Standard Lisp. Это был первый диалект языка Лисп, реализованный на различных программно-аппаратных платформах. Существенное влияние на развитие языка Лисп оказало появление во второй половине 70-х годов диалекта Scheme и технологии объективно-ориентированного программирования в языках Smalltalk и KRL. В диалекте Scheme было расширено понятия области действия переменных и введено понятие лексического замыкания. К началу 80-х годов вновь остро стал вопрос о стандартизации языка. В 1981 г. агентство DARPA при Министерстве обороны США инициировало работы по разработке нового стандарта языка. В ре-

зультате этих работ в 1984 г. была разработана версия языка под названием Коммон Лисп (Common Lisp). Язык Коммон Лисп вобрал в себя достоинства многих диалектов Лиспа, существовавших к тому времени – Machine Lisp, Mac Lisp, NIL, S-1 Lisp, SpiceLisp, Scheme. Хотя различные диалекты языка Лисп и оказали существенное влияние на Коммон Лисп, его не следует рассматривать как конгломерат отдельных элементов, заимствованных из них. Коммон Лисп – это высоко организованная, мощная среда программирования, содержащая весь арсенал средств, необходимый для разработки больших программных систем. Современные версии языка Коммон Лисп являются объектно-ориентированными, включают в себя среду визуального программирования и, по удобству работы, не уступают таким популярным оболочкам как DELPHI или CBuilder. Поэтому ниже будет рассматриваться именно этот диалект языка. С помощью Лиспа реализованы многие программные системы. Одна из самых известных систем – это AutoCAD фирмы AutoDesk.

5.2. Основные понятия языка

Лисп предназначен для обработки символьной информации. Поэтому основным типом данных, используемым в языке, является символьное выражение или s-выражение. Для того чтобы определить это и другие базовые понятия языка, воспользуемся расширенными металингвистическими формулами Бэкуса-Науэра (БНФ). При этом будем использовать обозначения, приведенные в таблице 5.1.

Алфавит языка Лисп состоит из цифр, букв и специальных знаков:

цифры::= 0| 1| 2| 3| 4| 5| 6| 7| 8| 9
буквы::= A| B| C … X| Y| Z| a| b| c … x| y| z
спецзнаки::= +| -| *| %| &| _| !| @| <| >| =| ?| /| {| }| ^| ~

Таблица 5.1 – Значения символов БНФ

Метасимвол	Значение
: : =	“есть по определению”
|	или (альтернатива)
$\{X\}^*$	ноль или более вхождений X, разделяемых пробелом
$\{X\}^+$	одно или более вхождений X, разделяемых пробелом
…	дальнейшее следование элементов
[X]	ноль или одно вхождение X
Метаимя	*нетерминальный символ с именем* Метаимя

Важное понятие языка Лисп – символы. *Символы* обозначают объекты, которыми манипулирует программа, и в этом смысле соответствуют именам переменных алгоритмических языков программирования. Однако символы в языке Лисп имеют и другие атрибуты, о которых будет сказано позже. Поэтому не следует их отождествлять с именами переменных алгоритмических языков программирования. Символы образуются из знаков алфавита по правилу:

символ::= *буква* | *символ буква* | *символ спецзнак* |
символ цифра | *цифра буква* | *цифра спецзнак* |
цифра символ | *спецзнак символ*

Примеры символов: A1, %1A, col13. В Лиспе не различаются прописные и строчные буквы. Поэтому символы, составленные из строчных букв, совпадают с соответствующими символами, составленными из прописных букв. Символы могут иметь значения. Для приписывания символам значений используются специальные формы. Изначально символы значений не имеют.

Наряду с символами, в Лиспе используются и числа. Лисп поддерживает работу с целыми, рациональными и вещественными числами:

число::=*целое число* | *рациональное число* | *вещественное число*

Числа записывают с помощью цифр и букв и, в отличие от символов, обозначают самих себя. Примеры чисел: 537, 2/3, 2.0E-3.

Символы и числа представляют простейшие объекты языка Лисп, поэтому их называют атомами. *Атом* определяется с помощью следующей формулы:

атом::=*символ* | *число* | ()

Пара скобок ( ) в языке Лисп имеет двойное значение. С одной стороны, она обозначает пустой список, а с другой – логическое значение *ложь*. Иными словами, в Лиспе ложь представляется пустым списком. Все остальные значения интерпретируется как *истина*. Для обозначения логического значения истина (true) в Лиспе используется символ T. Ложь или пустой список также обозначаются символом NIL. Символы T и NIL имеют фиксированное значение, и их нельзя использовать в качестве имен других лисповских объектов.

Из атомов можно построить основную структуру данных языка Лисп – *s-выражение*, которое определяется с помощью следующей рекурсивной формулы:

s-выражение ::= *атом* | (*s-выражение* . *s-выражение*)

Приведем примеры s-выражений, составленных в соответствии с введенным определением:

gruppa
(gruppa1 . gruppa2)
((gruppa1 . gruppa2) . gruppa3)

Обратите внимание на то, что точка отделяется от знаков, стоящих слева и справа, пробелом. Конструкция (*s-выражение* . s-*выражение*) представляет собой точечную пару. Элементами точечной пары могут быть как атомы, так и точечные пары. *Точечную пару* можно определить с помощью следующей БНФ:

*точечная пар*а ::= (*атом* . *атом*)|
(*атом* . *точечная пара*)|
(*точечная пара* . *атом*)|
(*точечная пара* . *точечная пара*)

Удобно точечные пары изображать в виде диаграмм. На рисунке 5.1 показана диаграмма для точечной пары (A . B). Такая диаграмма дает наглядное представление о форме хранения точечной пары в памяти ЭВМ и называется лисповской ячейкой.

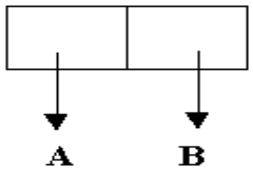


Рисунок 5.1 – Лисповская ячейка

Эта ячейка состоит из двух полей. В первом поле хранится указатель на первый элемент точечной пары, а во втором поле – указатель на второй элемент.

Если элементами точечной пары являются тоже точечные пары, то в ячейке хранятся два указателя, ссылающиеся на другие ячейки. На рисунке 5.2 изображена диаграмма для точечной пары ((A . B) . (C . D))

Сравнивая определение s-выражения и точечной пары, приходим к выводу, что

s-выражение ::= *атом* | *точечная пара*

В Лиспе выделяют специальный тип s-выражений, который называется списком. *Список* определяется следующим образом:

список ::= NIL | ((*s-выражение* . *список*))

Например, s-выражения (A .(B .(C .(D . NIL)))) и ((A . B) . ((C . D) . NIL)) – списки. Элементы первого списка – это атомы A, B, C, D, элементы второго списка – это точечные пары (A . B) и (C . D) (рисунок 5.3).

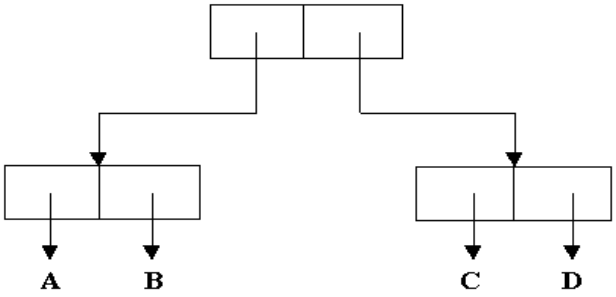
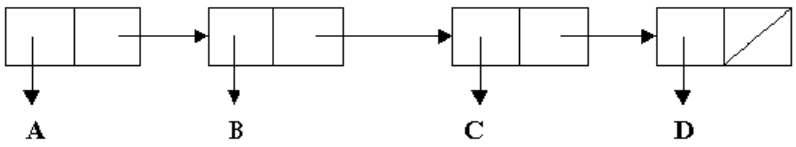


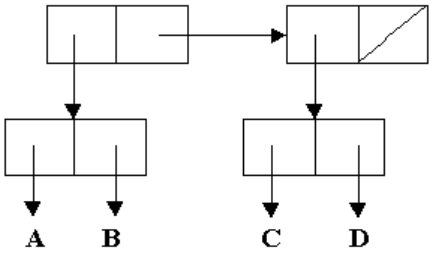
Рисунок 5.2 – Диаграмма точечной пары ((A . B) . (C . D))

В Лиспе принято также изображать список в виде последовательности элементов, отделяемых пробелом и заключенных в круглые скобки: (A B C D), ((A . B) . (C . D)). В этом случае список представляет собой упорядоченную последовательность, элементами которой являются как атомы, так и списки. Списки, записанные в такой форме, весьма удобны для представления различных моделей знаний. Вспомните, например, фреймовые модели. Список, заданный в виде последовательности элементов, всегда можно представить в точечной записи. Например, следующие записи эквиваленты:

(A B C D) ↔ (A . (B . (C . (D . NIL))))



а)



б)

Рисунок 5.3 – Диаграммы списков (A. (B . (C . (D . NIL)))) (а) и ((A . B) . ((C . D) . NIL)) (б)

Однако не каждое s-выражение, записанное в точечной форме, является списком. Например, выражение (A . (B . (C . D))) не является списком, так как не соответствует определению понятия список.

Для перехода от точечной записи выражения к записи его в форме последовательности элементов используют следующее правило. Если в s-выражении встречается точка, стоящая перед открывающейся круглой скобкой, то эту точку можно заменить пробелом, удалив упомянутую открывающуюся и парную закрывающуюся скобку. Так, s-выражение (A . (B . (C . D))) приводится к виду (A B C . D). Отсюда следует, что оно не является списком. Вместе с тем запись (A . (B . (C . (D . NIL)))) легко преобразуется к виду (A B C D) , если учесть, что атом NIL эквивалентен паре скобок (). Существует и обратное правило, позволяющее восстановить точечную запись [48].

Отметим, что некоторые записи, составленные из символов, точек и круглых скобок, не будут являться s-выражениями. Так, запись (A . B C) не может быть получена с помощью введенных выше определений, а потому не является s-выражением.

5.3. Интерпретация Лисп-программ

В Лиспе не только данные представляются в виде s-выражений, но и программы. Выполнение программы состоит в вычислении s-выражений. Однако не любое s-выражение может быть вычислено. Вычисляемое s-выражение характеризуется определенной семантикой, которая анализируется интерпретатором.

Если результаты анализа не противоречат семантике языка, то интерпретатор вычисляет значение s-выражения. S-выражение, значение которого может быть вычислено Лисп–интерпретатором, называется *формой.*

На рисунке 5.4 приведена обобщенная схема интерпретации s-выражений, работа которой состоит из трех шагов. На первом шаге происходит считывание s-выражения с помощью встроенной функции READ.

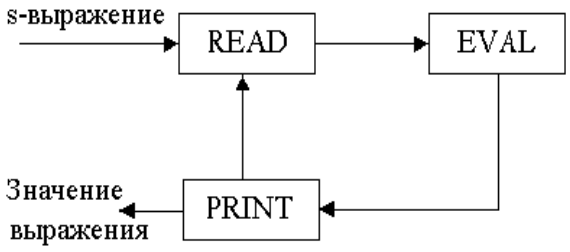


Рисунок 5.4 – Схема интерпретации s-выражений

По умолчанию исходные s-выражения вводятся с клавиатуры. На втором шаге выполняется интерпретация s-выражения с помощью функции EVAL, которая является вызовом интерпретатора. На третьем шаге функция PRINT осуществляет вывод значения s-выражения. По умолчанию вычисленные значения выводятся на дисплей. Затем цикл READ-EVAL-PRINT повторяется для другого s-выражения и т.д.

Программы на языке Лисп состоят из *форм и функций*. Лисповская функция может рассматриваться как коллекция форм, которые вычисляются при вызове функции. В свою очередь, *вызов функции* выполняется обычно по ее имени из формы, записанной в виде списка:

$$(f_n \quad a_1 \quad a_2 \quad \ldots \quad a_n),$$

где f_n – имя вызываемой функции; a_1, a_2 … a_n – аргументы функции, которые задаются вычисляемыми s- выражениями.

Например, если требуется вычислить сумму чисел 5 и 3, то с клавиатуры вводится следующее s-выражение (+ 5 3). Результат интерпретации показан на рисунке 5.5. Здесь знак “+” соответствует имени функции, выполняющей сложение.

```
> :   ( + 5   3)                ; вводимое S-выражение
8                                ; результат
>:                               ; приглашение ввода
```

Рисунок 5.5 – Пример интерпретации

В общем случае формы могут быть заданы константами, переменными, списками. *Константы* соответствуют *самоопределяемым формам*, которые представляют самих себя и имеют фиксированные значения (числа, символы T и NIL, строки и др.). *Переменные* в Лиспе обозначаются символами. При вычислении такой формы возвращается значение символа. Форма, заданная в виде списочной структуры, может представлять: вызов функции, рассмотренный выше, вызовы специальных форм, макровызовы. *Специальные формы* позволяют выполнять действия, недостижимые с помощью обычных функций, например, присваивать значения переменным, осуществлять условные вычисления. К таким формам относят: SETQ, QUOTE, IF и др. *Макровызовы* внешне соответствуют вызову функции, но отличаются по способу вычисления. Они вычисляются в два этапа: сначала из аргументов макроса строится форма, а затем она вычисляется. Подробнее макросы будут рассмотрены позже.

При вычислении s-выражений, представленных в форме констант, переменных или списков, интерпретатор EVAL реализует следующие упрощенные правила:

а) если s-выражение – это константа (самоопределяемая форма), например, число или атомы T и NIL, то EVAL возвращает такое s-выражение без изменений;

б) если s-выражение – это переменная, представленная символом, то функция EVAL возвращает последнее значение, которое было связано с этим символом; если символ не имеет значения, что выдается сообщение об ошибке;

в) если s-выражение – это список, и первый элемент списка – символ, то EVAL интерпретирует его либо как имя функции, либо как имя специальной формы, либо как имя макроса. Указанные имена должны быть известны системе. Если первый элемент списка – имя функции, то EVAL интерпретирует оставшиеся элементы списка как ее аргументы, которые подлежат вычислению с помощью EVAL.

Важным достоинством режима интерпретации является удобство отладки программ. Если случаются ошибки, то их можно тут же исправить, введя правильное s-выражение. Недостатком режима интерпретации является низкая скорость выполнения программы. Режим компиляции программ позволяет существенно улучшить этот показатель [48].

5.4. Арифметические функции

В Лиспе имеется большое количество встроенных арифметических функций. Аргументы этих функций должны иметь числовые значения. Числа в Лиспе подразделяются на целые, вещественные и рациональные. Рациональные числа представляются в виде дроби, например, 2/3. Кроме этого, в Лиспе предусмотрена обработка комплексных чисел. Комплексное число 5+3i на языке Лисп записывается в форме #C(5 3).

Ниже приведены примеры вызовов функций, выполняющих основные арифметические операции:

```
(+ 2 3)          → 5        ; сложение 2-х аргументов
(+ 3 1 2 3)      → 9        ; сложение 2-х и более аргументов
(+ 9)            → 9        ; унарный +
(- 13 1 2 3)     → 7        ;вычитание 2-х и более аргументов
(- - 9)                → 9          ;
(* 2 1.5)        → 3,0      ; умножение 2-х аргументов
(* 2 2 3)        → 12       ;умножение 2-х и более аргументов
(/ 7 2 2)        → 7/4      ;деление целочисленных аргументов
```

```
(/ 7 2 2.0)          → 1.75     ;деление с преобразованием
                                ;к вещественному значению
(/ 5.0)              → 0.2      ; обратное значение
(/ 2)                → 1/2      ;
(min  2  3  1)       → 1        ; минимум
(max  2  3  1)       → 3        ; максимум
(abs  - 5)           → 5        ; абсолютное значение
(rem 17  4)          → 1        ; остаток от деления.
```

Если в арифметических операциях участвуют вещественные аргументы, то результаты представляются в вещественной форме.

Для сравнения скалярных чисел в Лиспе используются функции, имена которых задаются знаками отношений (=, | = (не равно), <, >, <=, >=). Эти функции могут иметь произвольное количество аргументов. В этом случае проверяется выполнение соответствующего отношения для всех аргументов. Если отношение выполняется, то функция возвращает в качестве результата значение T, иначе – NIL. Например,

```
(>  5  4  3  2  1)   →  T
```

В Лиспе имеется большой набор иррациональных и трансцендентных функций: EXP, EXPT, LOG, SQRT, SIN, COS, TAN, ASIN, ACOS, ATAN и др. Аргументы тригонометрических функций задаются в радианах. Ниже приведены примеры вызовов некоторых из указанных функций:

```
(expt 2  10) →  1024
(log  10  2) →  3.321928
(sqrt  -4)   →  #C(0.0   2.0)
```

Функции TRUNCATE, ROUND, FLOAT выполняют преобразование чисел из одного формата в другой. Функция TRUNCATE возвращает два значения – целую и дробную часть числа:

```
(truncate  - 1.7) →  -  1   - 0.7
```

Функция ROUND выполняет округление числа и также возвращает в качестве результата два значения – ближайшее целое и разность между исходным значением и найденным ближайшим целым:

```
(round   5.7) → 6   - 0.3
```

Функция FLOAT преобразует целое число в число с плавающей точкой.

Хотя изначально Лисп предназначался для символьной обработки, современные версии языка не менее эффективно выполняют и числовую обработку.

5.5. Основные функции обработки списков

Для обработки списков в Лиспе имеется небольшой набор базовых функций, образующих ядро языка. Такие функции обеспечивают доступ к различным частям списков и позволяют конструировать из них другие списки.

К операциям выделения отдельных частей списков и создания из них новых структур сводятся многие алгоритмы обработки символьной информации. Основными функциями, на основе которых выполняются указанные действия, являются встроенные функции CAR, CDR и CONS. Ниже рассматриваются также и другие функции, облегчающие работу со списками и имеющиеся в языке Коммон Лисп. Эти функции определяются на основе функций CAR, CDR, CONS. Поскольку список – частный случай точечной пары, то ниже рассматриваются функции, аргументы которых заданы точечными парами.

Для доступа к первому элементу точечной пары или списка применяется функция CAR. Например,

(car '(a . b)) → A
(car '(a b)) → A

Здесь апостроф, обозначает вызов специальной формы QUOTE , которая блокирует вычисление значения s-выражения (a . b) или (a b). Если этого не сделать, то Лисп-интерпретатор (EVAL) предпримет попытку вычислить эти s-выражения. В соответствии с правилами работы интерпретатора, первый элемент s-выражения воспринимается как имя функции. Однако, на самом деле, функции с именем A нет. Поэтому возникает ошибка: UNDEFINED-FUNCTION (неопределенная функция). Применение функции QUOTE сообщает Лисп-системе, что аргументы функции CAR не подлежат вычислению и представляют самих себя. Необходимость блокировки вычислений возникает всякий раз, когда рассматриваются символьные данные.

Для выделения второго элемента точечной пары применяется функция CDR. Например,

(cdr '(a . b)) → B
(cdr '(a b)) → (B)

В первом случае возвращается атом B, а во втором случае – список, содержащий атом B. Этот результат объясняется тем, что список (A B) в точечной записи имеет вид (A . (B . NIL)). Функция CDR возвращает второй элемент точечной пары (B . NIL), т.е. (B).

Функция CAR и CDR определены только для точечных пар. Для аргументов атомов результаты функций CAR и CDR неопределены. Поэтому при попытке вызова (car 'a) формируется сообщение об ошибке.

Для выделения произвольных элементов списка используются *композиции функций* CAR и CDR. Например,

(car (car '((a b) (b c)))) → A
(car (cdr (cdr '(1 2 3)))) → 3

Композиции функций CAR и CDR встречаются довольно часто, поэтому для них введены специальные имена:

(car (car x)) ↔ (caar x),
(car (cdr x)) ↔ (cadr x),
(cdr (car x)) ↔ (cdar x),
(cdr (cdr x)) ↔ (cddr x),
…
(cdr (cdr (cdr (cdr x)))) ↔ (cddddr x).

В Коммон Лиспе для выделения отдельных элементов списков можно использовать более удобные имена функций, отражающие суть выполняемой операции. Например:

REST – эквивалентна CDR, возвращает хвост вписка;

FIRST– эквивалента CAR, возвращает первый элемент вписка;

SECOND – эквивалентна CADR, возвращает второй элемент списка;

THIRD - эквивалентна CADDR, возвращает третий элемент списка и т.д.;

TENTH – возвращает десятый элемент списка. Имена указанных функций соответствуют порядковым числительным английского языка.

Для создания точечной пары из s-выражений в Лиспе применяется функция CONS (конструктор):

(cons 'a 'b) → (A . B)

Конструирование списков с помощью функции CONS осуществляется последовательными вызовами:

(cons ('x (cons 'y (cons 'z NIL)))) → (X Y Z)

Здесь сначала строится точечная пара (Z . NIL), затем точечная пара – (Y . (Z . NIL)), а в конце – точечная пара (X .(Y . (Z . NIL))), которая и представляет список (X Y Z). Таким образом, функция CONS включает очередной элемент в начало списка, представленного ее вторым аргументом. Конструирование списков с помощью CONS не очень наглядно. В Лиспе имеется функция LIST с произвольным числом аргументов, которая упрощает задачу создания списков:

(list 1 'b 'c) → (1 B C)

Списки можно также создавать из других списков путем их объединения. Для этого применяется функция APPEND. Например,

(append '(1 2 a) '(b 4 5)) → (1 2 A B 4 5)

В Лиспе имеется ряд других полезных функций для обработки списков:

LAST – выделяет последний элемент списка (результат – список);
NTH – выделяет n-й элемент списка;
SUBST – выполняет замену элементов в списке;
BUTLAST – выделяет список без последних элементов;
MEMBER – проверяет принадлежность элемента списку;
LENGTH – вычисляет длину списка;
REVERSE – выполняет обращение списка;
ADJOIN – добавляет элемент в множество, представленное списком;
REMOVE– удаляет элемент из списка.

Примеры:

```
(last '(a b c d))          → (D)
(nth 3 '(a b c d))         → D    ; счет начинается с нуля
(subst 'a 'b '(a b c))     → (A A C)
(butlast '(a b c d) 2)     → (A B)
(member 'c '(a b c d))     → (C D)
(length '(a b (2 3)))      → 3
(reverse '(1 2 e 4))       → (4 E 2 1)
(adjoin 'a '(a b c d))     → (A B C D)
(remove 'a '(a b a b a b)) → (B B B)
```

Отметим, что все рассмотренные функции никоим образом не воздействуют на свои аргументы. Вычисление значений этих функций не приводит к побочным эффектам, т.е. следствием вызова функции является значение, возвращаемое в точку вызова.

5.6. Присваивание значений

Переменные в языке Лисп обозначаются с помощью символов. Переменные – это объекты программы, которые меняют свои значения в ходе её выполнения. Когда символ интерпретируется как форма, представленная переменной, то возвращается его значение. Однако из этого не следует, что переменная и символ в Лиспе тождественны. Интерпретация символа как переменной это только одна из возможностей. Иные возможности будут рассмотрены позже.

Для того чтобы присвоить символу значение, применяется функция SET. Данная функция присваивает символу некоторое значение и возвращает его в качестве результата. Присваивание является побочным действием этой функции, поэтому она представляет собой *псевдофункцию*.

Функция SET имеет два аргумента. Первый аргумент представляется формой, задающей символ, а второй – формой, определяющей значение символа, например:

(set 'x '(a b c)) → (A B C)
(set (car x) '(d e)) → (D E)

Здесь символу X присваивается значение (A B C), а символу A значение – (D E).

Символу можно также присвоить значение с помощью вызова специальной формы SETQ или макроса SETF. Формат вызова специальной формы SETQ:

(setq {*символ форма* }*)

В отличие от функции SET, форма SETQ блокирует вычисление значений символов. Например,

(setq x '(a b c)) → (A B C)
(setq y 12 z (+ y 10)) → 25

Присваивание значений символам в форме SETQ выполняется последовательно слева направо, т.е. в момент присваивания значения символу Z, символ Y уже получил значение 12. Результатом вызова SETQ является последнее присвоенное значение или NIL, если аргументы формы SETQ не были заданы.

Макрос SETF (set field) является обобщенной формой, используемой для изменения значений различных объектов. Формат вызова SETF:

(setf {*место форма* }*)

Здесь *место* – вычисляемое выражение, позволяющее определить лисповскую ячейку памяти, значение которой необходимо изменить.
Примеры:

(setf a 8.8 d 2.0) → 2.0
(setq x '(a b c)) → (A B C)
(setf (car x) 2 (cadr x) 3) → 3
x → (2 3 C)

В первом примере символ A получает значение 8,8 , а символ d – 2,0. Результатом вызова является последнее присвоенное значение – 2,0. Если принять, что значением символа X является список (A B C), то второй вызов макроса SETF меняет в этом списке вхождение A на 2, а B на 3. Иными словами, после вызова SETF символ X будет иметь значение (2 3 C). Если при вызове SETF аргументы не задавать, то результатом вызова будет NIL. Обработка пар аргументов макроса SETF выполняется последовательно слева направо. Макрос SETF позволяет изменять не только значение символа, но и другие его атрибуты.

5.7. Предикаты

Для обработки данных необходимы различные функции, позволяющие выяснять те или иные свойства данных. Так, выполняя арифметические функции, следует убедиться в том, что аргументы – числа, а в случае применения функций CAR и CDR в том, что аргументы представлены точечными парами. Функции, выполняющие такую проверку, возвращают логические значения и называются *предикатами.* В Лиспе имеется большой набор предикатов. Рассмотрим основные встроенные предикаты Лисп-систем.

Функция EQ сравнивает значения двух своих аргументов. При этом сравнение выполняется не на логическом уровне, а на физическом, т.е. выясняется, представлены ли аргументы в памяти ЭВМ одной и той же физической структурой данных. Функция EQ возвращает значение T, если ее аргументы – символы, и в памяти ЭВМ они представлены одной и той же структурой. Сравните следующие вызовы:

(setq x '(a b c)) → (A B C)
(setq y '(a b c)) → (A B C)
(eq x y) → NIL

и

(setq x '(a b c)) → (A B C)

```
setq  y  x ) → (A  B  C)
(eq  x  y ) → T
```

В первом случае, несмотря на логическое равенство значений символов X и У, функция EQ возвращает NIL. Это объясняется тем, что символы X и У представляются в памяти ЭВМ двумя разными структурами. Так как функция EQ требует, чтобы ее аргументы были символами, то при сравнении чисел могут получаться неожиданные результаты. Например,

```
(eq  5.0  5.0) → NIL
```

В Лиспе имеется функция EQL, которая аналогична EQ, но дополнительно позволяет сравнивать числовые значения одинаковых типов. Сравнение чисел разных типов выполняется с помощью предиката "=". Например,

```
(eql  5.0  5.0) → T
(=  5  5.0  0.5e1) → T
```

Сравнение объектов Лисп-программы на уровне абстрактных структур выполняется с помощью предиката EQUAL. Данный предикат обобщает предикат EQL и может сравнивать символы, однотипные числа, строки, списки:

```
(equal  'a  'a) → T
(equal  '(a  b  c)  '(a  b  c)) → T
(equal  "lisp"  "lisp" ) → T
(equal  5.0  5.0) → T
```

Однако

```
(equal  5.0  5) → NIL
```

Наиболее общее логическое равенство лисповских объектов устанавливается с помощью предиката EQUALP. Дополнительно к возможностям EQUAL данный предикат устанавливает логическое равенство разнотипных чисел.

Предикат NULL позволяет установить, представляет ли его аргумент пустой список:

```
(null  (cdr  '( a ))) → T
(null  nil) → T
(null  T) → NIL
```

Из двух последних вызовов предиката NULL следует, что применительно к логическим аргументам он выполняет функцию логического отрицания.

В Лиспе имеется большой выбор предикатов, классифицирующих типы значений: ATOM, NUMBER, INTEGERP, FLOATP, RATIONALP, SYMBOLP, LISTP и др. Предикат ATOM проверяет, представляет ли его аргумент атом:

```
(atom 'a) → T
(atom nil) → T
```

Предикат NUMBERP устанавливает, является ли его аргумент числом:

```
(number 5.0) → T
(number 'a) → NIL
```

Предикаты INTEGERP, FLOATP, RATIONALP позволяют выяснить, относится ли значение аргумента к целым, вещественным или рациональным числам, соответственно.

Предикаты SYMBOLP, CONSP, LISTP позволяют из множества различных значений выделять символы, точечные пары и списки:

```
(setq x '(a b c)) → (A B C)
(symbolp x) → NIL
(consp x) → T
(listp x) → T
(consp '( )) → NIL
(listp '( )) → T
```

Для проверки свойств числовых значений в Лиспе используются предикаты: ZEROP (равенство нулю), ODDP (нечетность), EVENP (четность), MINUSP (отрицательность).

Рассмотренные предикаты обычно применяются при построении тест-форм условных выражений.

5.8. Определение функций

С математической точки зрения функция f представляет отображение множества X на множество Y:

$$f: X \rightarrow Y.$$

Множество X – задает область определения функции, а множество Y – область значений функции. Говорят, что f – это функция типа $X \rightarrow Y$.

При этом каждому элементу $x \in X$ соответствует единственный элемент $y \in Y$. Это обычно записывают в хорошо известной форме y= f(x).

Таким образом, функция f – это множество пар (x, y), связанных свойством y=f(x), т.е.

$$f = \{(x, y) \in X \times Y : y = f(x)\},$$

где $X \times Y$ – прямое произведение множеств X и Y.

В общем случае функция может задавать отображение многих множеств X_1, X_2, … , X_n на множество Y. Такая функция относится к типу $X_1 \times X_2 \times \ldots \times X_n \to y_n$ и записывается в виде

$$f : X_1 \times X_2 \times \ldots \times X_n \to Y.$$

Приведенная запись означает, что функция имеет n-аргументов. Причем тип первого аргумента функции задается множеством X_1, второго – множеством X_2 и т.д.

Для того чтобы определить функцию, необходимо указать количество аргументов функций и их тип, а также правило f, устанавливающее суть отображения.

В языке Лисп для этого применяется *лямбда-выражение:*

(lambda ({*переменная*}*){*форма*}*)

Здесь конструкция ({*переменная*}*) называется *лямбда-списком* и представляет собой список формальных параметров, которые соответствуют аргументам функции. Последовательность форм в лямбда выражении определяет правило формирования значения функции и называется *телом* функции.

Пусть требуется определить функцию, вычисляющую x^5. Тогда лямбда-выражение будет иметь вид:

```
(lambda (x) ( * x x x x x))
```

Чтобы задать функцию, строящую список из двух аргументов x и y, можно использовать выражение

```
(lambda (x y) (cons x (cons y nil )))
```

Лямбда-выражение нельзя вычислить, оно лишь определяет правило вычисления функции. Для того чтобы применить функцию, определенную лямбда-выражением, к некоторым аргументам, необходимо в вызове функции записать вместо имени функции лямбда-выражение. Например:

((lambda (x) (* x x x x)) 4) → 1024
((lambda (x y) (cons x (cons y nil))) 'a 'b) → (A B)

Такие вызовы называются *лямбда-вызовами.* В первом примере формальному параметру X лямбда-списка присваивается фактическое значение 4, а затем вычисляется тело лямбда-выражения. Во втором примере формальным параметрам X и У назначаются значения фактических параметров, заданных атомами A и B, из которых строится список (A B). По окончании вычислений связи между формальными параметрами и значениями фактических параметров разрушаются.

Проводить вычисление с помощью лямбда-вызовов неудобно, так как приходится каждый раз при очередном вызове заново переписывать тело функции. Проблема решается, если с лямбда-выражением связать некоторое имя (символ), которое будет выступать в роли имени функции, определяемой лямбда-выражением. Для связи имени и лямбда-выражения в Лиспе применяется форма DEFUN:

(defun *имя лямбда-выражение*)

При этом для упрощения записи в лямбда-выражении опускается слово LAMBDA и скобки. В итоге получаем:

(defun *имя лямбда-список* {*форма*}*)

Результатом вызова формы DEFUN является имя определенной функции. Например:

(defun stepen 5 (x) (* x x x x)) → STEPEN 5
(defun spisok (x y) (cons x (cons y nil))) → SPISOK

Теперь указанные выше лямбда-вызовы можно заменить следующими вызовами:

(stepen 5 4) → 1024
(spisok 'a 'b) → (A B)

Отметим, что имя функции представляется символом. Поэтому можно говорить о том, что DEFUN устанавливает связь между символом и определением функции.

При задании формальных параметров в форме DEFUN или в лямбда-списке языка Коммон Лисп выделяют:

- обязательные параметры;
- необязательные параметры (optional);
- остаточные параметры, связываемые с хвостом списка параметров (rest);
- ключевые параметры (key);

- вспомогательные переменные (auxiliary).

Рассмотренные выше примеры определения функций содержали только обязательные параметры. Если необходимо в списке параметров указать параметры, отличные от обязательных, то их отмечают соответствующими ключевыми словами: &OPTIONAL, &KEY, &REST, &AUX. Указанные средства существенно расширяют возможности управления передачей параметров в функции.

Необязательные параметры при вызове функции можно не задавать. В этом случае они получают значение NIL или значения по умолчанию, определяемые формой, связанной с таким параметром. В списке формальных параметров необязательные параметры записываются после слова &OPTIONAL. Например,

```
(defun fn  (x &optional (y (+ x 5)))  (+ x y ) )
```

Здесь X – обязательный параметр, а Y – необязательный. Если параметр y не будет задан при вызове функции, то он получит значение по умолчанию, равное X+5:

```
(fn 2)   →   9
(fn 2 3 ) → 5
```

Параметр, указанный после ключевого слова &REST, является *остаточным* и соответствует списку, составленному из тех значений фактических параметров, которым не были назначены формальные параметры. Остаточный параметр является необязательным. В следующем примере Z– остаточный параметр. Функция PRINT_ARG печатает значения своих аргументов:

```
(defun  print_arg  ( x  &optional y &rest z)
          ( print x)
          ( print y)
          z )
 (print_arg 1 )
 1
 NIL
 →NIL
(print_arg 1 2 3 4 5 )
1
2
→ (3 4 5 )
```

Введение остаточных параметров позволяет легко определять функции с произвольным количеством аргументов.

Ключевые параметры при определении функции следуют после слова &KEY, а при вызове функции указываются с помощью имени параметра,

перед которым стоит двоеточие. Ключевые параметры удобны тем, что при вызове функции их можно перечислять в любом порядке:

```
(defun print_arg_key (&key x y  (z  5 ))
        (print x )
        (print y )
        z )
(print_arg_key  :y 4 x: 6 )
6
4
→5
(print_arg_key  :y 2 :z 1 :x  0 )
0
2
→1
```

Ключевые параметры являются необязательными, поэтому в первом вызове Z получает значение, равное пяти.

Рассмотренные виды параметров функций часто используют в справочниках при описании форматов функций. Например, если хотят подчеркнуть, что функция имеет произвольное количество аргументов, то пишут

(+ &REST *числа*)

или

(LIST &REST *элементы*)

Если функция имеет ключевые управляющие параметры, то в формат ее описания могут включаться имена этих параметров [48]:

```
(MEMBER элемент  список
        &KEY  : TEST
              : TEST-NOT
              :  KEY )
```

Значения некоторых ключевых слов, используемые во встроенных функциях, будут рассмотрены позже.

5.9. Связывание и область действия переменных

Ранее были рассмотрены формы, присваивающие значения переменным (SET, SETQ, SETF). Значения, присвоенные переменным с помощью этих форм, остаются действующими до тех пор, пока не будет сделаны новые назначения. Процесс назначения переменной определенного значения на определенном участке программы называется *связыванием* (binding).

Формы группы SET выполняют связывание в результате побочных эффектов. Возникающие при этом связи значений и переменных остаются в силе после завершения SET-вызовов. В Лиспе имеются и другие средства, выполняющие связывание. Область действия устанавливаемых связей определяется рядом обстоятельств.

Если требуется установить локальные связи, то используется форма LET:

(let ({(*переменная форма-инициализатор*)}*){ *форма* }*)

Здесь *форма-инициализатор* задает новое значение переменной, которое доступно только в переделах LET. По окончании LET-вызова восстанавливается старое значение переменной в соответствии со связями, которые имела переменная во внешнем окружении LET. Значение, возвращаемое LET-вызовом, соответствует последней вычисленной форме, входящей в LET. Например,

```
(setf x 5) → 5
(set y 7) → 7
let ((x 10) (z 6)) (+ x y z)) → 23
x → 5
z → Error: UNBOUND-VARIABLE
```

В этом примере в LET-вызове устанавливаются локальные связи для переменных X и Z. Переменная У в LET-вызове является *свободной*. Значение свободной переменной определяется связями внешнего окружения LET, т.е. У=7. После завершения LET-вызова переменная X восстанавливает свое значение, так как была ранее связана с помощью SETF. Переменная Z не имела связей во внешнем окружении LET, поэтому попытка определить ее значение за пределами LET завершается ошибкой – UNBOUND VARIABLE (несвязанная переменная).

Связывание переменных внутри формы LET выполняется параллельно. Поэтому в следующем примере результат LET-вызова будет пять, а не три:

```
(setf a 3) → 3
(let ((a 1) (b (+ a 2)) b) → 5
```

Иными словами, при связывании переменной B здесь используется не локальное значение переменной A, а глобальное, присвоенное вызовом SETF.

Если необходимо, чтобы связи локальных переменных устанавливались последовательно, то применяется форма LET*. Определим функцию, вычисляющую выражение

$$x = \frac{y + z \ / \ 2}{y^{\,2} + \sqrt{y + z.}}$$

Воспользуемся формой LET*:

```
(defun  f1 (y  z)
 (let*  (( a ( sqrt  ( + y  z  )))
         ( b  ( +  (*  y  y )  a ))
         ( c  ( +  y  ( /  z   2 )))
         ( x  (/  c  b )))  ; Конец списка инициализаторов
         x  )) ; возвращаемое значение
```

Здесь переменная B получает значение после связывания переменной A, переменная C – после связывания B и т.д. Поэтому присваивание значения переменной B происходит, когда переменная A уже получила значение, равное $\sqrt{y+z}$.

Форма LET представляет собой видоизмененный лямбда-вызов:

(let ((x_1 a_1) (x_2 a_2) … (x_n a_n)) {*форма* }*) ↔
((lambda (x_1 x_2 … x_n) ; формальные параметры
{*форма* }*)
a_1 a_2 … a_n) ; фактические параметры

Отсюда следует, что в LET-форме формальные параметры представляются переменными x_1, x_2,…, x_n, а фактические параметры a_1, a_2,…,a_n задают значения этих переменных. Связь формальных и фактических параметров является локальной и действует только в пределах этих форм. Так как лямбда-выражение подставляется вместо имени функции, то и при обычном вызове функции устанавливается связь фактических и формальных параметров, которая действительна только в пределах вызываемой функции. Так, в рассмотренной выше функции F1 значения формальных параметров Y и Z будут локальными. Их начальные значения будут соответствовать значениям фактических параметров, переданных в момент вызова функции. Никакие изменения значений формальных параметров внутри функции не могут отразиться на значениях фактических параметров, так как их связь является локальной. Иными словами, передача значений от фактических параметров к формальным выполняется копированием (т.е. по значению).

Переменные, имеющие локальные связи, в Коммон Лиспе называются *лексическими* или *статическими*. Область действия такой переменной ограничивается той формой, в которой переменная определена, т.е. связи лексической переменной действительны в некотором фиксированном текстуальном (лексическом) окружении. Эти связи не действуют вне тексту-

ального окружения, а также в функциях, которые вызываются в этом окружении, но описаны за его пределами. Например,

```
(setf  x  '(a   b )) → (A  B)
(defun   print_car ( )
          (print  (car  x ))) →  PRINT_CAR
(let  ( ( x  '( c  d ))) ( print  (car  x ))
                          ( print_car))
C
A
→ A
```

Здесь свободная переменная X в функции PRINT_CAR связана с глобальным значением (A B). Эта связь сохраняется также в том случае, когда функция PRINT_CAR вызывается в LET-форме после локального (лексического) связывания переменной X со списком (C D). Таким образом, лексическим окружением, в котором переменная X имеет значение списка (C D), является форма LET, за исключением вызываемой функции PRINT_CAR.

Область действия лексических переменных статическая, т.е. она не изменяется в ходе выполнения программы. Например, формальные параметры функций, являющиеся лексическими переменными, могут при выполнении функции получать новые значения. Однако областью действия (областью видимости) этих значений будет только форма, определяющая функцию:

```
(setf  x  5 ) → 5
(defun  inc_x  ( x )
         (setq  x  ( +  x  1 ))) → INC_X
(inc_x   x) → 6
 x → 5
```

Альтернативой лексическим переменным являются *динамические* или *специальные* переменные.

Область действия такой переменной определяется не текстуальным окружением, в котором переменная используется, а динамическим окружением, возникающим в процессе выполнения программы.

Динамические переменные создаются с помощью формы

(defvar *имя* [*начальное значение*])

Чтобы отличать такие переменные, им дают особые имена. Обычная практика состоит в том, чтобы такие имена начинались и заканчивались знаком “ * ”. Значение динамической переменной определяется по последней связи:

```
( defvar  * x *) → * x *
( setf  * x *  '( a  b )) → (A B)
( defun  print_car ( )
          (print  ( car  * x *))) →  PRINT_CAR
( let  (( * x *  '( c  d ))) ( print  ( car  * x *))
                    ( print_car ))
C
C
→ C
```

При вызове функция PRINT_CAR обрабатывает значение переменной *x*, связанное не в форме SETF, а в форме LET. Именно оно было связано последним.

В связи с введением лексических и динамических переменных возникают интересные особенности в работе функции SET. Функция SET не может изменить значение лексической переменной:

```
(set  x  100) → 100
(defun   f ( x ) (set  'x  5 ) (print  x )) → F
(f  3)
3
→ 3
x → 5
```

Как видно из примера, при вызове функции F печатается значение 3, соответствующее лексической связи переменной X. В то же время функция SET изменила глобальное значение переменной X, т.е. первым аргументом функции SET является не лексическая переменная, а динамическая.

Применение динамических и свободных переменных в функциях может приводить к путанице и затруднять отладку программ. Поэтому предпочтение следует отдавать лексическим переменным.

Важно, чтобы с символом были связаны значения. Попытка вычисления значения несвязанного символа приводит к ошибке. Для того чтобы проверить наличие значения у символа применяется предикат BOUNDP. Само значение символа можно определить при помощи функции SYMBOL-VALUE .

5.10. Последовательные и условные вычисления

Последовательные вычисления в Лиспе выполняются с помощью форм PROG1, PROG2, PROGN. Данные формы соответствуют составным операторам алгоритмических языков и имеют схожий формат вызова:

(progn {*форма*}*)
(prog1 *форма1* {*форма*}*)
(prog2 *форма1 форма2* {*форма*}*)

Здесь формы, выступающие в роли аргументов PROG-вызовов, вычисляются последовательно, и в качестве результата возвращается значение последней формы (PROGN), первой формы (PROG1) или второй формы (PROG2). Рассмотрим пример моделирования стекового механизма:

(setf stack '(a b c d e f)) → (A B C D E F)
(prog1 (first stack) (setf stack (rest stack))) → A
stack → (B C D E F)

Последовательные вычисления можно также выполнять с помощью рассмотренной выше формы LET*. Эта форма позволяет использовать локальные переменные и в качестве результата возвращает значение последней вычисленной формы, входящей в LET-вызов.

Для выполнения условных вычислений в Лиспе могут применяться формы COND, IF, WHEN, UNLESS и др. Форма COND представляет собой макрос и позволяет выбрать одну из n-ветвей алгоритма. Формат вызова формы COND:

$$(\text{cond } \{(\ \textit{тест-форма}\ \{\ \textit{форма}\ \}^*)\}^+)$$

В этой записи *тест-форма* представляет собой логическое выражение, построенное с помощью различных предикатов и логических функций: AND (логическое И), OR (логическое ИЛИ), NOT (логическое отрицание). Конструкцию (*тест-форма* {*форма*}*) называют *клозом* (clause). Клоз представляет собой список форм.

Приведенный формат вызова формы COND является обобщением следующей записи:

(cond (*тест-1 форма-11 форма-12* …)
(*тест-2 форма-21 форма-22* …)
…
(*тест-i форма-i1 форма-i2* …)
…
(*тест-N форма-N1 форма-N2*…))

Значение, возвращаемое формой COND, определяется следующими правилами. Последовательно вычисляются значения тест-форм, пока не встретится тест-форма, например *тест-i*, значением которой является T. Затем последовательно вычисляются формы, соответствующего клоза, т.е. *форма-i1*, *форма-i2* и т.д. Значение последней вычисленной формы клоза

будет представлять результат, возвращаемый формой COND в точку вызова. Если ни одна из тест-форм не получила значение T, то результатом вызова формы COND будет NIL.

Обычно в качестве последней тест-формы используется символ T. Соответствующие ей формы клоза будут вычисляться, когда ни одна из предшествующих тест-проверок не выполняется. Рассмотрим простой пример:

```
(cond ((eq x 'Bern) 'Switzerland)
      ((eq x 'Paris) 'France)
      ((eq x 'Berlin) 'Germany)
      ((eq x 'Kiev) 'Ukraine)
      (t 'unknown))
```

Здесь тест-формы реализованы с помощью предиката EQ. Если значением X является атом, представляющий название одной из указанных столиц, то в качестве результата возвращается название соответствующего государства. Во всех остальных случаях возвращается значение UNKNOWN.

В рассмотренном примере выбиралась одна из пяти ветвей алгоритма. Когда требуется выбрать одну из двух ветвей алгоритма, то используют специальную форму IF. Формат вызова формы:

(if *тест-форма then-форма* [*else-форма*])

Если результатом вычисления тест-формы является значение T, то IF возвращает значение, определяемое *then-формой*, иначе – *else-формой*, которая может отсутствовать. Пусть требуется вычислить абсолютное значение:

$$y = \begin{cases} x, & x \geq 0, \\ -x, & x < 0. \end{cases}$$

Соответствующее выражение на языке Лисп будет иметь вид:

```
(if ( >= x 0) x ( - x ) )
```

Условные вычисления можно также организовать с помощью макроформ WHEN и UNLESS. Форма WHEN имеет следующий формат вызова:

(when *тест-форма* { *форма* }*)

Здесь формы вычисляются лишь в том случае, если результат вычисления *тест-формы* – T. Результатом вызова макроса WHEN является значение последней вычисленной формы. Например,

```
(setq  x  3) → 3
(when  ( >  x  0 ) (setf  a  5) (setf  b  6)) → 6
```

Если *тест-форма* имеет значение NIL, то и вызов макроса WHEN возвращает NIL. Формат макровызова UNLESS аналогичен формату WHEN, но формы будут вычисляться лишь в том случае, если результат вычисления тест-формы – NIL.

Рассмотренные вызовы форм COND, IF, WHEN, UNLESS называют *условными предложениями.*

5.11. Итерационные циклические вычисления

В Лиспе имеется большой арсенал средств для организации циклических вычислений. Одним из самых старых средств организации итерационных циклов является макроформа LOOP. Различают два варианта записи вызова LOOP – простой и расширенный. Рассмотрим простой вызов LOOP:

(loop {*форма*}*)

Здесь конструкция {*форма*}* образует тело цикла. Формы, входящие в тело цикла, вычисляются циклически. Цикл завершается, когда управление, в одной из форм, передается макровызову возврата RETURN. При этом RETURN вычисляет значение, возвращаемое в точку вызова LOOP.

Рассмотрим пример. Пусть требуется найти сумму чисел от 1 до n. Ниже приведен текст функции, решающий эту задачу, с помощью макровызова LOOP:

```
(defun nsum (n)
        (setf sum 0)
        (loop (cond ((zerop n) (return sum))
                    (t (setf sum (+ sum n)) (setf n (- n 1)) ) )))
```

Другая конструкция, применяемая для организации итерационных циклов, – макроформа DO. Эта форма позволяет задавать: локальные (лексические) переменные цикла, их начальные значения и правила обновления; условие окончания цикла и значение, которое возвращает форма DO; формы, образующие тело цикла.

Общий формат макровызова DO:

(do (({*переменная*} [*начальное_значение* [*обновление*]])}*)
(*условие_окончания* {*форма*}*)
{*форма*}*)

Первый аргумент DO-вызова представляет собой список, состоящий из подсписков. Элементами подсписков являются: переменные цикла; формы, задающие начальные значения переменных; выражения, определяющие правила обновления переменных. Если форма, устанавливающая начальное значение переменной, не задана, то значение переменной равно NIL. Если не задана форма обновления, то значение переменной может обновляться в теле цикла.

На каждой итерации вычисляется условие окончания цикла. Когда условие окончания цикла получает значение “истина“, то последовательно вычисляются формы, входящие во второй список DO-вызова. Значение последней вычисленной формы и будет результатом. Если условие окончания цикла ложно, то вычисляются формы, образующие тело цикла.

Пусть требуется вычислить факториал числа N. Определим для этого функцию FACTORIAL-DO:

```
(defun factorial-do (n)
       (do ((counter 1 (+ counter 1))       ; управление counter
            (result 1 (* result counter)))    ; управление result
            ((=  counter (+ n 1)) result)))  ; условие  окончания
```

В этом примере для вычисления факториала используется формула n!=(n-1)! n, т.е. факториал каждого следующего числа вычисляется через факториал предыдущего числа. Результаты вычислений на каждом шаге сохраняются в локальной переменной RESULT. Количество выполненных итераций запоминается в счетчике COUNTER. Выход из цикла происходит при условии COUNTER = (N+1).

Присваивание значений переменным цикла в DO-вызове выполняется параллельно. Если необходимо последовательное присваивание, то применяется макровызов DO*.

Определим функцию MY-INTERSECTION, которая формирует список общих элементов двух списков:

```
(defun my-intersection (a b)
       (do* ((aa a (rest aa))                ; управление переменной aa
             (e (first a) (first aa))        ; управление переменной e
             (inter ( )) )                   ; управление переменной inter
             ((null aa) inter)               ; условие окончания цикла
             (if (member e b)                ; тело цикла
                    (setf inter (cons e inter)))    ))
```

В цикле сначала вычисляется значение переменной AA, а затем E. При этом переменная E соответствует первому элементу очередного хвоста списка AA, т.е. E – это очередной элемент входного списка, представленного формальным параметром A. В переменной INTER накапливается результат. Её начальное значение равно ( ). Когда список AA будет исчерпан, то функция вернет в качестве результата значение переменной INTER.

Часто необходимо выполнять циклическую обработку каждого элемента списка. В этом случае применяется макроформа DOLIST:

(dolist (*переменная список* [*результат*])
{ *форма* }*)

Если требуется выполнить некоторые действия N раз, то используется макроформа DOTIMES:

(dotimes (*переменная счетчик* [*результат*])
{*форма*}*)

Здесь действия выполняются для целочисленных значений переменной, задаваемых формой *счетчик* и лежащих в диапазоне от 0 до N-1. Когда переменная цикла станет равной N, цикл завершится, возвратив значение формы *результат*.

Циклические вычисления, использующие передачу управления операторам с метками, можно выполнить с помощью макроформы PROG. Упрощенный формат PROG-вызова имеет вид:

(prog ({*переменная*}*) {*форма*}*)

Указанные в начале PROG-вызова переменные являются локальными. Их начальные значения равны NIL. При необходимости они могут быть инициализированы другими значениями аналогично LET-вызову. Даже если список переменных PROG-вызова пустой, его необходимо всё равно задавать.

Формы, записываемые внутри PROG-вызова, называются операторами. Если какая-либо форма представлена символом или числом, то она рассматривается как метка. Передать управление на метку можно с помощью оператора GO:

(go *метка*)

Для окончания вычислений и возврата результатов из PROG-вызова применяется форма RETURN.

В следующем примере PROG-форма применяется для вычисления факториала:

```
(defun factorial-prog (n)
       (prog ((counter 1)(result 1))
            A  (setq result (* result counter))
               (setq counter (+ counter 1))
               (if (= counter (+ n 1)) (return result))
               (go A)))
```

Здесь переменные COUNTER и RESULT получают начальные значения, равные единице.

PROG-форма была введена в ранних версиях языка Лисп и называлась PROG-механизмом. В связи с введением более развитых форм циклических вычислений PROG-форма находит ограниченное применение.

5.12. Рекурсивные функции

Функция называется рекурсивной, если в ее определении содержится обращение к ней самой. Рекурсивное определение функций часто встречается в математике. Например, определение факториала

$$N! = (N-1)!\,N$$

или чисел Фибоначчи

$$F_1 = 1,$$
$$F_2 = 1,$$
$$F_n = F_{n-1} + F_{n-2}, \qquad n > 2.$$

Лисп специально создавался для рекурсивных вычислений, и вся сила языка как раз и заключена в возможности широкого использования рекурсивных функций. Продемонстрируем это на примере вычисления факториала:

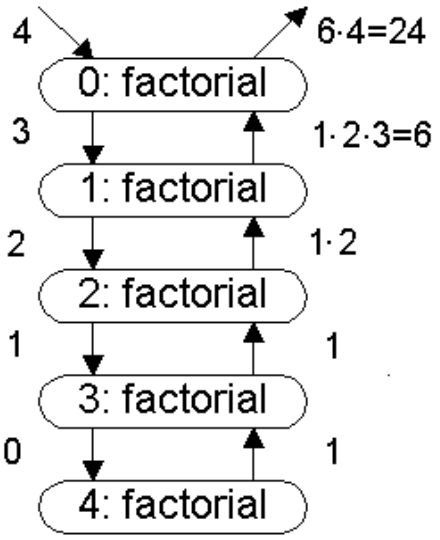
```
(defun factorial  (n)
       (cond ((zerop n) 1)
             (t (* (factorial  (- n 1)) n))))
```

Данная функция непосредственно реализует рекурсивное определение факториала, применяемое в математике. На рисунке 5.6,а изображена схема рекурсивных вызовов при вычислении факториала числа 4. Овалы, представленные на рисунке, изображают вызовы функции FACTORIAL.

Числа, записанные рядом со стрелками, направленными вниз, соответствуют значениям аргументов вызова. Числа при стрелках, направленных вверх, представляют возвращаемые значения. Вычисление факториала числа 4 требует четырех рекурсивных вызовов. Рекурсивные вызовы прекращаются при вычислении факториала нуля. Возвращаемые значения рекурсивных вызовов подставляются в точку предыдущего вызова функции FACTORIAL, формируя конечный результат. Рассмотренная схема представляет собой простую рекурсию. Здесь рекурсивный вызов происходит в одной ветви COND и только один раз.

Процесс рекурсивных вызовов можно исследовать, используя возможности трассировки (рисунок 5.6,б). Включение и выключение механизма трассировки выполняется с помощью входящих в состав интерпретатора директив TRACE и UNTRACE:

```
(trace factorial)
(factorial 4)
(untrace factorial)
```




```
> (trace factorial)
(factorial)
> (factorial 4)
 0: (factorial 4)
   1: (factorial 3)
     2: (factorial 2)
       3: (factorial 1)
         4: (factorial 0)
         4: returned 1
       3: returned 1
     2: returned 2
   1: returned 6
 0: returned 24
> 24
```

а) б)

Рисунок 5.6 – Рекурсивное вычисление факториала и трассировка

Рассмотрим более сложную схему рекурсивных вызовов на примере формирования чисел Фибоначчи. Каждое число последовательности Фибоначчи определяется как сумма двух предыдущих:

1, 1, 2, 3, 5, 8, 13, 21, …

Для того чтобы вычислить очередное число Фибоначчи, необходимо знать два предшествующих. Это легко реализуется с помощью следующей функции:

```
(defun fibonacci (n)
        (cond ((= n 0)1)
                ((= n 1) 1)
                (t (+ (fibonacci(- n 1))
                        (fibonacci (- n 2))))))
```

На рисунке 5.7 изображена схема рекурсивных вызовов при определении четвертого числа Фибоначчи. Каждый вызов функции FIBONACCI сводится к двум другим рекурсивным вызовам этой же функции, т.е. функция реализует схему так называемой *параллельной рекурсии* [48]. Программирование подобной схемы управления вычислительным процессом с помощью итерационных циклов было бы весьма затруднительно. В то же время приведенный пример демонстрирует не только ясность и простоту функции FIBONACCI, но и ее недостатки. Как видно из рисунка 5.7, при вычислении функции происходят повторные вызовы для одних и тех же значений аргументов. К счастью, имеется возможность исключить лишние вызовы [48].

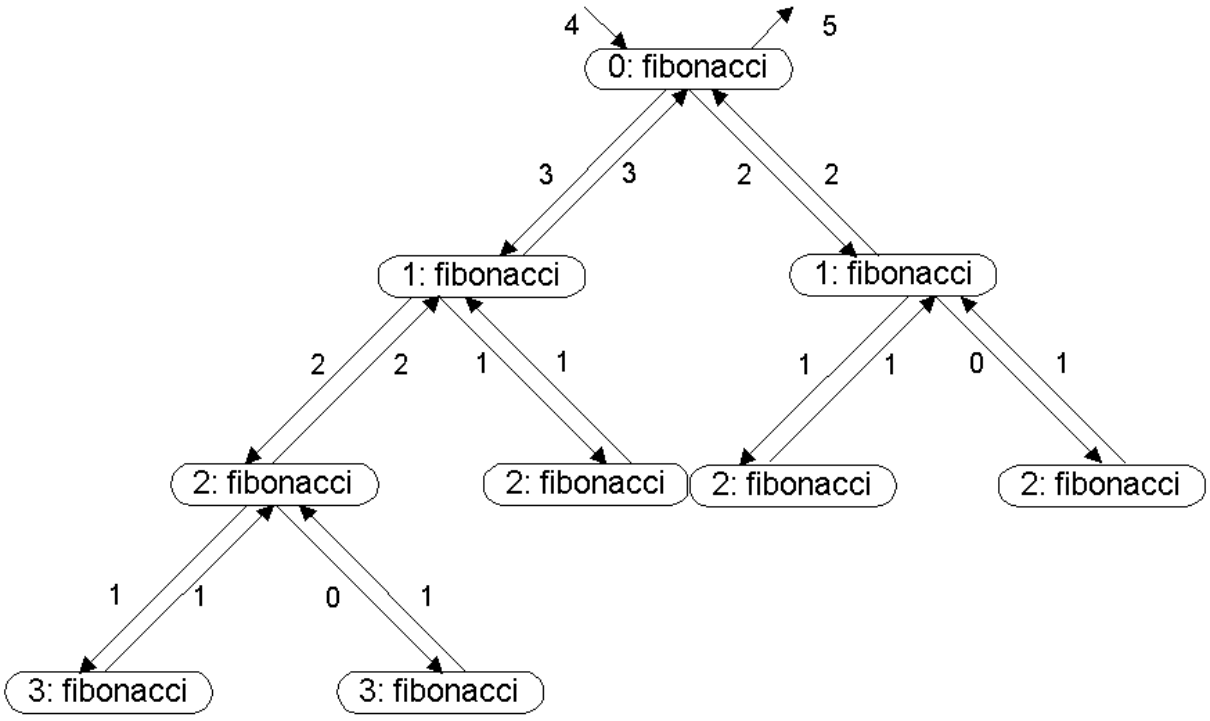


Рисунок 5.7 – Рекурсивные вызовы при вычислении функции FIBONACCI

Рассмотренные примеры касались числовой обработки. Лисп позволяет также эффективно определять рекурсивные функции для обработки символьных данных. Определим упрощенную версию функции MEMBER. Чтобы отличать ее от функции, встроенной в систему назовём ее

MY-MEMBER. Функция проверяет, входит ли элемент X в список У и возвращает подсписок У, начинающующийся с обнаруженного элемента:

```
(defun my-member (x  y)
        (cond ((null y) nil)
                ((equal x (first y)) y)
                (t (my-member x (rest y)))))
```

В функции выполняется сравнение элемента X с первым элементом списка У. Если обнаруживается совпадение, то функция возвращает список У. Если совпадение не обнаружено, то продолжается поиск элемента X в хвосте списка У. Для этого рекурсивно вызывается функция MY-MEMBER. Первое условие формы COND на каждом шаге рекурсии контролирует список У. Если этот список исчерпан, то определяемая функция MY-MEMBER возвратит значение NIL. Функция MY-MEMBER использует простую рекурсию.

Примером функции, выполняющей символьную обработку и основанной на параллельной рекурсии, может служить рекурсивная версия функции MY-INTERSECTION:

```
(defun my-intersection (a b)
        (cond
                ((null a) nil)                                  ;1)
                ((member (first a) b)                           ;2)
                (cons (first a)(my-intersection (rest a) b)))
                (t (my-intersection (rest a) b))))              ;3)
```

Данная функция вычисляет список общих элементов двух списков A и B. В определении функции реализованы три правила:

1) если список A исчерпан, то список общих элементов пустой;

2) если первый элемент списка A входит в список B, то этот элемент добавляется к списку общих элементов списков (REST A) и B; такой список формируется рекурсивным вызовом функции MY-INTERSECTION, применяемой к хвосту списка A и исходному списку B;

3) если первый элемент списка A не входит в список B, то строится список общих элементов из хвоста списка A и списка B с помощью рекурсивного вызова функции MY-INTERSECTION.

Ранее функция MY-INTERSECTION была определена с помощью итерационных циклов. Рекурсивная версия функции ближе к естественному пониманию отношения “общие элементы” и выглядит элегантнее. Однако в общем случае, по быстродействию и эффективности использования памяти, рекурсивные версии функций часто уступают итерационным версиям этих функций.

5.13. Структурно-разрушающие функции

Все рассмотренные до сих пор функции манипулировали точечными парами или списками, не модифицируя их представление в памяти. Например, добавление элемента Z в множество, представленное символом L , не меняет исходного списка L:

```
(setf  l '(a b c d e))→ (A B C D E)
(adjoin 'z l)    → (Z A B C D E)
l → (A B C D E)
```

В Лиспе имеются функции, при помощи которых можно непосредственно изменять структуры, хранящиеся в памяти. Такие функции называют *структурно-разрушающими*. К ним относятся функции NCONC, RPLACA, RPLACD и DELETE.

Для того чтобы лучше понять суть операций, выполняемых этими функциями, рассмотрим представление списков в памяти ЭВМ. Пусть требуется создать два списка L1 и L2, а затем объединить их в один список LONGLIST:

```
(setf l1 '(a b c) l2 '(d e f)) → (D E F)
(setf longlist (append l1 l2)) → ( A B C D E F)
```

Структуры данных, образующиеся в памяти при выполнении указанных функций, показаны на рисунке 5.8.

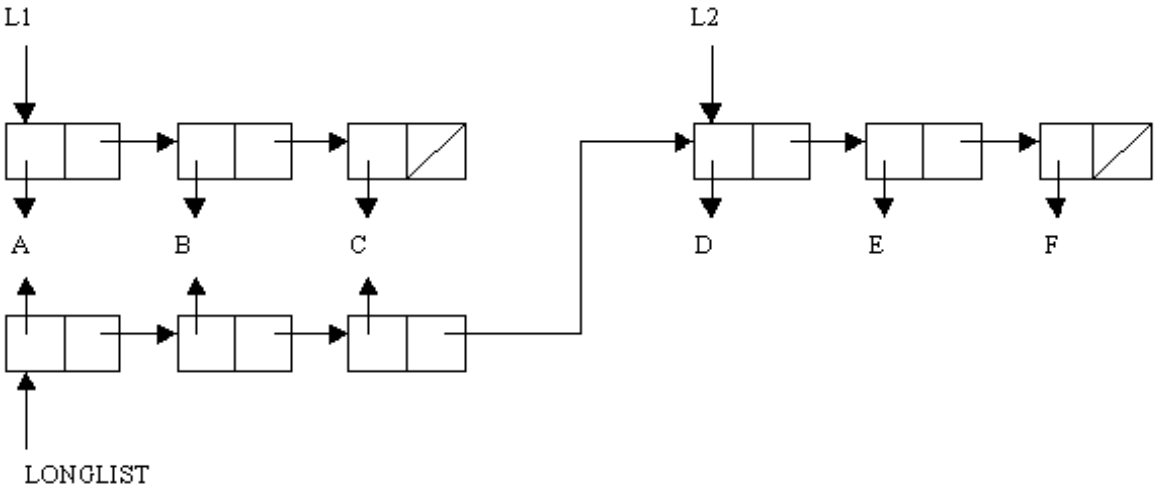


Рисунок 5.8 – Формирование списка LONGLIST с помощью функции APPEND

С помощью первого вызова формы SETF создаются списки (A B C) и (D E F), на которые указывают соответственно указатели L1 и L2. Во втором вызове SETF происходит обращение к функции APPEND, которая ко-

пирует структуру списка L1 и объединяет ее со структурой списка L2. Таким образом, создание нового списка LONGLIST не привело к разрушению исходных списков L1 и L2, т.е.

l1 → (A B C)
l2 → (D E F)
longlist → (A B C D E F)

Функция APPEND определяется в Лиспе через функцию CONS, конструирующую лисповскую ячейку. Именно с помощью CONS создается копия первого списка. На это требуется определенное время и дополнительная память. Если разрушение первого списка несущественно, то создание списка LONGLIST можно выполнить быстрее, применив вместо APPEND функцию NCONC:

(setf l1 '(a b c) l2 '(d e f)) → (D E F)
(setf longlist (nconc l1 l2)) → (A B C D E F)

Соответствующие структуры данных изображены на рисунке 5.9. Функция NCONC выполняет объединение списков без копирования, меняя значение второго указателя в последней ячейке списка L1.

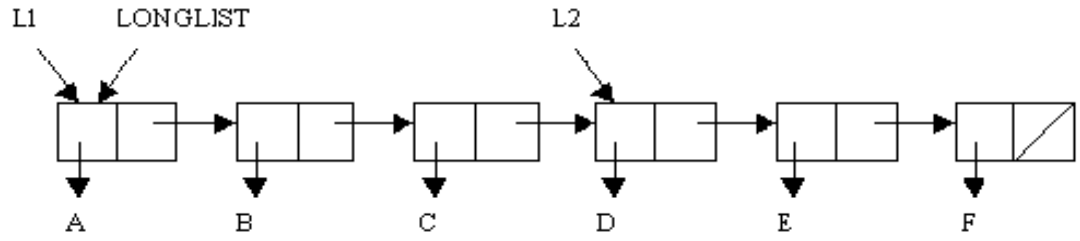


Рисунок 5.9 – Формирование списка LONGLIST с помощью функции NCONC

При запросе значений символов L1, L2 и LONGLIST, интерпретатор вернет следующее:

l1→ (A B C D E F)
l2→ (D E F)
longlist→ (A B C D E F)

В Лиспе имеются функции RPLACA и RPLACD, позволяющие непосредственно изменять указатели лисповских ячеек. Обе эти функции в качестве первого аргумента требуют точечную пару. Второй аргумент может быть символом или точечной парой. Функция RPLACA меняет первый ука-

затель точечной пары таким образом, чтобы он указывал на второй аргумент, заданный в вызове функции:

(setf x ’(a b c)) → (A B C)
(rplaca x ’d) → (D B C)
x → (D B C)

Функция RPLACD выполняет аналогичные действия над вторым указателем точечной пары:

(setf x ’(a b c)) → (A B C)
(rplacd x ’(1 2)) → (A 1 2)
x → (A 1 2)

Функция DELETE является деструктивной версией функции REMOVE. Если функция REMOVE сначала копирует список, а затем удаляет из него заданный элемент, то функция DELETE разрушает исходный список в процессе удаления из него элемента:

(setf s ’(x x y y x y)) → (X X У У X У)
(delete ’x s) → (У У У)
s → (X X У У У)

Здесь значение списка S соответствует некоторому переходному состоянию. Для того чтобы исключить такие неопределенные результаты, необходимо сохранять результирующий список с помощью SETF:

(setf s (delete ‘x s)) → (У У У)
s → (У У У)

При работе с рассмотренными выше функциями следует учитывать два обстоятельства:

1) функции NCONC, RPLACA, RPLACD и DELETE выполняют преобразования непосредственно над исходными списками, и поэтому они весьма эффективны, хотя и являются деструктивными;

2) функции APPEND и CONS безопаснее, но не столь эффективны, так как требуют свободных ячеек памяти.

Необходимо отметить, что в процессе преобразования списков в памяти ЭВМ могут возникать лисповские структуры, на которые потеряны указатели. Например,

(setf s ’(a b c d e)) → (A B C D E)
(setf s ’(x y z)) → (X Y Z)

Здесь, в результате связывания символа S с новым списком (X Y Z), доступ к списку (A B C D E) станет невозможным. Структуры, недоступные программе, представляют собой *мусор*. Для повторного использования ячеек памяти в Лисп-системах предусмотрен *сборщик мусора* (garbage collector – gc), который запускается автоматически. Сборщик мусора может быть запущен и принудительно, путем ввода директивы gc. Он собирает ячейки памяти, на которые нет указателей, в список свободной памяти (free-storage-list).

5.14. Ассоциативные списки и списки свойств

Ассоциативным списком или *а-списком* называется список, состоящий из точечных пар

$$((x_1 \,.\, y_1)(\, x_2 \,.\, y_2) \ldots (x_n \,.\, y_n)),$$

где x_i и y_i– произвольные s-выражения, i=1, … , n.

Первый элемент каждой точечной пары представляет собой некоторый ключ, а второй – данные, ассоциированные с ключом. Ассоциативный список ставит в соответствие ключу x_i выражение y_i . Так как любой список можно представить в виде точечной пары, то список, состоящий из подсписков – а-список. Например,

```
((ivanov 1978 1)(petrov 1979 2) (sidorov 1980 3))
```

Для работы с ассоциативными списками в Лиспе имеется ряд встроенных функций. Функция PAIRLIS строит а-список из списков ключей и данных. Формат функции:

(pairlis *ключи данные* [*а-список*])

Функция добавляет новые точечные пары, образованные из списков *ключи* и *данные*, в начало *а-списка*. Например,

```
(setf x (pairlis '(a b d) '(1 2 3) '((c . 4))) → ((D . 3) (B . 2) (A . 1) (C . 4))
```

Если а-список при вызове функции PAIRLIS не задан, то образуемые точечные пары добавляются в пустой список.

Функция ASSOC выполняет поиск в ассоциативном списке. Она возвращает в качестве значения первую по порядку точечную пару, у которой ключ совпадает с заданным аргументом поиска:

(assoc 'd x) → (D . 3)

В Коммон Лиспе имеется также обратная функция RASSOC, которая ищет по данным ключ

(rassoc 4 x) → (C . 4)

Функция ACONS строит точечную пару из двух своих аргументов и добавляет ее в начало а-списка:

(acons 'a 1 x) → ((A . 1)(D . 3)(B . 2)(A . 1)(C . 4))

Дальнейшим развитием понятия а-список является *список свойств*. В Лиспе с символом можно связать значение. Это значение может быть числом, списком или другим символом. Например, символу можно присвоить значение а-списка:

```
(setf person '((name ivan)
               (age 25)
               (sport football)))
→ ((NAME IVAN)(AGE 25)(SPORT FOOTBALL))
```

В этом случае значение символа можно рассматривать как множество подсписков, характеризующих то или иное свойство символа PERSON. Однако Лисп позволяет символам непосредственно иметь свойства. *Свойство символа* представляется в виде двух элементов: имени свойства и значения свойства. Свойства символа записываются в хранимый вместе с символом список свойств.

Выяснить значение свойства символа можно с помощью функции GET, аргументами которой являются имя символа и имя свойства. Если предположить, что с символом PERSON связан указанный выше список свойств, то мы получим следующие результаты:

```
(get 'person 'name) → IVAN
(get 'person 'sport) → FOOTBALL
(get 'person 'address) → NIL
```

Если имя свойства не найдено в списке свойств, то возвращается значение NIL. Внесение нового свойства и его значения в список свойств или изменение значения существующего свойства в языке Коммон Лисп выполняются с помощью функций GET и формы SETF:

```
(setf (get 'person 'name) 'aleksey) → ALEKSEY
(setf (get 'person 'team) 'spartak) → SPARTAK
```

Удаление свойства и его значения из списка свойств выполняется с помощью функции REMPROP. Аргументы функции – имя символа и имя удаляемого свойства. В качестве результата функция REMPROP возвращает значение T или NIL. NIL возвращается в случае, если у символа нет указанного свойства:

```
(remprop 'person 'team) ⟶ T
(remprop 'person 'address) ⟶ NIL
```

Просмотреть список свойств, связанных с символом можно с помощью функции SYMBOL-PLIST:

```
(symbol-plist 'person) ⟶ (SPORT FOOTBALL AGE 25 NAME ALEKSEY)
```

Отметим, что свойства символов глобальны. Изменение значения символа не влияет на список свойств символа.

5.15. Типы данных

Используемые в программе данные характеризуется типом значений. *Тип* фиксирует, во-первых, множество значений, которые могут принимать объекты, принадлежащие данному типу, и, во-вторых, совокупность допустимых операций.

Ранее уже рассматривались некоторые типы данных языка Лисп: числа, атомы, списки. Классификация основных типов данных языка Лисп представлена на рисунке 5.10.

Типы данных, расположенные ниже по дереву классификации, автоматически входят в вышестоящий тип данных. Поэтому встроенные функции, определенные для расположенного выше по иерархии типа данных, можно применять к типам данных, являющихся его потомками.

В Лиспе используется как явный, так и неявный способ указания типа значения. Например, символы могут вводиться без предварительного описания их типа. Их тип автоматически определяется на основании их позиции в форме, в которой они применяются. Символу можно присвоить значение любого типа.

В Коммон Лиспе можно явно определить тип символа. Для этого применяется специальная форма DECLARE. Например,

```
(declare (type fixnum x y))
```

Здесь символы X, Y определяются как целочисленные значения, лежащие в диапазоне от минус 2^{15} до плюс 2^{15} .

Для проверки типов данных в Лиспе имеется большое число встроенных предикатов (ATOM, CONSP, LISTP, NUMBERP, STRINGP, ARRAYP и др.). Определить принадлежность данных к одному из типов, указанных на рисунке 5.10, можно также с помощью обобщенного предиката проверки типа:

(typep *объект тип-данного*)

Например,

(typep '(a b c) 'list) → T

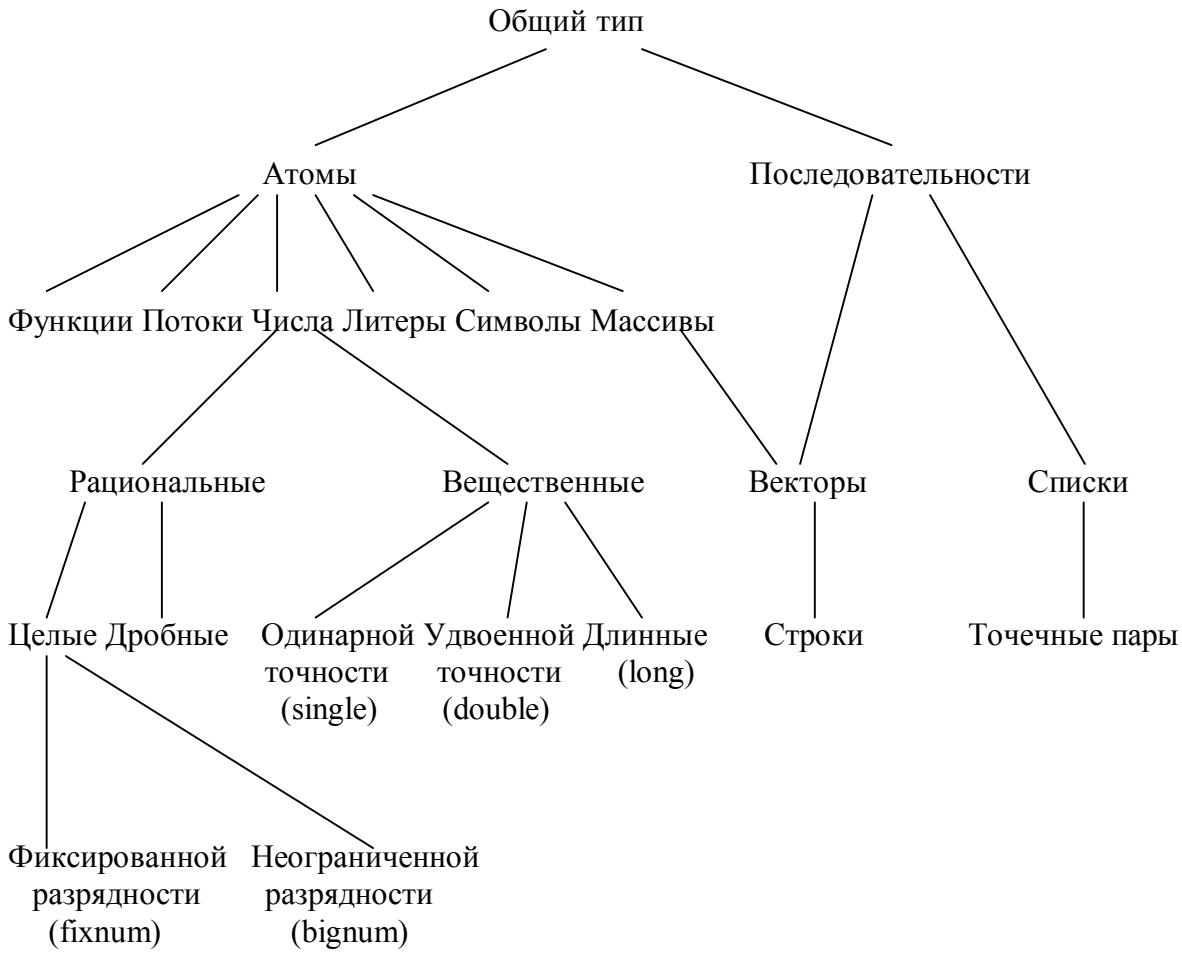


Рисунок 5.10 – Классификация основных типов данных языка Лисп

Выяснить тип объекта позволяет функция TYPE-OF, которая возвращает в качестве результата тип своего аргумента:

(type-of '(a . b)) → CONSP

Получить описание объекта можно с помощью функции DESCRIBE:

(setf x '(a . b))

(describe x) ⟶ (a . b) is a cons

Для преобразования одного типа данных в другой используется обобщенная функция преобразования

(coerce *объект тип*)

Рассмотрим подробнее некоторые из типов данных, представленных на рисунке 5.10.

Символы (**symbol**). Ранее уже отмечалось, что с символом может быть связано значение, список свойств и определение функции. В общем случае символ представляет структурную переменную, состоящую из пяти элементов (рисунок 5.11):

- имени символа (отображается в виде строки);
- указателя значения символа;
- указателя свойств;
- указателя определения функции;
- указателя пакета (пространства имен, к которому принадлежит символ).

Для доступа к каждому из указанных элементов используются следующие функции:

- к имени символа – (symbol-name *символ*);
- к значению символа – сам символ или (symbol-value *символ*);
- к определению функции – (symbol-function *символ*);
- к списку свойств – (symbol-plist *символ*) и (get *символ имя_свойства*);
- к пакету – (symbol-package *символ*).

Изменение составных элементов символа выполняется с помощью следующих средств:

- имя символа – (setf (symbol-name *символ*) *значение*);
- значение символа – (setq *символ значение*), (setf *символ значение*).
- определение функции – (setf (symbol-function *символ*) *лямбда-выражение*) или (defun …);
- список свойств – (setf (symbol-plist *символ*) *список-свойств*), (setf (get *символ имя-свойства*) *значение- свойства*).

Если требуется включить в имя символа пробел или скобку или другой специальный знак, то применяют знак отмены “\” (backslash). Например,

(setf \ (a\ b\) 5) ⟶ 5

Другой способ включения в имя символа указанных знаков состоит в написании имени между двумя вертикальными линиями:

(setf | (a b) | 5) → 5

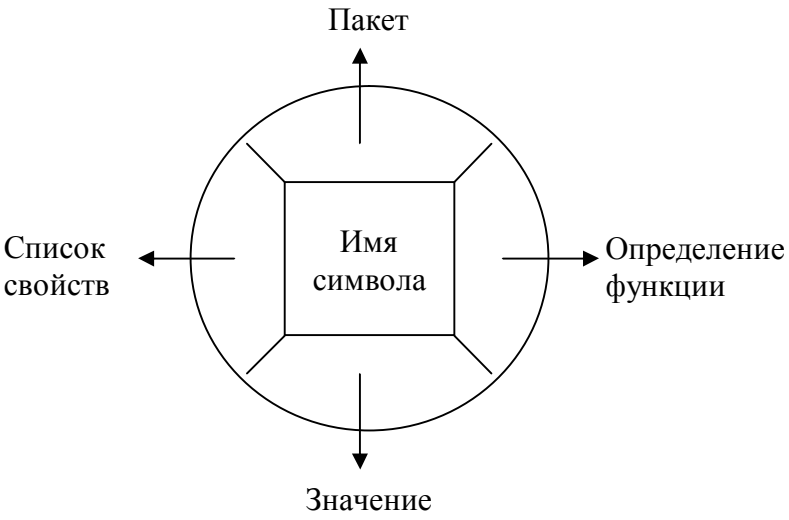


Рисунок 5.11-Элементы, образующие символ

Принадлежность объекта данных к типу данных symbol может быть установлена предикатом:

(symbolp *объект*)

Литеры (character). Литеры представляют собой знаки языка и изображаются в виде #\a. Данная запись соответствует литере ‘a’. Значением литеры является сама литера. Чтобы выяснить, относится ли объект данных к литерному типу, применяется предикат

(characterp *объект*)

Литера может быть преобразована в строку с помощью функции STRING:

(string #\a) → ”A”

Строки (string). Строка состоит из последовательности литер (знаков), заключенных в двойные кавычки. Например, “яблоко”. Строка, записанная таким образом, представляет собой константу, и ее значением является сама строка.

Для работы со строками в Лиспе имеется большой набор функций. Конкатенация (объединение) строк выполняется функцией CONCATENATE. Например,

(concatenate ’string ”abc” ”def”) → ”ABCDEF”

Выделение из строки заданной литеры реализуется функцией CHAR:

(char ”abcde” 3) → #\d
(char ”abc d” 3) → #\SPACE

Здесь #\SPACE представляет неотображаемую литеру “пробел”. Имеются и другие, неотображаемые литеры, например: #\RETURN – перевод строки, #\TAB – табуляция.

Изменить произвольный элемент строки можно с помощью формы SETF:

(setf (char ”торт” 0) #\п) → ”ПОРТ”

Строки можно сравнивать с помощью функций STRING=, STRING>, STRING<. Сравнение строк выполняется в лексикографическом порядке.

Возможны преобразования символа в строку

(string ’symbol) → ”SYMBOL”,

а также двусторонние преобразования “строка – список литер ”:

(coerce ”SYMBOL” ’list) → (#\S #\Y #\M #\B #\O #\L)
(coerce ’(#\A #\T #\O #\M) ’string) → ”ATOM”

Чтобы выяснить, относится ли объект к строковому типу, применяется предикат

(stringp *объект*)

Для создания символа из строки и внесения его в пространство имен применяется функция INTERN:

(intern *строка*)

Векторы (vector). Векторы в Коммон Лиспе представляют одномерные массивы. Элементами векторов могут быть числа, символы, списки или другие векторы. N – элементный вектор отображается в виде последовательности элементов, заключенной в круглые скобки, перед которыми стоит знак #:

#(x_1 x_2 … x_n)

Создание векторов выполняется с помощью функций VECTOR или MAKE-ARRAY. Например,

(vector ’a ’b ’c ’d) → # (A B C D)
(make-array ’(4)) → # (NIL NIL NIL NIL)
(make-array ’(3) :initial-element ’b) → #(B B B)
(make-array ’(4) :initial-contents ’(a b c d)) → #(A B C D)

Указанные функции обычно применяют в комбинации с формой SETF:

(setf v (vector ’a ’b ’c ’d)) → #(A B C D)
v → #(A B C D)
(setf p (make-array ’(4) :initial-contents ’(1 2 3 4)) → #(1 2 3 4)
p → #(1 2 3 4)

Ключевые параметры :initial-element и :initial-contents обеспечивают соответствующие начальные значения элементам вектора.

Доступ к элементам вектора выполняется в соответствии с позицией, которую занимает элемент в векторе. Номер позиции называют индексом. Нумерация начинается с нуля. Для доступа к элементам вектора применяется функция AREF:

(aref v 2) → C
(aref p 0) → 1

Изменение значений векторов выполняется с помощью присваивания:

(setf (aref v 2) 5.14) → 5.14
v → # (A B 5.14 D)

Векторы могут иметь указатели заполнения (fill-pointer). Указатель заполнения позволяет обозначить активные элементы вектора и соответствует наибольшему индексу элемента, который входит в активную часть вектора. Значение указателя заполнения определяется ключевым параметром FILL-POINTER функции MAKE-ARRAY:

(setf stack (make-array '(9) :fill-pointer 0)) → #( ).

Многие функции выполняют действия над векторами с учетом значения указателя заполнения, например, функции VECTOR-PUSH и VECTOR-POP, позволяющие организовать стековый механизм работы с вектором. При добавлении элемента в стек (функция VECTOR-PUSH) он вносится по индексу, соответствующему указателю заполнения, после этого значение указателя заполнения автоматически инкрементируется. В качестве результата функция VECTOR-PUSH возвращает индекс добавленного элемента, а функция VECTOR-POP – значение удаленного элемента. При удалении выполняются обратные операции. Выяснить текущее значение указателя можно с помощью функции FILL-POINTER:

(vector-push 'a stack) → 0
(vector-push 'b stack) → 1
stack → #(A B)
(fill-pointer stack) → 2
(vector-pop stack) → B
(fill-pointer stack) → 1
stack → #(A)

Функция VECTOR-PUSH возвращает значение NIL, если указатель заполнения достиг предельного значения, соответствующего размерности вектора. Функция VECTOR-POP генерирует ошибку, если указатель заполнения равен нулю.

Отметим, что строка – частный случай вектора, элементами которого являются литеры. Поэтому к строкам могут применяться функции, определенные над векторами.

Выяснить принадлежность объекта данных к векторному типу можно с помощью предиката:

(vectorp *объект*)

Последовательности (sequence). Последовательности представляют упорядоченные множества элементов данных. В Коммон Лиспе последовательности обобщают понятие векторов и списков. На элементы последовательности можно ссылаться по их позиции в последовательности. Над последовательностями можно выполнять общие действия, как-то:

- определение длины последовательности (LENGTH);
- объединение последовательностей (CONCATENATE);
- обращение последовательности (REVERSE);
- определение позиции элемента в последовательности (POSITION);
- удаление элемента из последовательности (REMOVE);
- замена элементов последовательности (SUBSTITUTE);
- заполнение последовательности (FILL) и др.

Создание последовательности можно выполнить с помощью функции MAKE-SEQUENCE, имеющей формат:

(make-sequence *тип количество* &KEY :initial-element)

Обязательные параметры функции MAKE-SEQUENCE определяют тип и количество элементов последовательности. Например,

```
(setf  s  (make-sequence 'list  ;тип элемента последовательности
                                5        ;количество элементов
                                :initial-element 'a)); инициализация
→ ( A  A  A  A  A)
```

Доступ к элементу последовательности выполняется вызовом функции ELT:

(elt *последовательность индекс*)

Например, (elt s 4) → A.

Изменить значение элемента последовательности можно с помощью присваивания:

```
( setf  (elt  s  3)  'B) → B
S → (A A A B A)
```

Массивы (array). Массивы в языке Лисп могут иметь произвольное количество измерений, в том числе и ноль. Массив нулевой размерности содержит один элемент. Любая версия Коммон Лиспа должна поддерживать, как минимум, работу с семимерными массивами. Создание массива выполняется функцией MAKE-ARRAY:

```
(setf feld (make-array  '(2 3) :initial-contents '((a b c)(d e f))))
→#2A ((A B C)(D E F))
```

Для доступа к элементам массива применяется функция AREF:

```
(aref feld 0 1) →B
```

Изменение значений элементов массивов выполняется присваиванием по указателю, определяемому с помощью вызова функции AREF:

(setf (aref feld 1 2) 'Z) → Z
feld → #2A((A B C)(D E Z))

Определить размерность массива можно с помощью функций:

(array-dimension feld 1) → 3
(array dimensions feld) → (2 3)

Выяснить, принадлежит ли объект данных к типу массив, можно с помощью предиката:

(arrayp *объект*)

Потоки (stream). В Лиспе операции ввода-вывода выполняются с помощью потоков. Потоки представляют собой хранилища данных, из которых можно читать данные или в которые можно записывать данные. Потоки могут быть связаны с устройствами ввода-вывода, файлами на дисках, строками в памяти. Поток характеризуется направлением (direction) и типом элементов, которые берутся из потока или направляются в поток.

Поток в Лиспе связывается с динамической переменной, значения которой и представляют поток. Имеется несколько стандартных потоков: *STANDARD-INPUT*, *STANDARD-OUTPUT*, *ERROR-OUTPUT*, *TRACE-OUTPUT* и др. Эти динамические переменные определяют для функций ввода (READ) и вывода (PRINT) файлы ввода-вывода. В начале сеанса работы с Лисп системой эти файлы связаны с терминалом.

Если требуется выполнять ввод-вывод из других файлов, то их необходимо открыть. Для этого применяется функция OPEN, упрощенный формат которой можно представить в форме

(open *имя-файла* &KEY
: direction *значение*
: element-type *значение*
: if-exists *значение*
: if-does-not-exist *значение*)

Ключевые параметры имеют следующие значения:

direction – направление передачи данных в потоке; возможные значения: 1) :input – ввод; 2) :output – вывод; 3) :io – двунаправленный поток и др.;

element-type – тип элементов, передаваемых в потоке (по умолчанию character);

if-exists – выполняемые операции, если заданный файл уже существует; возможные значения: 1) :error - ошибка; 2) :override – перезаписать; 3) :append – добавить новые записи в конец; 4) :rename – переименовать и др.;

if-does-not-exist – выполнение операций, если заданный файл отсутствует; возможные значения: 1) :error – ошибка (для файлов открытых в режиме чтения); 2) :create – создать.

Пример вызова функции OPEN:

```
(setf vyvod (open ”extern.dat”  :direction  :output
                                :if-exists  :append
                                :if-does-not-exist     :create))
```

Здесь функция OPEN направляет выходной поток в файл ”extern.dat”, а функция SETF связывает с открытым потоком символ VYVOD, который в дальнейшем используется в качестве имени потока.

Связи стандартных потоков могут быть переопределены с помощью присваивания. Например,

```
(setf  *standard-output* vyvod)
```

После завершения работы с потоком его необходимо закрыть

```
(close vyvod)
```

Выяснить, является ли объект данных потоком, можно с помощью предиката

(streamp *объект*)

5.16. Ввод-вывод

При работе с интерпретатором Лиспа ввод-вывод выполняется автоматически. Интерпретатор читает вводимое пользователем s-выражение (READ), вычисляет его (EVAL) и выводит результаты (PRINT). В простейшем случае функция READ не требует аргументов. После вызова этой функции происходит обращение к стандартному входному потоку и считывание введенного пользователем s-выражения. Введенное выражение возвращается в точку вызова READ.

Определим функцию, которая находит сумму четырех введенных с клавиатуры чисел:

```
(defun sum4 ( )
        (+ (read) (read) (read) (read))
```

Для вызова этой функции необходимо напечатать (sum4), а затем ввести четыре числа:

```
 > (sum4)
1 2 3 4
    → 10
>
```

Если требуется вводить данные из файлов, то сначала необходимо открыть с помощью функции OPEN соответствующий входной поток:

```
(setf vvod (open ”extern.dat” :direction  :input))
```

Для чтения выражений из открытого потока VVOD необходимо сообщить его имя функции READ, т.е.

```
(read vvod)
```

В этом случае произойдет обращение к файлу ”extern.dat”, и из него будет считано одно s-выражение. Результатом вызова функции READ будет s-выражение, введенное из файла.

Для вывода значений в Лиспе используется функция PRINT. Эта функция имеет один аргумент, значение которого выводится в выходной поток. Функция PRINT перед выводом значения осуществляет переход на новую строку, а после печати значения выводит пробел

```
(print  ’(a b c))
(a b c)
 → (A B C)
```

Здесь список (A B C) дважды отображается на экране. Дополнительный автоматический вызов функции PRINT происходит в цикле READ-EVAL-PRINT интерпретатора.

Для вывода значений могут также использоваться функции PRIN1 и PRINC. Функция PRIN1 аналогична PRINT, но не выполняет переход на новую строку и не выводит пробел. Функция PRINC дает возможность напечатать строку без ограничивающих кавычек:

```
(princ ”lisp”) → lisp
```

Для перехода при выводе значений на новую строку можно использовать функцию TERPRI. У функции TERPRI нет аргументов. Если требуется вывести на печать имя символа, содержащее пробелы, скобки или строчные и заглавные буквы, то применяют знак отмены “\” или ограничивающие вертикальные линии “|”. Все рассмотренные функции вывода позволяют осуществлять вывод в выходной поток открытый, с помощью функции OPEN:

```
(setf stran (open ”extern.dat”  :direction :output  :if-exists :append))
(print ’|a b c d| stran) → |a b c d|
(print ’\a\C \  \d stran) → |aC   d|
(princ ’|a b c d| stran) → |a b c d|
```

Здесь вторым аргументом функции является выходной поток STRAN, связанный с файлом EXTERN.DAT.

При работе с файлами удобно использовать макроформу WITH-OPEN-FILE. Она позволяет открыть поток, задать режим работы с ним, выполнить необходимую его обработку и закрыть его:

(with-open-file (*поток имя-файла* {*опция*}*) {*декларация*}* {*форма*}*)

Вызов WITH-OPEN-FILE базируется на использовании функции OPEN. Режимы открытия потока задаются с помощью аргумента *опция*. Они полностью соответствуют ключевым параметрам, указываемым при вызове функции OPEN. Формы вызова WITH-OPEN-FILE вычисляются последовательно. В качестве результата возвращается значение последней формы. В приведенном ниже примере открывается поток с именем STORE, который связывается с файлом TEST1.LSP. В файл циклически записываются s-выражения, вводимые с клавиатуры. Формирование файла завершается, если с клавиатуры вводится атом EOF:

```
(defun writer ( )
(with-open-file (store ”test1.lsp”      : direction  :output
                                        : if-exists    : append
                                        : if-does-not-exist :create )
      (princ ”Введите s-выражение”)
      (terpri)
      (princ ”Для завершения ввода напечатайте EOF”)
      (terpri)
      (do   ((item (read)  (read))) ;чтение s-выражения
      ((eq item ’eof )  ’end-of-session )
      (print item store )   )))
```

Для управления формой вывода в Коммон Лиспе применяется функция FORMAT:

(format *поток шаблон* &REST *аргументы*)

Если поток задан символом T, то функция осуществляет вывод на экран. Шаблон представляет собой строку, которая может содержать коды для управления печатью. Перед управляющими кодами записывается знак “~” (тильда). Если в шаблоне нет управляющих кодов, то функция FORMAT выводит строку шаблона так же, как и функция PRINC. Некоторые управляющие коды и их назначения приведены в таблице 5.2.

Если в строке шаблона встречается символ перехода на новую строку, то при печати будет выполняться переход на новую строку. Приведем пример:

```
(format  t ”Смотри: ~S! ~% A сейчас по-английски: ~R dog ~:P ” ’| A B |  5 )
→ Смотри:  | A B | !
     A сейчас по-английски: five dogs
```

Таблица 5.2 – Коды управления печатью

Код	Назначение
~ %	Новая строка
~*	Пропуск аргумента
~A	Вывод очередного аргумента с помощью PRINC
~S	Вывод очередного аргумента с помощью PRIN1
~nT	Вывод со столбца n
~D	Десятичный вывод
~B	Двоичный вывод
~O	Восьмеричный вывод
~X	Шестнадцатеричный вывод
~F	Вывод в форме с плавающей запятой
~R	Вывод числа словами
~P	Вывод слова во множественном числе
:	Используется совместно с управляющими кодами и уточняет их действие
~:P	Вывод во множественном числе, если предыдущий аргумент это не 1

5.17. Функционалы

Иногда необходимо передать в функцию через ее формальные параметры имена других функций. Такие параметры называют функциональными, а функцию, имеющую такие параметры, – функционалом.

Например, пусть требуется написать функционал, вычисляющий сумму

$$y = \sum_{i=1}^{n} f(i),$$

где $f(i)$ – арифметическая функция одного аргумента. Тогда определение функционала могло бы быть следующим:

```
(defun funsum (func n )
      (do     (( i 1 ( + i 1 ))
              ( result 0 ( + result  ( func i ) )))
              (( =  i  (+ n 1 ))  result )))
```

Здесь формальный параметр FUNC предназначен для передачи имени функции f в функционал FUNSUM. Однако вызов функционала

```
(funsum 'sqrt 5 )
```

завершится ошибкой: “Undefined function : func” (неопределена функция: func). Обусловлено это тем, что с символом FUNC не связано определение функции. FUNC в этом случае является локальной переменной, которая при вызове получает значения SQRT. Поэтому попытка интерпретации FUNC как имени функции, т.е. (func i) завершается ошибкой. Для того чтобы исправить ошибку, необходимо использовать встроенный функционал FUNCALL, который позволяет вызывать другие функции:

(funcall *имя-функции* &REST *аргументы*)

FUNCALL применяет функцию, заданную своим именем, к аргументам. Количество аргументов должно соответствовать применяемой функции.

Переопределим функционал FUNSUM с помощью FUNCALL:

```
(defun funsum ( func n )
        (do     (( i 1 ( + i 1 ))
                ( result 0 ( + result ( funcall func i ))) )
                (( =  i  ( + n 1 ))  result )))
```

Вызов функционала FUNSUM можно выполнять двумя способами:

(funsum ’sqrt 5) → 8.382333

или

(funsum #’sqrt 5) → 8.382333

Второй вызов соответствует *функциональной блокировке*, которая будет рассмотрена позже. Функционал FUNSUM позволяет вычислить сумму значений любой арифметической функции одного аргумента. Например,

(funsum ’exp 5) → 233.2041

или

(funsum #’exp 5) → 233.2041

В Лиспе также имеется встроенный функционал APPLY, который по своему действию аналогичен FUNCALL, но применяется не к отдельным аргументам, а к списку аргументов:

(APPLY *имя-функции список*)

Например,

(apply ’+ ’(1 2 3 4)) → 10

или

(apply #’+ ’(1 2 3 4)) → 10

Функционалы FUNCALL и APPLY называют *применяющими* или *аппликативными*. Функциональные параметры, передаваемые в функцию, внутри неё можно использовать только с помощью вызова применяющих функционалов FUNCALL и APPLY.

Кроме применяющих функционалов, в Лиспе имеется группа MAP-функционалов, которые называются *отображающими функционалами*. Эти функционалы обеспечивают повторное применение функции к списку и, тем самым, преобразуют (отображают) одну последовательность элементов в другую. Общий формат вызова отображающих функционалов можно представить в виде

(MAPx *имя-функции* &REST *список*)

Здесь MAPx представляет имя одного из MAP-функционалов: MAPCAR, MAPLIST, MAPCAN, MAPCON, MAPC, MAPL. Рассмотрим примеры вызовов этих функционалов.

MAPCAR. Этот функционал применяет функцию, заданную именем, к каждому элементу списка и в качестве результата возвращает список:

(mapcar 'sqrt '(1 4 9 2 3)) → (1.0 2.0 3.0 1.414214 1.732051)
(mapcar '+ '(1 4 9 2 3) '(9 6 1 8 7) → (10 10 10 10 10)

MAPLIST. Применяется в тех случаях, когда необходимо повторить вычисления для хвостовых частей списка:

(maplist 'cons '(a b) '(1 2)) → (((a b) 1 2) ((b) 2))

Здесь функция CONS сначала применяется к спискам (A B) и (1 2), а затем к их CDR-частям , т.е. к спискам (B) и (2). Результат вычислений представляется в форме списка.

MAPCAN и MAPCON. Являются аналогом MAPCAR и MAPLIST, но объединяют результаты, получаемые от применения функции к элементам списка, в один список с помощью функции NCONC:

(mapcan 'list '(A B C) '(1 2 3)) → (A 1 B 2 C 3)
(mapcon 'list '(A B C) '(1 2 3)) → ((A B C) (1 2 3) (A B) (1 2) (C) (3))

MAPC и MAPL. Эти функционалы также аналогичны MAPCAR и MAPLIST. Их обычно применяют с целью получения побочного эффекта. В качестве результата они возвращают значение второго аргумента вызова:

(mapc 'set '(x y z) '(1 4 9)) → (x y z)
x → 1

y → 4
z → 9

Рассмотрим пример. Пусть требуется вычислить среднее значение элементов списка (5 3 6 7 2), затем среднее значение списка без первого элемента, т.е. (3 6 7 2) и т.д. Для решения задачи определим сначала функцию AVERAGE, вычисляющую среднее значение элементов списка, а затем применим функционал MAPLIST:

```
(defun average (l)
        (let    ( (z (length l))
                  (sum (apply  #'+ l)) )
                  (float (/ sum z)) ) )

(maplist  #'average  '(5 3 6 7 2)) → (4.6  4.5  5.0  4.5  2.0)
```

Рассмотренные отображающие функционалы удобно представлять в виде таблицы 5.3, обобщающей порядок применения функциональных параметров и способы формирования результатов.

Таблица 5.3 – Отображающие функционалы

Порядок применения	Способы формирования результатов		
	Игнорирование результатов	Формирование результатов в список	Объединение результатов-списков с помощью NCONC
К каждому элементу списка	mapc	mapcar	mapcan
К хвостовым частям списков	mapl	maplist	mapcon

Для работы с последовательностями в Коммон Лиспе существует ряд мощных функционалов: MAP, REMOVE, DELETE, SUBSTITUTE, FIND, POSITION, COUNT. Функционал MAP является обобщением функции MAPCAR. Он применяет заданную функцию к каждому элементу последовательности, но дополнительно позволяет указать тип результата:

(MAP *тип-результата функция* &REST *последовательности*)

Например,

```
(map  'list   #'-   '(1 2 3 4 )) → (-1 -2 -3 -4 )
(map 'string  #'(lambda (x) (if (evenp x ) #\1 #\0)) '(1 2 3 4 ))
→ "0101"
```

Во втором примере функциональный параметр задан с помощью лямбда-выражения, которое применяется к элементам списка (1 2 3 4) и устанавливает, является ли соответствующий элемент списка четным числом. Полученный результат приводится к типу STRING.

Функционалы REMOVE, DELETE и др., упомянутые выше, имеют несколько форм – базовую форму и формы, оканчивающиеся на -IF и –IF-NOT. Указанные функционалы имеют сходные форматы вызова.

В качестве примера рассмотрим формат функционала REMOVE:

(REMOVE *элемент последовательность*
& KEY :from-end *значение*
:test *значение*
:test-not *значение*
:start *значение*
:end *значение*
:count *значение*
:key *значение*)

Функционал удаляет элемент из последовательности и допускает различные варианты удаления с учетом значения ключевых параметров. Формы данного функционала, оканчивающиеся на – IF (-IF-NOT), удаляют элемент, удовлетворяющий (не удовлетворяющий) определенному тесту:

(remove-if *тест последовательность* &KEY *аргументы*)

Здесь *тест* представляет функциональный параметр, а *аргументы* соответствуют ключевым параметрам функционала REMOVE, за исключением: TEST и TEST-NOT.

Ключевые параметры определяют следующее значения:

from-end – задает порядок просмотра элементов последовательности; по умолчанию элементы просматриваются от начала к концу; обратный порядок можно задать, если присвоить параметру :FROM-END значение T;

test – определяет предикат (тест), которому должны удовлетворять обрабатываемые элементы последовательности; по умолчанию применяется предикат EQL;

test-not – определяет предикат, которому не должны удовлетворять обрабатываемые элементы последовательности; по умолчанию применяется EQL;

start – задает начальную позицию элемента, с которого следует начать обработку; по умолчанию 0;

end	– задает конечную позицию элемента, на котором следует прекратить обработку.
count	– определяет количество обрабатываемых элементов последовательности; по умолчанию обрабатываются все элементы.
key	– задает функцию, которая применяется к элементу до тест-проверки и которая позволяет уточнить свойство проверяемого элемента.

Приведем примеры:

```
(remove 3 '(1 2 3 4 3 2 1)) → (1 2 4 2 1)
(remove 3 '(1 2 3 4 3 2 1) :count 1 ) → (1 2 4 3 2 1)
(remove 3 '( 1 2 3 4 3 2 1) :count 1 :from-end t) → (1 2 3 4 2 1)
(remove 3 '(1 2 3 4 3 2 1) :start 2 :end 5) → (1 2 4 2 1)
(remove 3 '(1 2 3 4 3 2 1) :test #'> ) → (3 4 3)
(remove 'A '((A B ) (C D) (A E)) :test #'eq :key #'car)→((C D))
(remove-if #'oddp '(1 2 3 4 3 2 1)) → (2 4 2)
(remove-if-not #'evenp '(1 2 3 4 3 2 1)) → (2 4 2)
```

5.18. Макролитеры (макросы чтения)

Чтение литеры, входящей в алфавит языка, не требует никаких специальных действий. В то же время чтение некоторых особых литер: круглых скобок, точки с запятой, двойных кавычек – приводит к выполнению специальных действий. Литеры, требующие специальных действий, называются *макролитерами* или *макросами чтения*. Соответствие между различными литерами и действиями задается таблицей чтения. Могут существовать различные таблицы чтения. Однако в каждый момент времени рабочая таблица одна. Она определяется значением динамической переменной *readtable*. Таблица чтения доступна пользователю, и он, при желании, может вносить в нее новые интерпретации литер и, тем самым, менять синтаксис языка.

Действие, выполняемое при чтении макролитеры, задается с помощью обычной функции. Запись литеры и определенной для нее функции в таблицу чтения выполняется с помощью функции

(set-macro-character *литера функция*)

Например, действия, выполняемые при чтении литеры “ ’ ”(quote), блокирующей вычисления, могут быть внесены в таблицу чтения следующим образом:

```
(set-macro-character # \'
#'(lambda (stream char) (list (quote quote)(read stream))))
```

К стандартным макролитерам чтения относятся следующие:

макролитеры:: = (|) | ’ | ; | ” | # | ‘ | ,

Здесь литеры “ (“ и “) “ начинают и заканчивают ввод списка или точечной пары, литера “ ’ “ блокирует вычисления, литера “ ; “ начинает комментарий, литера “ ” ” начинает и заканчивает ввод строки.

Литера “ # ” называется диспетчером. Ее значение зависит от дополнительной литеры, следующей после знака # . Например, #’ обозначает функциональное замыкание, #\ - используется для обозначения стандартных литер, #C - используется для обозначения комплексных чисел и т. д. Имеется более двадцати возможных комбинаций литеры диспетчера с другими литерами. Для того чтобы внести в таблицу чтения последовательность, состоящую из литеры диспетчера и дополнительной литеры, и связать с ней необходимую функцию обработки, применяется функция SET-DISPATCH-MACRO-CHARACTER.

Литера " ‘ " называется *обратной блокировкой вычислений*, она действует аналогично обычной блокировке. Но внутри обратно блокированного выражения можно локально отменить блокировку вычислений. Локальная отмена блокировки выполняется с помощью макролитеры “ , “. Для этого литера “ , “ записывается непосредственно перед выражением, для которого необходимо отменить блокировку:

```
(setf  v ’dog  w ’cat) → CAT
‘(,v and ,w) → (DOG AND CAT)
```

Кроме запятой, для отмены блокировки можно использовать комбинацию знаков “ ,@ “:

```
(setf x ’(dog and cat)) → (DOG AND CAT)
‘(i see ,@x) → (I SEE DOG AND CAT)
```

В этом случае выражение, перед которым стоит комбинация знаков “ ,@ “, вычисляется и присоединяется к элементам обратно блокированного списка без ограничивающих его скобок.

Рассмотренные макросы чтения позволяют расширять и изменять синтаксис языка Лисп. Механизм обратной блокировки облегчает написание макросов, рассматриваемых ниже.

5.19. Замыкания

При рассмотрении функционалов было отмечено, что задание функционального аргумента в вызове функционала требует блокировки вычисления такого аргумента, так как он передает имя функции. Данная блокировка может быть сделана либо с помощью формы QUOTE, либо с помощью последовательности знаков #'. Последняя блокировка указывает на то, что записанная после нее форма должна рассматриваться как функция. Аналогично тому, как запись '(*форма*) эквивалентна (quote (*форма*)), так и запись #'(*форма*) эквивалентна записи (function *форма*). Специальную форму FUNCTION называют функциональной блокировкой.

Форма, указанная в вызове (function *форма*), может быть либо символом, либо лямбда-выражением. Если *форма* представлена символом, то значение, возвращаемое таким вызовом, является именем функции, код которой и будет применяться к соответствующим аргументам с помощью функции FUNCALL или APPLY (см. примеры в §5.17). Если *форма* представлена лямбда-выражением, то генерируется так называемое лексическое замыкание (lexical closure). Суть его состоит в том, что к определению функции, которое заданно лямбда-выражением, добавляются связи свободных переменных, входящих в лямбда-выражение, т.е. возникает замыкание функции и некоторого контекста ее определения. Замыкания можно использовать в качестве функционального аргумента. При вызове замыкания значения свободных переменных лямбда-выражения берутся из контекста создания замыкания.

Рассмотрим пример. Определим функцию ADDER-GENERATOR, которая генерирует некоторое число, превышающее на величину Y значение, запомненное в момент лексического замыкания:

```
(defun adder-generator (x) #'(lambda (y) ( + x y )) ) → ADDER-GENERATOR
```

Замыкание можно запоминать, присваивая его какому-нибудь символу (переменной):

```
(setf add-7 (adder-generator 7)) → #<LEXICAL-CLOSURE…>
```

В данном случае замыкание запоминается как значение символа ADD-7. Можно также запомнить замыкание как функцию, которая связана с символом:

```
(setf (symbol-function 'add-3 ) (adder-generator 3 )) → #<LEXICAL-CLOSURE …>
```

В случае первого присваивания свободная переменная X лямбда-выражения получила значение 7, а в случае второго присваивания – 3.

Кроме того, с символом add-3 связано определение функции. Поэтому вызовы данных замыканий выполняются по-разному:

```
(funcall add-7  2) → 9
(add-3  2) → 5
```

Замыкания хорошо подходят для написания различных программ-генераторов. Рассмотрим пример:

```
(defun even-odd-generator (seed)
    (list
        #'(lambda( )
            (setf seed (cond (( evenp seed) ( + seed 2))
                             (t ( + seed 1 )))))
        #'(lambda ( )
            (setf seed (cond (( oddp seed) ( + seed 2))
                             (t ( + seed 1 )))))))
```

Функция EVEN-ODD-GENERATOR позволяет генерировать четные и нечетные числа. Она включает в себя два замыкания. Запомним эти замыкания в переменной FUNS:

```
(setf funs (even-odd-generator 0 )) → #<LEXICALL- CLOSE…>
```

Теперь можно генерировать четные и нечетные числа с помощью следующих вызовов:

```
(funcall (car funs)) → 2
(funcall (car funs)) → 4
(funcall (cadr funs)) → 5
(funcall (cadr funs)) → 7
```

5.20. Структуры

Ассоциативные списки и списки свойств позволяют объединять элементы данных в комплексы для их более адекватного представления в памяти ЭВМ. При работе с комплексами данных выделяют три основные операции: создание (конструирование) комплекса данных; чтение значений элементов данных комплекса; изменение значений элементов данных комплекса.

Применительно к ассоциативным спискам указанные операции выполняются с помощью функций PAIRLIS, ASSOC и формы SETF. В случае списков свойств указанные действия выполняются с помощью функции GET и обобщенного присваивания SETF. Достоинством ассоциативных

списков и списков свойств является то, что они предоставляют возможность доступа к элементам данных не в соответствии с их позицией, а по их имени. Это освобождает программиста от необходимости помнить порядок расположения данных в указанных комплексах данных.

Объединение элементов данных в единое целое в Коммон Лиспе можно также выполнить с помощью составного типа, который называется *структурой*. Структуры языка Коммон Лисп напоминают структуры языка Си или комбинированный тип языка Паскаль. Однако создание, чтение и изменение значений элементов, образующих структуру, выполняется иначе. Для определения структуры в Лиспе используется макрос DEFSTRUCT. Упрощенный формат макровызова DEFSTRUCT можно записать в виде:

(defstruct *имя* {*описания_слотов*}*)

Данный вызов определяет составной тип с заданным именем и набором слотов (полей). Конкретизируем данное определение на примере:

```
(defstruct sportsman
                (name nil)  ; имя
                (sport nil)   ; вид спорта
                (data nil))  ; дата рождения
→ SPORTSMAN
```

В этом определении описывается абстрактная структура SPORTSMAN, которая включает в себя три слота: имя спортсмена (NAME), вид спорта (SPORT), дату рождения (DATA). Каждый слот (поле) структуры инициализируется значением NIL. При необходимости можно инициализировать слоты структуры иными значениями, а также указывать типы их значений.

Вызов макроса DEFSTRUCT приводит к выполнению ряда побочных действий.

1. Генерируется функция-конструктор, имя которой образуется из слова MAKE- и имени структуры. Например, MAKE-SPORTSMAN. Данная функция используется для создания конкретного экземпляра структуры SPORTSMAN:

```
(setf x (make-sportsman))
→ #S(SPORTSMAN :NAME NIL :SPORT NIL :DATA NIL)
```

Здесь знак “#S” обозначает структуру. Слотам конкретного экземпляра структуры можно присвоить значения с помощью ключевых параметров в вызове MAKE-SPORTSMAN . Ключевой параметр представляет собой имя соответствующего слота структуры, перед которым стоит двоеточие:

```
(setf x (make-sportsman : name 'Victor : sport 'football))
→ #S(SPORTSMAN        :NAME VICTOR :SPORT FOOTBALL)
```

2. Символ SPORTSMAN становится именем типа. Принадлежность некоторого объекта программы к типу SPORTSMAN можно проверить с помощью предиката TYPEP:

```
(typep x 'sportsman) → T
```

Если объект не принадлежит типу SPORTSMAN, то вызов предиката TYPEP вернет значение NIL.

3. Генерируется предикат, имя которого образуется из имени структурного типа и строки "-P". Например, SPORTSMAN-P. Предикат проверяет принадлежность объекта программы к введенному структурному типу:

```
(sportsman-p x) → T
```

4. Генерируются функции-селекторы, обеспечивающие доступ к слотам структуры. Имя функции-селектора состоит из имени абстрактного типа и имени соответствующего слота. Например, чтение слотов структуры X можно выполнить с помощью вызовов:

```
(sportsman-name x) → VICTOR
(sportsman-sport x) → FOOTBALL
```

Функции-селекторы применяются также для изменения значений слотов структуры:

```
(setf (sportsman-name x)  'Aleksey) → ALEKSEY
(setf (sportsman-sport x)  'Tennis) → TENNIS
```

Структуры могут определяться иерархически, т. е. одни структуры могут наследовать слоты, определенные в других структурах:

```
(defstruct (sportsman_footballer (: include sportsman ))
                                 (team nil)
                                 (position nil))
```

Здесь определяется структура SPORTSMAN_FOOTBALLER на основе структуры SPORTSMAN, в которую добавляются слоты TEAM и POSITION, предназначенные для хранения названия команды и типовой позиции футболиста на игровом поле, соответственно. При создании экземпляра структуры SPORTSMAN_FOOTBALLER доступны все слоты родительской структуры SPORTSMAN:

```
(setf y (make-sportsman_footballer
                :name ‘Dmitriy
                :sport ‘football
                :team ‘Dinamo
                :position ’centre_forward ))
```

Допускается также вложение структур. Определим структуру DATEOFBIRTH, предназначенную для хранения даты рождения:

```
(deftstruct dateofbirth
                (day nil)       ; день
                (month nil)     ; месяц
                (year nil))     ; год
```

Теперь можно создать экземпляр структуры SPORTSMAN, представив слот DATA в виде структуры:

```
(setf z (make-sportsman
                :name ’Olga
                :sport  ’chess
                :data (make-dateofbirth
                                : day 20
                                : month 10
                                : year 1983 )))
```

Доступ к вложенным слотам выполняется с помощью последовательных вызовов функций-селекторов:

```
(dateofbirth-day       (sportsman-data z)) → 20
(dateofbirth-month (sportsman-data z)) → 10
(dateofbirth-year      (sportsman-data z)) → 1983
```

5.21. Объектно-ориентированное программирование

Определение класса и создание экземпляров класса. Объектно-ориентированная система языка Коммон Лисп (Common Lisp Object System-CLOS) использует понятие родовой или общей функции. Этим она существенно отличается от языков Object Pascal или C++. Объектная модель языка Коммон Лисп включает в себя такие понятия как класс, экземпляр класса, множественное наследование и методы.

Интересно отметить, что понятие класс является неотъемлемой частью системы типов языка Коммон Лисп. Можно, например, выяснить от-

носится ли экземпляр объекта к заданному классу с помощью предиката TYPEP. Более того, многие типы данных – это классы. Например, тип ARRAY соответствует классу ARRAY.

Класс представляет собой объектно-ориентированный инструмент для создания новых абстрактных типов данных. Определение того или иного типа данных сопряжено с заданием множеств значений и определением допустимых операций над этими множествами. Поэтому можно сказать, что *класс* – это объект программы, который содержит (инкапсулирует) в себе данные и функции по их обработке.

Определение класса выполняется с помощью макроса DEFCLASS и в общих чертах напоминает определение структуры:

(defclass *имя-класса* ({*имя-суперкласса*}*)
({*описание-слота*}*))

Имя класса представляет собой символ. За именем класса следует список имен суперклассов, которые выступают в роли родительских классов для определяемого класса. Это даёт возможность дочерним классам наследовать свойства родительских классов. Список имен суперклассов может быть пустым. Описание слота представляет собой имя слота или список, состоящий из имени слота и его опций:

описание-слота::=*имя-слота* | (*имя-слота* {*опции-слота*}$^{+}$)

Опции слота обеспечивают инициализацию значений слота, задание функций, обеспечивающих доступ к значениям слота, и др. Определим в качестве примера класс MOVING-OBJECT (подвижный объект), который будет характеризоваться своими координатами (слоты X-POS и Y-POS), скоростью (слоты X-SPEED и Y-SPEED), массой (слот MASS):

```
(defclass moving-object( )
                (x-pos  y-pos  x-speed  y-speed mass ))
→ #<STANDARD-CLASS MOVING-OBJECT>
```

Для создания экземпляра класса применяется конструктор MAKE-INSTANCE. Аргументом конструктора является имя класса. В качестве результата конструктор возвращает экземпляр класса. Связать некоторый символ с экземпляром класса можно с помощью формы setq:

```
(setq mv1 (make-instance 'moving-object))
→ #< MOVING-OBJECT @ #x20ae2c22>
```

Синтаксис опций слота в макроформе DEFCLASS определяется с помощью следующей формулы:

опции-слота::=	:reader	*имя_функции_чтения*
	:writer	*имя_функции_записи*
	:accessor	*имя_функции_доступа*
	:initarg	*имя_функции_инициализатора*
	:initform	*форма*
	:allocation	*место_размещения*

Опции слота объявляются с помощью ключевых параметров, которые являются необязательными. Перед ключевым параметром ставится двоеточие. Опции слота определяют имена функций, обеспечивающих обработку значений слота, а также определяют правила вычисления начальных значений слота. Опция READER объявляет функцию, возвращающую значение слота. Опция WRITER объявляет функцию, изменяющую значение слота. Опция ACCESSOR комбинирует возможности опций READER и WRITER. Она определяет функцию, которая может читать значения слота или использоваться, совместно с формой SETF, для изменения значений слота. Функцию, объявляемую с помощью ключа ACCESSOR, будем называть функцией доступа к слоту. Опция INITARG определяет имя, с помощью которого можно задавать начальное значение слота, а опция INITFORM определяет форму, с помощью которой при каждом вызове конструктора MAKE-INSTANCE вычисляются начальные значения слота. Опция ALLOCATION определяет способ распределения памяти для слота. Память, выделяемая под слот, может быть связана либо с классом, либо с экземпляром класса. Следовательно, *место_размещения* может иметь значение либо CLASS (класс), либо INSTANCE (экземпляр). Если в качестве места размещения указан класс, то доступ к слоту выполняется как к участку разделяемой памяти. Поэтому все экземпляры класса будут иметь одно и то же значение слота. Если в качестве места размещения слота указан экземпляр класса, то слоты экземпляров класса будут иметь уникальные значения. Если опция ALLOCATION не задана, то *место_ размещения* получает значение INSTANCE.

Следующий пример демонстрирует применение указанных опций:

```
(defclass moving-object()
((x-pos      :initarg  :x-pos      :initform 0. :accessor get-x-pos)
(y-pos       :initarg  :y-pos      :initform 0. :accessor get-y-pos)
(x-speed     :initarg  :x-speed    :initform 0. :accessor get-x-speed)
(y-speed     :initarg  :y-speed    :initform 0. :accessor get-y-speed)
(mass        :initarg  :mass       :initform 0. :accessor get-mass )))
→#< STANDARD-CLASS MOVING-OBJECT>
```

В этом случае для задания начальных значений слотов в вызове MAKE-INSTANCE могут использоваться ключевые аргументы x-pos, y-pos и т.д. Если эти аргументы не будут заданы, то начальное значение слота будет определяться с помощью выражения, стоящего после ключевого слова INITFORM. Доступ к значениям слотов выполняется с помощью функций GET-X-POS, GET-Y-POS и т.д. Кроме этого, для доступа к значениям слотов можно использовать функцию SLOT-VALUE. Аргументами функции являются имя экземпляра класса и имя слота. В качестве результата возвращается значение слота:

```
(setf (slot-value  mv1 'x-pos) 40)
```

Рассмотрим пример. Пусть требуется запрашивать начальное значение слота у пользователя, если оно не задано при вызове конструктора MAKE-INSTANCE. Определим функцию READ-SLOT-VALUE, которая будет вводить значение слота:

```
(defun read-slot-value (zapros)
    (print zapros)(read))
```

Аргумент ZAPROS предназначен для вывода на экран строки запроса. Теперь переопределим класс MOVING-OBJECT

```
(defclass moving-object()
   ((x-pos      :initarg   :x-pos
                :initform (read-slot-value "Введите координату X")
               :accessor get-x-pos)
   (y-pos       :initarg :y-pos
                :initform (read-slot-value "Введите координату У")
                :accessor get-y-pos)
   (x-speed     :initarg :x-speed
                :initform (read-slot-value "Введите скорость  X")
                :accessor get-x-speed)
   (y-speed     :initarg :x-speed
                :initform (read-slot-value "Введите скорость У")
                :accessor get-y-speed)
   (mass        :initarg :mass
                :initform (read-slot-value "Введите массу")
                :accessor get-mass)))
```

и создадим экземпляр класса с помощью MAKE-INSTANCE и SETF:

```
(setq mv2 (make-instance 'moving-object
                :x-pos 35 :y-pos 40))
"Введите скорость  X" 10
"Введите скорость  У" 20
"Введите массу" 250000
```

→ #< MOVING-OBJECT @ #x20bfddaa>

При запросе значений слотов X-SPEED и Y-SPEED получим:

(get-x-speed mv2) → 10
(get-y-speed mv2) → 20

Определение методов. В объектно-ориентированном программировании с понятием класс связывают не только информацию о некотором объекте, но и правила ее обработки, оформленные в виде программ. Такие программы называют *методами* класса.

В языке Коммон Лисп методы представляются функциями и определяются вне класса с помощью макроформы DEFMETHOD, которая вызывается с помощью следующей упрощенной записи:

(defmethod *имя_функции специальный_лямбда_список*
{*форма*}*)

Вызов формы DEFMETHOD напоминает вызов формы DEFUN. Отличие состоит в том, что в вызове формы DEFMETHOD присутствует *специальный_лямбда_список.* Специальный лямбда список подобен обычному списку, за исключением одного – его формальные параметры могут представлять пары *(символ спецификатор_символа)*, где *символ* – это имя формального параметра, а *спецификатор_символа* определяет класс принадлежности формального параметра. Наиболее общий класс в языке Коммон Лисп представляется атомом T. Если спецификатор символа не задан, то параметр относится по умолчанию к классу T.

В качестве примера определим методы для класса MOVING-OBJECT, позволяющие определять скорость движения объекта (VELOCITY), направление движения (DIRECTION) и кинетическую энергию (EKIN):

```
(defmethod velocity ((mv moving-object))
            (sqrt (+ (*(get-x-speed mv)(get-x-speed mv))
                     (*(get-y-speed mv)(get-y-speed mv)) )))

(defmethod direction ((mv moving-object))
            (atan (get-y-speed mv)(get-x-speed mv)) )

(defmethod ekin ((mv moving-object))
            (* 0.5 (get-mass mv)(velocity mv)(velocity mv)) )
```

Как видно из приведенных примеров, связь метода и класса устанавливается посредством специального лямбда-списка.

Иногда различные классы характеризуются сходным поведением. Например, рассматривая перемещение на экране точки и круга, легко заметить сходство этих операций. В этом случае целесообразно определить функцию, которую можно было бы применять к разным классам. В языке Коммон Лисп для решения этой задачи применяется *общая* или *родовая* функция (generic function). Действия, выполняемые общей функцией, зависят от типов аргументов, заданных при её вызове. Можно говорить о том, что аргументы переопределяют поведение общей функции, и поэтому ее иногда называют переопределяемой функцией. Общая функция – это функция, на основе которой определяются один или несколько методов, каждый из которых посредством специального лямбда-списка связывается со своим классом.

Общая функция может быть определена либо с использованием формы DEFMETHOD, либо с помощью формы DEFGENERIC.

Форма DEFGENERIC позволяет одновременно определять несколько методов, включаемых в общую функцию. Форма DEFMETHOD предоставляет возможность отдельного определения каждого метода. Каждый новый вызов DEFMETHOD добавляет очередной метод к определению общей функции.

Синтаксис формы DEFGENERIC задается следующей упрощенной формулой:

(defgeneric *имя_функции лямбда_список*
{*описание_метода*}*)

описание метода::=(:method *специальный_лямбда_список*
{*форма*}*)

Аргументы формы DEFGENERIC заданы именем функции, лямбда списком и описаниями методов. Описание метода включает в себя *специальный_лямбда_список*. Каждый специальный лямбда-список по количеству параметров должен совпадать с основным лямбда-списком формы DEFGENERIC.

В качестве примера общей функции определим функцию, вычисляющую площадь различных геометрических фигур. Определим сначала классы, соответствующие геометрическим фигурам:

```
(defclass rectangle()        ;определение класса - прямоугольник
          ((height :initarg :height :accessor get-height)
          (width    :initarg :width  :accessor get-width)) )
(defclass circle-1() ; определение класса - круг
          ((radius :initarg :radius :accessor get-radius)) )

(defgeneric area(figure)   ;общая функция AREA
```

```
(:method ((figure rectangle))  ;метод для прямоугольника
                (* (get-height figure)
                (get-width figure)) )
(:method        ((figure circle-1))  ;метод для круга
                        (* (get-radius figure)
                        (get-radius figure) pi)) )

(setf rectangle1 (make-instance 'rectangle  :height 5  :width 5))
(setf circle1 (make-instance 'circle-1  :radius 10))
```

Теперь можно использовать функцию AREA для вычисления площади различных фигур:

```
(area rectangle1) → 25
(area circle1) → 314.159265
```

Необычным для общей функции является то, что в нее можно добавлять новые методы. Это можно выполнять с помощью формы DEFMETHOD. Добавим в функцию AREA метод, позволяющий вычислить площадь прямоугольного треугольника:

```
(defclass triangle()
        ((side-a :initarg :side-a :accessor get-side-a)
         (side-b :initarg :side-b :accessor get-side-b) ))
(defmethod area ((figure triangle))
        (* (get-side-a figure)(get-side-b figure) 0.5))
```

Вызов DEFMETHOD не отменяет предыдущих определений функции AREA, он лишь добавляет новый метод вычисления площади в общую функцию:

```
(setf triangle1 (make-instance 'triangle
                                        :side-a 5
                                        :side-b 3))
(area triangle1) → 7.5
```

Наследование. Классы в языке Коммон Лисп упорядочены в иерархическую структуру. Это позволяет дочерним классам наследовать свойства родительских классов. Родительские классы в Коммон Лиспе называют суперклассами. Дочерние классы могут наследовать свойства сразу нескольких родителей, т.е. допускается множественное наследование.

Определим класс SHIP, представляющий судно. При этом будем исходить из того, что судно обладает свойствами подвижного объекта (класс MOVING-OBJECT), а также для него характерны свойства, приписываемые объектам, построенным человеком. Объекты построенные (сконструированные) человеком можно представить в виде класса CONSTRUCTED-OBJECT:

```
(defclass constructed-object()
    ((constructor :initarg :constructor :accessor constructor)
     (date-of-construction :initarg :date-of-construction
                           :accessor date-of-construction)
      (next-date-of-revision :initarg :next-date-of-revision
                           :accessor next-date-of-revision)))
```

С данным классом будет связан метод, определяющий время эксплуатации объекта в днях до следующей проверки, начиная с текущего момента:

```
(defmethod days-to-go ((co constructed-object))
            (truncate (- (next-date-of-revision co)
                         (get-universal-time)) 86400))
```

Здесь GET-UNIVERSAL-TIME – встроенная функция языка Коммон Лисп, возвращающая значение Всемирного времени. Число 86400 обозначает количество секунд в сутках.

Теперь определим класс SHIP (судно), дополнительно указав слоты названия судна, порта приписки и фамилии капитана:

```
(defclass ship (moving-object constructed-object)
            ((name :initarg :name :accessor name)
             (native-harbour :initarg :native-harbour
                                  :accessor native-harbour)
             (captain :initarg :captain :accessor captain)))
```

При создании экземпляра объекта в вызове MAKE-INSTANCE необходимо определить начальные значения всех атрибутов:

```
(setf ship1 (make-instance 'ship
                           :x-pos 1289
                           :y-pos 311
                           :x-speed 12
                           :y-speed 28
                           :mass 250000
                           :constructor "Blohm"
                           :date-of-construction 312348876
                           :next-date-of-revision  4045060272
                           :name "AMOCO"
                           :native-harbour "Hamburg"
                           :captain "Ole Jensen"))
```

В данном примере символ SHIP1 представляет экземпляр класса SHIP, который унаследовал от своих родительских классов MOVING-OBJECT и CONSTRUCTED-OBJECT все их слоты (атрибуты) и методы. Поэтому вызов (ekin ship1) вернет значение $1.16*10^8$, хотя метод EKIN непосредственно не связан с классом SHIP.

В рассмотренном примере с классами MOVING-OBJECT и CONSTRUCTED-OBJECT не были связаны одноименные методы, поэтому вопросы о порядке наследования методов не возникали. Если с родительскими классами связаны одноименные методы, то необходимо определить порядок наследования методов. Этот вопрос в Коммон Лиспе решается с помощью списка предшествований, который определяет общую иерархию классов. Список предшествований упорядочен в направлении от ближайших, по времени определения, классов к классам, определенным ранее. Кроме этого, если дочерний класс наследует свойства нескольких родительских классов, то они просматриваются в порядке их упоминания в вызове DEFCLASS.

На рисунке 5.12 изображена иерархия классов для рассмотренного выше примера. Стрелками показан порядок просмотра суперклассов.

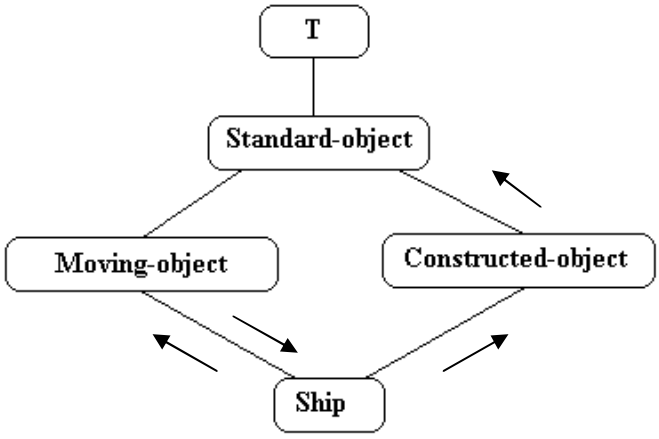


Рисунок 5.12 – Иерархия классов

Указанный порядок просмотра соответствует списку предшествований:

(ship, moving-object, constructed-object, standard-object ,t)

Объектно-ориентированная система языка Коммон Лисп (CLOS) весьма развита. Здесь были рассмотрены только ее основные возможности. За рамками рассмотрения оказались дополнительные методы, комбинации методов, мета-классы и др.

5.21. Макросы

Аналогично функциям макросы позволяют упростить написание программ, в которых встречаются повторяющиеся участки кода. Поэтому синтаксис определения макроса напоминает определение функции:

(defmacro *имя лямбда-список* {*форма*}*)

Однако макрос имеет существенные отличия от функций. Во-первых, вычисление вызова макроса состоит из двух этапов. На первом этапе осуществляется вычисление тела макроопределения с аргументами, заданными при вызове. В отличие от функций аргументы макровызова не вычисляются, а подставляются в тело макроопределения в той форме, в которой они были заданы. Благодаря этому на первом этапе выполняется построение некоторой формы, соответствующей определению макроса. Этот этап называют *макрорасширением*, или раскрытием макроса. На втором этапе происходит вычисление раскрытой формы макроса, и в качестве результата возвращается значение этой формы. Во-вторых, на этапе макрорасширения действуют связи, соответствующие текстуальному окружению макроопределения, т.е. макрорасширение аналогично вычислению функции в отношении связей аргументов и переменных. Однако во время вычисления расширенной формы макроса лексические связи, устанавливаемые в макроопределении, уже не действуют.

Рассмотрим пример, демонстрирующий отличие макроса от функции. Пусть требуется запрограммировать операцию добавления элемента в стек. Решим эту задачу, определив сначала функцию, а затем макрос. Функцию, добавляющую элемент в стек, можно определить следующим образом:

```
(defun our-push (element stack)
        (setf stack (cons element stack)))
```

Следующий вызов функции OUR-PUSH

```
(setf w '(a b c)) ⟶ (A B C)
(our-push 11 w) ⟶ (11 A B C)
```

показывает, что она возвращает необходимый результат. Вместе с тем, если попытаться вычислить значение переменной W, то получим:

```
w ⟶ (A B C)
```

Это объясняется тем, что фактический параметр W передается в функцию OUR-PUSH по значению. Переменная STACK является локальной, и ее значение недоступно вне функции OUR-PUSH.

Решим поставленную выше задачу посредством макроса:

```
(defmacro our-push (element stack)
        (list 'setf stack (list 'cons element stack)))
```

На этапе расширения данного макроса возникает форма

```
(setf stack (cons element stack))
```

С целью облегчения чтения макрорасширений и упрощения написания макросов, применяют обратную блокировку. Например, приведенное выше макроопределение можно также записать в виде:

```
(defmacro our-push (element stack)
        '(setf ,stack (cons ,element ,stack)))
```

Здесь обратно-блокированный список задает прежнее макрорасширение, которое, безусловно, воспринимается легче, чем в первом случае.

Теперь макровызов OUR-PUSH не только вернет правильный результат, но и изменит глобальное значение W:

```
(setf w '(a b c)) → ( a b c)
(our-push 11 w) → (11 a b c)
w → (11 a b c)
```

Это происходит потому, что локальные связи формальных параметров ELEMENT и STACK действуют только на первом этапе вычисления макроса, т.е. на этапе макрорасширения. На этапе вычисления раскрытой формы макроса действуют внешние связи указанных параметров.

Макросы представляют мощный инструмент вычислений, который позволяет определять новые синтаксические формы, новые типы данных, программы, формирующие другие программы, и др. Однако при написании макросов могут возникать неожиданные макрорасширения, которые приводят к ошибкам. Поиск ошибок в определении макроса намного сложнее, чем поиск ошибки в функции. Для тестирования макросов существуют специальные функции MACROEXPAND-1 и MACROEXPAND, которые строят расширения макровызова:

(macroexpand-1 *макровызов*)

Например,

```
(macroexpand-1 '(our-push  11 W))
→ (setf w (cons 11 w))
T
```

Функция MACROEXPAND-1 выполняет макрорасширения один раз, а функция MACROEXPAND – многократно, т.е. она позволяет раскрывать вложенные макроформы.

При написании макросов полезно соблюдать следующую последовательность действий. Во-первых, записать макровызов в желаемой форме. Например,

```
(our-pop stack)
```

Во-вторых, определить выражение, выполняющее необходимые операции по преобразованию данных. Например,

```
(prog1 (car stack) (setf stack (cdr stack))
```

В данном случае выполняются два действия. Сначала выделяется первый элемент списка, представленного переменной STACK, а затем в ней сохраняется хвост списка. В качестве результата форма PROG1 возвратит значение первого элемента списка, т.е. в данном случае извлекается элемент из стека. В-третьих, записать макроопределение в соответствии с определенным выше выражением. Например,

```
(defmacro our-pop (stack)
        '(prog1 (car ,stack)(setf ,stack (cdr ,stack))))
```

Здесь для записи выражения использовалась обратная блокировка. Это позволяет определить макрос в форме, близкой к его раскрытому виду.

5.23. Пакеты

Для контроля доступа к различным объектам программ в языке Коммон Лисп используются пакеты. Пакет представляет собой пространство имен, в котором устанавливается соответствие между именами и символами. В Коммон Лиспе определенно несколько стандартных пакетов. Однако в каждый момент времени только один пакет является текущим. Текущий пакет определяется значением специальной переменной *package*. При чтении имен поиск символов выполняется в пакетах. В программах можно использовать символы, которые определенны не в текущем пакете. В этом случае перед символом через двоеточие записывают префикс, представляющий имя пакета, например:

```
my-package: list-of-date
```

Здесь LIST-OF-DATE представляет символ, определенный в пакете MY-PACKAGE.

Наряду с полным именем, каждый пакет может иметь сокращенное мнемоническое имя, которое приписываются пакету в момент его создания.

Символы, содержащиеся в пакете, подразделяются на два класса: внешние и внутренние. Внешние символы представляют интерфейсную часть пакета и видимы со стороны других пакетов. Символы становятся внешними, если они экспортируются за пределы пакета. Внутренние символы – это те символы, которые доступны внутри пакета и невидимы за его пределами.

Ранее было отмечено, что с символом в языке Коммон Лисп связаны различные атрибуты: имя, значение, функция, список свойств и пакет, в котором этот символ введен. Для того чтобы выяснить значение атрибута пакета, применяется функция SYMBOL-PACKAGE. Например,

```
(symbol-name 'A) ⟶ "A"
(symbol-package 'A) ⟶ #<PACKAGE "USER"…>
```

В состав языка Коммон Лисп входят четыре стандартных пакета.

COMMON-LISP-USER − (мнемоническое имя CL-USER) пакет, который становится текущим в момент запуска Лисп-системы.

COMMON-LISP − (мнемоническое имя CL) содержит примитивы системы Коммон Лисп (за исключением ключевых слов), в нем определены такие внешние символы, как : CAR, CDR, *PACKAGE* и др.

SYSTEM − (мнемоническое имя SYS) содержит символы, зависящие от реализации языка.

KEYWORD − (мнемонического имени не имеет) содержит символы, представляющие ключевые слова. Например, ключевые параметры лямбда-списка. Символы, введенные в этот пакет, автоматически становятся внешними и представляют собой константные значения. Поэтому при записи ключевых слов в вызовах функций не требуется блокировки вычислений, т.е. вместо

```
':some-symbol
```

следует писать

```
:some-symbol
```

Для того чтобы создать и внести некоторый символ в пакет, применяется функция INTERN:

(intern *строка* &OPTIONAL *пакет*)

Функция вносит символ, имя которого представляется аргументом *строка*, в указанный пакет (по умолчанию символ вносится в текущий пакет). Введенный символ рассматривается как внутренний символ пакета. Если символ вносится в пакет KEYWORD, то он рассматривается как внешний символ. В качестве результата функция возвращает два значения: первое значение представляет имя символа, а второе – свойства добавленного символа. Если добавленный символ отсутствовал в пакете, то в качестве второго значения возвращается NIL. Если добавленный символ уже имелся в пакете, то второе значение может быть равно:

- :internal – добавляемый символ имеется в пакете, и он определен как внутренний;
- :external – добавляемый символ уже имеется в пакете, и он определен как внешний;
- :inherited – добавляемый символ уже имеется в пакете, и он унаследован из другого пакета.

Например,

```
(intern  "NEW-SYMBOL"  "KEYWORD") → :| NEW-SYMBOL |,  NIL
(intern "NEW-SYMBOL" "KEYWORD") → :|NEW-SYMBOL|,  EXTERNAL
```

Символ можно создать, не включая его в какой-либо пакет. Для этого применяется функция MAKE-SYMBOL. Символ, созданный таким образом, не имеет ни значения, ни списка свойств. С ним не связано определение какой-либо функции. При выводе такой символ помечается знаком “ # ”. Например,

```
(setf x ( intern "FOOL")) → FOOL
(setf y ( make-symbol "FOOL")) → #:FOOL
```

Здесь значениями символов X и Y является символ с именем FOOL. Но на этом сходство указанных символов заканчивается. X представляет символ, включенный в систему, а Y – нет.

Генерировать новые неповторяющиеся имена символов можно с помощью функции GENSYM. Если эта функция вызывается без параметров, то формируемые имена символов представляются в виде: #:G5, #:G6 и т.д. Буквенными обозначениями и номером символа можно управлять через параметры функции GENSYM:

```
(setq symbol1 (gensym)) → #:G324
(symbol-package symbol1) → NIL
(setq symbol2 (gensym "T")) → #:T325
(setq symbol3 (gensym 10)) → #:G10
```

Исключить символ из системы можно с помощью функции UNINTERN:

```
(setf fool 11.1) → 11.1
(setf x 'fool) → FOOL
(unintern 'fool) → T
x → #:FOOL
```

Если один из пакетов использует другой пакет, то все внешние символы наследуются первым пакетом. При этом они рассматриваются как внутренние.

Символы можно экспортировать из пакета и импортировать в пакет. Для этого применяются функции EXPORT и IMPORT. Формат вызова функции EXPORT:

(export *список_символов имя_пакета*)

Символы, заданные в списке и доступные в указанном пакете, становятся внешними. Следовательно, данные символы становятся доступными во всех пакетах, использующих пакет, указанный в вызове функции EXPORT.

Функция IMPORT добавляет в заданный пакет символ или символы:

(import *список_символов имя_пакета*)

Пакеты могут наследовать внешние символы других пакетов. Достигается это с помощью вызова

(use-package имя_пакета)

Создание пакета можно выполнить с помощью функции MAKE-PACKAGE или с помощью функции DEFPACKAGE. Функция MAKE-PACKAGE создает новый пакет с заданным именем и, если необходимо, с дополнительными символическими именами (nicknames):

```
(make-package "temporal" :nicknames ("TEMP", "temp"))
→#<PACKAGE "TEMPORAL">
```

Введение пакетов в Коммон Лиспе позволяет использовать модульный подход программирования. Для внесения в память системы определений из соответствующего пакета можно переопределить значения переменной *PACKAGE*:

```
(setf *package* (make-package 'new-package))
 → #<PACKAGE "NEW-PACKAGE"
```

В этом случае становятся доступными все определения, сделанные в модуле NEW-PACKAGE. Если необходимо обратиться к содержимому уже имеющегося пакета, то следует воспользоваться функцией FIND-PACKAGE:

```
(setf *package* (find-package 'old-package))
 → #<PACKAGE "old-package">
```

В результате последнего вызова текущим пакетом, используемым системой, будет пакет OLD-PACKAGE.

5.24. Множественные значения

В Коммон Лиспе имеются функции, которые возвращают несколько значений. При выводе на терминал множественные значения располагаются в нескольких строках. Например,

```
(truncate 5 3)
 → 1
2
```

Если данный результат использовать в качестве аргумента других функций, то второе значение будет просто игнорироваться. Для того чтобы воспользоваться вторым значением, необходимо объединить получаемые значения в список. Добиться этого можно с помощью макроса MULTIPLE-VALUE-LIST:

```
(multiple-value-list (truncate 5 3)) → (1 2)
```

Для доступа к первому и второму значению следует использовать функции CAR и CADR.

Можно использовать макрос MULTIPLE-VALUE-SETQ, выполняющий множественные присвоения:

```
(multiple-value-setq (div rest) (truncate 5 3)) → 2
div → 1
rest → 2
```

Данный макрос обобщает действия, выполняемые специальной формой SETQ.

Локальное связывание множественных значений можно выполнить с помощью макроса MULTIPLE-VALUE-BIND, который обобщает LET-вызов:

```
(multiple-value-bind (div rest) (truncate 5 3) (+ div rest ))
→ 3
```

Макрос возвращает результат, вычисляемый последней формой его вызова.

Для работы с множественными значениями полезной может оказаться специальная форма MULTIPLE-VALUE-CALL, которая обобщает функционал FUNCALL:

```
(multiple-value-call  #'list  (truncate 5 3)) → (1 2)
```

Также в Коммон Лиспе имеется форма, которая формирует множественные значения из своих аргументов:

```
(values 4 8)
→4
8
```

Если в ходе обработки вызова

```
(values (read) (read))
```

ввести следующие s-выражения (+ 1 1) и (+ 2 2), которые читаются с помощью функции READ, то результатом вызова формы VALUES будет множественное значение: (+ 1 1), (+ 2 2).

5.25. Программирование задач ИИ на языке ЛИСП

5.25.1. Случайный поиск

Общий алгоритм случайного поиска решений в пространстве состояний описан в главе 2. Стратегия поиска заключается в случайном выборе оператора, обеспечивающего переход к следующему состоянию задачи. Поиск заканчивается, когда будет выполнен переход в целевое состояние.

Воспользуемся для представления графа состояний списком свойств символа. Представим граф состояний, изображенный на рисунке 5.13, в виде множества символов, соответствующих вершинам графа. С каждым символом свяжем свойство “соединение” (connection), значением которого будет являться список дочерних вершин:

```
(setf  (get  'a  'connection)  '(b e))
(setf  (get  'b  'connection)  '(a c f))
(setf  (get  'c  'connection)  '(b f))
(setf  (get  'd  'connection)  '(e g))
(setf  (get  'e  'connection)  '(a d f))
(setf  (get  'f  'connection)  '(b c e h))
(setf  (get  'g  'connection)  '(d h))
(setf  (get  'h  'connection)  ' (f g))
```

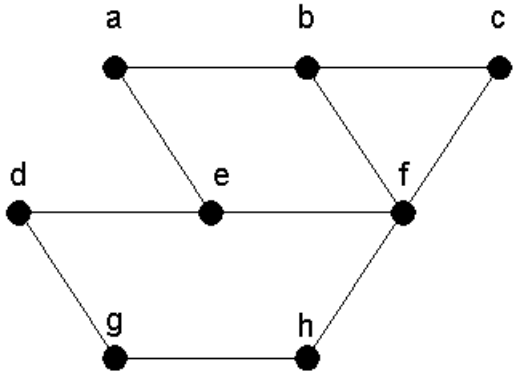


Рисунок 5.13 – Граф состояний

Случайный поиск можно организовать с помощью одного из двух алгоритмов – итерационного или рекурсивного. Определим для этого две функции random-search-i и random-search-r :

```
;итерационная версия
(defun random-search-i (start goal)
  (do ((current start (next current)))
      ((eq goal current)(print goal)(values))
      (print current)))

;рекурсивная версия
(defun random-search-r (start goal)
  (cond ((eq start goal)(list goal))
        (t (cons start (random-search-r (next start) goal)))))
```

В приведенных определениях выбор следующего состояния выполняется с помощью функции next:

```
(defun next (point)
  (nth (random (length (get point 'connection)))
       (get point 'connection)))
```

Здесь функция random генерирует случайное целое число, значение которого не превышает количества вершин, связанных с текущей вершиной (point). Это число определяет номер следующей вершины, извлекаемой из списка свойств текущей вершины. Таким образом, каждый раз следующая вершина выбирается случайно.

Вызов функции random-search-i обеспечивает простую печать пройденных вершин, а вызов функции random-search-r – списка, представляющего путь от стартовой вершины (start) до целевой вершины (goal):

```
(random-search-r 'a 'h) → ( a  e  a  b  c  b  f  h )
```

```
(random-search-i 'a 'h) → a e f b c b a e d g h
(random-search-r 'a 'h) → ( a e f h )
(random-search-i 'a 'h) → a e f b c f c f h
```

Из примера видно, что в ходе случайного поиска возможен многократный возврат к одним и тем же вершинам (состояниям). Кроме того, каждый новый вызов функции поиска возвращает различные пути от исходной до целевой вершины.

5.25.2. Поиск в глубину и ширину

Воспользуемся представлением графа состояний в виде a-списка, каждый подсписок которого будет представлять возможный переход на графе состояний. Так, граф состояний, изображенный на рисунке 5.13, можно задать в виде следующего a-списка:

```
(setq graph '((a b)(a e)(b a)(b f)(b c) (c b)(c f)(d e)(d g)(e a)(e d)(e f)
              (f b)(f c)(f e)(f h)(g d)(g h)(h g)(h f)))
```

Определим функцию depth-first в соответствии с алгоритмом поиска в глубину, описанном в главе 2. При этом списки OPEN и CLOSED будем также представлять в виде a-списков, в которых на первом месте любого подсписка располагается "дочерняя" вершина, а на втором – соответствующая "родительская" вершина. Это позволит после завершения поиска восстановить по списку CLOSED путь из начальной вершины в целевую вершину.

```
(defun depth-first(start goal)
  ;поместить подсписок (start start) в open
  (setq open (list (list start start)))
  ;инициализировать список closed
  (setq closed nil)
  (loop                    ; цикл
  (cond ((null open)(return 'неудача)) ; на графе нет пути
        (t
          ; запомнить первый подсписок списка open
           (setq n (first open))
          ; внести n в начало closed
           (setq closed (cons n closed))

           ; проверить, является ли  (first n) целевой вершиной
           (if (eq (first n) goal) (return 'удача))
           ; добавить в начало open дочерние вершины
```

```
(setq open (append (list-children (first n))(rest open)))
; печать промежуточных результатов
(terpri)
(princ ”closed=”)(prin1 closed)
(terpri)
(princ ”open=”) (prin1 open)))))
```

Здесь функция list-children возвращает для текущей вершины n a-список дочерних вершин, которые не входят в списки OPEN и CLOSED:

```
(defun list-children(n)
  (setq L nil)
  (dolist (temp graph L) ;выполнить действия для каждого
                         ;подсписка из graph
      ; инвертировать подсписок temp
       (setq rtemp (rev temp))
      ; выясняем, содержит ли temp дочернюю вершину
       (if (and (eq n (first temp))
                ; проверяем, входят ли дочерние  вершины
                ; в списки OPEN и CLOSED
            (not (assoc (first rtemp) open))
            (not (assoc (first rtemp) closed)))
        ; добавляем дочернюю вершину вместе с родительской
        ; к результирующему списку L
          (setq L (cons rtemp L)))))

; функция, инвертирующая подсписок temp
(defun rev(temp)
  (setq a (first temp))
  (setq b (second temp))
  (list b a))
```

Путь от стартовой до целевой вершины можно получить с помощью функции BACK-WAY, которая анализирует список CLOSED. В списке CLOSED содержатся “закрытые” вершины графа с указанием их родительских вершин. Используя ссылки на родительские вершины, функция BACK-WAY восстанавливает искомый путь:

```
(defun back-way (goal start)
  (setq g goal)
  (setq L nil)
  (dolist (temp closed L)
    (if (eq (first temp)g)
      (prog1 (setq L (cons (rev temp) L))
             (setq g (second temp))))))
```

Функция поиска в глубину depth-first может быть легко преобразована в функцию, реализующую метод поиска в ширину. Для этого необходимо поменять местами аргументы функции append:

```
( append ( rest open ) (list-children (first n )))
```

Это обеспечит добавление дочерних вершин в конец списка OPEN, что соответствует поиску в ширину.

5.25.3. Эвристический поиск на основе А-алгоритма

В А-алгоритме список OPEN упорядочен в соответствии со значением оценочной функции вида (2.1). Опишем граф состояний с помощью а-списка, в котором каждый подсписок имеет вид

```
( p  d  c  h ),
```

где p – родительская вершина; d – соответствующая дочерняя вершина; c – стоимость участка графа между вершинами p и d; h– оценка стоимости кратчайшего пути из вершины d целевую вершину. Тогда граф состояний, изображенный на рисунке 2.9, можно представить с помощью а-списка:

```
(setq graph '((s d 1 10)(s c 3 7)(s b 5 4)
              (s a 7 1)(a g 10 0) (b a 1 1)
              (c b 1 4)(c a 2 1)(d c 1 7)))
```

Определим на языке Лисп функцию A-SEARCH, соответствующую процедуре эвристического поиска с помощью А-алгоритма, описанного в главе 2. Общая схема построения функции A-SEARCH будет соответствовать приведенной выше функции DEPTH-FIRST. Отличия будут заключаться в том, что список открытых вершин OPEN упорядочивается в соответствии со значением оценочной функции, а элементы списков OPEN и CLOSED будут представляться в виде подсписков

```
( d  p  g  f ),
```

где d – некоторая вершина на графе состояний; p – ссылка на родительскую вершину; g – оценка стоимости пути из начальной вершины в вершину d; f – значение оценочной функции (2.1).

В соответствии с А-алгоритмом вершины из списка CLOSED могут перемещаться обратно в список OPEN. Это происходит в том случае, когда обнаруживается более короткий путь к вершине, которая уже была закры-

та. Тогда необходимо удалить соответствующую вершину из списка CLOSED и вставить ее в список OPEN с учетом значения оценочной функции. Кроме этого, если в процессе раскрытия очередной вершины будет установлено, что ее дочерние вершины уже включены в список OPEN, то выполняется сравнение оценочных функций вершин, уже имеющихся в списке OPEN, и вновь полученных дочерних вершин. Если оценочные функции вновь полученных дочерних вершин меньше, то они заменяют уже имеющиеся вершины в списке OPEN. Приведем определение функции A-SEARCH:

```
(defun a-search(start goal)
; поместить в open стартовую вершину, описываемую
; подсписком со структурой ( s s 0 h)
  (setq open (cons start nil))
; инициализация closed
  (setq closed nil)
  (loop            ; цикл
   (cond ((null open)(return ’неудача)) ; на графе нет пути
         (t
          ; запомнить первый подсписок списка open
          (setq n (first open))
          ; удалить n из open
           (setq open (remove n open :test #'equal))
          ; внести n в closed
           (setq closed (cons n closed))
          ; проверить, является ли (first n) целевой вершиной
           (if (eq (first n) goal) (return ’удача))
          ; внести в open дочерние вершины и удалить
          ; их при необходимости из closed
           (put-in-list (list-children n))
          ; печать промежуточных результатов
           (terpri)
           (princ ”closed=”)(prin1 closed)
           (terpri)
           (princ ”open=”) (prin1 open)))))
```

В этом определении функция LIST-CHILDREN возвращает а-список дочерних вершин для вершины (first n), которая при вызове LIST-CHILDREN является первым подсписком списка OPEN.

```
(defun list-children (n)
  (setq L nil)
  (dolist (temp graph L) ; выполнить действия для каждого
                         ; подсписка из graph
    ; переставить первый и второй элементы temp
     (setq rtemp (rev temp))
    ; выяснить, содержит ли temp дочернюю вершину для (first n)
```

```
        (when (eq (first n) (first temp))
            ; вычислить значения оценки g(n) для дочерней вершины
            (setq rtemp (put-third rtemp (+ (third n)(third rtemp))))
            ; вычислить значения оценки f(n) для дочерней вершины
            (setq rtemp (put-fourth rtemp (+ (third rtemp) (fourth rtemp))))
            ;добавляем описание дочерней вершины в виде подсписка rtemp
            ;к результирующему списку L
            (setq L (cons rtemp L)))))

; функция, переставляющая местами первый и второй элементы в temp
(defun rev(temp)
  (setq a (first temp))   (setq temp (rest temp))
  (setq b (first temp))   (setq temp (rest temp))
  (setq temp (cons b (cons a temp))))

; функция, меняющая значение третьего элемента
; в temp (оценка g(n)) на x
(defun put-third (str x)
  (setq temp1 (butlast str 2))
  (setq temp2 (last str))
  (append temp1 (cons x temp2)))

; функция, меняющая значение четвертого элемента
; в temp (оценка f(n)) на x
(defun put-fourth (str x)
  (setq temp1 (butlast str 1))
  (append temp1 (cons x nil)))
```

Список дочерних вершин, возвращаемый функцией LIST-CHILDREN, используется в функции PUT-IN-LIST для соответствующей модификации списков OPEN и CLOSED:

```
(defun put-in-list (dv) ;    dv – a-список дочерних вершин
  (cond
      ; условие завершения рекурсии
      ((null dv ) nil)
      ; если очередная дочерняя вершина не входит в
      ; списки OPEN и CLOSED, то добавить её  в список OPEN
      ((and (not (member1 (caar dv) open))
            (not (member1 (caar dv) closed)))
       (setf open (add (first dv) open))
       (put-in-list (rest dv)))
      ; если очередная дочерняя вершина  входит в
      ; один из списков OPEN или CLOSED, то удалить её из
      ; этих списков и заново внести в список OPEN с учетом значения
      ; оценочной функции
```

```
(t  (setf open (del (first dv) open))
    (setf closed (del (first dv) closed))
    (setf open (add (first dv) open))
    (put-in-list (rest dv)))))
```

Функция DEL(V L) выполняет удаление подсписка V, представляющего вершину графа состояний, из списка L. Удаление выполняется, если оценочная функция для удаляемой вершины V меньше либо равна значению оценочной функции вершины из списка L:

```
(defun del(v l)
  (cond ((null l) nil)  ; завершение рекурсии
        ; проверка вхождения v в l
        ((and (eq (first v)(first(first l)))
              ; сравнение оценочных функций
              (<= (fourth v)(fourth (first l))))
         (setf l (cdr l)))
        (t (append (list(car l)) (del v (cdr l))))))
```

Вставка дочерних вершин в упорядоченный список OPEN выполняется с помощью функции ADD(V L):

```
(defun add(v l)
  (cond ((null l) (cons v l))  ; выход из рекурсии
        ;сравнить значения оценочных функций
        ((<= (fourth v)(fourth (first l)))
        ; если меньше, то вставить вершину v в l
        (setf l (cons v l)))
        (t (append (list(car l)) (add v (cdr l))))))
```

Функция MEMBER1(V L) определяется следующим образом:

```
(defun member1 (v l)
  (cond ((null l) nil)
        ((eq v (first(first l))) t)
        (t (member1 v (rest l)))))
```

Если функция A-SEARCH завершает поиск сообщением “УДАЧА”, то восстановить путь из исходной вершины в целевую вершину можно с помощью функции BACK-WAY, описанной в предыдущем параграфе.

Рассмотрим выполнение эвристического поиска на примере графа состояний, который изображен на рисунке 2.9. Найдем путь из стартовой вершины S в целевую вершину G. Для этого выполним вызовы:

```
(a-search ’(s s 0 14) ’g)
(back-way ’g ’s))
```

В ходе выполнения функции A-SEARCH будут получены следующие промежуточные значения списков OPEN и CLOSED:

```
> (poisk)

closed=((s s 0 14))
open=((a s 7 8) (b s 5 9) (c s 3 10) (d s 1 11))
closed=((a s 7 8) (s s 0 14))
open=((b s 5 9) (c s 3 10) (d s 1 11) (g a 17 17))
closed=((b s 5 9) (s s 0 14))
open=((a b 6 7) (c s 3 10) (d s 1 11) (g a 17 17))
closed=((a b 6 7) (b s 5 9) (s s 0 14))
open=((c s 3 10) (d s 1 11) (g a 16 16))
closed=((c s 3 10) (s s 0 14))
open=((a c 5 6) (b c 4 8) (d s 1 11) (g a 16 16))
closed=((a c 5 6) (c s 3 10) (s s 0 14))
open=((b c 4 8) (d s 1 11) (g a 15 15))
closed=((b c 4 8) (c s 3 10) (s s 0 14))
open=((a b 5 6) (d s 1 11) (g a 15 15))
closed=((a b 5 6) (b c 4 8) (c s 3 10) (s s 0 14))
open=((d s 1 11) (g a 15 15))
closed=((d s 1 11) (a b 5 6) (b c 4 8) (s s 0 14))
open=((c d 2 9) (g a 15 15))
closed=((c d 2 9) (d s 1 11) (s s 0 14))
open=((a c 4 5) (b c 3 7) (g a 15 15))
closed=((a c 4 5) (c d 2 9) (d s 1 11) (s s 0 14))
open=((b c 3 7) (g a 14 14))
closed=((b c 3 7) (c d 2 9) (d s 1 11) (s s 0 14))
open=((a b 4 5) (g a 14 14))
closed=((a b 4 5) (b c 3 7) (c d 2 9) (d s 1 11) (s s 0 14))
open=((g a 14 14))
```

Функция BACK-WAY возвратит искомый путь в виде:

```
((s s 0 14) (s d 1 11) (d c 2 9) (c b 3 7) (b a 4 5) (a g 14 14))
```

5.25.4. Сопоставление с образцом

Сопоставление с образцом является важнейшей операцией, выполняемой во многих программах искусственного интеллекта. В ходе сопоставления с образцом осуществляется сравнение поступающих на вход списков (или других структур данных) с известными символьными структурами (образцами). Благодаря этому выполняется распознавание входных данных.

Сопоставление с образцом – неотъемлемая часть СИИ, использующих продукционную модель представления знаний. В этом случае предпосылки продукционных правил выступают в качестве образцов, а факты, хранящиеся в рабочей памяти, выступают в роли входных данных. Если предпосылка правила сопоставима с данными, имеющимися в рабочей памяти, то правило активизируется. Сопоставление с образцом широко применяется в СИИ, основанных на логических моделях представления знаний. В частности, унификация предикатов представляет собой сопоставление, в ходе которого выявляют условия совпадения предикатов.

Рассмотрим сопоставление с образцом, представляя образец и сопоставляемые с ним факты (данные) в виде списков языка Лисп. Простейшие возможности сопоставления реализуются встроенными предикатами Лиспа, например EQ, EQL, EQUAL. Если необходимо осуществлять сопоставление произвольных s-выражений, то следует определить отдельную функцию. Для начала определим функцию, выполняющую сопоставление одноуровневых списков.

К сопоставлению образца и данных можно предъявлять различные требования. Пусть символ ‘?’ в образце отождествляется с любым элементом из списка данных, а символ ‘$’ – с произвольным числом (возможно нулевым) элементов списка данных. Символы ‘?’ и ‘$’ рассматривают как универсальные символы подстановки, которым может быть сопоставлен любой элемент или сегмент списка, соответственно. Обозначим сопоставимость двух выражений знаком :=:, и сформулируем простые правила сопоставления [39]:

```
ОБРАЗЕЦ :=: ДАННЫЕ
если            ; правило 1
        ДАННЫЕ = NIL и ОБРАЗЕЦ = NIL
или             ; правило 2
        ДАННЫЕ = NIL   и (CAR ОБРАЗЕЦ) = $
                       и (CDR ОБРАЗЕЦ) = NIL
или             ; правило 3
        (ДАННЫЕ ≠ NIL) и (ОБРАЗЕЦ) ≠ NIL
                       и (CAR ОБРАЗЕЦ) = (CAR ДАННЫЕ)
                       и (CDR ОБРАЗЕЦ) :=: (CDR TEST)
или             ; правило 4
        (ДАННЫЕ ≠ NIL) и (ОБРАЗЕЦ ≠ NIL)
                       и (CAR ОБРАЗЕЦ) = ?
                      и (CDR ОБРАЗЕЦ) :=: (CDR ДАННЫЕ)
или             ; правило 5
        (ДАННЫЕ ≠ NIL) и (ОБРАЗЕЦ ≠ NIL)
                          и (CAR ОБРАЗЕЦ) = $
                          и (CDR ОБРАЗЕЦ) :=: ДАННЫЕ
```

```
или                ; правило 6
        (ДАННЫЕ ≠ NIL) и (ОБРАЗЕЦ ≠ NIL)
                              и (CAR ОБРАЗЕЦ) = $
                              и  ОБРАЗЕЦ :=: (CDR ДАННЫЕ)
```

Введенные правила рекурсивно определяют понятие “сопоставимость”. Первое правило утверждает сопоставимость пустых списков и обеспечивает выход из рекурсии. Правило 2 позволяет сопоставить списки ($) и ( ), ($ $) и ( ), … , т.е. в этом случае символ ‘$’ не отождествляется ни с одним из элементов списка данных. Третье правило утверждает, что образец сопоставим с данными, если первый элемент образца равен первому элементу данных, и хвостовые части образца и данных сопоставимы. Например,

```
(a      b      c      d) :=: (a      b      c      d).
```

Правило 4 обеспечивает сопоставление образца с данными, если первый элемент образца представлен символом ‘?’, и хвостовые части образца и данных сопоставимы:

```
(?      b      c      d) :=: (a      b      c      d).
```

Правила 5 и 6 позволяют учесть особенности сопоставления символа ‘$’. В соответствии с правилом 5 символ ‘$’ может не отождествляться ни с одним из элементов списка данных. Например,

```
($      b      c      d) :=: (a      b      c).
```

Правило 6 сопоставляет с символом ‘$’ сегмент списка данных. Например,

```
($      d) :=: (a      b      c      d).
```

В этом случае символ ‘$’ отождествляется с последовательностью элементов a, b, c.

Определим функцию (MATCH P D), выполняющую сопоставление образца P и данных D с помощью введенных выше правил:

```
; функция сопоставления  образца p и данных d
(defun match (p d)
        (cond
                ;правило 1
                 ((and (null p)(null d)) t)
                 ;правило 2
                 ((and (null d)
                        (eq (car p) '$)
                        (null (cdr p))) t)
```

```
; один из списков исчерпан
((or (null p)(null d)) nil)

;правило 3 и правило 4
((or (equal (car p) '?)
     (equal (car p) (car d)))
 (match (cdr p)(cdr d)))

;правило 5 и 6
((eq (car p) '$)
 (cond ((match (cdr p) d) t)
       ((match p (cdr d)) t))) ))
```

Здесь предварительное условие для правил 3 – 6 – (ДАННЫЕ ≠ NIL) и (ОБРАЗЕЦ ≠ NIL) – записано перед правилом 3 в виде:

```
((or (null p)(null d)) nil)
```

Если один из списков p или d исчерпан, то это исключает применение последующих правил, и функция MATCH возвращает NIL.

Функцию MATCH можно легко расширить, чтобы она могла сопоставлять списки с вложенными подсписками. Для этого достаточно добавить в текст функции следующее правило:

```
;правило 7 - сопоставление списков,включающих подсписки
((and (not (atom (car p)))
      (not (atom (car d)))
      (match (car p)(car d)))
 (match (cdr p) (cdr d)) )
```

В этом случае функция MATCH будет обеспечивать сопоставление более сложных символьных структур. Например,

```
(match '( (a    b) ? (?   d)  $)
       '( (a    b) c (c   d)  1  2  3  4  5))
→T
```

Несмотря на то, что введенная функция MATCH реализует простейшую схему сопоставления, её уже можно использовать для решения практических задач, например, доступа к базе данных. Определим для этого функцию GET-MATCHES с двумя аргументами, представленными s-выражениями. Первый аргумент будет представлять собой запрос к базе данных, записываемый в виде образца, а второй – список, соответствующий базе данных, в которой выполняется поиск. Функция GET-MATCHES

возвращает в качестве результата записи базы данных, сопоставимые с запросом (образцом).

Пусть рассматривается база данных, содержащая сведения о служащих:

```
(setq *database*
      '(((Иванов Сергей)    14000 1235)
        ((Абрамов Николай)  17500 3325)
        ((Гавриленко Ольга)  14000 3143)
        ((Денисенко Сергей) 22000 2528)
        ((Коль Татьяна) 18250 3133)
         … ))
```

В базе данных хранятся данные о годовых доходах служащих и их табельные номера. Определим функцию GET-MATCHES, используя рекурсию:

```
(defun get-matches (p db)
  ;db - база данных
  ;p  - запрос (образец)
  (cond ((null db) nil)
        ((match p (car db))
         (cons (car db) (get-matches p (cdr db))))
        (t (get-matches p (cdr db)))))
```

С помощью функции GET-MATCHES, можно получать ответы на следующие запросы:

```
;проверка наличия конкретной записи
(get-matches '((Денисенко Сергей) 22000 2528) *database*)
→ (((Денисенко Сергей) 22000 2528))

;выдать сведения о всех сотрудниках с годовым доходом 14000
(get-matches '(? 14000 ?) *database*)
→ ((( Иванов Сергей) 14000 1235) ((Гавриленко Ольга) 14000 3143))

;выдать сведения о всех сотрудниках, которых зовут Сергей
(get-matches '((? Сергей) ? ?) *database*)
→ (((Иванов Сергей) 14000 1235) ((Денисенко Сергей) 22000 2528))
```

Приведенный выше вариант реализации функции MATCH, хотя и позволяет сопоставлять сложные символьные структуры, имеет ряд недостатков. Во-первых, подстановки в символы ‘?’ и ‘$’ не запоминаются. Во-вторых, отсутствует возможность указать, что на определенных позициях списков могут находиться любые, но одинаковые выражения.

С целью устранения указанных недостатков введем в образец переменные. Имена переменных в образце будут начинаться с символов ‘?’ или

‘$’. Например, ?V или $V. В первом случае в переменную V подставляется произвольный символ данных, во втором – сегмент списка данных.

Расширим возможности функции MATCH так, чтобы она могла запоминать подстановки в переменных. Например:

```
(match ’(AVTO ?MODEL ?YEAR) '(AVTO ford 1998))⟶ T
model ⟶ ford
year ⟶ 1998
```

Сопоставления с конструкциями ?V или $V могут быть реализованы с помощью двух новых функций – CAR-LETTER и CDR-NAME. Функция CAR-LETTER обеспечивает выделение первой литеры из имени очередного атома образца, а функция CDR-NAME формирует имя переменной из оставшихся литер:

```
(defun car-letter (x)  (car (coerce (string x) ’list)))

(defun cdr-name (x)
        (intern  (coerce (cdr (coerce (string x) ’list)) ’string)))
```

Здесь x – атом образца, который с помощью вызова (string x) преобразуются в строку. Затем из строки выделяются первая литера (функция car-letter) и все оставшиеся литеры (функция cdr-name), которые образуют имя переменной. Полученное имя переменной с помощью функции INTERN включается в список имён символов. Функция INTERN проверяет, есть ли символ с данным именем в списке имен символов. Если такой символ имеется, то функция возвращает ссылку на имя этого символа, если нет, то добавляет новый символ в список имен и возвращает ссылку на него.

Добавим в функцию MATCH правило, обеспечивающее сопоставление очередного элемента данных с параметром вида ?V:

```
;правило 8
        ((and (atom (car p))
              (eq (car-letter (car p)) #\?)
              (match (cdr p)(cdr d)))
          (set  (cdr-name (car p)) (car d) t)
```

В соответствии с правилом сначала убеждаемся, что очередной элемент образца представляет собой атом, начинающийся с литеры #\?, а также проверяем сопоставимость хвостовых частей образца и данных. Если указанные проверки выполняются успешно, то переменной присваивается значение очередного элемента списка данных.

Для обработки параметра образца вида $V введем в функцию MATCH дополнительное правило:

```
;правило 9
        ((and (atom (car p))
              (eq (car-letter (car p)) #\$))
         (cond ((match (cdr p)(cdr d))
                (set (cdr-name (car p)) (list (car d)))
                 t)
                ((match p (cdr d))
                 (set (cdr-name (car p))
                      (cons (car d)(eval (cdr-name (car p)))))
                  t)))
```

Аналогично предыдущему правилу вначале убеждаемся, что очередной элемент образца – это атом, начинающийся с символа #\$. Затем, после проверки сопоставимости хвостовых частей списков P и D или списка P и хвостовой части списка D, выполняется функция SET. Данная функция присваивает переменной образца значение, соответствующее сегменту списка данных. Рассмотрим этот процесс на примере:

```
(match   '(?F $V) '(Petrov 1975   5   October))
```

При попытке сопоставления параметра ?F с первым элементом списка данных функция MATCH в соответствии с правилом 8 осуществит рекурсивный вызов

```
(match   '($V)  '(1975     5     October))
```

Данный вызов будет обрабатываться с помощью правила 9. Так как (CDR P) и (CDR D) не сопоставимы, то будет выполнен рекурсивный вызов (MATCH '($V) '(5 October)). Данный процесс повторится еще один раз. После вызова (MATCH '($V) '(October)) функция MATCH устанавливает, что хвостовые части списков P и D сопоставимы (т.е. NIL:=:NIL). Это приводит к выполнению подстановки V←(October) с помощью вызова:

```
(set (cdr-name (car p)) (list (card)))
```

Затем управление передается в точку предыдущего вызова функции MATCH, т.е. во вторую ветвь условного выражения COND правила 9. Это обеспечивает добавление в начало списка, доступ к которому осуществляется через переменную V, сначала числа 5, а затем числа 1975. Переменная V получает значение списка (1975 5 October). Указанное добавление элементов в список V реализует с помощью следующего вызова функции SET:

```
(set
      (cdr-name (car p))          ; выделить имя переменной
      (cons                       ; добавить
            (car d)               ; первый элемент из d
            (eval (cdr-name (car p))))); к значению переменной
```

Значение переменной можно использовать в процессе сопоставления. Например, можно потребовать, чтобы все встречающиеся в образце переменные с одинаковым именем сопоставлялись с одним и тем же значением. При этом возможны два случая. Если сопоставляемая переменная встретилась первый раз, то результат сопоставления просто запоминается. Если переменная встречалась ранее, то следует соответствующий параметр в образце заменить его значением и повторить процесс сопоставления. Обычно значения параметров хранят с помощью ассоциативного списка. Например, результаты предыдущего сопоставления параметров ?F и $V можно хранить в виде а-списка – ((F . Petrov) (V . (1975 5 October))).

Формировать такой список можно с помощью функции ACONS. Для реализации возможности сопоставления повторяющейся переменной образца с одним и тем же значением, необходимо изменить соответствующим образом правила 8 и 9 функции MATCH. Присвоение значений переменным должно выполняться только в том случае, если они отсутствуют в а-списке. Организовать соответствующий поиск переменных в а-списке можно с помощью функции ASSOC. Предоставим возможность расширения правил 8 и 9 читателю.

Функцию MATCH можно совершенствовать дальше, используя ограничивающий параметр в образце, который дает возможность накладывать определенные ограничения на сопоставляемое значение. Обычно такие ограничения задаются с помощью предикатов. Будем представлять ограничения в виде подсписков образца [95]:

(restrict < *?* | *?переменная* > {< *предикат* >}$^{+}$)

Здесь *предикат* – функция с одним аргументом, который соответствует текущему сопоставляемому элементу списка данных. Если предикат (предикаты) принимает для текущего элемента данных истинное значение, то считается, что сопоставление успешно, и переменная получает соответствующее значение. Например,

```
(match '((restrict  ?V  integerp  evenp) b c)
                '(36  b  c)) → T
 (match '((restrict ?V integerp evenp) b c)
                '(36.0  b  c)) → NIL
```

В примере параметр ?V из подсписка ограничений сопоставляется с первым элементом списка данных. Если рассматриваемый элемент данных является целым четным числом, то сопоставление успешно.

Для обработки подсписков ограничений образца введем в функцию MATCH два новых правила. Одно из правил будет применяться, когда в

подсписке ограничений используется параметр вида ?, а другое – параметр вида ?V:

```
; правило 10
      ((and (not(atom (car p)))
            (eq (caar p)  'restrict)
            (eq (cadar p) '?)
            (and-to-list
            (mapcar  #'(lambda (pred)
                        (funcall pred (car d))) (cddar p))))
       (match (cdr p)(cdr d)))

; правило 11
       ((and (not (atom (car p)))
             (not (atom  d))
             (eq (caar p)  'restrict)
             (eq (car-letter (cadar p)) #\?)
             (and-to-list
             (mapcar  #'(lambda (pred)
                         (funcall pred (car d))) (cddar p)))
              (match (cdr p)(cdr d)))
       (set (cdr-name (cadar p)) (car d))
       t)
```

В соответствии с правилом 10, после проверки необходимых тестов, позволяющих установить, что очередной элемент образца – подсписок ограничений, к сопоставляемому элементу данных посредством отображающего функционала MAPCAR применяются последовательно предикаты, заданные в подсписке ограничений. Результаты применения предикатов возвращаются в виде списка, элементами которого являются логические значения. Функция AND-TO-LIST проверяет, все ли элементы списка имеют значение T:

```
(defun and-to-list ( l )
  ;l - список логических значений
  (setq res t)
  (dolist (temp l res)
        (setq res (and res temp))))
```

Если отмеченное условие выполняется, то это означает, что параметр ? из подсписка ограничений успешно сопоставлен, и можно перейти к сопоставлению хвостовых частей списков P и D.

Правило 11 выполняется сходным образом. Отличие заключается только в присвоении переменной значения с помощью вызова

```
(set (cdr-name (cadar p)) (car d))
```

Используя рассмотренную функцию сопоставления с образцом, можно написать простейшую программу, поддерживающую беседу на естественном языке. Такая программа основана на распознавании ответа пользователя с помощью функции MATCH и формировании подходящего по смыслу вопроса.

Определим функцию, обеспечивающую общение на естественном языке, и назовем ее ELIZA в честь известной программы, разработанной Дж. Вейценбаумом. В приведенном варианте функция ELIZA обеспечивает ведение краткого диалога, касающегося погоды и отдыха. Оригинальная программа ELIZA имитировала общение с психотерапевтом [11].

В функции будут использоваться переменные, которые инициализируются в начале ее выполнения. Данные переменные позволят при формировании вопросов, задаваемых программой, учесть предыдущие ответы пользователя.

```
(defun eliza()
       (setq name nil)
       (setq otpusk nil)

(loop
       (terpri)
       (princ "Говорите >")
       (setq fact (read))

     ;правила
      (cond ((equal fact '( )) (return “До свидания”))

      ((match '($ Добрый день $) fact)
        (princ “Добрый день. Как вас зовут ?”)
        (setq name t))

      (name (match '(?n  $) fact)
              (setq name nil)
              (princ n)(princ “, О чём вы хотели поговорить ?”))


      ((match '($ о чём $ говорить $) fact)
       (princ “Конечно о погоде”)
       (terpri)
       (princ “Какую погоду вы любите?”))

      ((match '($ снег $)  fact)
       (princ “В такую погоду хорошо кататься на лыжах”)
       (terpri)
       (princ “На чём вы любите кататься ?”))

      ((match '(На ?thing) fact)
```

```
(princ “Кататься на “)(princ thing)(princ “ – тоже не плохо”)
(terpri)
(princ “У вас будет отпуск зимой или летом?”))


((match ’($ летом $) fact)
 (princ “А где вы отдыхали прошедшим летом?”))

((match ‘($ отдыхал $ (restrict ?place place_p) ) fact)
  (princ “Куда вы планируете поехать в следующий отпуск?”)
  (setq otpusk t))

 ((match ’($ na ?p $) fact)
       (princ p)(princ “Прекрасное место”)
       (terpri)
       (princ “Вы бывали там раньше?”))

  (otpusk (princ “Вы говорили о”)
        (princ place) (terpri)
        (princ “Рассказать вам мою историю о”)
        (princ place)(princ “?”)
        (setq otpusk nil))

 (t  (princ “К сожалению, нам пора прощаться.
             Продолжим нашу беседу в следующий раз.
             Напечатайте ( ).”)))))

; предикат, проверяющий наличие сведений о месте отдыха
(defun place_p(x)
   (cond ((eq x ’Крыму) t)
         ((eq x ’Кавказе) t)
         ((eq x ’Киеве) t)))
```

Приведенная программа является тривиальной, однако, даже в таком простейшем варианте она способна на краткий диалог:

```
(eliza)
Говорите> (Добрый день)
ДОБРЫЙ ДЕНЬ. КАК ВАС ЗОВУТ?
Говорите> (Иван)
Иван, о чём вы хотели поговорить?
Говорите> (о чём можно говорить?)
КОНЕЧНО О ПОГОДЕ. КАКУЮ ПОГОДУ ВЫ ЛЮБИТЕ?
Говорите> (Мне нравится, когда идет снег)
В ТАКУЮ ПОГОДУ ХОРОШО КАТАТЬСЯ НА ЛЫЖАХ.
НА ЧЁМ ВЫ ЛЮБИТЕ КАТАТЬСЯ?
Говорите> (На санках)
КАТАТЬСЯ НА САНКАХ ТОЖЕ НЕПЛОХО.
У ВАС БУДЕТ ОТПУСК ЗИМОЙ ИЛИ ЛЕТОМ?
```

Говорите> (Летом)
А ГДЕ ВЫ ОТДЫХАЛИ ПРОШЕДШИМ ЛЕТОМ?
Говорите> (Я отдыхал на Кавказе)
КУДА ВЫ ПЛАНИРУЕТЕ ПОЕХАТЬ В СЛЕДУЮЩИЙ ОТПУСК?
Говорите> (На Сахалин)
САХАЛИН ПРЕКРАСНОЕ МЕСТО.
ВЫ БЫВАЛИ ТАМ РАНЬШЕ?
Говорите> (Несколько раз)
РАССКАЗАТЬ ВАМ МОЮ ИСТОРИЮ О КАВКАЗЕ
Говорите> (Да, пожалуйста)
К СОЖАЛЕНИЮ, НАМ ПОРА ПРОЩАТЬСЯ. ПРОДОЛЖИМ НАШУ БЕСЕДУ В СЛЕДУЮЩИЙ РАЗ. НАПЕЧАТАЙТЕ ().

5.25.5. Поиск в семантических сетях

Рассмотрим семантическую сеть, изображенную на рисунке 5.14. Представим объекты и отношения семантической сети с помощью списка свойств. При этом объекты семантической сети будут представляться соответствующими символами языка ЛИСП, а отношения – свойствами символа:

```
(setf (get 'животное 'покров) 'кожа)
(setf (get 'птица 'покров) 'перья)
(setf (get 'птица 'передвижение) 'летать)
(setf (get 'рыба 'передвижение) 'плавает)
(setf (get 'страус 'передвижение) 'ходит)
(setf (get 'пингвин 'передвижение) 'ходит)
(setf (get 'пингвин 'цвет) 'черно-белый)
(setf (get 'канарейка 'цвет) 'желтый)
(setf (get 'канарейка 'звук) 'пение)
(setf (get 'малиновка 'цвет) 'красный)
(setf (get 'малиновка 'звук) 'пение)
```

Используя введенное представление семантической сети, определим функции, обеспечивающие поиск значений свойств в соответствии с иерархией объектов:

```
(setf (get 'малиновка 'ako) 'птица)
(setf (get 'канарейка 'ako) 'птица)
(setf (get 'пингвин 'ako) 'птица)
(setf (get 'страус 'ako) 'птица)
(setf (get 'птица 'ako) 'животное)
(setf (get 'рыба 'ako) 'животное)
(setf (get 'чилли-вилли 'is-a) 'пингвин)
(setf (get 'матильда 'is-a) 'канарейка)
```

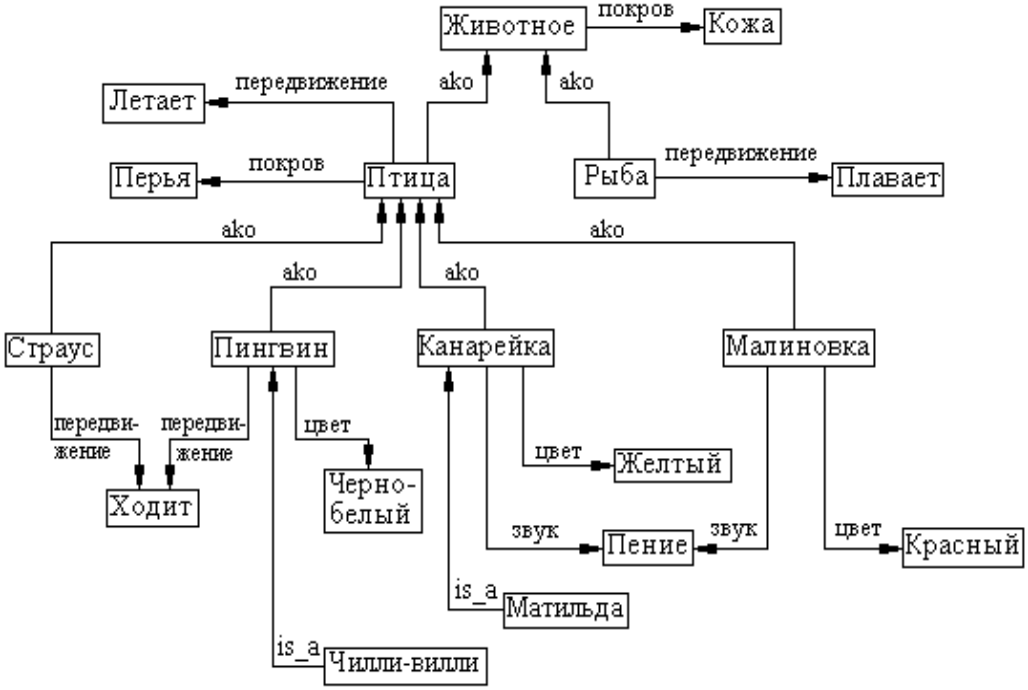


Рисунок 5.14 – Семантическая сеть части животного мира

Рассмотрим простейший вариант поиска в глубину. Определим функцию findproperty(p x n), возвращающую значения свойства p для заданного объекта x. Третий параметр функции – ограничивает глубину поиска:

```
(detun findproperty(p x n)
;  p  - свойство
;  x  - объект
;  n  - глубина поиска
(cond ((and    (get x 'is-a) (eq  p 'ako))
          (format t "~s является экземпляром класса" x)
          (values))
         ((zerop n) nil) ;  достигнута максимальная глубина
         ((get x p)) ;  свойство найдено
         (t (or   (findproperty p (get x 'ako) (- n 1))
                  (findproperty p (get x 'is-a) (- n 1))
```

Здесь сначала выясняется, не является ли объект x экземпляром некоторого класса. С этой целью проверяется наличие свойства is-a. Если данное свойство обнаружено, то проверяется, не запрашивается ли при этом значение свойства ako. В случае обнаружения указанного совпадения функция возвращает сообщение: "*x* является экземпляром класса". Такое сообщение формируется, например, при вызове

(findproperty 'ako 'чилли-вилли' 5)

Второй тест, выполняемый в функции findproperty, обеспечивает проверку граничной глубины поиска.

Извлечение значения необходимого свойства объекта x выполняется с помощью теста (get x p). Если данный вызов возвращает значение nil, то выполняется рекурсивный поиск свойства p с учетом иерархии ako или is-a связей. Для этого функция findproperty вызывается рекурсивно, и предпринимается попытка найти заданное свойство у объекта, который возвращается в результате вызова (get x 'ako) или (get x 'is-a). Эти вызовы возвращают имя соответствующего суперкласса или класса для рассматриваемого объекта. Таким образом, поиск выполняется по иерархии объектов в соответствии со структурой семантической сети.

Вопросы для самопроверки

1. Определите следующие основные понятия языка Лисп: атом, s-выражение, точечная пара, список.
2. Нарисуйте диаграмму точечной пары ((a . b) .((c . d) . (e . f)))
3. Что в языке Лисп называют формой, и какими конструкциями языка она представляется?
4. Сформулируйте правила интерпретации, используемые в Лиспе.
5. Приведите примеры вызовов основных арифметических функций языка Лисп.
6. Объясните на примерах назначение функций CAR, CDR, CONS.
7. Приведите примеры вызовов встроенных функций обработки списков.
8. Объясните на примерах особенности применения форм: SET,SETQ,SETF.
9. Какие функции в языке Лисп выполняют сравнение данных?
10. Перечислите предикаты языка Лисп, которые позволяют проверять типы значений.
11. Что называется лямбда-выражением и лямбда-вызовом?
12. Определите на языке Лисп функцию, возвращающую список, составленный из двух ее аргументов.
13. Объясните на примерах использование следующих параметров функций: обязательных, необязательных, остаточных, ключевых.
14. Объясните понятия лексической и специальной переменных языка Лисп.
15. Приведите пример вызова формы COND.
16. Какие средства можно использовать для организации итерационных циклических вычислений?
17. Определите функцию MEMBER, используя рекурсию.
18. Приведите примеры вызовов функций RPLACA, RPLACD, NCONC.
19. Что называют ассоциативным списком? Какие функции имеются в языке Лисп для работы с ассоциативными списками.
20. Что представляет собой символ языка Лисп?
21. Как выполнить ввод-вывод данных из файла? Приведите примеры.
22. Что в языке Лисп называют функционалом?

23. Объясните действия, выполняемые функционалами FUNCALL, APPLY, MAPCAR, MAPLIST.
24. Что называют обратной блокировкой?
25. Что называют замыканием?
26. Определите на языке Лисп структуру. Как создать экземпляр структуры? Как получить доступ к слотам структуры?
27. Приведите пример определения простого класса на языке Лисп и создайте экземпляр класса.
28. Объясните понятие “опции слота”, используемое при вызове макроформы DEFCLASS.
29. Как определить метод класса?
30. Что в языке Лисп понимают под общей (родовой) функцией?
31. Объясните схему наследования свойств классов в языке Лисп?
32. Объясните отличие макроса от функции в языке Лисп.
33. Напишите на языке Лисп функцию, реализующую поиск в глубину в пространстве состояний.
34. Сформулируйте правила сопоставления с образцом, когда образец и данные представляются в виде линейных списков.
35. Напишите на языке Лисп функцию сопоставления с образцом для случая, когда образец и данные представляются в виде линейных списков.
36. Опишите на языке Лисп семантическую сеть, изображенную на рисунке 3.7.

ГЛАВА 6
ЯЗЫК ПРОЛОГ

Язык Пролог был разработан в начале семидесятых годов в университете Марселя (Франция) под руководством А. Колмероэ. Само название языка определяет его суть – язык логического программирования.

Пролог – язык, предназначенный для программирования приложений, использующих принципы искусственного интеллекта. Диапазон его применения достаточно широк: реляционные базы данных; экспертные системы; понимание естественного языка; математическая логика; перевод с одного языка на другой; символьные вычисления и др.

В главе рассматриваются основные понятия и типовые приёмы программирования на языке Пролог, описываются встроенные предикаты пролог-систем. Возможности языка Пролог иллюстрируются на примерах классических задач ИИ, таких, как поиск в глубину и ширину, эвристический поиск, поиск в пространстве версий.

6.1. Общая характеристика языка

Пролог – язык программирования, существенно отличающийся от традиционных языков, хотя некоторые его возможности свойственны и другим языкам программирования. Так, обработка списков и рекурсия на Прологе имеют много общего с соответствующими средствами языка Лисп. Однако Пролог характеризуется более высоким уровнем абстракции: если Лисп позволяет программисту не заботиться о динамическом выделении памяти под списки, то Пролог, кроме этого, обеспечивает встроенные механизмы управления выполнением программы. Поэтому программа на Прологе в большей степени представляет собой декларативное описание отношений некоторой предметной области. Выполнение программы заключается в постановке вопросов, относящихся к предметной области, и автоматическом поиске ответов на вопросы с помощью встроенных механизмов логического вывода.

Пролог берет свое начало от логики предикатов. В основу Пролога положены хорновские предложения исчисления предикатов (см. § 4.1.8). Программист определяет необходимые факты и правила, а Пролог использует дедуктивный вывод, реализуемый на основе принципа резолюции, для выдачи ответов на вопросы пользователя. Такой подход к решению задач представляет собой полную противоположность программированию на традиционных языках. В ходе программирования на Прологе программист не указывает “как решается задача”, а указывает “что решается”. Поэтому написание программы на Прологе сводится к составлению декларативных спецификаций предметной области, выраженных дизъюнктами Хорна.

Существуют различные версии языка Пролог: Quintec-Prolog, Micro-Prolog, Arity-Prolog, Prolog2, Expert-Prolog, Turbo-Prolog, MProlog и др. Первая эффективная реализация Пролога была создана в Эдинбургском университете в 1977 для ЭВМ DEC-10. Данная версия языка является фактическим стандартом. В дальнейшем изложении будем придерживаться синтаксиса этой версии языка Пролог.

Пролог содержит множество встроенных предикатов, обеспечивающих выполнение операций с файлами, построения графических изображений, воспроизведения звуков, управления цветом и т.п. Ограничимся рассмотрением минимального числа встроенных предикатов, необходимых для работы с базами знаний и выполнения операций ввода-вывода.

6.2. Основные понятия языка

Программа на Прологе есть совокупность утверждений, являющихся дизъюнктами Хорна. В конце утверждения ставится точка. Существуют два типа утверждений: факты и правила. *Факт* представляет собой истинное утверждение. Факт на Прологе представляется в виде предиката. Например, утверждение “Игорь – отец Святослава” записывается на Прологе в следующей форме:

отец(´Игорь´,´Святослав´).

Здесь **отец** – это имя предиката, а аргументы **'Игорь'** и **'Святослав'** представляют собой символьные константы. Символьные константы должны начинаться со строчной буквы или заключаться в одинарные кавычки. Имена предикатов могут начинаться с любой буквы. Однако следует помнить, что предикаты **Отец( )** и **отец( )** будут разными предикатами во многих реализациях Пролога.

В Прологе используется соглашение, позволяющее отличать имена конкретных объектов (констант) от имен переменных. Имя, начинающееся

с прописной буквы или символа подчеркивания, рассматривается как *переменная*. Так, **X,Y, T34, X+Y, _a, _132** – это имена переменных.

Правило – представляет собой утверждение, которое истинно при некоторых условиях. Например, утверждение "X – дед Y" истинно при условии, что X – это отец Z, а Z – это родитель Y. На Прологе это правило запишется в виде

дед(X,Y): – отец (X,Z), родитель(Z,Y).

Правило должно заканчиваться точкой. Правило состоит из заголовка (головы) и тела. Заголовок и тело соединяются знаком "**: –** ", который соответствует импликации "←". Запятая, записанная в теле правила, соответствует логической связке "и". Приведенное выше правило интерпретируется следующим образом: "если верно **отец(X,Z)** и **родитель(Z,Y)**, то верно **дед(X,Y)**". Заголовок правила представляет факт, для определения которого предназначено правило. В рассматриваемом случае – это **дед(X,Y).** Данный факт будет иметь место, если будут верными утверждения, образующие тело правила, которые называют *условиями*.

В теле правила можно использовать логические связки "или", которые обозначаются точкой с запятой. Например,

родитель(X,Y): – отец(X,Y); мать(X,Y).

Иными словами, **X** является родителем **Y**, если **X** – это отец или мать лица **Y**. В приведенных правилах использовались переменные **X**, **Y**, **Z**, каждое вхождение которых в правило соответствует одному и тому же объекту. Однако при повторном использовании правила этим переменным могут быть сопоставлены уже другие объекты.

Факты и правила образуют базу данных. Рассмотрим пример:

```
отец('Иван','Сергей' ).          /*Факт:  Иван – отец  Сергея */
отец( 'Иван','Анна' ).           /* Факт:  Иван – отец   Анны  */
мать('Мария','Сергей').          /*Факт: Мария – мать Сергея */

/*Правило: O и M являются родителями ребенка P,
 если O является отцом P и M является мамой P */
 родители(O,M,P ): – отец(O,P), мать(M,P).
```

Когда база данных создана, можно обращаться к ней с вопросами. Например: "Является ли Иван отцом Сергея?" Для этого необходимо ввести следующее целевое утверждение:

? – отец('Иван','Сергей').

Обращение с вопросом к пролог-системе инициирует процедуру поиска в базе данных. Пролог ищет факты, сопоставимые с фактом целевого утверждения. При этом выполняется процедура унификации (см. § 4.1.4). Два факта сопоставимы, если совпадают имена предикатов, и соответствующие аргументы попарно сопоставимы. В рассматриваемом случае Пролог находит в базе данных факт **отец('Иван','Сергей'),** сопоставимый с целевым утверждением. Обнаружив этот факт, Пролог система ответит **Yes** (да).

К рассмотренной базе данных можно обращаться и с другими вопросами. Например, ответом на вопрос

? – отец('Сергей','Иван').

будет **No** (нет), так как в базе данных нет факта, сопоставимого с вопросом. На вопрос

? – отец('Иван',P). / * Кто дети Ивана? */

пролог-система выдаст ответ **P='Сергей'**. Если ввести с терминала знак "**;**", то пролог-система продолжит поиск, и получит второе значение **P='Анна'**. Если еще раз ввести точку с запятой, то пролог-система вернет значение **No**. Можно задать и более сложный вопрос: "Кто родители Сергея?":

? – родители(O,M,'Сергей').

В результате получим

O='Иван';
M='Мария';
No.

В данном случае было использовано правило. При этом попытка доказательства целевого утверждения

родители(O,M,'Сергей')

была заменена доказательством двух новых подцелей **отец(O,'Сергей')** и **мать(M,'Сергей').** В результате было выяснено, что первая подцель достижима, если **O='Иван'**, вторая подцель достижима, если **M='Мария'**. Следовательно, целевое утверждение верно при указанных значениях переменных.

В общей форме правило можно записать следующим образом

$$P: - P_1, P_2, \ldots, P_n,$$

где $P, P_1, P_2, \ldots, P_n$ – элементарные (атомарные) формулы. Атомарная формула имеет вид

$$P(t_1, t_2, \ldots, t_n),$$

где **P** – предикатный символ; $\mathbf{t_1, t_2,\ldots, t_n}$ – множество термов, являющихся аргументами атомарной формулы.

Терм может быть *константой*, *переменной* или *составным* термом (структурой). *Константы* языка Пролог подразделяются на *числа и атомы*. Примеры числовых констант: **-1, 176.5, 0.29e–3**. Некоторые реализации Пролога не поддерживают работу с вещественными числами. *Атомы* языка Пролог представляют символьные константы, которые или начинаются строчной буквой, или заключаются в одинарные кавычки, или состоят из специальных символов. Примеры атомов:

сергей
'Сергей'
x400
'Иван Петров'
<-->
===>

Имена переменных в Прологе начинаются с заглавных букв или символа подчеркивания "_". Областью действия переменной является утверждение. В пределах утверждения одно и то же имя принадлежит одной переменной. Два утверждения могут использовать одно имя переменной различным образом. Переменные, которым присвоены значения, называются *конкретизированными*. Существуют также *анонимные* переменные, то есть переменные без имени. Анонимные переменные обозначаются символом подчеркивания. Например, **отец('Иван', _).** Это означает, что нас не интересует, какие подстановки будут выполняться в анонимную переменную. Анонимные переменные используются в тех случаях, когда имя переменной несущественно. Вывести на печать значение анонимной переменной невозможно. Каждая анонимная переменная уникальна, т.е. отличается от всех других анонимных переменных в утверждении.

Константы и переменные образуют *элементарные объекты* языка Пролог. Все, что не может быть отнесено к переменной или константе, называется *составным термом* или *структурой*.

Структура (или составной терм) является объектом, состоящим из нескольких компонент. Структура состоит из атома, который называется *функтором*, и последовательности термов, которые рассматриваются в качестве компонент структуры. Компоненты структуры разделяются запятыми и заключаются в круглые скобки. Функтор записывается перед скобками. Например, следующая структура имеет функтор **f** и три компоненты:

f(T1,T2,T3).

Функтор **f** объединяет термы **T1, T2,** и **T3** в структуру. Число компонент структуры называется *арностью*. Так, структура **f** имеет арность 3. Отметим, что атом можно рассматривать как структуру арности 0.

Структуры можно изображать в виде деревьев. Корнем дерева является функтор структуры, а ветви, выходящие из корня представляют собой компоненты. Введем иерархическую структуру, представляющую информацию о некотором сотруднике:

```
сотрудник(имя('Сергей','Петренко'),
          адрес(улица('Тенистая',7),город('Одесса')),
          телефон('0482-68-23-42')).
```

Структура **сотрудник** содержит вложенные подструктуры: **имя, адрес, телефон**. В свою очередь, подструктура **адрес** состоит из двух подструктур: **улица** и **город**. Данную структуру можно представить в виде дерева, изображенного на рисунке 6.1. Корень дерева называется *главным функтором* структуры.

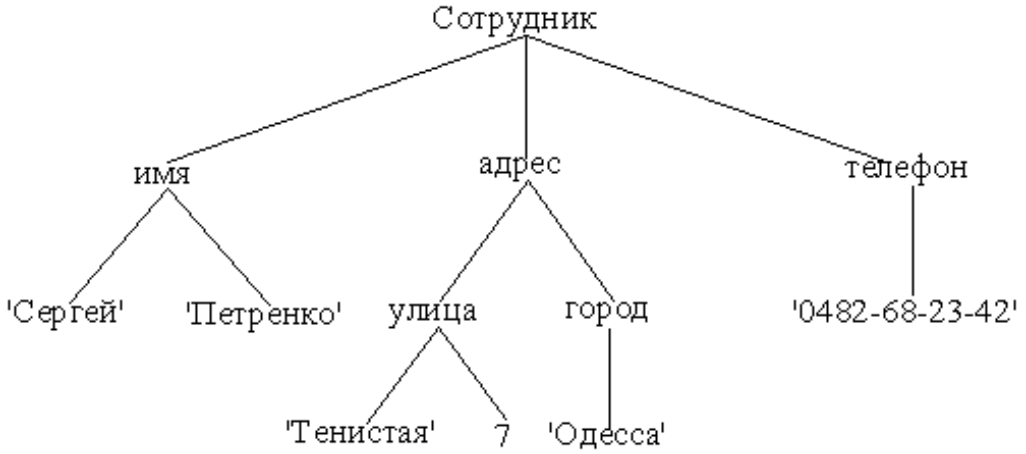


Рисунок 6.1 – Представление структуры в виде дерева

Таким образом, все структурные объекты Пролога это деревья, представляемые в программе составными термами, компонентами которых являются другие термы. Следовательно, синтаксически все объекты данных в Прологе представляют собой термы. Например, введенные выше структуры **f** и **сотрудник** – это термы.

Одной из основных операций, выполняемых над термами в ходе доказательства целевого утверждения, является *сопоставление*. Два терма сопоставляются по следующим правилам:

а) если термы **X** и **Y** – константы, то они сопоставимы, только когда одинаковы;

б) если терм **X** представлен константой или структурой, а терм **Y** – не конкретизированной переменной, то термы **X** и **Y** сопоставимы, и **Y** принимает значение **X**;

в) если термы **X** и **Y** – структуры, то они сопоставимы, когда у них совпадают главные функторы и арность, а также сопоставимы соответствующие компоненты структуры.

Возможность сопоставления двух термов проверяется с помощью оператора "**=**". Так, сопоставление

? – отец('Иван',P) = отец(O,'Сергей').

закончится успехом при **O='Иван'** и **P='Сергей'**.

Пролог находит наиболее общий унификатор термов (НОУ). В качестве примера рассмотрим вопрос:

? – отец(X, P) = отец(O,'Сергей'), отец('Иван',P).

При достижении первой цели пролог-система выполнит следующие подстановки: **X=O, P='Сергей'**. Здесь можно унифицировать переменные **X** и **O** бесконечным числом способов. Однако пролог-система находит наиболее общий унификатор **X=O**. В этом случае переменные **X** и **O** называют *сцепленными*. После достижения второй цели, значения переменных будут следующими **X='Иван', P='Сергей'**.

При рассмотрении пролог-программы полезно выделять два уровня ее смысла: декларативный и процедурный. Декларативный смысл пролог-программы связан с отношениями, объявленными (декларируемыми) в программе, он определяет, достижимо ли целевое утверждение, и при каких значениях переменных оно будет верным. Процедурный смысл определяет, как пролог-система обрабатывает отношения пролог-программы, каким образом пролог-система отвечает на вопросы.

Важным преимуществом языка Пролог является наличие встроенного механизма доказательства истинности целевых утверждений, основанного на принципе резолюции (см. §4.1.4). Поэтому программирование на Прологе в значительной степени сводится к декларации отношений предметной области, а не к определению алгоритмов выполнения тех или иных операций. Поиск решения в пролог-системах протекает в автоматическом режиме, использующем принцип резолюции и принцип возврата к альтернативным вариантам возможных решений. Наличие указанных встроенных механизмов позволяет программисту сосредоточиться больше на декларативном смысле программы и не отвлекаться на особенности вычислительного процесса. Считается, что такой декларативный подход делает программирование на Прологе более легким, чем на традиционных алгоритмических языках [53].

Вместе с тем программист не может полностью игнорировать особенности вычислительного процесса, если требуется обеспечить эффективную реализацию программ. Для более точного понимания процедурного смысла пролог-программы строят деревья вывода. Рассмотрим пример:

```
d(X,Y): – c(X, Y).                    /*1*/
d(X,Y): – b(Y, X).                    /*2*/
c(X,Y): – a(X), b(X, Y).              /*3*/
c(0,0).                               /*4*/
a(a1).                                /*5*/
a(a2).                                /*6*/
a(a3).                                /*7*/
b(a1,1).                              /*8*/
b(a1,2).                              /*9*/
b(a2,3).                              /*10*/
? – d(X,Y).                           /*11*/
```

Процесс доказательства целевого утверждения **d(X, Y)** можно представить в виде дерева вывода, изображенного на рисунке 6.2.

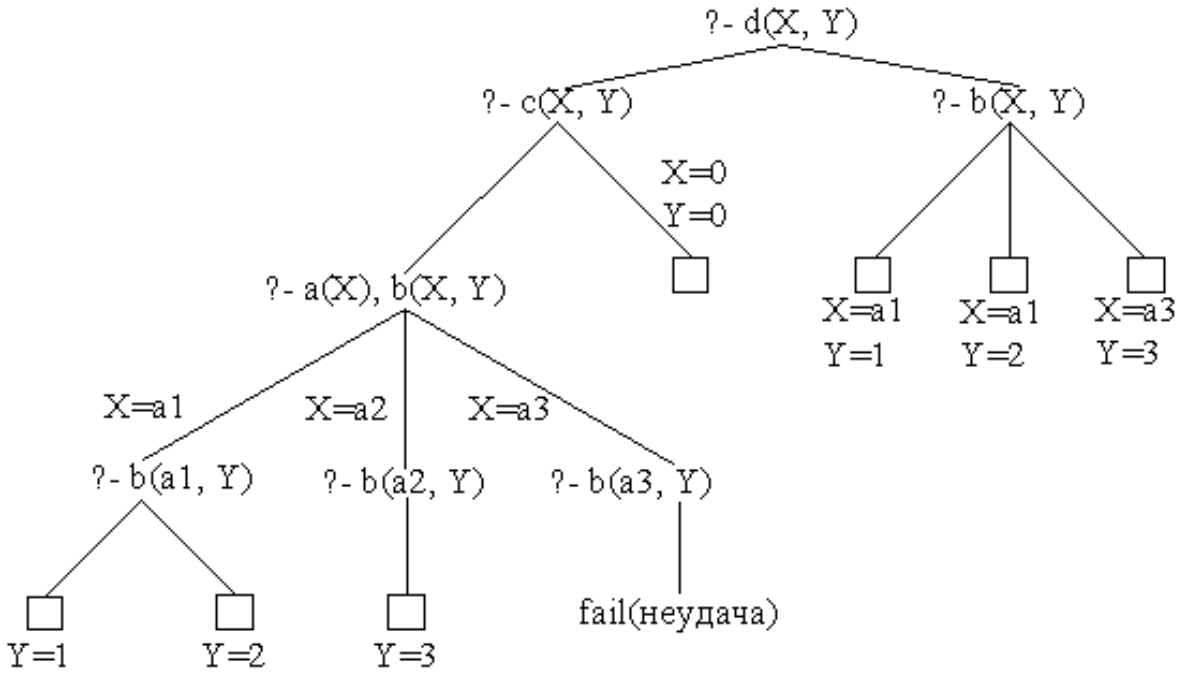


Рисунок 6.2 – Дерево вывода

Каждая ветвь дерева вывода описывает один шаг программы, а узел, находящийся в конце ветви, определяет то целевое утверждение (простое или сложное), которое должно быть доказано после определенного числа шагов выполнения. Корень дерева содержит исходное целевое утверждение. Ветви, идущие от корня, определяют последовательность шагов выполнения, с помощью которых может быть найдено решение для целевого

утверждения. В конце пути, представляющем успешно завершенную последовательность выполнения, фигурирует пустое условие, обозначаемое на рисунке квадратом. Существуют такие последовательности шагов выполнения, которые заканчиваются неудачей (**fai**l).

Согласно определениям 1 и 2, вычисление **d(X,Y)** сводится либо к вычислению **c(X,Y)**, либо **b(Y,X)**. Поэтому из вершины **? – d(X,Y)** выходят две альтернативные ветви, соединяющие вершину **? – d(X,Y)** с вершинами **? – c(X,Y)** и **? – b(X,Y)**. Вершины **? – c(X,Y)** и **? – b(X,Y)** выступают в качестве новых подцелей. Подцель **? – c(X,Y)** достижима, если достижимы подцели **a(X)** и **b(X,Y)** (определение 3) или **c(0,0)**. Доказательство истинности подцели **? – a(X), b(X,Y)** предполагает проверку выполнимости условия **a(X)**, а затем и **b(X,Y)**. Условие **a(X)** выполнимо при следующих подстановках: **X=a1**, **X=a2, X=a3** (определения 5, 6, 7). Следовательно, данное условие порождает три варианта возможных продолжений дерева вывода. При **X=a1** пролог-система проверяет выполнимость условия **? – b(X,Y)**. Данное условие выполнимо при следующих подстановках **Y=1** и **Y=2** (определение 8, 9). Таким образом, вычисление **b(a1,Y)** порождает две ветви. Пролог-система строит сначала левую ветвь (рисунок 5.13) и выдает в качестве ответа на вопрос **? – d(X,Y)** значения переменных **X=a1** и **Y=1**. Если пользователь введет с клавиатуры команду “**;**”, то пролог-система выполнит построение правой ветви условия **b(a1,Y)** и выведет на дисплей новые значения переменных **X=a1** и **Y=2**. Ввод с клавиатуры команды “**;**” заставляет пролог-систему выполнить построения альтернативной ветви дерева вывода. При этом дерево строится по правилу сверху вниз и слева направо. Определения размещаются в узлах дерева в порядке их записи в программе (сверху вниз).

Очередной ввод команды “**;**” заставит пролог-систему выполнить *возврат* к точке выбора, в которой остались нереализованные варианты выполнения. При возврате происходит автоматическое стирание подстановок, которые были сделаны в переменные после соответствующей точки выбора. В рассматриваемом случае возврат произойдет к точке выбора различных вариантов реализации условия **a(X)**. Так как вариант **X=a1** уже опробован, пролог-система выберет подстановку **X=a2.** В конечном итоге это приведет к ответу: **X=a2, Y=3**. После очередного ввода команды “**;**” опять произойдет возврат и будет опробована ветвь **X=a3**. В этом случае проверка выполнимости условия **b(a3,Y)** завершится неудачей (**fail**), так как в базе данных нет ни одного утверждения сопоставимого с **b(a3,Y)**. Неудача автоматически заставит пролог-систему осуществить возврат к возможным точкам выбора вариантов выполнения тех или иных условий. Так как все варианты выполнения условия **a(X)** уже опробованы, то пролог-система вернется к альтернативному варианту выполнения условия **c(X,Y)** и выпол-

нит подстановки **X=0** и **Y=0** (определение 4). После этого у пролог-системы останется в запасе ветвь, соответствующая определению 2.

Если будут опробованы все варианты этой ветви, и пользователь введет команду "**;**", то пролог-система ответит "**No**". В конкретном примере это означает, что больше нет вариантов доказательства целевого утверждения.

В заключение еще раз отметим, что пролог-программа содержит утверждения трех типов: факты, правила и вопросы. Факты представляют собой утверждения, которые являются всегда верными. Правила – это утверждения, истинность которых зависит от некоторых условий. Вопросы позволяют выяснить, какие утверждения являются истинными. Утверждения Пролога – это предложения Хорна, которые на Прологе записываются в виде головы и тела. При этом факт – это предложение, содержащее только голову. Вопросы содержат только тело. Правила имеют голову и тело.

Простые объекты Пролога – это атомы, числа, переменные. Структуры используются для представления сложных объектов, состоящих их нескольких компонент. Структуры образуются с помощью функторов и характеризуются именем и арностью. Структуры естественным образом отображаются в виде деревьев. Поэтому Пролог можно рассматривать как язык обработки деревьев [7].

Пролог программы имеют двойную семантику: декларативную и процедурную. Декларативная семантика касается отношений, определенных в программе. Процедурная семантика определяет порядок выполнения шагов программы. Точное понимание процедурой семантики является весьма важным для правильного выполнения программы.

6.3. Операторы

Выше рассмотрены основные объекты пролог-программы, среди которых были введены структуры. Структура состоит из термов, которые называются компонентами, и функтора. Компоненты структуры записываются в круглых скобках и разделяются запятыми, главный функтор представляется в виде атома, который записывается перед списком компонент. Например, **+(1, 2)** или **+(1, -(7, 2))**. Во втором примере компонентом структуры с главным функтором, обозначенным знаком "**+**", является структура **–(7, 2).**

Возможна более удобная операторная запись структур, арность которых равна одному или двум. Для этого необходимо объявить главный функтор структуры одноместным или двуместным оператором. При этом компоненты структуры записываются как аргументы оператора.

Так, если “**+**” и “**–**” объявить операторами, то структуры **+(1, 2)** и **+(1, -(7, 2))** можно записать в более удобной форме арифметических выражений:

```
1+2.
1+(7-2).
```

Программист может определить новые операторы в программе с помощью директивы

: – op(<приоритет>, <спецификатор>, <имя оператора>).

Здесь **<приоритет>** – аргумент, определяющий приоритетный номер оператора. Приоритет задается целым числом и обычно находится в пределах от 1 до 1200 (зависит от реализации языка). Чем выше приоритет оператора, тем меньше его приоритетный номер. Аргумент **<спецификатор>** определяет тип оператора и его ассоциативность. Различают три типа операторов: *инфиксные* (оператор располагается между двумя своими аргументами, например **X+1**), *префиксные* (оператор располагается перед единственным аргументом, например **–1**), *постфиксные* (оператор располагается после единственного аргумента, например **X+**). *Ассоциативность* определяет порядок интерпретации выражения, в котором встречается оператор. Применяют следующие виды спецификаторов:

– инфиксные операторы: **xfx, yfx, xfy**;

– префиксные операторы: **fx, fy**;

– постфиксные операторы: **xf, yf**.

В этой форме записи спецификаторов приняты следующие соглашения:

f – обозначает оператор, объявляемый с помощью директивы **op**;

x, y – аргументы оператора, которые могут быть простыми или структурными.

Если аргумент является простым или заключен в круглые скобки, то его приоритет равен нулю. Если аргумент структурный, то его приоритет равен приоритету его главного функтора. В записи спецификаторов **x** обозначает аргумент, приоритет которого выше, чем приоритет оператора **f**, а **y** – аргумент, приоритет которого выше или равен приоритету оператора **f**.

Пролог имеет ряд встроенных операторов, некоторые из которых перечислены в таблице 6.1. Если оператор объявлен со спецификатором **yfx**, то он является левоассоциативным. Например, выражение a*b*c интерпретируется как **(a*b)*c**, а не как **a*(b*c)**. Действительно, во втором случае приоритет структуры **b*c** не выше, чем приоритет оператора “ * ”. Это противоречит объявлению оператора “ * ” со спецификатором **yfx**. Поэтому верной является первая интерпретация.

Если оператор объявлен со спецификатором **xfy**, то он является правоассоциативным.

Таблица 6.1 – Операторы языка Пролог

Имя	Спецификатор	Приоритет
:-	*xfx*	1200
?-	*fx*	1200
;	*xfy*	1100
,	*xfy*	1000
.	*xfy*	750
=..	*xfx*	700
=	*xfx*	700
=\=	*xfx*	700
is	*xfx*	700
=:=	*xfx*	700
<	*xfx*	700
=<	*xfx*	700
>	*xfx*	700
>=	*xfx*	700
==	*xfx*	700
\==	*xfx*	700
–	*yfx*	500
+	*yfx*	500
/	*yfx*	400
*	*yfx*	400
div	*yfx*	400
mod	*xfx*	300

6.4. Арифметические выражения

Язык Пролог предназначен, главным образом, для программирования задач символьной обработки информации. Тем не менее, в любую пролог систему включаются все обычные арифметические операторы: **+** (сложение), **-** (вычитание), ***** (умножение), **/** (деление), **mod** (остаток от деления целых чисел), **div** (целочисленное деление).

В Пролог входят также все необходимые операторы сравнения чисел: **<** (меньше), **>** (больше), **>=** (больше или равно), **=<** (меньше или равно), **=\=** (не равно), **=:=** (равенство целочисленных выражений).

Арифметические выражения в Прологе строят из арифметических операторов, числовых переменных, констант и скобок. При этом арифметические выражения представляют собой структуры. Например, выражение **1+2*3** представляется структурой, изображенной на рисунке 6.3. Если пролог-системе задать вопрос

? – X=1+2*3.

то в результате получим **X=1+2*3**, а не **X=7**.

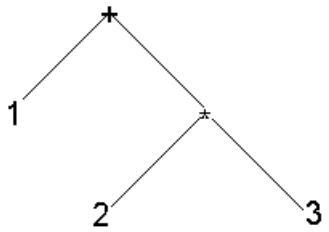


Рисунок 6.3 – Представление арифметического выражения **1+2*3**.

Объясняется это тем, что оператор "**=**" выполняет сопоставление и переменной X ставит в соответствие структуру, изображенную на рисунке 6.3. При этом арифметическое выражение не вычисляется. Для того чтобы заставить пролог-систему выполнить необходимые вычисления, следует применить специальный оператор **is**. Например,

? – X is 1+2*3.

В этом случае ответ будет **X=7**.

Оператор **is** определен как инфиксный оператор, который в общем виде записывается так:

X is Y.

Здесь **X** – или число, или не конкретизированная переменная, а **Y** – арифметическое выражение. Если **X** – число, то вычисляется значение выражения **Y**, которое сравнивается с **X**. При равенстве значений **X** и **Y** попытка доказательства **X is Y** заканчивается успехом. Если **X** – не конкретизированная переменная, то вычисленное значение выражения **Y** приписывается переменной **X**. Примеры:

```
? – 3 is 1+2.
Yes
? – X is 7-2.
X=5
? – 1+2 is 1+2.
No
? – X=1, X is X+1.
No
```

Рассмотрим еще один пример. Определим функцию двух переменных *f(x, y)=x+y+1*:

```
f(X,Y,F):- F is X+Y+1.
```

Тогда в ответ на вопрос

```
? – f(1, 2, F).
```

будут получен ответ **F=4**. Однако доказательство целевого утверждения

```
? – f(X,2,4).
```

завершится неудачей, так как оператор **is** “работает” только в направлении справа налево.

Точно так же, как и оператор **is**, операторы сравнения вычисляют свои аргументы перед выполнением:

```
? – 5+3-1>2+1.
Yes
```

В большинстве современных реализаций языка Пролог имеются встроенные предикаты для вычисления элементарных функций: возведение в квадрат (**sqr**), логарифм (**ln**), синус (**sin**), косинус (**cos**), абсолютное значение (**abs**) и др.

6.5. Списки и рекурсия

Список – это структура данных, составленная их произвольного числа элементов. Элементы списка отделяются друг от друга запятыми и заключаются в квадратные скобки. Например, список из четырех элементов **'a', 'b', 'c', 'd'** на языке Пролог запишется в виде:

```
['a', 'b', 'c', 'd'].
```

Список полезно рассматривать как структуру данных, состоящую из двух частей: головы списка и хвоста списка. Для приведенного выше списка элемент **'a'** – это голова списка, а подсписок **['b', 'c', 'd'] –** хвост списка.

Во внутреннем представлении списки – это структуры, состоящие из головы и хвоста. В качестве функтора, объединяющего голову и хвост, используется “**.**” . Поэтому список можно представить в виде:

.(**Голова, Хвост**)

Представляя список [**'a', 'b', 'c', 'd'**] в виде структуры, получаем

```
.(a, . (b, . (c, . (d, [ ] ))))
```

Здесь пара квадратных скобок **[ ]** обозначает пустой список.

Для представления списка в виде структуры данных, состоящей из головы и хвоста, в Прологе широко используется еще одно обозначение, в котором голова и хвост списка отделяются вертикальной чертой. Например, в записи **[H|T]** переменная **H** – представляет голову списка, а переменная **T** – хвост. Применив символ "|", список **['a', 'b', 'c', 'd']** можно представить следующими различными способами:

['a','b','c','d']=['a'| ['b','c','d']]=['a', 'b'| ['c','d']]=
=['a', 'b', 'c'| ['d']]=['a', 'b', 'c', 'd'| [ ]].

При попытке доказательства целевого утверждения

? – [x1, x2, x3 | x4]=[1, 2, 3, 4].

пролог-система выдаст ответ: **x1=1, x2=2, x3=3, x4=[4]**.

Над списками часто выполняют следующие операции: добавление элемента в список, удаление элемента из списка, объединение списков, поиск элемента в списке.

Наиболее просто определяется предикат, осуществляющий добавление элемента в список:

добавить (X, L, [X|L]).

Здесь **X** – добавляемый элемент; **L** – список, в который добавляется элемент; **[X|L]** – результирующий список. Таким образом, элемент **X** добавляется в начало списка **L**.

Представление списков в виде головы и хвоста, где хвост, в свою очередь, тоже список, является рекурсивным. Поэтому обработка списков часто выполняется с помощью рекурсивных предикатов. Рекурсивными называют предикаты, в определениях которых содержатся ссылки на самих себя. Примеры простейших рекурсивных предикатов, выполняющих поиск элемента в списке и объединение двух списков, были приведены в §4.1.8. В общем случае все такие определения строятся по следующей схеме [53]:

```
предикат(…[ ]…).
предикат(…[Голова|Хвост]…): –
                обработка(Голова),
                предикат(…[Хвост]…).
```

Рекурсивные вызовы прекратятся, когда хвост списка окажется пустым списком. В приведенном определении первое утверждение определяет условие выхода из рекурсии, а второе утверждение – правило, предусматривающее при каждом вызове обработку очередного элемента списка и ре-

курсивный вызов определяемого предиката, аргументом которого является хвост списка.

Определим, например, предикат **реверс(X,Y)**, где список **Y** представляет инверсную копию списка **X**:

```
реверс([ ], [ ]).                          %1
реверс([H|T],Y): – реверс(T,Ys),           %2
                   append(Ys,[H],Y).
```

Первое утверждение фиксирует условия выхода из рекурсии, а второе – определяет правило преобразования списка **X** в список **Y**. Для того чтобы выполнить инверсию списка **[H|T]** , необходимо выполнить инверсию хвоста **T** и получить список **Ys**, а затем добавить голову **H** в конец списка **Ys** и получить результирующий список **Y**. Интересно отметить, что предикат **реверс(X,Y)** позволяет не только выполнять преобразование списка **X** в список **Y**, но и осуществлять обратные преобразования. Например, на вопрос

```
? – реверс(X, [1, 2, 3]).
```

будет получен ответ **X=[3, 2, 1].**

Удаление элемента **X** из списка **L** можно представить в виде предиката

```
удалить(X, L, L1),
```

где **L1** – результирующий список. Если **X** является головой списка **L**, то результирующий список **L1** – это хвост списка **L**. Если **X** находится в хвосте списка **L**, то для удаления **X** необходимо рекурсивно вызвать предикат **удалить**, подставив в качестве второго аргумента хвост списка **L**. Данным правилам будет соответствовать следующее определение [7]

```
удалить(X, [X|T], T).
удалить(X, [H|T], [H|T1]): –
                  удалить(X, T, T1).
```

Предикат **удалить** можно использовать и в обратном порядке. Например,

```
? – удалить (1, L, ['a', 'b']).
L=[1, 'a', 'b'];
L=['a', 1, 'b'];
L=['a', 'b', 1].
```

В данном случае выполняется вставка элемента **1** в произвольные позиции списка. В итоге получаются различные списки **L**, исключив из которых элемент **1**, получим список **['a', 'b'].**

6.6. Управление возвратом (отсечение)

В Пролог имеется встроенный предикат, ограничивающий возвраты. С этой целью отсекаются некоторые ветви дерева вывода, к которым возможен возврат. Предикат, обеспечивающий отсечение, обозначается знаком "**!**" . Рассмотрим его применение на простых примерах.

Определим предикат **abs(X,Y)**, присваивающий переменной Y абсолютное значение **X**:

$$Y = \begin{cases} X, & X \geq 0 \\ -X, & \textit{иначе}\ . \end{cases}$$

Попробуем непосредственно реализовать данное выражение на Прологе, выделив случай $X \geq 0$:

```
% Определение1
abs(X,Y): – X>=0, Y is X;
                  Y is –X.
```

Проверим данное определение на целевом утверждении

```
? – abs(3, -3).
```

Пролог-система подставит вместо **X** значение **3**, а вместо **Y** – значение **-3** и попытается выяснить выполнимость условий **3>=0, -3 is 3**. Так как второе условие не выполнимо, то пролог-система выполнит возврат и проверит альтернативную версию **Y is -X**, то есть **-3 is -3**. В этом случае выполнение предиката **is** завершится удачей. Следовательно, в ответ на вопрос **abs(3,-3)** будет получен ошибочный ответ **Yes** (да).

Попробуем исправить ошибку. Определим предикат **abs(X,Y)** следующим образом:

```
% определение 2
abs(X, Y): –  X>=0, Y is X;
              X<0, Y is –X.
```

Теперь при ответе на вопрос

```
? – abs(3, -3).
```

будет получен ответ **No** (нет). Определение 2 будет корректно и при других значениях аргументов **X** и **Y**. Однако возникает вопрос об его эффективности. В данном определении выполняются две проверки: **X>=0** и **X>0**. В то же время ясно, что при выполнении условия **X>=0** вторая строка опре-

деления 2 не должна выполняться. Для того чтобы пролог-системе сообщить это, воспользуемся предикатом отсечения. Введем третье определение предиката **abs(X,Y)**:

```
% определение 3
abs(X, Y): -  X>=0, !, Y is X;
                       Y is -X.
```

Предикат отсечения "**!**" всегда успешно выполняется. Использование предиката "**!**" требует осторожности, так как после его выполнения невозможен возврат к точкам выбора, порождаемым определением, в котором используется предикат отсечения.

Рассмотрим дерево вывода целевого утверждения **? – abs(3, -3)**, изображенное на рисунке 6.4. Здесь показано, что определение 3 порождает альтернативы, которым соответствуют две ветви, исходящие из узла **? – abs(3, -3)**. После успешной проверки условия **3>=0**, выполняется предикат отсечения, который стирает альтернативную ветвь определения. Теперь, после неудачной проверки выполнимости утверждения **-3 is 3**, пролог-система не имеет возможности вернуться к альтернативной версии и возвращает в качестве результата значение **No** (нет). Таким образом, в рассмотренном примере предикат отсечения был использован для программирования взаимоисключающих версий определения.



Рисунок 6.4 – Дерево вывода для вопроса **? – abs(3, -3)**

Предикат отсечения применяется также для прерывания рекурсивных вызовов и устранения бесконечных циклов. В качестве примера рассмотрим программу, обеспечивающую вычисление квадратного корня $y = \sqrt{x}$. Для этого воспользуемся итерационной формулой:

$$\begin{aligned} y_n &= y_{n-1} + \frac{1}{2}\left(\frac{x}{y_{n-1}} - y_{n-1}\right), \\ y_0 &= 1, \end{aligned} \qquad (6.1)$$

где y_n – значение корня на n-ом шаге вычислений, y_0=1 – начальное приближение. Вычисления заканчиваются, когда будет достигнута заданная точность вычислений $\varepsilon = \left| y_n - y_{n-1} \right|$. Пусть ε=10^{-5}. Определим предикат **квадратный_корень(X,Y)**:

```
квадратный_корень(X,Y): –
        X>0, !, поиск_квадратного_корня(X, Корень, 1),
         ( Y is Корень ;
           Y is  -Корень);
           X=:=0, Y is 0.
```

В данном определении учитываются два возможных варианта вычисления корня. Если **X>0**, то с помощью предиката **поиск_квадратного_корня(…)** осуществляется поиск значения корня с заданной точностью. Если **X=0**, то **Y** присваивается нулевое значение. Предикат отсечения здесь используется для программирования указанных взаимоисключающих вариантов вычислений. Покажем, как с помощью предиката отсечения прерываются рекурсивные вызовы. Определим для этого предикат **поиск_квадратного_корня(X, Корень, Приближение)**.

```
поиск_квадратого_корня(X, Корень, Приближение): –
 Yn is (X/ Приближение+Приближение)/2,
 (abs(Yn - Приближение)< 1.0e-5, !, Корень is Yn;
 поиск_квадратного_корня(X, Корень, Yn)).
```

Здесь переменная **Приближение** соответствует y_{n-1} в формуле (6.1). Когда абсолютное значение разности между y_n-y_{n-1} станет меньше 10^{-5}, то с помощью предиката отсечения стирается альтернативная ветвь, соответствующая рекурсивному вызову предиката **поиск_квадратного_корня**. Переменной **Корень** присваивается текущее значение Yn, и происходит выход из рекурсии.

В определениях рассмотренных выше предикатов использована группа. *Группа* – это заключенная в скобки последовательность вариантов, от-

деленных друг от друга точкой с запятой. Группа позволяет, с одной стороны, рассматривать совокупность вариантов как одно утверждение, а с другой – ограничивать область действия связки ”или”. По существу, группа – это тело правила без головы.

Уточним область действия предиката отсечения. Рассмотрим пример. Определим предикат **абсолютный_элемент(X,L)**, который будет иметь истинное значение, если список **L** будет содержать числовой элемент, абсолютное значение которого равно **X**. Приведем два определения этого предиката:

```
% Определение 1
абсолютный_элемент(X, L): –
        элемент(E, L), abs(E, X).

% Определение 2
абсолютный_элемент(X, L): –
      элемент(E, L),
      (E>=0, !, X is E;
            X is –E).
```

Во втором определении предикат **abs(E,X)** заменен группой, которая представляет его тело. Предикат **элемент(E,L)** соответствует предикату **member(E,L)**, определенному в §4.1.7. Проверим данные определения на вопросе:

```
? – абсолютный_элемент(3, [2, -3]).
```

При использовании первого определения ответ на вопрос будет положительным, а применение второго определения даст отрицательный ответ **No** (нет). Происходит это из-за того, что предикат отсечения расширил область своего действия. В первом определении он действует локально в пределах определения **abs(E,X)**. Во втором определении областью его действия является весь предикат **абсолютный_элемент(X,L)**. Поясним сказанное с помощью дерева вывода, изображенного на рисунке 6.5.

Как видно из рисунка, предикат “!” отсекает не только альтернативные ветви, порождаемые определением **abs(E,X)**, но и ветви, обусловленные недетерминированным исполнением предиката **элемент**. Поэтому правильным будет первое определение предиката **абсолютный_элемент**.

В заключение отметим, что введение предиката отсечения – это отступление от логических принципов. Отсечение делает программы чувствительными к порядку следования предложений, что нарушает декларативное восприятие программы. Поэтому при программировании на Прологе следует избегать применения отсечения без достаточных оснований.



Рисунок 6.5 – Вывод утверждения **абсолютный_элемент(3, [2, -3])**

6.7. Отрицание в языке Пролог

Определим предикат **не_элемент(X,L)**, противоположный предикату **элемент(X, L)**:

```
не_элемент(X, L): –
        элемент (X, L),!,fail;
        true.
```

Здесь предикат **не_элемент** определяется через предикат **элемент**. Если подцель **элемент(X,L)** доказуема, то отсечение исключает выбор альтернативы, а **fail** вызывает неудачу. Иначе считается, что цель **не_элемент(X, L)** достижима (**true** – успех). Такое определение предиката **не_элемент** основано на использовании *предположения о замкнутости мира*. При этом под миром понимается множество утверждений конкретной программы. Полагается, что в этих утверждениях содержатся все знания о решаемой задаче. Если некоторое утверждение не представлено в программе, то считается, что оно ложно. В этом случае доказательство истинности исходного утверждения подменяется доказательством недоказуемости противоположного утверждения, что не одно и то же. Поэтому рассмотренной схемой

построения отрицания необходимо пользоваться осторожно. Пусть в программе имеются следующие факты и правила:

```
отец('Иван', 'Сергей').
отец('Иван', 'Ольга' ).
отец('Петр', 'Николай').
не_отец(X, Y): –
        отец(X, Y), !, fail;
        true.
```

При попытке ответа на вопрос

```
?- не_отец ('Петр', 'Татьяна').
```

будет получен положительный ответ. Но это означает только то, что база данных не содержит утверждения **отец('Петр', 'Татьяна').**

Многие реализации Пролога содержат встроенный предикат **not**. Тогда предикат **не_отец(X, Y)** можно определить следующим образом:

```
не_отец(X, Y): – not(отец(X, Y)).
```

При вызове предиката **not** на место аргумента подставляется конкретное утверждение, для которого необходимо найти отрицание. Обычно предикат **not** описывается в виде префиксного оператора. Поэтому цель **not(отец(X, Y))** можно также записать в виде **not отец(X, Y)**.

6.8. Метаусловия

Пролог допускает использование переменных не только в качестве аргументов предикатов, но и вместо условий (предикатов). Такие условия (переменные) называют *метаусловиями (метапеременными)*. При выполнении метаусловия переменная, представляющая такое метаусловие, должна быть конкретизирована. В приведенном ниже правиле

```
p(A, B, C): – A, B; C.
```

переменные **A**, **B**, **C** представляют метаусловия. При вызове предиката **p(A, B, C)** на место аргументов должны подставляться конкретные условия. Используя метаусловия, определим предикат **not**:

```
not(P): – P, !, fail;
          true.
```

В данном определении **P** – метаусловие. Подстановка вместо **P** конкретного утверждения обеспечивает успех, если метаусловие **P** не достижимо. Предикат **not(P)** реализует отрицание в соответствии с предположением замкнутости мира.

Введение метаусловий позволяет реализовать управляющую конструкцию **if_then_else**, характерную для традиционных языков программирования:

if(Если, То, Иначе): – Если, !, То; Иначе.

Здесь **Если, То, Иначе** – метаусловия. Предикат **if** удобно использовать при программировании взаимоисключающих версий. Например, переопределим предикат **abs(X, Y)**:

abs(X, Y): – if(X>=0, Y is X, Y is -X).

В этом определении метаусловие **Если** задано отношением **X>=0**. Если данное отношение справедливо, то выполняется условие **Y is X**, иначе условие **Y is -X**.

Подобным образом можно использовать предикат **if** при выборе максимального из двух чисел:

max(X, Y, Z): – if(X>=Y, Z is X, Z is Y).

Отметим, что Пролог содержит встроенный предикат **call(P)**, который обеспечивает оценивание (доказательство) условия **P**. Вызов предиката **call(P)** эквивалентен использованию метаусловия. Например, предикат **not** можно определить так:

not(P): – call(P), !, fail; true.

6.9. Организация циклов

При решении практических задач часто существует необходимость в повторении некоторых операций. В процедурных языках программирования для этого используются операторы циклов или рекурсия. Возможность организации циклических вычислений с помощью рекурсии рассматривалась выше. Здесь рассмотрим организацию циклов в Прологе с использование механизма возврата.

Вернемся к рассмотренному в § 6.2. примеру, где представлена база данных, состоящая из фактов:

отец(Отец, Ребенок).

Определим предикат **дети(Отец)**, обеспечивающий вывод на экран имен всех детей конкретного отца. В этом случае программа должна последовательно просматривать все факты базы данных при конкретном значении переменной **Отец** и выводить на экран соответствующее значение переменной **Ребенок**. Для того чтобы заставить программу перебирать факты базы данных, организуем состояние искусственной неудачи (**fail**). Это приведет к тому, что пролог-система выполнит возврат к иному варианту (если такой имеется) унификации предиката **отец(Отец, Ребенок)**. Вывод на экран имени ребенка будем выполнять с помощью встроенного предиката **write(X)**, где **X** – переменная, значение которой отображается на экране. Тогда для предиката **дети(Отец)** можно записать следующее определение:

```
дети(Отец): –
            отец(Отец, Ребенок),
            write(Ребенок), fail;
            true.
```

На рисунке 6.6 изображено дерево вывода целевого утверждения

? – дети('Иван').

Имеются два варианта унификации предиката **отец('Иван',Ребенок)**. Пролог-система проверяет оба варианта. После вывода на экран первого значения переменной **Ребенок**, пролог-система встречает условие fail и переходит к варианту **Ребенок='Анна'**. В конце второго пути пролог-система опять встречает условие **fail** и переходит к третьему варианту, который завершается условием **true**.

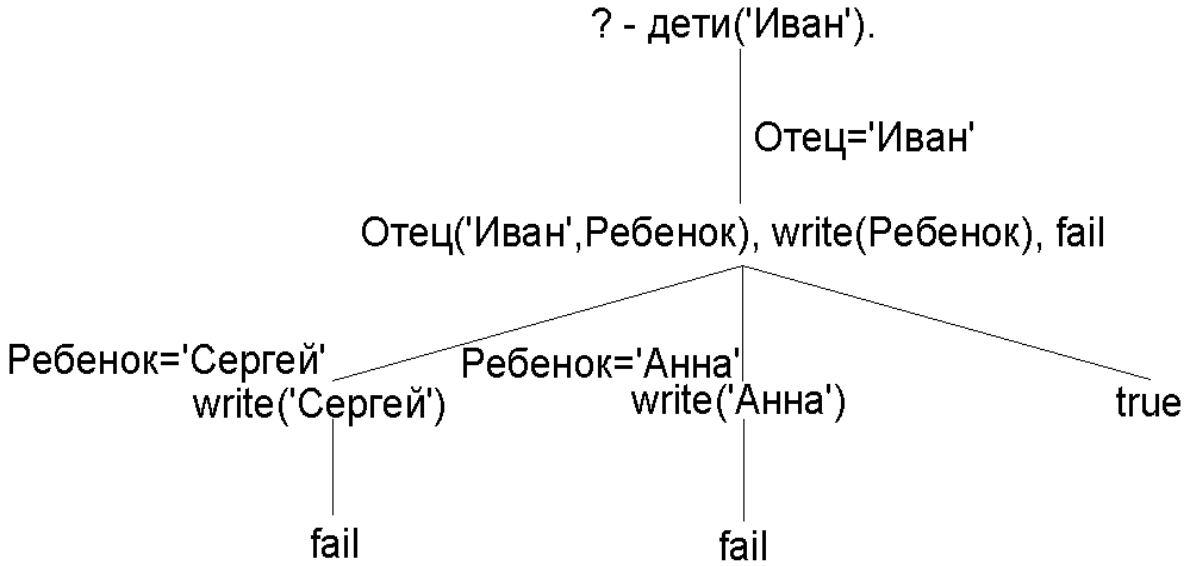


Рисунок 6.6 – Дерево вывода целевого утверждения дети('Иван')

Схема выполнения пролог-программы, основанная на создании искусственной неудачи, называется *циклом с возвратом.*

Рассмотренный предикат **дети(Отец)** можно обобщить. Определим предикат, который выполняет поиск всех решений некоторого известного метаусловия:

```
все_решения(Условие, X): –
        Условие, write(X), fail;
        true.
```

Здесь переменная **X** является также аргументом метаусловия **Условие**.

Все решения, соответствующие предикату **дети(Отец)**, можно получить с помощью вызова:

```
все_решения(отец(Отец, X), X).
```

С помощью рассмотренной схемы организации циклов легко организовывать вычисления, соответствующие вложенным циклам процедурных языков. Пусть требуется вывести на экран общие элементы списков **L1** и **L2**. Иными словами, для каждого элемента списка **L1** необходимо найти соответствующие элементы списка **L2**. Данная задача может быть решена следующим образом:

```
общие_элементы(L1, L2): –
        элемент(E, L1), элемент(E, L2),write(E), fail;
        true.
```

В рассмотренных примерах пролог-система возвращалась к точкам выбора, порождаемым предикатами: **отец(Отец, Ребенок), элемент(E, L1)**, **элемент(E, L2)**. Предикаты, создающие точки выбора, называют *недетерминированными*. Иногда необходимо осуществлять повторное выполнение предикатов, не создающих точек выбора (детерминированных). Для этого можно организовать цикл с возвратом, создав искусственные точки выбора.

Пусть требуется последовательно вводить с клавиатуры числа и вычислять для каждого числа квадратный корень. В нашем распоряжении имеются два предиката: встроенный предикат **read(X)** , обеспечивающий чтение значений с клавиатуры (подробнее предикаты ввода рассматриваются в следующем параграфе), и предикат, вычисляющий квадратный корень **квадратный_корень(X, Y)**. Для того чтобы организовать повторное выполнение указанных предикатов, необходимо перед предикатом **read(X)** поместить предикат, который будет создавать бесконечное число точек выбора и всегда выполняться успешно (рисунок 6.7).

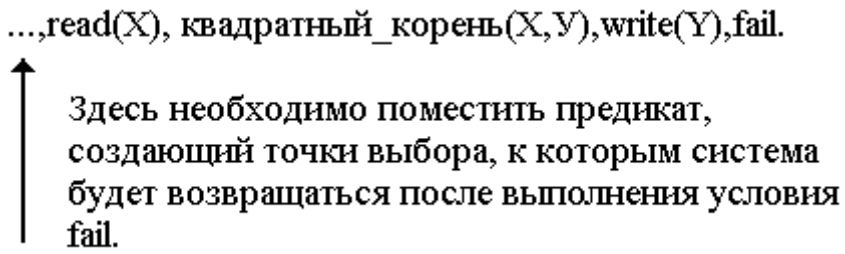


Рисунок 6.7 – Создание искусственных точек выбора

Таким предикатом является предикат **repeat**, встроенный во многие реализации языка Пролог:

```
repeat: – true;
          repeat.
```

Дерево вывода предиката **repeat** изображено на рисунке 6.8. Каждый вызов предиката **repeat** порождает новую ветвь вычислений.

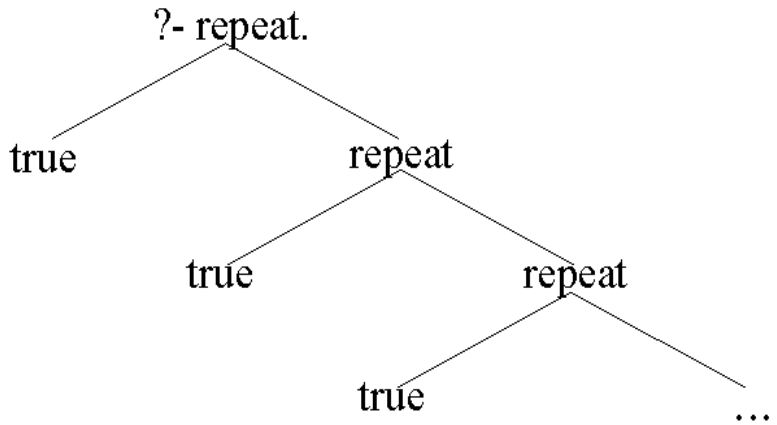


Рисунок 6.8 – Дерево вывода предиката **repeat**

Воспользуемся предикатом **repeat** и напишем циклическую программу, вычисляющую значение квадратного корня очередного числа, введенного с клавиатуры. Если вместо числа с клавиатуры будет введен атом **'стоп'**, то вычисления прекращаются:

```
цикл: –
        repeat,nl,write('число?'), read(X),
        (X='стоп', !;
        квадратный_корень(X, Y),
        write('корень='), write(Y),fail).
```

Здесь встроенный предикат **nl** обеспечивает перевод курсора на новую строку.

Предикат **repeat** можно переопределить так, чтобы он обеспечивал построение циклов с заданным числом повторений. В этом случае он должен порождать конечное множество ветвей вычислений, а не бесконечное. Для этого введём предикат **repeat** с одним аргументом **N**, фиксирующим количество циклов:

```
repeat(1): – !.
repeat(N): – true;
             M is N-1, repeat(M).
```

Рассмотренные выше циклические программы вычисления квадратного корня можно было выполнить, используя рекурсивную программу:

```
цикл: –
      nl, write('число?'), read(X),
      (X='стоп', !;
      квадратный_корень(X, Y),
       write('корень='),write(Y),цикл).
```

Сравнив два возможных варианта циклических вычислений, отметим, что цикл с возвратом используется ограниченно. Это связано с тем, что при каждом возврате конкретизации переменных теряются. Поэтому передача промежуточных результатов от одной ветви вычислений к другой невозможна. Циклы с возвратом обычно применяются в тех случаях, когда промежуточные результаты выводятся на экран или фиксируются в базе данных с помощью специальных встроенных предикатов, рассматриваемых ниже.

Рекурсивный вариант организации циклических вычислений, по сравнению с циклом **repeat-fail**, требует бóльшего объема стековой памяти, так как на каждом шаге рекурсии все конкретизации переменных сохраняются. Исходя из этих соображений, цикл **repeat-fail** иногда предпочитают рекурсии.

6.10. Встроенные предикаты

В параграфе описываются встроенные предикаты Пролога, обеспечивающие выполнение операций ввода-вывода, обработки термов и обновления базы данных. Встроенные предикаты реализуются либо средствами самого Пролога, либо средствами языков более низкого уровня, например, ассемблера. Встроенные предикаты не могут быть изменены пользователем.

6.10.1. Предикаты ввода-вывода

Пролог взаимодействует с несколькими *входными* и *выходными потоками.* В каждый момент времени Пролог взаимодействует с одним входным и одним выходным потоками, которые называются текущими потоками. По умолчанию, текущим входным потоком является клавиатура, а текущим выходным потоком – дисплей.

Перечень встроенных предикатов, предназначенных для управления входными и выходными потоками, приведен в таблице 6.2. Для считывания значений из входных потоков и записи их в выходные потоки применяют следующие предикаты:

read(X) – вызывает чтение очередного терма из входного потока и сопоставление его с **X**, вводимые термы должны заканчиваться точкой, в конце файла **X** конкретизируется атомом **end_of_file**;

write (X) – выводит терм X в текущий выходной поток;

nl – переход на новую строку;

tab(N) – выводит в выходной поток N-пробелов;

put(X) – выводит символ с ASCII кодом X на терминал;

get0(X) – считывает один символ с клавиатуры и сопоставляет его **X**;

get(X) – сопоставляет **X** с первым символом, отличным от пробела;

display(X) – выводит терм **X** в стандартной скобочной записи в текущий выходной поток.

Таблица 6.2– Предикаты управления потоками

Входной поток	Выходной поток	Интерпретация предиката
see(<имя файла>)	**tell(<имя файла>)**	Определяет в качестве текущего входного или выходного потока соответствующий файл.
seeing(<имя файла>)	**telling(<имя файла>)**	Отождествляет имя файла с текущим входным или выходным потоком.
seen	**told**	Закрывает текущий входной или выходной поток и опять связывает с текущим входным или выходным потоком пользовательский терминал.

Файлы Пролога являются текстовыми и обрабатываются последовательно. Типовая последовательность вызова встроенных предикатов при выполнении ввода из файла является следующей:

```
…,
see('имя файла'),                    % открытие файла
read(X),                             % чтение терма X
…,                                   % обработка
seen.                                % закрытие файла
```

Аналогично строится работа с выходным потоком:

```
…,
tell('имя файла'),                   % открытие файла
write(X),                            % запись терма X
…,
told.                                % закрытие файла
```

Предикаты **seen** и **told** закрывают соответственно входные и выходные файлы и опять связывают с текущим входным или выходным потоком терминал пользователя.

Рассмотрим применение предикатов ввода-вывода на примерах.

Пример 1. Вывод списка на экран.

```
вывод _списка([ ]).
вывод _списка([H | L]):– write(H),nl,вывод _списка(L).
```

В этом случае список **['мяч ', 'заяц', 'мишка']** будет отображен на экране в виде:

```
мяч
заяц
мишка
```

Пример 2. Обработка запроса к базе данных.

Пусть имеется база данных, содержащая факты:

```
адрес (Фамилия, Адрес).
```

Определим предикат, запрашивающий фамилию и осуществляющий вывод адреса:

```
вывод _адреса:–
        repeat, write ('Фамилия?'), nl,
        read (X),
        (X= 'стоп', ! ;
        адрес( X, Адрес), write('Адрес:'),
        write( Адрес), nl, fail).
```

Этот предикат повторяет запрос фамилии, пока не будет введен атом **‘стоп’**. Вводимые фамилии должны заканчиваться точкой.

Пример 3. Определим предикат, вычисляющий значения арифметических выражений, вводимых с клавиатуры:

```
калькулятор:–
        write(‘Введите- выражение’), nl,
        read(X),
        (X= ’стоп ’, ! ;
        Y is X, write(Y), nl,
        калькулятор).
```

В этом случае в ответ на вопрос

```
?-калькулятор.
```

пролог-система предлагает пользователю ввести арифметическое выражение. После ввода выражения, например,

```
107+(302*3)/4.
```

оно будет сопоставлено переменной **X** и вычислено с помощью предиката **is**. В результате на экран будет выведено значение **333.5.**

Пример 4. Напишем программу, выполняющую чтение арифметических выражений из входного файла, вычисление их значений и запись результатов в выходной файл:

```
% основной предикат
обработка_файлов(Файл1,Файл2):-
    see(Файл1),                    % открытие файлов
    tell(Файл2),
    чтение_обработка_запись,
    seen, told.                    % закрытие файлов

% предикат, выполняющий ввод выражений,
% вычисления и запись результатов
чтение_обработка_запись:–
            read( Выражение ),
            обработка(Выражение, Результат),
            запись(Выражение, Результат),
            Выражение \= = end_of_file,
            чтение_обработка_запись.
чтение_обработка_запись:–!.

% предикат, вычисляющий значения выражения
обработка(end_of_file,end_of_file):–!.
```

```
обработка(Выражение, Результат):–
        Результат is Выражение, ! .

% запись результатов в файл
запись(end_of_file,_):–!.
запись(Выражение, Результат):–
write(Выражение), write ('='),
write(Результат), write ('.'), nl, !.
```

Для запуска программы необходимо вызвать основной предикат, указать имена входных и выходных файлов. Например,

```
?–обработка_файлов('f1.txt', 'f2.txt').
```

Здесь предполагается, что файлы **f1.txt** и **f2.txt** размещаются в текущем каталоге. Если есть необходимость в размещении этих файлов в иных каталогах, то необходимо указывать полные пути доступа к файлам.

Входной файл **f1.txt** может быть подготовлен с помощью любого текстового редактора. Пусть файл **f1.txt** содержит следующие выражения:

```
1+2*3.
7-27/3.
1+sin(1.5417).
```

Тогда результирующий файл будет содержать следующие записи:

```
1+2*3=7.
7-27/3= -2.0.
1+sin(1.5417)=1.99999982.
```

В рассмотренном примере процедура, выполняющая чтение, обработку и запись результатов, определена рекурсивно. При большом числе записей во входном файле это может привести к исчерпанию стековой памяти. В этом случае можно воспользоваться циклом **repeat–fail**:

```
чтение_обработка_запись(Файл1, Файл2):–
repeat,
        read(Выражение),
        обработка(Выражение, Результат),
        запись(Выражение, Результат),
        Выражение == end_of_file,!.
```

Здесь состояние неудачи (**fail**) возникает всякий раз при проверке равенства терма, представленного переменной **Выражение**, и атома **end_of_file.** Когда входной файл будет исчерпан, переменная **Выражение** получает значение **end_of_file**, и указанная проверка успешно завершается,

что приводит к отсечению всех ветвей вычислений, порождаемых предикатом **repeat**.

Многие версии языка Пролог имеют и другие встроенные предикаты ввода-вывода, учитывающие особенности конкретных внешних устройств. Программировать сложный ввод-вывод с помощью таких предикатов проще, чем на основе стандартных предикатов **read** и **write**.

6.10.2. Предикаты обработки термов

Рассмотрим некоторые встроенные предикаты Пролога, выполняющие обработку термов. Такие предикаты обеспечивают проверку типа значений терма, выясняют, была ли некоторая переменная конкретизирована, выделяют составные части терма, образуют новые термы и т. п.

Выяснить тип терма можно с помощью следующих встроенных предикатов, которые имеются во многих реализациях языка Пролог:

var(X) – возвращает значение значение истина, если **X** – не конкретизированная переменная;

nonvar(X) – предикат **nonvar** будет истинным, если **X** – терм любого вида, кроме не конкретизированной переменной;

atom(X) – предикат atom будет истинным, если **X** – атом;

atomic(X) – предикат **atomic** даст значение истина, если **X** обозначает целое число или атом;

integer(X) – предикат **integer** будет истинным, если **X** обозначает целое число.

Следующие примеры иллюстрируют применение некоторых предикатов:

```
? – var(X).
        Yes
 ? – X=пролог, var(X).
        No
? – X= пролог, integer(X).
        No
? – atom(пролог).
        Yes
? – atom(220).
        No
? – atomic(220).
        Yes
```

В Прологе имеется группа встроенных предикатов, которые обеспечивают декомпозицию термов на составные части и конструирование термов из простых элементов. К ним относятся следующие предикаты: **name(A,L), functor(T,F,A), arg(N,T,A), T = ..L**. Рассмотрим их.

Вызов предиката name вернёт значение истина, если **L** – список ASCII-кодов символов, образующих атом **A**, например:

```
? – name (X, [97,105]).
X = ai
? – name (ab,L).
L = [97,98]
```

Предикат **functor(T,F,A)** будет истинным, если главный функтор **F** имеет терм **T** с арностью **A**, например:

```
? – functor(f(a, x), F, A).
F = f, A = z
```

Вызов предиката **arg(N,T,A)** вернёт значение истина, если аргумент **A** – N-ый аргумент терма **T**, например:

```
? – arg(2, f(x, a (1), b), A).
A = a(1)
```

Предикат **T = ..L** будет истинным, если **L**– список, первым элементом которого является главный функтор терма **T**, за которым следуют аргументы терма **T**, например:

```
? – f(a, b) = ..L.
L = [f, a, b]
? – T = ..[быстро, автомобиль].
T = быстро(автомобиль).
```

Приведем более содержательные примеры использования рассмотренных предикатов при создании и декомпозиции термов.

Пример 1. Определим предикат **ввод_атома**, выполняющий чтение символов из входного потока и формирующий из них атом. При этом воспользуемся предикатом **name**. Определяемый предикат будет считывать символы входного потока, пока не встретится символ начала новой строки. Аргументом предиката **ввод_атома(X)** является переменная, которая конкретизируется последовательностью символов, вводимых пользователем.

```
ввод_атома(X):–                 % ввести символ
get0(C),                        % если С- символ перехода
(C=13;                          % на новую строку, то
                                % закончить работу

ввод_символов(C, Список)),      % ввести символы в список
name(X, Список)),!.             % преобразовать список в атом
```

```
% ввод символов в список, пока не будет нажата клавиша Enter

        ввод_символов(13, [ ])):–!.
        ввод_символов(C, [C| T]):–
        get0(C1), ввод_символов(C1, T),!.
```

Здесь дополнительно определен предикат **ввод_символов**. Данный предикат определен рекурсивно. На каждом шаге рекурсии очередной символ из входного потока вставляется в начало формируемого списка. Список формируется в обратном порядке, т. е. его заполнение начинается с обнаружения ситуации нажатия клавиши Enter (символ с ASCII кодом 13). После этого в пустой список добавляются все символы, которые были введены ранее. В частности, при вызове предиката **ввод_атома(X)** переменная **X** получит следующее значение:

```
? – ввод_атома(X).
    Иванов_Петр        % ввод  с клавиатуры
    X = 'Иванов_Петр'
```

Пример 2. Сформируем программно с помощью предиката **functor** факт **data** и запомним его в переменной **D**.

```
?–functor(D, data, 3),
    arg(1, D, 28),
    arg(2, D, июль),
    arg(3, D, 2001).
    D = data(28, июль, 2001).
```

Пример 3. Определим предикат **map**, являющийся аналогом функционала **map** языка Лисп. Предикат **map** будет иметь три аргумента. Первый аргумент будет являться функтором, два других аргумента будут представлены списками. Предикат **map** будет обеспечивать применение функтора последовательно к первым элементам указанных списков, ко вторым элементам и т. д.

```
map(_, [ ], [ ]).
map(F, [X1|Xt], [Y1|Yt]): –
        Q=..[F,X1,Y1], call(Q),  %применение  терма F(X1,Y1)
        map(F, Xt, Yt).
```

Пусть

```
square(X, Y): – Y is X*X.
```

Тогда при вводе запроса

```
? – map(square,[1, 2], Y).
```

будет получен ответ

Y = [1, 4].

6.10.3. Предикаты работы с базой данных

В Прологе имеется ряд встроенных предикатов, которые позволяют корректировать базу данных в процессе выполнения программ. К ним относят следующие предикаты: **assert(X), asserta(X), assertz(X), retract(X)**.

Предикат **assert(X)** добавляет в базу данных утверждение **X**. Например:

```
? – assert(столица('Украина', 'Киев')).
    Yes
? – столица('Украина', X).
    X = Киев
```

Предикат **asserta(X)** добавляет утверждение **X** в базу данных и помещает его перед другими утверждениями, соответствующими этому же факту или правилу. Предикат **assertz(X)** определяется аналогично **asserta(X)**, но утверждения помещаются после других утверждений. Предикат **retract(X)** выполняет поиск утверждения **X** в базе данных и удаляет первое найденное утверждение.

Совместное использование рассмотренных предикатов позволяет изменять базу данных. Определим простой предикат, для изменения значения факта, указывающего текущий день месяца:

```
день (9).
изменить_день:-
        день(X), Y is X+1,
        retract(день (X)),
        assert(день (Y)).
```

При каждом вызове предиката **изменить_день** текущая дата будет увеличиваться на единицу:

```
? – изменить_день.
        Yes
? – день(X).
        X = 10.
```

Если предикаты группы **assert** используются для добавления в базу данных правила, то добавляемое правило дополнительно заключается в скобки:

```
? – assert((a(X): – b(X), c(X))).
```

Предикат **assert** позволяет накапливать в базе данных уже вычисленные ответы на вопросы [7]. Например, пусть в программе используется предикат **решить(Задача,Решение).** Тогда можно задать вопрос и добавить полученное решение в базу данных:

```
? – решить(Задача1, Решение),
asserta(решить(Задача1, Решение)).
```

Если теперь задавать вопросы, использующие добавленный факт **решить(Задача1, Решение)**, то ответ будет получен быстрее, так как никаких вычислений выполнять уже не требуется. Пролог будет извлекать в качестве ответов запомненные факты.

Во многих реализациях Пролога имеется встроенный предикат **retractall(X)**, который удаляет из базы данных все утверждения, функтор и арность которых сопоставимы с **X**. Данный предикат может быть определен следующим образом:

```
retractall(Факт): – retract(Факт), fail.
retractall(Голова): – retract((Голова: – Тело)),
                      fail.
retractall(_).
```

Пользователь может вводить новые утверждения в базу данных из файла, используя предикат **consult(Файл)**. Например, запрос

```
? – consult('prog1.ari').
```

считывает содержимое файла "**prog1.ari**". Если при чтении утверждений из файла обнаруживается синтаксическая ошибка, то выдается соответствующее сообщение, но обработка оставшихся утверждений продолжается.

Утверждения, считываемые предикатом **consult(Файл)** добавляются в конец существующей базы данных. Допускается замена существующих утверждений базы данных новыми утверждениями из файла. В этом случае используется предикат **reconsult(Файл)**.

После модификации базы данных с помощью рассмотренных выше предикатов, пользователь имеет возможность запомнить текущее состояние системы в файле. Для этого применяется предикат **save(Файл)**. Восстановить запомненное состояние системы можно, воспользовавшись предикатом **restore(Файл)**.

Для просмотра утверждений, входящих в базу данных, можно воспользоваться предикатами:

listing – выводит утверждения, содержащиеся в базе данных, в выходной поток;

listing(X) – печатаются утверждения, сопоставимые с термом **X**.

6.11. Решение задач на языке Пролог

6.11.1. Создание и обработка бинарных деревьев

Одной из широко используемых структур данных, применяемых в программировании, являются бинарные деревья. Бинарное дерево – это структура данных, состоящая из левого бинарного поддерева, корня и правого бинарного поддерева. Элементы, расположенные в вершинах бинарного дерева, упорядочены в соответствии с некоторым признаком. При этом любой элемент левого поддерева всегда меньше корневого элемента, а любой элемент правого поддерева всегда больше корня. На рисунке 6.9 изображено бинарное дерево, вершины которого представлены буквами. Дерево упорядочено в соответствии с алфавитом.

Будем представлять бинарное дерево термом вида **дерево(Лд,К,Пд)**, где **Лд** – левое поддерево, **К** – корень, а **Пд** – правое поддерево. Пустое бинарное дерево будем обозначать атомом **nil**. Тогда бинарное дерево, изображенное на рисунке 6.9, можно описать следующим образом:

```
дерево(дерево(дерево(nil, а, nil), д, дерево(nil, е, nil)), ж,
       дерево(nil, к, nil))
```

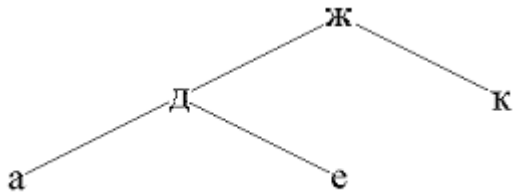


Рисунок 6.9 – Бинарное дерево

Применение бинарных деревьев существенно повышает эффективность поиска элементов данных по сравнению со списками.

Определим отношение **принадлежит(Х,Д)**, которое позволяет установить, входит ли элемент **Х** в бинарное дерево **Д**. Определение отношения будет основано на следующих правилах:

а) если **Х** совпадает с корнем дерева **Д**, то **Х** входит в дерево;

б) если **Х** меньше, чем корень дерева **Д**, то выполнить поиск **Х** в левом поддереве;

в) если **Х** больше, чем корень дерева **Д**, то выполнить поиск **Х** в правом поддереве.

Запишем соответствующие утверждения на Прологе:

```
принадлежит(Х, дерево(Лд, Х, Пд)).          % определение а)
```

```
принадлежит(X, дерево(Лд, К, Пд)): –          % определение б)
        больше(К, X),
        принадлежит(X, Лд).
принадлежит(X, дерево(Лд, К, Пд)): –          % определение в)
        больше(X, К)
        принадлежит(X, Лд).
```

Предикат **больше(X,Y)** обеспечивает сравнение элементов, запоминаемых в узлах дерева, и требует определения при решении задачи поиска в конкретном дереве. Если элементы, находящиеся в вершинах дерева, — это числа, то данное отношение запишется в виде **X>Y**.

Важно отметить, что эффективность поиска в бинарном дереве зависит от сбалансированности дерева. Бинарное дерево считается приближенно сбалансированным, если для каждой вершины дерева два ее поддерева содержат примерно равное число элементов. Так, если сбалансированное бинарное дерево содержит n=2048 чисел, то при ответе на вопрос

```
? – принадлежит(27, Д).
```

Пролог выполнит не более $\log_2 n+1$ сравнений, т.е. 12. При поиске в линейном списке максимальное число сравнений будет 2048. Поэтому важно не допускать вырождения бинарного дерева в список.

Задача создания бинарного дерева **Д1** путем добавления элемента **X** к другому бинарному дереву **Д** может быть решена с помощью предиката **добавить(Д, X, Д1)**. Правила добавления элементов в дерево следующие:

а) добавление **X** к **nil** дает **дерево(nil, X, nil)**;

б) добавление **X** в дерево **Д** с корнем **X** дает **Д1=Д**;

в) если корень дерева больше **X**, то **X** добавляется в левое поддерево, если **X** больше корня, то **X** добавляется в правое поддерево.

Соответствующие утверждения запишутся на Прологе следующим образом:

```
добавить(nil, X, дерево(nil, X, nil).
добавить(дерево(Лд, X, Пд), X, дерево(Лд, X, Пд)).
добавить(дерево(Лд, К, Пд), X, дерево(Лд1, К, Пд): –
        больше(К, X), добавить(Лд, X, Лд1).
добавить(дерево(Лд, К, Пд), X, дерево(Лд, К, Пд1): –
        больше(X, К), добавить(Пд, X, Пд1).
```

При вводе вопроса

```
? – добавить(nil, д, Д1), добавить(Д1, в, Д2).
```

будут получены ответы:

```
Д1=дерево(nil, д, nil).
Д2=дерево(дерево(nil, в, nil), д, nil).
```

Отношение **добавить(Д, Х, Д1)** можно использовать для преобразования списка в бинарное дерево:

```
список_в_дерево([ ], nil).
список_в_дерево([Н|Т], Д1): –
        список_в_дерево(Т, Д),
        добавить(Д, Н, Д1).
```

Рассмотренное отношение не гарантирует построения сбалансированного дерева. При его использовании необходимо следить за тем, чтобы список был неупорядоченным, иначе построенное дерево может оказаться вырожденным. Отношения, обеспечивающие построение приближенно сбалансированных деревьев, рассматриваются в [7].

Иногда требуется выполнить преобразование дерева в список. В этом случае можно воспользоваться определением:

```
дерево_в_список(nil, [ ]).
дерево_в_список(дерево(Лд, К, Пд), С): –
        дерево_в_список(Лд, С1),
        дерево_в_список(Пд, С2),
        слияние(С1, [К|С2], С).
```

Здесь предикат **слияние(С1,С2,С3)** выполняет слияние списков **С1, С2** и формирует список **С3**. Данный предикат соответствует предикату **append**, рассмотренному в §4.1.8.

Помимо поиска и добавления, при работе с бинарными деревьями необходимо уметь удалять вершины из бинарного дерева. Если требуется удалить вершину на уровне листа бинарного дерева, то для этого можно воспользоваться отношением **добавить**:

```
удалить_лист(Д1,Х,Д): – добавить(Д, Х, Д1).
```

Иными словами, удаление листа **Х** из дерева **Д1** равносильно поиску такого дерева Д, добавление к которому элемента **Х** дает дерево **Д1**.

Если же требуется выполнить удаление элемента на уровне корня дерева, то задача становится сложнее. Основная проблема заключается в том, что удаление корневой вершины приводит к образованию “дыры”. В этом случае левое и правое поддеревья, находящиеся ниже удаленного элемента, могут быть потеряны. Чтобы избежать этого, образовавшуюся дыру необходимо заполнить, сохранив упорядоченность дерева. “Дыру” заполняют либо самым “левым” элементом из правого поддерева, либо самым “правым” элементом из левого поддерева. Ниже приведена программа, реализующая указанные требования:

```
удалить(дерево(nil, Х, Пд), Х, Пд).        %  Лд отсутствует
```

```
удалить(дерево(Лд, X, nil), X, Лд).          %  Пд отсутствует

% заполнение "дыры" самым левым элементом Пд
удалить(дерево(Лд, X, Пд), X, дерево(Лд, У, Пд1)): –
                                    удалить_левый(Пд, У, Пд1).

% поиск удаляемого корня
удалить(дерево(Лд, К, Пд), X, дерево(Лд1, К, Пд)): –
                    больше(К, X), удалить(Лд, X, ЛД1).
удалить(дерево(Лд, К, Пд), X, дерево(Лд, К, Пд1): –
                    больше(X, К), удалить(Пд, X, Пд1).

%удаление самого левого элемента из правого поддерева
  удалить_левый(дерево(nil, У, Пд), У, Пд).
  удалить_левый(дерево(Лд, К, Пд), У, дерево(Лд1, К, Пд)): –
                удалить_левый(Лд, У, Лд1).
```

6.11.2. Поиск решений в пространстве состояний

В главе 2 рассмотрено представление задач в пространстве состояний и методы их решения. Пространство состояний – это граф, вершины которого соответствуют состояниям решаемой задачи, а дуги – возможным переходам состояний. При решении конкретной задачи задаются стартовая и целевая вершины. Решению задачи соответствует путь на графе. Рассмотрим программирование основных методов поиска решений в пространстве состояний на языке Пролог.

6.11.2.1. Поиск в глубину. Простейший способ организации поиска в глубину – это использование механизма поиска решений, заложенного в пролог-системе. Рассмотрим граф состояний, изображенный на рисунке 6.10.

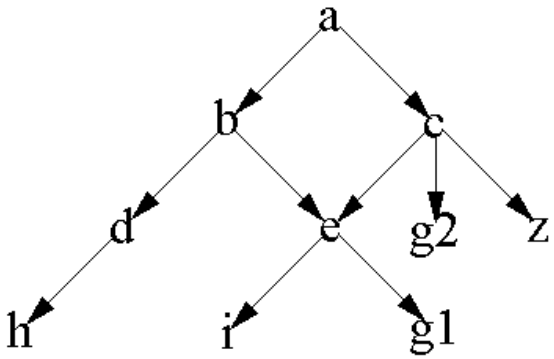


Рисунок 6.10 – Граф состояний

Здесь “a” – стартовая вершина, а “g1” – целевая. Данный граф можно описать в виде следующего набора утверждений:

```
b: – a.        % "b" достижимо, если достигнуто "a"
c: – a.        % "c" достижимо, если достигнуто "a"
d: – b.
. . .
g1: – e.       % "g1" достижимо, если достигнуто "e"
a.             % начальное состояние
```

Для того чтобы узнать, имеет ли рассматриваемая задача решение, необходимо просто задать вопрос:

```
? – g1.
```

Получив такой вопрос, пролог-система выполнит поиск в глубину на графе, изображенном на рисунке 6.10. При этом поиск будет выполняться обратным методом – от цели к известным фактам. После обхода всех необходимых вершин графа, будет сформирован ответ “да”.

Для того чтобы осуществить прямой поиск (от фактов к цели), необходимо описать граф состояния (рисунок 6.10) в виде следующего набора правил:

```
a: – b.
b: – d.
b: – e.
. . .
e: – g1.
g1.                % цель
```

В этом случае процесс поиска решения инициируется вопросом:

```
? – a.
```

Преимущество рассмотренного способа программирования поиска в глубину заключается в его простоте. К недостаткам следует отнести следующее:

– программа формирует ответ “да” или “нет”, но не возвращает при этом решающий путь, соответствующий ходу решения;
– если на графе состояний имеются циклы, то пролог- система может войти в бесконечный цикл, обусловленный косвенными рекурсивными вызовами.

Чтобы исключить указанные недостатки изменим представление графа состояний. Опишем его не с помощью правил, а с помощью фактов вида:

```
после(X, Y).
```

Такой факт означает, что состояние задачи **Y** непосредственно следует из состояния **X**. Тогда граф состояний, изображенный на рисунке 6.10, можно представить следующим набором фактов:

```
после(a, b).
после(a, c).
после(b, d).
после(b, e).
   …
после(e, g1).
```

Чтобы выяснить, имеется ли путь из стартовой вершины в целевую вершину, воспользуемся предикатом:

```
путь(X, X).
путь(X, Y): – после(X, Z),путь(Z, Y).
```

Иными словами, путь из вершины **X** в вершину **Y** существует, если имеется вершина **Z**, следующая после **X**, и известен путь из вершины **Z** в вершину **Y**. Факт **путь(X, X)** обеспечивает выход из рекурсии. Для того чтобы выяснить имеется ли путь, ведущий из вершины "**a**" в целевую вершину "**g1**", необходимо задать вопрос:

```
? – путь(a, g1).
```

Данный вариант программы также не возвращает в качестве результата решающий путь, а возвращает ответ "да" или "нет". Чтобы запомнить решающий путь введем в рассмотрение список закрытых вершин **ЗКР**, т.е. вершин, которые уже пройдены. Определим предикат, выполняющий поиск в глубину:

```
в_глубину(Старт, Цель, Реш_путь): –
        путь(Старт, Цель, [Старт], Путь),
        реверс(Путь, Реш_путь).
```

Здесь **предикат в_глубину** переопределяется через предикат **путь**. Третьим аргументом предиката **путь** является список **ЗКР**. При первом вызове в него помещается стартовая вершина. Четвертый аргумент предиката **путь** – список вершин, расположенных на обратном пути из целевой вершины в стартовую вершину. Поэтому значение переменной **Реш_путь**, представляющей решающий путь, получается путем инвертирования списка **Путь**.

Предикат **путь** определяется рекурсивно. При этом в списке **ЗКР** последовательно накапливаются вершины, лежащие на искомом пути:

```
путь(X, X, ЗКР, Путь): – Путь=ЗКР.      % выход из рекурсии
путь(X, Y, ЗКР, Путь): – после(X, Z), путь(Z, Y, [Z|ЗКР], Путь).
```

При каждом рекурсивном вызове очередная вершина **Z**, следующая после **X**, вносится в начало списка **ЗКР**. Если вершина **Z** совпадает с целевой вер-

шиной **Y**, то рекурсивные вызовы прерываются и переменная **Путь** получает значение списка **ЗКР**. Если вершина **Z** является тупиковой, то цель **после(Z, _)** терпит неудачу, и пролог-система выполняет возврат к предыдущей вершине, в которой имеется альтернативный путь для продолжения поиска. В ходе возврата все подстановки, которые были сделаны в список **ЗКР** на тупиковом направлении, стираются. Чтобы проиллюстрировать это, внесем изменения в определение предиката **путь**:

```
путь(X, Y, ЗКР, Путь): – write(ЗКР), nl, после(X, Z), …
```

Тогда при запросе

```
? – в_глубину(a, g1, P).
```

для графа, изображенного на рисунке 6.10, получим следующие ответы:

```
[a]
[b, a]
[d, b, a]
[h, d, b, a]
[e, b, a]
[i, e, b, a]
P=[a, b, e, g1]
```

В этом случае пролог-система отображает промежуточные состояния списка **ЗКР**. После захода в вершину "**h**" и выяснения того, что цель **после(h,_)** недостижима, возникает возврат к альтернативному варианту продолжения списка. Подстановки, которые были сделаны в список **ЗКР** на пути **b-d-h,** стираются, и список **ЗКР** получает новое значение **[e, b, a]**.

Рассмотренный вариант программы поиска в глубину имеет существенный недостаток: если на графе состояний имеются циклы, то программа может зациклиться. Чтобы устранить этот недостаток, необходимо добавлять очередную вершину **Z** в список **ЗКР** только в том случае, если ее там нет. Рекурсивное правило для предиката **путь** перепишется следующим образом:

```
путь(X, Y, ЗКР, Путь): – после(X, Z), not элемент(Z, ЗКР),
                          путь(Z, Y, [Z|ЗКР], Путь).
```

Рассмотрим задачу манипулирования кубиками (рисунок 6.11)[1)]. Имеются три стола "**p**", "**q**", "**r**" и три кубика "**a**", "**b**", "**c**". Дано начальное положение кубиков, изображенное на рисунке 6.11 слева. Необходимо найти последовательность команд по перемещению кубиков, обеспечивающую их конечное расположение, изображенное на рисунке 6.11 справа.

[1)] Данную программу разработал Дж. Криз (J. Kriz)

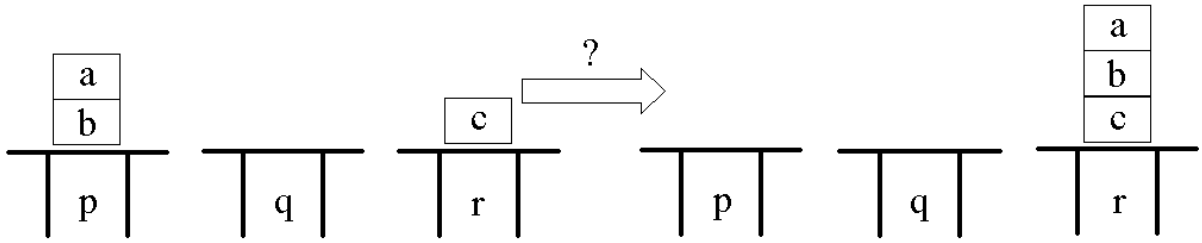


Рисунок 6.11 – Задача манипулирования кубиками

Данная задача легко решается в пространстве состояний. При этом участок графа состояний, соответствующий переходу из состояния **S** в состояние **S1**, будем представлять в виде предиката

t(S, S1, переместить(Что, Откуда, Куда)),

где **переместить(Что, Откуда, Куда)** – терм, соответствующий команде перемещения кубика. Например, возможный переход из начального состояния, изображенного на рисунке 6.11, можно описать следующим образом:

```
t(s(p([a, b]), q([ ]), r([c])),
  s(p[b]), q([a]), r([c])),
  переместить(a, откуда(p), куда(q)).
```

Начальное состояние представлено предикатом **s(p([a, b]), q([ ]), r([c])).** Аргументы предиката – термы, отражающие расположение кубиков на столах. Так, **p([a, b])** означает, что на столе “**p**” находятся кубики “**a**” и “**b**”, причем кубик “**a**” находится сверху кубика “**b**“. Новое состояние **S1**, в которое необходимо перейти, описывается предикатом **s(p([b]), q([a]), r([c]))**. Переход в это состояние обеспечивается командой **переместить(a, откуда(p), куда(q))**, то есть необходимо переставить кубик “**a**” со стола “**p**” на стол “**q**”.

Введем следующие правила, определяющие возможные перемещения кубиков:

```
t(s(p[P1|Ps], q(Q), r(R)),
        s(p(Ps), q(Q1), r(R1)),
        переместить(P1, откуда(p), куда(D))): –
        (D=q, Q1=[P1|Q], R1=R);
        (D=r, Q1=Q, R1=[P1|R]).
t(s(P, q([Q1|Qs]), r(R)),
        s(P, q(Qs), r(R1)),
        переместить(Q1, откуда(q), куда(r))): –
        R1=[Q1|R].
```

Первое правило позволяет перемещать кубик **P1** со стола “**p**“ на стол “**q**” (**D=q**), либо на стол “**r**” (**D=r**). При этом кубик **P1** устанавливается сверху

кубиков, которые находились либо на столе “**q**” (**Q1=[P1|Q]**), либо на столе “**r**” (**R1=[P1|R]**). Второе правило обеспечивает перемещение кубиков со стола “**q**” на стол “**r**”.

Решение задачи можно найти с помощью следующего предиката поиска в глубину:

```
в_глубину(Старт, Решение): –
        путь(Старт, [Старт], Решение).
путь(S, ЗКР, [ ]): – цель(S).
путь(S, ЗКР, [T|Ts]): –
        t(S, S1, T), not элемент(S1, ЗКР),
        путь(S1, [S1|ЗКР], Ts).
```

Здесь в переменной **Решение** будет накапливаться последовательность команд по перемещению кубиков. Список **ЗКР**, как и прежде, служит для исключения повторного прохождения через одни и те же состояния. Предикат **цель(S)** определяет целевое состояние. Для рисунка 6.11 целевое состояние задается фактом:

```
цель(s(p([ ]), q([ ]), r([a, b, c]))).
```

Зададим вопрос

```
? – в_глубину(s(p([a, b]), q([ ]), r([c])), Решение).
```

Тогда пролог-система найдет следующее решение

```
Решение=[переместить(a, откуда(p), куда(q)),
        переместить(b, откуда(p), куда(r)),
        переместить(a, откуда(q), куда(r))].
```

Рассмотренный вариант программы будет успешно осуществлять поиск в конечных пространствах состояний. Если пространство состояний задачи бесконечно, то направление поиска вдоль одной бесконечной ветви графа может оказаться ошибочным. Чтобы исключить рассмотренную ситуацию, в программу вводят ограничение глубины поиска. Разрешается вести поиск на глубине не большей, чем значение переменной **Глубина**:

```
путь(S, ЗКР, [T|Ts], Глубина): –
        Глубина>0, Глубина1 is Глубина-1,
        t(S, S1, T),
        путь(S1, [S1|ЗКР], Ts, Глубина1).
```

При очередном выборе ветви поиска, значение переменной **Глубина** уменьшается на единицу. В этом случае поиск идет сначала в глубину, а затем в ширину.

6.11.2.2. Поиск в ширину. Программирование метода поиска в ширину сложнее, чем поиска в глубину. Обусловлено это тем, что необходимо запоминать несколько возможных направлений поиска. Напишем программу поиска в ширину в соответствии с алгоритмом, рассмотренным в главе 2.

Введем в рассмотрение предикат **в_ширину(ОТК,ЗКР,Решение)**, где **ОТК** – список открытых вершин; **ЗКР** – список закрытых вершин; **Решение** – список обследованных вершин, соответствующий списку **ЗКР** в момент достижения целевой вершины. Указанные списки состоят из подсписков, каждый из которых содержит два элемента. Первый элемент подсписка – это имя некоторой вершины графа состояний, а второй – имя ее родительной вершины. Список **Решение** формируется таким образом, что его первый подсписок в качестве первого элемента содержит целевую вершину. Просмотрев список **Решение** от целевой вершины до стартовой вершины, можно восстановить решающий путь. Будем использовать для этого предикат **решающий_путь(Решение, Цель, Старт, Temp, Реш_путь)**, где **Цель** – целевая вершина; **Старт** – начальная вершина; Temp – вспомогательная переменная, предназначенная для накопления результатов поиска; **Реш_путь** – список, фиксирующий найденный путь от стартовой до целевой вершины.

При определении предиката **в_ширину(ОТК,ЗКР,Решение)** для раскрытия очередной вершины и генерации возможных продолжений пути будем использовать встроенный предикат **bagof**. Цель

```
bagof(X, C, L)
```

формирует список **L** всех объектов **X**, удовлетворяющих условию **C**. Например,

```
после(a, b).
после(a, c).
после(b, d).
после(d, h).
после(a, e).
после(e, k).
? – bagof([Z | a ], после(a, Z),L).
L = [[b | a], [c | a], [e | a]]
```

В этом случае в списке **L** накапливаются подсписки, содержащие дочерние вершины для вершины ‘**a**’.

Используя рекурсию, определим предикат **в_ширину( )**:

```
в_ширину([[O1|O2]|Ост_ОТК ], ЗКР,Решение):–
        цель(O1),!, Решение = [[O1|O2]|ЗКР].
```

```
в_ширину([[O1|O2]|Ост_ОТК], ЗКР, Решение):–
    bagof([Z|O1], (после(O1,Z), not элемент1(Z, ЗКР),
          not элемент1(Z,Ост_ОТК), L),
    слияние(Ост_ОТК, L, Нов_ОТК), !,
    в_ширину(Нов_ОТК, [[O1|O2]|ЗКР], Решение);
    в_ширину(Ост_ОТК, ЗКР, Решение).
```

В начале поиска список **ЗКР** пустой, а список ОТК содержит подсписок, включающий начальную вершину. На каждом шаге рекурсии выделяется первый подсписок **[O1|O2]** списка **ОТК**. Выясняется, является ли вершина **O1** целевой (**цель(O1)**). Если **O1** целевая вершина, то подсписок **[O1|O2]** добавляется в начало списка **ЗКР**, и рекурсия прерывается. Переменная **Решение** получает значение списка **[[O1|O2] | ЗКР]**, элементами которого являются подсписки, представляющие пройденные отрезки графа состояний.

Если **O1** не целевая вершина, то находятся все продолжения пути после вершины **O1**. Для этого применяется предикат **bagof**, который обеспечивает поиск всех отрезков графа состояний **[Z|O1]** таких, что вершина **Z** не входит в список **ЗКР** и в остаток списка **ОТК** (**Ост_ОТК**). Все найденные дочерние вершины накапливаются в списке **L** в виде подсписков и помещаются в конец списка **Ост_ОТК**, образуя список **Нов_ОТК**. Затем обработанный первый подсписок **[O1|O2]** списка **ОТК** перемещается в начало списка **ЗКР**, и предикат **в_ширину** вызывается при новых значениях аргументов.

Если вершина **O1** является тупиковой, то предикат **после(O1,Z)** завершается неудачей. В этом случае никакие новые вершины не добавляются в конец списка **ОТК**, и обеспечивается альтернативный вызов предиката **в_ширину**:

```
в_ширину(Ост_ОТК, ЗКР, Решение).
```

В рассмотренном определении предиката **в_ширину** используется предикат **элемент1(Вершина,Список)**. Данный предикат позволяет установить, входит ли **Вершина** в один из подсписков, образующих **Список**:

```
элемент1(H, [[H|T] | _] ).
элемент1(T, [[H|T] | _] ).
элемент1(H, [[Y|Z] |Ост] ):– элемент1(H,Ост).
```

Для выдачи решающего пути список **Решение** необходимо обработать с помощью следующей процедуры:

```
решающий_путь( _, X, X, Temp,Реш_путь) :–Реш_путь=Temp.
решающий_путь(Решение, Цель, Старт, Temp, Реш_путь):–
        найти(Цель, Решение, [[H1|T1] | Ост1] ),
        слияние(Temp, [[H1|T1]], Temp1),
        найти(T1, Ост1, [[H2|T2] | Ост2]),
```

```
слияние(Temp1, [[H2 | T2]], Temp2),
решающий_путь(Ост2, T2, Старт, Temp2, Реш_путь).
```

Предикат **найти(Элемент,Список1,Список2)** выполняет поиск заданного элемента в списке **Список1**, состоящем из подсписков. Результатом поиска является список **Список2**, представляющий хвостовую часть списка **Список1**, в первый подсписок которого входит искомый элемент:

```
найти(H, [[H|T]|Ост], [[H|T]|Ост] ).
найти(H, [[Y|T]|Ост], X):–
        найти(H, Ост, X).
```

Чтобы лучше уяснить работу описанных выше предикатов, введем вопрос:

```
? – в_ширину( [[a | a]], [ ], Решение),
решающий_путь(Решение, g1, a, [ ], Реш_путь).
```

Начальное значение списка **ОТК = [[a | a]]**. Список **ЗКР** пустой. Цель заключается в том, чтобы найти решающий путь из начальной вершины **"a"** в конечную вершину “**g1**” (рисунок 6.10). В результате обработки вопроса получим значения переменных:

```
Решение = [[g1 | e], [e | b], [d | b], [c | a], [b | a], [a | a]]
Реш_путь = [[g1 | e], [e | b], [b | a], [a | a]]
```

Значение переменной **Решение** соответствует состоянию списка **ЗКР** в момент завершения поиска. Головным элементом каждого из подсписков является вершина, которая раскрывалась в процессе поиска (за исключением целевой вершины). Хвостовой элемент каждого из подсписков есть родительная вершина для головного элемента. Перемещаясь по указанным подспискам в направлении от целевой вершины до начальной, используя ссылки на родительские вершины, восстанавливаем значения переменной **Реш_путь**.

6.11.2.3. Эвристический поиск. В эвристических алгоритмах список открытых вершин **ОТК** сортируется с учетом возрастания оценочной функции. Рассмотрим **А**– алгоритм с оценочной функцией вида (2.1).

Пусть граф состояний описывается с помощью фактов, задаваемых предикатом **после(P,D,C,H)**, где **P** – родительская вершина; **D** – дочерняя вершина; **C** – стоимость участка пути между вершинами **P** и **D**; **H** – оценка стоимости кратчайшего пути из вершины **D** в целевую вершину. Тогда граф состояний, изображенный на рисунке 2.9, можно описать с помощью следующего набора фактов:

```
после(s, d, 1, 10).
после(s, c, 3, 7).
```

```
после(s, b, 5, 4).
после(s, a, 7, 1).
после(a, g, 10, 0).
после(b, a, 1, 1).
после(c, b, 1, 4,).
после(c, a, 2, 1).
после(d, c, 1, 7).
```

Определим на Прологе предикат **a_поиск**, соответствующий процедуре эвристического поиска **A_search** (см. § 2.2.2.3):

a_поиск(ОТК, ЗКР, Решение),

где **ОТК**, **ЗКР** – списки открытых и закрытых вершин; **Решение** – список, соответствующий искомому пути из начальной вершины в целевую вершину. Каждая из вершин графа состояний представляется в этих списках в виде подсписков **[D, P, F, G]**, где **D** – вершина на графе состояний; **P** – ссылка на родительную вершину; **F** – значение оценочной функции (2.1); **G** – оценка стоимости пути от начальной вершины до вершины **D**.

Определение предиката **a_поиск** близко к определению предиката **в_ширину**. Отличие заключается в том, что в случае эвристического поиска вершины из списка **ЗКР** могут перемещаться обратно в список **ОТК**. Происходит это из-за того, что в ходе поиска может быть обнаружен путь меньшей стоимости к вершине, которая уже была раскрыта. Это приводит к необходимости удаления такой вершины из списка **ЗКР** и вставки ее в список **ОТК** (с учетом значения оценочной функции). Кроме этого, если в процессе раскрытия очередной вершины обнаруживается, что ее дочерние вершины уже имеются в списке **ОТК**, но с бóльшими значениями оценочной функции, то такие вершины также удаляются из списка **ОТК**, и вместо них в список **ОТК** вставляются вершины с меньшими значениями оценочной функции. Приведем определение предиката **a_поиск**:

```
a_поиск([D, P, F, G] | Ост_ОТК], ЗКР, Решение):–
        цель(D), !, Решение = [[D,P,F,G] | ЗКР].

a_поиск([[D,P,F,G] | Ост_ОТК], ЗКР, Решение):–
        bagof([Z,D,FZ,GZ], следующая(D,Z,FZ,GZ,G]), DB),! ,
        вставить([D,P,F,G], ЗКР, Н_ЗКР),
        поместить(Ост_ОТК,DB,Н_ЗКР,Нов_ОТК, Нов_ЗКР),
        a_поиск(Нов_ОТК, Нов_ЗКР, Решение);
        a_поиск(Ост_ОТК, ЗКР, Решение).
```

В этом определении с помощью предиката **bagof** находятся все дочерние вершины для вершины **D**, находящейся на первом месте в списке

ОТК и имеющей наименьшее значение оценочной функции **F**. Для поиска дочерних вершин используется условие:

```
следующая(D,Z,FZ,GZ,G): –
        после(D,Z,C,H), GZ is G+C, FZ is GZ+H.
```

Здесь переменные **FZ** и **GZ** соответствуют оценочным функциям $\hat{f}(n)$ и $\hat{g}(n)$ для вершины **Z**. Переменная **G** соответствует оценке $\hat{g}(n)$ для вершины **D**. При вызове предиката переменные **D** и **G** конкретизированы, а переменные **Z, FZ, GZ** вычисляются, формируя подсписки **[Z,D,FZ,GZ]**, которые накапливаются в списке дочерних вершин **DB**. Если вершина **D** может быть раскрыта (т. е. имеет дочерние вершины), то вызов **bagof** завершается успешно, и вершина **D** перемещается в список **ЗКР**. Это выполняется с помощью вызова:

```
вставить( [D,P,F,G], ЗКР, Н_ЗКР).
```

Дочерние вершины для вершины **D**, размещаемые в списке **DB**, вносятся в остаток списка **ОТК** (**Ост_ОТК**), если они отсутствуют в списках **Ост_ОТК** и **Н_ЗКР**. Если дочерняя вершина уже имеется в списке **Ост_ОТК**, а значение её оценочной функции выше, чем у вершины из списка **DB**, то она заменяется на вершину из списка **DB**. Кроме того, если дочерняя вершина имеется в списке **Н_ЗКР**, то в список **Ост_ОТК** вносится соответствующая вершина из списка **DB**. Все указанные действия выполняются с помощью вызова предиката

```
поместить(Ост_ОТК, DB, Н_ЗКР,Нов_ОТК, Нов_ЗКР).
```

В результате выполнения предиката формируются новые списки **Нов_ОТК** и **Нов_ЗКР**. После этого предикат **а_поиск** вызывается рекурсивно для новых значений списков **ОТК** и **ЗКР**. Если у вершины D не имеется дочерних вершин, то вызов **bagof** завершается неудачей, и предикат **а_поиск** вызывается рекурсивно при старых значениях списков **ОТК** и **ЗКР**. Рекурсия прерывается, когда первая вершина в списке **ОТК** будет целевой. При этом она вносится в начало списка **ЗКР**, формируя значение переменной **Решение**.

Предикат **поместить** определяется следующим образом:

```
поместить(Ост_ОТК, [ ], ЗКР, Ост_ОТК, ЗКР).
поместить(Ост_ОТК, [X | Ост_DB], ЗКР, Нов_ОТК, Нов_ЗКР): –
   (not элемент(X, Ост_ОТК), not элемент(X, ЗКР), !,
   вставить(X, Ост_ОТК, Н_ОТК), Н_ЗКР = ЗКР;
   удалить(X,Ост_ОТК,Нов),вставить(X,Нов,Н_ОТК), Н_ЗКР = ЗКР;
   удалить(X,ЗКР,Н_ЗКР), вставить(X, Ост_ОТК, Н_ОТК)),
   поместить(Н_ОТК,Ост_DB, Н_ЗКР, Нов_ОТК, Нов_ЗКР).
```

Предикат **удалить(X,Список,Нов_список)** выполняет удаление подсписка **X**, представляющего вершину графа состояний, из списка **Список**. Результатом является новый список **Нов_Список**. Удаление выполняется, если оценочная функция для удаляемого элемента X меньше значения оценочной функции элемента из списка **Список**.

```
удалить([ZD,PD,FZD,GZD], [[Z, _, FZ, _] | T], T):– ZD==Z, FZD=<FZ,!.
удалить(X, [Y|T], [Y|T1]):– удалить(X,T,T1).
```

Предикат **вставить(X,Список,Нов_Список)** осуществляет включение подсписка **X**, представляющего вершину графа состояний, в **Список**, формируя новый список **Нов_Список**. При этом включение выполняется с учетом значения оценочной функции. Результирующий список **Нов_Список** упорядочен по возрастанию оценочной функции:

```
вставить(X,[ ],Нов_Список):– Нов_Список = [X].
вставить([ZD,PD,FZD,GZD], [ [Z,P,FZ,GZ] | Ост], Нов_Список): –
        FZD <FZ, !,
        Нов_Список = [[ZD,PD,FZD,GZD], [Z,P,FZ,GZ] | Ост].
вставить(X, [H|T], [H|T1]):– вставить(X,T,T1).
```

Предикат **элемент(X,Список)** в рассматриваемом случае определяется следующим образом:

```
элемент([Z | X], [[Z | _ ] | T] ).
элемент(H, [Y | T] ):– элемент(H, T).
```

Рассмотрим выполнение эвристического поиска на примере графа состояний, изображенного на рисунке 2.9. При этом будем выводить на печать состояния списков **ОТК** и **ЗКР** при каждом вызове предиката **a_поиск**. Введем целевое утверждение

```
? – a_поиск( [[s, s, 14, 0]], [ ], Решение).
```

Здесь список **ОТК** содержит начальную вершину, представленную подсписком **[s, s, 14, 0]**. Список **ЗКР** пустой. В ходе поиска будут получены следующие значения списков **ОТК** и **ЗКР**:

```
ОТК=[[a,s,8,7],[b,s,9,5],[c,s,10,3],[d,s,11,1]],
ЗКР=[[s,s,14,0]],
ОТК=[[b,s,9,5],[c,s,10,3],[d,s,11,1],[g1,a,17,17]],
ЗКР=[[a,s,8,7],[s,s,14,0]],
ОТК=[[a,b,7,6],[c,s,10,3],[d,s,11,1],[g1,a,17,17]],
ЗКР=[[b,s,9,5],[s,s,14,0]],
ОТК=[[c,s,10,3],[d,s,11,1],[g1,a,16,16]],
ЗКР=[[a,b,7,6],[b,s,9,5],[s,s,14,0]],
ОТК=[[a,c,6,5],[b,c,8,4],[d,s,11,1],[g1,a,16,16]],
```

```
ЗКР=[[c,s,10,3],[s,s,14,0]],
ОТК=[[b,c,8,4],[d,s,11,1],[g1,a,15,15]],
ЗКР=[[a,c,6,5],[c,s,10,3],[s,s,14,0]],
ОТК=[[a,b,6,5],[d,s,11,1],[g1,a,15,15]],
ЗКР=[[b,c,8,4],[c,s,10,3],[s,s,14,0]],
ОТК=[[d,s,11,1],[g1,a,15,15]],
ЗКР=[[a,b,6,5],[b,c,8,4],[c,s,10,3],[s,s,14,0]],
ОТК=[[c,d,9,2],[g1,a,15,15]],
ЗКР=[[a,b,6,5],[b,c,8,4],[d,s,11,1],[s,s,14,0]],
ОТК=[[a,c,5,4],[b,c,7,3],[g1,a,15,15]],
ЗКР=[[c,d,9,2],[d,s,11,1],[s,s,14,0]],
ОТК=[[b,c,7,3],[g1,a,14,14]],
ЗКР=[[a,c,5,4],[c,d,9,2],[d,s,11,1],[s,s,14,0]],
ОТК=[[a,b,5,4],[g1,a,14,14]],
ЗКР=[[b,c,7,3],[c,d,9,2],[d,s,11,1],[s,s,14,0]],
ОТК=[[g1,a,14,14]],
ЗКР=[[a,b,5,4],[b,c,7,3],[c,d,9,2],[d,s,11,1],[s,s,14,0]].

Решение=[[g1,a,14,14],[a,b,5,4],[b,c,7,3],[c,d,9,2], [d,s,11,1],[s,s,14,0]].
```

6.11.3. Поиск решений в И/ИЛИ графах

Поиск решений в И/ИЛИ графах заключается в нахождении подграфа решения (см. § 2.3). В процессе поиска строятся потенциальные подграфы решений, которые определенным образом представляются в памяти ЭВМ. Воспользуемся представлением, предложенным в [7]. Введем в рассмотрение два инфиксных оператора:

```
: – op(600, xfx, --->).
: – op(500, xfx, : ).
```

Тогда И/ИЛИ граф, изображенный на рисунке 2.14, можно задать с помощью фактов:

```
s ---> или : [a, b].
a ---> и : [c, d].
c ---> или : [f, g].
f ---> и : [t1, t2].
g ---> или :[t3, h].
b ---> и : [d, i].
d ---> или : [g, h].
h ---> или : [ i ].
i ---> или : [ l ].
```

Здесь первый факт указывает, что задача “**s**” может быть решена, если решена или подзадача “**a**”, или подзадача “**b**” . Второй факт фиксирует,

что подзадача "**a**" разрешима, если решены подзадачи "**c**" и "**d**". Терминальные вершины будем задавать с помощью фактов:

```
цель(t1).
цель(t2).
цель(t3).
```

Рассмотрим программирование процедуры поиска в глубину. При этом воспользуемся следующими фактами:

а) если рассматриваемая вершина является терминальной, то решение соответствующей подзадачи известно;

б) если рассматриваемая вершина – это вершина ИЛИ типа, то необходимо решить одну из подзадач;

в) если рассматриваемая вершина – это вершина И типа, то необходимо решить все подзадачи.

Определим предикат **ao_поиск(Верш, РешДер)**, который аналогичен процедуре **Depth_first-AO2**, рассмотренной в § 2.3.1. Воспользуемся встроенным механизмом поиска решений пролог-системы. Дерево вывода, формируемое пролог-системой, является И/ИЛИ деревом. Оно строится методом поиска в глубину. Если какая-либо из подзадач не решена, то пролог-система автоматически возвращается к имеющейся альтернативной подзадаче и продолжает ее решение. Это существенно упрощает программирование предиката **ao_поиск(Верш, РешДер)**, где переменная **Верш** соответствует вершине И/ИЛИ дерева, а переменная **РешДер** – поддереву решения, расположенному ниже этой вершины. Дерево решений будем представлять следующим образом [7]:

а) если **Верш** – терминальная вершина, то дерево решений тривиально (оно представляется вершиной **Верш**);

б) если **Верш** – это вершина ИЛИ типа, то дерево решений будет иметь вид **Верш ---> Поддерево**, где переменная **Поддерево** будет представлять одно из возможных поддеревьев решений для задачи, заданной переменной **Верш**;

в) если **Верш** – это вершина И типа, то дерево решений будет иметь вид **Верш---> и: Поддеревья**, где **Поддеревья** – список поддеревьев решений И-подзадач.

Тогда программа поиска в глубину запишется так [7]:

```
ao_поиск(Верш, Верш): –                  % правило а)
        цель(Верш).

ao_поиск(Верш, Верш---> Дер): –          % правило б)
        Верш---> или : Вершины,
        элемент(Верш1, Вершины), ao_поиск(Верш1, Дер).

ao_поиск(Верш, Верш---> и: Деревья): – % правило в)
```

```
                Верш---> и : вершины,
                ао_поиск_для_всех_вершин(Вершины, Деревья).

ао_поиск_для_всех_вершин([ ], [ ] ).
ао_поиск_для_всех_вершин([Верш|Вершины], [Дер|Деревья] ): –
                ао_поиск(Верш, Дер),
                ао_поиск_для_всех_вершин(Вершины, Деревья).
```

При запросе

```
? – ао_поиск(s, РешДер).
```

получим два решения:

```
РешДер = s---> (a---> и: [c---> (f---> и: [t1, t2] ),
                          d---> (g---> t3) ] );
РешДер = s---> (a---> и: [c---> (g---> t3),
                          d---> (g---> t3) ]).
```

Недостатком рассмотренной программы является возможность возникновения бесконечных циклов. Для их исключения можно ввести ограничения на глубину поиска, воспользовавшись переменной **Глубина**.

6.11.4. Поиск в пространстве версий

Рассмотрим задачу формирования понятий по предъявленным примерам. Одно из решений данной задачи основано на поиске в пространстве версий в соответствии с алгоритмом Митчелла (см §4.3.3). В алгоритме рассматриваются два множества: **G** – множество наиболее общих понятий; **S** – множество наименее общих (наиболее специализированных) понятий. В ходе предъявления примеров область действия понятий, входящих в **G**, сужается, а область действия понятий, входящих в **S**, расширяется. Иными словами, в процессе поиска понятия из **G** специализируются, а понятия из **S** – обобщаются. Поиск заканчивается успешно, когда множества **G** и **S** будут содержать одно и то же понятие. Рассмотрим пример из § 4.3.3. Пусть каждый из объектов представляется списком

[Размер, Цвет, Форма].

Например, круг среднего размера красного цвета представляется в виде списка

[средний, красный, круг],

который можно рассматривать как вектор признаков. В этом случае операция обобщения будет выполняться простой заменой константы на переменную. Так, вектор признаков

[средний, красный, X]

представляет все объекты среднего размера и красного цвета. Этот вектор признаков будет охватывающим для вектора **[средний, красный, круг].**

Определим отношение **охватывает(P,Q)**, устанавливающее идентичность векторов признаков **P** и **Q** или выясняющее, охватывает ли вектор признаков **P** вектор **Q** [77]:

```
охватывает([ ], [ ]).                  % пустые векторы идентичны

охватывает([H1|T1], [H2|T2]): – % переменные H1 и H2
   var(H1), var(H2),                    %охватывают друг друга
   охватывает(T1, T2).

охватывает([H1|T1], [H2|T2]): – %переменная охватывает
   var(H1), atom(H2),                   % константу
   охватывает(T1,T2).

охватывает([H1|T1], [H2|T2]): –  % отношение имеет место,
   atom(H1), atom(H2), H1=H2,   %если константы сопоставимы
   охватывает(T1, T2).
```

Алгоритм Митчелла требует выяснения, будет ли некоторое понятие строго более общим, чем другое. Это можно сделать с помощью правила:

```
более_общее(X, Y): – not охватывает(Y, X),
                      охватывает(X, Y).
```

В ходе поиска необходимо выполнять операции обобщения и специализации. Определим для этого соответствующие предикаты. Предикат обобщения

обобщить(Гипотеза, Пример, Обобщение),

выполняющий поиск значения переменной **Обобщение**, которое минимально обобщает понятие-кандидат, представляемое аргументом **Гипотеза**, и охватывает понятие, заданное аргументом **Пример**. С этой целью рекурсивно сопоставляются элементы векторов признаков **Гипотеза** и **Пример**. Если соответствующие элементы двух векторов признаков сопоставимы, то в результирующий вектор **Обобщение** помещается элемент вектора **Гипотеза**. Если соответствующие элементы векторов не сопоставимы, то в соответствующую позицию вектора Обобщение помещается анонимная переменная [77]:

```
% обобщение понятия-кандидата
обобщить([ ], [ ]).                    % Выход из рекурсии
обобщить([H1|T1], [H2|T2], [H1|T3]): –
        not(H1\=H2),                   % Сопоставить без присвоения
        обобщить(T1, T2, T3).
обобщить([H1|T1], [H2|T2], [_|T3]): –
        H1\=H2,
        обобщить(T1, T2, T3).
```

Минимальную специализацию понятий-кандидатов будем выполнять с помощью предиката

специализировать(Гипотеза, Пример, Подстановки),

где **Гипотеза** и **Пример** – это векторы признаков соответственно специализируемого понятия и примера. **Подстановки** – список признаков понятий, в котором возможные значения того или иного признака представляются в виде подсписка констант. Например,

```
[[маленький, средний, большой],
 [желтый, зеленый, красный],
 [круг, квадрат, треугольник]].
```

В ходе специализации переменные вектора признаков понятия-кандидата заменяются соответствующей константой из списка **Подстановки** [77]:

```
специализировать([H1|_], [H2|_], [L|_]): –
        var(H1),
        элемент(H1, L), H1\=H2.
специализировать([_|T1], [_|T2], [_|T3]): –
        специализировать(T1, T2, T3).
```

Отметим два обстоятельства. Во-первых, данный предикат специализирует понятие-кандидат таким образом, чтобы оно не охватывало **Пример**. Это обеспечивается проверкой условия **H1\=H2**. Во-вторых, если вызвать данный предикат, то будет выполнена подстановка константы только вместо первой переменной. Для того, чтобы обеспечить выполнение всех возможных специализаций, вызов рассмотренного предиката будет выполняться внутри цели **bagof**.

В алгоритме Митчелла необходимо специализировать понятия-кандидаты множества **G**. Определим для этого предикат

специализировать_мн-во(G, Нов_G, Пример, Подстановки).

Данный предикат будет обеспечивать минимальную специализацию понятий, входящих в **G**. В результате выполнения предиката формируется но-

вое множество понятий-кандидатов **Нов_G**. Предикат определяется рекурсивно [77]:

```
специализировать_мн-во([ ], [ ], _, _).

специализировать_мн-во([HG|TG],Нов_G,Пример,Подстановки): –
        охватывает(HG, Пример),
        (bagof(HG,специализировать(HG,Пример,Подстановки),
                Нов_G_Голова);
         Нов_G_Голова=[ ]),
         специализировать_мн-во(TG, Нов_G_Хвост, Пример, Подстановки),
         соединить(Нов_G_Голова, Нов_G_Хвост, Нов_G).

специализировать_мн-во([HG|TG], [HG|Нов_G_Хвост],
                                Пример,Подстановки): –
         not охватывает(HG, Пример),
         специализировать_мн-во(TG, Нов_G_Хвост, Пример,Подстановки).
```

Аналогично определим предикат, обобщающий понятия-кандидаты из множества **S**:

```
обобщить_мн-во(S, Нов_S, Пример).
```

Данный предикат рекурсивно просматривает понятия из множества **S** и выполняет их минимальное обобщение по сравнению с понятием-примером:

```
обобщить_мн-во([ ], [ ], _, _).

обобщить_мн-во([HS|TS], Нов_S, Пример): –
     not охватывает(HS, Пример),
     (bagof(X, обобщить(HS, Пример, X), Нов_S_Голова);
     Нов_S_Голова=[ ]),
     обобщить_мн-во(TS, Нов_S_Хвост, Пример),
     соединить(Нов_S_Голова, Нов_S_Хвост, Нов_S).

обобщить_мн-во([HS|TS], [HS|Нов_S_Хвост], Пример): –
     охватывает(HS, Пример),
     обобщить_мн-во(TS, Нов_S_Хвост, Пример).
```

Для реализации алгоритма Митчелла также определим предикат удаления

```
удалить(X, L, C, Нов_L),
```

который выполняет удаление элемента **X** из списка **L**, если не выполняется условие **C**. В результате формируется список **Нов_L**. Данный предикат легко определяется с помощью встроенного предиката **bagof**:

```
удалить(X, L, C, Нов_L): –
        (bagof(X, (элемент(X, L), not C), Нов_L);
         Нов_L=[ ]).
```

Напишем программу, использующую введенные выше предикаты и выполняющую поиск в пространстве понятий в полном соответствии с алгоритмом Митчелла, рассмотренном в § 4.3.3. Для этого введем предикат

v_поиск(Пример, G, S, Нов_G, Нов_S, Подстановки).

Здесь все переменные имеют введенную ранее интерпретацию. В случае положительного примера происходит исключение всех понятий **X** из **G**, которые не охватывают пример. Затем обобщаются те понятия, входящие в **S**, которые не сопоставимы с предъявляемым примером. В заключение из **S** сначала удаляются те понятия, которые являются более общими, чем любое другое понятие, входящее в **S**, а затем – понятия, которые охватывают некоторые понятия, входящие в **G**. В случае предъявления отрицательного понятия, из множества **S** исключаются все понятия, которые охватывают пример. После этого выполняется специализация понятий, входящих в **G**. Затем из **G** исключаются все понятия менее общие, чем любое другое понятие, входящее в **G**. И наконец, из **G** исключаются понятия, не охватывающие соответствующих понятий, входящих в **S**. Предикат **v_поиск** определится следующим образом [77]:

```
v_поиск(отрицат(Пример), G, S, Нов_G, Нов_S, Подстановки): –
      удалить(X, S, охватывает(X, Пример), Нов_S),
      специализировать_мн-во(G,Конкр_G, Пример, Подстановки),
      удалить(X, Конкр_G, (элемент(Y, Конкр_G),
                           более_общее(Y, X)),Сокращенное_G),
      удалить(X, Сокращенное_G, (элемент(Y, Нов_S),
                           not охватывает(X, Y)), Нов_G).

v_поиск(положит(Пример), G, [ ], Нов_G, [Пример], _ ): –
       удалить(X, G, not охватывает(X, Пример), Нов_G).

v_поиск(положит(Пример), G, S, Нов_G, Нов_S, _ ): –
       удалить(X, G, not охватывает(X, Пример), Нов_G),
       обобщить_мн-во(S, Обобщ_S, Пример),
       удалить(X,обобщ_S,(элемент(Y,обобщ_S),
                           более_общее(Y, X)),Сокращенное_S),
       удалить(X, Сокращенное_S, not(элемент(Y, Нов_G),
                                     охватывает(Y, X)),Нов_S).
```

В соответствии с алгоритмом Митчелла (алгоритмом исключения понятий-кандидатов), вызов предиката **v_поиск** выполняется до тех пор, пока множества **G** и **S** не совпадут:

```
исключ_понятий_кандидатов([G], [S], _ ): –
        охватывает(G, S), охватывает(S, G),
        write('искомое понятие – '), write(G), nl.
исключ_понятий_кандидатов(G, S, Подстановки): –
        write('G='), write(G), nl, write('S='), write(S), nl,
        write('Введите пример:'), read(Пример),
        v_поиск(Пример, G, S, Нов_G, Нов_S, Подстановки),
        иключ_понятий_кандидатов(Нов_G,Нов_S,Подстановки).
```

Проиллюстрируем работу программы с помощью следующего вызова

```
? – исключ_понятий_кандидатов([[ _, _, _]], [ ],
        [[маленький, средний, большой],
         [желтый, зеленый, красный],.
         [круг, квадрат, треугольник]]).

G=[[ _0048, _0050, _0058]]
S=[ ]
Введите пример: положит([средний, круг, красный]).
G=[[ _0048, _0050, _0058]]
S=[[средний, круг, красный]
Введите пример: отрицат([средний, квадрат, зеленый]).
G=[[ _07C0, круг, _07D0], [ _ 0748, 0750, красный]]
S=[[средний, круг, красный]
Введите пример: положит([большой, круг, красный]).
G=[[ _07C0, круг, _07D0], [ _ 0748, 0750, красный]]
S=[[ _13F8, круг, красный]]
Введите пример: отрицат([средний, треугольник, красный]).
G=[[ _07C0, круг, _07D0]]
S=[[ _13F8, круг, красный]]
Введите пример: отрицат([средний, круг, зеленый]).
Искомое понятие – [ _231C, круг, красный]
Yes.
```

Здесь числовые кодовые комбинации, начинающиеся с символа подчеркивания, – переменные. Поэтому сформированный программой вектор признаков **[_231C, круг, красный]** соответствует понятию “красный круг”.

6.11.5. Поиск в сети фреймов

Рассмотрим сеть иерархически упорядоченных фреймов, представляющих графические объекты (рисунок 6.12).

Фрейм на языке Пролог можно представить совокупностью предикатов **фрейм(Имя-фрейма,Имя-слота,Фацет,Значение)**, каждый из которых определяет значение соответствующего фацета (аспект) слота. Например, корневой графический объект можно описать предикатом

фрейм(граф-объект, позиция, значение, [0, 0]).

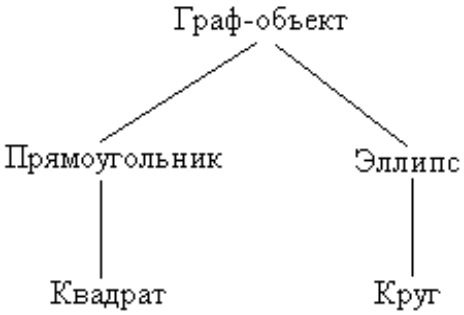


Рисунок 6.12 – Иерархия графических объектов

Фрейм прямоугольника отличается от обобщенного графического объекта наличием дополнительных слотов:

```
фрейм(прямоугольник, is_a, значение, граф-объект).
фрейм(прямоугольник, сторона-x, значение, 1).
фрейм(прямоугольник, сторона-x, диапазон, [ 1, 511]).
фрейм(прямоугольник, сторона -y, значение, 2).
фрейм(прямоугольник, сторона -y, диапазон, [ 1, 311]).
фрейм(прямоугольник, площадь, если-нужно, площадь-прямоугольника).
```

Поиск значений по иерархии фреймов можно выполнить с помощью предиката:

```
поиск(Фрейм, Слот, Фацет, Значение):-
        фрейм(Фрейм, Слот, Фацет, Значение).

поиск(Фрейм, Слот, Фацет, Значение):-
        фрейм(Фрейм, is_a, значение, Род-фрейм),
        поиск(Род-фрейм, Слот, Фацет, Значение).
```

Так как значения некоторых слотов могут вычисляться с помощью процедуры "**если-нужно**", то на основе предиката **поиск** определим общий предикат определения значений:

```
определить-значение(Фрейм, Слот, Значение):-
                поиск(Фрейм, Слот, Фацет, 3),
                (Фацет = значение, !, Значение = 3;
                 Фацет = если-нужно,
                 Демон = .. [ 3, Фрейм, Слот, Значение ],
                 call(Демон)), ! .
```

В этом случае процедуры-демоны должны представляться в виде предикатов:

имя_процедуры_демона(Фрейм, Слот, Значение):- . . .

Например:

```
площадь-прямоугольника(Фрейм. Слот, Значение):-
        определить-значение(Фрейм, сторона-x, X),
        определить-значение(Фрейм, сторона-y, Y),
        Значение  is  X * Y, !.
```

Вопросы для самопроверки

1. Приведите примеры следующих основных понятий языка Пролог: константа, переменная, список, факт, правило, терм.
2. Сформулируйте правила сопоставления термов, используемые в Прологе.
3. Объясните на примере декларативный и процедурный смысл пролог-программы.
4. Постройте дерево вывода для целевого утверждения **member(c,[a, d, c])**.
5. Как в пролог-программе объявить оператор? Как специфицируются типы операторов?
6. Объясните на примере порядок вычисления арифметического выражения, представив его в виде древовидной структуры.
7. Каким образом выполняется оператор **is**?
8. Какие формы записи списков используются в Прологе?
9. Приведите общую схему предиката, выполняющего обработку каждого элемента списка.
10. Определите на языке Пролог предикаты удалить и реверс, обрабатывающие списки рекурсивно.
11. Объясните на примере назначение предиката отсечения.
12. Что в языке Пролог называют группой?
13. Как в Прологе реализуется отрицание? Приведите примеры.
14. Что называют метапеременными (метаусловиями)?
15. Определите с помощью метапеременных конструкцию “если-то-иначе”. Приведите пример её использования.
16. Как в Прологе организовать итерационный цикл? Определите предикат **все_решения(Условие,X)**, обеспечивающий поиск всех значений **X**, при которых выполняется метаусловие **Условие(X)**.
17. Перечислите и объясните процедурный смысл встроенных предикатов, осуществляющих ввод-вывод.
18. Определите на Прологе предикат, осуществляющий ввод арифметического выражения с клавиатуры (из файла) и вывод его значения на экран (в файл).
19. Объясните следующие встроенные предикаты: **var, nonvar, atom, atomic, integer.**
20. Объясните предикаты: **name(A,L), functor(T,F,A), arg(N,T,A), T=..L.**
21. Объясните предикаты: **assert(X), asserz(X), retract(X), consult(F), save(F), restore(F)**.

22. Определите на Прологе предикат, проверяющий вхождение узла **X** в бинарное дерево **Д**.
23. Определите на Прологе предикат, добавляющий узел **X** в бинарное дерево **Д**.
24. Определите на Прологе предикат, удаляющий узел **X** из бинарного дерева **Д**.
25. Реализуйте на Прологе поиск в глубину в пространстве состояний.
26. Реализуйте на Прологе поиск в ширину в пространстве состояний, используя для раскрытия вершин встроенный предикат **bagof.**
27. Реализуйте на Прологе А-алгоритм поиска решений в пространстве состояний.
28. Приведите пример простой фреймовой сети и реализуйте на Прологе поиск значений слотов.

ГЛАВА 7
ЭКСПЕРТНЫЕ СИСТЕМЫ

Среди систем искусственного интеллекта наиболее широкое применение получили экспертные системы (ЭС). Под *экспертной системой* понимают программную систему, аккумулирующую знания эксперта в определенной области и вырабатывающую решения и рекомендации на уровне эксперта. ЭС отличаются от традиционных программных систем обработки данных тем, что они:

- реконструируют методику решения задачи экспертом в соответствующей области;
- формируют выводы, основываясь на знаниях, которые хранятся отдельно от программного кода, управляющего процессом вывода;
- используют при решении задач эвристические методы, которые не требуют исчерпывающей исходной информации, и, следовательно, обеспечивают решение, характеризуемое определенной степенью уверенности;
- имеют возможность накапливать знания в процессе функционирования;
- обладают способностью объяснять предлагаемые решения;
- обеспечивают “дружественный”, как правило, естественно-языковый интерфейс с пользователем.

В главе рассматриваются основные функции и компоненты ЭС, обсуждаются этапы разработки ЭС. Важная роль в ходе разработки ЭС отводится приобретению знаний, поэтому в главе рассматриваются основные проблемы приобретения знаний в процессе общения с экспертами. Существенное значение имеет повышение эффективности поиска решений в ЭС, а также способность объяснять ход своих рассуждений. Эти вопросы рассматриваются на примере продукционных ЭС. Описывается реализация ЭС средствами языков Лисп и Пролог на базе продукционной и фреймовой моделей представления знаний.

7.1. Основные функции и компоненты экспертных систем

Принципиальное отличие ЭС от других систем, основанных на знаниях, состоит в том, что знания, хранящиеся в ЭС, приобретаются в процессе общения с экспертом. ЭС представляет собой программную систему, использующую эвристические методы, разработанные экспертами для решения задач в узкоспециальной области. Основываясь на знаниях и эвристических методах поиска решений, ЭС характеризуются следующими свойствами:

- открытостью;
- гибкостью;
- недетерминированностью решений и рекомендаций.

Открытость ЭС означает, что пользователь может проверить решения, принимаемые ЭС, на любом шаге выполнения программы. Наличие данного свойства ЭС важно по следующим причинам. Во-первых, пользователь не сможет доверять рекомендациям ЭС, если у него не будет возможности проверить обоснованность сделанных выводов. Пользователь должен получить всю информацию, необходимую ему для того, чтобы быть уверенным в том, что принятое решение правильно. Поэтому ЭС должна обладать способностью объяснять предлагаемые решения. Во-вторых, открытость позволяет оценивать корректность знаний, задействованных на каждом этапе логического вывода. Данное обстоятельство весьма важно при отладке базы знаний.

Гибкость ЭС означает простоту модификации базы знаний. ЭС должна иметь средства, обеспечивающие добавление, изменение и удаление элементов базы знаний. Эта возможность предполагает отделение базы знаний от процедур поиска решений. Наиболее просто выполняется модификация баз знаний продукционного типа. Добавление и удаление правил-продукций не требует изменения программного кода.

Недетерминированность обусловлена применением в ЭС эвристических методов решения задач, которые не гарантируют успех. Эвристическая природа как экспертных знаний, так и методов поиска решений создает определенные трудности при оценке качества функционирования ЭС. Один из путей решения этой проблемы состоит в сравнении решений, предлагаемых ЭС и экспертом. Считается, что ЭС функционирует на уровне эксперта, если пользователь, которому неизвестно, как получены результаты экспертизы, не может отличить лучшие решения, предложенные ЭС, от соответствующих решений эксперта.

ЭС ориентированы на классы задач, которые плохо поддаются решению с помощью математических методов. Перечень типовых задач, решаемых ЭС в самых различных областях, включает [20]:

- интерпретацию – извлечение информации из первичных данных;
- диагностику – обнаружение неисправностей и причин их появления в некоторой системе;
- мониторинг – непрерывная интерпретация данных в реальном времени с сигнализацией о выходе тех или иных параметров за допустимые пределы;
- прогноз – предсказание вероятных последствий на основе прошедших и настоящих событий;
- планирование – определение последовательности действий, направленных на достижение заранее поставленных целей;
- проектирование – определение конфигурации системы при заданных ограничениях;
- отладку и ремонт – выполнение последовательности действий по приведению той или иной системы к требуемым режимам функционирования;
- обучение – интерпретация, диагностика и коррекция знаний и умений обучаемого;
- управление – формирование управляющих воздействий, определяющих поведение сложных систем.

Типовая архитектура ЭС изображена на рисунке 7.1. Взаимодействие с ЭС осуществляется с помощью интерфейса. В качестве пользователя ЭС выступает либо конечный пользователь (лицо, принимающее решение), либо эксперт, либо инженер по знаниям. Этим категориям пользователей предоставляются разные возможности взаимодействия с системой. В ЭС применяются различные варианты реализации интерфейса: меню-ориентированный, графический, командный, естественно-языковый, речевой. Кроме обеспечения взаимодействия с различными категориями пользователей, интерфейс выполняет необходимые функции сопряжения с внешними объектами: базами данных, различными датчиками, коммуникационным оборудованием и т.п.

Ядро ЭС образует база данных, база знаний и машина вывода. База данных представляет собой рабочую память, в которой хранятся текущие данные, заключения и другая информация, имеющая отношение к анализируемой системой ситуации. База знаний обеспечивает хранение знаний, представленных с помощью одной из моделей: логической, продукционной, фреймовой, сетевой (см. главу 2).

Машина вывода (подсистема поиска решений), используя данные и знания, организует управление выводом в соответствии с используемой моделью представления знаний. Например, при использовании продукционной модели представления знаний выполняется цикл “распознавание-действие”. Процедуры, реализующие цикл управления, отделены от про-

дукционных правил. Отделение знаний, хранящихся в ЭС, от алгоритмов поиска решений позволяет:

- представлять знания в более естественной форме по сравнению со знаниями, встроенными непосредственно в процедуры;
- разработчику ЭС в большей степени сосредоточиться на накоплении и организации предметных знаний, а не на деталях компьютерной реализации;
- изменять содержимое баз знаний без необходимости внесения изменений в программный код;
- создавать *оболочки экспертных систем* (рисунок 7.1), которые можно наполнять знаниями различных предметных областей;
- экспериментировать с различными режимами управления ЭС без изменения баз знаний.

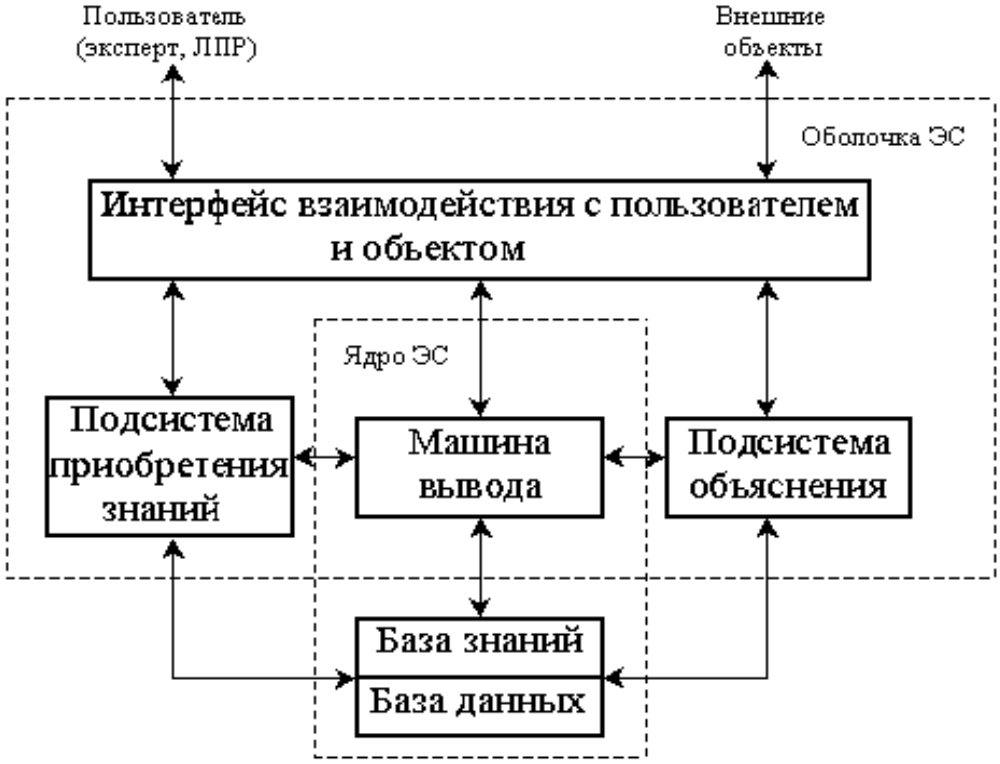


Рисунок 7.1 – Структурная схема ЭС

Подсистема объяснения ЭС позволяет пользователю выяснить, как система получила решение задачи, и какие знания были при этом использованы. Это повышает доверие пользователя к системе и облегчает её тестирование. Подсистема объяснения позволяет отвечать на две группы вопросов:

- вопросы типа “*как*” (как система пришла к этому выводу?);

- вопросы типа “*почему*” (почему требуются значения тех или иных фактов?).

Подсистема приобретения знаний используется как с целью автоматизации процесса наполнения ЭС знаниями, так и при корректировке базы знаний, при ее обновлении, пополнении или исключении элементов знаний. Данные процессы реализуются либо при непосредственном участии инженера по знаниям (эксперта), либо в автоматическом режиме в ходе обучения системы.

7.2. Разработка экспертных систем

Создание ЭС сопряжено с большими финансовыми и трудовыми затратами. Поэтому попытка применить ЭС для решения некоторой проблемы, которая не отвечает ряду требований, может обернуться существенными потерями. Укажем критерии, на основе которых должны отбираться задачи для решения с помощью экспертных систем.

Использовать ЭС для решения задачи следует тогда, когда разработка ЭС *возможна*, *оправдана* и *уместна* [20]. Чтобы разработка ЭС была *возможна*, необходимо выполнение следующих требований:

- должны быть эксперты, которые эффективно решают проблемы в заданной области;
- оценки правильности решений, сделанные различными экспертами, должны в основном совпадать, иначе нельзя будет определить качество разработанной ЭС;
- эксперты должны уметь объяснять методы, которые они используют в ходе решения задачи, в противном случае знания экспертов не могут быть перенесены в память ЭС;
- задача должна решаться на основе рассуждений, а не физических действий (если требуется совмещение интеллектуальных навыков и физических действий, то интеллектуальная часть реализуется средствами искусственного интеллекта, а физические действия – различными механическими системами, что характерно для робототехники);
- задача, возлагаемая на ЭС, не должна быть слишком трудной, ее решение должно осуществляться экспертом за несколько часов, а не дней или недель;
- задача должна относиться к “проработанной” области, иначе потребуются основательные исследования;
- задача не должна в значительной степени опираться на соображения “здравого смысла”, так как подобные знания плохо переносятся в ЭС.

Применение ЭС к решению той или иной задачи может быть *оправдано* следующими факторами:

- решение задачи создает значительный экономический эффект;
- привлечение экспертов к решению задачи невозможно из-за их малочисленности и высокой оплаты их труда;
- увольнения персонала организации приводят к значительному снижению уровня компетенции в организации;
- необходимость принятия решений во вредных условиях, исключающих присутствие человека.

Использование ЭС *уместно*, если решаемая задача обладает следующими характеристиками:

- решение задачи осуществляется путем манипулирования символьными структурами, а не числами, что характерно для традиционных математических пакетов;
- процесс решения задачи носит эвристический характер и основан на использовании эмпирических правил, а не формальных математических процедур;
- для подготовки эксперта требуются многие годы обучения;
- задача должна представлять практический интерес и быть достаточно узкой, чтобы допускать решение методами обработки знаний и.

ЭС представляют собой сложные программные комплексы, разработка которых требует нескольких человеко-лет. В разработке ЭС участвуют следующие специалисты: инженер по знаниям, эксперт и программисты.

Инженер по знаниям является специалистом в области представления знаний и языков искусственного интеллекта. Его основные задачи: выбор инструментальных средств реализации проекта, извлечение знаний из области экспертизы, структурирование знаний и корректное заполнение базы знаний.

Эксперт определяет знания, необходимые для решения задачи с помощью ЭС, обеспечивает их полноту и правильность.

Программист обеспечивает реализацию всех основных составных частей ЭС и их сопряжение друг с другом, а также с внешними подсистемами.

Существенную роль при разработке ЭС играет и конечный пользователь, который определяет многие проектные ограничения. Если требования конечного пользователя не будут полностью удовлетворены, то проект будет неудачным. В ходе проектирования необходимо постоянно учитывать вопросы, ассоциируемые с конечным пользователем:

- какой уровень объяснения требуется пользователю;
- может ли пользователь обеспечить ввод корректных данных;
- удовлетворителен ли пользовательский интерфейс;

- упрощает ли система решение задач пользователем и др.

Аналогично большинству программных систем искусственного интеллекта при разработке ЭС используется концепция “быстрого прототипа”. Согласно этой концепции за короткий промежуток времени создается упрощенная версия ЭС – прототип, который последовательно эволюционирует в законченную систему. Такая опытная технология разработки характеризуется постоянным взаимодействием инженера по знаниям с экспертом. Работа над проектом начинается с “погружения” инженера по знаниям в предметную область с помощью учебных пособий. Предварительное знакомство инженера по знаниям с основными понятиями предметной области помогает его общению с экспертом. Затем инженер по знаниям начинает процесс извлечения экспертных знаний. Обычно это выполняется фиксацией объяснений эксперта при решении простых задач. Полезно, чтобы инженер по знаниям был новичком в предметной области. Эксперты обычно не объясняют достаточно детально последовательность действий при решении сложных задач. Часто они опускают шаги решения задачи, которые являются тривиальными или кажутся им очевидными за многие годы работы в заданной области. Инженер по знаниям, благодаря его наивным представлениям о предметной области, должен быть способен обнаружить “белые пятна” в объяснениях эксперта.

После получения инженером по знаниям общего представления о предметной области и решения с помощью эксперта нескольких задач, он готов приступить к действительному проектированию системы. Инженер по знаниям выбирает подходящую модель представления знаний, определяет поисковую стратегию (прямую, обратную и пр.) и проектирует пользовательский интерфейс. Данный этап работы завершается созданием прототипа ЭС. Прототип ЭС должен решать задачи из ограниченной предметной области и обеспечивать возможность тестирования принятых проектных допущений.

Используя прототип ЭС для решения новых задач, инженер по знаниям и эксперт проверяют и устраняют недостатки знаний, заложенных в ЭС. Если выясняется, что допущения, сделанные при создании прототипа, корректные, то можно перейти к проектированию действующего варианта ЭС. Преобразование прототипа в действующую ЭС может включать перепрограммирование ее составных частей на языках более низкого уровня, позволяющих учесть особенности конкретной вычислительной системы и повысить эффективность реализации ЭС.

Построение ЭС выполняется последовательной корректировкой решений, принимаемых на различных этапах проектирования. Это позволяет говорить, что ЭС скорее “выращивается”, чем конструируется. На рисунке 7.2 изображена структурная схема проектирования ЭС, в основе которой лежит цикл “испытание-корректировка”.

В процессе разработки ЭС выделяют пять этапов: идентификацию, концептуализацию, формализацию, реализацию и испытание [20,12].

На этапе идентификации формулируется проблема, определяются задачи, ресурсы и цели. В результате получается неформальное описание задач, подлежащих решению, и формируются основные требования к ЭС.

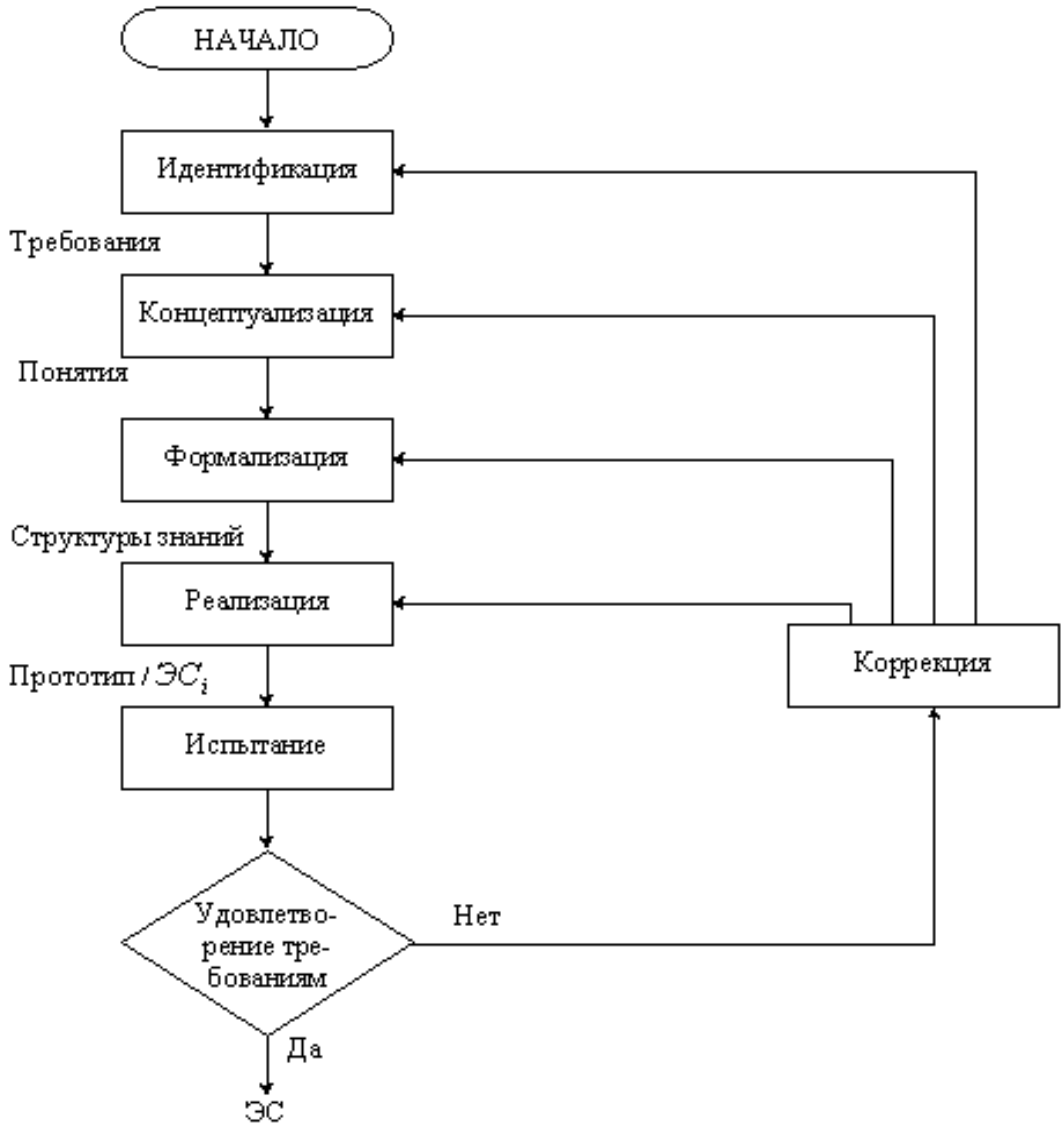


Рисунок 7.2 – Этапы разработки ЭС

На этапе концептуализации инженер по знаниям совместно с экспертом выясняет, какие понятия, отношения, стратегии поиска решений требуются для решения задач в заданной области. Этап завершается разработкой неформального описания предметной области в виде схемы, которая отражает основные понятия предметной области и их взаимосвязи.

Формализация состоит в выборе модели представления знаний и разработке на её основе формального представления предметной области.

На этапе реализации формализованные знания закладываются в базу знаний и разрабатываются необходимые программы. Этап первоначальной реализации должен осуществляться быстро, с использованием языков высокого уровня абстракции данных, в частности, с помощью оболочек ЭС. С весьма большой вероятностью, полученная первоначальная реализация (прототип) будет в дальнейшем пересмотрена.

Испытания позволяют выяснить правильность работы системы при решении требуемых задач. В ходе испытаний могут быть обнаружены ошибки, допущенные на разных этапах проектирования, что потребует пересмотра соответствующих решений.

На этапе испытаний промежуточных версий ЭС с системой взаимодействует и конечный пользователь, который определяет, отвечает ли система его требованиям.

В ходе разработки ЭС приходится неоднократно возвращаться на более ранние этапы, пока не будут удовлетворены все требования. Создание прототипа ЭС с применением наиболее распространенных продукционных моделей занимает от нескольких месяцев до 1-2 лет. Доводка прототипа до промышленной ЭС с возможным перепрограммированием на алгоритмических языках, например C++, требует от 2 до 4 лет. Ускорения процесса разработки можно добиться при использовании средств автоматизации проектирования ЭС [12].

7.3. Приобретение знаний

Приобретение знаний состоит в передаче и преобразовании опыта решения проблем от источника знаний к системе. Термин приобретение знаний не фиксирует способ передачи знаний. Например, передача может осуществляться с помощью специальных программ, выявляющих необходимые элементы знаний в больших массивах данных. Функции, которые должна выполнять подсистема приобретения знаний (рисунок 7.1), в значительной степени зависят от того, какие возможности имеются в системе по формализации поступающей информации до уровня знаний. Если в системе не предусмотрено никаких средств автоматизации процесса приобретения знаний, то все необходимые операции по приобретению знаний выполняет инженер по знаниям. В случае если передача знаний выполняется путем опроса эксперта, то говорят не о приобретении знаний, а об *извлечении знаний*.

Извлечение знаний осуществляется в процессе систематического диалога инженера по знаниям и эксперта. При этом инженер по знаниям должен учитывать следующие обстоятельства:

- знания и умения эксперта за многие годы практической работы организуются в сложную интегрированную структуру, в которой сложно выделить отдельные элементы;
- существует большой разрыв между умением эксперта решать задачи и простым набором декларативных сведений заданной предметной области;
- теоретические и практические знания эксперта представляют собой индивидуальную модель “реального мира”, на формирование которой большое влияние оказывают многие факторы, в том числе и социальные;
- знания изменчивы и могут радикально меняться при смене господствующих идей предметной области.

Получение информации от экспертов может выполняться на основе двух подходов: наблюдательном и интуитивном [20]. При наблюдательном подходе следят за тем, как эксперт решает задачи, стараясь не сделать ничего, что могло бы повлиять на ход решения. Процесс решения оформляется в виде протокола и анализируется инженером по знаниям совместно с экспертом.

При интуитивном подходе осуществляется либо самоанализ эксперта, либо анализ моделей, разработанных инженером по знаниям. В первом случае эксперт (путем самоанализа) выявляет суть своих знаний и пытается с помощью инженера по знаниям внести их в систему. Во втором случае инженер по знаниям изучает литературу и другие источники информации, разрабатывает модель представления знаний, проверяет ее адекватность с помощью эксперта.

Оба подхода сопряжены с определенными трудностями. В методе наблюдения эксперт может допускать пробелы в описании шагов решения задачи, которые восстанавливаются инженером по знаниям с большим трудом, даже тогда, когда он привлекает в помощники эксперта. При интуитивном методе эксперт часто получает решение, не осознавая, какие приемы он использует. В этом случае эксперт может построить правдоподобные рассуждения, которые не соответствовали действительному ходу решения задачи.

На практике инженер по знаниям может использовать оба подхода, руководствуясь следующими приемами извлечения знаний [20]:

- наблюдение за экспертом в ходе решения задачи;
- обсуждение задач, подготовленных инженером по знаниям;
- описание типовых задач предметной области экспертом;
- анализ задач, близких к реальным, с целью выявления стратегии, которую использует эксперт;
- проверка системы (прототипа), в ходе которой эксперт предлагает перечень задач для решения с помощью выявленных правил;

- оценивание прототипа системы экспертом;
- оценивание прототипа системы другими экспертами.

Полученные от эксперта и из других источников знания фиксируются в виде концептуальной схемы, которая отражает представления предметной области инженером по знаниям. Хотя эта модель существенно отличается от модели эксперта, в действительности она соответствует структуре разрабатываемой базы знаний.

Рассмотренный процесс извлечения экспертных знаний является дорогостоящим, так как требует постоянного общения с экспертом (экспертами). Кроме этого, он характеризуется неизбежными ошибками из-за проблем взаимопонимания между инженером по знаниям и экспертом. Если учесть тот факт, что многие экспертные системы требуют непрерывной поддержки базы знаний в актуальном состоянии, то становится ясным, что извлечение знаний, в ходе которого инженер по знаниям преобразует знания эксперта во внутреннюю форму представления – это длительный и дорогостоящий процесс.

С момента появления указанной проблемы непрерывно предпринимались попытки автоматизировать процесс приобретения знаний с помощью методов обучения. Данное направление получило название *формирование знаний* и базируется на моделях индуктивного вывода. Наиболее часто для автоматического приобретения знаний используют различные вариации алгоритма ID3 (см. главу 4). Однако автоматическое приобретение знаний на основе этих методов, пока не стало промышленной технологией [12].

Шире применяются инструментальные средства поддержки деятельности инженера по знаниям и эксперта на этапе концептуализации и структурирования знаний. Для того чтобы такие инструментальные средства могли легко использоваться инженером по знаниям или экспертом, они должны учитывать способ рассуждения эксперта и опираться на специальные языки. По сравнению с естественным языком, узкоспециальные языки более формализованы. Кроме этого, в конкретной предметной области используются определенные методы решения задач. Наличие специального языка и конкретных методов решения задач позволяет разрабатывать специализированные программные средства автоматизированного приобретения знаний. К таким средствам относят:

- универсальные инструменты для разработки ЭС, которые поддерживают основные модели представления знаний и соответствующие механизмы вывода (OPSS, KEE, ART, Knowledge Graft [12,63]);
- проблемно-ориентированные системы приобретения знаний, характеризуемые определенными методами решения задач, пред-

ставлением знаний и формами ввода знаний (например, SALT – система приобретения знаний для задач конструирования [63]);
- предметно-ориентированные системы приобретения знаний, которые в дополнение к предыдущему классу инструментальных средств снабжаются основными предметными знаниями, например, техническими терминами (OPAL – система приобретения медицинских знаний в области онкологии [63]).

Инструментальные средства приобретения знаний существенно отличаются друг от друга по показателю "время разработки – гибкость". Универсальные средства характеризуются наибольшим временем разработки и большой гибкостью. Предметно-ориентированные системы, наоборот, характеризуются наименьшим временем на разработку и узкой, конкретной областью применения. Проблемно-ориентированные средства занимают промежуточное положение. Подробнее ознакомиться с вопросами приобретения знаний можно в [12].

7.4. Поиск и объяснение решений

При создании ЭС используют различные модели представления знаний: логические, продукционные, сетевые (см. главу 3). Наибольшее распространение получили ЭС, основанные на продукционных правилах. Рассмотрим особенности управления поиском решений в таких системах, а также выясним, как формируются объяснения принятых решений.

База знаний ЭС продукционного типа состоит из множества правил вида:

Если C то X,

где C – последовательность выражений (образцов), определяющих условия активизации правила; X – последовательность действий, которые могут модифицировать состояние рабочей памяти (рисунок 3.2). Продукционное правило напоминает условный оператор процедурных языков программирования. Однако, несмотря на внешнюю схожесть с условным оператором, продукционные правила характеризуются следующими отличительными признаками:
- действия X выполняются, когда выражения (образцы) из C сопоставимы с соответствующими элементами (фактами) рабочей памяти;
- возможно одновременное сопоставление выражения из C с несколькими элементами рабочей памяти, что порождает необходимость запоминания экземпляров потенциально конфликтующих правил;

- действия X могут заключаться как в добавлении новых элементов в рабочую память (базу данных), так и в исключении существующих элементов;
- правила часто содержат дополнительную управляющую информацию, например: название модуля базы знаний, к которому относится правило; приоритет правила; фаза выполнения, в течение которой разрешается выполнять правило.

Алгоритм поиска решения в ЭС продукционного типа использует понятие списка конфликтных правил *S*. К конфликтным правилам относят все экземпляры правил, условная часть которых сопоставима с соответствующими элементами рабочей памяти. Обобщенный алгоритм прямого вывода на правилах-продукциях можно записать в виде:

```
Определить экземпляры конфликтных правил и
поместить их в список S;
While S ≠ NIL Do
Begin
    Выбрать экземпляр правила из S;
    Изменить состояние рабочей памяти в соответствии
    с заключением правила;
    Модифицировать список S, добавив или удалив
    экземпляры правил в соответствии с изменением
    состояния рабочей памяти
End.
```

Здесь задание начальных условий цикла сопряжено со значительным объемом вычислений, так как формирование списка конфликтных правил S требует сопоставления всех фактов рабочей памяти со всеми выражениями (образцами) из предпосылок всех правил. Отметим, что один и тот же образец правила может быть сопоставлен с несколькими фактами. Поэтому список S состоит не из правил, а из экземпляров правил, которые представляют собой частные случаи правил с конкретными подстановками. В ходе выполнения алгоритма нет необходимости повторять сопоставление всех фактов рабочей памяти и всех образцов правил. На каждой итерации анализируются только изменения состояния рабочей памяти, связанные с выполнением последнего правила.

Для того чтобы определить экземпляры правил, которые можно будет активизировать в дальнейшем и, следовательно, включить их в список *S*, машина вывода систем, основанных на правилах, выполняет на этапе инициализации *частичное сопоставление*. Частичное сопоставление осуществляется, когда часть образцов предпосылки правила сопоставима с фактами рабочей памяти. Если частичные сопоставления были запомнены на этапе инициализации, то на очередной итерации сопоставляются толь-

ко оставшиеся образцы из предпосылки экземпляра правила. Частичное сопоставление позволяет повысить скорость вывода.

Рассмотрим пример. Пусть имеется правило

$$(P_1 \wedge P_2 \wedge ... \wedge P_8) \Rightarrow Q.$$

На этапе инициализации для этого правила были успешно сопоставлены образцы P_1, ..., P_6, и соответствующие подстановки запомнены машиной вывода. После выполнения очередного правила из списка *S*, машина вывода определяет, имеются ли в рабочей памяти изменения, оказывающие влияние на результаты частичного сопоставления для рассматриваемого правила (т.е. на P_1, ..., P_6). Если изменения не касаются, например, образцов P_1, ..., P_5, но делают невозможным сопоставление для образца P_6, то машина вывода сохраняет подстановки для образцов P_1, ..., P_5. Если же изменения не касаются образцов P_1, ..., P_6 и дополнительно успешно выполняется сопоставление для образцов P_7 и P_8, то рассматриваемое правило помещается в список активизируемых правил.

Дальнейшее повышение эффективности операции сопоставления основано на использовании *алгоритма RETE*, предложенного Форджи [64] Идея алгоритма базируется на распространении результатов частичного сопоставления между правилами. Это позволяет исключить многократное сопоставление, по сути, одних и тех же образцов, встречающихся в различных правилах. Пусть имеются два правила

$$P_0 \wedge P_1 \wedge P_2 \wedge P_3 \Rightarrow Q_1,$$
$$P_0 \wedge P_1 \wedge P_4 \wedge P_2 \Rightarrow Q_2.$$

Тогда результаты частичного сопоставления образцов P_0 и P_1 для первого правила могут быть распространены на второе правило. Но подстановки, сделанные в ходе сопоставления образца P_2 первого правила, не могут использоваться во втором правиле из-за того, что раньше должно выполняться сопоставление образца P_4, которое может сделать недопустимыми запомненные подстановки для P_2.

В соответствии с алгоритмом RETE, в результате компиляции правил формируется сеть, состоящая из двух частей. Первая часть сети представляет сеть образцов, а вторая часть – сеть объединения. Рассмотрим следующие правила, в которых образцы представлены триплетами “объект - атрибут - значение” [63]:

(P_0 (длина ?x)(высота ?x))(P_1 (длина ?x)(ширина ?y)) $\Rightarrow Q_1$,

(P_0 (длина ?y)(высота ?y))(P_1 (длина ?x)(ширина ?x)) $\Rightarrow Q_2$.

В ходе сопоставления в переменные, начинающие символом ‘?’, выполняются подстановки. Вхождениям одной и той же переменной в различные образцы должны назначаться одинаковые значения. Если переменная входит несколько раз в один и тот же образец, то она получает значения в соответствии с сетью образцов. Если переменная входит в различные образцы, то значения назначаются с помощью сети объединения (рисунок 7.3)

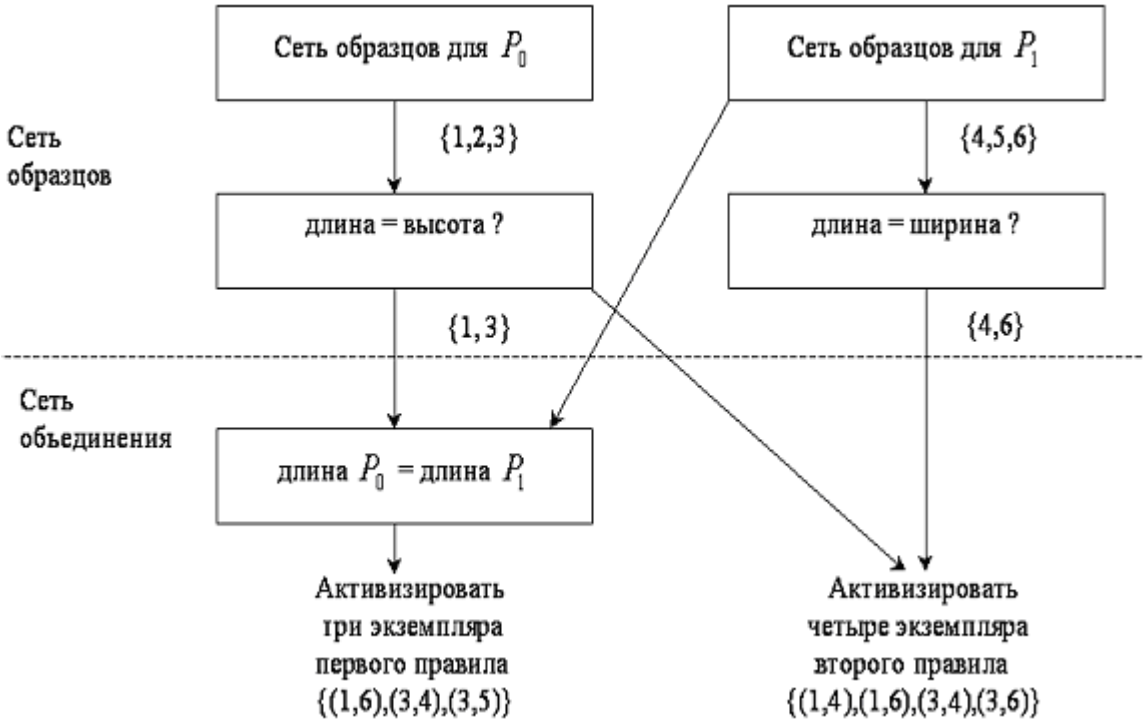


Рисунок 7.3 – Сеть, формируемая алгоритмом RETE

Предположим, что в рабочей памяти имеются следующие факты, относящиеся к объектам P_0 и P_1:

1) (P_0 (длина 12)(высота 12));
2) (P_0 (длина 15)(высота 10));
3) (P_0 (длина 15)(высота 15));
4) (P_1 (длина 15)(ширина 15));

5) (P_1 (длина 15)(ширина 14));

6) (P_1 (длина 12)(ширина 12)).

Тогда в результате сопоставления образцов и фактов рабочей памяти будут активизированы три экземпляра первого правила, с подстановками фактов {(1,6),(3,4),(3,5)}, и четыре экземпляра второго правила, с подстановками фактов {(1,4),(1,6),(3,4),(3,6)}. Из примера видно, что с помощью сети объединения подстановки распространяются между правилами, что исключает лишние сопоставления.

На каждом шаге машина вывода выбирает экземпляр правила из множества конфликтных правил *S*. Основные стратегии, используемые при этом, рассмотрены в § 3.3.2.

Рассмотрим принципы формирования объяснений выводов в ЭС продукционного типа. Формирование объяснений выводов в ЭС важно по многим причинам. Во-первых, конечные пользователи не смогут доверять рекомендациям ЭС, если они не будут уверены в том, что рассуждения выполнены корректно. Во-вторых, инженеры по знаниям должны получать полную информацию о работе всех подсистем ЭС. В-третьих, эксперты в предметной области нуждаются в проверке хода рассуждений ЭС, чтобы убедиться в том, что экспертные знания применяются правильно.

Для получения объяснений выводов пользователи ЭС могут задавать вопросы двух типов:

- почему система сочла необходимым задать пользователю определенный вопрос;
- как система пришла к соответствующему заключению.

Выясним механизм формирования ответов на эти вопросы на примере обратного вывода. Пусть имеются следующие правила, относящиеся к диагностике неисправностей автомобиля:

```
Правило 1:
        Если  топливо_ поступает = ‘да’ и
                состояние_двигателя = ‘работает_с_перебоями’,
        то
                причина = ‘свечи’;

Правило 2:
        Если  состояние_двигателя = ‘не_запускается’ и
                состояние_освещения = ‘не_ работает’,
        то
                причина = ‘аккумулятор’;
```

Правило 3:
Если состояние_двигателя = ‘не_запускается’ и
состояние_освещения = ‘работает’,
то
причина = ‘стартер’;

Правило 4:
Если наличие_топлива_в_баке = ‘да’ и
наличие_топлива_в_карбюраторе = ‘да’,
то
топливо_поступает = ‘да’.

Процесс вывода начинается с ввода в рабочую память цели

причина = X,

где X – переменная, которая может быть сопоставлена с любой строкой. Значение переменной X, полученное в ходе вывода, будет указывать причину неисправности двигателя. Если предположить, что правила активизируются в порядке их записи, то на первом шаге, в результате сопоставления с заключением первого правила, X получит значение ‘свечи’. Затем система в соответствии с предпосылками правила 1 попытается установить, поступает ли топливо и как работает двигатель. Для решения первой подзадачи будет активизировано правило 4. Так как факты наличие_топлива_в_баке и наличие_топлива_в_карбюраторе в рабочей памяти не содержатся, и нет правил, с помощью которых система могла бы установить их значения, то пользователю будут заданы вопросы относительно указанных фактов. Кроме этого, после выполнения правила 4 пользователю будет задан вопрос, касающийся состояния двигателя.

Обычно системе известны допустимые значения фактов. Тогда пользователю предлагается выбрать один из допустимых ответов. В рассматриваемом случае возможен следующий диалог пользователя с ЭС:

Наличие_топлива_в_баке (да/нет)?
да
Наличие_топлива_в_карбюраторе (да/нет)?
да
Состояние_двигателя (работает_с_перебоями, не_запускается)?
работает_с_перебоями

После чего система сообщает причину неисправности двигателя: причина = ‘свечи’. Если пользователь желает выяснить, почему система задает тот или иной вопрос, то в качестве ответа вводится слово “почему”. Например:

Состояние_двигателя (работает_с_перебоями, не_запускается)?
почему

В результате будет получен ответ:

1) топливо_поступает = ‘да’,
   следовательно, если
2) состояние_двигателя = ‘работает_с_перебоями’,
   то причина = ‘свечи’.

В этом случае система демонстрирует пользователю текущее правило, с которым был связан заданный вопрос.

Чтобы ответить на вопрос “как” в системе сохраняется стек всех решений, принятых в течение сеанса работы. Просматривая стек, можно выяснить, достижение каких подцелей привело к данному решению. Например, при вводе вопроса

Как получено решение топливо_поступает =‘да’

будет получен ответ:

Это следует из правила 4:
Если наличие_топлива_в_баке =‘да’ и
наличие_топлива_в_карбюраторе =‘да’,
то
топливо_поступает = ‘да’;

Факт наличие_топлива_в_баке = ‘да’ введен пользователем;
Факт наличие_топлива_в_карбюраторе =‘да’ введен пользователем.

Таким образом, формирование ответа на вопрос “как” соответствует просмотру дерева целей в направлении от целевого утверждения, фигурирующего в вопросе, до листьев дерева, соответствующих введенным фактам. Из примера видно, что формирование объяснений для случая обратного вывода реализуется с помощью стекового механизма. Сложнее обстоит дело при прямом выводе, т.е. выводе, управляемом данными.

Прямая цепочка рассуждений обеспечивает пользователя менее полезной информацией. Обусловлено это тем, что на промежуточных этапах вывода трудно судить, куда ведет цепочка рассуждений. Так, в ответ на вопрос “почему” пользователю демонстрируется текущее правило. Дальнейшие объяснения не могут быть получены, пока не выполнится следующее правило. Кроме этого, сложно формировать полные ответы на вопрос “как”, даже после достижения цели. Информация, которая предоставляется, ограничивается списком выполненных правил.

7.5. Реализация экспертной системы на языке Лисп

Рассмотрим реализацию обратного вывода в ЭС, основанной на правилах, на примере “игрушечной” задачи классификации животных [94]. Будем представлять правила классификации в форме списка языка Лисп:

```
(ЕСЛИ      условие1 И
           условие2 И
            ...
           условиеМ
ТО         вывод1   И
           вывод2   И
           ...
           выводN).
```

База знаний ЭС будет состоять из 15 правил, обеспечивающих классификацию животных по внешним признакам на 7 групп. Хранение базы знаний осуществим в глобальной переменной *БАЗА_ЗНАНИЙ* [1)]:

```
(defvar *база-знаний*)
(setq    *база-знаний*
     ’ (правило1 правило2 правило3 правило4
       правило5 правило6 правило7 правило8
       правило9 правило10 правило11 правило12
       правило13 правило14 правило15))
(setq  правило1
       ’(если X  имеет шерсть то  X млекопитающее))
(setq  правило2
       ’(если X кормит детенышей молоком то  X млекопитающее))
(setq  правило3
       ’(если X  имеет перья то X птица))
(setq  правило4
      ’(если  X летает и
              X  откладывает яйца
       то X птица))
(setq правило5
      ’(если X млекопитающее  и
              X  ест мясо
       то X  хищник))
(setq правило6
      ’(если X  млекопитающее и
              X  имеет острые зубы и
              X  имеет острые когти и
```

[1)] В программе используются обозначения переменных на русском языке с целью упрощения её понимания. При вводе программы в ЭВМ соответствующие обозначения необходимо заменить англоязычными.

```
                X  имеет прямо посаженные глаза
        то  X  хищник))
(setq правило7
      '(если X  млекопитающее  и
                X  имеет копыта
        то X жвачное))
(setq  правило8
       '(если X  млекопитающее  и
                X  жует жвачку
        то  X жвачное))
(setq  правило9
       '(если X  хищник  и
                X  желто-коричневое и
                X  имеет темные пятна
         то X  гепард))
(setq  правило10
       '(если X  хищник и
                X  желто-коричневое и
                X  полосатое и
                X  имеет черные полосы
         то     X  тигр))
(setq  правило11
       '(если X  жвачное и
                X  длинношеее и
                X  длинноногое и
                X  желто-коричневое и
                X  имеет темные пятна
          то  X  жираф))
(setq  правило12
       '(если X  жвачное и
                X  полосатое и
                X  имеет черные полосы
          то   X зебра))
(setq  правило13
      '(если X  птица и
                X  не летает и
                X  длинношеее и
                X  длинноногое и
                X  черно-белое
         то  X  страус))
(setq  правило14
      '(если X птица и
                X  не летает и
                X  плавает и
                X  черно-белое
         то X пингвин))
(setq  правило15
     '(если  X  птица и
                X  хорошо летает
```

```
то X  альбатрос))
```

С целью упрощения программы преобразуем базу знаний из исходной формы в список, элементами которого являются структуры:

```
(defstruct    rule          ;имя структуры
              имя           ; имя правила
              условия       ; список предпосылок правила
              выводы)       ; список заключений правила
```

Напомним, что создать экземпляр структуры в языке Лисп можно с помощью функции-конструктора, имя которой образуется из слова MAKE и имени структуры. Например:

```
(make-rule
     : имя   'правило15
     : условия  '( (X птица) (X хорошо летает))
     : выводы  '((X альбатрос))  )
```

В этом случае условия и заключения правила задаются подсписками, каждый из которых выступает в роли образца, сопоставляемого с фактами рабочей памяти. Преобразование базы знаний в список, элементами которого являются указанные структуры, выполним с помощью функции ПРЕОБРАЗОВАТЬ:

```
(defvar  *правила*)
(setq  *правила* (преобразовать  *база-знаний*))
```

Здесь значением глобальной переменной *ПРАВИЛА* будет список из структур. Функция ПРЕОБРАЗОВАТЬ определяется следующим образом [49]:

```
(defun преобразовать (база-знаний)
       (mapcar  #'преобразовать-правило  база-знаний ))

;;преобразует правило из формы в виде списка в структуру
(defun преобразовать-правило (имя)
   (let (( правило (eval  имя)))
      (make-rule
           : имя   имя
           : условия  (prem правило)
           : выводы   (concl правило))))

;;возвращает условия правила в виде списка,
;;элементами которого являются образцы
(defun  prem (правило)
```

```
        (список-образцов (cdr правило) nil  nil ))

;; возвращает заключения правила в виде списка,
;; элементами которого являются образцы
(defun concl (правило)
       (список-образцов (cdr (member 'то правило)) nil nil ))

;; формирует список образцов, разделяемых в
;; правиле символом  'и'
 (defun  список-образцов (правило  часть  результат)
      ( cond
           ((null правило)                    ; выход из рекурсии
            (concat результат часть))         ; объединить результат и часть
           ((eq  (car правило) 'то)           ; окончание анализа условий
             (concat  результат часть))
           ((eq (car правило) 'и)             ; начало нового образца
            (список-образцов (cdr  правило)
                             nil
                            (concat результат часть)))
           (t  (список-образцов  (cdr правило)
                                 (concat часть (car правило))
                                  результат))))

  (defun   concat (x y)  (append x (list y)))
```

Преобразование правил в структуры позволяет обращаться к предпосылкам и заключениям правил с использованием имен соответствующих полей. Это упрощает программирование, так как функции, выполняющей поиск решения, нет необходимости знать порядок следования элементов правил.

Машина вывода применяет правила к фактам, содержащимся в рабочей памяти, пока не будет получен искомый результат. Обратный вывод начинается с внесения целевого утверждения рабочую память. Представим возможные целевые утверждения (гипотезы) с помощью глобальной переменной *ГИПОТЕЗЫ*:

```
(defvar *гипотезы*)
(setq *гипотезы*
        ((X  гепард)
        (X  тигр)
        (X  жираф)
        (X  зебра)
        (X  страус)
        (X  пингвин)
        (X  альбатрос)))
```

Переменная *ГИПОТЕЗА* представляет список целевых утверждений, которые система будет пытаться последовательно доказывать. Следовательно, в основной программе должна вызываться рекурсивная функция SOLVE, последовательно обрабатывающая элементы списка *ГИПОТЕЗЫ* [49]:

```
;;; главная программа
(defvar  *факты*)          ;  список фактов
(defvar  *вопросы*)        ;  список фактов, о которых
                           ;  задавались вопросы пользователю
(setq  *факты*  nil)
(setq  *вопросы*  nil)

;; экспертная система классификации животных
(defun  expert ()
   (solve  *гипотезы*))

;; функция последовательно проверяет гипотезы
(defun  solve (гипотезы)
        (cond
              ((null  гипотезы)  “Не могу доказать ни одну гипотезу”)
              ((back-prove                ;  вызов функции доказательства
                 (car  гипотезы))
               (car  гипотезы))                  ;  результат
              (t (solve (cdr  гипотезы)))))      ;  следующая попытка
```

Рассмотрим функцию BACK-PROVE, осуществляющую обратный вывод. Функция базируется на следующих правилах:

1) если гипотеза уже имеется в списке фактов (рабочей памяти), то она доказана;
2) если гипотезы нет в списке фактов, то все правила, которые могут подтвердить гипотезу, помещаются в список КАНДИДАТЫ, и если какое-либо из правил списка КАНДИДАТЫ непосредственно выполняется на множестве известных фактов, то гипотеза доказана, иначе предпринимается попытка доказать все условия одного из правил списка КАНДИДАТЫ, применив рекурсивно рассматриваемую процедуру обратного вывода;
3) если условие, которое требуется доказать, не выводится с помощью правил, то необходимо запросить его значение у пользователя; если условие выполняется, то добавить его к списку фактов.

Выполнимость правила на множестве фактов будем проверять с помощью функции TEST-RULE:

```
(defun  test-rule (правило)
    (match1 (rule-условия  правило) *факты*))
```

```
;;Функция сопоставления образцов правила
;;с множеством фактов
(defun  match1 (образцы  факты)
    (equal  образцы (intersection  образцы  факты)))
```

Здесь функция MATCH1 выполняет сопоставление условий правила (образцов) с фактами. При этом сопоставление реализуется простой проверкой на равенство списка образцов и списка общих элементов (INTERSECTION) двух списков ОБРАЗЦЫ и ФАКТЫ.

Добавление заключений, полученных с помощью выполняемых правил, к списку фактов реализуется функцией ADD-CONCL:

```
(defun add-concl (правило)
(do  ((выводы  (rule-выводы  правило) (cdr  выводы)))
        ((null  выводы)  *факты*)
        (if  (member  (car  выводы)  *факты* : test  # 'equal)
          nil
         (prog1
          (format  t  "~S утверждает, что" (rule-имя  правило))
          (print-el  (car  выводы)) (terpri)
          (push  (car  выводы)  *факты*)))))

;; печать элементов списка
(print-el  (список)
(mapc  # '(lambda  (элемент)  (princ  элемент) (princ  " "))
             список ) t )
```

Описанные выше правила обратного вывода соответствуют трем ветвям COND формы, вызываемой в функции BACK-PROVE:

```
(defun  back-prove(гипотеза)
        (let ((кандидаты))                  ; список правил-кандидатов,
        (cond                               ; способных подтвердить гипотезу
          ((member гипотеза *факты* : test  #'equal) t)       ; ветвь 1

          ((setq кандидаты (поиск-кандидатов гипотеза))   ; ветвь 2
           (if  (непосредственно гипотеза кандидаты)
                t
               (рекурсивно гипотеза кандидаты)))
          (t  (cond                                          ; ветвь 3
                ((member гипотеза *вопросы* : test  #'equal) nil)
                ((and
                     (princ "Верно, что:" )
                     (print-el гипотеза) (progn (princ "?") (terpri) t)
                     (eq (read) 'да))
                     ;если ответ 'да', то добавляем гипотезу к списку фактов
```

```
                (setq *факты* (union (list гипотеза) *факты*))
                ;добавляем гипотезу к списку вопросов независимо от ответа
                (push гипотеза *вопросы*)  t)
          (t  (push гипотеза *вопросы*)  nil))))))

;; формирование списка правил-кандидатов, способных
;; подтвердить гипотезу
(defun поиск-кандидатов (гипотеза)
      (mapcan  #'(lambda (x)
                    (if (member гипотеза (rule-выводы x) : test  #'equal)
                       (list x)))
                *правила*))

 ;; проверяет, можно ли непосредственно доказать
 ;; гипотезу с    помощью какого-либо правила-кандидата
 (defun непосредственно(гипотеза  кандидаты)
      (cond
        ((null кандидаты) nil)
        ((null *факты*) nil)
        ;если правило применимо, то добавляем его заключение к фактам
        ((test-rule (car кандидаты)) (add-concl (car кандидаты)))
        (t  (непосредственно  гипотеза (cdr  кандидаты)))))

;; проверяет, можно ли рекурсивно доказать гипотезу с
;; помощью какого-либо правила-кандидата
(defun  рекурсивно(гипотеза кандидаты)
      (cond
         ((null кандидаты) nil)
         ; проверить возможность доказательства условий правила
         ((test-условия (rule-условия (car кандидаты)))        ; новая подцель
          (add-concl (car кандидаты)))
         (t  (рекурсивно гипотеза (cdr  кандидаты)))))

;; проверяет возможность доказательства всех условий
;; правила с помощью функции обратного вывода
(defun test-условия (условия)
      (every  # 'back-prove  условия))
```

Процесс классификации животных по внешним признакам начинается с вызова основной функции EXPERT. Ниже приведен возможный диалог с системой:

```
(expert)
Верно, что: X  имеет шерсть ?                    ; вопрос
Нет                                              ; ответ
Верно, что: X  кормит детенышей молоком ?
Нет
Верно, что: X  имеет перья ?
```

```
Да
Правило3 утверждает, что   X   птица
Верно, что: X  не летает ?
Нет
Верно, что: X  хорошо летает ?
Да
Правило15 утверждает, что   X   альбатрос
( X  альбатрос )                                    ; результат
```

Рассмотренная реализация ЭС характеризуется простейшими возможностями. Расширить возможности системы можно, реализовав функции объяснения и применив более совершенный вариант функции сопоставления с образцом.

7.6. Реализация экспертной системы на языке Пролог

Рассмотрим два варианта реализации ЭС, один из которых основан на продукционной модели представления знаний, а другой – на фреймовой модели[1)].

7.6.1. Представление знаний правилами

В качестве примера выберем задачу диагностики. Пусть требуется установить причину неисправности электрической плиты, оборудованной нагревательным элементом, контрольной лампочкой и выключателем. Плита подключена к источнику напряжения с помощью электрического кабеля, который обеспечивает передачу тока. Набор правил "неисправность-причина" представим в виде графа (рисунок 7.4).

Простейший способ построения ЭС на Прологе – использование механизма поиска решений, заложенного в Пролог-системе. Для этого представим правила в виде предикатов языка Пролог:

```
/* структура правила:  причина:– неисправность */
нагреватель(неисправен):– лампа(светится),          /* r1 */
                          плита(холодная).
выключатель(не_включен):– тока(нет).                /* r2 */
напряжения(нет):– тока(нет).                        /* r3 */
тока(нет):– плита(холодная),                        /* r4 */
            лампа(не_светится).
лампа(неисправна):– лампа(не_светится),             /* r5 */
                    плита(горячая).
```

[1)]) Примеры и программы данного параграфа разработал Дж. Криз (J. Kriz)

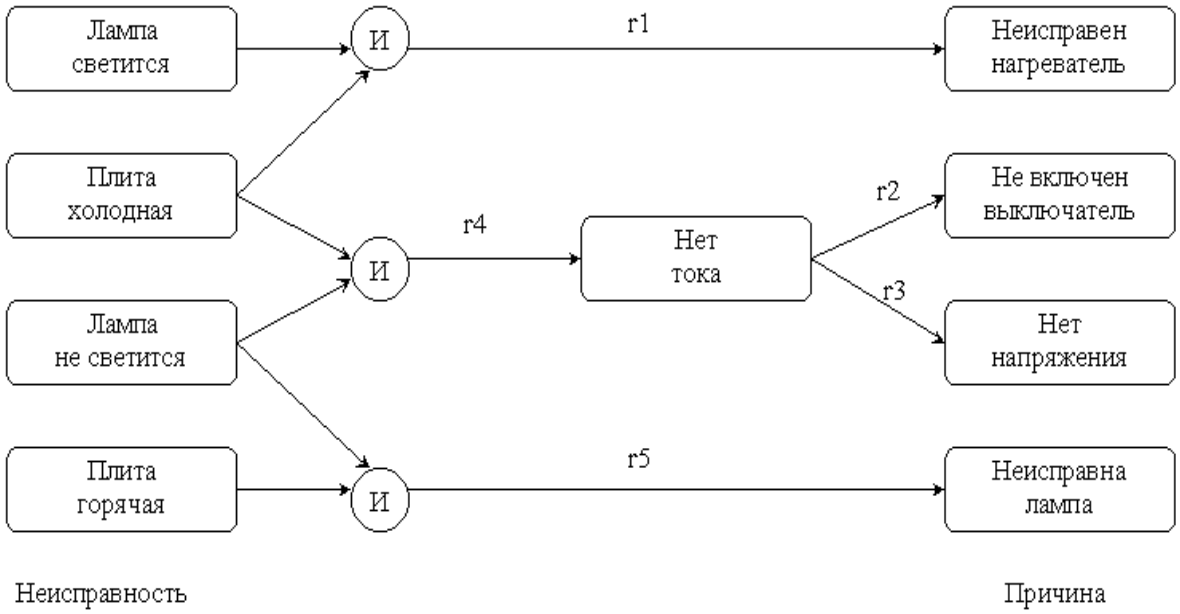


Рисунок 7.4 – Сеть вывода

Определить причину поломки электрической плиты можно, указав обнаруженные неисправности и обеспечив проверку выполнимости каждой из возможных гипотез. Например:

```
/* вариант 1 */
лампа(не_светится).  /*факты*/
плита(холодная).

причина(нагреватель(неисправен)).  /*гипотезы*/
причина(выключатель(не_включен)).
причина(напряжения(нет)).
причина(лампа(неисправна)).

найти(X): – причина(X), call(X).  /*проверка гипотез */
```

Для того чтобы узнать, какая из гипотез неисправности выполняется, необходимо задать вопрос:

```
? – найти(X).
```

Получив такой вопрос, Пролог-система выполнит необходимый обратный поиск с помощью встроенного механизма вывода и вернет ответ:

```
X = выключатель(не_включен);
X = напряжения(нет).
```

Недостатком рассмотренного варианта программы является необходимость внесения обнаруженных неисправностей непосредственно в текст программы в виде фактов. Устранить указанный недостаток можно, запросив значения соответствующих неисправностей у пользователя в процессе выполнения программы:

```
/* вариант 2 */
лампа(не_светится): – спроси('Лампа не светится ?').
плита(холодная): – спроси('Плита холодная ?').
лампа(светится): – спроси('Лампа светится ?').
плита(горячая): – спроси('Плита горячая ?').
спроси(Вопрос): – write(Вопрос), read('да').
```

В этом случае система будет задавать вопросы пользователю относительно неизвестных фактов. Так как ответы пользователя не запоминаются, то возможно повторение вопросов. Для запоминания ответов можно воспользоваться встроенным предикатом **assert**.

Дальнейшие улучшение программы связано с упрощением формы записи правил. Запись правил в форме предикатов языка Пролог неудобна для пользователя. Представим правила в более естественной форме. Например:

```
правило1: если лампа(светится) и
                плита(холодная)
          то    нагреватель(неисправен).
```

Для представления правил в такой форме определим операторы:

```
определить_операторы: – op(920,  xfy, и),
                        op(950, xfx, то),
                        op(960, fx, если),
                        op(970, xfx, ':').
? – определить_операторы.
```

С помощью введенных операторов правило 1 представляется структурой, изображенной на рисунке 7.5.

Остальные правила рассматриваемого примера запишутся в виде:

```
правило 2:  если тока(нет)
            то    выключатель(не_включен).

правило 3:  если тока(нет)
            то    напряжения(нет).
```

правило 4: если плита(холодная) и
лампа(не_светится)
то тока(нет).

правило 5: если лампа(не_светится) и
плита(горячая)
то лампа(не_исправна).

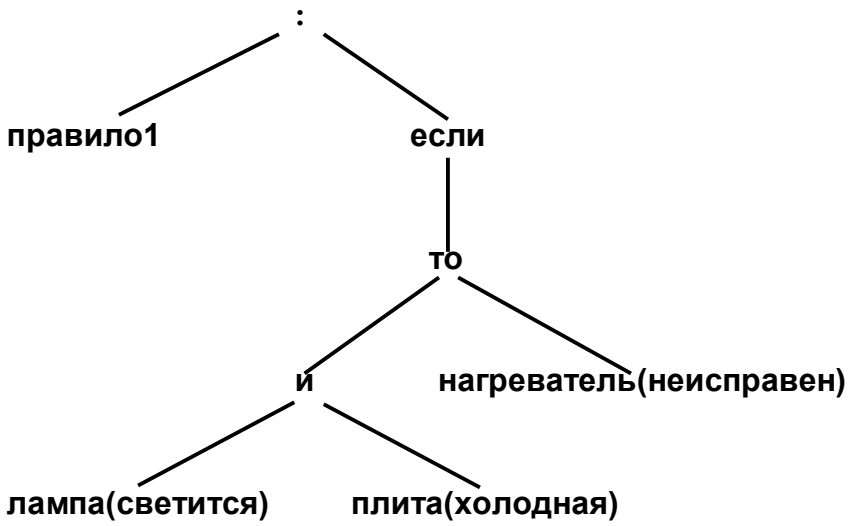


Рисунок 7.5 – Структура, соответствующая правилу

В ходе вывода метаинтерпретатор должен установить причину неисправности, проверяя различные гипотезы. Представим возможные гипотезы в форме:

Имя_факта : Факт.

Тогда в соответствии с введенными правилами база данных возможных причин неисправности (гипотез) запишется в виде:

h1 : причина(нагреватель(неисправен)).
h2 : причина(выключатель(не_включен)).
h3 : причина(напряжения(нет)).
h4 : причина(лампа(неисправна)).

Кроме этого, перечислим наблюдаемые признаки, на основе которых метаинтерпретатор должен установить причину неисправности.

h1 : признак(лампа(светится)).
h2 : признак(плита(холодная)).
h3 : признак(лампа(не_светится)).
h4 : признак(плита(горячая)).

Введение признаков позволит метаинтерпретатору задавать необходимый вопрос пользователю, если в базе данных нет значения соответствующего признака.

Реализуем метаинтерпретатор в виде предиката **найти(H)**, где **H** – возможная гипотеза, которую требуется подтвердить или опровергнуть. При этом возможны четыре случая:

1) гипотеза **H** подтверждается фактом, уже известным системе;
2) гипотеза **H** соответствует следствию одного из правил, тогда для подтверждения гипотезы необходимо доказать справедливость предпосылок правила;
3) гипотеза **H**, доказываемая на некотором шаге вывода, представляет собой конъюнкцию условий правила, т.е. **H1** и **H2**, тогда необходимо доказать достижимость конъюнкции подцелей: **найти(H1)** и **найти(H2)**;
4) гипотеза **H** сопоставима с одним из признаков, на основе которых устанавливаются причины неисправности, тогда необходимо задать соответствующий вопрос пользователю.

Для того чтобы исключить повтор вопросов, ответы пользователя будем запоминать в базе данных в виде фактов **сообщено(Факт, 'да|нет')**. Запоминание соответствующих фактов в базе данных будем выполнять с помощью предиката **assert**. Гипотезы, относительно которых можно задавать вопросы, определяются с помощью предиката **запрашиваемая(H)**. Такие гипотезы сопоставимы с признаками. С учетом сказанного, метаинтерпретатор опишется следующей совокупностью правил языка Пролог:

```
/*  обратный вывод  */
/*  вариант 1  */
найти(H): – Факт : H.                                  % случай 1
найти(H): – Правило : если H1 то H, найти(H1).         % случай2
найти(H1 и H2): – найти(H1), найти(H2).                % случай 3
найти(H): – запрашиваемая(H),                          % случай 4
              сообщено(H, да).                         % вопрос уже был
найти(H): – запрашиваемая(H),                          % случай 4
              not сообщено(H, _ ),                      % вопроса  не было
              спроси(H).
запрашиваемая(H): – Факт : признак(H).
спроси(H): – nl, write(H), write('?'),                 %  вопрос
               read(O), ответ(H, O).
ответ(H, да): – assert(сообщено(H, да)), !.            %  добавить в БД
ответ(H, no): – assert(сообщено(H, нет)), !, fail.     %  ответ
ответ(H, _ ): –  write('правильный ответ: да, нет'),
                   спроси(H).                          %  повторить вопрос
```

Вызов метаинтерпретатора с целью проверки всех гипотез выполняется следующим образом:

```
/*  ЭС для решения задачи диагностики  */
инициализация: – retractall(сообщено( _,_ )).

диагностика(Н):–
        инициализация,
        Факт : причина(Н),
        найти(Н).
```

Дальнейшее совершенствование разработанной программы связано с включением возможности ответов на вопросы типа “почему?” и “как?”. Возможный сценарий диалога с ЭС при ответе на вопрос “почему?” приведен в таблице 7.1.

Таблица 7.1 – Ответы на вопросы “почему?”

Вопрос или ответ системы	Ответ или вопрос пользователя
лампа(светится)?	*нет*
плита(холодная)?	*почему*
пытаюсь доказать тока(нет) с помощью правила: правило 4	
плита(холодная)?	*почему*
пытаюсь доказать выключатель(не_включен) с помощью правила: правило 2	
плита(холодная)?	*почему*
Моя гипотеза: выключатель(не_включен)	
плита(холодная)?	*почему*
Вы задаете слишком много вопросов	
Плита(холодная)?	*да*

Для того чтобы реализовать указанный диалог, необходимо переопределить предикат **найти(Н)**. Введем в этот предикат второй аргумент, который будет представлять стек из имен доказываемых гипотез и имен правил. Тогда, анализируя содержимое стека, можно будет организовать соответствующий диалог. В момент вызова предиката **найти** в стек помещается имя доказываемой гипотезы:

```
найти(Н, [Н]).
```

Так как при обратном выводе в Пролог-системе осуществляется поиск в глубину, то в стеке фиксируется путь, состоящий из имен правил, обеспечивающих подтверждение основной гипотезы. Введение второго аргумента в предикат **найти** требует следующих переопределений:

```
 /*  вариант 2  */
найти(Н, Стек): – Факт : Н.
найти(Н, Стек): – Правило: если Н1 то Н,
                  найти(Н1, [Правило | Стек]).
найти(Н1 и Н2, Стек): – найти(Н1, Стек), найти(Н2, Стек).
найти(Н, Стек): – запрашиваемая(Н), сообщено(Н, да).
найти(Н, Стек): – запрашиваемая(Н), not сообщено(Н, _ ),
                  спроси(Н, Стек).
```

В связи с необходимостью ввода вопроса “почему?” в предикаты **спроси** и **ответ** добавляется дополнительный аргумент:

```
спроси(Н, Стек): –  nl, write(Н), write('?'),
                    read(О), ответ(Н, О, Стек).
ответ(Н, да, Стек): – assert(сообщено(Н, да)), !.
ответ(Н, нет, Стек): – assert(сообщено(Н, нет)), !.
ответ(Н, почему, [ ]): – !, nl, write('Вы задаете слишком много вопросов'),
                        спроси(Н, [ ]).
ответ(Н, почему, [Н]): – !, nl, write('моя гипотеза :'),
                        write(Н), спроси(Н, [ ]).
ответ(Н, почему, [Правило | Стек]): – !,
                                    Правило : если Н1 то Н,
                                    nl, write('пытаюсь доказать'),
                                     write(Н),
                                    nl, write('с помощью правила : '),
                                    write(Правило),
                                    спроси(Н, Стек).
ответ(Н, _ , Стек): – write('правильный ответ : да, нет, почему'),
                      спроси(Н, Стек).
```

Формирование ответа на вопрос “как?” требует дополнительных изменений предиката **найти**. При ответе на вопрос “как?” система должна, по сути, построить дерево вывода. Например, если пользователя интересует, как была подтверждена гипотеза **выключатель(не_включен)**, то система должна обеспечить отображение дерева вывода, изображенного на рисунке 7.6.

Такое дерево можно представить в форме вложенных правил, условия и заключения которых конкретизированы:

```
правило 2 :  если
                    правило 4 : если сообщено(плита(холодная)) и
                                    сообщено(лампа(не_светится))
                                то тока(нет)
                то  выключатель(не_включен).
```

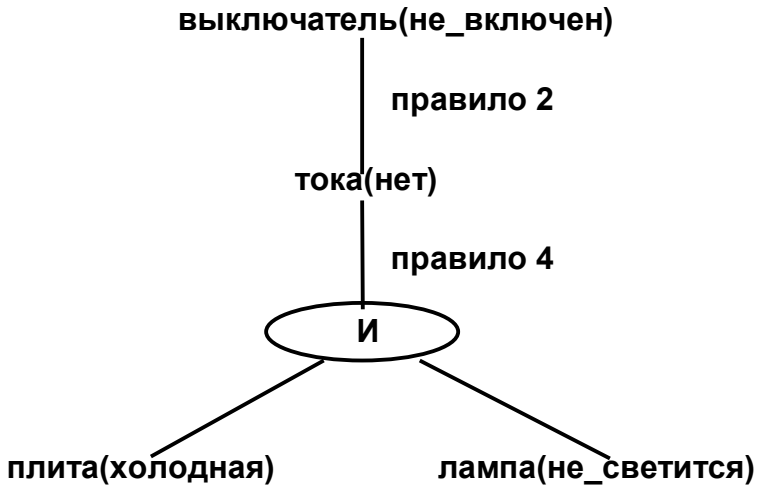


Рисунок 7.6 – Дерево вывода

Пусть **Д** обозначает дерево вывода целевого утверждения **Н**. Тогда построение дерева вывода выполним на основе следующих правил:

1) если **сообщено(Н)**, то **Д = сообщено(Н)**;
2) если имеется **Факт : Н**, то **Д = Факт : Н**;
3) если имеется правило **Правило : если Н1 то Н**, и если **Д1** – дерево вывода для **Н1**, то **Д = Правило : если Д1 то Н**;
4) если доказывается гипотеза **Н = Н1 и Н2**, то дерево вывода **Д = Дерево1 и Дерево2**, где **Дерево1** – дерево вывода гипотезы **Н1**, а **Дерево2** – гипотезы **Н2**.

Переопределим предикат **найти**, добавив третий аргумент, представляющий дерево вывода:

найти(Н, Стек, Дерево).

Воспользовавшись введенными выше правилами построения дерева вывода, получим:

```
/* правило1 */
найти(Н, Стек, сообщено(Н)):– сообщено(Н, да).
найти(Н, Стек, сообщено(Н)):– запрашиваемая(Н),
                              not сообщено(Н, _),
                              спроси(Н, Стек).
/* правило2 */
найти(Н, Стек, Факт : Н):– Факт : Н.

/* правило3 */
найти(Н, Стек, Правило : если Д1 то Н):–
        Правило : если Н1 то Н,
        Найти(Н1, [Правило|Стек], Д1).
```

```
/* правило4 */
найти(Н1 и Н2, Стек, Дерево1 и Дерево2):–
найти(Н1, Стек, Дерево1), найти(Н2, Стек, Дерево2).
```

Теперь можно будет выяснять у системы, как было получено то или иное заключение. Чтобы реализовать эту возможность, введем предикат **как(Н, Дерево)**, где **Н** – интересующее пользователя заключение, а **Дерево** – дерево вывода заключения, формируемое предикатом **найти**. Определяя предикат **как(Н, Дерево)**, учтем два случая:

```
% Дерево можно построить
как(Н, Дерево):– как1(Н, Дерево), !.

% Дерево нельзя построить
как(Н, _):– nl, write(Н), write( 'не доказано').
```

Здесь предикат **как1(Н, Дерево)** определяется с помощью правил, аналогичных правилам, используемых в определении предиката **найти**:

```
как1(Н, сообщено(Н)):– !,                          /* правило1 */
        nl, write(Н), write( 'было введено').

как1(Н, Факт : Н):– !,                             /* правило2 */
        nl, write(Н), write( 'является фактом'), write(Факт).

как1(Н, Правило : если _ то Н):– !,                /*правило3 */
        nl, write(Н), write( 'было доказано с помощью'),
        Правило : если Н1 то Н,
        отобрази_правило(Правило : если Н1 то Н).

как1(Н, Правило : если Дерево ):–                  /*правило4 */
        как ( Н, Дерево).

как1(Н, Дерево1 и Дерево2 то _):–                  /*правило5 */
        как ( Н, Дерево1); как ( Н, Дерево2).

отобрази_правило(Правило : если Н1 то Н):–
        nl, write(Правило), write( ':'),
        nl, write('если'), write(Н1),
        nl, write('то'), write(Н).
```

Вызов предиката **как(Н,Дерево)** будем выполнять из предиката **объясни(Дерево)**, после вызова предиката **найти(Н, [Н], Дерево)**:

```
/* ЭС для решения задач диагностики*/
инициализация:– retractall(сообщено(_,_)).
диагностика:– инициализация,
              Факт : причина(H),
              найти(H, [H], Дерево),
              nl, nl, write( 'решение:'), write(H),
              объясни(Дерево),
              возврат.
```

Здесь предикат **объясни(Дерево)** определяется так:

```
объясни(Дерево):–
        nl, nl, write( 'объяснить ? [цель/n]:'), read(H),
        (H\=no, !, как(H, Дерево), объясни(Дерево)), !.
```

Предикат возврат обеспечивает доказательство альтернативных гипотез:

```
возврат:–  nl, nl, write( ' следующее решение ?: '),
           read(no).
```

Процесс диагностики неисправностей начинается с вызова основного предиката **диагностика**. Ниже приведен возможный диалог с системой:

```
?-диагностика.
лампа(светится)? нет.
плита(холодная)? да.
лампа(не_светится)? да.
решение: выключатель(не_включен)
объяснить?[цель/n]: выключатель(не_включен).
выключатель(не_включен) было доказано с помощью
правило 2:  если тока(нет)
            то    выключатель(не_включен)
объяснить?[цель/n]: тока(нет).
тока(нет) было доказано с помощью
правило 4:  если плита(холодная) и
                  лампа(не_светится)
            то    тока(нет).
объяснить?[цель/n]: плита(холодная).
плита(холодная) было сообщено
объяснить?[цель/n]: нет.
решение: напряжения(нет)
объяснить?[цель/n]:нет.
плита(горячая)? почему.
пытаюсь доказать лампа(не_исправна)
с помощью правила: правило 5
плита(горячая)? нет.
нет.
```

Здесь ответы пользователя набраны курсивом.

Рассмотренная ЭС может работать только с такими утверждениями, которые либо истинны, либо ложны. Во многих случаях требуется учитывать неопределенность фактов и правил. Для работы в условиях неопределенности с правилами и фактами можно связать соответствующие коэффициенты уверенности. Например,

```
Правило1:  если лампа(светится) и
                плита(холодная)
           то   нагреватель(неисправен)  cf  0.8.
```

Здесь **cf** – инфиксный оператор, посредством которого задается коэффициент уверенности. Оператор **cf** определяется с помощью директивы:

```
:– op(940, xfx, cf).
```

Для обработки коэффициентов уверенности, назначенных фактам и правилам, можно использовать модель, описанную в § 4.2.2.2.

7.6.2. Представление знаний фреймами

В ЭС, основанных на правилах, знания сосредоточены исключительно в базе правил, а база данных представляет собой пассивное множество фактов. Фреймовая модель представления знаний обеспечивает лучшую структурированность фактов, более экономное использование памяти, а также активизацию фактов. Характерными отличиями фреймовой модели представления знаний являются: наследование по иерархии фреймов, наличие присоединенных процедур, наличие значений по умолчанию и др. (см. §3.5)

Рассмотрим простейшие возможности представления экспертных знаний совокупностью фреймов. При этом фрейм будем представлять в виде объекта с множеством атрибутов (слотов):

```
объект
  – –  атрибут1 :  значение1
  – –  атрибут2 :  значение2
  – –  …
  – –  атрибут_n :  значение_n.
```

Если рассматривать знаки “**– –**” и “**:**” как операторы языка Пролог (**op(910, xfx,:), op(910, xfy, – –)**), то такому описанию объекта будет соответствовать структура, изображенная на рисунке 7.7. Данная структура представляет терм языка Пролог (n = 3):

объект – – ((атр1 : з1) – – ((атр2 : з2) – – (атр3 : з3))).

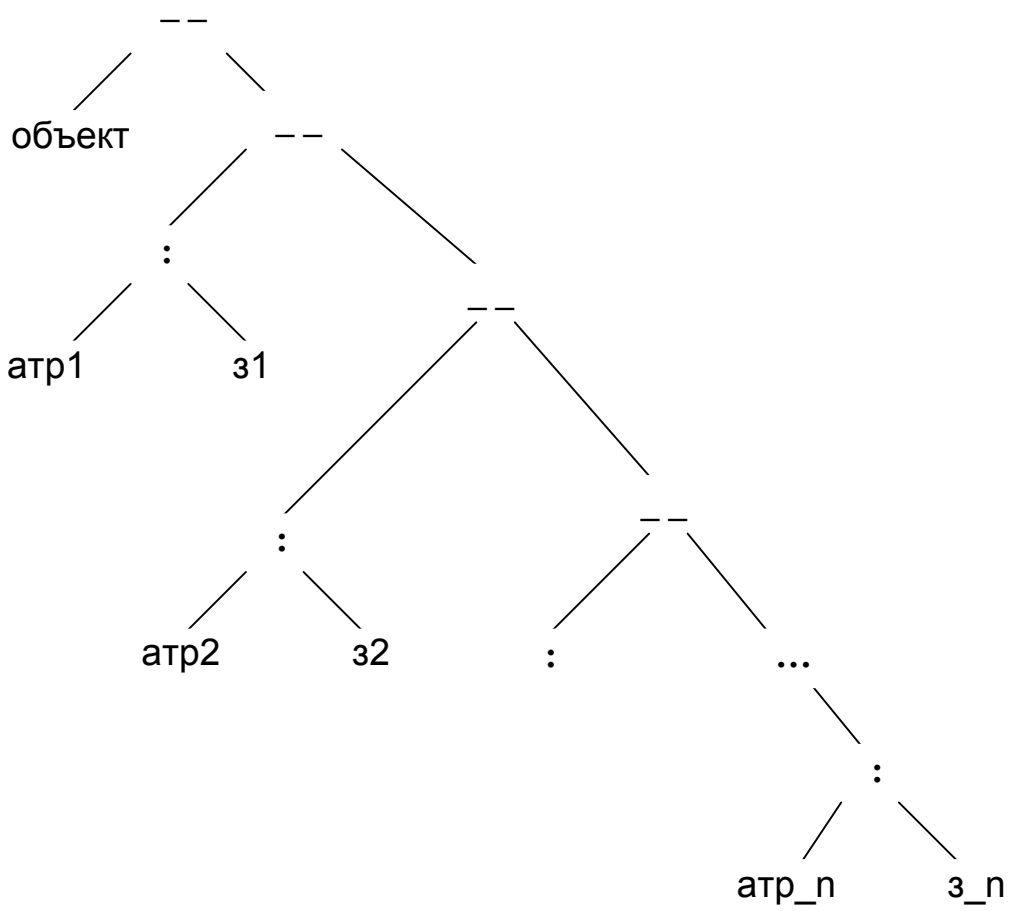


Рисунок 7.7 – Структура, соответствующая списку свойств

Доступ к значениям атрибутов объекта будем осуществлять с помощью предиката **объект(О,А,З)**, где **О** – переменная, обозначающая объект; **А** – соответствующий атрибут объекта; **З** – значение атрибута. Предикат **объект(О,А,З)** определяется рекурсивно:

```
объект(О, А, З):– О – – Атрибуты,
                  поиск(А, З, Атрибуты),!.
поиск(А, З, А : З).                          % утверждение 1
поиск(А, З, А : З – – Атрибуты1).            % утверждение 2
поиск(А, З, А1 : З1 – – Атрибуты1):–         % утверждение 3
                  поиск(А, З, Атрибуты1).
```

Здесь первое определение обеспечивает отделение имени объекта от его атрибутов с помощью оператора **О – – Атрибуты**. Переменная **Атрибуты** после выполнения данного оператора будет представлять правое поддерево древовидной структуры, изображенной на рисунке 7.7. Предикат

поиск(А, З, Атрибуты) осуществляет поиск значения атрибута **А** в поддереве **Атрибуты**. Поиск выполняется рекурсивно и основывается на утверждениях:

1) атрибут **А** со значением **З** входит в поддерево **Атрибуты**, если оно представлено элементарным поддеревом **А : З**;
2) атрибут **А** со значением **З** входит в поддерево **Атрибуты**, если его левое поддерево – элементарное поддерево **А : З**;
3) атрибут **А** со значением **З** входит в поддерево **Атрибуты**, если указанные элементы входят в его правое поддерево, т.е. в **Атрибуты1**.

Воспользуемся введенным представлением фрейма и построим простейшую ЭС, устанавливающую причину неисправности автомобиля. При этом в ходе построения ЭС будут использованы: иерархия агрегатов автомобиля, физическая обоснованность функционирования того или иного агрегата, гипотезы, ответы пользователя (тесты), эвристики.

На рисунке 7.8 изображен упрощенный пример дерева диагностики автомобиля. Предполагается, что автомобиль состоит из следующих основных составных частей: двигателя, шасси, электрооборудования.

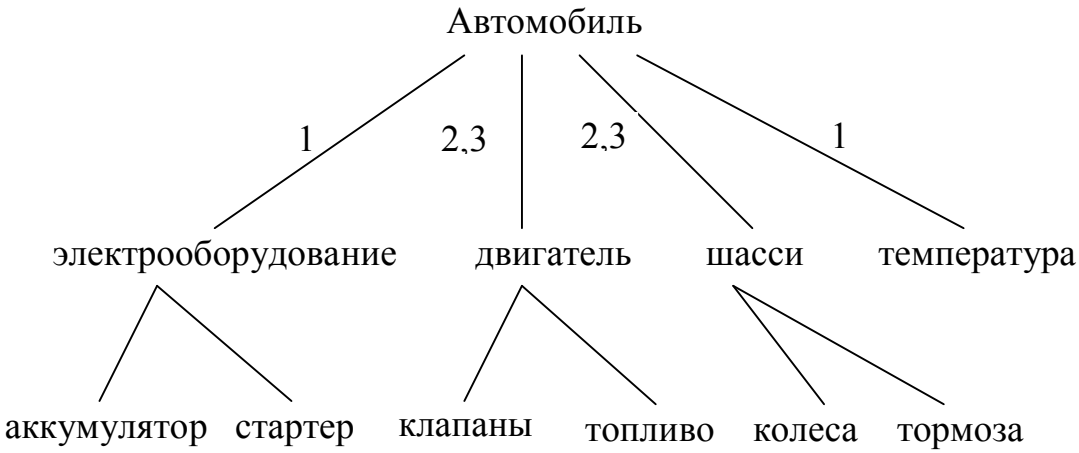


Рисунок 7.8 – Дерево диагностики автомобиля

В свою очередь, в состав шасси входят колеса и тормоза, а в состав электрооборудования – аккумулятор и стартер. Автомобиль неисправен, если неисправна одна из его составных частей. Аналогично любой агрегат автомобиля считается неисправным, если неисправен один из узлов агрегата. Например, шасси неисправно, если имеются проблемы с колесами или тормозами. Двигатель неисправен, если не отрегулированы клапаны или не поступает топливо (физическая обоснованность). В качестве эври-

стической причины неисправности автомобиля может выступать низкая температура (например, ниже -20^{o}).

Представим вершины и листья дерева диагностики в виде следующей структуры:

объект
– – вопрос : Q
– – ответы : A
– – причины : R
– – объяснения : E.

Здесь **объект** – имя, соответствующее вершине или листу дерева; **Q** – вопрос, задаваемый пользователю, относительно состояния объекта;
A = [1: A1, 2: A2, …] – список возможных ответов; **R** – список возможных причин неисправности; **E** – текст, объясняющий причину неисправности. Список **R** может быть представлен в одной из форм:

R = [i : Hi , j : Hj , …] или
R = [H1 , H2 , …].

Первая форма используется, когда **Hi** выступает в качестве возможного обоснования гипотезы **H**, в подтверждение которой был выбран ответ **Ai** из списка **A**. Вторая форма применяется, когда **H1, H2**, … являются возможными обоснованиями гипотезы **H**, но вопросы относительно **H** не формулировались.

В описании объекта атрибуты **вопрос** и **ответы** являются не обязательными, атрибут **причины** обязателен, атрибут **объяснения** обязателен только для объектов, представляющих листья дерева диагностики.

Приведем примеры описаний некоторых объектов на языке Пролог:

```
/*  база знаний  */

автомобиль
– – вопрос    : 'Что случилось Вашим автомобилем?'
– – ответы    : [1 : 'не заводится', 2 : 'не едет', 3 : другое]
– – причины   : [1 : температура ,1 : электрооборудование,
                 2 : двигатель, 2 : шасси,
                 3 : двигатель, 3 : шасси].

температура
– – вопрос    : 'Температура ниже –20?°'
– – ответы    : [1 : да, 2 : нет]
– – причины   : [1 : верно]
– – объяснения : 'Низкая температура. Нагрейте двигатель'.
```

```
топливо
– – вопрос        : 'Есть топливо в баке?'
– – ответы        :  [1 : да, 2 : нет]
– – причины       :  [2 : верно]
– – объяснения  : 'Нет бензина. Заправьте автомобиль'.
```

Описав подобным образом все вершины и листья дерева диагностики, можно приступить к поиску причины неисправности. Поиск выполняется методом в глубину от корня дерева:

```
найти(H, X): –
   (объект(H, вопрос, _),!, получить_ответ(H,O),
                              найти1(H,O,X).

найти1(H,O,X): –
     (объект(H, причины, R),
     (var(O),!,member(H1, R); member(O : H1, R)),
      найти2(H, H1, X).

найти2(H, верно, H).
найти2(H, H1, X): – найти(H1, X).

получить_ответ(Объект, O):- сообщено(Объект,O),!.
получить_ответ(Объект, O):-спроси(Объект,O),
                                 assert(сообщено(Объект,O)),!.
спроси(Объект,Ответ):-
        объект(Объект,вопрос, Вопрос),
        объект(Объект,ответы, Ответы),
        nl,write(Вопрос),
        печать_списка(Ответы), nl,read(Ответ),!.

        печать_списка([ ]).
        печать_списка([H|T]):- nl, write(H), печать_списка(T).
```

При вызове предиката **найти(H,X)** переменная **H** конкретизируется константой **автомобиль**. Предикат **получить_ответ(H,O)** обеспечивает ввод ответа **O** пользователя на вопрос, связанный с объектом **H**, и запоминание его в базе данных. Полученный ответ обрабатывается с помощью предиката **найти1(H,O,X)**. Данный предикат извлекает список **R**, элементами которого являются возможные обоснования (причины) гипотезы **H**. Из списка **R** выбирается с помощью предиката **member** обоснование **H1**, которое выступает в качестве новой подцели поиска. Если ответ **O** представляет собой не конкретизированную переменную, что проверяется вызовом **var(O)**, то это означает, что вопросы относительно **H** не формулировались.

Дальнейший поиск выполняется с помощью предиката **найти2(H,H1,X)**. Если подцель **H1** конкретизирована значением '**верно**', то по-

иск заканчивается, иначе рекурсивно вызывается предикат **найти(H1,X)**, и весь процесс повторяется. В результате поиска возвращается значение переменной **X**, которое фиксирует возможную причину неисправности. Основной предикат, обеспечивающий запуск ЭС, можно определить следующим образом:

```
/*   ЭС, основанная на фреймовой модели   */

инициализация:- retractall(сообщено( _ , _ )), !.
причина(автомобиль).

диагностика: –
        инициализация, причина(H),          /*  гипотеза  */
        найти(H, X),
        объект(X, объяснения, E),
        nl, write(E),
        возврат.
диагностика: – nl, write('Больше нет гипотез').
возврат: – nl, nl, write('Следующее решение ?:'),
            read(no).
```

Для приведенного фрагмента базы знаний можно получить следующее решение:

```
?- диагностика.
Что случилось с Вашим автомобилем?
1: не заводится
2: не едет
3: другое
1.
Температура ниже –20 ?
1: да
2: нет
1.
Низкая температура. Нагрейте двигатель.
Следующее решение? да.
Больше нет гипотез.
```

Рассмотренная ЭС является простейшей, и она не использует в полной мере достоинства фреймовой модели представления знаний. В частности, иерархическая упорядоченность объектов предметной области здесь фиксировалась косвенным способом, посредством значений списка возможных обоснований **R**. В то же время фреймовая модель позволяет делать это явно с помощью слотов **is_a** или **ako**. Кроме этого, в примере не использовались присоединенные процедуры и значения по умолчанию.

Особенности реализации этих возможностей на языке Пролог рассмотрены в § 6.11.5.

Вопросы для самопроверки

1. Приведите определение экспертной системы.
2. Назовите основные отличительные свойства экспертных систем.
3. Перечислите типовые задачи, решаемые с помощью ЭС.
4. Назовите основные компоненты ЭС и объясните их функции
5. Каким требованиям должны удовлетворять задачи, для решения которых планируется разработка ЭС?
6. Какие специалисты принимают участие в разработке ЭС, и каковы их функции?
7. В чем заключается концепция создания “быстрого прототипа”?
8. Назовите основные этапы разработки ЭС, перечислите задачи, решаемые на каждом из этапов.
9. Объясните термины “приобретение знаний” и “извлечение знаний”.
10. Сформулируйте основные подходы, используемые в процессе извлечения знаний.
11. Какие приемы использует когнитолог в ходе извлечения знаний?
12. Какая проблема исследуется в рамках направления “формирование знаний”?
13. Сформулируйте обобщенный алгоритм прямого вывода на правилах-продукциях.
14. Объясните на примере, что понимают под частичным сопоставлением.
15. Объясните основную идею алгоритма RETE.
16. Объясните суть вопросов типов “почему” и “как”, используемых для формирования объяснений вывода в ЭС.
17. Объясните на примере механизм формирования ответа на вопрос типа “как”.
18. Объясните на примере механизм формирования ответа на вопрос типа “почему”.
19. Как можно представить базу продукционных правил на языке Лисп?
20. Какие правила положены в основу лисповской функции BACK-PROVE, осуществляющей обратный вывод?
21. Как можно представить базу продукционных правил на языке Пролог?
22. Какие правила положены в основу предиката **найти(H)**, осуществляющего обратный вывод на множестве продукционных правил?
23. Напишите на языке Пролог простейший вариант реализации предиката **найти(H)**.

ГЛАВА 8
РАСПОЗНАВАНИЕ ОБРАЗОВ И ОБУЧЕНИЕ

Распознавание образов – одна из основных функций интеллектуальных систем. Любой интеллект, в том числе и искусственный, начинается с восприятия и распознавания объектов внешнего мира. Поэтому подсистемы распознавания образов называют “ушами и глазами” СИИ [55]. Существует большое множество задач ИИ, решение которых связано с автоматизацией процессов восприятия и распознавания образов. Например, автоматическое чтение рукописных текстов, распознавание речи, анализ изображений и сцен, дистанционная идентификация объектов, медицинская диагностика и др.

В главе изложены общие принципы распознавания образов, не учитывающие специфику конкретных задач, обусловленную физической природой восприятия. Особенности восприятия и распознавания речевых и зрительных образов рассматриваются в следующих главах.

Распознавание образов неразрывно связано с обучением, которое осуществляется на основе анализа и обобщения имеющихся данных. Поэтому значительная часть главы посвящена рассмотрению обучаемых классификаторов образов. Большое внимание при этом уделяется коннекционистским моделям обучения, к которым относят нейронные сети. Нейронные сети находят широкое применение при решении многих задач искусственного интеллекта. В главе подробно рассматриваются модели нейронных элементов, структуры нейронных сетей и основные алгоритмы их обучения.

Если распознаваемые объекты представляются с помощью системы примитивов и отношений между ними, то их распознавание может выполняться на основе синтаксического (структурного) подхода, использующего теорию формальных грамматик. В этом случае обучение рассматривается как процесс построения грамматики. Структурный метод распознавания рассматривается в заключительном параграфе главы.

8.1. Основные сведения о распознавании образов

Способность воспринимать образы – одно из основных свойств мозга. Воспринимая внешний мир, человек опознает окружающие его объекты и в соответствии с этим совершает определенные действия. Важно отметить, что воспринимаемые объекты разбиваются на группы похожих, но не тождественных объектов.

К одной группе относятся те объекты, которые обладают характерным набором свойств. Множество объектов, обладающих общими свойствами, называют *образом*. Объекты, относящиеся к одному образу (классу), могут отличаться второстепенными, не существенными с точки зрения решаемой задачи, свойствами.

Каждый образ (класс) представляется некоторым количеством объектов (экземпляров). Совокупность всех различных проявлений образа образует множество его возможных реализаций.

Выбор исходного описания образов – важнейшая задача, решаемая при построении систем распознавания образов. При введении понятия образа указывалось, что в образ объединяются объекты, имеющие общие свойства. Указанные свойства называют *признаками* образа (класса). Отметим диалектическое единство понятий образ и признак. Задание образов порождает признаки, а задание признаков порождает образы [10]. Для выделения признаков сигналы, получаемые от различных рецепторов (датчиков), подвергают предварительным преобразованиям.

Таким образом, распознавание образов можно определить как отнесение наблюдаемых данных к определенному классу на основе выделения существенных признаков, характеризующих эти данные. Этому определению соответствует обобщенная структурная схема, изображенная на рисунке 8.1.



Рисунок 8.1 – Обобщенная схема системы распознавания

Если предположить, что система имеет n рецепторов (датчиков), то любой объект может быть представлен вектором n – мерного пространства, которое называют *исходным пространством рецепторов*. Во многих случаях непосредственно по сигналам, действующим на выходе рецепто-

ров, невозможно выполнить классификацию объектов. Поэтому из сигналов рецепторов выделяют необходимые признаки, которые формируют описание объекта в *пространстве признаков*. С помощью полученного описания выполняется распознавание объекта, т.е. отнесение его к тому или иному классу. Возможность разделения образов в пространстве признаков основана на *гипотезе компактности*, которая формулируется так: если некоторые множества объектов представляют собой образы, то обязательно существует такое пространство признаков, в котором этим объектам соответствуют компактные множества точек [10]. Следовательно, решение задачи распознавания образов в значительной степени зависит от выбора множества признаков, обеспечивающих компактное представление объектов одного класса в пространстве признаков. Если указанные признаки выделены, то задачу распознавания образов можно рассматривать как задачу построения разделяющей поверхности, отделяющей одно компактное множество точек от другого.

В тех случаях, когда имеющиеся априорные данные о распознаваемых объектах недостаточно полны, чтобы по ним найти определенную разделяющую поверхность, применяют обучаемые системы распознавания образов. Обучение осуществляется путем показа примеров с указанием принадлежности объектов тому или иному классу (обучение с учителем) или без указания (обучение без учителя). В ходе обучения система определяет множество признаков, на основе которых можно классифицировать объекты. Важно отметить, что после обучения система должна приобрести способность к опознаванию объектов не только из обучающей последовательности, но и тех, которые ей не предъявлялись.

8.2. Геометрический метод распознавания

Геометрический метод распознавания основан на использовании функции расстояния в качестве меры сходства векторов признаков, представляющих образы.

На рисунке 8.2 изображены векторы признаков, объединенные в два класса k_1 и k_2. Для наглядности использованы только два признака x_1 и x_2. Так как вектор $\vec{x}$, представляющий некоторый объект, находится ближе к векторам класса k_1, то объект принадлежит классу k_1.

Интуитивно ясно, что результаты классификации с помощью функции расстояния будут удовлетворительными, если выполняются два условия. Во-первых, соблюдаются условия компактности, выражающееся в том, что точки, представляющие объекты одного класса, расположены друг к другу ближе, чем к точкам, представляющим объекты другого клас-

са. Во-вторых, классы не пересекаются между собой, т.е. между ними можно провести разделяющую кривую (поверхность).

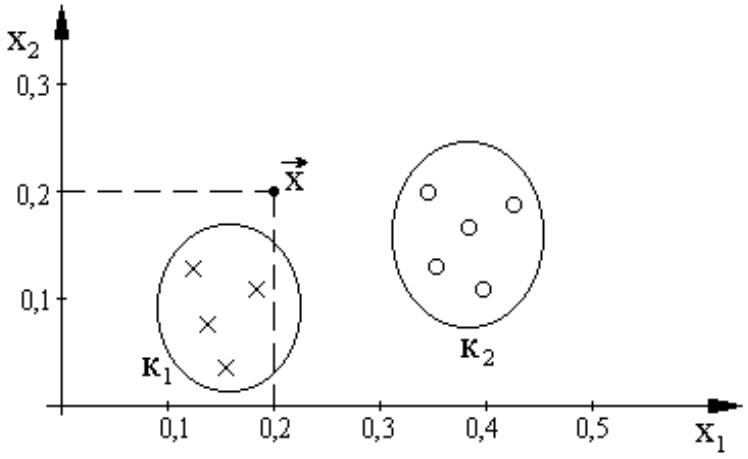


Рисунок 8.2 – Классификация объектов с помощью функции расстояния

Пусть имеется M классов, каждый из которых представляется с помощью эталонного вектора $\vec{z}_1, \vec{z}_2, ..., \vec{z}_M$. В качестве меры близости вектора $\vec{x}$ и соответствующего эталона может быть использовано расстояние Евклида [43]

$$D_i = \left\| \vec{x} - \vec{z}_i \right\|, \quad i = 1, 2, \ldots, \text{M} \tag{8.1}$$

Возведем (8.1) в квадрат

$$D_i^2 = (\vec{x} - \vec{z}_i)^T (\vec{x} - \vec{z}_i) = \vec{x}^T \vec{x} - 2\vec{x}^T \vec{z}_i + \vec{z}_i^T \vec{z}_i = \vec{x}^T \vec{x} - 2(\vec{x}^T \vec{z}_i - \frac{1}{2} \vec{z}_i^T \vec{z}_i), \tag{8.2}$$

где символ *T* обозначает транспонирование.

Объект, представленный вектором $\vec{x}$, относится к тому классу, который ближе к нему, т.е. классификация осуществляется по минимуму расстояния D_i. Из (8.2) следует, что выбор минимального расстояния D_i эквивалентен выбору максимального значения разности

$$d_i(\vec{x}) = \vec{x}^T \vec{z}_i - \frac{1}{2} \vec{z}_i^T \vec{z}_i \ , \tag{8.3}$$

которое называют *решающей функцией*.

Если $d_i(\vec{x}) > d_j(\vec{x})$ для всех $i \neq j$, $j = 1, 2, \ldots, \text{M}$, то вектор $\vec{x}$ относится к классу k_i. Отметим, что $d_i(\vec{x})$ представляет собой гиперплоскость в *n*–мерном пространстве. При *n*=2 $d_i(\vec{x})$ будет представлять прямую линию, отделяющую один класс от другого.

Если каждый класс характеризуется не единственным, а несколькими эталонами $\vec{z}_i^1, \vec{z}_i^2, ..., \vec{z}_i^N$, где N – количество эталонов, определяющих класс, то

$$D_i = \min_l \left\| \vec{x} - z_i^l \right\|, \quad l = 1, 2, \ldots, N.$$

В этом случае для каждого входного вектора $\vec{x}$ вычисляется расстояние D_i. Вектор $\vec{x}$ относится к классу k_i, если $D_i < D_j$ для всех $j \neq i$.

Отметим, что в качестве меры сходства, кроме расстояния Евклида, могут использоваться и другие метрические расстояния, обладающие следующими свойствами:

а) $D(\vec{x}, \vec{y}) \geq 0$;

б) $D(\vec{x}, \vec{y}) = D(\vec{y}, \vec{x})$ (симметрия);

в) $D(\vec{x}, \vec{y}) \leq D(\vec{x}, \vec{z}) + D(\vec{z}, \vec{y})$ (неравенство треугольника), где $D(\vec{x}, \vec{y})$ – расстояние между векторами $\vec{x}$ и $\vec{y}$.

Если известны статистические характеристики входных векторов, то в качестве меры сходства может быть использовано расстояние Махаланобиса

$$D(\vec{x}, \vec{m}) = (\vec{x} - \vec{m})^T C^{-1} (\vec{x} - \vec{m}), \tag{8.4}$$

где C – ковариационная матрица совокупности векторов признаков; m – вектор средних значений.

Нередки случаи, когда распознаваемые объекты представляются бинарными кодовыми последовательностями или строками символов. В этом случае в качестве меры сходства используют расстояние Хемминга и расстояние Левенштейна.

Рассмотрим два вектора $\vec{x}$ и $\vec{y}$, элементами которых являются бинарные значения 0 и 1 или символы некоторого алфавита. Расстояние Хемминга D_H между такими векторами равно количеству несовпадающих элементов векторов. Например, пусть $\vec{x}$ = (1, 0, 1, 1, 0) и $\vec{y}$ = (0, 1, 1, 1, 0). Тогда $D_H(\vec{x}, \vec{y}) = 2$. Или, пусть $\vec{x}$=(о, б, р, а, з, е, ц) и $\vec{y}$=(о, б, р, а, т, н, о). В этом случае $D_H(\vec{x}, \vec{y}) = 3$. Заметим, что расстояние Хемминга может быть определено только между векторами (строками) $\vec{x}$ и $\vec{y}$, равной длины.

Расстояние Левенштейна между строками X и Y определяется как наименьшее число преобразований, требуемых для получения строки Y из строки X. Условно это можно записать в виде выражения [73]

$$D_L(X,Y) = \min\{a(i) + b(i) + c(i)\}, \tag{8.4}$$

где $a(i)$ – количество замен символов при выполнении i -го варианта преобразований символов; $b(i)$ – количество вставок символов; $c(i)$ – количество удалений.

Существует большое число вариантов преобразования строки X в строку Y, определяемое возможными комбинациями операций замены, вставки и удаления. Поиск варианта преобразования X в Y, обеспечивающего минимальное число указанных операций, выполняется на основе метода динамического программирования. Так как в ходе распознавания символы строк могут искажаться с различной вероятностью, то часто рассматривают взвешенное расстояние Левенштейна

$$D_{WL}(X,Y) = \min\{pa(i) + qb(i) + rc(i)\},$$

где $p(c_1,c_2)$ – стоимость замены символа c_1 на c_2; $q(c_1)$ – стоимость вставки символа c_1; $r(c_1)$ – стоимость удаления символа c_1.

Взвешенное расстояние Левенштейна между строками $X = (x_1, x_2, ..., x_n)$ и $Y = (y_1, y_2, ..., y_m)$ вычисляется на основе алгоритма динамического программирования:

```
D [ 0, 0 ] : = 0;
For  i : = 1  To  n  Do
        D [ i, 0 ] : = D [ i - 1, 0 ] + r ( X [ i ] );
For  j : = 1  To  m  Do
        D [ 0, j ] : = D [ 0, j – 1 ] + q ( Y [ j ] );
For  i : = 1  To  n  Do
  For  j : = 1  To  m  Do
     Begin
        p1 : = D [ i – 1, j – 1 ] + p ( X [ i ], Y [ j ] );
        p2 : = D [ i, j – 1 ] + q ( Y [ j ] );
        p3 : = D [ i – 1, j ] + r ( X [ i ] );
        D [ i, j ] : = min ( p1, p2, p3 )
     End
        DWL : = D [ n, m ];
```

Простой пример вычисления расстояния Левенштейна показан на рисунке 8.3. Здесь полагается, что стоимости элементарных операций вставки и удаления символов равны 1. Стоимость операции замены символа равна 1, если сравниваемые символы строк различны. Если сравнивае-

мые символы строк одинаковы, то замена не выполняется, и стоимость операции равна 0.

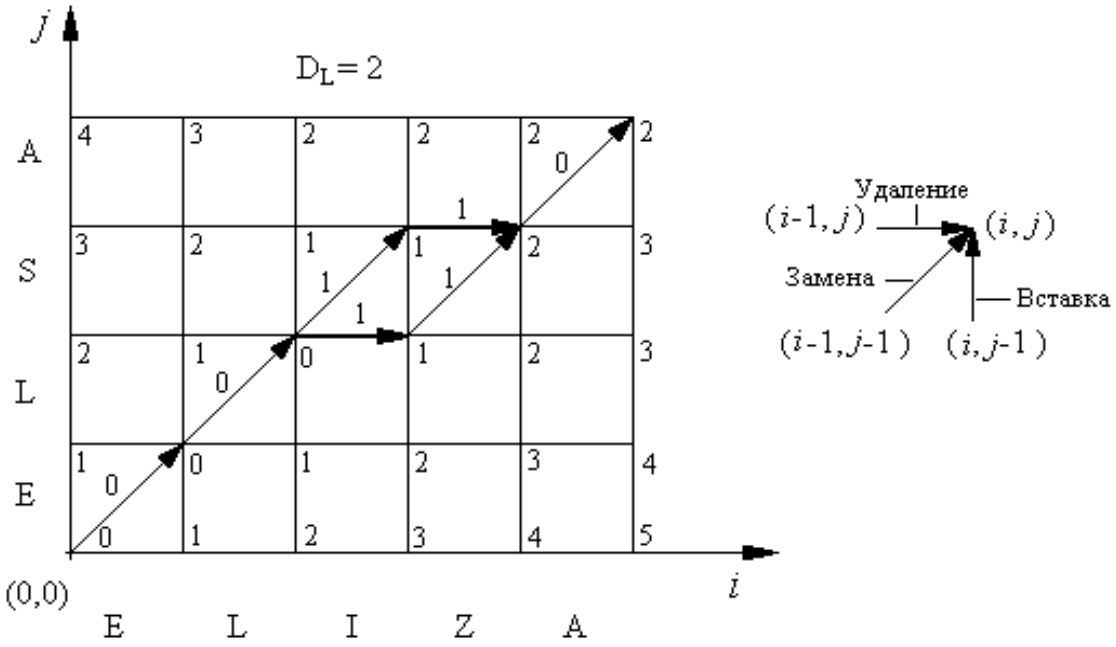


Рисунок 8.3 – Преобразование строки “ELIZA” в строку “ELSA”

Алгоритм динамического программирования позволяет построить оптимальную траекторию преобразования строк, выбирая на каждом шаге элементарное преобразование, обеспечивающее минимум текущего значения расстояния Левенштейна (на рисунке 8.3 указанные расстояния обозначены числами, находящимися в узлах координатной сетки).

Возвращаясь к геометрическому методу распознавания, отметим, что его применение требует задания эталонов классов, которые могут быть определены в результате решения задачи кластеризации. *Кластером* называют группу объектов, образующих в пространстве признаков компактную область. Выявление кластеров на множестве исходных данных также основано на использовании мер сходства. Рассмотрим алгоритм поиска кластеров для случая, когда в качестве меры сходства используется расстояние Евклида.

Решение задачи выделения кластеров требует введения критерия качества кластеризации. Наиболее часто используют сумму квадратов отклонений входных векторов $\vec{x}$ от центров кластеров $\vec{m}_j$ [43]

$$J = \sum_{j=1}^{N_c} \sum_{x \in s_j} \left\| \vec{x} - \vec{m}_j \right\|^2 , \qquad (8.5)$$

где s_j – множество входных векторов, относящихся к j–му кластеру; N_c – количество кластеров; $\vec{m}_j = (1/N_j)\sum_{x \in s_j} \vec{x}$ – вектор средних значений для множества s_j.

Рассмотрим алгоритм кластеризации, минимизирующий критерий (8.5). Алгоритм основан на вычислении K внутригрупповых средних и соответственно называется *алгоритмом K средних*. Процедура вычислений предусматривает выполнение следующих шагов [43].

Шаг 1. Фиксируют число кластеров K с центрами $\vec{z}_i(1), \vec{z}_2(1), ..., \vec{z}_k(1)$. Обычно в качестве центров используются K первых входных векторов.

Шаг 2. На r–ой итерации входной вектор $\vec{x}$ относится к кластеру j, если удовлетворяется следующее неравенство

$$\left\| \vec{x} - \vec{z}_j(r) \right\| < \left\| x - \vec{z}_i(r) \right\| , \quad \forall j \neq i ,$$

где $i = 1, 2, \ldots, k$.

Шаг 3. Определяются новые центры кластеров $\vec{z}_j(r+1)$ таким образом, чтобы минимизировать критерий (8.5), т.е.

$$J_j = \sum_{\vec{x} \in s_j(r)} \left\| \vec{x} - \vec{z}_j(r+1) \right\|^2 , \quad j = 1, 2, \ldots, k.$$

Минимум J_j обеспечивается лишь при одном $z_j(r+1)$, равном среднему арифметическому векторов $\vec{x}$, принадлежащих множеству $s_j(r)$. Тогда

$$\vec{z}_j(r+1) = \frac{1}{N_j} \sum_{\vec{x} \in s_j(r)} \vec{x} ,$$

где N_j – число векторов $\vec{x}$, входящих во множество $s_j(r)$. Таким образом, для определения новых значений центров кластеров требуется вычислить K средних значений. Отсюда и происходит название алгоритма.

Шаг 4. Если $z_i(r+1) \approx z_i(r)$, то выполнение алгоритма заканчивается, иначе повторяется шаг 2.

Выделенные центры кластеров или сами входные векторы, образующие соответствующий кластер, могут в дальнейшем использоваться в качестве эталонов класса.

Существуют и другие алгоритмы кластеризации. Например, алгоритм ISODATA [43].

Формирование эталонов на основе решения задачи кластеризации можно рассматривать как один из подходов к обучению при распознавании образов.

Общая идея, лежащая в основе классификации по минимуму расстояния, предполагает, что все точки, представляющие объекты одного класса в пространстве признаков, расположены близко друг к другу, а точки различных классов удалены друг от друга на некоторое расстояние. В этом случае улучшить результаты классификации можно, если минимизировать расстояния между точками одного класса и максимизировать расстояния между точками разных классов. С этой целью могут выполняться сложные линейные или нелинейные преобразования, деформирующие пространства признаков [26]. Из-за сложности вычислений геометрические методы распознавания используют в основном для иллюстрации других методов

8.3. Байесовский метод распознавания

Байесовский метод распознавания объектов предусматривает построение решающих правил на основе статистических свойств классов. Пусть $\vec{x}$ – вектор признаков, соответствующий k_i классу. Априорная вероятность реализации класса k_i равна $P(k_i)$. Задача состоит в том, чтобы отнести входной вектор $\vec{x}$ к одному из известных классов с минимальной ошибкой. Считается, что система распознавания допускает ошибку, если относит к классу k_j объект, на самом деле принадлежащий классу k_i.

Если в результате измерений определить $p(\vec{x} \mid k_i)$ – плотность распределения вектора $\vec{x}$ при условии, что он принадлежит классу k_i, то в соответствии с теоремой Байеса

$$p(k_i \mid \vec{x}) = \frac{p(\vec{x} \mid k_i) P(k_i)}{\sum_{j=1}^{M} p(\vec{x} \mid k_j) P(k_j)}, \qquad (8.6)$$

где M – число классов. Выражение (8.6) определяет вероятность того, что наблюдаемый вектор $\vec{x}$ принадлежит классу k_i.

Анализируя значения $p(k_i \mid \vec{x})$, можно вынести решение об отнесении наблюдаемого вектора к тому или иному классу. Решение об отнесении вектора $\vec{x}$ к классу k_i можно считать оправданным, если для любого j выполняется условие:

$$p(k_i \mid \vec{x}) \geq p(k_j \mid \vec{x}), \qquad \forall j \neq i. \qquad (8.7)$$

Данное правило называют правилом *максимума апостериорной вероятности*.

Из (8.6) и (8.7) следует байесовское решающее правило

$$p(\vec{x} \mid k_i)P(k_i) \geq p(\vec{x} \mid k_j)P(k_j), \ \forall j \neq i . \tag{8.8}$$

В частном случае, когда априорные вероятности реализации классов равны (т.е. $P(k_i) = P(k_j)$), решающее правило записывается в виде

$$p(\vec{x} \mid k_i) \geq p(\vec{x} \mid k_j), \quad \forall j \neq i \tag{8.9}$$

или

$$\frac{p(\vec{x} \mid k_i)}{p(\vec{x} \mid k_j)} \geq 1, \quad \forall j \neq i . \tag{8.10}$$

Функцию $p(\vec{x} \mid k_i)$ называют *функцией правдоподобия*, а правило (8.10) – правилом *максимального правдоподобия*.

Введем в рассмотрение функцию потерь $c(k_i, k_j)$, которая характеризует потери (риск), возникающие при ошибочном отнесении объекта, принадлежащего к классу k_i, к классу k_j. Математическое ожидание потерь, связанных с отнесением вектора $\vec{x}$ к классу k_j, определится из выражения:

$$R_j(\vec{x}) = \sum_{i=1}^{M} c(k_i, k_j) p(k_i \mid \vec{x}) . \tag{8.11}$$

Величину $R_j(\vec{x})$ называют *условным средним риском* или условными средними потерями. Общие средние потери при выборе класса k_j равны

$$R_j = \int_X R_j(\vec{x}) \, p(\vec{x}) d\vec{x} , \tag{8.12}$$

где X – область определения $\vec{x}$.

Классификатор, минимизирующий общие средние потери, называется байесовским классификатором. Покажем, что классификация на основе правила (8.8) обеспечивает минимизацию общих средних потерь, называемых *средним риском*.

Минимизация среднего риска сводится к минимизации $R_j(\vec{x})$. Рассмотрим частный вид функции потерь

$$c(k_i, k_j) = 1 - \delta_{ij}\,, \tag{8.13}$$

где $\delta_{ij} = 1$ при $i=j$, и $\delta_{ij} = 0$ при $i \neq j$. Такое определение функции потерь устанавливает потери, равные нулю при правильной классификации и единице при ошибочной классификации. Отметим, что в этом случае потери одинаковы при любых неправильных классификациях.

Подставив (8.13) в (8.11), получим

$$R_j(\vec{x}) = \sum_{i=1}^{M}(1-\delta_{ij})p(k_i \mid \vec{x}) = \sum_{i=1}^{M} p(k_i \mid \vec{x}) - \sum_{i=1}^{M}\delta_{ij}p(k_i \mid \vec{x}) = 1 - p(k_j \mid \vec{x})\,. \tag{8.14}$$

Минимум $R_j(\vec{x})$ достигается в том случае, когда $p(k_j \mid \vec{x})$ или $p(\vec{x} \mid k_j)\cdot P(k_j)$ (см. 8.6) имеют максимальные значения. Таким образом, применение байесовского правила (8.8) обеспечивает минимизацию (8.12) для случая функции потерь вида (8.13).

Структурная схема классификатора, функционирующего на основе правила (8.8), изображена на рисунке 8.4. [43].

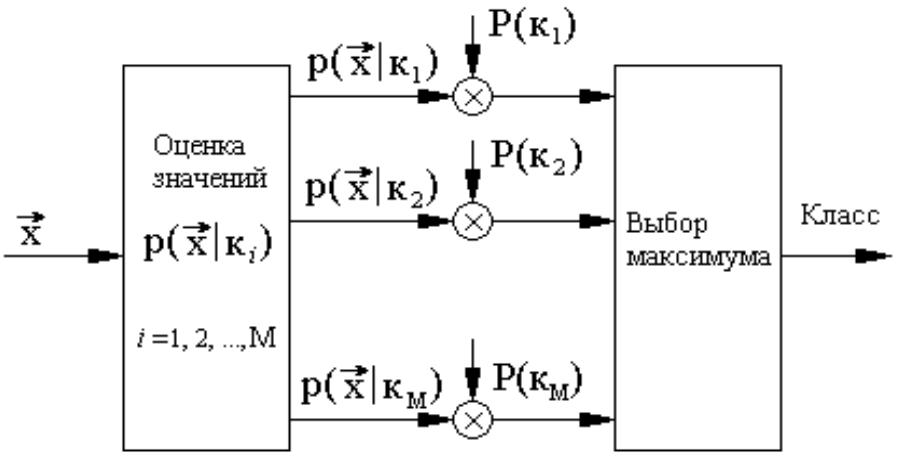


Рисунок 8.4 – Структурная схема байесовского классификатора

Выражение (8.14) получено для случая фиксированной функции потерь, имеющей равные значения при отнесении объекта к любому ошибочному классу. Если функции потерь характеризуется различными значениями при отнесении объекта к тому или иному ошибочному классу, то применяется решающее байесовское правило вида

$$R_i(\vec{x}) < R_j(\vec{x})\,, \quad \forall i \neq j\,. \tag{8.15}$$

Конкретизируем данное выражение для случая M=2. Тогда

$$R_1(\vec{x}) = C(1,1)p(\vec{x} \mid k_1)P(k_1) + C(2,1)p(\vec{x}, k_2)P(k_2),$$
$$R_2(\vec{x}) = C(1,2)p(\vec{x} \mid k_1)P(k_1) + C(2,2)p(\vec{x}, k_2)P(k_2).$$

Подставив значения $R_1(\vec{x})$ и $R_2(\vec{x})$ в (8.15), получим

$$(C(2,1) - C(2,2))p(\vec{x} \mid k_2)P(k_2) < (C(1,2) - C(1,1))p(\vec{x} \mid k_1)P(k_1).$$

Так как обычно $C(k_i, k_j) > C(k_i, k_i)$, то из последнего выражения следует, что вектор $\vec{x}$ принадлежит классу k_1, если

$$\frac{p(\vec{x} \mid k_1)}{p(\vec{x} \mid k_2)} > \frac{(C(2,1) - C(2,2))P(k_2)}{(C(1,2) - C(1,1))P(k_1)}. \qquad (8.16)$$

Отношение $L_{12}(\vec{x}) = p(\vec{x} \mid k_1) / p(\vec{x} \mid k_2)$ называют *отношением правдоподобия*. Отношение, записанное в правой части выражения (8.16), представляет собой некоторое пороговое значение. Обозначим его через θ_{12}. Вектор $\vec{x}$ относится к классу k_1, если выполняется условие $L_{12}(\vec{x}) > \theta_{12}$.

Применение байесовского подхода к решению задачи классификации требует знания априорных вероятностей появления классов $P(k_i)$, плотностей распределения вероятностей для каждого из классов $p(\vec{x} \mid k_i)$, а также соответствующих значений функций потерь.

Рассмотрим особенности байесовской классификации для случая, когда плотность распределения $p(\vec{x} \mid k_i)$ задана многомерным нормальным законом распределения

$$p(\vec{x} \mid k_i) = \frac{1}{(2\pi)^{n/2} \mid C_i \mid^{1/2}} \exp\left(-\frac{1}{2}(\vec{x} - \vec{m}_i)^T C_i^{-1}(\vec{x} - \vec{m}_i)\right), \qquad (8.17)$$

где $\vec{m}_i$ – вектор средних значений i – го класса; C_i – ковариационная матрица с определителем $|C_i|$; n – число элементов вектора $\vec{x}$. Ковариационная матрица C_i определяется из выражения

$$C_i = E\{(\vec{x} - \vec{m}_i)(\vec{x} - \vec{m}_i)^T\},$$

где $E\{.\}$ – символ математического ожидания.

Так как плотность нормального распределения выражается экспоненциальной функцией, то представим решающую функцию $d_i(\vec{x}) = p(\vec{x} \mid k_i)P(k_i)$, соответствующую правилу (8.8), в виде:

$$d_i(\vec{x}) = \ln(p(\vec{x} \mid k_i)P(k_i)) = \ln p(\vec{x} \mid k_i) + \ln P(k_i). \qquad (8.18)$$

Такое представление не меняет суть правила (8.8) в силу того, что натуральный логарифм – монотонно возрастающая функция. Поставив (8.17) в (8.18), получим

$$d_i(\vec{x}) = \ln P(k_i) - \frac{1}{2}\ln \mid C_i \mid - \frac{1}{2}[(\vec{x} - \vec{m}_i)^T C_i^{-1}(\vec{x} - \vec{m}_i)]. \qquad (8.19)$$

При выводе (8.19) член $(n/2)\ln 2\pi$ был опущен, так как он не зависит от *i*. Решающая функция (8.19) определяет разделяющую поверхность, представленную суммой линейных и квадратичных членов. Если ковариационные матрицы для всех классов одинаковы, то (8.19) можно переписать в виде [43]:

$$d_i(\vec{x}) = \ln P(k_i) + \vec{x}^T C^{-1}\vec{m}_i - \frac{1}{2}\vec{m}_i^T C^{-1}\vec{m}_i. \qquad (8.20)$$

Сравнив (8.20) с (8.3), приходим к выводу, что решающая функция (8.20) представляет собой гиперплоскость. Если положить $C = I$, где I - единичная матрица, и $P(k_i) = 1/M$, то

$$d_i(\vec{x}) = \vec{x}^T\vec{m}_i - \frac{1}{2}\vec{m}_i^T\vec{m}_i. \qquad (8.21)$$

Решающая функция (8.21) совпадает с решающей функцией (8.3), которая соответствует классификатору, функционирующему на основе минимума расстояния для случая единственного эталона класса. При этом в качестве эталона класса выступает вектор средних значений $\vec{m}_i$.

Вероятность ошибки байесовского классификатора при использовании решающей функции (8.20) зависит от расстояния Махалонобиса между классами k_i и k_j, представленных эталонами $\vec{m}_i$ и $\vec{m}_j$:

$$d(k_i, k_j) = (\vec{m}_i - \vec{m}_j)^T C^{-1}(\vec{m}_i - \vec{m}_j).$$

Вероятность ошибки монотонно убывает с ростом $d(k_i, k_j)$. При $d(k_i, k_j)$=11 вероятность ошибки меньше 0,05 [43].

Основная проблема применения байесовского классификатора, связана с необходимостью восстановления плотностей распределения вероятностей. В [43] показано, что в тех случаях, когда единственными известными параметрами плотности распределения являются математическое ожидание и дисперсия, целесообразно использовать нормальное распределение. Процесс обучения при этом сводится к оцениванию указанных параметров плотности распределения вероятностей по обучающей выборке.

8.4. Рекуррентные алгоритмы обучения распознаванию образов

Байесовское правило классификации минимизирует средний риск (8.12) и вероятность ошибки. Однако оно требует оценки плотности распределения $p(\vec{x} \mid k_i)$ для каждого класса. В тех случаях, когда объем априорной информации не достаточен для восстановления указанной плотности распределения, могут быть использованы алгоритмы обучения.

Рассмотрим постановку задачи обучения распознаванию образов для случая двух классов. Пусть система распознавания функционирует на основе множества решающих правил вида $f(\vec{x},\vec{w})$, где f – непрерывная функция; $\vec{w}$ – вектор настраиваемых параметров, $\vec{w} = (w_1, w_2, ..., w_m)$.

Если $f(\vec{x},\vec{w})>0$, то вектор $\vec{x}$ относится к первому классу, если $f(\vec{x},\vec{w})<0$, то $\vec{x}$ относится ко второму классу.

Выберем в качестве цели обучения минимум среднего риска. При этом функцию потерь представим в параметрической форме

$$C(\vec{x},\vec{w}) = (y - f(\vec{x},\vec{w}))^2, \tag{8.22}$$

где

$$y = \begin{cases} 1, & \textit{если} \quad \vec{x} \in k_1, \\ 0, & \textit{если} \quad \vec{x} \in k_2 \end{cases}$$

представляет собой классификацию, заданную “учителем”.

В соответствии с (8.22) функция потерь зависит от вектора настраиваемых параметров $\vec{w}$. При этих условиях цель обучения заключается в минимизации среднего риска

$$R(\vec{w}) = \sum_{y=0}^{1} \int_X C(\vec{x},\vec{w})\, p(y \mid \vec{x})\, p(\vec{x})\, d\vec{x}\,. \tag{8.23}$$

Минимум (8.23) достигается, если

$$\nabla_{\vec{w}} R(\vec{w}) = 0\,, \tag{8.24}$$

где $\nabla_{\vec{w}} R(\vec{w}) = grad_{\vec{w}} R(\vec{w}) = \left(\frac{\partial R(\vec{w})}{\partial w_1}, \ldots, \frac{\partial R(\vec{w})}{\partial w_m} \right)$ – градиент среднего риска по аргументу $\vec{w}$.

Так как плотность распределения $p(\vec{x})$ не известна и отсутствует возможность ее восстановления, то уравнение (8.24) не может быть задано в явной форме. В этом случае определение оптимального вектора $\vec{w} = \vec{w}^*$ осуществляется на основе алгоритмов обучения. Дискретный алгоритм обучения может быть записан в форме разностного рекуррентного уравнения

$$\vec{w}[t] = \vec{w}[t-1] - \Gamma[t] \nabla_{\vec{w}} C(\vec{x}[t], \vec{w}[t-1])\,, \tag{8.25}$$

где $\nabla_{\vec{w}} C(\vec{x}[t], \vec{w}[t-1])$ – градиент функции потерь по аргументу $\vec{w}$; $\Gamma[t]$ – диагональная матрица коэффициентов, определяющая скорость сходимости алгоритма. Алгоритм обучения (8.25) позволяет по наблюдаемому вектору $\vec{x}$ определять оценку вектора $\vec{w}[t]$, которая с течением времени стремится к оптимальному вектору $\vec{w}^*$. Случайность градиента $\nabla_{\vec{w}} C(\vec{x}, \vec{w})$ накладывает определенные ограничения на характер изменения во времени матрицы коэффициентов $\Gamma[t]$. С течением времени элементы матрицы $\Gamma[t]$ должны стремиться к нулю. Это обеспечивает сходимость векторов $\vec{w}[t]$ к оптимальному вектору $\vec{w}^*$ с вероятностью единица. При этом элементы матрицы $\Gamma[t]$ должны уменьшаться таким образом, чтобы устранять влияние шума, но не настолько быстро, чтобы остановиться в точке, отличной от минимума [50]. Если $\Gamma[t] = I \cdot \gamma[t]$, где I – единичная матрица, $\gamma[t]$ – элемент последовательности уменьшающихся положительных чисел, то алгоритм обучения (8.25) соответствует методу *стохастической аппроксимации.* Примером последовательности $\gamma[t]$ может служить гармонический ряд 1, 1/2, 1/3,…,1/*t*.

Отметим, что алгоритм обучения (8.15) может быть построен только для таких функций потерь $C(\vec{x}, \vec{w})$, для которых определен градиент $\nabla_{\vec{w}} C(\vec{x}, \vec{w})$. Найдем градиент функции потерь (8.22). Представим решающую функцию $f(\vec{x}, \vec{w})$ в виде

$$f(\vec{x},\vec{w}) = \vec{w}^T\varphi(\vec{x}) = \sum_{i=1}^{N} w_i\varphi_i(\vec{x}), \quad i = 1,2,...,N \tag{8.26}$$

где $\varphi_i(\vec{x})$ – линейно независимые функции. Тогда

$$\begin{aligned}\nabla_w C(y - f(\vec{x},\vec{w})) &= \nabla_w C(y - \vec{w}^T\varphi(\vec{x})) = \\ &= -C'(y - \vec{w}^T\varphi(\vec{x}))\varphi(\vec{x}) = -2(y - \vec{w}^T\varphi(\vec{x}))\varphi(\vec{x}).\end{aligned} \tag{8.27}$$

Подставив (8.27) в (8.25), получим рекуррентное уравнение обучения

$$\vec{w}[t] = \vec{w}[t-1] + 2\Gamma[t](y - \vec{w}^T[t-1]\varphi(\vec{x}[t]))\varphi(\vec{x}[t]). \tag{8.28}$$

Структурная схема обучающейся системы распознавания, реализующая алгоритм (8.28), изображена на рисунке 8.4 [50]. Так как разность $\delta = y - w^T\varphi(\vec{x})$ представляет собой ошибку распознающей системы, то правило изменения значений вектора параметров $\vec{w}[t]$ можно записать в виде:

$$\Delta\vec{w}[t] = 2\Gamma[t]\delta[t]\varphi(\vec{x}[t]). \tag{8.28}$$

Таким образом, коррекция вектора настраиваемых параметров $\vec{w}[t]$ выполняется в направлении вектора $\varphi(\vec{x}[t])$ и пропорциональна ошибке $\delta[t]$.

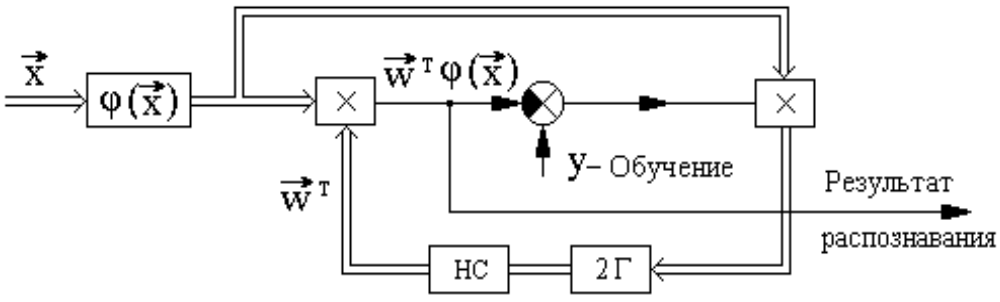


Рисунок 8.4 – Обучающаяся система распознавания

Правила с подобной структурой применяются при построении обучаемых классификаторов образов на основе моделей нейронных сетей. Рассмотрим подробнее основные модели нейронных сетей и алгоритмы их обучения.

8.5. Распознавание и обучение на основе моделей нейронных сетей

Коннекционистские модели машинного обучения используются в ИИ с момента появления персептрона Розенблатта. Название данных моделей происходит от английского слова connection (соединение, связь). В общем случае такие модели представляют собой сети, в которых узлы соединены направленными связями с определенными весами. Каждый узел выполняет обработку сигналов, поступающих на его вход, и рассматривается как некоторый процессорный элемент (ПЭ). Так как алгоритм функционирования процессорных элементов подобен алгоритмам преобразования информации в нервных клетках человека и животных – нейронах, то коннекционистские модели также называют *искусственными нейронными сетями* (ИНС). В этом случае узлы сети – это искусственные нейронные элементы.

За последние двадцать лет ИНС нашли широкое применение при решении самых разнообразных задач, к которым относят: распознавание рукописных знаков и речи, сжатие различных сигналов, обработка и понимание изображений, экспертные системы, оптимизация, прогнозирование и др.

Обучение ИНС сводится к модификации весов связей между нейронными элементами. Существуют три различных подхода к обучению ИНС: обучение с учителем, обучение с подкреплением, обучение без учителя (самоорганизующиеся нейронные сети). Ниже рассмотрим указанные модели обучения на примерах ИНС различной структуры. Однако предварительно введем основные понятия.

8.5.1. Модели нейронных элементов

Известно, что мозг человека содержит примерно 10^{11} нейронов [47]. Каждый нейрон представляет собой живую клетку, состоящую из дендритов, сомы и аксона (рисунок 8.5). *Дендриты* – ветвеобразные отростки, которые обеспечивают сбор сигналов от других нейронов или рецепторов. *Сома* нейрона представляет тело клетки. В соме происходят сложные биохимические процессы, благодаря которым осуществляются нелинейные преобразования сигналов, поступающих через дендриты. *Аксон* является отростком клетки, по которому ее выходной сигнал поступает на дендриты других нейронов. Аксон разветвляется на большое число волокон. Место соединения волокон с дендритами называется *синапсом*.

Сигналы, распространяющиеся в биологической нейронной сети, представляют собой короткие электрические импульсы. Под воздействием

электрических импульсов, поступающих через синапсы и дендриты, изменяется потенциал сомы нейрона. В момент достижения им некоторого порогового значения, нейрон вырабатывает электрический импульс, который распространяется вдоль аксона. Потенциал сомы резко снижается, нейрон как бы разряжается. Через некоторое время нейрон может опять сформировать импульс. Если импульсы, поступающие на синапс, приводят к повышению потенциала сомы, то такой синапс называют *возбуждающим*. Если импульсы, поступающие на синапс, снижают потенциал сомы, то синапс называют *тормозящим*. Биологический нейрон вырабатывает последовательность электрических импульсов, частота следования которых характеризует активность нейрона.

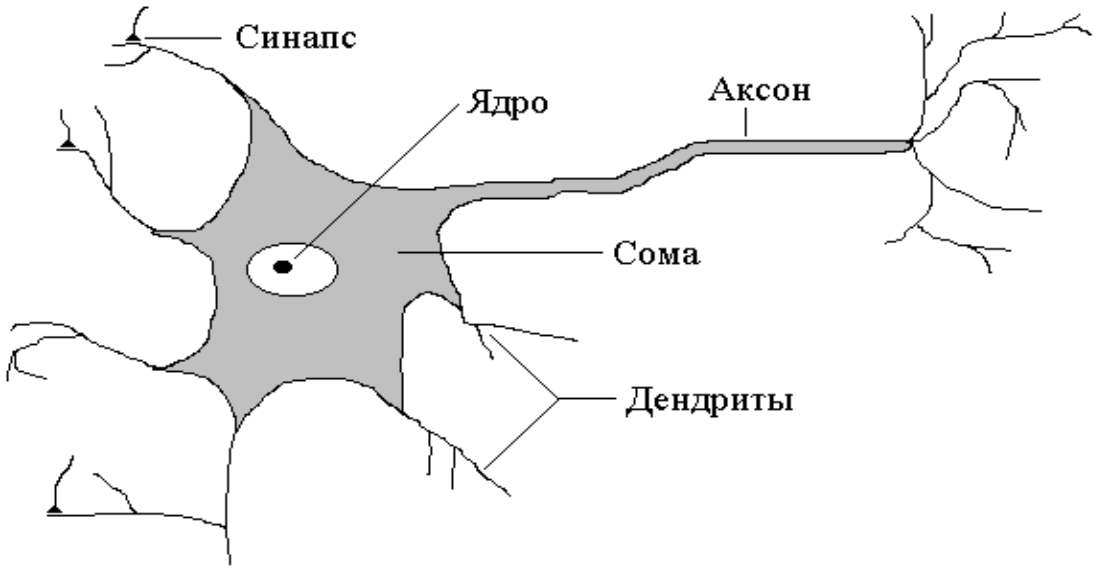


Рисунок 8.5 – Биологический нейрон

При построении ИНС в рассмотренную модель нейрона вносят существенные упрощения. При этом в зависимости от степени абстрагирования получают различные модели формальных (искусственных) нейроподобных элементов (НЭ) и, соответственно, ИНС. На практике широкое распространение получили ИНС с формально-логическими моделями НЭ, ведущими свою историю с 1943 г., когда Маккаллох и Питтс предложили соответствующую модель НЭ. Именно этот класс моделей с учетом их развития и рассматривается в дальнейшем. Вместе с тем в последние годы интенсивно развиваются исследования в области ИНС с импульсными НЭ, которые в большей степени соответствуют биологическому нейрону [82,76,3].

Рассмотрим модель формального нейрона, построенную Маккаллохом и Питтсом (рисунок 8.6). В этой модели входные сигналы $x_1, x_2, \ldots, x_m$, $j = 1, 2, \ldots, m$ умножаются на соответствующие веса синаптических

связей $w_{i1}, w_{i2}, \ldots, w_{im}$ и суммируются, формируя значение u_i. Выходной сигнал i-го НЭ определяется с помощью выражения

$$y_i(t+1) = g(\sum_{j=1}^{m} w_{ij} x_j(t) - \theta_i), \tag{8.29}$$

где $g(.)$ – функция преобразования нейрона; θ_i – пороговое значение; m – количество входов НЭ.

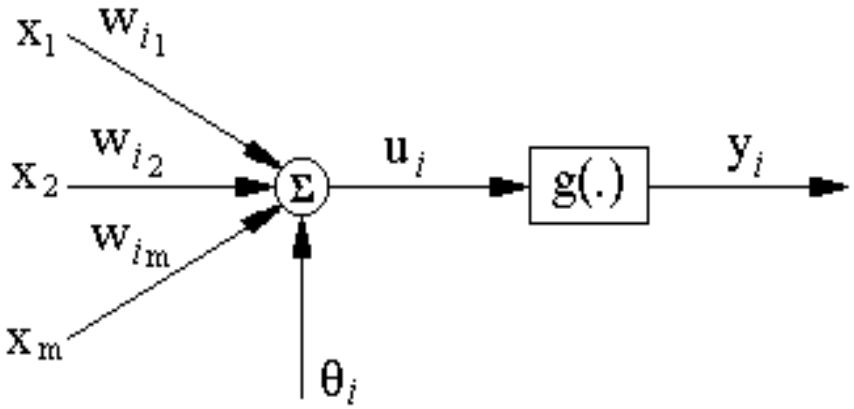


Рисунок 8.6 – Формальный нейронный элемент

Для модели НЭ Маккаллоха – Питтса функция преобразования $g(u_i)$ представляет единичную ступенчатую функцию, которая называется также функцией Хевисайда и обозначается $H(u_i)$:

$$g(u_i) = H(u_i) = \begin{cases} 1, & u_i \geq 0 \\ 0, & u_i < 0. \end{cases} \tag{8.30}$$

Таким образом, выходной сигнал НЭ $y_i = 1,$ если

$$\sum_{j=1}^{m} w_{ij} x_j \geq \theta_i .$$

В ином случае выходной сигнал НЭ равен 0. Заметим, что выходной сигнал НЭ может меняться только в дискретные моменты времени $t, t+1, t+2$ и т.д. Выражение

$$u_i = net_i = \sum_{j=1}^{m} w_{ij} x_j - \theta_i = \sum_{j=0}^{m} w_{ij} x_j \tag{8.31}$$

определяет способ объединения входных признаков x_j. Оно может быть представлено функцией $net_i = f(\vec{w}_i, \vec{x})$, называемой *сетевой* или *базовой* функцией НЭ. Формальные модели нейронов отличаются между собой видом сетевой функции net_i и видом функции преобразования $g(.)$.

Сетевая функция вида (8.31) является *линейной*. Кроме линейной сетевой функции, в ИНС широко используются *квадратическая*

$$net_i = \sum_{j=0}^{m} w_{ij} x_j^2 \tag{8.32}$$

и *радиальная* (*сферическая*) функции

$$net_i = \frac{1}{R^2} \sum_{j=0}^{m} (x_j - w_{ij})^2, \tag{8.33}$$

где R – радиус гиперсферы.

В качестве функции преобразования в ИНС применяются следующие основные функции (рисунок 8.7):

- ступенчатая функция (8.30) (рисунок 8.7,а);
- линейная функция с насыщением (рисунок 8.7,б)

$$g(u) = \begin{cases} 1, & u > 1, \\ u, & 0 \le u \le 1, \\ 0, & u < 0; \end{cases} \tag{8.34}$$

- знаковая функция (рисунок 8.7,в)

$$g(u) = \operatorname{sgn}(u) = \begin{cases} 1, & u \ge 0, \\ -1, & u < 0; \end{cases} \tag{8.35}$$

- униполярная сигмовидная функция (рисунок 8.7,г)

$$g(u) = \frac{1}{1 + e^{-2\beta u}}, \tag{8.36}$$

где β – постоянный коэффициент;

- биполярная сигмовидная функция (рисунок 8.7,д), представляющая собой гиперболический тангенс

$$g(u) = \frac{2}{1+e^{-2\beta u}} - 1 = \frac{e^{\beta u} - e^{-\beta u}}{e^{\beta u} + e^{-\beta u}} = \tanh(\beta u)\,; \qquad (8.37)$$

- гауссова функция (рисунок 8.7,е)

$$g(u) = ce^{-u^2/\sigma^2}, \qquad (8.38)$$

где σ ,с – постоянные коэффициенты.

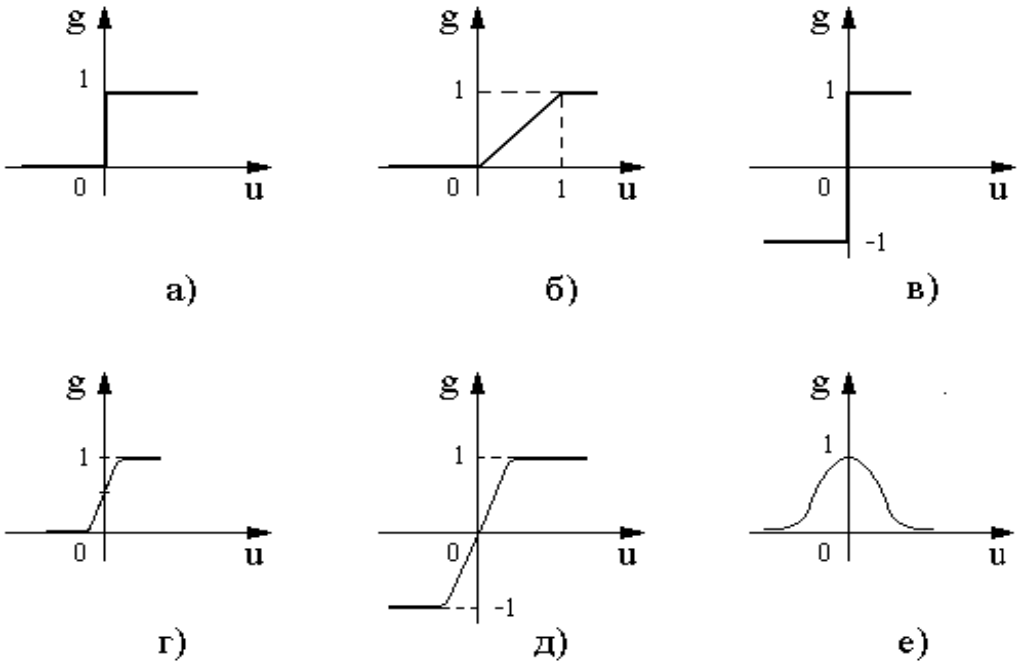
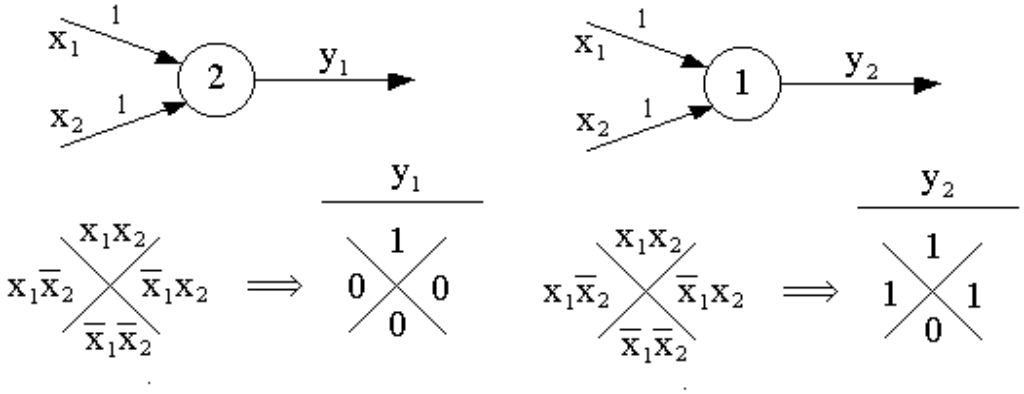


Рисунок 8.7 – Функции преобразования НЭ

В ИНС наиболее часто используется два типа НЭ:

- НЭ с линейной сетевой функцией вида (8.31) и функцией преобразования вида (8.30) или (8.35). Такие НЭ называются *линейными пороговыми элементами* (ЛПЭ). Сигналы, распространяющие в сетях, составленных из ЛПЭ, являются бинарными и принимают значения 0 или 1, либо +1 и –1.
- НЭ с линейной сетевой функцией (8.31) и функцией преобразования вида (8.36) или (8.37). Такие НЭ называются *линейно-непрерывными элементами* (ЛНЭ). Сигналы, распространяющие в сетях из ЛНЭ, принимают произвольные значения из диапазона [0,1] или [-1,1].

На рисунке 8.8 условно изображены два ЛПЭ с входами x_1 и x_2, принимающими значения 0 и 1. Пороговое значение, при котором на выходе ЛПЭ формируется единичный сигнал, вписано внутрь круга. ЛПЭ, изображенный на рисунке 8.8,а, выполняет логическую операцию “И”, а ЛПЭ, изображенный на рисунке 8.8,б, – логическую операцию “ИЛИ”.



а) б)

Рисунок 8.8 – ЛПЭ, реализующие логические функции “И” а) и “ИЛИ” б)

8.5.2. Структуры нейронных сетей

ИНС состоит из большого числа взаимосвязанных НЭ. Структура нейронной сети отображается направленным графом, в котором вершины – это НЭ, а дуги представляют связи между НЭ. Дуга, направленная от НЭ с номером j к НЭ элементу с номером i, характеризуется весом w_{ij}.

Выделяют следующие основные разновидности структур ИНС: однослойные ИНС с прямыми связями; многослойные ИНС с прямыми связями; однослойные рекуррентные ИНС (ИНС с обратными связями); многослойные рекуррентные ИНС. Структуры указанных ИНС изображены на рисунке 8.9.

Реакция i–го НЭ в однослойной сети с прямыми связями определяется выражением (8.29). Связи входов сети с соответствующими НЭ могут быть заданы матрицей весов связей W

$$W = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1m} \\ w_{21} & w_{22} & \dots & w_{2m} \\ \dots & & & \\ w_{n1} & w_{n2} & \dots & w_{nm} \end{bmatrix} = \begin{bmatrix} \vec{w}_1 \\ \vec{w}_2 \\ \dots \\ \vec{w}_n \end{bmatrix}, \tag{8.39}$$

где $\vec{w}_1$, $\vec{w}_2$, …, $\vec{w}_n$ - векторы весов НЭ с номерами 1, 2, … , n. Если между некоторым входом и НЭ нет связи, то соответствующий вес равен нулю.

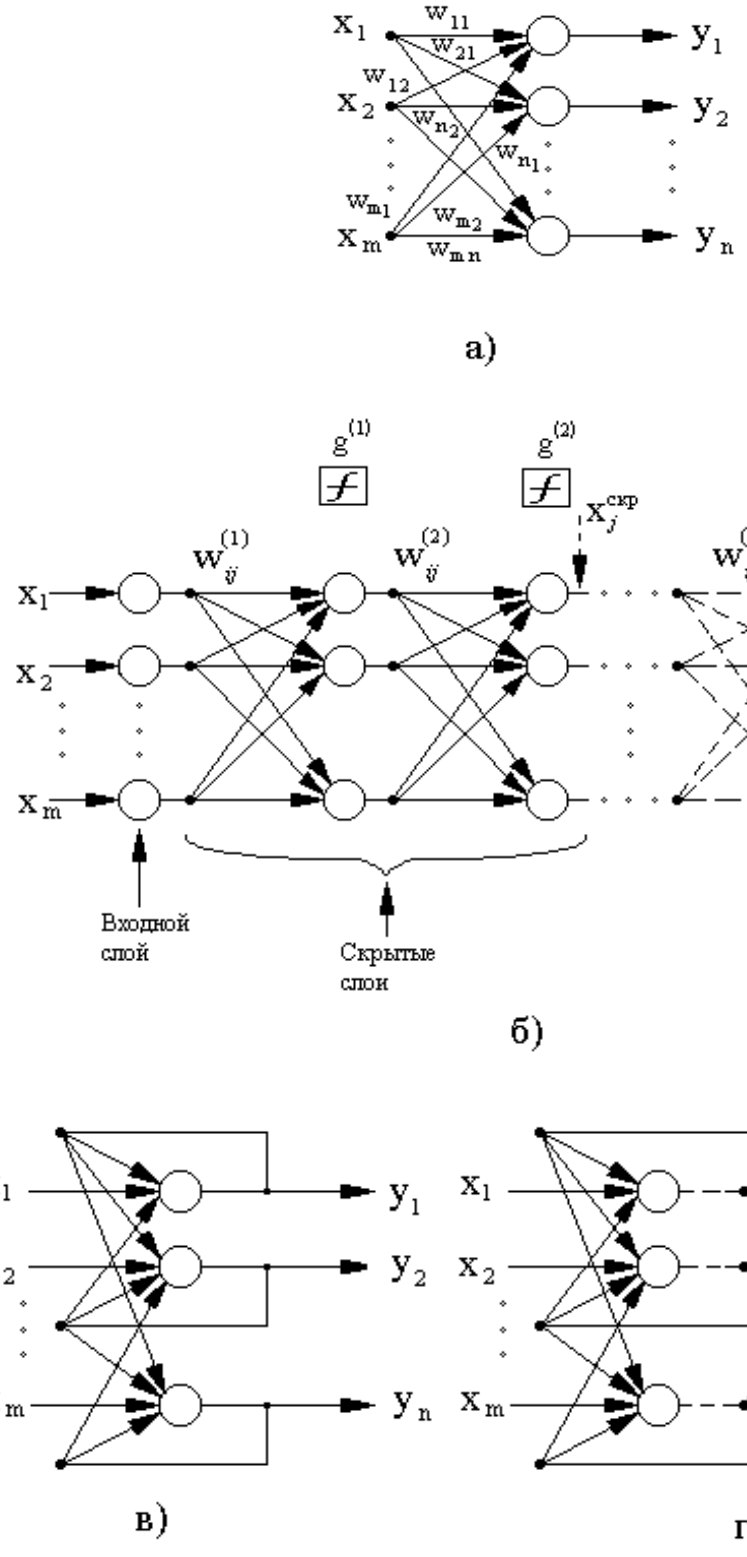


Рисунок 8.9 – Структуры ИНС: а) однослойные; б) многослойные с прямыми связями: в) однослойные рекуррентные; г) многослойные рекуррентные

В многослойной сети с прямыми связями выходы одного слоя НЭ выступают в роли входов другого слоя НЭ. Внутренние слои многослойной ИНС называют *скрытыми слоями*. Реакция *i*–го выходного НЭ многослойной ИНС вычисляется последовательным применением выражения (8.29). Например, если сеть является двухслойной, то

$$y_i = g(net_i^{(2)}) = g(\sum_j w_{ij}^{(2)} x_j^{(1)}) = g(\sum_j w_{ij}^{(2)} g(net_i^{(1)})) = g(\sum_j w_{ij}^{(2)} g(\sum_j w_{ij}^{(1)} x_j)), \quad (8.40)$$

где $x_j^{(1)}$ – реакция *j*-го нейрона скрытого слоя (в рассматриваемом случае первого слоя); $w_{ij}^{(1)}$, $w_{ij}^{(2)}$ – веса связей нейронов первого и второго слоев, соответственно; $net_i^{(1)}$, $net_i^{(2)}$ – сетевые функции нейронов первого и второго слоев.

Поведение рекуррентных ИНС сложнее из-за наличия обратных связей. Обратные связи, действующие в пределах одного слоя НЭ, называют *латеральными* (от английского lateral – горизонтальный). Часто используют ИНС с *латеральным торможением*. В этом случае латеральные связи, направленные к близко расположенным НЭ, являются возбуждающими, а связи, направленные к НЭ, находящимся на некотором удалении, являются тормозящими.

ИНС с латеральным торможением изображена на рисунке 8.10. Обратные связи здесь показаны только для одного НЭ. Латеральные связи, оканчивающиеся зачерненным кружочком, являются тормозящими.

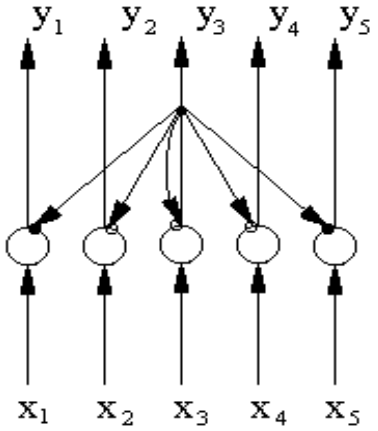


Рисунок 8.10 – ИНС с латеральным торможением

В многослойных рекуррентных ИНС обратные связи могут охватывать несколько слоев (рисунок 8.9,г). При этом внутренние слои могут иметь как прямые связи, так и обратные. В этом случае ИНС будет иметь комбинированную структуру.

8.5.3. Виды обучения ИНС

Обучение ИНС сводится к целенаправленному изменению весов связей *W*. Различают три вида обучения ИНС (рисунок 8.11):обучение с учителем; обучение с подкреплением; обучение без учителя [76].

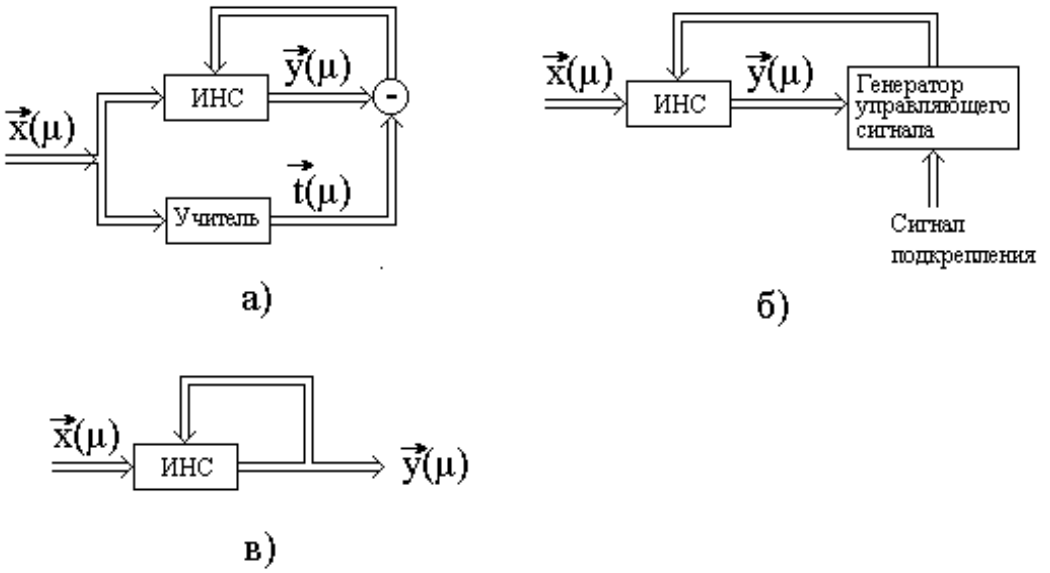


Рисунок 8.11 – Виды обучения: а) с учителем; б) с подкреплением; в) без учителя

При обучении с учителем предполагается, что для каждого μ–го входного вектора $\vec{x}(\mu)$, представляющего входной сигнал, мы заранее знаем желаемый выходной вектор ИНС $\vec{t}(\mu)$ и используем разность между желаемым $\vec{t}(\mu)$ и действительным выходным вектором ИНС $\vec{y}(\mu)$ (т.е. ошибку ИНС) для изменения в нужном направлении матрицы *W* (рисунок 8.11). Матрица весов *W* изменяется так, чтобы действительный выходной вектор ИНС $\vec{y}(\mu)$ приближался к желаемому вектору $\vec{t}(\mu)$. В этом случае предполагают, что имеется множество примеров *X*, элементами которого являются пары $(\vec{x}(\mu), \vec{t}(\mu))$:

$$X = \{(\vec{x}(\mu), \vec{t}(\mu)); \quad 1 \le \mu \le p\}. \tag{8.41}$$

Множество примеров *X* разбивается на две части, одна из которых называется *обучающей* последовательностью, а другая – *тестовой*.

Цель обучения состоит в том, чтобы ИНС после обучения формировала корректные ответы на тестовой последовательности входных векторов $\vec{x}(\mu)$, которые не входили в обучающую последовательность.

При обучении с подкреплением объем доступной информации о желаемых реакциях ИНС ограничен. Часто может быть известна лишь степень соответствия действительной реакции желаемой. Например, 70%. В предельных случаях действительная реакция может оцениваться двумя значениями – “верно” и “неверно”. Управление весами связей W в этом случае осуществляется на основе внешнего сигнала, который называют подкреплением. Этот сигнал носит качественный характер. Поэтому с помощью генератора управляющего сигнала формируется более информативный сигнал.

При обучении без учителя заранее не известна желаемая реакция ИНС на заданный входной вектор $\vec{x}(\mu)$ и, следовательно, невозможно использовать ошибку ИНС для изменения весов связей W. В этом случае ИНС должна самообучаться, обнаруживая закономерности, присущие пространству входных сигналов. Отметим, что термин обучения без учителя (unsupervised learning) не очень удачен, так как обучение без учителя в полной мере невозможно.

Роль учителя сводится не только к указанию принадлежности тех или иных входных векторов ИНС определенным классам, но и к заданию цели обучения [50]. Кроме этого, многие вопросы обучения решает на этапе создания ИНС ее разработчик – учитель. Например, он осуществляет выбор признаков классификации ситуаций, что устраняет произвольность классификации. Поэтому часто ИНС, обучаемые без учителя, называют *самоорганизующимися сетями*, подчеркивая тот факт, что в данных сетях процесс обучения отличается от процесса обучения с учителем отсутствием внешней корректировки. Дополнительной информации о правильности реакции ИНС в этом случае не имеется. Вместе с тем термин “обучение без учителя” получил широкое использование в исследованиях по нейронным сетям. Будем использовать его с учетом сделанной оговорки.

8.5.4. Обучение с учителем в ИНС с прямыми связями

8.5.4.1. Простой персептрон. Рассмотрим НЭ, описываемый выражением (8.29). В этом случае индекс i может быть опущен. Такой НЭ называют простым персептроном (от латинского perceptio – восприятие, познавание). При подаче на вход персептрона μ–го образа его реакция определится с помощью выражения

$$y(\mu) = H\left(\sum_{j=1}^{m} w_j x_j(\mu) - \theta\right), \qquad (8.42)$$

где $H(.)$ – единичная ступенчатая функция Хевисайда.

В качестве функции преобразования персептрона может использоваться также знаковая функция *sgn(.)*. В этом случае входные и выходные сигналы персептрона будут биполярными.

Задача обучения персептрона заключается в поиске такого вектора весов $\vec{w} = (w_1, w_2, ..., w_m)$, который обеспечит совпадения желаемой $t(\mu)$ и действительной реакции персептрона $y(\mu)$ на заданный входной вектор $\vec{x}(\mu)$. Если $net = \sum_{j=1}^{m} w_j x_j - \theta > 0$, то выходной сигнал НЭ будет равен +1. Это означает, что входной вектор $\vec{x}(\mu)$ принадлежит классу, на который настроен НЭ. В ином случае входной вектор не принадлежит классу. При этом сетевая функция НЭ $net = f(\vec{w}, \vec{x})$ играет роль дискриминантной функции, позволяющей установить принадлежность вектора $\vec{x}$ классу.

Поверхность, разделяющая векторы входного пространства на две группы, определится из выражения

$$\sum_{j=1}^{m} w_j x_j - \theta = 0 . \qquad (8.43)$$

Выражение (8.43) является линейным и представляет собой уравнение гиперплоскости. Таким образом, простой персептрон может классифицировать только такие образы входного пространства, которые разделимы с помощью гиперплоскости. Иными словами, задачи классификации, решаемые простым персептроном, – это линейные сепарабельные задачи. На рисунке 8.12,а изображена линейная сепарабельная задача классификации в двумерном пространстве. Здесь точки, обозначенные крестиками, отделяются от точек, обозначенных кружочками, прямой линией. Отметим, что пороговое значение θ в (8.43) характеризует расстояние от разделяющей гиперплоскости до начала координат. Если θ=0, то гиперплоскость будет проходить через начало координат.

На рисунке (8.12,б) изображена задача исключающего ИЛИ, которая не является линейно-сепарабельной и, следовательно, не может быть решена простым персептроном.

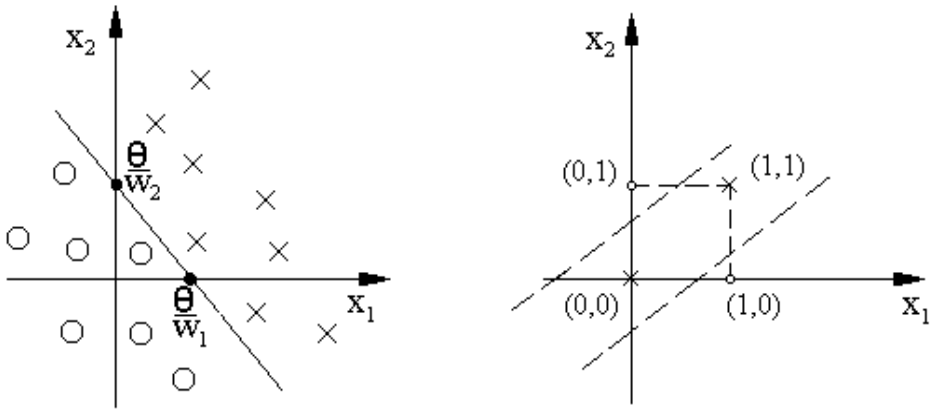
В то же время задача логического И (ИЛИ) является линейно-сепарабельной и успешно решается с помощью простого персептрона.

Идея обучения персептрона состоит в следующем. На вход персептрона подаются последовательно входные векторы $\vec{x}(\mu)$. Одновременно с этим выходная реакция персептрона $y(\mu)$ сравнивается с желаемой реакцией $t(\mu)$. Если указанные реакции совпадают, то никаких действий не предпринимают. Если $y(\mu) \neq t(\mu)$, то вектор весов персептрона $\vec{w}$ изменяется на величину $\Delta\vec{w}$. Адаптация весов выполняется с помощью правила

$$\Delta\vec{w} = \eta[t(\mu) - y(\mu)]\vec{x}(\mu), \qquad (8.44)$$

где η – коэффициент обучения, $\eta > 0$. Если обозначить ошибку обучения через $\delta(\mu)$, т.е. $\delta(\mu) = t(\mu) - y(\mu)$, то правило обучения можно переписать в виде

$$\Delta\vec{w} = \eta\delta(\mu)\vec{x}(\mu). \qquad (8.45)$$



а) б)

Рисунок 8.12 – Линейная сепарабельная задача а) и задача исключающего ИЛИ б)

Таким образом, если $\delta(\mu)$=0, то вектор весов не меняется. Если $\delta(\mu)$>0, то вектор весов $\vec{w}$ меняется в направлении вектора $\vec{x}(\mu)$ (рисунок 8.13). Если порог θ рассматривается как дополнительный вес w_0, то разделяющая гиперплоскость будет проходить через начало координат, а изменения вектора $\vec{w}$ будут соответствовать вращению гиперплоскости вокруг начала координат.

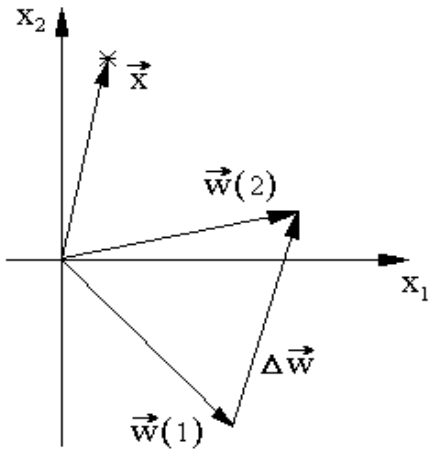


Рисунок 8.13 – Иллюстрация правила обучения персептрона

В дальнейшем будет показано, что большинство правил обучения ИНС имеют структуру, подобную правилу (8.45).

Важно отметить, что правило обучения персептрона обеспечивает схождение вектора весов $\vec{w}$ к требуемым значениям, определяемым условием (8.43), за конечное число шагов при условии существования решения (теорема сходимости) [68]. При этом число шагов в значительной степени зависит от коэффициента η, а также от структуры обучающей последовательности.

8.5.4.2. Адаптивный линейный элемент (Adaptive Linear Element). Адаптивный линейный элемент представляет собой НЭ с линейной функцией преобразования $g(u) = u$ (рисунок 8.14).

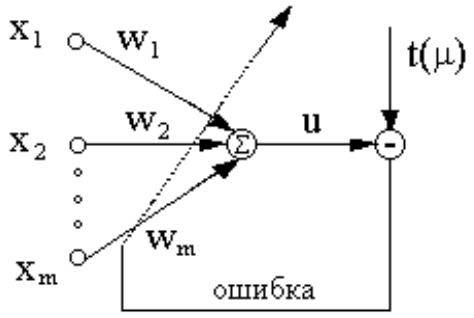


Рисунок 8.14 – Адаптивный линейный элемент

В ходе обучения с учителем вектор весовых коэффициентов корректируется так, чтобы выходной сигнал $u(\mu)$ имел наилучшее приближение к желаемой реакции $t(\mu)$. В большинстве практических случаев минимизируется среднеквадратическое значение ошибки

$$E(\vec{w}) = \frac{1}{2}\sum_{\mu=1}^{P}[t(\mu) - u(\mu)]^2 = \frac{1}{2}\sum_{\mu=1}^{P}[t(\mu) - \vec{w}^T\vec{x}(\mu)]^2 = \frac{1}{2}\sum_{\mu=1}^{P}[t(\mu) - \sum_{j=1}^{m} w_j x_j(\mu)]^2. \quad (8.46)$$

С целью минимизации ошибки (8.46) применяют метод градиента. В этом случае веса w_j изменяются в направлении, обратном направлению градиента

$$\nabla E(\vec{w}) = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_m}),$$

т.е.

$$\Delta w_j = -\eta \frac{\partial E}{\partial w_j}.$$

Найдем частную производную $\partial E / \partial w_j$, используя схему

$$\frac{\partial E}{\partial w_j} = \sum_{\mu=1}^{P} \frac{\partial E}{\partial net\,(\mu)} \frac{\partial net\,(\mu)}{\partial w_j} \quad ,$$

где $net(\mu) = \sum_{j=1}^{m} w_j x_j(\mu)$. Тогда

$$\frac{\partial E}{\partial w_j} = \sum_{\mu=1}^{P} [t(\mu) - net\,(\mu)] x_j(\mu) \, .$$

Отсюда

$$\Delta w_j = \eta \sum_{\mu=1}^{P} \delta(\mu) x_j(\mu) \, , \qquad (8.47)$$

где $\delta(\mu) = t(\mu) - net(\mu)$ – ошибка адаптивного линейного элемента.

Отметим, что в (8.47) суммирование выполняется по всем входным векторам $\vec{x}(\mu)$. Поэтому в соответствии с правилом (8.47) вес w_j изменяется после вычисления ошибок $\delta(\mu)$ для каждого из входных векторов. На практике часто применяют последовательное обучение, при котором вектор весов $\vec{w}$ корректируют сразу после поступления на вход НЭ очередного входного вектора. Тогда

$$\Delta\vec{w} = \eta\delta(\mu)\vec{x}(\mu) = \eta(t(\mu) - \vec{w}^T\vec{x}(\mu))\vec{x}(\mu) \, . \qquad (8.48)$$

Правило обучения (8.48) называют правилом обучения адаптивного линейного элемента или *правилом Уидроу-Хоффа*. По своей структуре оно совпадает с правилом обучения простого персептрона (8.45). На первый взгляд, отличие между (8.45) и (8.48) заключается только в способе вычисления ошибки $\delta(\mu)$. Вместе с тем это не так. Правило (8.48) получено из (8.47) в результате отказа от усреднения по всем входным векторам, что не гарантирует уменьшения среднеквадратической ошибки на каждом шаге. Частные изменения вектора $\vec{w}$, соответствующие входному вектору $\vec{x}(\mu)$ и уменьшающие текущую ошибку, могут не соответствовать следующим входным векторам. Это может приводить к увеличению среднеквадратической ошибки. Кроме этого, если правило обучения персептрона обеспечивает сходимость решения за конечное число шагов, то правило обучения Уидроу-Хоффа, выведенное на основе метода градиента, обеспечивает лишь асимптотическую сходимость при большом числе шагов.

Обобщим правило Уидроу-Хоффа на случай однослойной сети НЭ с дифференцируемой функцией преобразования $g(u)$. В этом случае среднеквадратическое значение ошибки будет равно

$$E(\vec{w}) = \frac{1}{2}\sum_{\mu=1}^{P}\sum_{i=1}^{n}[t_i(\mu) - y_i(u)]^2 = \frac{1}{2}\sum_{\mu=1}^{P}\sum_{i=1}^{n}[t_i(\mu) - g(net_i(\mu))]^2 , \quad (8.49)$$

где $net_i(\mu) = \vec{w}_i^T \vec{x}(\mu) = \sum_{j=1}^{m} w_{ij} x_j(\mu)$ – значение сетевой функции *i*–го нейрона при подаче на его вход вектора $\vec{x}(\mu)$.

Применив к (8.49) метод градиента, получим

$$\frac{\partial E}{\partial w_{ij}} = \sum_{\mu=1}^{p} [t_i(\mu) - g(net_i(\mu))]\, g'(net_i(\mu)]\, x_j(\mu) ,$$

где $g'(net_i(\mu))$ – производная функция преобразования НЭ. Отсюда для последовательного обучения

$$\Delta w_{ij} = \eta[t_i(\mu) - g(net_i(\mu))]\, g'(net_i(\mu))\, x_j(\mu) . \quad (8.50)$$

Выражение (8.50) называют *дельта-правилом обучения* и часто записывают в форме

$$\Delta w_{ij} = \eta \delta_i(\mu) x_j(\mu) , \quad (8.51)$$

где

$$\delta_i(\mu) = [t_i(\mu) - g(net_i(\mu))]\, g'(net_i(\mu)) .$$

Таким образом, в дельта-правиле ошибка $\delta_i(\mu)$ пропорциональна производной функции преобразования НЭ.

8.5.4.3. Многослойный персептрон. Многослойный персептрон состоит из нескольких слоев нейронов с прямым распространением сигналов (рисунок 8.9,б). Обучение многослойного персептрона выполняется на основе алгоритма обратного распространения ошибки [68,76].

Данный алгоритм получается в результате обобщения дельта-правила на многослойные ИНС с прямыми связями.

Рассмотрим двухслойную нейронную сеть с прямыми связями (рисунок 8.14). Определим реакцию *i*–го нейрона выходного слоя на входной вектор $\vec{x}(\mu)$. Для этого составим сначала выражения, определяющие реакцию нейронов скрытого слоя.

Сетевая функция *j*–го нейрона скрытого слоя для μ–го входного вектора равна

$$net_j(\mu) = \sum_{k=1}^{m} v_{jk} x_k(\mu), \tag{8.52}$$

где $j = 1, 2, \ldots, l$. Реакция j–го нейрона скрытого слоя определится из выражения

$$z_j(\mu) = g(net_j(\mu)) = g(\sum_{k=1}^{m} v_{jk} x_k(\mu)). \tag{8.53}$$

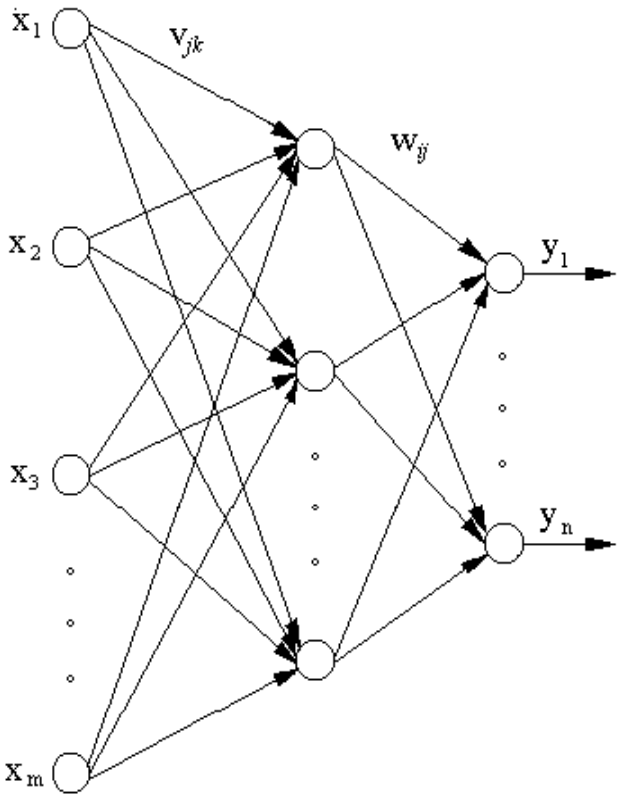


Рисунок 8.14 – Двухслойная нейронная сеть с прямыми связями

Сетевая функция i–го нейрона выходного слоя равна

$$net_i(\mu) = \sum_{j=1}^{l} w_{ij} z_j(\mu) = \sum_{j=1}^{l} w_{ij} g(\sum_{k=1}^{m} v_{jk} x_k(\mu)). \tag{8.54}$$

Тогда реакция i–го нейрона выходного слоя на входной вектор $\vec{x}(\mu)$ может быть вычислена с помощью выражения

$$y_i(\mu) = g(net_i(\mu)) = g(\sum_{j=1}^{l} w_{ij} z_j(\mu)) = g(\sum_{j=1}^{l} w_{ij} g(\sum_{k=1}^{m} v_{jk} x_k(\mu))). \tag{8.55}$$

Выражения (8.52) – (8.55) соответствуют прямому распространению сигналов в рассматриваемой ИНС.

Процесс обучения многослойного персептрона основан на минимизации ошибки (8.49) при учете (8.55). В соответствии с методом градиента изменение весов связей выходного слоя w_{ij} выполняется по правилу (8.47)

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} = \eta \sum_{\mu=1}^{P} [t_i(\mu) - y_i(\mu)] g'(net_i(\mu)) z_j(\mu) = \eta \sum_{\mu=1}^{P} \delta_i(\mu) z_j(\mu), \quad (8.56)$$

где

$$\delta_i(\mu) = [t_i(\mu) - y_i(\mu)]\, g'(net_i(\mu)) . \quad (8.57)$$

Изменение весов связей скрытого слоя v_{jk} определим, воспользовавшись следующей схемой

$$\Delta v_{jk} = -\eta \frac{\partial E}{\partial v_{jk}} = -\eta \frac{\partial E}{\partial z_j(\mu)} \frac{\partial z_j(\mu)}{\partial v_{jk}} .$$

Тогда

$$\begin{aligned} \Delta v_{jk} &= \eta \sum_{\mu=1}^{P} \sum_{i=1}^{n} [t_i(\mu) - y_i(\mu)] g'(net_i(\mu)) w_{ij} g'(net_j(\mu)) x_k(\mu) = \\ &= \eta \sum_{\mu=1}^{P} \sum_{i=1}^{n} \delta_i(\mu) w_{ij} g'(net_j(\mu)) x_k(\mu) = \eta \sum_{\mu=1}^{P} \delta_j(\mu) x_k(\mu), \end{aligned} \quad (8.58)$$

где

$$\delta_j(\mu) = g'(net_j(\mu)) \sum_{i=1}^{n} \delta_i(\mu) v_{ij} . \quad (8.59)$$

Заметим, что выражения (8.56) и (8.58) имеют одну и ту же форму. Отличие заключается в способе вычисления ошибок $\delta_i(\mu)$ и $\delta_j(\mu)$.

Часто (8.56) и (8.58) записывают в общей форме

$$\Delta w_{ij} = \eta \sum_{\mu=1}^{p} \delta_i^{вых}(\mu) x_j^{вх}(\mu) \quad (8.60)$$

и называют *обобщенным дельта правилом*. В соответствии с этим правилом изменение веса связи между нейронами j и i пропорционально произведению ошибки $\delta_i^{вых}(\mu)$ на выходе нейрона i на его входной сигнал,

формируемый нейроном *j*. При этом предполагается, что слой, в котором находится нейрон *j*, предшествует слою, в котором находится нейрон *i*.

Если рассматривается выходной слой нейронов, то ошибка вычисляется по формуле (8.57). Для скрытых слоев ошибка вычисляется по рекуррентной формуле (8.59), т.е. ошибка предыдущего слоя $\delta_j(\mu)$ вычисляется на основе ошибки следующего слоя $\delta_i(\mu)$. Это соответствует распространению ошибки в обратном направлении – с выхода на вход.

Если в качестве функции преобразования используется сигмовидная функция (8.10) или (8.11), то

$$g'(net) = 2\beta g(net)(1 - g(net) \text{ или } g'(net) = \beta(1 - g(net)) .$$

При этом обычно полагают $\beta = 1/2$ или $\beta = 1$.

Выражения (8.56) – (8.59) определяют *алгоритм обратного распространения* ошибки (ОРО), состоящий из следующих шагов [68]:

1) выполнить инициализацию весов связей небольшими случайными значениями, задать максимально допустимую среднеквадратичную ошибку E_{max}, положить текущее значение среднеквадратичной ошибки ИНС $E=0$;
2) подать на вход сети μ-ый входной вектор $\vec{x}(\mu)$;
3) выполнить распространение сигналов в соответствии с прямыми связями
$$x_i^{(k)} = g(net_i^{(k)}) = g(\sum_j w_{ij}^{(k)} x_j^{(k-1)}),$$
где k – номер слоя нейронов, $k = 1, 2, \ldots,$; $x_i^{(k)}$ –выходной сигнал *i*–го нейрона *k*–го слоя; $w_{ij}^{(k)}$ - веса связей *k*–го слоя;
4) вычислить среднеквадратичную ошибку ИНС
$$E := E + \frac{1}{2}\sum_{i=1}^{N} (t_i(\mu) - x_i^k(\mu))^2$$
и ошибку нейронов выходного слоя
$$\delta_i^{(k)} = g'(net_i^{(k)})[t_i(\mu) - x_i^{(k)}(\mu)];$$
5) выполнить распространение ошибки в обратном направлении, руководствуясь правилом

$$\delta_i^{(k-1)} = g'(net_i^{(k-1)}) \sum_j w_{ij}^{(k)} \delta_j^{(k)} ;$$

6) обновить веса связей каждого слоя в соответствии с правилом

$$\Delta w_{ij}^{(k)} = \eta \delta_i^{(k)} x_j^{(k-1)} ; \qquad (8.61)$$

7) если $\mu < p$, то $\mu := \mu + 1$ и перейти к пункту 2, иначе перейти к пункту 8;
8) проверить значение ошибки E: если $E < E_{max}$, то закончить вычисления, иначе присвоить E:=0, $\mu := 1$ и перейти к пункту 2.

Рассмотренный алгоритм является *последовательным*, так как предполагает обновление весов при поступлении очередного входного вектора. Возможен альтернативный вариант алгоритма, когда веса связей обновляются в точном соответствии с (8.56) и (8.58) после поступления всех векторов обучающей выборки. Такой режим обновления весов связей называют *блочным*.

Рассмотрим некоторые практические рекомендации по выбору ряда параметров алгоритма обратного распространения ошибки, влияющих на скорость обучения, сходимость решения, обобщающие возможности многослойного персептрона.

Начальные веса связей. Начальные веса связей представляют собой небольшие случайные числа, обеспечивающие для заданного входного вектора $\vec{x}(\mu)$ такое значение $net_i(\mu)$, при котором используется линейный участок сигмовидной функции преобразования. Обычно это достигается, если значения w_{ij} лежат в диапазоне $[-3/\sqrt{m}, +3/\sqrt{m}]$, где m – число входов соответствующего нейронного элемента [76]. Неправильный выбор начальных весов связей может приводить к решениям, соответствующим локальным минимумам ошибки персептрона, или к “попаданию” в область “плато”.

Коэффициент обучения. Коэффициент обучения η оказывает значительное влияние на скорость обучения многослойного персептрона и сходимость решения. Выбор η обычно осуществляют экспериментально. Относительно большие значения η обеспечивают быстрое обучение, но при этом существует опасность получить неверное решение. Малые значения η затягивают процесс обучения. Диапазон возможных значений η от 10^{-3} до 10 [76]. Кроме этого, конкретные значения η, выбранные в начале процесса обучения, могут оказаться не совсем подходящими в конце обучения. Поэтому в ряде случаев η изменяют в ходе обучения. Один из способов управления значением η заключается в контроле ошибки E. Если

коррекция весов связей при выполнении алгоритма ОРО приводит к увеличению ошибки, то необходимо η уменьшить. И наоборот, если ошибка монотонно уменьшается, то необходимо увеличить η. В [68] приведено правило управления коэффициентом обучения

$$\Delta\eta = \begin{cases} +a, & \textit{если} \quad \Delta E < 0, \\ -b\eta, & \textit{если} \quad \Delta E > 0, \\ 0, & \textit{если} \quad \Delta E = 0, \end{cases}$$

где a и b - положительные константы.

Функции ошибок. Квадратическая функция ошибки (8.46) не является единственно возможной. В выражении (8.46) могут использоваться иные дифференцируемые функции ошибок $F(t_i, y_i)$, которые получают минимальные значения, когда их аргументы t_i и y_i равны.

Можно вывести соответствующие правила обучения, базирующие на этих функциях. При этом изменится только правило вычисления выходной ошибки (8.57). Остальные правила останутся без изменений. Одна из часто используемых функций ошибок представляет собой L_S норму ($1 \le S \le \infty$)

$$E = \frac{1}{S}\sum_{i=1}^{N} (t_i - y_i)^S .$$

В предельном случае ($S \to \infty$) получим ошибку, базирующуюся на применении нормы Чебышева

$$E_\infty = \max_i \left| t_i - y_i \right| .$$

В этом случае ошибка выходного слоя персептрона может быть вычислена с помощью формулы

$$\delta_i(\mu) = \begin{cases} 0, & i \ne i^*, \\ g'(net_i(\mu))\,\mathrm{sgn}(t_{i^*}(\mu) - y_{i^*}(\mu)), & i = i^*, \end{cases}$$

где i^* - номер нейрона , на выходе которого наблюдается наибольшая ошибка. При этом в обратном направлении распространяется ошибка только нейрона i^*, так как остальные ошибки считаются нулевыми.

Добавление момента инерции. Так как алгоритм ОРО базируется на использовании метода градиента, то при больших значениях параметра η, по мере приближения к минимуму E, возможны осцилляции. Для того

чтобы исключить осцилляции, в правило обновления весов связей (8.56) вводят момент инерции.

Предположим, что изменение веса $w_{ij}^{(k)}$ на текущем шаге соответствовало $\Delta w_{ij}^{(k)}(t)$. Для следующего момента времени изменение веса $\Delta w_{ij}^{(k)}(t+1)$ должно быть пропорционально градиенту $-\eta \partial E / dw_{ij}^{(k)}(t)$. Добавление момента инерции позволяет на каждом шаге частично сохранять предыдущее направление изменения весов связей

$$\Delta w_{ij}^{(k)}(t+1) = -\eta \frac{\partial E}{dw_{ij}^{(k)}(t)} + \alpha \Delta w_{ij}^{(k)}(t),$$

где $0 < \alpha < 1$.

Переобучение. Множество примеров X (8.41), содержащее пары $(\vec{x}(\mu), \vec{t}(\mu))$, разбивается на два подмножества X_1 и X_2. Для обучения используются только элементы подмножества X_1, которое позволяет сформировать обучающую последовательность, состоящую из пар $(\vec{x}(\mu), \vec{t}(\mu))$, где $1 < \mu < P$. Совокупность элементов из X_2 образует проверочную последовательность.

На рисунке 8.15 показан качественный характер зависимости ошибки нейронной сети на этапе обучения $E_{\text{обучения}}$ и на этапе проверки $E_{\text{проверки}}$ от количества циклов обучения n. Один цикл обучения соответствует выбору всех элементов из X_1.

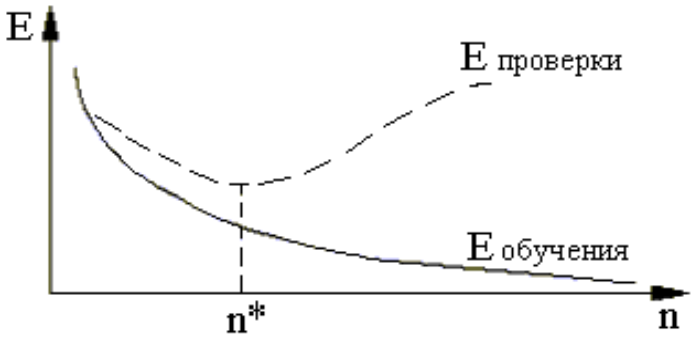


Рисунок 8.15. – Зависимость ошибки нейронной сети от количества циклов обучения.

Вначале обе ошибки уменьшаются. Это означает, что сеть корректно обучается на структуры данных, свойственные пространству входных сигналов. Однако если сеть является излишне гибкой (т.е. обладает большим количеством степеней свободы, чем требуется), то ошибка $E_{\text{проверки}}$ после некоторого числа циклов обучения начинает возрастать. Это означает, что при обучении сеть скорее учитывает особенности свойственные конкрет-

ному набору данных X_1, а не структурам данных, представляющих входные образы. Например, данные из X_1 может быть зашумлены, и сеть в этом случае обобщает свойства шума. Тогда говорят, что сеть “переобучилась”. В этой связи возникает вопрос о правильности структуры нейронной сети. Для контроля правильности структуры нейронной сети можно использовать зависимость, изображенную на рисунке 8.15. Если ошибка $E_{\text{проверки}}$ возрастает, то необходимо сократить число нейронов скрытых слоев ИНС. Вместе с тем, если обе ошибки уменьшаются, но остаются относительно большими к концу процесса обучения, то необходимо увеличить число нейронов скрытых слоев.

Альтернативным вариантом является запоминание весов связей, когда $E_{\text{проверки}}$ достигает минимума, и выдача этих весов в качестве решения. В этом случае структура сети не меняется. Такой способ обнаружения ситуации “переобучения” называют ранней остановкой. Однако это не означает, что обучение необходимо прекратить при $n = n^*$ (см. рисунок 8.15). Обучение следует продолжить, чтобы убедиться в том, что $E(n^*)$ не является локальным минимумом.

Алгоритм обратного распространения ошибки находит широкое применение при решении различных задач. Рассмотрим сеть Nettalk, базирующуюся на алгоритме обратного распространения ошибки. Сеть Nettalk предназначена для использования в системах синтеза речи по тексту. Она выполняет преобразование текста на английском языке в соответствующую последовательность фонем.

Решение данной задачи с помощью продукционных систем затруднено. Обусловлено это нерегулярностью правил произношения звуков английской речи. Поэтому системы синтеза английской речи (система DECtalk), основанные на правилах, значительно сложнее сети Nettalk [68]. На вход сети Nettalk поступает последовательность букв текста, а на выходе формируются соответствующие фонемы с указанием ударения. Так как произнесение буквы текста зависит от окружения, в котором находится буква, то на входе сети используется окно, фиксирующее окружение буквы. Окно перемещается по строкам текста и позволяет одновременно анализировать 7 букв.

Сеть Nettalk состоит из трех слоев (рисунок 8.16). Каждый знак текста, находящийся в окне, распознается соответствующим НЭ первого слоя. Для этого в первом слое сети имеется 7 групп НЭ, каждая из которых содержит 29 НЭ: 26 нейронов используются для распознавания букв и 3 – для знаков пунктуации. Выходной слой содержит 26 НЭ. Фонемы представляются с помощью 21 НЭ, а 5 оставшихся НЭ фиксируют ударение и границы слогов. Скрытый слой сети Nettalk содержит 80 НЭ. Всего в сети имеется 18529 соединений. В процессе обучения сети серединному знаку окна ставится в соответствие фонема. В одном из экспериментов обучаю-

щая последовательность состояла из 1024 слов. После 50 циклов обучения точность преобразования знаков в фонему составила 95%.

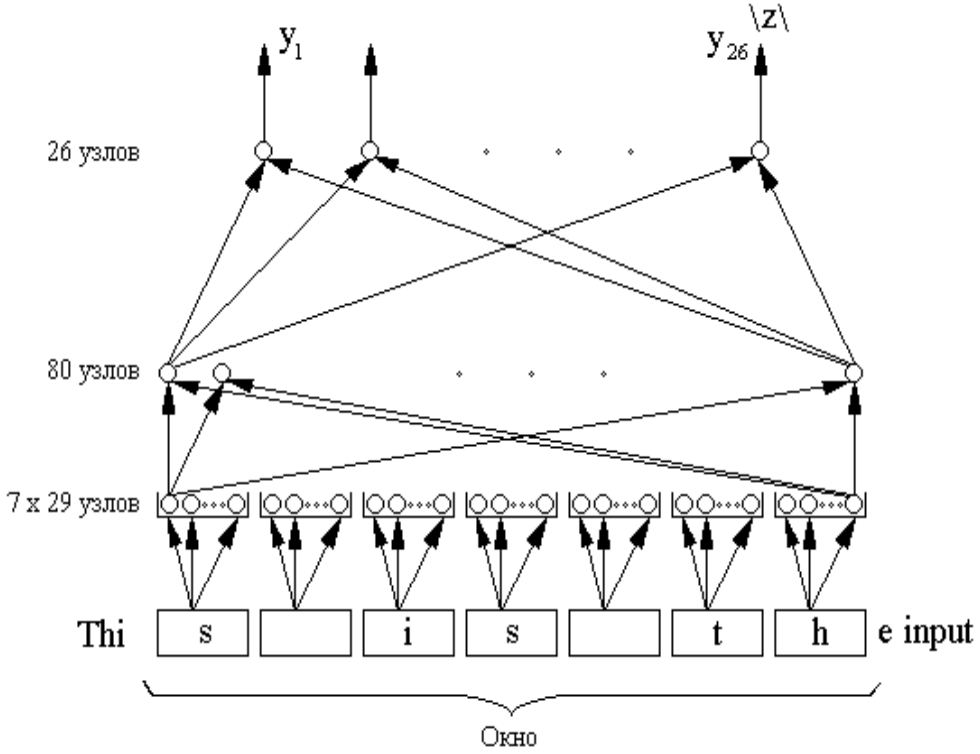


Рисунок 8.16 – Сеть Nettalk

8.5.5. Обучение без учителя

При обучении без учителя отсутствует информация о правильности реакции ИНС на тот или иной входной вектор. Нейронная сеть самостоятельно обнаруживает структурные отношения между элементами входных данных и преобразует их в соответствующую выходную реакцию, т.е. в ИНС происходят процессы самоорганизации. Самоорганизующиеся ИНС способны открывать отношения подобия на основе входных данных, обеспечивая тем самым выделение во входном пространстве признаков классов и категорий. Эта способность оказывается весьма полезной в тех областях, где необходимо автоматически формировать представления классов, категорий, опираясь на экспериментальные данные.

Рассмотрим два основных типа самоорганизующихся ИНС: ИНС, основанные на правиле обучения Хебба, и состязательные ИНС.

8.5.5.1. ИНС, основанные на правиле обучения Хебба. Правило обучения Хебба соответствует процессам, проходящим в биологических нейронных сетях. В 1949 году Хебб выдвинул следующую гипотезу: *если выходной сигнал нейрона А возбуждает нейрон В, и это постоянно по-*

вторяется, то вес связи между указанными нейронами увеличивается. Иными словами, он предположил, что веса синаптических связей в биологических нейронных сетях изменяются пропорционально корреляции между активностями нейрона *A* и нейрона *B*. Правило обучения Хебба можно записать в виде

$$\Delta w_{ij} = \eta\, y_i x_j, \qquad (8.61)$$

где y_i – выходной сигнал i-го нейрона; x_j – входной сигнал i-го нейрона, формируемый нейроном с номером j. Изменение веса связи в соответствии с (8.61) не требует привлечения сведений о желаемой реакции нейрона. Поэтому это правило позволяет нейрону самообучаться.

В общем случае правило обучения Хебба записывают в форме дифференциального уравнения

$$\dot{w}_{ij} = -w_{ij} + f(y_i)\cdot f(x_j), \qquad (8.62)$$

где $f(\cdot)$ – монотонная функция, которая ограничивает значение сигналов y_i и x_j. Дифференциальное уравнение (8.62) описывает изменение веса w_{ij} во времени. При этом второй член уравнения (8.62) определяет вынужденные изменения веса связи, обусловленные активностью нейронов, а первый член определяет уменьшение веса при отсутствии активности. Его называют *затуханием*. Таким образом, при отсутствии активности в сети веса связей затухают.

Правило Хебба в форме (8.61) не ограничивает увеличение весов связей и, следовательно, процесс обучения может продолжаться бесконечно долго. Поэтому на практике при построении самоорганизующейся ИНС применяют различные модификации правила (8.61). Например, правило Ойя содержит дополнительный член, обеспечивающий затухание весов пропорционально y_i^2. Правило Ойя для одиночного линейного НЭ записывается в виде [76]:

$$\Delta w_{ij} = \eta\, y_i\,(x_j - y_i w_{ij}). \qquad (8.63)$$

Это правило обеспечивает ограничение длины вектора связей i-го нейрона единичным значением, т.е. $|\vec{w}_i| = 1$.

Кроме этого, вектор весов $\vec{w}_i$ в ходе обучения приближается к максимальному собственному вектору корреляционной матрицы *C*, каждый

элемент которой равен $c_{ij} = E\{x_i x_j\}$, где $E\{.\}$ – символ математического ожидания. Последнее свойство широко применяется при анализе данных на основе метода главных компонент, широко используемому в статистике.

Правило Ойя, распространенное на однослойную сеть, состоящую из M линейных нейронных элементов, позволяет определить M главных пространственных компонент (направлений) распределения входных данных. Метод главных компонент сокращает размерность пространства, в котором представляются данные. Это обеспечивает широкое использование ИНС, основанных на правиле Ойя, и, в частности, на его модификации – правиле Сэнгера, в задачах классификации и сжатии изображений. Правило Сэнгера применительно к однослойной ИНС, состоящей из M линейных НЭ, записывается в виде:

$$\Delta w_{ij} = \eta y_i (x_j - \sum_{k=1}^{i} y_k w_{kj}) . \qquad (8.64)$$

В [76] показано, что в этом случае векторы весов соответствуют M главным компонентам. Правило Сэнгера не является локальным. При изменении веса w_{ij} требуется информация о весах других связей.

8.5.5.2. Состязательные ИНС. Главная особенность состязательных ИНС состоит в том, что в каждый момент времени в выходном слое сети выделяют один нейрон, имеющий наибольшее значение выходного сигнала. Данный нейрон рассматривают как нейрон-победитель, выигравший “состязание”. Структура простейшей состязательной сети изображена на рисунке 8.17.

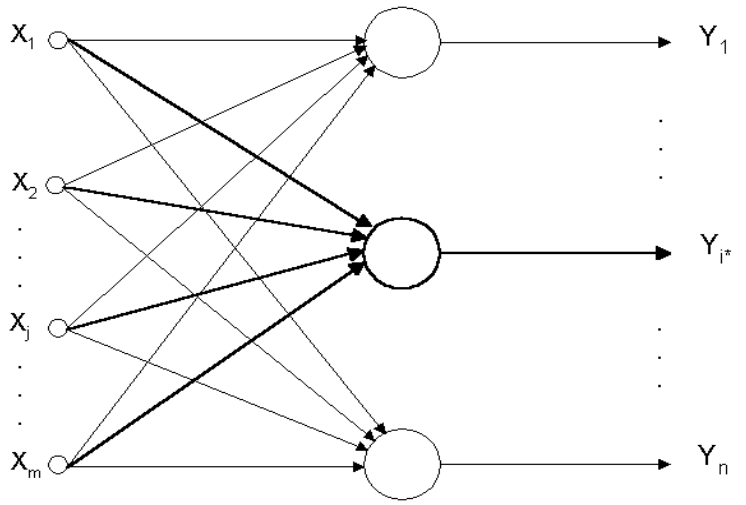


Рисунок 8.17 – Структура простейшей состязательной сети

Победителем считается нейрон с максимальным значением сетевой функции

$$net_i = \sum_{j=1}^{m} w_{ij} x_j = \vec{w}_i^T \vec{x} .$$

На рисунке 8.17 победителем является нейрон с номером i^*. Его связи изображены “жирными” линиями. Для нейрона-победителя i^* можно записать неравенство

$$\vec{w}_{i^*}^T \vec{x} \geq \vec{w}_i^T \vec{x} , \quad i = 1, 2, \ldots, n. \tag{8.65}$$

Если вектор $\vec{w}_i$ нормализован $(|\vec{w}_i| = 1)$, то неравенство (8.65) эквивалентно

$$|\vec{w}_{i^*} - \vec{x}| \leq |\vec{w}_i - \vec{x}| \tag{8.66}$$

Иными словами, вектор весов нейрона-победителя $\vec{w}_{i^*}$ менее всех отличается от текущего входного вектора $\vec{x}$. Таким образом, номер нейрона-победителя определяется с помощью выражения вида

$$i^* = \arg\min_i |\vec{x} - \vec{w}_i|, \tag{8.67}$$

если вектор $\vec{w}_i$ нормализован, или вида

$$i^* = \arg\max_i |\vec{w}_i^T \vec{x}|, \tag{8.68}$$

если вектор не нормализован. Последняя запись означает, что входной вектор $\vec{x}$ максимально коррелирует с вектором весов связей $\vec{w}_{i^*}$ нейрона-победителя.

Выражения (8.67) и (8.68) удобно использовать при компьютерном моделировании сети. В случае ее аппаратной реализации выбор нейрона-победителя может осуществляться на основе латерального торможения (рисунок 8.10). При этом латеральные связи нейрона-победителя обеспечивают торможение соседних нейронов сети.

Одним из простейших правил обучения, применяемым в состязательных сетях, является правило обучения Кохонена, в соответствии с которым в сети на основе соотношения (8.67) определяют нейрон-победитель, а затем изменяют веса связей только нейрона-победителя, используя выражение

$$\vec{w}_i(t+1) = \vec{w}_i(t) + \eta(t)(\vec{x} - \vec{w}_i(t)), \quad i = i^*, \tag{8.69}$$

где $\eta(t)$ – значения коэффициента обучения в момент времени t. Веса связей всех остальных нейронов не изменяются. В (8.69) предполагается, что векторы $\vec{w}_i$ и $\vec{x}$ нормализованы.

Заметим, что интегральный эффект от применения правила (8.69) к весам связей i-го нейрона связан с числом соответствующих событий, когда этот нейрон был победителем. Таким образом, состязательная ИНС, использующая (8.69), должна реагировать на вероятность распределения входных образов. Поэтому входные векторы, встречаемые редко, окажут меньшее влияние на конечное значение вектора весов $\vec{w}_i$.

Если записать скалярное произведение $\vec{w}_i^T \cdot \vec{x}$ в виде

$$\vec{w}_i^T \vec{x} = |\vec{w}_i| \cdot |\vec{x}| \cdot \cos(w_i, \vec{x}),$$

то можно предложить следующую геометрическую интерпретацию (8.69). Так как в сети отыскивается максимум скалярного произведения, то победителем в сети будет нейрон с наименьшим значением угла между векторами $\vec{w}_i$ и $\vec{x}$. Поэтому применение правила (8.69) сводится к повороту вектора весов $\vec{w}_i$ нейрона-победителя по направлению к входному вектору $\vec{x}$.

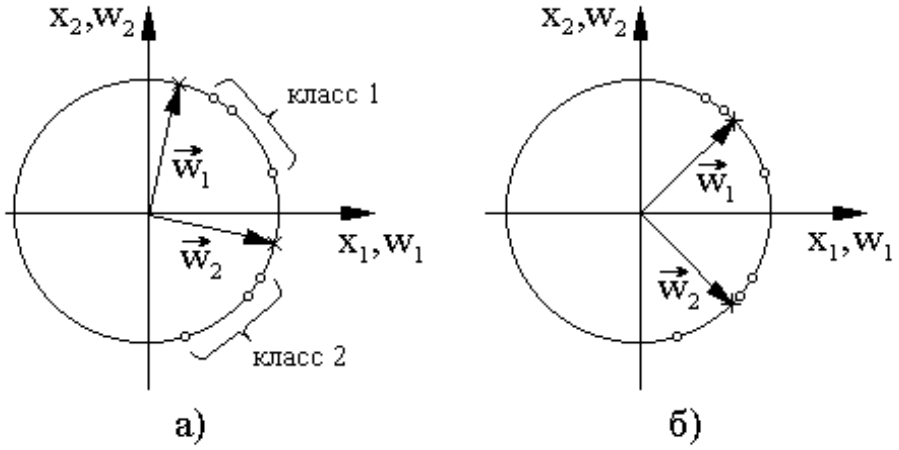


Рисунок 8.18 – Состязательное обучение

Рассмотрим двухвходовую состязательную ИНС с двумя нейронами. Начальное положение векторов весов $\vec{w}_1$ и $\vec{w}_2$ изображено на рисунке 8.18,а. Кружочками на рисунке отмечено положение входных образов, ко-

торые могут быть разделены на два класса. В ходе обучения векторы весов поворачиваются в направлении поступающих входных векторов.

Если на вход сети подается вектор $\vec{x}$, относящийся к первому классу, то победителем оказывается первый нейрон, и подстраиваются только веса первого нейрона. Если на вход сети подается вектор из второго класса, то подстраивается вектор весов второго нейрона. Благодаря указанному процессу нейроны сети обнаруживают классы, к которым относятся входные векторы, настраиваясь на центр их "гравитации" (рисунок 8.18,б). Теперь вектор весов первого нейрона является обобщенным вектором, представляющим первый класс входных сигналов, а вектор весов второго нейрона – обобщенным вектором, представляющим второй класс входных сигналов.

Правило обучения Кохонена требует задания начальных весов связей и коэффициента обучения $\eta(t)$. Часто веса связей инициализируются случайными значениями с равномерным законом распределения. Однако это может приводить к тому, что некоторые из нейронов сети никогда не окажутся победителями и, следовательно, не будут участвовать в процессе обучения. Другой способ задания начальных связей основан на инициализации их обучающими векторами $\vec{x}$, т.е. $\vec{w}_1 = \vec{x}(1)$, $\vec{w}_2 = \vec{x}(2)$, …, $\vec{w}_N = \vec{x}(N)$. Это гарантирует, что векторы $\vec{w}_i$ будут ориентированы в правильном направлении. Существуют и другие способы инициализации векторов связей [73,76].

Значение коэффициента $\eta(t)$ лежит в диапазоне $0<\eta(t)<1$. В начале обучения $\eta(t)$ имеет большие значения, которые уменьшаются в ходе обучения. Часто $\eta(t)$ вычисляют с помощью выражения вида

$$\eta(t) = 1/t, \quad t = 1,2,\ldots \tag{8.70}$$

или вида

$$\eta(t) = \eta_0 \cdot t^{-\beta} = \frac{\eta_0}{\sqrt[\beta]{t}}, \tag{8.71}$$

где $\beta \le 1$.

Правило Кохонена можно также использовать при обучении с учителем, когда принадлежность входных образов классам известна заранее. В этом случае в правиле (8.69) полагают $\eta > 0$, если предъявленный образ принадлежит классу, на который настраивается нейрон, и $\eta < 0$, если образ не принадлежит классу. Такое правило обучения называют *правилом векторного квантования:*

$$\Delta\vec{w}_{i^*} = \begin{cases} +\eta \quad (\vec{x} - \vec{w}_{i^*}), & \textit{если} \quad i^* \in \textit{класс}, \\ -\eta \quad (\vec{x} - \vec{w}_{i^*}), & \textit{если} \quad i^* \notin \textit{класс}. \end{cases} \tag{8.72}$$

Векторное квантование широко используется при сжатии данных. В этом случае множество входных векторов $\vec{x}(\mu)$ разбивается на n классов, а затем каждый входной вектор представляется номером класса, к которому он относится. При этом для хранения или передачи данных используют только номер класса. Конечно, при восстановлении данных необходимо иметь кодовую книгу, содержащую обобщенные векторы, представляющие классы.

Модифицированное правило Кохонена используется также при построении самоорганизующихся карт признаков. Карта признаков формируется сетью, отображающей образы из входного *n*-мерного пространства в одно- или двумерное пространство. Кроме сокращения размерности пространства, карта признаков обеспечивает сохранение отношений топологического соседства входных образов. При этом геометрическая близость выходных классов на карте признаков означает близость соответствующих образов во входном пространстве признаков.

Одна из возможностей построения самоорганизующихся ИНС, обеспечивающих формирование указанной карты признаков, основана на применении правила (8.69) не только к весам победившего нейрона, но и к весам соседних нейронов. Формально правило Кохонена для самоорганизующихся карт признаков записывается в виде

$$\vec{w}_i(t+1) = \begin{cases} \vec{w}_i(t) + \eta(t)[\vec{x} - \vec{w}_i(t)], & \textit{для} \quad i \in N_{i^*}(t) \\ \vec{w}_i(t), & \textit{для} \quad i \notin N_{i^*}(t), \end{cases} \tag{8.73}$$

где $N_{i^*}(t)$ – множество нейронов, соседних по отношению к нейрону-победителю i^*.

При использовании этого правила веса связей нейронов, входящих во множество $N_{i^*}(t)$, получают одинаковые приращения. На практике множество $N_{i^*}(t)$ и коэффициент обучения изменяют свои значения динамически в процессе обучения. Процесс обучения начинается при расширенном множестве $N_{i^*}(t)$ и больших значениях $\eta(t)$. По мере обучения количество НЭ, включаемых в $N_{i^*}(t)$ уменьшается, одновременно уменьшается и значение $\eta(t)$. Изменение $\eta(t)$ осуществляется на основе (8.70) или (8.71), а сужение множества $N_{i^*}(t)$ выполняется в соответствии с топологическими схемами, изображенными на рисунке 8.19 [73]. При этом

$N_{i^*}(t_3) \subset N_{i^*}(t_2) \subset N_{i^*}(t_1)$, *где* $t_1 < t_2 < t_3 \ldots$. Здесь полагается, что нейронная сеть представляет собой двумерный массив из НЭ.

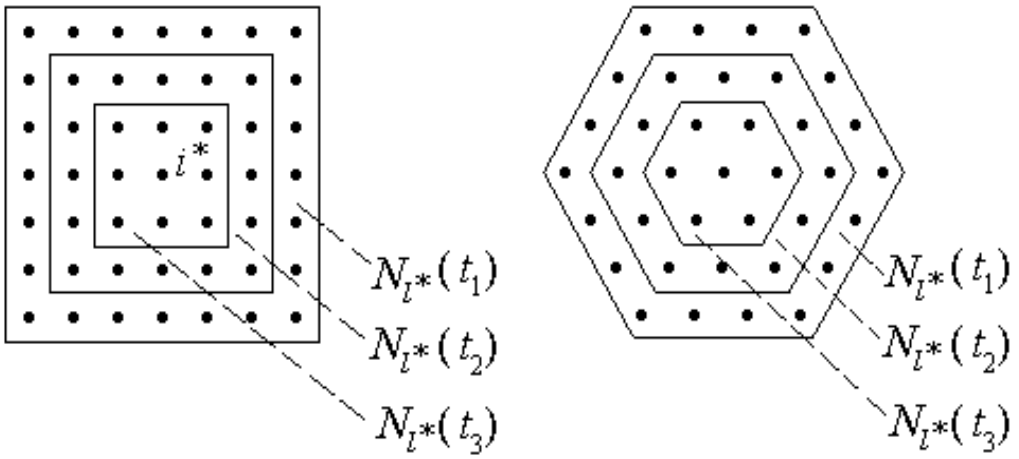


Рисунок 8.19 – Схемы топологического соседства

Множество $N_{i^*}(t)$ может быть задано с помощью функции соседства $\Lambda(i, i^*)$, которая равна 1 при $i = i^*$ и уменьшается по мере увеличения "расстояния" от нейрона i^* до нейрона i в выходном топологическом пространстве. В этом случае правило (8.73) записывается в виде:

$$\Delta \vec{w}_i = \eta(t) \cdot \Lambda(i, i^*)(\vec{x} - \vec{w}_i). \qquad (8.74)$$

Функция $\Lambda(i, i^*)$ может быть задана выражением

$$\Lambda(i, i^*) = \exp(-\left|r_i - r_{i^*}\right|^2 / 2\sigma^2),$$

где $r_i - r_{i^*}$ – расстояние между нейронами i и i^* в выходном двумерном пространстве; σ – параметр, задающий "диаметр" функции соседства. Параметр σ уменьшается во времени, например $\sigma(t) = 1/t$.

В случае применения правила (8.74) обучаются те нейроны i, для которых функция геометрического соседства $\Lambda(i, i^*)$ имеет относительно большие значения. Так как $\sigma(t)$ уменьшается в ходе обучения, то со временем "диаметр" функции $\Lambda(i, i^*)$ становится все меньше и, следователь-

но, одновременно с нейроном i^* обучается меньшее количество нейронов. В предельном случае обучается только нейрон i^*.

Самоорганизующиеся карты признаков находят широкое применение в самых разных областях, таких, как распознавание речи, сенсорные системы роботов, комбинаторная оптимизация, управление двигателями и др. [73]

Рассмотрим пример построения самоорганизующейся карты признаков Кохонена, используемой в системах распознавания речи [73]. На вход сети, организованной в виде двумерного массива НЭ, подается вектор $\vec{x}$, состоящий из 15 компонент. Каждая компонента вектора $\vec{x}$ соответствует энергии сигнала на выходе одного из 15 полосовых фильтров (рисунок 8.20) и вычисляется на интервале 10 мс. Полосы пропускания фильтров находятся в диапазоне от 200 Гц до 6400 Гц. Таким образом, компоненты вектора $\vec{x}$ представляют спектральные коэффициенты речевого сигнала. После обучения сети, выполняемому с помощью правила (8.73), группы нейронов становятся чувствительными к определенным фонемам. При этом близкие по звучанию фонемы занимают соседние позиции на двумерной схеме расположения НЭ.

На рисунке 8.21 изображена фонетическая карта, получаемая приписыванием нейронам сети соответствующих меток, обозначающих фонемы. Данная карта соответствует финскому языку, который фонетически близок к латинскому. Нейроны, обозначенные меткой #, соответствуют фонемам $|k, p, t|$, которые неуверенно выделяются рассмотренной сетью.

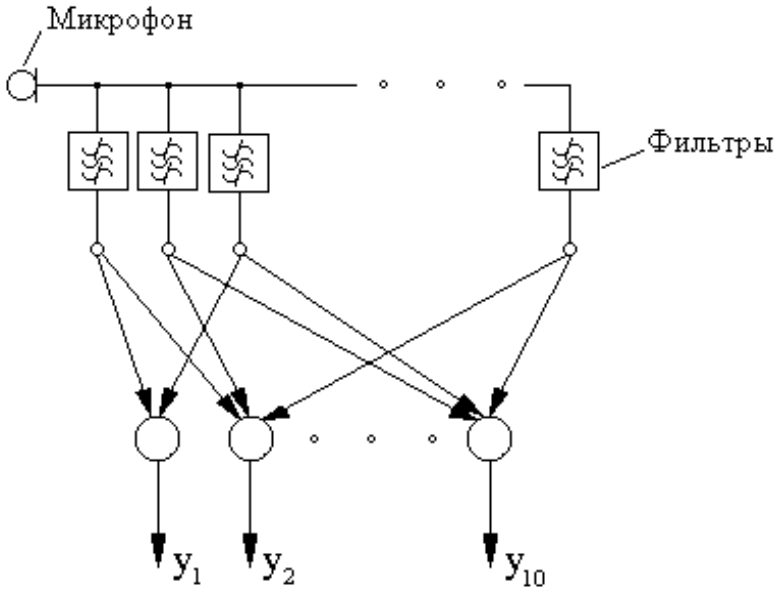


Рисунок 8.20 – ИНС, распознающая фонемы

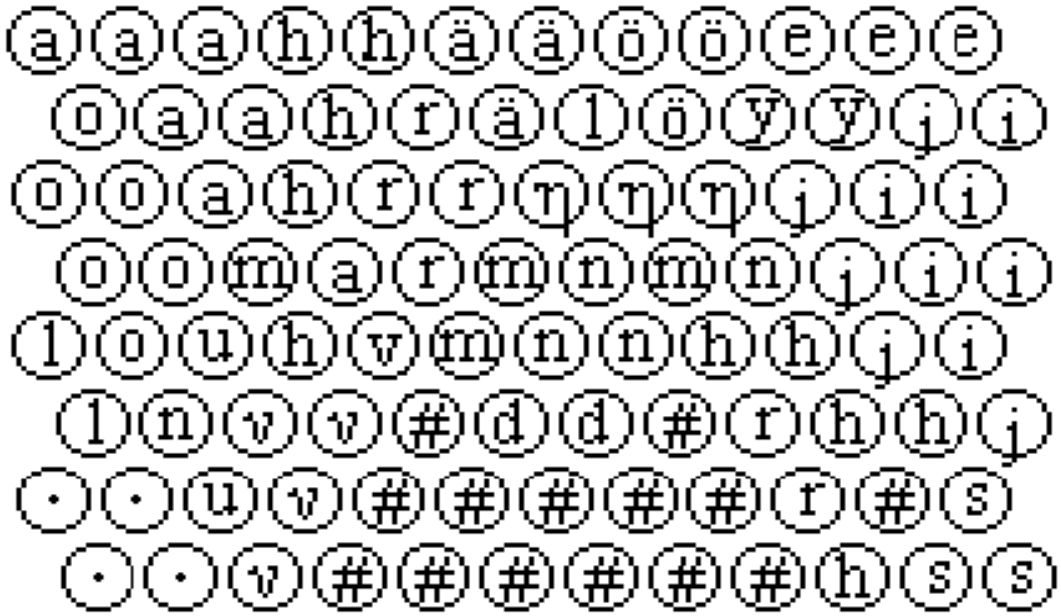


Рисунок 8.21 – Отображение фонем на двумерной схеме ИНС

В [73] отмечается, что правило (8.69) не гарантирует *устойчивости* классов, обнаруживаемых состязательной ИНС, если не обеспечить уменьшение коэффициента обучения до нуля. При этом сеть теряет *гибкость* – способность реагировать на новые данные. В состязательной ИНС, обучаемой на основе правила (8.69), трудно обеспечить необходимую гибкость сети и устойчивость формируемых классов. Данная проблема решается в состязательных ИНС, использующих *теорию адаптивного резонанса* (АРТ), которая была предложена С. Гроссбергом в 1976 году.

В сетях АРТ дилемма гибкость – устойчивость решается благодаря тому, что сохраняемые в сети векторы-прототипы классов модифицируются только в том случае, если входные векторы “достаточно близки” к этим прототипам. При этом говорят, что входной вектор и прототип находятся в резонансе друг с другом.

Если между входным вектором и ни одним из прототипов, сохраненных в сети, не наблюдается достаточного сходства (подобия), то в структуру сети вводится новый НЭ, вектор весов которого и будет в дальнейшем рассматриваться как новый прототип. Понятие “достаточное сходство” зависит от параметра $0 < \rho \le 1$, который называется *бдительностью.* Чем больше значение ρ, тем точнее должно быть сходство. Большие значения ρ приводят к формированию в сети большего количества классов. Малые значения ρ формируют небольшое число прототипов и приводят к “грубой” классификации.

Существуют две основные разновидности сетей АРТ: ART1 и ART2. Сети ART1 работают с бинарными входными значениями (0/1), а сеть ART2 – с аналоговыми.

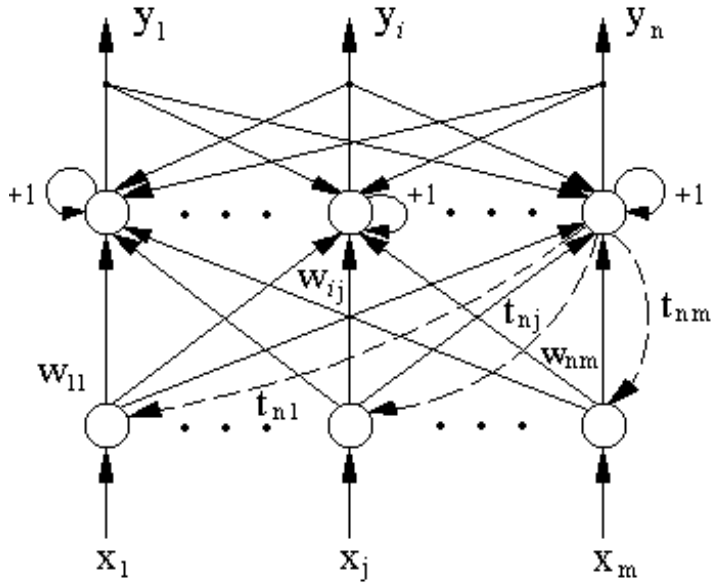


Рисунок 8.22 – Сеть, функционирующая на основе алгоритма ART1

На рисунке 8.22 изображена сеть ART1. В отличие от простейшей состязательной сети, изображенной на рисунке 8.17, в сети ART1 имеются прямые и обратные связи. Веса прямых связей w_{ij} представляют собой нормализованные копии весов обратных связей t_{ji} . На рисунке 8.22 обратные связи изображены штриховыми линиями и показаны только для одного выходного НЭ. В сети ART1 имеются также латеральные связи, обеспечивающие реализацию принципа состязательности НЭ. Латеральные связи i-го НЭ, направленные к соседним НЭ, являются тормозящими. Благодаря латеральным связям в каждый момент времени победителем оказывается НЭ выходного слоя с максимальным значением скалярного произведения $\vec{w}_i^T \vec{x}$, где $\vec{x}$ – входной бинарный вектор, компоненты которого принимают значения 0 и 1. Отметим, что бинарными являются также реакции y_i выходных НЭ и веса обратных связей t_{ji}

Векторы весов обратных связей $\vec{t}_i = (t_{i1}, t_{i2}, ..., t_{im})^T$ рассматриваются как векторы-прототипы, сохраняемые в сети. В рассматриваемой сети выполняется сопоставление векторов $\vec{t}_i$ с вектором $\vec{x}$, представляющим входной образ. Если сопоставление выполняется успешно (т.е. между вектором $\vec{x}$ и одним из векторов $\vec{t}_i$ имеет место “достаточное сходство”), то соответ-

ствующий вектор $\vec{t}_i$ представляет класс, к которому относится входной вектор.

Рассмотрим алгоритм обучения сети ART1. В начале обучения $t_{ji} = 1$ и $w_{ij} = 1/(1+m)$. В ходе обучения можно запрещать некоторым НЭ выходного слоя участвовать в состязательном процессе. Для этого используются две метки: “доступен” и “недоступен”. Алгоритм предусматривает выполнение следующих действий [68].

1. Подать на вход сети очередной вектор $\vec{x}$.
2. Снабдить все НЭ выходного слоя меткой ”доступен”.
3. Выполнить взвешивание элементов входного вектора $\vec{x}$

$$net_i = \sum_{j=1}^{m} w_{ij} \cdot x_j ,$$

где w_{ij} - нормализованные веса связей

$$w_{ij} = \frac{t_{ij}}{0.5 + \sum_{j=1}^{m} t_{ij}} .$$

4. По максимальному значению сетевой функции net_i выбрать нейрон-победитель i^*, и выполнить присвоение $y_{i^*} = 1$.
5. Выполнить сопоставление входного вектора $\vec{x}$ и прототипа $\vec{t}_{i^*}$, вычислив отношение

$$r = \frac{\vec{t}_{i^*}\vec{x}}{\sum_{j=1}^{m} x_j} = \frac{\sum_{j=1}^{m} t_{i^* j} x_j}{\sum_{j=1}^{m} x_j} . \qquad (8.75)$$

Данное отношение определяет подобие битовых структур векторов $\vec{t}_{i^*}$ и $\vec{x}$, устанавливая относительное количество совпадающих единичных битов векторов $\vec{t}_{i^*}$ и $\vec{x}$ по сравнению с количеством единичных битов вектора $\vec{x}$.

6. Если $r \geq \rho$, где ρ – параметр бдительности, то имеет место резонанс $\vec{x}$ и $\vec{t}_{i^*}$; перейти к пункту 7.

Если $r < \rho$ и в выходном слое имеются еще НЭ, отмеченные меткой "доступен", то отметить НЭ i^* меткой "недоступен", положить $y_{i^*} = 1$, перейти к пункту 4.

Если в выходном слое все нейроны отмечены меткой "недоступен", то ввести в выходной слой новый НЭ i^* с начальными весами $w_{ij} = 1/(1+m)$ и $t_{ji} = 1$.

7. Обновить веса связей НЭ i^*, используя правило

$$t_{i^* j}(t+1) = t_{i^* j}(t) \wedge x_j . \qquad (8.76)$$

При заданном входном векторе $\vec{x}$ алгоритм АРТ1 может иметь два исхода. Если сопоставление векторов $\vec{t}_{i^*}$ и $\vec{x}$ успешно, то вектор $\vec{t}_{i^*}$ обновляется в соответствии с правилом (8.76), и на выходе сети активным будет НЭ i^*, представляющий класс, к которому отнесен входной сигнал.

Если в сети не обнаруживается подходящих векторов-прототипов $\vec{t}_i$, то в выходной слой вводится дополнительный НЭ i^*, вектор весов связей которого $\vec{t}_{i^*}$ будет равен входному вектору $\vec{x}$ (см.(8.76)). Данный нейрон будет представлять новый класс. На практике в выходной слой сети ART1 заранее включают определенное число НЭ, которые активизируются, если в сети не находится ни одного НЭ, вектор весов которого сопоставим с входным вектором. Если рассматривать вектор-прототип $\vec{t}_{i^*}$ как строку битов, то правило (8.76) обеспечивает только удаление единичных битов из этой строки в зависимости от значений $\vec{x}$. Добавление единичных битов в $\vec{t}_{i^*}$ никогда не происходит. Это означает, что вектор-прототип не может повторно получать одни и те же значения. Данное свойство гарантирует устойчивость алгоритма, так как после конечного числа предъявлений входных векторов $\vec{x}$ веса связей перестают изменяться. В то же время расширение сети ART1 за счет дополнительных НЭ обеспечивает ее необходимую гибкость.

В заключение отметим, что оригинальная сеть ART1, разработанная Гроссбергом, была аналоговой, и ее поведение описывалось дифференциальными уравнениями. Сеть хорошо выполняла классификацию бинарных входных сигналов. Однако она оказалась чувствительной к шумам, содержащимся во входных данных.

8.5.5.3. Неокогнитрон. Рассмотренные выше состязательные ИНС являются однослойными. В результате состязательного процесса, протекающего в этих сетях, для каждого входного вектора $\vec{x}$ в выходном слое выбирается только один НЭ, который считается победителем. Если допус-

тить существование в одном слое нескольких нейронов – победителей, то можно построить многослойную самоорганизующуюся (состязательную) ИНС, обеспечивающую выделение обобщенных признаков классов на уровне каждого слоя.

В качестве примера многослойной иерархической ИНС, использующей состязательный принцип обучения, рассмотрим *неокогнитрон*, разработанный К.Фукушимой и С.Миякой. Неокогнитрон изначально предназначался для распознавания рукописных знаков, хотя его авторы подчеркивали, что они заинтересованы в разработке модели зрительной системы человека [38]. Поэтому принципы, положенные в основу неокогнитрона, базировались на исследованиях Д. Хьюбела [47] и Т. Визела в области функциональной архитектуры нейронов зрительной коры (см. главу 9).

Структурная схема неокогнитрона изображена на рисунке 8.23, где каждый прямоугольник представляет собой двумерный массив нейронов, который называют плоскостью. Неокогнитрон состоит из девяти слоев, которые содержат различное число плоскостей НЭ. Числа, указанные внизу слоя, показывают размерность плоскости НЭ и количество плоскостей, входящих в слой. Входной слой U_o образован из фоточувствительных элементов и называется сетчаткой. Каждый следующий уровень неокогнитрона состоит из двух слоев, обозначаемых U_s и U_c. Слой U_s содержит *S*-нейроны с изменяемыми весами связей, а слой U_c – *C*-нейроны с фиксированными весами связей. НЭ слоя U_{s1} соединены с небольшими участками сетчатки, представляющими рецептивные поля. При этом соединения организованы так, чтобы НЭ, находящиеся на одних и тех же позициях в разных массивах (плоскостях) слоя, были связаны с одним и тем же рецептивным полем сетчатки. Нейроны первого *S*-слоя реагируют на простые признаки предъявляемых изображений, например, прямолинейные отрезки, имеющие разную ориентацию на плоскости. Каждый *S*-нейрон одной и той же плоскости реагирует на один и тот же признак, который может локализоваться в различных позициях входной плоскости. *S*-нейроны разных плоскостей реагируют на разные признаки. Таким образом, множество плоскостей слоя U_{s1} позволяет выделить в исходном изображении, проецируемом на сетчатку, 12 различных признаков. Причем эти признаки могут обнаруживаться на любом участке сетчатки.

НЭ элементы *C*-слоя интегрируют реакции группы нейронов предыдущего *S*–слоя. При этом реакция *C*-нейрона оказывается менее чувствительна к локализации обнаруживаемого признака во входном слое. Это свойство позволяет неокогнитрону распознавать знаки вне зависимости от их расположения на сетчатке.

Если количество *C*-плоскостей равно количеству *S*-плоскостей предыдущего слоя, то каждый *C*-нейрон связан с группой нейронов только одной *S*-плоскости. Если количество *C*-плоскостей меньше количества

плоскостей предыдущего *S*-слоя, то *C*-нейрон может быть связан с несколькими разными *S*-плоскостями.

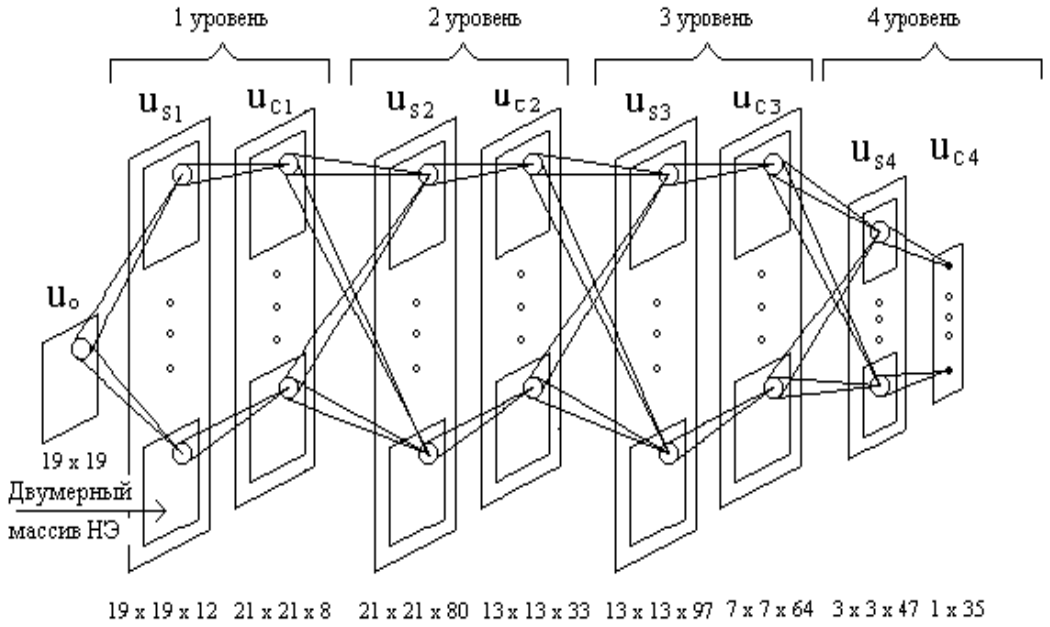


Рисунок 8.23 – Структурная схема неокогнитрона

Отметим некоторое отличие между первым *S*-слоем и последующими *S*-слоями. Нейроны первого *S*-слоя соединены с сетчаткой и воспринимают первичные признаки. Нейроны последующих *S*-слоев связаны с соответствующими группами нейронов, расположенными в разных *C*-плоскостях предыдущего слоя, и воспринимают некоторые обобщенные признаки.

На рисунке 8.24 изображена детальная схема связей нейронов в неокогнитроне. С каждым *S*-нейроном связан добавочный *V*-нейрон, на вход которого подаются сигналы того же рецептивного поля. Выходной сигнал *V*-нейрона является тормозящим для *S*-нейрона. В процессе обучения веса связей *S*-нейрона изменяются по правилу Хебба. Ранее отмечалось, что правило Хебба может приводить к неограниченному увеличению весов связей и реакции линейного НЭ. Введение дополнительного *V*-нейрона, оказывающего тормозящее воздействие, ограничивает реакцию связанного с ним *S*-нейрона.

Наличие нескольких уровней обработки позволяет неокогнитрону интегрировать простейшие локальные признаки, обнаруженные на нижних уровнях, в более сложные признаки, выделяемые на последующих уровнях. Нейроны последнего *C*-слоя интегрируют информационные признаки, выделяемые на нижних уровнях, и распознают конкретный образ.

Обучение неокогнитрона можно выполнять в режиме обучения с учителем или без учителя. Рассмотрим режим обучения без учителя. В

этом случае удобно представить все *S*-плоскости одного слоя в виде стопки плоскостей, расположенных вертикально одна над другой. При таком представлении *S*-нейроны, реагирующие на одни и те же локальные области предыдущего слоя, будут расположены один над другим, образуя столбец *S*-нейронов.

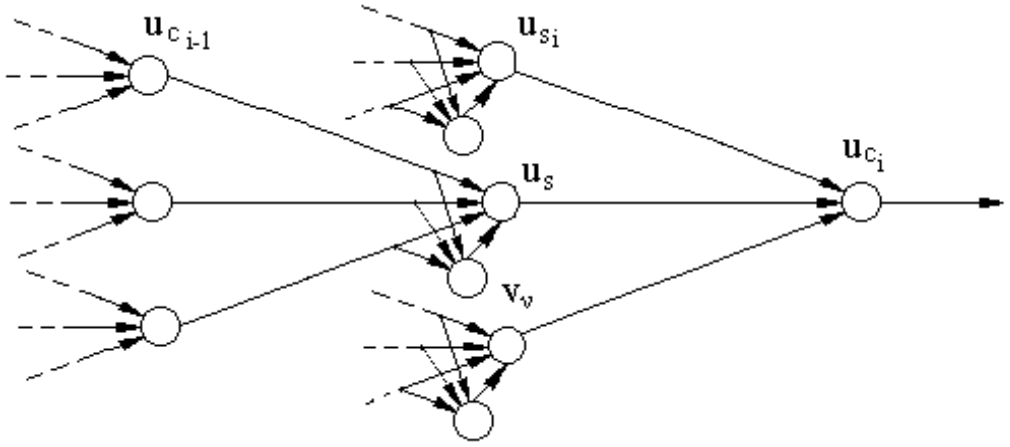


Рисунок 8.24 – Связи НЭ неокогнитрона

В начале обучения веса возбуждающих связей инициализируется небольшими положительными случайными числами, а весам тормозящих связей присваиваются нулевые значения. После предъявления очередного изображения, в каждом столбце *S*-нейронов выбирается нейрон-победитель с максимальной реакцией. Затем анализируются *S*-нейроны одной плоскости. Если на *S*-плоскости имеются два или более нейрона-победителя, то среди них выбирается нейрон с наибольшей реакцией. Такая двухэтапная процедура поиска нейрона-победителя позволяет выбрать один *S*-нейрон с наиболее сильной реакцией в пределах плоскости. При этом нейроны-победители каждой из *S*–плоскостей должны принадлежать разным столбцам. Иными словами, каждый нейрон-победитель должен быть связан со своим рецептивным полем. Веса связей нейрона-победителя каждой из *S*-плоскостей увеличиваются пропорционально входному сигналу. В дальнейшем все нейроны одной и той же *S*-плоскости получают веса нейрона-победителя. Данный способ назначения весов связей называют *механизмом разделения весов*. Этот механизм обеспечивает настройку всех *S*-нейронов одной и той же плоскости на одну и ту же совокупность признаков, выделенных в пределах рецептивного поля нейрона. При этом каждый из *S*-нейронов является “ответственным” за свой участок входной плоскости.

В заключение отметим, что неокогнитрон может включать обратные связи, что позволяет строить на основе неокогнитрона системы ассоциативной памяти, которые способны при вводе части запомненных изображений восстанавливать все изображение.

8.5.6. ИНС с фиксированными весами связей и ассоциативная память

8.5.6.1. Ассоциативная память и сеть Хэмминга. Известны два способа доступа к информации, хранящейся в запоминающем устройстве: адресный и ассоциативный. *Адресный* способ характеризуется тем, что искомая информация жестко задана адресом – местом ее расположения в памяти. *Ассоциативный* способ доступа к информации основан на установлении соответствия (ассоциации) между хранимой в памяти информацией и поисковыми признаками. Содержимое ассоциативной памяти можно представить в виде множества пар

$$\{(\vec{x}_1,\vec{y}_1),(\vec{x}_2,\vec{y}_2),...,(\vec{x}_p,\vec{y}_p)\}, \qquad (8.77)$$

где $\vec{x}_1,\vec{x}_2,...,\vec{x}_p$ – входные векторы, задающие значения поисковых признаков образов; $\vec{y}_1,\vec{y}_2,...,\vec{y}_p$ – выходные векторы, соответствующие извлекаемым образам.

Ассоциативная память, реализуемая на основе ИНС, была независимо приложена Т. Кохоненом и Дж. Эндерсоном. Выделяют два основных типа ассоциативной памяти: *гетероассоциативную* и *автоассоциативную*. В первом случае входному вектору $\vec{x}$, который "достаточно близок" к вектору $\vec{x}_\mu$, хранимому в памяти, ставится в соответствие выходной вектор $\vec{y}_\mu$, где $\mu = 1,2,...P$. Во втором случае $\vec{x}_\mu = \vec{y}_\mu$ для всех пар из (8.77). Поэтому при подаче на вход запоминающего устройства вектора $\vec{x}$, "достаточно близкого" к вектору $\vec{x}_\mu$, на выходе формируется вектор $\vec{x}_\mu$.

Можно сказать, что ассоциативная память выполняет отображение образов входного пространства в выходное пространство. В случае автоассоциативной памяти размерности входного и выходного пространств совпадают, в случае гетероассоциативной памяти размерности указанных пространств различные. Разработка средств ассоциативной памяти – важное направление интеллектуализации современных ЭВМ [32].

Структурная схема линейной ассоциативной памяти (ЛАП) изображена на рисунке 8.25.

ЛАП представляет собой однослойную ИНС с прямыми связями. Важно отметить, что веса связей в такой сети являются фиксированными. Они вычисляются на этапе подготовки сети к работе. Цель функционирования сети заключается в восстановлении выходного образа на основе полной или неполной входной информации. Входные и выходные значения сети могут быть либо вещественными, либо бинарными.

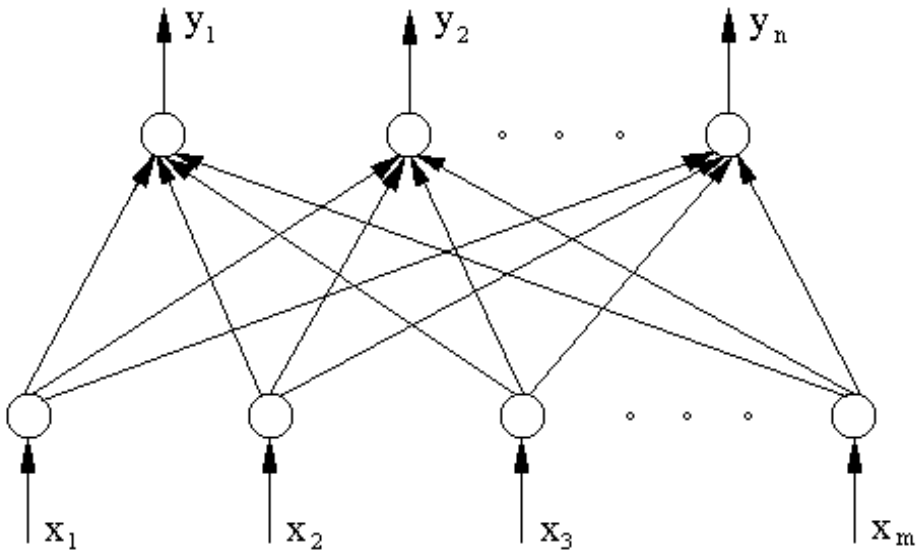


Рисунок 8.25 – Линейная ассоциативная ИНС

Матрица весов связей сети соответствует корреляционной матрице, устанавливающей взаимосвязь входных и выходных образов сети

$$W = \vec{y}_1\vec{x}_1^T + \vec{y}_2\vec{x}_2^T + ... + \vec{y}_\mu\vec{x}_\mu^T + ... + \vec{y}_p\vec{x}_p^T ,$$

где $\vec{y}_\mu\vec{x}_\mu^T$ – внешнее векторное произведение. Внешнее векторное произведение $\vec{y}\vec{x}^T$ задается матрицей

$$\vec{y}\vec{x}^T = \begin{bmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \dots & x_m \end{bmatrix} = \begin{bmatrix} y_1x_1 & y_1x_2 & \dots & y_1x_m \\ y_2x_1 & y_2x_2 & \dots & y_2x_m \\ \dots & & & \\ y_nx_1 & y_nx_2 & \dots & y_nx_m \end{bmatrix}.$$

Такое формирование матрицы весов связей вытекает из правила обучения Хебба, в соответствии с которым

$$w_{ij}(t+1) = w_{ij}(t) + \eta y_i(t)x_j(t) .$$

Предположим, что входные векторы ортогональны и нормализованы, т.е. $\vec{x}_\mu^T\vec{x}_\nu$ =0, если $\mu \neq \nu$, и $\vec{x}_\mu^T\vec{x}_\nu$ =1, если $\mu = \nu$. При подаче на вход сети вектора $\vec{x}_k$, неискаженного шумом, получим

$$W\vec{x}_k = \left(\sum_{\mu=1}^{p} \vec{y}_\mu \vec{x}_\mu^T\right)\vec{x}_k = \vec{y}_k .$$

Иными словами, на выходе сети будет восстановлен вектор $\vec{y}_k$, ассоциированный с входным вектором. Если предположить, что на вход сети подается вектор $\vec{x} = \vec{x}_k + \vec{\Delta}_k$, соответствующий искаженному вектору $\vec{x}_k$, то

$$W\vec{x} = (\sum_{\mu=1}^{P} \vec{y}_\mu \vec{x}_\mu^T)(\vec{x}_k + \vec{\Delta}_k) = \vec{y}_k + \sum_{\mu=1}^{P} \vec{y}_\mu (\vec{x}_\mu^T \vec{\Delta}_k), \qquad (8.78)$$

где $\vec{\Delta}_k$ – вектор шумов.

Исключив второе слагаемое в (8.78), получим на выходе опять вектор $\vec{y}_k$. Исключение второго слагаемого (8.78), представляющего *перекрестный шум*, требует введения в рассматриваемую ИНС нелинейных операций, которые будут рассмотрены ниже.

Если входные и выходные векторы являются униполярными бинарными векторами со значениями 0 и 1, то матрица весов связей определяется выражением вида [74]

$$w_{ij} = \sum_{\mu=1}^{P} (2y_i(\mu) - 1)(2x_j(\mu) - 1), \qquad (8.79)$$

где $y_i(\mu)$ и $x_i(\mu)$ – соответственно *i*–ая и *j*–ая компоненты μ–го выходного и входного векторов. В этой формуле автоматически учитывается среднее значение двоичного признака, равное 0,5. Для того чтобы сформировать бинарный выходной вектор, каждый выходной НЭ сети сравнивает компоненты вектора $W\vec{x}_k$ с порогом

$$\theta_i = \frac{1}{2}\sum_{j=1}^{m} w_{ij}. \qquad (8.80)$$

Если

$$\sum_{j=1}^{m} w_{ij}x_j - \theta_i \geq 0,$$

то соответствующий НЭ формирует $y_i = 1$, в ином случае $y_i = 0$.

Рассмотрим пример. Пусть

$$\vec{x}_1 = [1,0,0,0], \quad \vec{y}_1 = [0,1,1],$$
$$\vec{x}_2 = [0,0,0,1], \quad \vec{y}_2 = [1,0,1].$$

Тогда матрица весов в соответствии с (8.79) будет равна

$$W = \begin{bmatrix} -2 & 0 & 0 & 2 \\ 2 & 0 & 0 & -2 \\ 0 & -2 & -2 & 0 \end{bmatrix}.$$

Отсюда вектор значений порогов, вычисляемый на основе (8.80), будет равен

$$\vec{\theta} = [0 \quad 0 \quad -2].$$

Если на вход сети подать вектор $\vec{x}_1$, то $W\vec{x}_1 = [-2 \quad 2 \quad 0]$. Сравнивая $W\vec{x}_1$ с пороговыми значениями, получаем на выходе вектор $\vec{y}_1 = [0 \quad 1 \quad 1]$, что соответствует правильному решению.

Если же на вход сети подать искаженный вектор $\vec{x}_1$, например $\vec{x} = [1 \quad 0 \quad 1 \quad 0]$, то $W\vec{x} = [-2 \quad 2 \quad 2]$, и на выходе сети будет сформирован бинарный вектор $\vec{y} = [0 \quad 1 \quad 0]$, который не соответствует ни одному из векторов-прототипов $\vec{x}_1$ и $\vec{x}_2$, сохраненных в памяти. В данном случае сеть формирует ошибочный выходной образ, что согласуется с (8.78). Таким образом, линейная ассоциативная ИНС не обеспечивает подавление перекрестных шумов и, следовательно, может использоваться только для восстановления предварительно запомненных образов.

Лучшие результаты восстановления можно получить при использовании ИНС Хэмминга, входные векторы которой являются бинарными. При подаче на вход такой сети искаженного вектора признаков $\vec{x}$ восстанавливается тот сохраненный вектор-прототип $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_p$, который ближе всего к вектору $\vec{x}$ в смысле расстояния Хэмминга. Расстояние Хэмминга H_D между двумя бинарными векторами равно количеству несовпадающих битов векторов. Например, расстояние Хэмминга между бинарными векторами $\vec{x}_1 = (1, 0, 0, 1)$ и $\vec{x}_2 = (1, 1, 0, 1)$ равно 1.

В отличие от ЛАП, строки матрицы весов связей сети Хэмминга представляют запомненные векторы-прототипы:

$$W_H = \begin{bmatrix} x_1(1) & x_2(1) & \ldots & x_m(1) \\ x_1(2) & x_2(2) & \ldots & x_m(2) \\ \ldots & & & \\ x_1(\mu) & x_2(\mu) & \ldots & x_m(\mu) \\ \ldots & & & \\ x_1(p) & x_2(p) & \ldots & x_m(p) \end{bmatrix},$$

где $x_1(\mu)$, $x_2(\mu)$, …, $x_m(\mu)$ – бинарные значения компонент μ–го вектора-прототипа $\vec{x}_\mu$. При подаче на вход сети Хэмминга бинарного вектора признаков $\vec{x}$ вычисляются сетевые функции

$$net(\mu) = \vec{x}^T \vec{x}_\mu = \sum_{i=1}^{m} x_i x_i(\mu) = m - H_D(\vec{x}, \vec{x}_\mu).$$

В сети Хэмминга имеется второй слой нейронов, который обеспечивает выбор нейрона-победителя среди НЭ первого слоя. Победителем считается НЭ первого слоя с максимальным значением сетевой функции $net(\mu)$ (минимальным значением $H_D(\vec{x}, \vec{x}_\mu)$). Это обеспечивает формирование на выходе сети индекса вектора-прототипа, ближайшего к вектору входных признаков. Зная указанный индекс, можно на выходе сети восстановить либо вектор $\vec{x}_\mu$, либо ассоциированный с ним вектор $\vec{y}_\mu$. Таким образом, сеть Хэмминга может быть использована при построении как автоассоциативной, так гетероассоциативной памяти.

8.5.6.2. Сеть Хопфилда. Рассмотренные выше ИНС представляли сети с прямыми связями. Иным важным типом ИНС являются сети с обратными связями. На рисунке 8.26 изображена однослойная сеть с симметричными обратными связями, называемая сетью Хопфилда. Различают два вида сетей Хопфилда: непрерывные и дискретные.

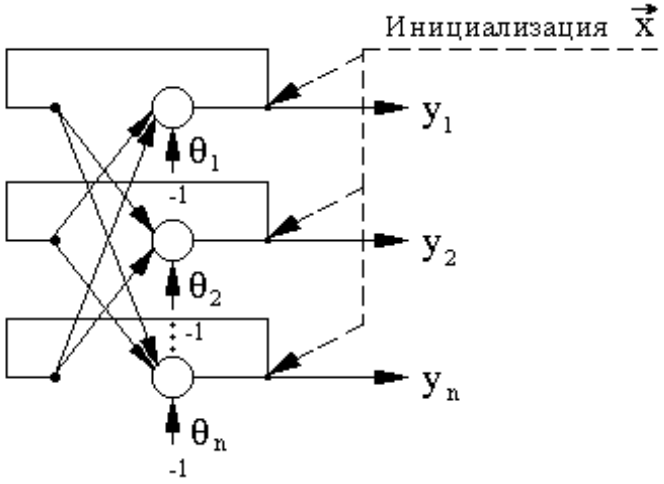


Рисунок 8.26 – Сеть Хопфилда.

Рассмотрим дискретную сеть Хопфилда, которая является простейшей рекуррентной сетью. В начальный момент времени выходные значения сети инициализируются входным вектором $\vec{x}$. Затем значения с выхода сети подаются на ее вход. Это приводит к установлению на выходе сети новых значений, которые опять поступают на вход и т.д. Указанный про-

цесс повторяется до тех пор, пока значения, циркулирующие в сети, не зафиксируются, т.е., пока сеть не перейдет в одно из возможных устойчивых состояний.

Отметим, что в сети Хопфилда $w_{ii} = 0$ и $w_{ij} = w_{ji}$. Выходные значения НЭ вычисляются по алгоритму

$$y_i(t+1) = \text{sgn}\left(\sum_{j=1}^{n} w_{ij} y_j(t) - \theta_i\right), \quad i = 1, 2, \ldots, n, \tag{8.81}$$

где t – номер шага вычислений.

Возможны два режима вычислений в сети: асинхронный и синхронный. В случае асинхронных вычислений в каждый момент времени устанавливается новое значение только на выходе одного НЭ. При этом на вход каждого вновь активизируемого НЭ подаются последние значения, установленные на выходах сети. Синхронное обновление выходных значений нейронов может приводить к возникновению предельных циклов [76]. Ниже рассмотрим только асинхронные сети.

Поведение сети Хопфилда определяется энергетической функцией

$$E(\vec{y}) = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij} y_i y_j + \sum_{i=1}^{n} \theta_i y_i \ . \tag{8.82}$$

Основное свойство энергетической функции заключается в том, что значение $E(\vec{y})$ уменьшается (или остается постоянным) при смене состояния любого узла сети. Покажем это. Пусть *i*–й нейрон переходит из состояния $y_i(t)$ в состояние $y_i(t+1)$. Тогда изменение значения энергетической функции составит [76]

$$\Delta E = E(y_i(t+1)) - E(y_i(t)) = -\left(\sum_{j=1}^{n} w_{ij} y_j(t) - \theta_i\right)(y_i(t+1) - y_i(t)) = -net_i \Delta y_i, \tag{8.83}$$

где $\Delta y_i = y_i(t+1) - y_i(t)$. При выводе выражения (8.83) учитывалось, что в асинхронной сети при смене состояния *i*–го НЭ остальные НЭ не меняют своих состояний, т.е. $y_j(t+1) = y_j(t)$.

Для того чтобы показать, что ΔE имеет отрицательные значения, рассмотрим два случая. Предположим, что выходной сигнал *i*–ого НЭ меняет свое значение с -1 на +1. Тогда значение net_i должно быть положительным, чтобы на выходе *i*–ого элемента установилось значение +1. Так как $\Delta y_i = 1 - (-1) = 2$, то из (8.83) следует, что $\Delta E < 0$.

Если предположить, что выходной сигнал *i*–го НЭ меняет свое значение с +1 на -1,то сетевая функция net_i должна иметь отрицательное значение, чтобы обеспечить указанную смену значений. В этом случае $\Delta y_i = -1 - 1 = -2$ и $\Delta E < 0$.

Таким образом, при смене состояния любого НЭ значение энергетической функции уменьшается. Устойчивые состояния, которые возникают в сети, соответствуют локальным минимумам энергетической функции.

С геометрической точки зрения энергетическую функцию можно рассматривать как некоторую холмистую поверхность, имеющую ложбины. Шар, установленный на вершине холма, будет стремиться занять устойчивое состояние, что приведет к скатыванию его в ложбину. Такие устойчивые состояния сети Хопфилда называют *аттракторами* (от англ. to attract – притягивать).

Наличие аттракторов позволяет использовать сеть Хопфилда в качестве автоассоциативной памяти. При этом сеть Хопфилда обеспечивает подавление перекрестных шумов и позволяет восстанавливать запомненные образы по неполным или зашумленным данным.

Веса связей сети Хопфилда, используемой для запоминания биполярных бинарных векторов $\vec{x}_\mu$, определяются выражением

$$w_{ij} = \sum_{\mu=1}^{p} x_i(\mu) x_j(\mu), \text{ где } i \neq j \text{ и } w_{ij} = 0. \qquad (8.84)$$

Выражение (8.84) соответствует автокорреляционной матрице, устанавливающей взаимосвязь координат запоминаемых векторов. Пороги θ_i имеют в этом случае нулевые значения.

Если входной вектор $\vec{x}_\mu$ является униполярным, т.е. $x_i(\mu) \in \{0,1\}$, то веса связей и значения порогов определяются формулами:

$$w_{ij} = \sum_{\mu=1}^{p} (2x_i(\mu) - 1)(2x_j(\mu) - 1), \text{ где } i \neq j \text{ и } w_{ij} = 0; \qquad (8.85)$$

$$\theta_i = \frac{1}{2} \sum_{j=1}^{n} w_{ij} . \qquad (8.86)$$

Правила назначения весов связей сети Хопфилда (8.84) и (8.85) аналогичны правилам, используемым в ЛАП, и вытекают из правила обучения Хебба.

Рассмотрим пример. Пусть требуется построить автоассоциативную память на основе ИНС Хопфилда для двух векторов:

$$\vec{x}(1) = [-1 \quad 1 \quad 1 \quad -1]^T \quad \text{и} \quad \vec{x}(2) = [1 \quad -1 \quad 1 \quad -1]^T .$$

Используя (8.84), вычислим матрицу весов связей сети Хопфилда

$$w = \begin{bmatrix} 0 & -2 & 0 & 0 \\ -2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -2 \\ 0 & 0 & -2 & 0 \end{bmatrix}.$$

Предположим, что на вход сети подан вектор $\vec{x} = [1 \quad -1 \quad 1 \quad 1]^T$. Тогда в соответствии с (8.81) будем получать на выходе ИНС следующие результаты:

- первая итерация:
  начальный вектор = [1 -1 1 1]^T;
  обновление 1-го бита = [1 -1 1 1]^T;
  обновление 2-го бита = [1 -1 1 1]^T;
  обновление 3-го бита = [1 -1 1 1]^T;
  обновление 4-го бита = [1 -1 1 -1]^T;
- вторая итерация:
  обновление 1-го бита = [1 -1 1 -1]^T;
  обновление 2-го бита = [1 -1 1 -1]^T;
  обновление 3-го бита = [1 -1 1 -1]^T;
  обновление 4-го бита = [1 -1 1 -1]^T.

Отметим, что в данном примере реально было выполнено изменение только четвертого бита во время первой итерации, так как

$$net_4(1) = \sum_{j=1}^{4} w_{1j} y_j(1) = -2 < 0 .$$

Векторы $\vec{x}(1)$, $\vec{x}(2)$, $\vec{x}$ и $\vec{y}$ рассматриваемого примера можно представить в виде черно-белых пиктограмм, показанных на рисунке 8.27. Сеть распознает пиктограмму, соответствующую вектору $\vec{x}$, как искаженную версию пиктограммы вектора $\vec{x}(2)$. Следовательно, на выходе сети восстанавливается образ $\vec{x}(2)$. Такое поведение сети объясняется тем, что векто-

ры $\vec{x}$ и $\vec{x}(2)$ отличаются только одним битом, а векторы $\vec{x}$ и $\vec{x}(1)$ – тремя. Вектор $\vec{x}$ ближе к вектору $\vec{x}(2)$ в смысле расстояния Хэмминга, и, следовательно, на выходе сети восстанавливается вектор $\vec{x}(2)$. Однако более полное представление о поведении сети можно получить из анализа энергетической функции.

Для рассматриваемого примера

$$E(\vec{y}) = -\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij} y_i y_j \,. \qquad (8.87)$$

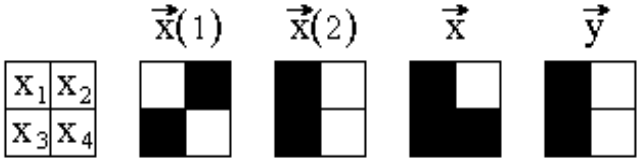


Рисунок 8.27 – Изображение бинарных векторов.

Подставив в (8.87) соответствующие значения элементов матрицы *W*, получим

$$E(\vec{y}) = -\frac{1}{2}(-2y_1y_2 - 2y_2y_1 - 2y_3y_4 - 2y_4y_3) = 2(y_1y_2 + y_3y_4)\,. \qquad (8.88)$$

Так как выходной вектор является четырех элементным, то возможны 16 различных состояний, изображенных на рисунке 8.28. Устойчивыми являются состояния, соответствующие минимумам энергетической функции (8.88). Возможные направления смены состояний показаны с помощью стрелок. Переход из одного состояния в другое возможен только в случае, если векторы состояний различаются одним битом (в силу асинхронного принципа работы сети) и значение энергетической функции нового состояния сети уменьшается.

В рассматриваемой сети Хопфилда имеется четыре устойчивых состояния, характеризуемых минимальным значением энергетической функции, равным -4. Два устойчивых состояния сети соответствуют векторам $\vec{x}(1)$ и $\vec{x}(2)$, а два других состояния представляют векторы:

$$\vec{x}_d(1) = [1 \;\; -1 \;\; -1 \;\; 1] \text{ и } \vec{x}_d(2) = [-1 \;\; 1 \;\; -1 \;\; 1].$$

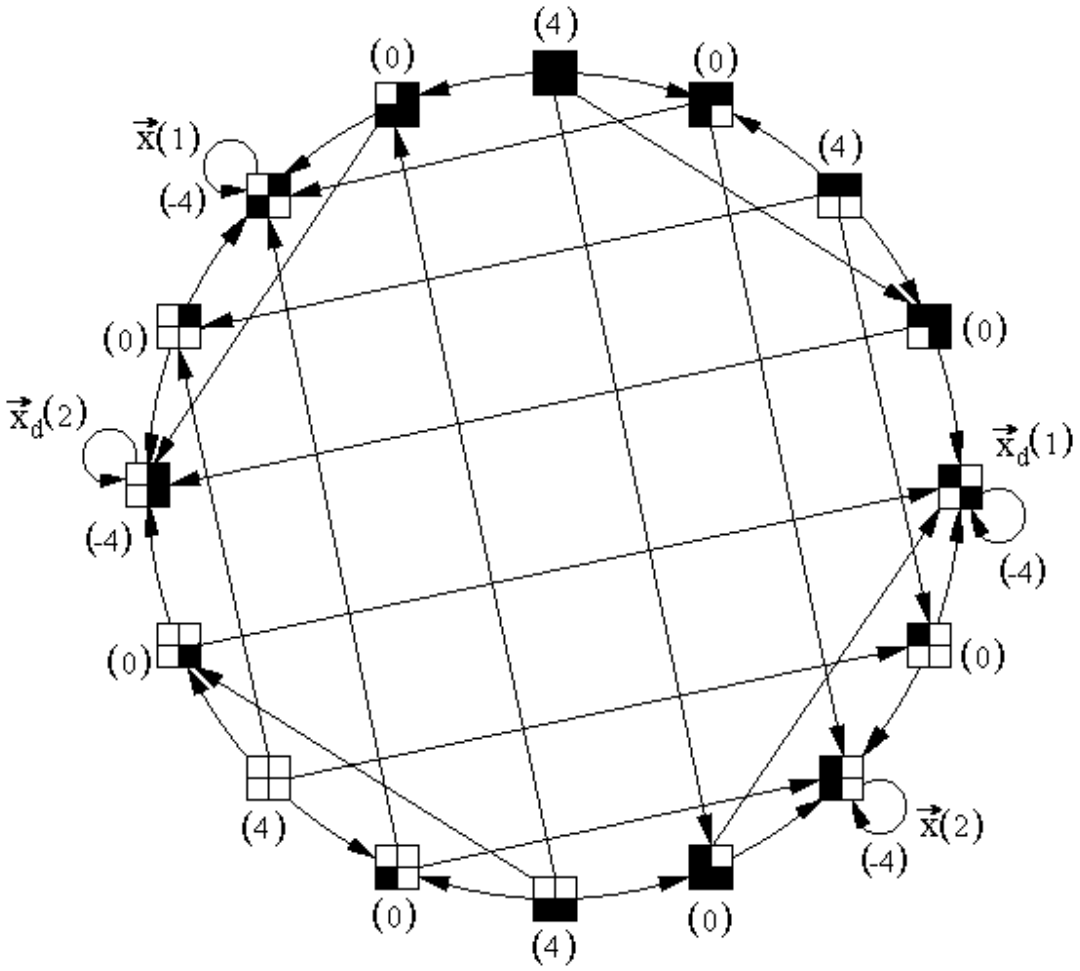


Рисунок 8.28 – Состояния асинхронной сети Хопфилда.

Векторы $\vec{x}_d(1)$ и $\vec{x}_d(2)$ получаются из векторов $\vec{x}(1)$ и $\vec{x}(2)$ простой инверсией значений битов, поэтому их называют дополнительными векторами. Наличие в сети дополнительных устойчивых состояний приводит к зависимости поведения асинхронных ИНС Хопфилда от порядка активизации НЭ сети. Например, при исходном начальном состоянии, заданном вектором $[-1 \;\; -1 \;\; -1 \;\; -1]$, в сети возможны следующие переходы (см. рисунок 8.28):

$$[-1 \;\; -1 \;\; -1 \;\; -1] \to [1 \;\; -1 \;\; -1 \;\; -1] \to [1 \;\; -1 \;\; 1 \;\; -1] = \vec{x}(2);$$
$$[-1 \;\; -1 \;\; -1 \;\; -1] \to [-1 \;\; 1 \;\; -1 \;\; -1] \to [-1 \;\; 1 \;\; -1 \;\; 1] = \vec{x}_d(2);$$
$$[-1 \;\; -1 \;\; -1 \;\; -1] \to [-1 \;\; 1 \;\; -1 \;\; -1] \to [-1 \;\; 1 \;\; 1 \;\; -1] = \vec{x}(1);$$
$$[-1 \;\; -1 \;\; -1 \;\; -1] \to [1 \;\; -1 \;\; -1 \;\; -1] \to [1 \;\; -1 \;\; 1 \;\; -1] = \vec{x}_d(1).$$

Это происходит потому, что расстояние Хэмминга между исходными и результирующими векторами $\vec{x}(1), \vec{x}_d(1), \vec{x}(2), \vec{x}_d(2)$ одинаковое и равно 2. Таким образом, построение автоассоциативной памяти на основе ИНС Хопфилда сопряжено с двумя проблемами:

1) проблемой дополнительных устойчивых состояний, соответствующих инверсным бинарным векторам запоминаемых образов;
2) проблемой неоднозначности результатов восстановления.

Наличие первой проблемы означает, что сеть Хопфилда запоминает не только исходные образы, но и инверсные образы, задаваемые дополнительными векторами. Вторая проблема тесно связана с емкостью сети Хопфилда. *Емкостью* сети называют количество различных образов (векторов-прототипов), которые можно запомнить в сети из N нейронов.

При оценивании емкости ассоциативной памяти, реализованной в виде ИНС Хопфилда, вводится параметр, называемый радиусом аттрактора ρ. Радиус ρ представляет собой расстояние, которое определяет область "притяжения" вокруг аттрактора $\vec{y}$. Ассимптотическая оценка емкости сети Хопфилда равна [76]

$$C = \frac{(1-2\rho)^2 N}{4\ln N},\ 0 \le \rho < \frac{1}{2}.$$

Если $P < C$, то все образы, запоминаемые в сети, будут восстанавливаться с вероятностью близкой к единице. Если допустимы некоторые ошибки восстановления, то

$$C = \frac{(1-2\rho)^2 N}{2\ln N},\quad 0 \le \rho < \frac{1}{2}.$$

Если положить $\rho = 0$, то емкость сети Хопфилда может быть определена из неравенства

$$\frac{N}{4\ln N} < C < \frac{N}{2\ln N}.$$

Например, емкость сети Хопфилда из 100 нейронов будет находиться в пределах $5{,}4 < C < 10{,}8$. Для сравнения емкость сети Хэмминга определяется соотношением $2^{\beta N}$, $\beta \le 1$ [74].

8.5.7. Коннекционистские экспертные системы

Интеграция моделей представления знаний и правил обучения нейронных сетей – важное направление развития СИИ. Объединение ней-

ронных сетей и баз знаний может выполняться различными способами. В простейших случаях нейронная сеть выполняет предварительную или завершающую обработку информации в системах, основанных на знаниях. В более интересных случаях осуществляется встраивание нейронных сетей в базы знаний, и наоборот.

Системы, объединяющие в себе возможности, предоставляемые нейронными сетями и моделями представления знаний, относят к группе *гибридных интеллектуальных систем*.

Рассмотрим два примера. В первом примере рассмотрим принцип интеграции моделей представления знаний и нейронных сетей, а во втором – совместное функционирование нейронной сети и ЭС продукционного типа.

Интеграция ИНС с продукционными правилами. В этом случае множество продукционных правил определяет первоначальную структуру нейронной сети. После параметрического или структурного обучения сети выполняется обратное восстановление множества правил по нейронной сети. Это позволяет улучшить базу правил с помощью алгоритмов обучения нейронных сетей.

Для реализации такого подхода предложено использовать правила в форме дизъюнктов Хорна [38,76]

$$l_1 \,\&\, l_2 \,\&\, \ldots \,\&\, l_n \to r,$$

где $l_1, l_2, \ldots l_n$ – предпосылки правила; r – заключение правила. Каждое правило соответствует фрагменту нейронной сети, изображенной на рисунке 8.29.

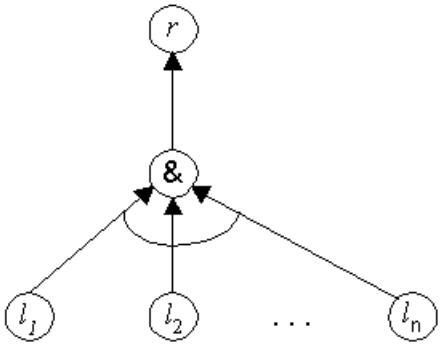


Рисунок 8.29 – Фрагмент ИНС в виде правила

Множество правил образует сеть. При этом узлы первого слоя сети соответствуют предпосылкам правил, которые связаны с конъюнктивными узлами первого скрытого слоя. В свою очередь, узлы представляющие заключения различных правил дизъюнктивно связаны с соответствующими конъюнктивными узлами. В итоге, база продукционных правил отобража-

ется в виде многослойной нейронной сети с чередующимися конъюнктивными и дизъюнктивными слоями (рисунок 8.30). Коэффициенты уверенности правил ассоциируются с весами связей НЭ сети.

На этапе обучения сети осуществляется подстройка весов связей, и выполняются структурные преобразования сети. Результаты обучения сети в дальнейшем переносятся в базу правил, работа с которой поддерживается с помощью операторов модификации коэффициентов уверенности, удаления, обобщения и специализации. Первый оператор непосредственно связан с изменением весов связей ИНС в ходе обучения. При этом обучение может осуществляться различными методами в зависимости от структуры сети. Например, на основе алгоритма обратного распространения ошибки. Остальные операторы связаны с введением новых или удалением существующих соединений в ИНС. Такие изменения обычно происходят в структурно-адаптивных ИНС.

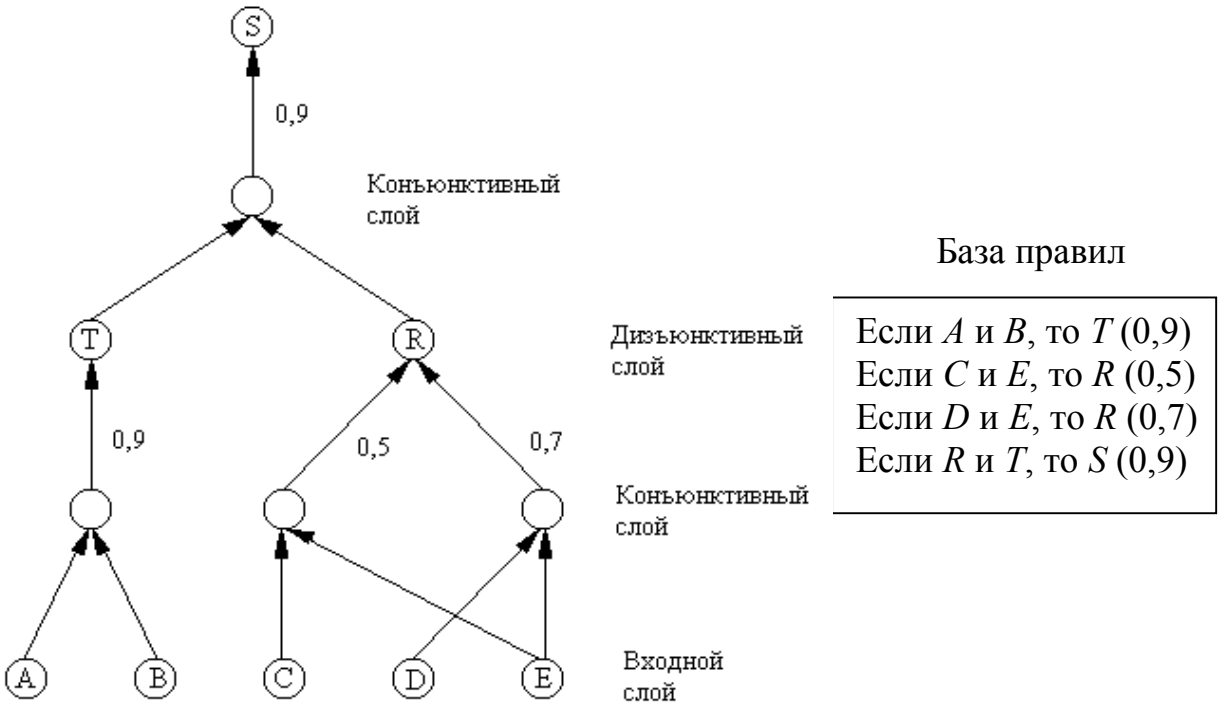


Рисунок 8.30 – Пример ИНС, основанной на правилах

Удаление соединения связано с обнулением весов связей, характеризуемых небольшими значениями. Удаление соединения, идущего от НЭ конъюнктивного слоя к НЭ дизъюнктивного слоя, соответствует удалению правила. Удаление соединения, идущего от НЭ дизъюнктивного слоя или входного слоя к НЭ конъюнктивного слоя, соответствует оператору обобщения, так как исключение лишней конъюнктивной связи делает условие правила более общим. Добавление соединения между НЭ конъюнктивного

слоя и НЭ дизъюнктивного слоя означает создание нового правила, а добавление соединения, идущего от НЭ дизъюнктивного слоя (или входного слоя) к НЭ конъюнктивного слоя, соответствует специализации правила.

После обучения структура нейронной сети представляется в виде множества правил. Для такого преобразования может быть использован КТ-алгоритм [76].

Нейронная продукционная система. Рассмотрим нейронную продукционную систему, осуществляющую навигацию автономного мобильного робота [76]. Одно из продукционных правил, используемых в системе, имеет вид:

если
робот находится близко к пересечению путей,
то
снизить скорость
и
на пересечении повернуть направо.

Предпосылка данного правила не вполне определена. Чтобы объяснить системе смысл понятия “близко к пересечению путей”, можно предпосылку правила представить с помощью нейронной сети и затем осуществлять обучение сети. В ходе обучения следует подавать на вход сети изображения различных пересечений, добиваясь правильной реакции сети на ситуацию, когда робот приближается к пересечению.

Также можно представить в виде нейронной сети и заключение правила (т.е. “на пересечении повернуть направо”). При этом процесс обучения заключается в формировании необходимого управляющего воздействия на рулевой механизм робота. Этого можно добиться аппроксимацией направлений контуров пересечения по изображению, снимаемому видеокамерой.

Структурная схема нейронной продукционной системы изображена на рисунке 8.31. Рабочая память содержит список фактов и целей, отражающих текущее состояние системы. Факты и цели представляются в памяти в виде цепочек символов (см. рис. §2.3). Например, в случае движения по коридору рабочая память может содержать следующие факты:

местонахождение(коридор_C_2),
ориентация(север),
текущее_положение (близко, пересечение_J_2),
текущая_цель(двигаться_вдоль_коридора),
следующая_цель(повернуть_направо, пересечение_J_2).

Препроцессор выполняет предварительную обработку поступающих изображений. По запросам продукционной системы он выделяет необходи-

мые признаки изображений, которые могут поступать либо на вход нейропроцессора, либо непосредственно в рабочую память.

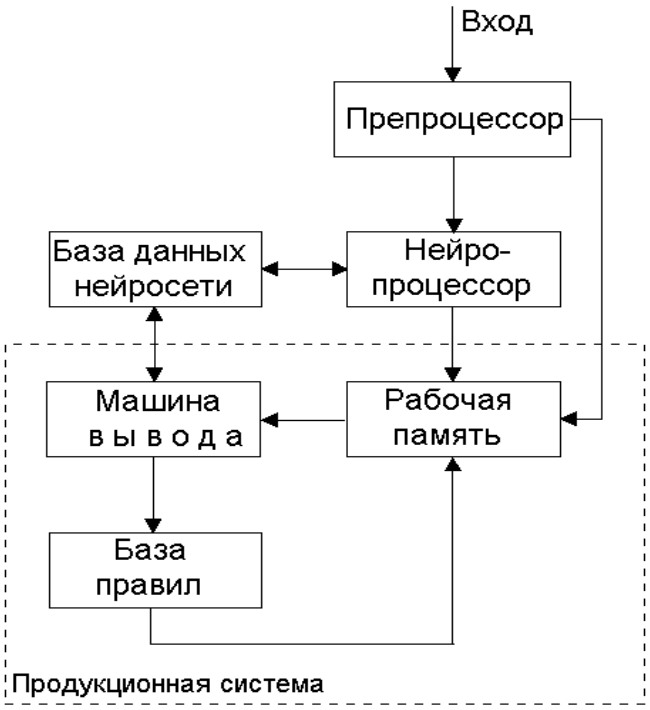


Рисунок 8.31 – Нейронная продукционная система

Нейронные сети, реализуемые нейропроцессором, обучаются формированию последовательности символов (термов), которые хранятся в базе данных нейронной сети, либо в рабочей памяти. Каждая нейронная сеть представляется именем, набором параметров, определяющих структуру сети, матрицей весов связей и соответствующим алгоритмом функционирования. Термы, формируемые нейронными сетями, загружаются в рабочую память.

База правил продукционной системы состоит из гибридных правил, включающих как описания, формируемые нейронными сетями, так и символьные выражения традиционных продукционных систем. Например,

Правило 1: если
следующая цель(повернуть_налево, пересечение)
и
ориентация(пересечение)
и
текущее_положение(*близко, пересечение*),
то
двигаться_вдоль_коридора(*медленно*).

В этом правиле строки, выделенные курсивом, представляются с помощью нейронной сети.

Рассмотренная нейронная продукционная система использовалась для управления перемещением мобильного робота, снабженного специальной системой зрения. Применение нейронной продукционной системы позволило роботу анализировать визуальную информацию в процессе движения и выполнять сложные заключения относительно навигационной обстановки.

8.6. Синтаксический метод распознавания

Предыдущие параграфы были посвящены распознаванию образов, основанному на разделении объектов в пространстве признаков. Синтаксический подход, рассматриваемый в настоящем параграфе, базируется на использовании структурных отношений, характеризующих распознаваемый образ. С этой целью в распознаваемом объекте выделяют простейшие структурные элементы и выполняют описание сложного объекта в виде иерархической структуры, состоящей из простых элементов. Типичным примером задач, где применяется такой подход, является распознавание изображений и, в частности, анализ сцен (см. главу 10). В качестве примера рассмотрим сцену, изображенную на рисунке 8.32,а [22]. Иерархическое описание данной сцены представлено в виде дерева на рисунке 8.32,б. Здесь сцена А состоит из подообразов, часть которых представляется элементарными подобразами – гранями, треугольником. Сцена определяется как некоторая структура, состоящая из подобразов, аналогично тому, как предложения естественного языка строятся из слов при помощи грамматических правил, а слова образуются из букв. Благодаря этой аналогии рассматриваемый подход называют синтаксическим (лингвистическим, структурным) подходом к распознаванию образов.

Первым шагом в построении структурного описания объекта распознавания является выделение простейших элементов – примитивов. Примитивы образа должны легко выделяться. На основе выделенных примитивов строится структурное описание объекта распознавания. Процесс распознавания завершается выполнением синтаксического анализа, в ходе которого устанавливается, является ли это описание объекта синтаксически правильным относительно заданной грамматики. Объект зачисляется в тот класс, в котором его описание оказывается синтаксически правильным.

Структурная схема системы синтаксического распознавания образов изображена на рисунке 8.33 .

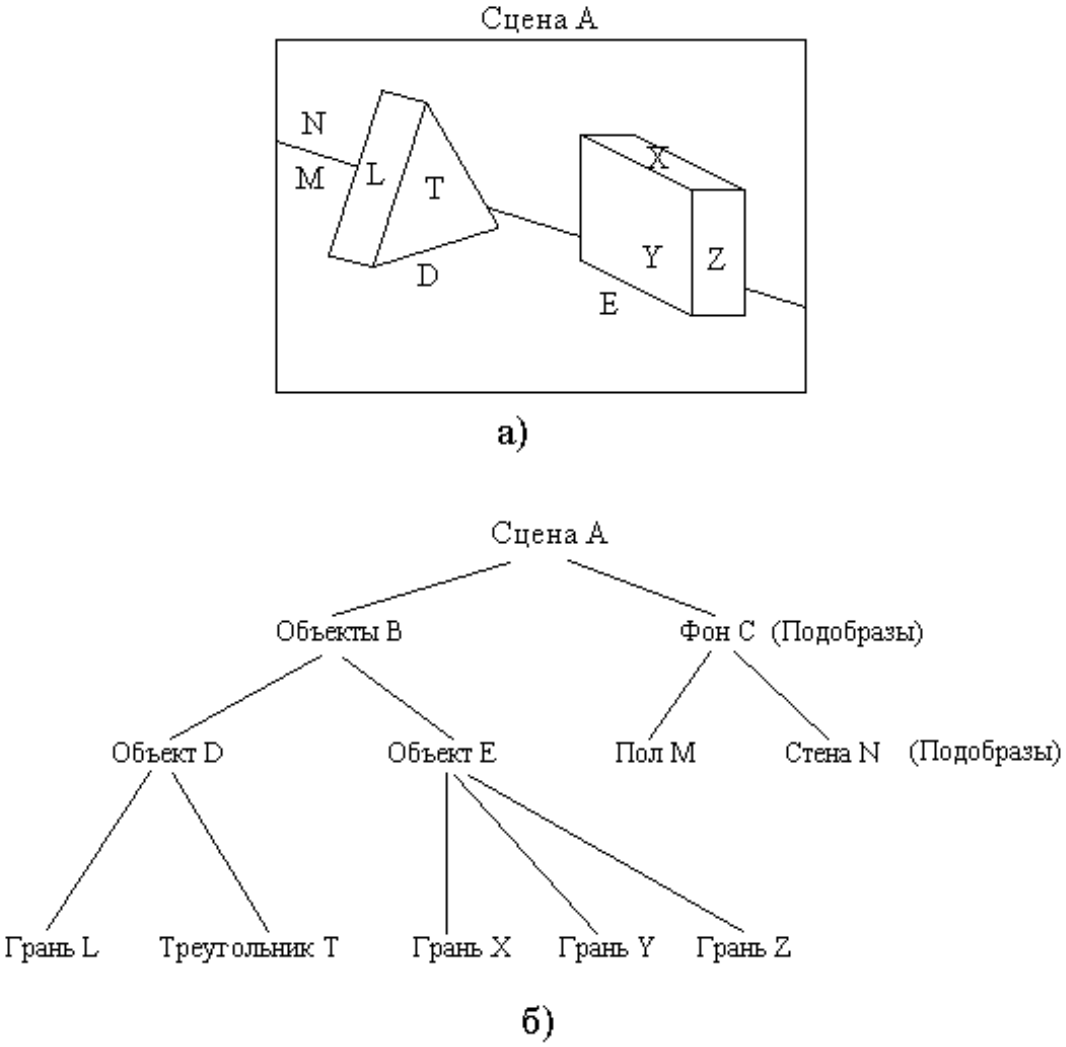


Рисунок 8.32 – Сцена А и ее иерархическое описание

На этапе предварительной обработки данные об объекте распознавания с целью улучшения качества подвергаются фильтрации, аппроксимации, восстановлению и т.п. Процесс построения описания объекта предполагает выполнение двух процедур: сегментации объекта и выделения примитивов объекта. Объект сегментируется на подобъекты и примитивы в соответствии с заранее определенными синтаксическими операциями. В результате каждый объект представляется с помощью некоторого набора примитивов и фиксированных синтаксических операций. Если используется операция конкатенации, то объект представляется строкой, состоящей из последовательности примитивов.

На этапе синтаксического анализа могут выполняться либо простейшее сопоставление с эталоном (образцом), либо полный грамматический разбор. В случае сопоставления с эталоном строка примитивов, представ-

ляющая исходный объект, сравнивается со строками, представляющими объект-прототип. В соответствии с выбранным критерием подобия строк объект зачисляется в тот класс, к которому относится объект-прототип. Информация об иерархической структуре объекта в этом случае игнорируется. Полный грамматический разбор строки, описывающий распознаваемый объект, позволяет выявить особенности структуры объекта и учесть их при принятии решения. Возможны промежуточные варианты реализации синтаксического анализа, определяемые спецификой решаемых задач.

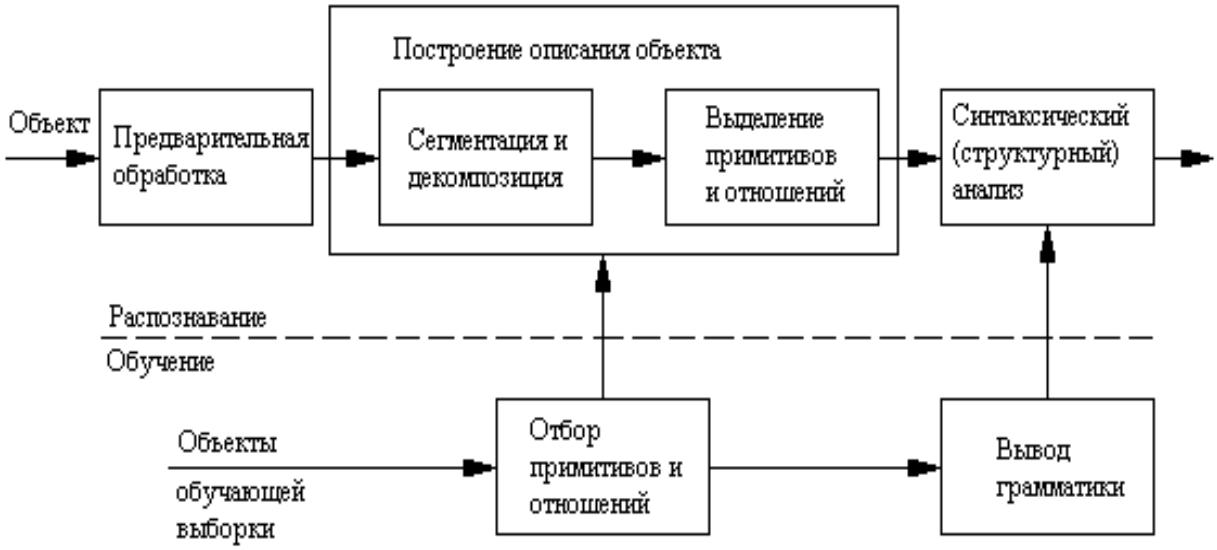


Рисунок 8.33 – Система синтаксического распознавания образов

Вывод грамматики, необходимой для выполнения синтаксического анализа, осуществляется по заданной обучающей выборке объектов. При этом отбор примитивов и структурных отношений во многих случаях выполняется проектировщиком системы распознавания. Вывод грамматики, по сути, представляет обучение синтаксической системы распознавания.

Рассмотрим основные понятия теории формальных грамматик. Грамматика G – это четверка [22]

$$G = (V_N, V_T, P, S),$$

где V_N и V_T – конечные множества соответственно начальных и терминальных символов, $V_N \cap V_T = 0$ и $V_N \cup V_T = V$; P – конечное множество продукционных правил вида $\alpha \to \beta$, где α и β представляют строки символов из V, причем α содержит, по крайней мере, один символ из V_N; $S \in V_N$ – начальный символ. Множество V_T называют терминальным (ос-

новным) словарем, а множество V_N – нетерминальным (вспомогательным) словарем. Терминальный словарь – это набор исходных элементов, из которых строятся цепочки символов (строки), порождаемые грамматикой. Нетерминальный словарь – это набор символов, которыми обозначаются классы цепочек из исходных элементов. Начальный символ S представляет собой элемент множества V_N и обозначает класс всех тех объектов, для описания которых и предназначена грамматика.

Для произвольного словаря V обозначение V^* будет использоваться для идентификации всех строк, составленных из V, включая пустую строку λ. Обозначение V^+ будет соответствовать множеству строк из V без пустой строки, т.е. $V^+ = V^* - \{\lambda\}$. Например, пусть $V = \{a, b\}$, тогда $V^* = \{\lambda, a, b, aa, ab, ba, ...\}$ и $V^+ = \{a, b, aa, ab, ba, ...\}$.

Множество продукционных правил P состоит из выражений вида $\alpha \to \beta$, где α – строка в словаре V^+, а β – строка в словаре V^*. Запись $\alpha \to \beta$ означает, что строка α замещается строкой β. Символ $\underset{G}{\Rightarrow}$ будет использован для обозначения вывода $w_1 \alpha w_2 \underset{G}{\Rightarrow} w_1 \beta w_2$ в грамматике G. При этом строка α замещается строкой β, а w_1 и w_2 остаются неизменными.

Язык, порождаемый грамматикой G и обозначаемый $L(G)$, представляет множество строк (цепочек) [22, 43]:

$$L(G) = \{x \mid x \in V_T^* \quad и \quad S \overset{*}{\underset{G}{\Rightarrow}} x\}.$$

Каждая из строк языка состоит только из терминальных символов. При этом любая строка может быть выведена из S путем применения правил из P, что обозначено как $S \overset{*}{\underset{G}{\Rightarrow}} x$.

Грамматики различают по типу правил подстановки, допустимых в каждой из них. *Контекстно-зависимая грамматика* характеризуется правилами подстановки вида $\alpha_1 A \alpha_2 \to \alpha_1 \beta \alpha_2$, где α_1 и α_2 – элементы словаря V^*, $\beta \in V^+$ и $A \in V_N$. Такая грамматика позволяет выполнять замещение нетерминального символа A строкой β, если A появляется в контексте строк α_1 и α_2. *Контекстно-свободная грамматика* характеризуется правилами подстановки вида $A \to \beta$. Здесь символ $A \in V_N$ может замещаться строкой β независимо от контекста, в котором появляется A. Грамматика называется *регулярной*, если используются правила подстановки вида $A \to aB$ или $A \to a$, где $A, B \in V_N$, $a \in V_T$. При этом альтернативными правилами подстановки являются $A \to Ba$ и $A \to a$. Однако выбор одной группы правил исключает применение другой группы [22].

В качестве примера рассмотрим контекстно-зависимую грамматику $G_0 = (V_N, V_T, P, S)$, где $V_N = \{S, A, B\}$, $V_T = \{a, b, c\}$, и множество P задано правилами продукции:

1) $S \to aSBA$;
2) $S \to abA$;
3) $AB \to BA$;
4) $bB \to bb$;
5) $bA \to bc$;
6) $cA \to cc$;
7) $aB \to ab$.

Язык $L(G)$, порождаемый грамматикой G_0, является множеством всех строк, состоящих из одинакового числа символов *a*, *b* и *c*:

$$L(G_0) = \{a^n b^n c^n \mid n \geq 1\}.$$

Например, $S \overset{(2)}{\Rightarrow} abA \overset{(5)}{\Rightarrow} abc$ или

$$S \overset{(1)}{\Rightarrow} aSBA \overset{(2)}{\Rightarrow} aabABA \overset{(3)}{\Rightarrow} aabBAA \overset{(4)}{\Rightarrow} aabbAA \overset{(5)}{\Rightarrow} aabbcA \overset{(6)}{\Rightarrow} aabbcc.$$

Здесь числа в скобках указывают номера использованных продукций.

Если символы *a*, *b* и *c* будут соответствовать графическим примитивам, изображенным на рисунке 8.34, то грамматика $L(G_0)$ будет порождать строки, соответствующие равносторонним треугольникам.

Отметим, что контекстно-зависимые грамматики представляют значительные трудности для автоматического анализа. Поэтому на практике чаще применяют контекстно-свободные грамматики.

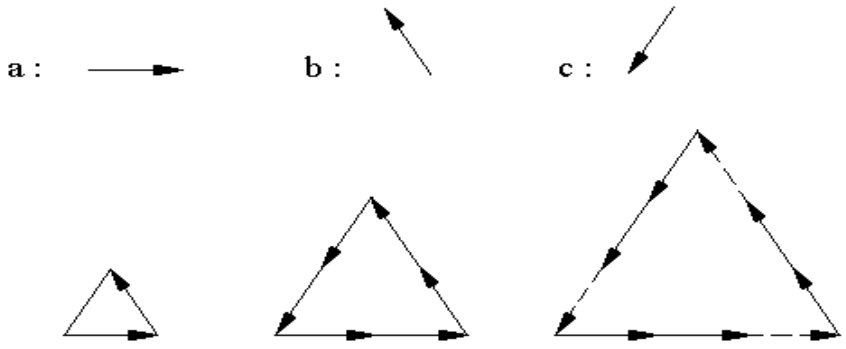


Рисунок 8.33 – Множество фигур, порожденных грамматикой $L(G_0)$

Значение формальных грамматик для распознавания состоит в том, что они позволяют построить алгоритм для ответа на вопрос о синтаксической правильности или неправильности представления объекта с помощью примитивов. Этот алгоритм выполняет процедуру синтаксического анализа (грамматического разбора). Существует два основных вида синтаксического анализа: разбор сверху вниз и разбор снизу вверх. В процедуре разбора сверху вниз правила-продукции применяются к начальному символу S, чтобы получить заданную строку, представляющую распознаваемый объект. Данная процедура реализует процесс подстановок, управляемый целью. При разборе снизу вверх продукционные правила применяются в обратном направлении с целью приведения строки, представляющей распознаваемый объект, к начальному символу. Прямые реализации указанных схем грамматического разбора неэффективны, так как требуют выполнения полного перебора. Повысить эффективность синтаксического разбора можно при учете синтаксиса грамматики. Соответствующие грамматические анализаторы называют синтаксически ориентированными [43].

Важной подзадачей, возникающей в ходе применения синтаксического метода распознавания, является построение (конструирование, вывод) грамматики по обучающей выборке. Конструирование грамматики является лингвистическим эквивалентом алгоритмов обучения, рассмотренных ранее. К сожалению, не существует универсальных автоматических методов построения требуемой грамматики по примерам. Известные подходы имеют ограниченные области приложений [43].

Вопросы для самопроверки

1. Определите понятие “образ”.
2. Что называют признаками образа?
3. Как формулируется задача распознавания образов
4. Сформулируйте гипотезу компактности.
5. Объясните принцип классификации объектов с помощью функции расстояния.
6. Определите расстояния Махалонобиса, Хемминга, Левенштейна.
7. Объясните на примере алгоритм вычисления расстояния Левеншнейна.
8. Сформулируйте алгоритм К-средних.
9. Объясните байесовский метод распознавания.
10. Запишите правило максимального правдоподобия.
11. Запишите правило обновления параметров решающей функции в ходе рекуррентного алгоритма обучения распознаванию образов.
12. Нарисуйте схему формального нейронного элемента.
13. Что называют сетевой (базовой) функцией нейроподобного элемента?
14. Какие виды функций преобразования применяются при построении моделей нейроподобных элементов?
15. Приведите примеры возможных структур ИНС.

16. Объясните термины “обучение с учителем” и “обучение без учителя”.
17. Сформулируйте правило обучения простого персептрона и его свойства.
18. Что представляет собой адаптивный линейный элемент? Запишите правило обучения Уидроу-Хоффа.
19. Запишите дельта-правило обучения. Нарисуйте структуру сети, в которой оно применяется.
20. Нарисуйте многослойный персептрон. Запишите обобщенное дельта правило.
21. Объясните алгоритм обратного распространения ошибки.
22. Сформулируйте практические рекомендации по выбору параметров алгоритма обратного распространения ошибки.
23. Сформулируйте правило обучения Хебба.
24. Нарисуйте структурную схему однослойной состязательной ИНС и объясните правило обучения Кохонена.
25. Сформулируйте правило обучения Кохонена для самоорганизующихся карт признаков.
26. Приведите пример применения самоорганизующихся карт признаков в задаче распознавания речи.
27. Объясните принцип функционирования ИНС, использующей теорию адаптивного резонанса.
28. Объясните назначение и принцип функционирования неокогнитрона.
29. Назовите виды ассоциативной памяти.
30. Объясните механизм линейной ассоциативной памяти.
31. Как функционирует ИНС Хэмминга?
32. Что представляет собой ИНС Хопфилда?
33. Как определяются веса связей в сети Хопфилда?
34. Что называют энергетической функцией, и какие у неё свойства?
35. Что понимают под емкостью сети Хопфилда?
36. Поясните принцип возможной интеграции ИНС и продукционных правил.
37. Объясните суть синтаксического метода распознавания.
38. Приведите пример порождающей грамматики.

ГЛАВА 9
ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА

Одним из основных направлений ИИ является разработка методов, обеспечивающих реализацию процесса общения с ЭВМ на естественном языке (ЕЯ). Данное направление получило название “обработка естественного языка” (Natural Language Processing).

Успех общения с ЭВМ на естественном языке, которое может протекать либо в текстовой, либо речевой форме, в значительной степени определяется наличием необходимых знаний. Общение не сводится к простой передаче множества слов. Оно требует установления контекста диалога, понимания высказываний, выявления целей пользователя, генерации высказываний и др. Все это предполагает наличие обширных знаний, например, о языке общения, о процессе диалога, предмете диалога и пр.

Настоящая глава посвящена краткому введению в проблему обработки естественного языка. Глава начинается с рассмотрения методов понимания высказываний, представленных в текстовой форме. С этой целью анализируется метод расширенных сетей переходов. Рассматриваются принципы построения ЕЯ-интерфейса для доступа к базам данных. Значительная часть главы посвящена принципам распознавания и синтеза речевых сообщений.

9.1. Компоненты ЕЯ-системы

Общение с ЭВМ на естественном языке – это сложная задача, решение которой предполагает реализацию следующих основных функций [20]: ведение диалога, понимание высказываний, обработка высказываний, генерация выходных высказываний.

Под *диалогом* понимают процесс обмена взаимосвязанными высказываниями между пользователем и ЭВМ, направленный на достижение

целей пользователя. При этом высказывания участников диалога образуют связный текст – *дискурс.* Процесс диалога может быть разбит на отдельные шаги. Шаг диалога представляет пару высказываний, состоящую из высказывания активного участника диалога и реакции пассивного. В ходе диалога возможен *перехват инициативы*, когда пассивный участник выдвигает новую подцель. Функция *ведения диалога* заключается в определении его структуры и управлении активностью участников диалога. Тематическая структура диалога обычно представляется в виде сценария. При этом сценарий может встраиваться в систему при её создании, либо генерироваться системой.

Функция *понимания высказываний* предполагает выяснение смысла ЕЯ-высказываний и представление его в формализованном виде на внутреннем языке системы. При реализации функции понимания высказываний предусматривают два этапа: *анализ* и *интерпретацию* [20]. На этапе анализа выделяют сущности (понятия) предметной области, отражаемые во входном высказывании, их свойства и отношения. Интерпретация состоит в сопоставлении формализованного высказывания и знаний, которыми обладает система. Знания системы классифицируются по следующим областям: предметная область, область языка, область диалога (дискурса), область пользователя, область знаний системы о самой себе. Различные области определяют возможные интерпретации входных высказываний. Конечная задача интерпретации состоит в установлении цели пользователя.

Обработка высказываний сводится к планированию и решению задачи в соответствии с намерениями пользователя на данном шаге диалога.

Генерация высказываний состоит в формировании смысла высказывания на внутреннем языке системы и преобразовании его в ЕЯ-высказывание. На этапе формирования смысла выходных высказываний решаются следующие задачи [34]: выявление сущностей предметной области, которые интересуют пользователя; определение уровня общности синтезируемой информации; выделение тех аспектов сущностей, которые соответствуют целям, стоящим перед пользователем; разбиение сообщаемой информации на фрагменты, соответствующие будущим предложениям; определение лексем и построение семантического представления фраз. На этапе синтеза ЕЯ-высказывания осуществляется: построение синтаксической структуры предложений, приписывание морфологической информации вершинам синтаксической структуры, определение порядка следования слов, морфологический синтез словоформ.

Центральной функцией, выполняемой ЕЯ-системами, является функция понимания высказываний. При реализации этой функции обычно выделяют анализ слов, предложений и текста (дискурса). *Анализ слов* за-

ключается в поиске основ слов с отнесением их к определенным грамматическим категориям (морфологический анализ), обнаружении и исправлении орфографических ошибок. *Анализ предложений* сводится к синтаксическому и семантическому анализу, который выполняется с помощью отдельного функционального блока, называемого анализатором.

Цель синтаксического анализа заключается в установлении синтаксической правильности предложения и построении его лингвистической структуры на основе выделения определенных лингвистических отношений между словами. Конкретная структура предложения определяется выбранной системой лингвистических отношений и часто представляется в виде дерева грамматического разбора. В ходе синтаксического анализа используются знания морфологии и синтаксиса входного языка, а также некоторые сведения о семантике языковых конструкций.

Цель семантической интерпретации – формирование описаний проблемных ситуаций, заданных входными предложениями, на внутреннем языке системы, например, с помощью концептуальных графов, фреймов, логических представлений. Получение таких описаний подразумевает расширение сведений, которые неявно представлены во входном ЕЯ – предложении.

На рисунке 9.1 изображен пример анализа и интерпретации предложения “Тарзан поцеловал Джейн”[77]. Здесь на этапе синтаксического анализа используется контекстно-свободная грамматика, заданная с помощью правил продукций, рассмотренных в §2.1.2. На этапе семантической интерпретации анализатор, используя знания о глаголе “целовать”, добавил в концептуальный граф понятие “губы”, которое находится в отношении “инструмент” с указанным глаголом. На этапе семантической интерпретации также могут проверяться некоторые ограничения. Например, определение глагола “целовать” может содержать требование, чтобы “объект” был представлен некоторой персоной, если “агент” – персона. На третьем этапе (*этап анализа дискурса*), к структуре исходного ЕЯ предложения добавляются по определенным правилам структуры, полученные на предыдущих шагах, определяющие контекст рассматриваемого высказывания. В результате получается расширенное описание ситуации, заданной исходным ЕЯ-текстом. На рисунке 9.1 в концептуальный граф исходного ЕЯ-предложения вносятся факты: “Тарзану нравится Джейн” и “Джейн и Тарзан находятся в джунглях”. Сформированное описание представляет смысл ЕЯ-текста и поступает на вход СИИ для дальнейшей обработки. Например, в интеллектуальных вопросно-ответных системах полученное описание может быть использовано при ответе на вопросы.

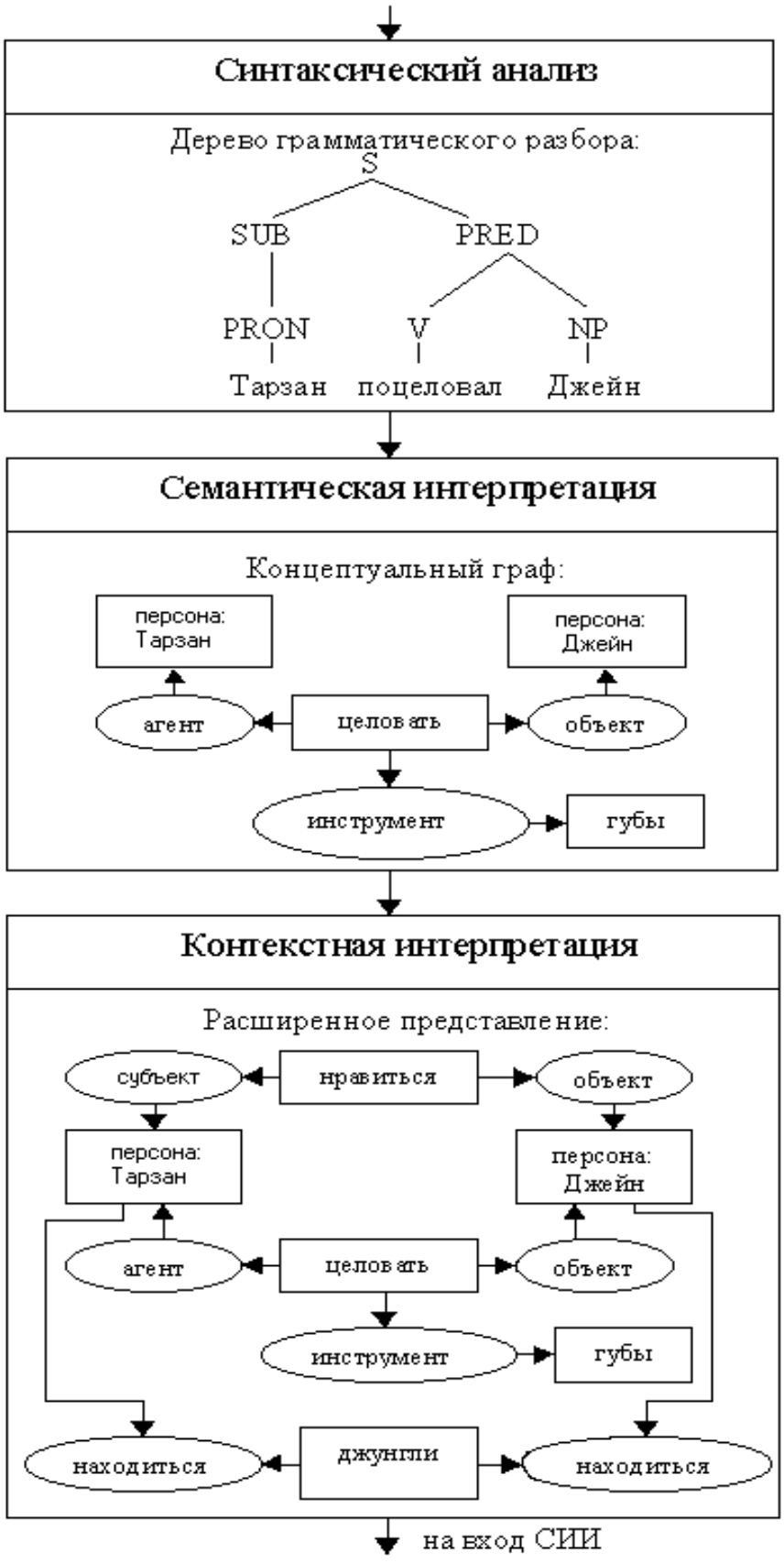


Рисунок 9.1 – Анализ и интерпретация ЕЯ-предложений

Обобщая сказанное, представим этапы анализа в подсистеме понимания в виде структурной схемы, изображенной на рисунке 9.2.

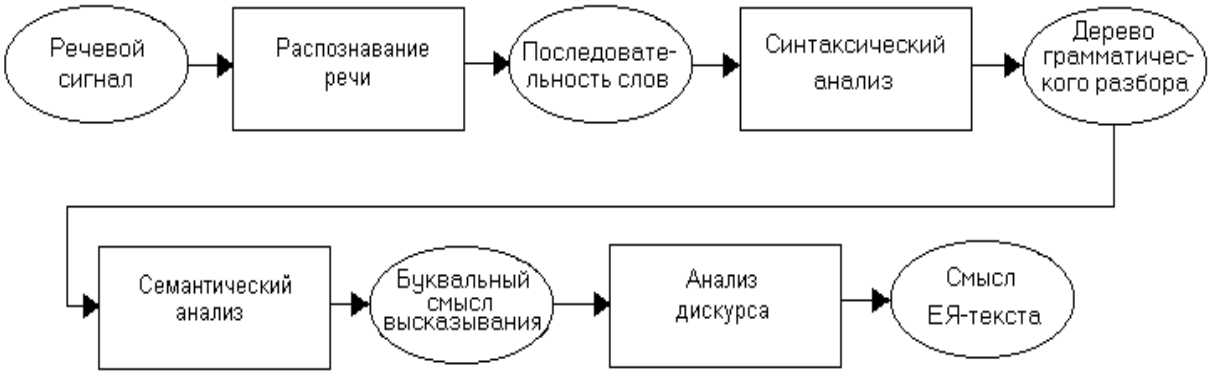


Рисунок 9.2 – Этапы анализа ЕЯ-высказываний

Здесь, дополнительно к рассмотренным выше этапам преобразований входных высказываний, добавляется этап распознавания речи, если входное ЕЯ-высказывание представлено в речевой форме.

Указанные этапы анализа и интерпретации ЕЯ-высказываний характерны для многих систем естественно-языкового общения, однако, детали их реализации могут существенно различаться. Часто синтаксический и семантический анализ выполняется одним функциональным блоком ЕЯ-системы, который называют *анализатором предложений*. Рассмотрим основные подходы, используемые при реализации подсистемы понимания высказываний.

9.2. Понимание ЕЯ-высказываний

Понимание ЕЯ-высказываний сложная задача, в результате решения которой должно формироваться формализованное представление входного предложения на внутреннем языке системы. Если охарактеризовать причину сложности задачи обработки естественного языка, то следует указать на неопределенность, которая проявляется в виде множественности значений слов (полисемия), неоднозначности смысла фраз, неполноты высказываний и др. Неполнота высказываний часто допускается в диалогах между людьми. При этом возможны лексические и грамматические ошибки, пропуски одного или нескольких слов с целью сжатия высказывания (*эллипсис*), повторы. Важным является ответ на вопрос о том, насколько методы понимания высказываний приспособлены к обработке указанных видов неопределенностей. Сравним по этому показателю основные методы понимания ЕЯ-высказываний.

Выделяют два основных подхода к построению подсистем понимания высказываний [61]: рациональный (детерминистский) и эмпирический (стохастический).

В рамках первого подхода основное внимание уделяется разработке грамматических правил и баз знаний, отражающих специфические знания ограниченной предметной области.

Второй подход основан на привлечении к решению задачи методов статистики и машинного обучения. В этом случае подсистема понимания высказываний обучается на больших количествах реальных языковых данных. Различие между рациональным и эмпирическим подходом условно изображено на рисунке 9.3. Рассмотрим подробнее указанные подходы.

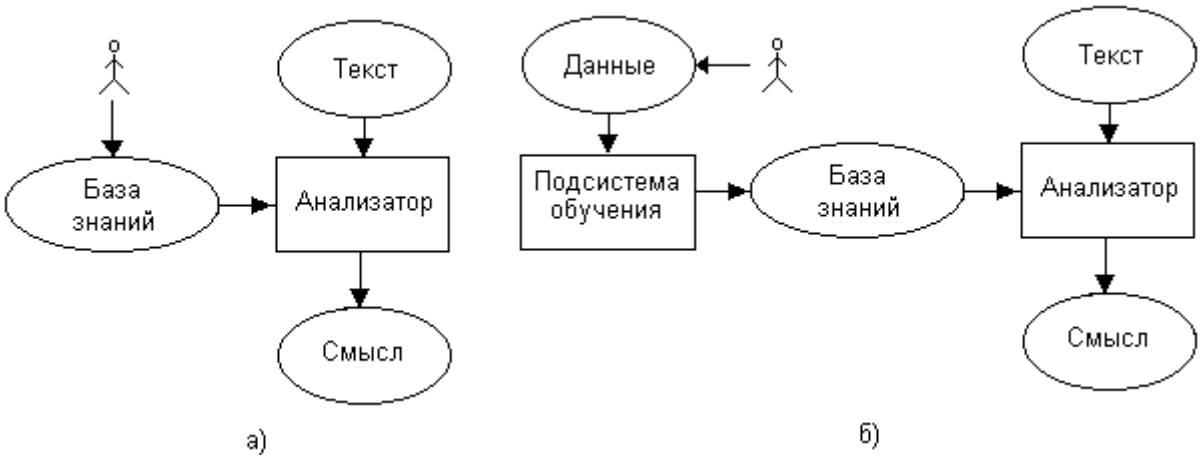


Рисунок 9.3 – Различие между рациональным а) и эмпирическим б) подходами

9.2.1. Рациональный подход

К рациональному подходу относят большую группу методов анализа и интерпретации ЕЯ-предложений [77]: метод сетей переходов, метод расширенных сетей переходов, метод сопоставления с образом, метод трансформационных грамматик, метод семантических грамматик, метод падежной грамматики и др.

В качестве примера рационального подхода рассмотрим методы сетей переходов, основанные на выполнении грамматического разбора с помощью продукционных грамматик. Алгоритмы грамматического разбора в таких грамматиках подразделяются на две группы: разбор сверху вниз и разбор снизу вверх. При разборе сверху вниз, который начинается с символа, представляющего предложение, предпринимается попытка построить дерево грамматического разбора, листья которого сопоставимы со словами анализируемого предложения. В ходе разбора снизу вверх выполняется

поиск последовательности правил-продукций, сводящих множество слов предложения к одному корневому символу дерева грамматического разбора, соответствующего анализируемому предложению. Основной проблемой, которая возникает на каждом шаге применения указанных алгоритмов разбора, является выбор правила. Если выбор будет ошибочным, то анализатор может не распознать правильное предложение. Чтобы исключить такие случаи применяется алгоритм возврата к альтернативным правилам-продукциям. На практике это может приводить к экспоненциальному росту объема вычислений.

Существует большое число алгоритмов построения дерева грамматического разбора. Широкое распространение при анализе ЕЯ-предложений получили алгоритмы, основанные на методе расширенных сетей переходов (ATN-Augmented transition network). Рассмотрим данный метод подробнее.

Анализаторы, осуществляющие разбор предложений с помощью сети переходов, представляют грамматику в виде множества направленных подграфов (сетей переходов) с конечным числом состояний. Каждая сеть соответствует нетерминальному символу грамматики. Метки дуг таких сетей могут быть представлены как терминальными, так и нетерминальными символами. Путь из начального состояния сети перехода в конечное состояние соответствует продукционному правилу, ассоциированному с сетью перехода. При этом метки дуг, которые встретились на пути из начального в конечное состояние, образуют правую часть правила-продукции.

Представим контекстно-свободную грамматику, рассмотренную в §2.1.2, в виде множества сетей переходов, где S_0 и Sg – обозначают соответственно начальное и конечное состояния сети перехода (рисунок 9.4). Если нетерминальный символ представляется несколькими правилами, то в соответствующей сети переходов будет несколько параллельных путей из начального состояния в конечное состояние. Например, правила-продукции SUB→PRON и SUB→NP порождают два альтернативных пути в сети переходов SUB, изображенной на рисунке 9.4

При разборе сверху вниз анализатор пытается найти путь в сети переходов S, представляющей предложение. Для этого анализируются метки дуг сети переходов. Если метка соответствует терминальному символу, то анализатор сопоставляет его с очередным словом анализируемого предложения. Если дуга отмечена меткой нетерминального символа, то анализатор рекурсивно ищет путь в сети переходов данного нетерминального символа. В случае неудачи выполняется возврат к другим возможным вариантам продолжения поиска пути в сети переходов S. Если же в сети переходов S найден путь, ведущий из начального состояния в конечное, то рассматриваемое входное предложение является синтаксически правильным

относительно заданной грамматики. На рисунке 9.5 изображен процесс успешного грамматического разбора предложения “He saw a dog”. Фиксируя имена вызываемых сетей переходов и имена успешно сопоставленных терминальных символов, можно легко построить дерево грамматического разбора.

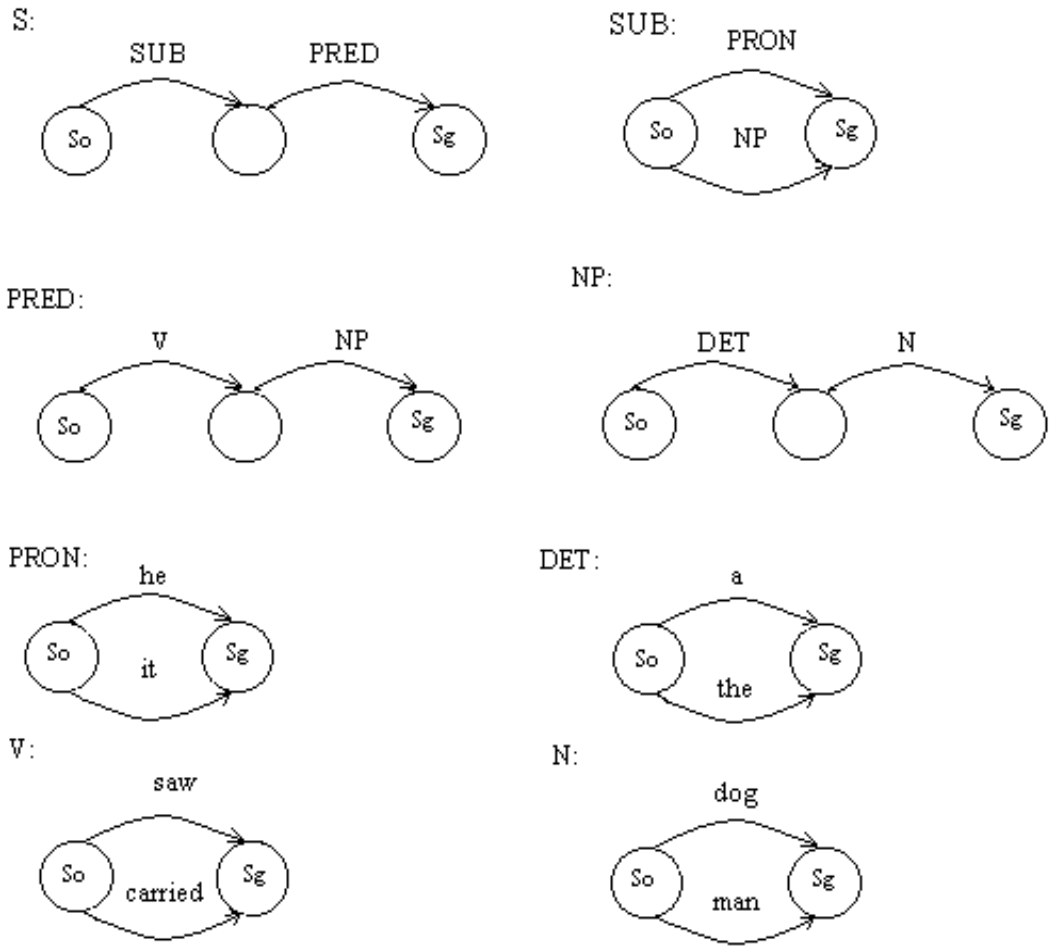


Рисунок 9.4 – Представление грамматики в виде сети переходов

Сети переходов обеспечивают построение деревьев разбора для контекстно-свободных грамматик. Такие грамматики широко применяются при определении языков программирования, но они плохо подходят для представления синтаксических правил естественных языков, относящихся к группе контекстно-зависимых языков.

Для того чтобы адаптировать контекстно-свободные грамматики к анализу естественных языков, с правилами продукциями связывают присоединенные процедуры, выполняющие необходимый контекстный анализ. Соответствующие сети переходов, в которых с дугами связывают как метки, так и присоединенные процедуры, называют *расширенными сетями переходов* (РСП). Указанные присоединенные процедуры выполняются,

когда активизируется соответствующая дуга сети переходов.

В РСП-анализаторах терминальным и нетерминальным символам грамматики приписываются определенные признаки, например: часть речи, корень, число, время, лицо и т.п. Присоединенные процедуры обеспечивают присвоение значений признакам символов и проверку определенных условий. Если некоторое условие не выполняется, например, не согласуется число (единственное, множественное), то соответствующий переход в РСП-сети терпит неудачу. С помощью присоединенных процедур также обеспечивается построение дерева грамматического разбора, которое в дальнейшем используется при формировании внутреннего описания ЕЯ-высказывания, отражающего его семантику.

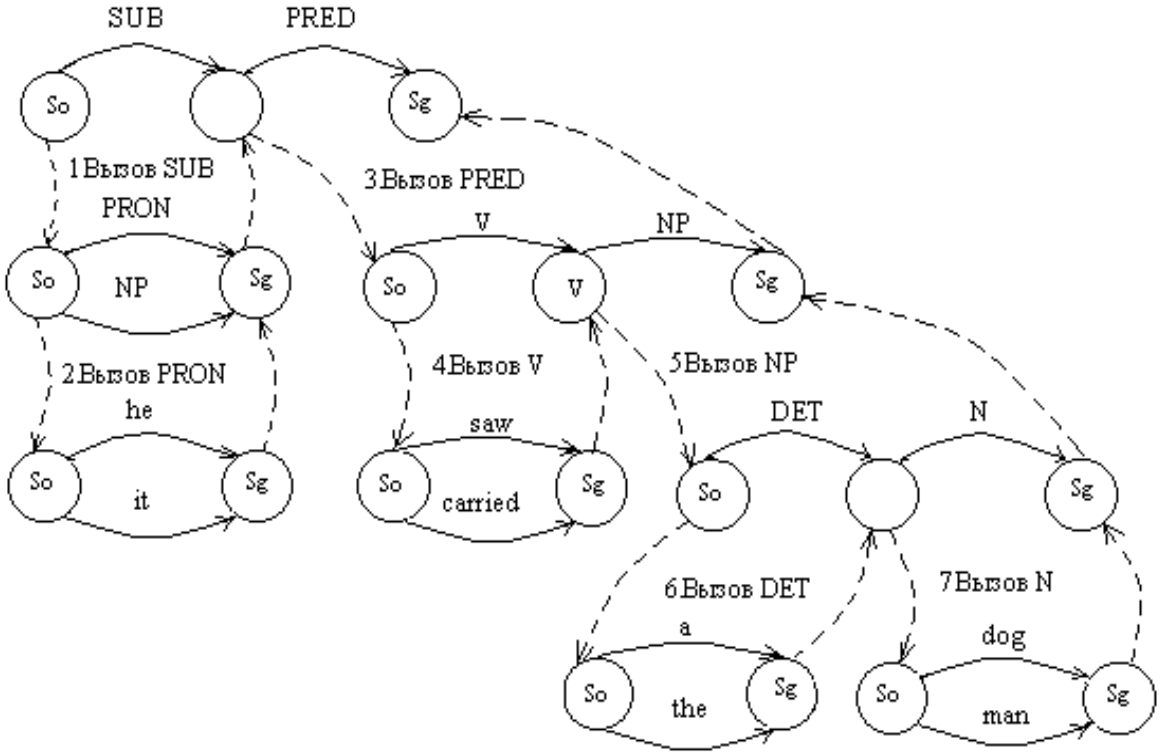


Рисунок 9.5 – Разбор предложения “He saw a dog”

Расширим простейшую грамматику английского языка, изображенную на рисунке 9.4, и представим её в виде РСП-сетей, следуя подходу, изложенному в [77]. Терминальные и нетерминальные символы грамматики будем представлять фреймоподобными структурами. Структуры, соответствующие символам терминального словаря, изображены на рисунке 9.6. В слотах данных структур фиксируются следующие признаки слов: часть речи (POS – part of speech), корень (ROOT), число (NUMBER).

Структуры для хранения нетерминальных символов изображены на рисунке 9.7. Структура предложения (S) содержит два слота: группу подлежащего (SUB - subject) и группу сказуемого (PRED - predicate).



Рисунок 9.6 – Словарь терминальных символов

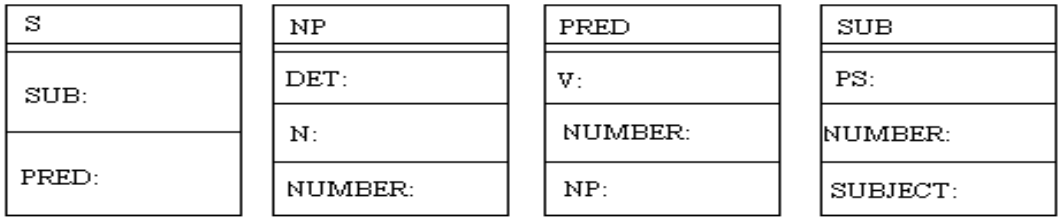
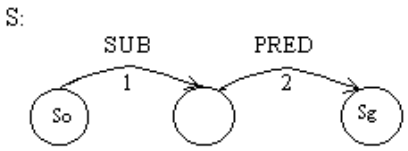


Рисунок 9.7 – Структуры нетерминальных символов

Группа подлежащего представляется либо местоимением (PRON - pronoun), либо подструктурой NP, состоящей из артикля (DET) и существительного (N). В структуре SUB введены следующие слоты: PS – фиксирует чем представлена группа подлежащего (возможные значения PRON – местоимение и NP – артикль и существительное); NUMBER – значение числа; SUBJECT – само подлежащие. Структура группы сказуемого содержит три слота: V – глагол, NUMBER – число, NP – подструктура из артикля и существительного, соответствующая дополнению предложения.

Рассматриваемая РСП-грамматика, изображена на рисунках 9.8-9.11. Дуги сетей переходов отмечены цифрами, соответствующими номерам присоединенных процедур. Описания присоединенных процедур приведены рядом с изображением соответствующего графа РСП. Доступ к значениям слотов соответствующих структур осуществляется с помощью

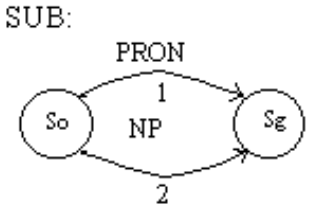
составных имен, образуемых из имени структуры и имени слота. Например, запись SUB.NUMBER обеспечивает доступ к слоту NUMBER структуры SUB. При каждом вызове РСП для нетерминального символа создается новая структура, слоты которой заполняются с помощью соответствующих присоединенных процедур. Если достигается конечное состояние, то РСП возвращает указатель на экземпляр структуры нетерминального символа с заполненными значениями слотов. Поэтому слоты структур верхних уровней содержат ссылки на экземпляры структур нижних уровней. Структура, соответствующая уровню предложения, будет соответствовать корневому узлу дерева грамматического разбора входного высказывания.




```
Procedure S.1;
  SUB:= структура, возвращаемая сетью SUB;
  S.SUB:= SUB;
End;

Procedure S.2;
  PRED:= структура, возвращаемая сетью PRED;
  If SUB.NUMBER = PRED.NUMBER Then
     Begin
              S.PRED:=PRED;
              Возврат (S)
     End
  Else  Возврат (НЕУДАЧА)
 End;
```

Рисунок 9.8 – РСП предложения




```
Procedure SUB.1;
  PRON:=определение текущего слова из словаря;
  If PRON.POS = местоимение Then
    Begin
      SUB.PS = 'PRON';
      SUB.NUMBER = PRON.NUMBER;
      SUB.SUBJECT = PRON;
      Возврат (SUB);
     End;
  Else Возврат (НЕУДАЧА)
End;
Procedure SUB.2;
  NP:= структура, возвращаемая сетью NP;
  If (NP.DET.POS = артикль) and (NP.N.POS = существительное) Then
              Begin
                      SUB.PS = 'NP';
                      SUB.NUMBER = NP.N.NUMBER;
                      SUB.SUBJECT = NP;
                      Возврат (SUB);
              End;
  Else Возврат (НЕУДАЧА)
End;
```

Рисунок 9.9 – РСП группы подлежащего

PRED:

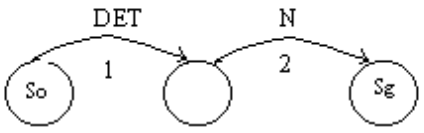



```
Procedure PRED.1;
  V:= определение текущего слова из словаря;
  If V.POS = глагол Then
     Begin
        PRED.V = V;
        PRED.NUMBER = V.NUMBER
     End
End;

Procedure PRED.2;
Begin
      NP:= структура, возвращаемая сетью NP;
      PRED.NP:=NP;
      Возврат (PRED)
End;
```

Рисунок 9.10 – РСП группы сказуемого

NP:




```
Procedure NP.1;
  DET:=определение текущего
        слова из словаря;
  If  DET.POS = артикль Then
        NP.DET:= DET
End;

Procedure NP.2;
      N:= определение текущего слова из словаря;
      If (N.POS = существительное) and
        (N.NUMBER согласуется с NP.DET.NUMBER) Then
            Begin
                  NP.N:= N;
                  NP.NUMBER:= N.NUMBER;
                  Возврат(NP)
            End
      Else    Возврат (НЕУДАЧА)
End;
```

Рисунок 9.11 – РСП артикля и существительного

В качестве примера на рисунке 9.12 изображено дерево грамматического разбора предложения “He likes the dog” (Он любит собаку).

Сформированное дерево грамматического разбора используется в целях семантической интерпретации. Интерпретатор формирует внутреннее представление входного предложения в процессе обхода дерева грамматического разбора, начиная с его корня. В каждом узле дерева процедура интерпретации вызывается рекурсивно, обеспечивая интерпретацию дочерних вершин узла. Возвращаемые результаты рекурсивных вызовов комбинируются, образуя единый концептуальный граф. Рекурсивные вызовы прерываются на уровне листьев дерева грамматического разбора. Если листья представлены существительными, глаголами, местоимениями или прилагательными, то они ассоциируются с сущностями предметной области, извлекаемыми из базы знаний. В рассматриваемом примере база знаний должна содержать определения следующих понятий: dog (собака), man (мужчина), he (он), it (оно), like (любить), see (видеть). Возможная иерархия типов понятий рассматриваемого примера изображена на рисунке

9.13. Дополнительно к понятиям будем использовать лингвистические отношения: агент, субъект, объект, инструмент (см. § 3.4). Понятия, выражаемые глаголами, представляются в базе знаний с помощью падежных рамок (рисунок 9.14).

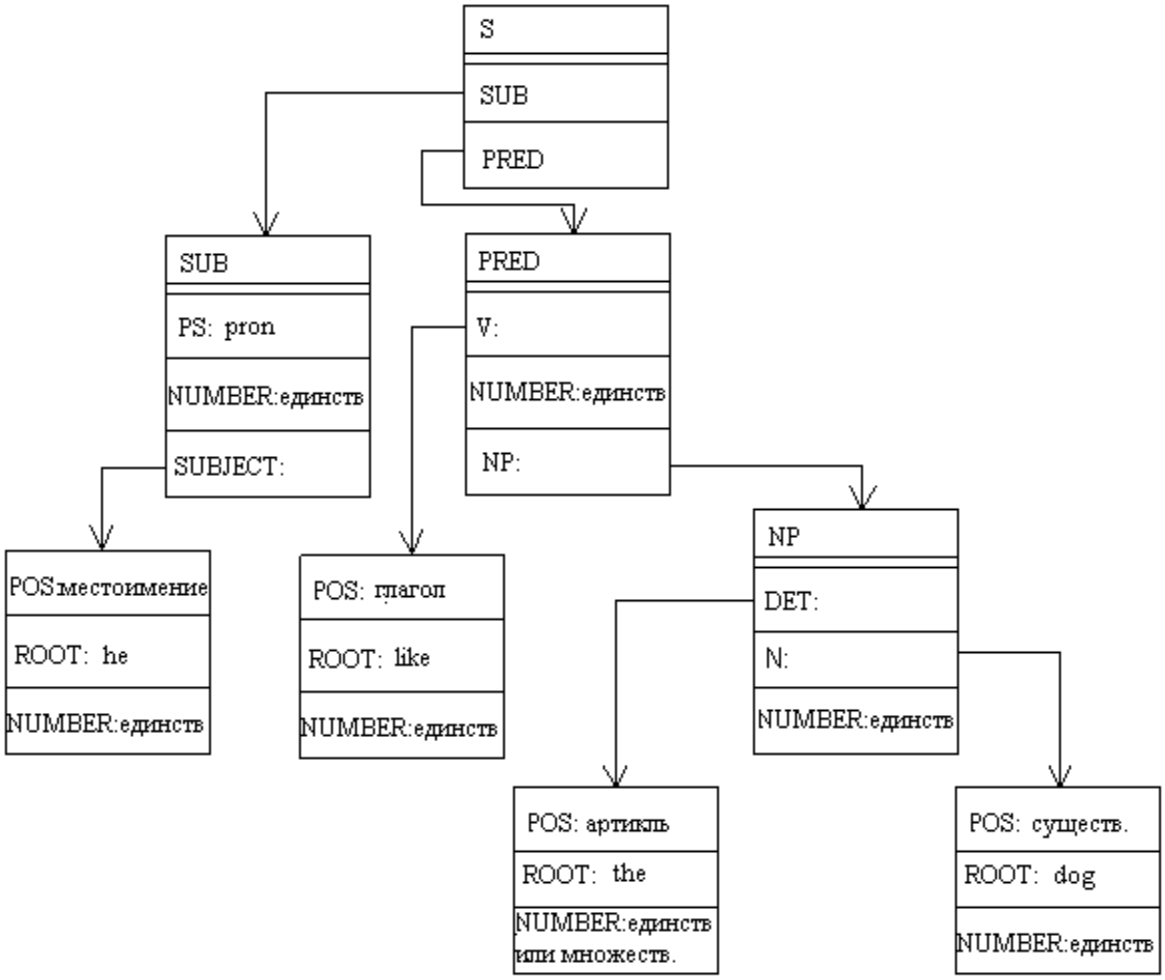


Рисунок 9.12 – Дерево грамматического разбора предложения

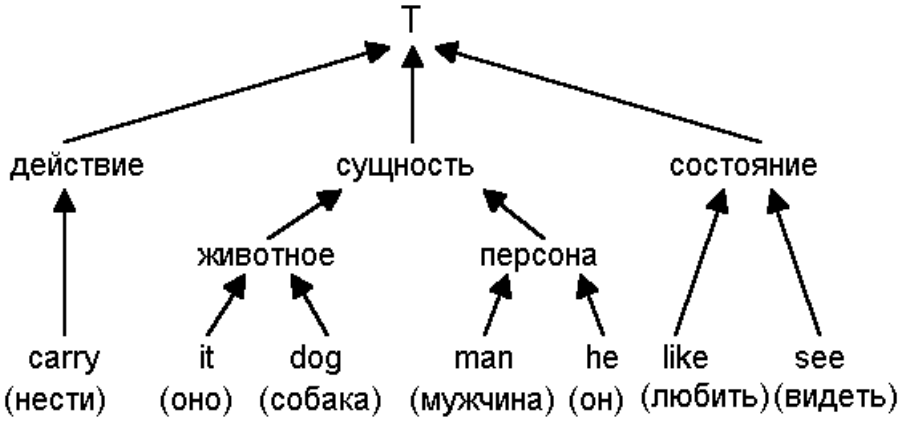


Рисунок 9.13 – Иерархия типов понятий примера

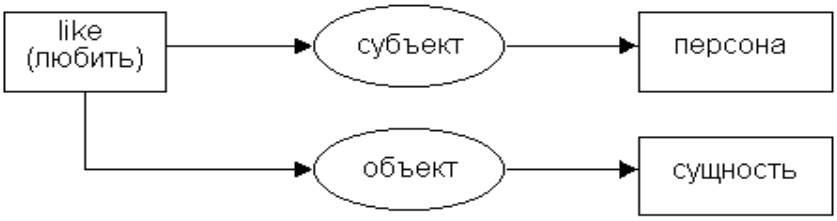


Рисунок 9.14 – Падежная рамка глагола “like”

Формирование внутреннего представления входного предложения выполняется с помощью приведенных ниже процедур интерпретатора, вызываемых в соответствие с деревом грамматического разбора:

```
Procedure SENTENCE(S);
Begin
      Вызвать процедуру SUBJECT(S.SUB) для интерпретации подле-
      жащего S.SUB;
      Вызвать процедуру PREDICATE(S.PRED) для интерпретации ска-
      зуемого S.PRED;
      Используя операции объединения и специализации связать поня-
      тие, выраженное подлежащим, с субъектом (агентом) на концепту-
      альном графе понятия, представленным сказуемым;
End;

Procedure SUBJECT(SUB);
Begin
      If SUB.PS = 'PRON' Then
        Вызвать PRONOUN(SUB.SUBJECT) для интерпретации понятия,
        выраженного местоимением;
      Else
        Вызвать P_NOUN(SUB.SUBJECT) для интерпретации понятия,
        выраженного существительным с артиклем
End;

Procedure PREDICATE(PRED);
Begin
      Вызвать процедуру VERB (PRED.V) с целью интерпретации глаго-
      ла PRED.V;
      Вызвать процедуру P_NOUN(PRED.NP) для интерпретации суще-
      ствительного с предлогом;
      Используя операции объединения и специализации, связать объ-
      ект концептуального графа глагола с существительным;
End;

Procedure VERB (X);
Begin
      Извлечь из базы знаний падежную рамку глагола X;
End;
```

```
Procedure P_NOUN (NP);
Begin
      Вызвать процедуру NOUN (NP.N) для интерпретации существи-
      тельного NP.N
      If  (NP.NUMBER = единственное) and (NP.DET.ROOT = a) Then
            Представить понятие, выраженное существительным, в об-
            щей форме;
      If (NP.NUMBER = единственное) and (NP.DET.ROOT = the) Then
            Связать с понятием, представленным существительным,
            индивидный числовой маркер;
      If(NP.NUMBER = множественное) Then
            Связать с понятием, представленным существительным,
            метку множественного числа;
End;

Procedure NOUN (X);
Begin
      Извлечь из базы знаний существительное X
End.

Procedure PRONOUN (X);
Begin
      Извлечь из базы знаний местоимение X
End.
```

Артикли английского предложения не представляются в виде отдельных понятий на концептуальных схемах. В рассматриваемом примере они лишь определяют, является ли объект предметной области обобщенным или индивидным. Если объект обобщенный, то на концептуальной схеме он представляется общим понятием с учетом иерархии понятий, изображенной на рисунке 9.13. Если объект индивидный, то он снабжается специальным числовым или именным маркером, отделяющим данное понятие от аналогичных понятий, имеющихся в базе знаний. Указанные особенности интерпретации отражены в тексте процедуры P_NOUN.

На рисунке 9.15 изображен процесс интерпретации предложения “He likes the dog”, использующий дерево грамматического разбора (рисунок 9.12).

Анализаторы, построенные на основе рассмотренных принципов, оказываются чрезвычайно чувствительными к грамматике входного языка. Если некоторое предложение содержит отклонения от грамматики, что характерно для диалога между людьми, то ATN – анализатор терпит неудачу. Один из подходов преодоления указанного недостатка ATN-анализаторов заключается в параллельном применении нескольких грамматик и объединении результатов интерпретации.

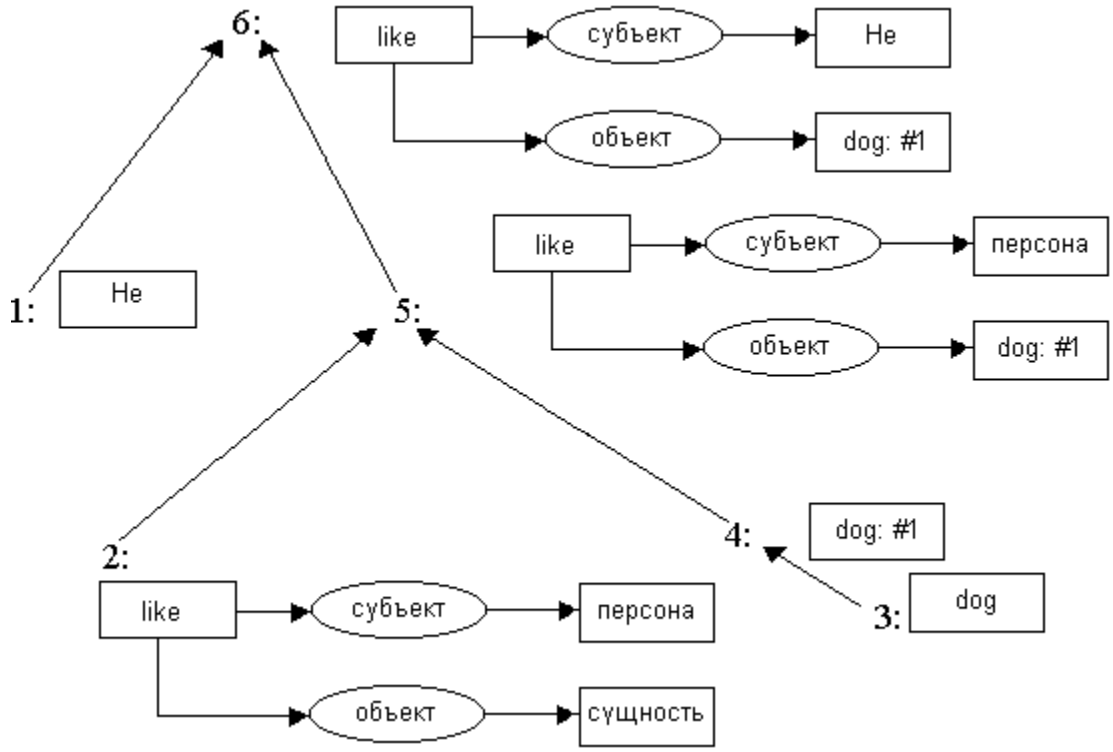


Рисунок 9.15 – Интерпретация предложения “He likes the dog”

9.2.2. Эмпирический подход

Эмпирический подход к анализу ЕЯ первоначально был популярен в 50-х годах, когда К. Шеннон создал основы теории информации. В этот период разрабатывались методы автоматического извлечения лексической и синтаксической информации из текстов. Основная цель состояла в разработке алгоритмов, обеспечивающих определение структуры языка на основе анализа вероятностей распределения слов.

Работы в этом направлении были приостановлены после того, как Н.Хомский, в конце 50-х годов, определил новые задачи компьютерной лингвистики, заложив основы теории порождающих формальных грамматик. Эмпирический подход вновь получил распространение к концу 80-х годов. В это время были разработаны несколько программ индуктивного извлечения лексической и семантической информации из текстов, на основе которых Р. Гэрсайдом и коллегами был проведен вычислительный анализ английского языка. Кроме того, фирма IBM достигла значительных успехов в области распознавания речи, благодаря использованию статистических методов, в частности, скрытых марковских моделей (СММ). Параметры моделей определялись с помощью обучающих алгоритмов на ре-

альных речевых данных. Позже статистические методы обработки речи были распространены на другие области обработки естественного языка.

Одно из первых успешных применений обучающих алгоритмов в лингвистике связано с задачей распознавания частей речи (существительных, глаголов, артиклей, и пр.) в предложении. Современные технологии позволяют решать эту задачу с точностью близкой к точности, которая получается при анализе предложений людьми [61]. Распознавание частей речи – важная составляющая систем понимания высказываний. С появлением больших баз данных, содержащих предложения с соответствующими деревьями грамматического разбора, эмпирические методы распространились на задачу синтаксического анализа предложений. В настоящее время эмпирический подход применяется для установления смысла неоднозначных слов, сегментации дискурса, выделения анафор (повторение в разных частях предложения или текста одного и того же понятия, которое может представляться разными словами), семантического анализа.

Широкое использование в настоящее время эмпирического подхода объясняется различными причинами. Основная причина состоит в том, что эмпирический подход предлагает потенциальные решения давних проблем, связанных с обработкой естественного языка, как-то:

- автоматическое пополнение знаний;
- широту охвата понятий рассматриваемой предметной области;
- устойчивость соответствующих моделей по отношению к шуму и неполноте данных;
- простота расширения систем по отношению к новым данным или новым предметным областям.

К иным причинам роста популярности эмпирического подхода в последнее время следует отнести:

- относительную доступность высокопроизводительных вычислительных средств для многих исследователей;
- разработку и создание больших баз лингвистических данных, необходимых для обучения и тестирования ЕЯ-систем;
- ориентацию на создание систем, работающих в реальных условиях.

Рассмотрим особенности эмпирического подхода на примере распознавания частей речи. Задача состоит в присвоении словам предложения меток: существительное, глагол, предлог, прилагательное и т.п. Кроме этого, необходимо определять некоторые дополнительные признаки существительных и глаголов, например, для существительного – число, а для глагола – форму.

Формализуем задачу. Представим предложение в виде последовательности слов $W = w_1 w_2 \ldots w_n$, где $w_1, w_2, \ldots, w_n$ случайные переменные, каждая из которых получает одно из возможных значений, принадлежа-

щих словарю языка. Последовательность меток, назначаемых словам предложения, представим последовательностью $X = x_1 x_2 \ldots x_n$, где $x_1, x_2, ..., x_n$ – случайные переменные, значения которых определены на множестве возможных меток. Тогда задача распознавания частей речи состоит в поиске наиболее вероятной последовательности меток $x_1 x_2 ... x_n$ по заданной последовательности слов $w_1 w_2 \ldots w_n$. Иными словами, необходимо найти такую последовательность меток $X^* = x_1 x_2 \ldots x_n$, которая обеспечивает максимум условной вероятности $P(x_1, x_2, \ldots x_n \mid w_1 w_2 \ldots w_n)$, т.е.

$$X^* = \arg \max_X P(X \mid W). \tag{9.1}$$

Перепишем $P(X \mid W)$ в виде

$$P(X \mid W) = P(X, W) / P(W). \tag{9.2}$$

Так как требуется найти максимум $P(X \mid W)$ по переменной X, то из (9.2) следует, что

$$X^* = \arg \max_X P(X, W). \tag{9.3}$$

Совместную вероятность $P(X \mid W)$ можно записать в виде произведения условных вероятностей

$$P(X, W) = P(x_1) P(w_1 \mid x_1) P(x_2 \mid x_1 w_1) P(w_1 \mid x_1 w_1) P(x_3 \mid x_1 x_2, w_1 w_2) ... =$$

$$= \prod_{i-1}^{n} P(x_i \mid x_1, \ldots, x_{i-1}, w_1, \ldots w_{i-1}) P(w_i \mid x_1, \ldots, x_{i-1}, w_1, \ldots w_{i-1}). \tag{9.4}$$

Непосредственный поиск максимума выражения (9.4) представляет собой сложную задачу, так как при больших значениях *n* поисковое пространство становится очень большим. Поэтому вероятности $P(x_i \mid x_1, \ldots, x_{i-1}, w_1, \ldots w_{i-1})$ и $P(w_i \mid x_1, \ldots, x_{i-1}, w_1, \ldots w_{i-1})$ аппроксимируют более простыми условными вероятностями $P(x_i \mid x_{i-1})$ и $P(w_i \mid x_i)$. В этом случае полагают, что значение метки x_i связано только с предыдущей меткой x_{i-1} и не зависит от более ранних меток, а также, что вероятность слова w_i определяется только текущей меткой x_i. Указанные предположения на-

зывают *марковскими.* Для решения задачи (9.3) привлекают теорию марковских моделей (см.§ 9.3.5).

С учетом марковских предположений (9.3) запишется в виде

$$X^* = \arg\max_{x_1,\ldots,x_n} \prod_{i=1}^{n} P(x_i \mid x_{i-1}) P(w_i \mid x_i), \tag{9.5}$$

где условные вероятности $P(x_i \mid x_{i-1})$ и $P(w_i \mid x_i)$ оцениваются на множестве обучающих данных. Поиск последовательности меток X^* осуществляют с помощью алгоритма динамического программирования Витерби.

Алгоритм Витерби может рассматриваться как вариант алгоритма поиска на графе состояний, где вершинам соответствуют метки слов. Характерно, что для любой текущей вершины множество дочерних вершин всегда одно и то же. Более того, для каждой дочерней вершины множества родительских вершин тоже совпадают. Это объясняется тем, что на графе состояний осуществляются переходы с учетом всех возможных сочетаний меток:

предлог предлог;
предлог глагол;
… .
предлог существительное;
… .
существительное предлог;
… .
существительное существительное.

Если имеется *n* меток и $P(x_i \mid x_{i-1}) \neq 0$ для всех значений i, то в каждую вершину-метку будут вести *n* путей. Наилучшим считается путь с наибольшим значением вероятности $P(X_1^t \mid W_1^t)$, где $X_1^t = x_1, x_2, \ldots, x_t$ и $W_1^t = w_1, w_2, \ldots, w_t, t \leq n$. Таким образом, на каждом шаге поиска продолжаются *n* путей. На последнем шаге из *n* последовательностей (путей) выбирается одна с учетом (9.5). Подробнее алгоритм Витерби будет описан на примере задачи распознавания речи.

Предположения Маркова обеспечивают существенное упрощение задачи распознавания частей речи при сохранении высокой точности назначения меток словам. Так, при наличии 200 меток точность назначения меток примерно равна 97% [77]. Успешное определение частей речи предложения – ключ к решению задачи синтаксического анализа ЕЯ-предложений.

Многие достижения в области синтаксического анализа предложений на основе эмпирического подхода стали возможны благодаря созда-

нию открытых банков грамматических данных. В качестве примера можно указать Пенн-банк, содержащий около 40000 предложений с деревьями грамматического разбора, и WORDNET – лексическую базу слов английского языка, связанных семантическими отношениями типа ako, part_of и другими [61]. Наличие таких банков данных обеспечивает тестирование и обучения систем обработки ЕЯ, разрабатываемых различными исследователями, на одних и тех же наборах данных.

Долгое время эмпирический анализ выполнялся с помощью стохастических контекстно-свободных грамматик (СКС - грамматик, англ. PCFG – probabilistic context-free grammars). Однако для СКС-грамматик характерен существенный недостаток. Он заключается в том, что различным грамматическим разборам могут назначаться одинаковые вероятности. Это происходит из-за того, что вероятность грамматического разбора представляется в виде произведения вероятностей правил, участвующих в разборе. Если в ходе разбора используются различные правила, характеризуемые одинаковыми вероятностями, то это и порождает указанную проблему.

Лучшие результаты даёт грамматика, учитывающая лексику языка. В этом случае в правила включаются необходимые лексические сведения, которые обеспечивают различные значения вероятности для одного и того же правила в разных лексических окружениях [61].

В заключение отметим, что эмпирический синтаксический анализ в большей степени соответствует распознаванию образов, чем традиционному грамматическому разбору в его классическом понимании. Сравнительные исследования показали, что правильность эмпирического грамматического разбора предложений естественного языка оказывается выше, по сравнению с традиционным грамматическим разбором.

9.3. ЕЯ-интерфейс доступа к базам данных

В настоящее время успешное применение технологий обработки естественного языка ограничивается узкоспециальными областями с хорошо определенной семантикой, к которым относятся базы данных (БД). Рассмотрим построение ЕЯ-интерфейса БД на основе рационального подхода, в ходе которого выполняется анализ и интерпретация входных ЕЯ-запросов средствами традиционных грамматик. Основная задача такого интерфейса – преобразование ЕЯ-запросов в запрос на внутреннем языке базы данных, например SQL.

Применение ЕЯ-интерфейса для доступа к базам данных характеризуется следующими достоинствами:

- пользователю не требуется изучать сложные искусственные языки доступа к БД;
- простота формулировки сложных запросов, содержащих отрицания и кванторы общности, которые весьма трудно сформировать при использовании традиционных “оконных” интерфейсов доступа к БД;
- возможность обработки запросов с учетом дискурса, что допускает ввод запросов, содержащих анафоры и эллипсисы.

Существующие ЕЯ-интерфейсы баз данных используют различные подходы [59]: метод сопоставления с образцом, синтаксические грамматики, семантические грамматики, некоторый внутренний язык.

Интерфейсы, основанные на методе сопоставления с образцом, являются наиболее простыми. Обобщенный алгоритм сопоставления с образцом и пример его применения к ведению диалога на ограниченном ЕЯ рассмотрен в § 5.25. Уточним особенности анализа ЕЯ-запросов на основе процедуры сопоставления с образцом. Рассмотрим таблицу реляционной БД, содержащую сведения о странах (рисунок 9.16).

Страны

Страна	Столица	Язык
Франция	Париж	Французский
Германия	Берлин	Немецкий
Италия	Рим	Итальянский
…	…	…

Рисунок 9.16 – Таблица реляционной БД

В этом случае при ответе на запрос могут использоваться следующие правила:

образец: … “столиц&” … ?S;
действие: вывести значение поля Столица для строки таблицы, удовлетворяющей условию Страна = S;

образец: … “столиц&” … “стран&”;
действие: вывести значения полей Страна и Столица для каждой строки таблицы.

В соответствии с первым правилом, если в запросе пользователя после слова “столиц&” будет следовать название страны (т. е. одно из возможных значений поля Страна), то необходимо найти строку таблицы, для которой значение поля Страна сопоставимо со значением переменной образца S, и сообщить пользователю соответствующее значение поля Столица. Например, если пользователь введет запрос “Столица Германии ?”, то

ответ будет “Берлин”. Этот же ответ будет получен, если пользователь введет запрос в иной форме: “Назовите столицу Германии?”. Второе правило активизируется, если в запросе пользователя после слова “столиц&” следует слово “стран&”. Например: “Список столиц всех стран?”. В этом случае пользователю сообщается весь список стран и их столиц. ЕЯ-интерфейсы БД, построенные на основе метода сопоставления с образцом, могут приводить к ошибочным ответам, так как не учитывают семантических отношений между элементами запросов.

Более гибкими являются ЕЯ-интерфейсы, реализующие синтаксический разбор входных запросов к БД. Строящееся при этом дерево грамматического разбора преобразуется в выражение на языке запроса БД. Например, в БД LUNAR, содержащей сведения о результатах химического анализа образцов лунного грунта, используется простая грамматика, подобная грамматике, изображенной на рисунке 2.5. Если система получает запрос: “Which rock contains magnesium?” (“Какая порода содержит магний?”), то строится дерево синтаксического разбора, изображенное на рисунке 9.17. Затем дерево разбора отображается в запрос на языке БД [59]

```
( for_every x   (is_rock x)
                (contains x magnesium);
                (printout x )),
```

где x – переменная. Такое отображение выполняется с помощью следующих простых правил:

“which” → for_every x ;
“rock” → (is_rock x);
группа подлежащего (NP), состоящая из определителя (DET) и существительного (N), представленных словами which и rock, записывается в виде выражения for_every x (is_rock x);
“contains” → contains x; “magnesium” → magnesium;
группа сказуемого (VP), состоящая из глагола (V – contains) и существительного (N – magnesium), заменяется выражением (contains x magnesium); предложение S представляется в виде (NP VP; (print_out x)).

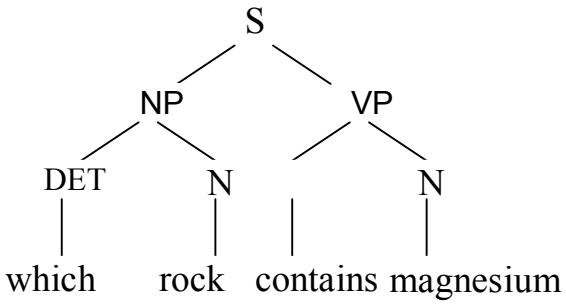


Рисунок 9.17 – Дерево синтаксического разбора в БД LUNAR

ЕЯ-интерфейс, реализующий рассмотренное преобразование дерева синтаксического разбора в конструкции языка запросов конкретной БД, оказывается узкоспециальным. Создание универсального множества правил, преобразующих дерево синтаксического разбора, в конструкции полноценного языка запросов, например SQL, проблематично.

Иная возможность построения ЕЯ-интерфейсов доступа к БД основана на использовании семантических грамматик, в которых нетерминальные элементы представляются семантическими категориями. В этом случае также строится дерево грамматического разбора, которое затем преобразуется в выражения на языке запросов к БД. Недостатком ЕЯ-интерфейсов, реализующих метод семантической грамматики, является их непереносимость на другую предметную область, так как это требует разработки новой грамматики.

Во многих случаях существующие ЕЯ-интерфейсы, обеспечивающие доступ к БД, первоначально преобразуют запрос в некоторую внутреннюю логическую форму. Такая промежуточная логическая форма отражает смысл вопроса пользователя в терминах, не зависящих от структуры БД. Затем логический запрос преобразуется в выражение языка запросов БД и обрабатывается системой управления базы данных (СУБД). На рисунке 9.18 изображена возможная структурная схема ЕЯ-интерфейса доступа к БД с промежуточным логическим представлением запроса [59].

Входной ЕЯ-запрос вначале подвергается синтаксическому анализу, в ходе которого строится дерево грамматического разбора с помощью синтаксических правил. Семантический интерпретатор преобразует дерево разбора в промежуточный логический запрос на основе семантических правил. Семантические правила обеспечивают построение логического выражения для левой части синтаксических правил в виде логической функции от элементов, входящих в их правую часть. Логическое выражение соотносится с понятиями, хранящимися в словаре. Во многих случаях синтаксические и соответствующие семантические правила не зависят от предметной области. Однако слова (понятия), соответствующие листьям дерева разбора, а также логические выражения, построенные с помощью понятий, оказываются проблемно-зависимыми. Поэтому их хранение организуется в виде словаря (лексикон). В словаре содержатся слова, которые могут встречаться в пользовательском запросе (“страна”, “страны”, “граница”, “граничат”, “граничить” и т.п.), и понятия, представленные логическими выражениями, описывающими смысл каждого слова. Например, смысл слов “столица”, “граничат”, “страна” может представляться с помощью следующих предикатов языка Пролог: **столица(Столица,Страна), граничат(Страна1,Страна2), явл_страной(Страна)**.

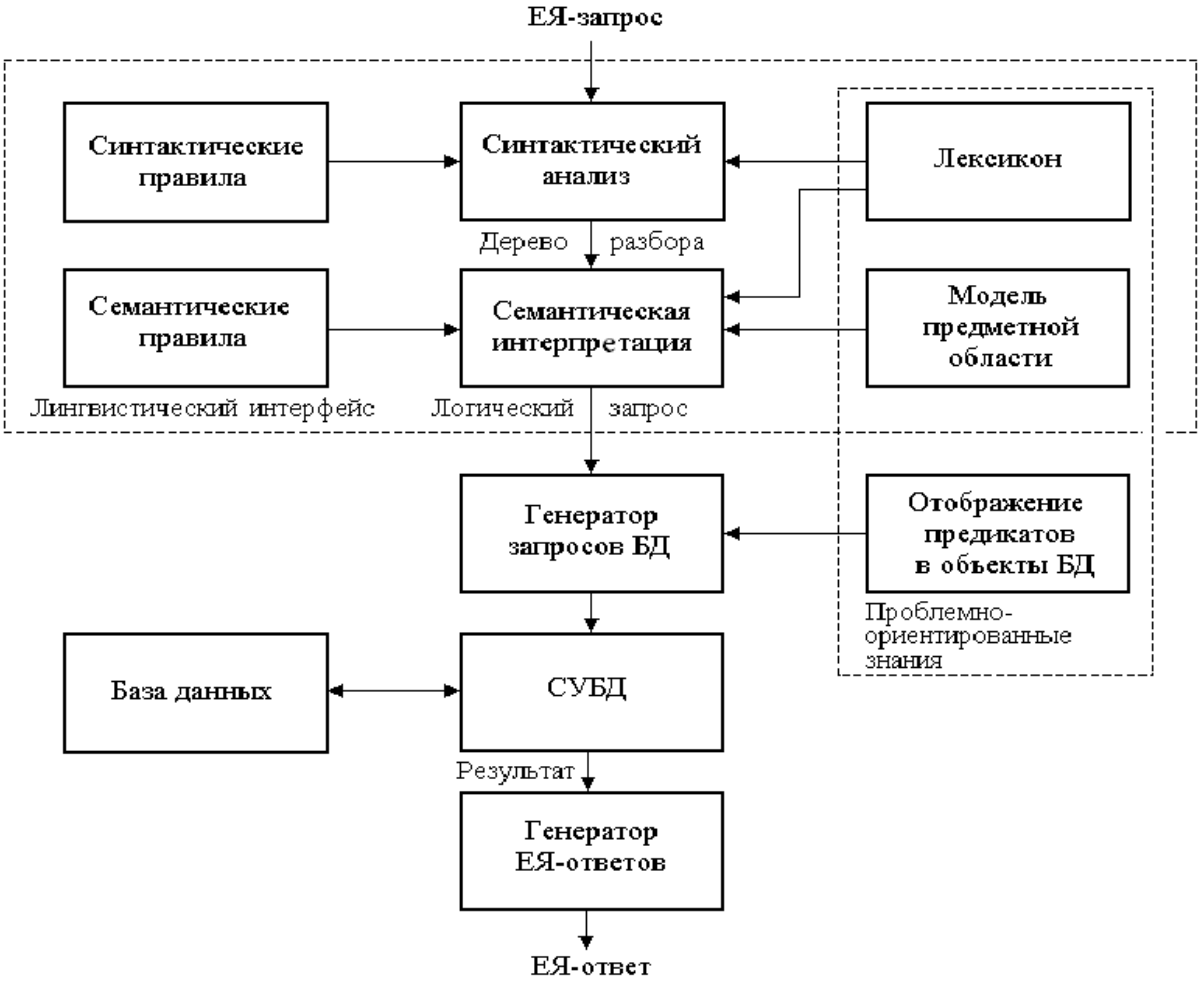


Рисунок 9.18 – ЕЯ интерфейс с промежуточным логическим представлением запроса

Семантический интерпретатор конструирует логический запрос из логических выражений лексикона. Так, вопрос: “Назовите столицы стран, которые граничат с Германией?” преобразуется в логический запрос, конструируемый из предикатов лексикона:

```
ответ([Столица, Страна]):-
        явл_страной(Страна),
        граничат(Страна, ‘Германия’),
        столица(Столица, Страна).
```

В ходе конструирования логического запроса семантический интерпретатор учитывает структурные отношения, задаваемые моделью предметной области. Обычно эта модель содержит иерархию классов предметной области и накладывает ограничения на типы аргументов предикатов. Для того чтобы проиллюстрировать необходимость ограничения типов аргументов предикатов, рассмотрим простой пример из области бизнеса.

Предположим, что БД содержит сведения о заработной плате служащих некоторой компании. Иерархия классов предметной области изображена на рисунке 9.19. Тогда вопрос "Какова зарплата каждого клиента?" должен остаться без ответа, так как в базе данных нет сведений о зарплате клиентов (в этом случае полагается, что служащий компании не может быть ее клиентом).

Предположив, что в словаре со словом "зарплата" связан предикат **зарплата(Зарплата,Персона)**, а со словом "клиент" – предикат **явл_клиентом(Персона)**, получим следующий логический запрос:

```
ответ ([Зарплата, Персона]):-
        явл_клиентом (Персона),
        зарплата (Зарплата, Персона).
```

Разумно потребовать, чтобы аргумент предиката **явл_клиентом** принадлежал классу *клиенты*, а аргументы предиката **зарплата** – классам *зарплата* и *служащие*. В этом случае приведенный выше логический запрос будет противоречивым, так как переменная **Персона** должна будет принадлежать двум классам одновременно, что не соответствует рисунку 9.19. Поэтому пользователю будет выведено сообщение о несоответствии запроса модели предметной области.

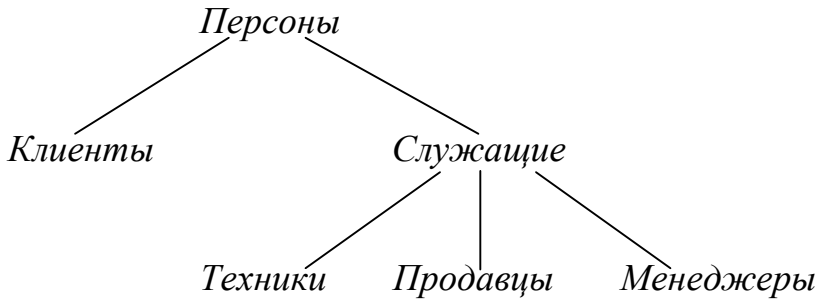


Рисунок 9.19 – Иерархия классов

Если же пользователем введен запрос "Какова зарплата каждого техника?", то он будет преобразован в следующий логический запрос:

```
ответ ([Зарплата, Персона]):-
        явл_техником(Персона),
        зарплата (Зарплата, Персона).
```

В этом случае запрос будет признан корректным, так как класс *техники* является подклассом класса *служащие*.

Введение ограничений на классы принадлежности аргументов позволяет также избежать двусмысленности. Рассмотрим, например, вопрос: "Список всех служащих компании с лицензией на экспорт". Этот вопрос может интерпретироваться двояко:

```
; вариант1
        ответ ([Служащий]):-
                явл_служащим (Служащий),
                явл_компанией (Компания),
                явл_экспортн_лицензией (Экспорт_Лицензия),
                работает_в (Служащий, Компания),
                имеет (Компания, Лицензия).

; вариант 2
        ответ ([Служащий]):-
                явл_служащим (Служащий),
                явл_компанией (Компания),
                явл_экспортн_лицензией (Экспорт_Лицензия),
                радотает_в (Служащий, Компания),
                имеет (Служащий, Экспорт_Лицензия).
```

Можно потребовать, чтобы аргументы предикатов **явл_служащим**, **явл_компанией**, **явл_экспортн_лицензией** относились к классам: *служащие*, *компании* и *экспортные_лицензии*. Кроме этого, можно наложить ограничения на типы аргументов предиката **имеет**. Если второй аргумент данного предиката принадлежит классу *экспортные_лицензии*, то первый аргумент должен принадлежать классу *компании*. В этом случае второй вариант интерпретации вопроса будет признан некорректным.

Так как логический запрос не содержит ссылок на объекты БД, то он не может быть непосредственно использован в процессе поиска необходимых сведений. Генератор запросов БД (рисунок 9.18) осуществляет преобразование логического запроса в запрос на языке БД. При этом используются сведения об отображениях предикатов в объекты базы данных. В случае реляционных БД устанавливается простая связь между каждым предикатом и оператором SELECT структурированного языка запросов SQL. Пусть БД содержит таблицы, изображенные на рисунке 9.20. Требуется ответить на вопрос: “Назовите столицы всех стран, которые граничат с Германией? ”.

Табл_Страны

Страна	Столица	Язык
Франция	Париж	Французский
Германия	Берлин	Немецкий
Италия	Рим	Итальянский
…	…	…

Табл_граничат

Страна1	Страна2
Франция	Германия
Франция	Швейцария
Германия	Франция
…	…

Рисунок 9.20 – Таблицы реляционной БД

В результате выполнения синтаксического и семантического анализа получим:

ответ ([Столица, Страна]): –
явл_страной (Страна),
граничат (Страна, ‘Германия’),
столица (Столица, Страна).

Сведения об отображениях предикатов в отношения БД позволяют сформировать следующие фрагменты SQL запроса:

- для предиката **явл_страной**

```
SELECT страна
FROM Табл_страны;
```

- для предиката **граничат**

```
SELECT страна1, страна2
FROM Табл_граничат;
```

- для предиката **столица**

```
SELECT столица, страна
FROM Табл_страны.
```

С учетом исходного вопроса может быть сгенерирован следующий запрос на языке SQL:

```
SELECT   Табл_страны.столица,  Табл_страны.страна
FROM     Табл_страны,  Табл_граничат
WHERE    Табл_страны.страна =Табл_граничат.страна1
         AND Табл_граничат.страна2 = ’Германия’
```

Данный запрос поступает на вход СУБД, которая обеспечивает поиск необходимого ответа и передает его на вход генератора ЕЯ-ответов (рисунок 9.18).

Рассмотренный ЕЯ-интерфейс БД с промежуточным логическим представлением запросов характеризуется следующими достоинствами:

- благодаря выделению лингвистического интерфейса в самостоятельный модуль, настройка интерфейса на конкретную СУБД требует адаптации только генератора запросов БД;
- выделение проблемно-ориентированных знаний позволяет достаточно просто адаптировать интерфейс к различным предметным областям;
- использование внутреннего логического представления запросов позволяет организовать взаимодействие с модулями логического вывода и, тем самым, использовать в ходе вывода сведения из БД.

Последнее обстоятельство весьма важно, когда ответ на запрос пользователя не может быть сформирован с помощью простого поиска необходимых сведений в БД. Например, пусть речь идет о медицинской базе дан-

ных, содержащей сведения об именах пациентов, их возрасте, диагнозе заболевания, методах лечения и пр. Предположим, что пользователя интересует ответ на вопрос: “У каких пациентов вероятно развитие заболевания легких?”. Для ответа на данный вопрос система должна уметь на основе сведений, хранящихся в БД, делать необходимые логические выводы. Это требует введения в структурную схему, изображенную на рисунке 9.18, модуля логического вывода. Соответствующая структурная схема системы, обеспечивающей получение выводов на данных, содержащихся в БД, представлена на рисунке 9.21 [59]:

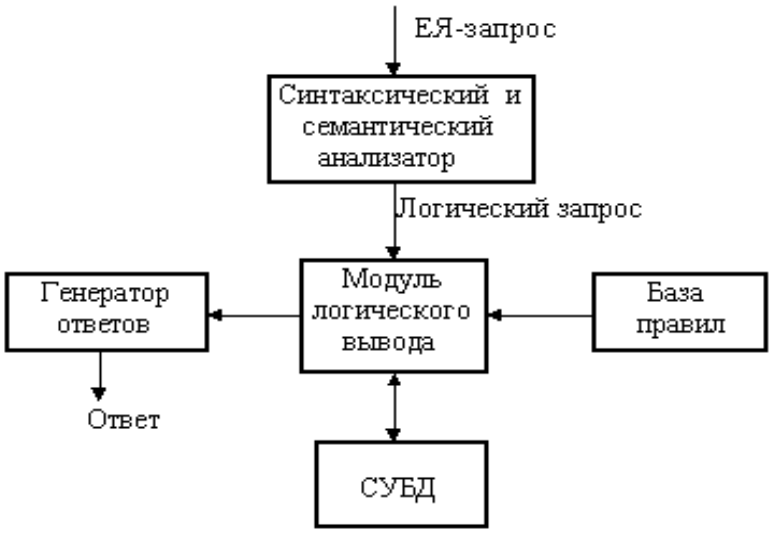


Рисунок 9.21 – ЕЯ-интерфейс БД с возможностью логических выводов

В этом случае приведенный выше вопрос пользователя с помощью синтаксического и семантического анализатора преобразуется в логическую форму:

ответ ([Пациент]): –
явл_пациентом (Пациент),
вероятно_заболеет (Пациент, заболевание_легких).

Здесь предикат **явл_пациентом(Пациент)** может быть отображен на отношение Табл_пациенты базы данных, содержащее сведения о пациентах клиники, а предикат **вероятно_заболеет(Пациент,заболевание_легких)** не может быть непосредственно ассоциирован с тем или иным отношением БД. Для таких предикатов в базе правил системы должны содержаться необходимые правила, обеспечивающие определение их значений. Например,

вероятно_заболеет(Пациент, Болезнь) если
вероятность(развитие(Пациент, Болезнь)) > 0,7.

Правило говорит, что **Пациент** вероятно заболеет болезнью **Болезнь**, если вероятность развития этой болезни больше **0,7**. Другие правила должны объяснять, как использовать сведения из БД, чтобы вычислить вероятность развития заболевания:

если имеет(Пациент, заболевание_крови) то
вероятность(развитие(Пациент, заболевание_легких)) = 0,6;
если курящий (Пациент) то
вероятность(развитие(Пациент, заболевание_легких)) = 0,7.

В системе также могут быть правила, которые позволяют определить вероятность развития заболевания, если к пациенту применимы несколько правил. Например, когда пациент имеет заболевание крови и курит.

Часто вопросы пользователя не полностью отражают его цели. В этом случае, если в системе не будет найден прямой ответ на вопрос, то ответ может быть кратким, например, “нет”. Рассмотрим следующий возможный диалог с системой:

Имеет ли авиакомпания MALEV рейс до Хельсинки из Борисполя?
Нет.

Было бы лучше, если бы система сформировала такой ответ:

Нет, но имеется рейс Украинских авиалиний.

Наличие модуля логического вывода позволяет глубже понимать цели пользователя и, тем самым, формировать ответы в духе сотрудничества, не ограничиваясь кратким ответом “да/нет”. Формирование таких ответов возможно, если в базе правил будет правило, утверждающее, что цель пользователя состоит в достижении конечного пункта маршрута, если запрос касается рейса. Тогда появляется возможность расширить ответ, исключив из логического запроса перевозчика. Полнее учесть цели пользователя позволяют диалоговые системы, структура которых дополняется моделью пользователя [16].

9.4. Распознавание речи

Речь выступает в роли основного средства коммуникации между людьми и поэтому речевое общение считается одним из важнейших компонентов СИИ. *Распознавание речи* представляет собой процесс преобразования акустического сигнала, формируемого на выходе микрофона или телефона, в последовательность слов. Более сложной является задача *по-*

нимания речи, которая сопряжена с выявлением смысла акустического сигнала. В этом случае выход подсистемы распознавания речи служит входом подсистемы понимания высказываний (рисунок 9.2).

Автоматическое распознавание речи (АРР) является одним из быстрорастущих и многообещающих направлений применения технологий обработки ЕЯ. АРР применяется при автоматизации ввода текстов в ЭВМ (диктант), при формировании устных запросов к базам данным или информационно-поисковым системам, при формировании устных команд различным интеллектуальным устройствам и др.

В настоящем параграфе рассматривается применение эмпирического подхода к решению задачи АРР.

9.4.1. Основные понятия

Системы распознавания речи характеризуются многими параметрами. Одним из основных параметров является ошибка распознавания слов (ОРС). ОРС представляет собой отношение количества нераспознанных слов к общему количеству произнесенных слов. Другими параметрами, характеризующими системы АРР, являются:

1) размер словаря;
2) режим речи;
3) стиль речи;
4) предметная область;
5) дикторо-зависимость;
6) уровень акустических шумов;
7) качество входного канала.

В зависимости от размера словаря системы АРР подразделяются на три группы: с малым размером словаря (до 100 слов); средним размером словаря (от 100 слов до нескольких тысяч слов); большим размером словаря (более 10000 слов).

Режим речи характеризует способ произнесения слов и фраз. Выделяют системы распознавания слитной речи и системы, позволяющие распознавать только изолированные слова речи. В режиме распознавания изолированных слов требуется, чтобы диктор делал краткие паузы между словами.

По стилю речи системы АРР подразделяются на две группы: системы детерминированной речи и системы спонтанной речи. В системах распознавания детерминированной речи диктор воспроизводит речь, следуя грамматическим правилам языка. Спонтанная речь характеризуется нарушениями грамматически правил и её сложнее распознавать.

В зависимости от предметной области выделяют системы АРР, ориентированные на применение в узкоспециальных областях (например, доступ к базам данным), и системы АРР с неограниченной областью применения. Последние требуют наличия словаря большого объема и должны обеспечивать распознавание спонтанной речи.

Многие системы АРР являются дикторо-зависимыми. Это предполагает предварительную настройку системы на особенности произношения конкретного диктора. Дикторо-независимые системы не требуют такой настройки.

По уровню окружающих акустических шумов выделяют системы, работающие при низких (<10дБ) и высоких уровнях шумов (>30дБ).

Качество входного канала характеризуется шириной полосы пропускания, стабильностью его характеристик, уровнем помех. По ширине полосы пропускания входные каналы АРР разделяют на широкополосные и узкополосные. Широкополосные каналы обеспечивают передачу речи в диапазоне частот 20 – 20000 Гц. Узкополосные каналы ограничивают полосу пропускания. Например, телефонная линия имеет ограниченную полосу пропускания от 300 до 3400 Гц.

В таблице 9.1 приведены сравнительные характеристики некоторых дикторо-независимых систем АРР [88]. Информационная система воздушных судов обеспечивает выдачу полетной информации по запросу к базе данных. Североамериканская система деловых новостей обеспечивает распознавание сообщений дикторов, осуществляющих тщательное чтение газетных статей. Система новостей радио и телевидения выполняет распознавание речи различных стилей. Система распознавания телефонных сообщений предназначена для распознавания спонтанной речи. Из таблицы 9.1 следует, что ошибка распознавания слов значительно увеличивается, если на стиль речи не накладываются ограничения или ухудшаются условия ввода акустических сигналов. Особенно большие ошибки наблюдаются при распознавании телефонных сообщений.

Сложность решения задачи распознавания речи объясняется большой изменчивостью акустических сигналов. Эта изменчивость объясняется несколькими причинами. Во-первых, различной реализацией *фонем* – основных единиц звукового строя языка. В русском языке насчитывают 42 фонемы (6 гласных и 36 согласных). Изменчивость реализации фонем вызвана влиянием соседних звуков в потоке речи. Оттенки реализации фонем, обусловленные звуковым окружением, называют *аллофонами.* Общее число аллофонов русского языка превышает 9000 [20]. Во-вторых, положением и характеристиками акустических приёмников. В-третьих, изменениями параметров речи одного и того же диктора, которые обусловлены различным эмоциональным состоянием диктора, темпом речи. В-

четвертых, различиями между дикторами, которые вызваны особенностями голосового тракта, диалектами языка.

Таблица 9.1 – Параметры некоторых систем АРР

Система	Размер словаря, слов	Стиль Речи	Уровень шумов	Качество канала	ОРС, %
Информационная система воздушных судов	2000	Спонтанный	Низкий	Широкополосный	2.1
Североамериканская система деловых новостей	60000	Детерминированный	Низкий	Широкополосный	6.6
Система новостей радио и телевидения	60000	Комбинированный	Различный	Различное	27.1
Система распознавания телефонных сообщений	23000	Спонтанный	Низкий	Телефонная линия	35.1

На рисунке 9.16 изображены основные компоненты системы распознавания речи. Оцифрованный речевой сигнал поступает на вход блока предварительной обработки, где осуществляется выделение признаков, необходимых для распознавания звуков. Распознавание звуков часто осуществляют с помощью моделей искусственных нейронных сетей (ИНС). Выделенные звуковые единицы используют в дальнейшем для поиска последовательности слов, в наибольшей степени соответствующей входному речевому сигналу. Поиск последовательности слов выполняется с помощью акустической, лексической и языковой моделей. Параметры моделей определяют по обучающим данным на основе соответствующих алгоритмов обучения.

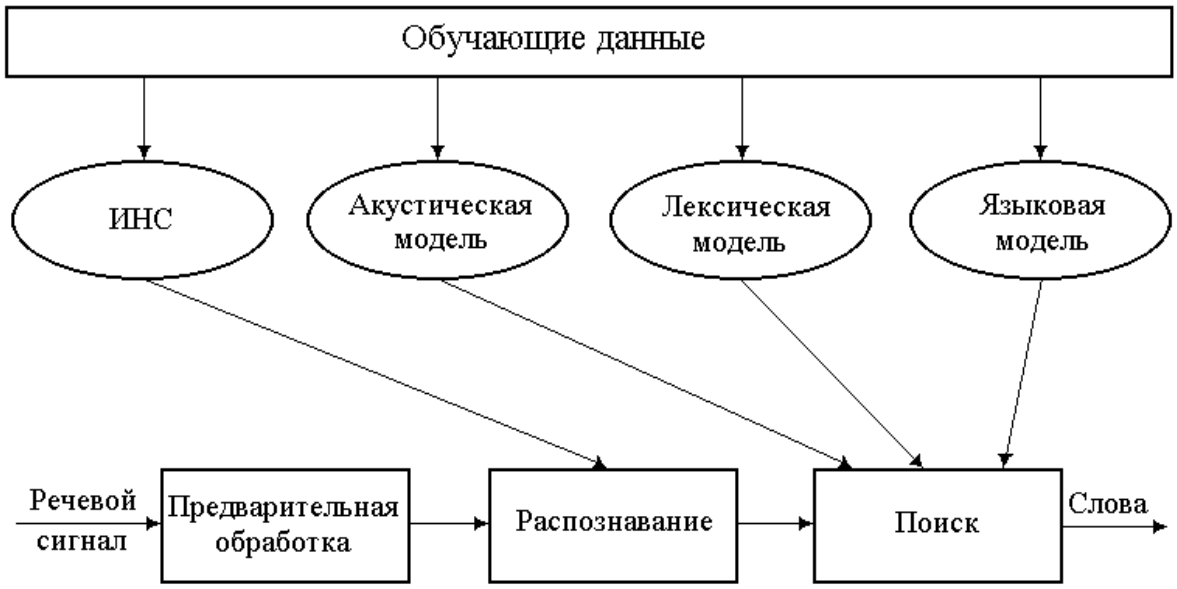


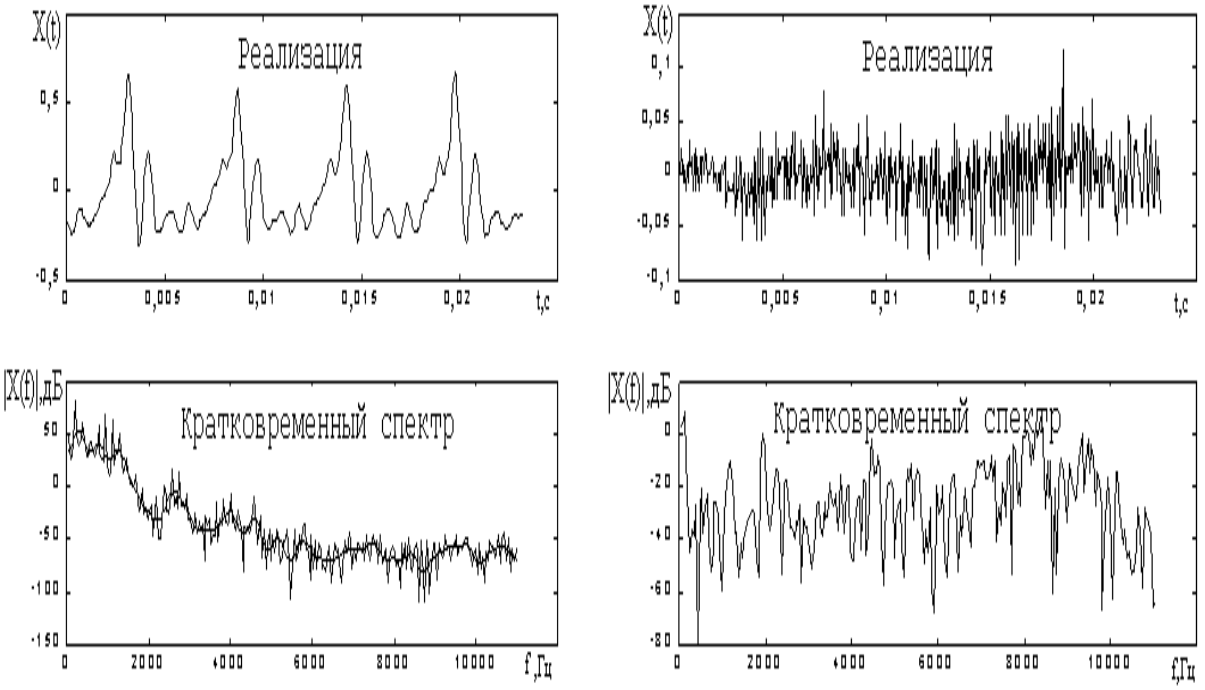
Рисунок 9.22 – Компоненты системы АРР

Рассмотрим подробнее указанные компоненты системы распознавания речи.

9.4.2. Предварительная обработка и распознавание звуков

Голосовой аппарат человека представляет собой акустическую систему, состоящую из ротового и носового каналов, возбуждаемую квазипериодическими импульсными колебаниями голосовых связок или турбулентным шумом. Турбулентный шум образуется путем проталкивания воздуха через сужения в определенных областях голосового тракта. Голосовой аппарат, возбуждаемый указанными источниками, действует как линейный фильтр с изменяющимися во времени параметрами, на выходе которого формируется речевой сигнал. На коротких интервалах времени речевой сигнал можно аппроксимировать сверткой возбуждающего сигнала с импульсной характеристикой голосового тракта. В соответствии с рассмотренной моделью звонкие (вокализованные) звуки формируются с участием голосовых связок, а шумовые (фрикативные) – за счет прохождения воздуха через сужения голосового тракта. Частота колебания голосовых связок при произнесении звонких звуков называется *частотой основного тона*. Форма голосового тракта остается неизменной на интервале 10-30 мс. На этом интервале речь рассматривается как стационарный случайный процесс. Поэтому большинство алгоритмов предварительной обработки обеспечивают анализ речи на указанном интервале времени.

Простейшее цифровое представление речи заключается в непосредственной дискретизации непрерывного цифрового сигнала. Выбор частоты дискретизации определяется решаемой задачей. Фрикативные звуки речи занимают сравнительно широкую полосу частот (примерно до 10 КГц). Вокализованные звуки, значительно влияющие на разборчивость речи, занимают полосу частот примерно до 3 КГц. Таким образом, частота дискретизации обычно выбирается в пределах от 8 до 20 КГц. При этом разрядность преобразования составляет 8-12 бит. Если в ходе аналого-цифрового преобразования речи используются частота дискретизации 8 КГц и разрядность преобразования 8 бит, то речевой сигнал длительностью 15 с будет занимать в памяти примерно 1 Мбайт. Манипулировать таким объемом данных при решении задачи распознавания звуков не эффективно. Поэтому после аналого-цифрового преобразования осуществляется предварительная обработка речевого сигнала с целью выделения существенных признаков, обеспечивающих распознавание звуков.



а) б)

Рисунок 9.23 – Вокализованный (а) и фрикативный (б) речевые сигналы и их спектры

Во многих случаях признаковое описание речевого сигнала получают на основе анализа кратковременного спектра Фурье, представленного в логарифмическом масштабе. На рисунке 9.23 изображены реализации вокализованного и фрикативного участков речи и их кратковременные спектры Фурье [6]. Структура кратковременного спектра вокализованных участков

речи характеризуются наличием медленно меняющейся и быстро меняющейся составляющих. Быстро меняющаяся или пульсирующая составляющая обусловлена квазипериодическими колебаниями голосовых связок. Медленно меняющаяся составляющая (огибающая) связана с собственными резонансными частотами голосового тракта – *формантами*. В среднем на графике спектра насчитывается 3-5 формант. Первые три форманты оказывают существенное влияние на разборчивость речи.

Быстро меняющаяся составляющая определяет индивидуальные параметры диктора. Чтобы снизить зависимость качества распознавания от диктора, выполняют сглаживание кратковременного спектра, подавляя быстро меняющуюся составляющую. Такое сглаживание может выполняться различными способами. Один из способов заключатся в разбиении частотной оси на 12-20 полос различной ширины с учетом слухового восприятия человека [6]. В каждой полосе вычисляют среднее значение спектральных составляющих. В итоге получают 12-20 элементный вектор спектральных признаков речевого сигнала. Следует отметить, что указанный вектор можно также получить с помощью банка цифровых фильтров. В этом случае речевой сигнал пропускают через блок из 12-20 полосовых фильтров и определяют на интервале 10-30 мс энергию выходного сигнала каждого из фильтров (рисунок 8.20)

Другие подходы, используемые при формировании вектора признаков, основаны на выполнении кепстрального или линейного предиктивного анализа. Кепстральный анализ выполняется путем вычисления обратного преобразования Фурье от логарифма кратковременного спектра. В этом случае необходимая лингвистическая информация сосредоточена в кепстральных коэффициентах, относящихся к области малых времен [6]. Из этих коэффициентов и формируется вектор признаков. Линейный предиктивный анализ выполняется на основе предсказания текущего отсчета речевого сигнала по линейной комбинации предыдущих отсчетов, т.е.

$$\widetilde{s}[n] = -\sum_{k=1}^{P} a[k]s[n-k],$$

где $\widetilde{s}[n]$ – предсказанное значение речевого сигнала; $s[n-k]$ – отсчёты речевого сигнала, $a[k]$ – коэффициенты линейного предсказания (ЛПК-коэффициенты); P – порядок модели предсказания [6,23]. ЛПК-коэффициенты отражают свойства голосового тракта и содержат существенную лингвистическую информацию. Однако они весьма чувствительны к шуму, присутствующему в речевом сигнале.

Во многих системах АРР при формировании вектора признаков учитывается изменение спектральных свойств речевого сигнала во време-

ни. В простейшем случае такая динамическая информация определяется на основе сравнения соседних кратковременных спектров. В настоящее время наиболее распространенной является предварительная обработка речи на перекрывающих интервалах длительностью 25 мс, в ходе которой формируется 39-элементный вектор, содержащий энергию сигнала, 12 спектральных коэффициентов, полученных с учетом свойств слухового восприятия человека, их первые и вторые производные [88].

На этапе распознавания звуков во многих АРР системах применяются алгоритмы, основанные на векторном квантовании. Если на сегменте (10-30 мс) речевого сигнала вычисляется n признаков, то вектор признаков можно представить точкой в n-мерном пространстве. Алгоритмы векторного квантования обеспечивают разбиение этого пространства на ограниченное количество областей, каждая из которых соответствует определенной группе звуков речи. Конкретный звук может быть задан указанием центра области или эталонным вектором. Такие области называют *кластерами.* В процессе распознавания вектор признаков речевого сигнала отображается в некоторый номер эталона. Иными словами, n-элементный вектор заменяется числом. Подобная схема распознавания выполняется, например, с помощью нейронной сети, изображенной на рисунке 8.20. Формирование эталонных векторов осуществляется в процессе обучения. Векторное квантование часто реализуют с помощью алгоритма K-средних (см. главу 8). Если вектор признаков является 20 элементным, и каждый из признаков представим целым числом, занимающим в памяти 2 байта, то при квантовании векторного пространства, например, на 256 областей (в этом случае номер эталона представляется одним байтом) происходит 40-кратное сжатие данных. Безусловно, часть информации входе векторного квантования теряется. Однако тщательное определение параметров эталонных векторов позволяет минимизировать эти потери. Таким образом, на выходе блока распознавания (рисунок 9.22) формируется последовательность номеров эталонных векторов, которая заменяет собой реальный речевой сигнал, поступающий на вход системы АРР.

В некоторых системах этап векторного квантования не выполняется. В этом случае при поиске последовательности слов, соответствующих фрагменту речевого сигнала, используют непосредственно векторы акустических признаков.

9.4.3. Статистический подход к распознаванию речи

Распознавание речи может рассматриваться как задача восстановления последовательности слов W по известному акустическому сигналу,

представленному номерами векторов эталонов X.[1)] Это классическая задача принятия решений в условиях неопределенности. Неопределенность здесь порождается акустическими шумами, изменением свойств входных каналов во времени, изменчивостью фонем и многими другими причинами.

Воспользуемся статистическим подходом к поиску решений в условиях неопределенности, основанном на правиле Байеса. Задача состоит в том, чтобы по совокупности номеров векторов эталонов X найти наиболее вероятную последовательность слов W*, обеспечивающую максимум апостериорной вероятности $P(W|X)$, т.е.

$$P(W^* \mid X) = \max_W P(W \mid X).$$

Модели, используемые в системах АРР, не позволяют непосредственно вычислять апостериорную вероятность $P(W|X)$. Однако они обеспечивают вычисление вероятности $P(X|W)$, которая представляет собой априорную вероятность формирования последовательности X при произнесении последовательности слов W. Апостериорная вероятность $P(W|X)$ может быть определена из правила Байеса

$$P(W \mid X) = \frac{P(W)P(X \mid W)}{P(X)} = \frac{P(W)P(X \mid W)}{\sum_{W_j} P(X \mid W_j)P(W_j)}, \qquad (9.6)$$

где $P(W)$ – априорная вероятность последовательности слов W; $P(X)$ – априорная вероятность последовательности X (номера векторов эталонов). В процессе распознавания $P(X)$ не меняет своих значений и может не приниматься во внимания при принятии решения в соответствии с (9.6). Следовательно, последовательность слов W*, максимизирующая $P(W|X)$, также максимизирует произведение $P(W)P(X|W)$. Безусловная вероятность последовательности слов $P(W)$ вычисляется на основе *модели языка*, которая позволяет отбирать в ходе поиска наиболее вероятные последовательности слов W. В простейшем случае распознавания изолированных слов ограниченного словаря $P(W)=1/V$, где V – количество слов словаря. Условная вероятность $P(X|W)$ вычисляется на основе *акустической модели*, обеспечивающей отбор более вероятных вариантов звуковой реализации последовательности слов W, представленной номерами векторов эталонов X. Таким образом, формально задача АРР может быть сформулирована в следующем виде:

$$W^* = \arg\max_W P(X \mid W)P(W). \qquad (9.7)$$

[1)] Случай, когда поиск выполняется непосредственно с помощью последовательности векторов признаков, рассмотрим позже.

Параметры языковой и акустической модели оцениваются по экспериментальным данным с определенной погрешностью. Так как эта погрешность может быть различной для указанных выше моделей, то вклад каждой из моделей в результат поиска последовательности *W** регулируется с помощью весового коэффициента β (β≤1) [80]:

$$W^* = \arg\max_W \{\log P(X \mid W) + \beta \log P(W)\}. \qquad (9.8)$$

Выражение (9.8) может рассматриваться в качестве критерия принятия решения в условиях двух экспертных оценок.

9.4.4. Модель языка

Модель языка (МЯ) обеспечивает формирование последовательностей слов, которые потенциально могут быть распознаны системой АРР. В частности, она позволяет оценить вероятность появления слова w_i в окружении слов из последовательности *W*. МЯ интегрирует в себе лингвистические знания, знания о предметной области и другую информацию с целью сокращения пространства поиска.

Так как вероятность появления слова зависит от того ранее произнесенных слов, то вероятность последовательности слов $w_1 w_2 w_3 ... w_M$ можно представить в виде

$$P(w_1 ... w_M) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_1 w_2)\ldots P(w_M \mid w_1 \ldots w_{M-1}). \qquad (9.9)$$

Например, совместная вероятность появления последовательности слов *“*книга лежит на столе*”* определится по формуле:

$$P(\textit{книга лежит на столе}) = P(\textit{книга})\ \mathrm{P}(\textit{лежит} \mid \textit{книга}) \times \times P(\textit{на} \mid \textit{книга лежит}) \times \times P(\textit{столе} \mid \textit{книга лежит на}).$$

Перепишем (9.9) в форме:

$$P(w_1 \ldots w_M) = P(W_1^M) = \prod_{i=1}^{M} P(w_i \mid w_1, w_2, \ldots w_{i-2}, w_{i-1}). \qquad (9.10)$$

Выражение (9.10) позволяет ввести понятие *N-грамной* модели, в соответствии с которой вероятность появления некоторого слова зависит только от появления N предшествующих слов:

$$P(w_i \mid W_1^{i-1}) = P(w_i \mid W_{i-N+1}^{i-1}).$$

Определение условных вероятностей $P(w_i \mid W_{i-N+1}^{i-1})$ при больших значениях N требует значительных вычислительных ресурсов. Поэтому на практике рассматривают вероятность появления слова w_i при ограниченной длине предшествующего контекста. Наиболее часто используется *биграмная модель* (N=2). В этом случае условная вероятность $P(w_i \mid W_{i-N+1}^{i-1})$ аппроксимируется вероятностью $P(w_i \mid w_{i-1})$. Иными словами, вероятность появления слова w_i определяется только предыдущим словом w_{i-1} в последовательности W. Тогда совместная вероятность появления последовательности слов W определится из формулы

$$P(w_1 \ldots w_n) = P(w_1)P(w_2 \mid w_1)P(w_3 \mid w_2)\ldots P(w_n \mid w_{n-1}) = \prod_{i=1}^{n} P(w_i \mid w_{i-1}).$$

Например,

$$P(\textit{книга лежит на столе}) = P(\textit{книга})P(\textit{лежит}|\textit{книга}) \times \\ \times P(\textit{на}|\textit{лежит})P(\textit{столе}|\textit{на}).$$

Существенным преимуществом биграмной модели является простота оценивания ее параметров. Оценка $P(w_i \mid w_{i-1})$ получается путем подсчета частоты появления слова w_i, если предыдущим было слово w_{i-1}. Например, если в совокупности обучающих данных слово “на” встретилось 1000 раз, а в сочетании “лежит на” 29 раз, то

$$\hat{P}\ (\textit{на}|\textit{лежит}) = 29/1000 = 0{,}029.$$

Если обучающие данные не полные, то допустимым последовательностям слов могут быть назначены нулевые вероятности.

Кроме биграмной модели, часто применяется *триграмная* языковая модель, основанная на использовании условных вероятностей вида $P(w_i \mid w_{i-1}, w_{i-2})$. Эта модель предоставляет больше возможностей. В частности, с ее помощью можно непосредственно установить, что последователь-

ность слов “он ест торт” более вероятна, чем “он ест порт”. Оценка условных вероятностей триграмных моделей выполняется с помощью формулы:

$$\hat{P}(w_i \mid w_{i-1}, w_{i-2}) = \frac{C(i-2,i-1,i)}{C(i-2,i-1)}, \qquad (9.11)$$

где $C(i-2,i-1,i)$ и $C(i-2,i-1)$ – количество случаев, когда наблюдались последовательности слов $\{w_{i-2}, w_{i-1,} w_i\}$ и $\{w_{i-2}, w_{i-1}\}$, соответственно.

Применение триграмных моделей требует значительных объемов данных для корректного оценивания соответствующих вероятностей. Для словаря размером V существует V^3 возможных триграм. Например, при словаре 20000 слов потенциально возможно 8 млрд. триграм. Однако реальные обучающие данные могут содержать не все триграммы. С учетом формулы (9.11) триграмам, которые не встретились в обучающем наборе данных, будут назначены нулевые оценки вероятностей появления. Чтобы избежать этого, применяют интерполяционную оценку вероятности, комбинирующую триграмную, биграмную и униграмную оценки

$$P = \lambda_3 \hat{P}(w_i \mid w_{i-1}, w_{1-2}) + \lambda_2 \hat{P}(w_i \mid w_{i-1}) + \lambda_1 \hat{P}(w_i) + \lambda_o \frac{1}{V},$$

где $\lambda_3, \lambda_2, \lambda_1, \lambda_0$ – коэффициенты интерполяции.

Биграммные и триграммные модели полностью игнорируют лингвистическую структуру предложений. В частности, они не позволяют учесть “дальние” синтаксические отношения, существующие между словами предложения. Другим недостатком *N*-грамных моделей является чувствительность к подмножеству языка, представленного обучающими наборами данных. Поэтому *N*-грамные модели, построенные для одной области приложений, плохо переносятся на другую область. Это приводит к необходимости индивидуального отбора обучающих данных для каждой новой прикладной области.

Вместе с тем, благодаря своей простоте, рассмотренные модели применяются во многих АРР-системах.

9.4.5. Акустическая модель

Акустическая модель (АМ) позволяет установить, какие звуки соответствуют заданной последовательности слов. АМ можно представить в виде двух частей. Первая часть модели описывает слово в виде последовательности звуков – транскрипции. Вторая часть модели показывает, как

каждый из звуков соотносится с последовательностью векторов эталонов, получаемых в результате векторного квантования.

Некоторые из слов характеризуются простой моделью произношения. Например, произношение слова “вал” совпадает с его написанием. Многие слова имеют различные варианты произношения, которые объясняются особенностями диалекта языка, стилем произношения (полный и неполный), коартикуляционными эффектами и другими причинами. Например, слово “молочная” можно произнести несколькими способами:[молочнаja], [молошнаja], [малошнаja], [малочнаja].

Различные варианты произношения слова можно представить в виде модели (цепи) Маркова. В общем случае цепь Маркова позволяет описывать процессы, последовательно проходящие через конечное множество состояний.

Марковская цепь представляется в виде графа, узлами которого являются состояния, а дуги соответствуют возможным переходам состояний. С каждым узлом цепи Маркова связывают определенный символ, а с каждой дугой – вероятность смены состояния. Вероятность смены одного состояния на другое зависит только от предыдущего состояния.

На рисунке 9.24 изображены различные варианты произношения слова “молочная” в виде цепи Маркова. Каждому состоянию цепи соответствует определенный звук. Направленные дуги показывают возможные переходы состояний. Числа, записанные рядом с дугами, представляют вероятности соответствующих переходов. Единичные значения вероятностей на рисунке не показаны.

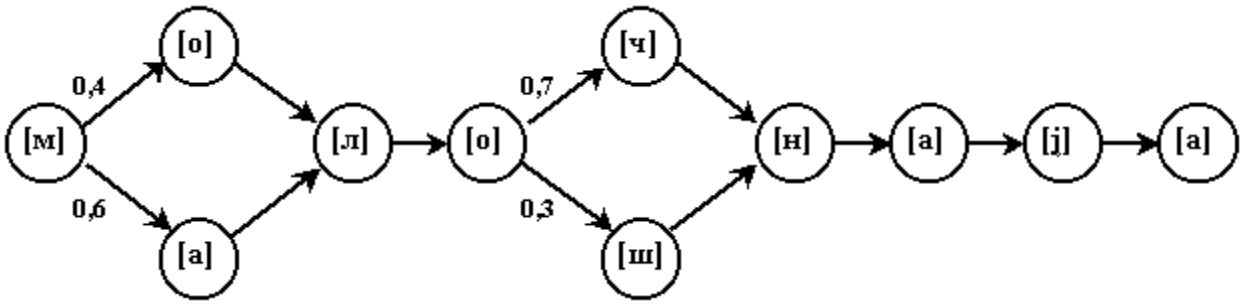


Рисунок 9.24 – Модель произношения слова “молочная”

Вероятность перехода из начального состояния в конечное состояние вычисляется как произведение вероятностей переходов вдоль соответствующего пути. В рассматриваемом случае имеется четыре пути, ведущих из начального состояния в конечное. Следовательно, различным вариантам произношения слова “молочная” можно назначить следующие вероятности:

P([молочнаja]|“молочная”)=0,28;
P([молошнаja]|“молочная”)=0,12;
P([малочнаja]|“молочная”)=0,42;
P([малошнаja]|“молочная”)=0,18.

Подобные модели могут быть построены для каждого слова, которое распознается системой АРР.

Если бы речевой сигнал был эквивалентен последовательности фонем, задаваемых транскрипцией, то, используя цепи Маркова, можно было бы вычислить вероятность *P*(*сигнал*|*слова*) (т.е.*P(X*|*W*)) для различных последовательностей слов *W*. Комбинируя полученную вероятность с априорной вероятностью *P(W*), вычисляемой на основе МЯ, можно было бы найти последовательность слов *W*,* максимизирующую вероятность *P*(*слова*|*сигнал*), т.е. *P(W*|*X*). К сожалению, в ходе предварительной обработки речевых сигналов и векторного квантования не формируется последовательность фонем. Поэтому в рассматриваемом случае обеспечивается лишь нахождение максимального значения вероятности *P*(*слова*|*фонемы*). Отсюда следует, что для решения поставленной задачи необходимо дополнительно определять вероятность *P*(*сигнал*|*фонемы*). С этой целью применяют *скрытые марковские модели* (СММ).

СММ описывается с помощью двух вероятностных процессов: скрытого процесса смены состояний цепи Маркова и наблюдаемого процесса формирования выходных символов при смене состояний. Пусть имеется *N* состояний. Вероятность того, что в момент времени *t* текущим будет состояние S_i, зависит от предыстории смены состояний. В моделях Маркова первого порядка полагается, что данная вероятность зависит только от состояния, в котором находился скрытый процесс в предыдущий момент времени *t*-1. Именно такие модели используются чаще всего. Наблюдаемый процесс формирования выходных символов описывается с помощью распределения вероятностей $b_i(x)$, которое показывает вероятность формирования выходного символа x при переходе в состояние S_i. На рисунке 9.25 изображен пример СММ. Здесь в состояниях S_1, S_2, S_3 формируется один из выходных символов $c_1, c_2, ..., c_M$ в соответствии с распределением вероятностей $b_i(x)$, где $x = c_j$. Состояния S_0 и S_4 называются входным и конечным состояниями. В этих состояниях не формируются выходные символы.

Формально СММ, имеющая *N* состояний, характеризуется:

- текущим состоянием q_t в момент времени *t*;

- матрицей переходных вероятностей $A=[a_{ij}]$, где $1 \le i, j < N$ и $a_{ij} = P(q_{t+1} = S_j \mid q_t = S_i)$ – вероятность перехода из состояния S_i в состояние S_j;
- матрицей распределения выходных вероятностей $B=[b_{ij}]$, где $b_{ij} = b_i(c_j) = P(x_t = c_j \mid q_t = S_i)$ – вероятность того, что в момент времени *t* выходной символ x_t примет значение c_j, если текущим является состояние S_i, $\sum_{j=1}^{M} b_i(c_j) = 1$;
- вектором распределения начальных состояний $\Pi=[\pi_i]$, где $\pi_i = P(q_1 = S_i)$ – вероятность того, что в начальный момент времени *t*=1 текущим будет состояние S_i.

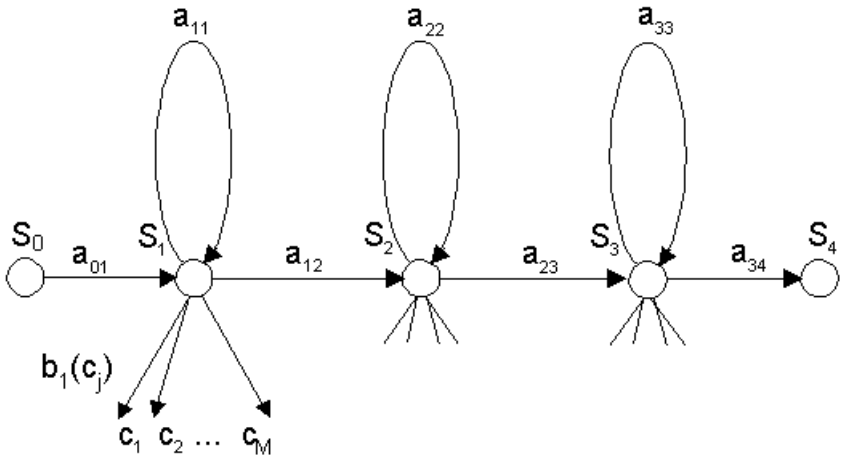


Рисунок 9.25 – Пример СММ

Вероятности начальных состояний могут быть включены в матрицу *A*. В этом случае они представляют собой вероятность смены входного состояния на одно из состояний, которое обеспечивает формирование выходного символа, т.е. $\pi_i = a_{0i}$. Тогда СММ представляет собой структуру

$$\lambda = (A, B),$$

характеризуемую параметрами *A* и *B*. Эти параметры определяют вероятность формирования последовательности символов *X* с помощью модели λ, т.е. $P(X|\lambda)$.

Применение СММ при распознавании речи сопряжено с решением трех основных подзадач:

- вычислительной подзадачи;

- подзадачи восстановления (декодирования);
- подзадачи оценивания.

Вычислительная подзадача состоит в следующем. Пусть заданы модель $\lambda = (A, B)$ и последовательность выходных символов $X = x_1, x_2, ..., x_T$. Требуется построить алгоритм, вычисляющий вероятности формирования последовательности *X* моделью λ, т.е. $P(X|\lambda)$.

Подзадача восстановления (декодирования) состоит в поиске оптимальной последовательности состояний $Q = q_1, q_2, ..., q_T$ по известной последовательности $X = x_1, x_2, ..., x_T$ для заданной модели λ. Иными словами, требуется выяснить, какая из возможных последовательностей состояний *Q* в наибольшей степени соответствует, наблюдаемой последовательности символов *X*.

Целью *подзадачи оценивания* является определение параметров модели $\lambda = (A, B)$, обеспечивающих максимум вероятности $P(X|\lambda)$.

Рассмотрим вычислительную подзадачу. Чтобы вычислить вероятности $P(X|\lambda)$, необходимо рассмотреть возможные последовательности состояний $Q = q_1, q_2, ..., q_T$, заданной длины *T*, обеспечивающие формирование последовательности $X = x_1, x_2, ..., x_T$. Если СММ имеет *N* состояний, то общее число таких последовательностей будет равно N^T. Так как все последовательности являются взаимно исключающими, то вероятность $P(X|\lambda)$ будет равна сумме совместных вероятностей $P(X,Q|\lambda)$:

$$P(X|\lambda) = \sum_{\forall Q} P(X,Q|\lambda). \qquad (9.12)$$

Последовательность состояний *Q* можно рассматривать как путь, при прохождении которого формируется последовательность *X*. Тогда $P(X,Q|\lambda)$ будет представлять собой вероятность формирования последовательности *X* на пути *Q* моделью λ. Эта вероятность вычисляется как произведение вероятностей переходов a_{ij} и выходных вероятностей $b_i(x_j) = b_{ij}$ вдоль пути *Q,* т.е.

$$P(X, Q \mid \lambda) = a_{0,q1} b_{q1}(x_1) a_{q1,q2} b_{q2}(x_2) ... a_{qT-1,qT} b_{qT}(x_T) a_{qT,N},$$

где $a_{0,q1}$ и $a_{qT,N}$ – соответственно вероятности смены входного состояния на состояние q_1, формирующее выходной символ x_1, и состояния q_T на конечное состояние.

Рассмотрим вычисление вероятности $P(X|\lambda)$ на примере. На рисунке 9.26 изображена типовая СММ фонемы с тремя состояниями: начальным

(q_1), средним (q_2) и конечным (q_3). Кроме переходных вероятностей на рисунке показаны распределения вероятностей выходных символов, ассоциированных с состояниями q_1, q_2 и q_3. Здесь выходные символы соответствуют номерам эталонных векторов, формируемым в процессе векторного квантования. Предположим, что в результате векторного квантования формируется последовательность $X = x_1 x_2 x_3 x_4 x_5 x_6$, состоящая из номеров эталонных векторов $[c_1 c_1 c_3 c_3 c_6 c_6]$.

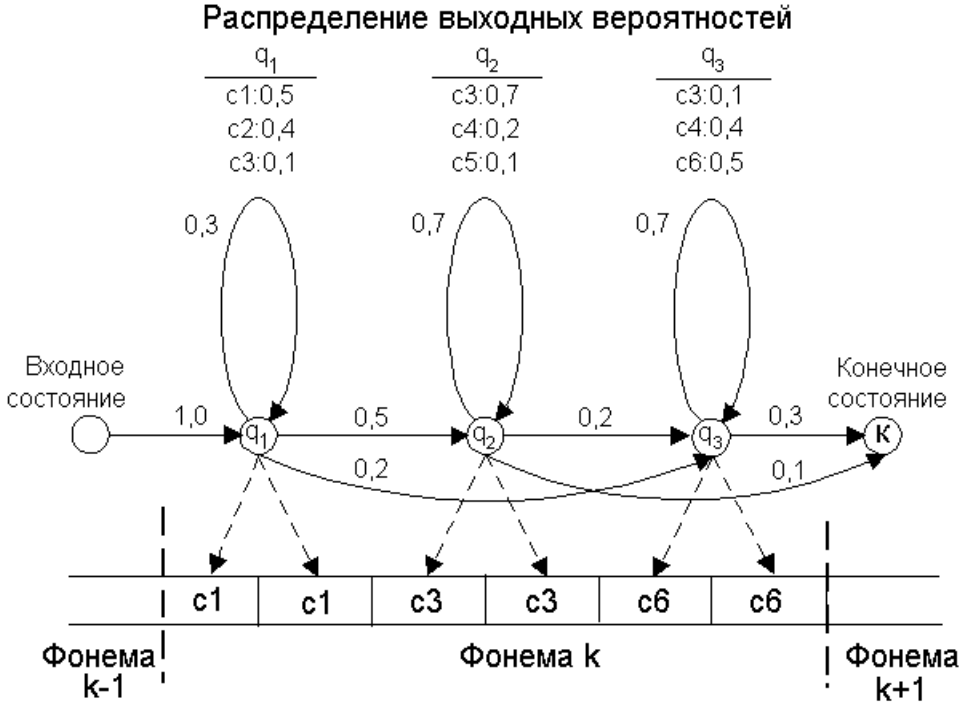


Рисунок 9.26– СММ фонемы

В модели фонемы имеются пять путей смены состояний, обеспечивающие формирование последовательности $[c_1, c_1, c_3, c_3, c_6, c_6]$. Суммарная вероятность формирования последовательности моделью фонемы в соответствии с (9.12) равна:

$$
\begin{aligned}
&P(X \mid \textit{фонема}) = P([c_1, c_1, c_3, c_3, c_6, c_6] \mid \textit{фонема}) = \\
&= (0{,}3 \cdot 0{,}5 \cdot 0{,}7 \cdot 0{,}2 \cdot 0{,}7 \cdot 0{,}3)(0{,}5 \cdot 0{,}5 \cdot 0{,}7 \cdot 0{,}7 \cdot 0{,}5 \cdot 0{,}5) + \\
&+ (0{,}3 \cdot 0{,}3 \cdot 0{,}5 \cdot 0{,}2 \cdot 0{,}7 \cdot 0{,}3)(0{,}5 \cdot 0{,}5 \cdot 0{,}1 \cdot 0{,}7 \cdot 0{,}5 \cdot 0{,}5) + \\
&+ (0{,}3 \cdot 0{,}5 \cdot 0{,}2 \cdot 0{,}7 \cdot 0{,}7 \cdot 0{,}3)(0{,}5 \cdot 0{,}5 \cdot 0{,}7 \cdot 0{,}1 \cdot 0{,}5 \cdot 0{,}5) + \\
&+ (0{,}3 \cdot 0{,}3 \cdot 0{,}3 \cdot 0{,}2 \cdot 0{,}7 \cdot 0{,}3)(0{,}5 \cdot 0{,}5 \cdot 0{,}1 \cdot 0{,}1 \cdot 0{,}5 \cdot 0{,}5) + \\
&+ (0{,}3 \cdot 0{,}2 \cdot 0{,}7 \cdot 0{,}7 \cdot 0{,}7 \cdot 0{,}3)(0{,}5 \cdot 0{,}5 \cdot 0{,}1 \cdot 0{,}1 \cdot 0{,}5 \cdot 0{,}5) = \\
&= 1{,}6499 \cdot 10^{-4}.
\end{aligned}
$$

Циклы вокруг состояний на рисунке 9.26 позволяют моделировать изменение продолжительности звуков. Переход из состояния q_1 в состояние q_3 позволяет моделировать звуки короткой продолжительности (20 – 40 мс).

9.4.6. Композиция моделей

Выше были введены три модели:

- модель языка, представляемая биграмами или триграмами и характеризуемая вероятностями $P(w_i \mid w_{i-1})$ и $P(w_i \mid w_{i-1}, w_{i-2})$;
- модель произношения слова (лексическая модель), которая обычно представляется в виде цепи Маркова и характеризуется вероятностью $P(\textit{фонемы} \mid \textit{слово})$;
- модель фонемы, которая соответствует СММ и позволяет вычислить вероятность $P(X \mid \textit{фонема})$, X – последовательность номеров векторов эталонов.

Так как для решения задачи распознавания речи, необходимо вычислять вероятность $P(W \mid X)$, т.е. по заданному речевому сигналу, представленному последовательностью X, восстанавливать последовательность слов W, соответствующую этому сигналу, то необходимо построить составную модель, объединяющую три указанных модели. С этой целью выполняют композицию рассмотренных моделей в одну большую СММ.

Например, биграмная модель может рассматриваться как модель Маркова, в которой каждому состоянию соответствует слово, а слова связаны между собой дугами с вероятностями переходов $P(w_i \mid w_{i-1})$. На рисунке 9.27 изображена биграмная МЯ, представляющая собой сеть слов. Самый левый узел сети соответствует начальному состоянию, а самый правый конечному.

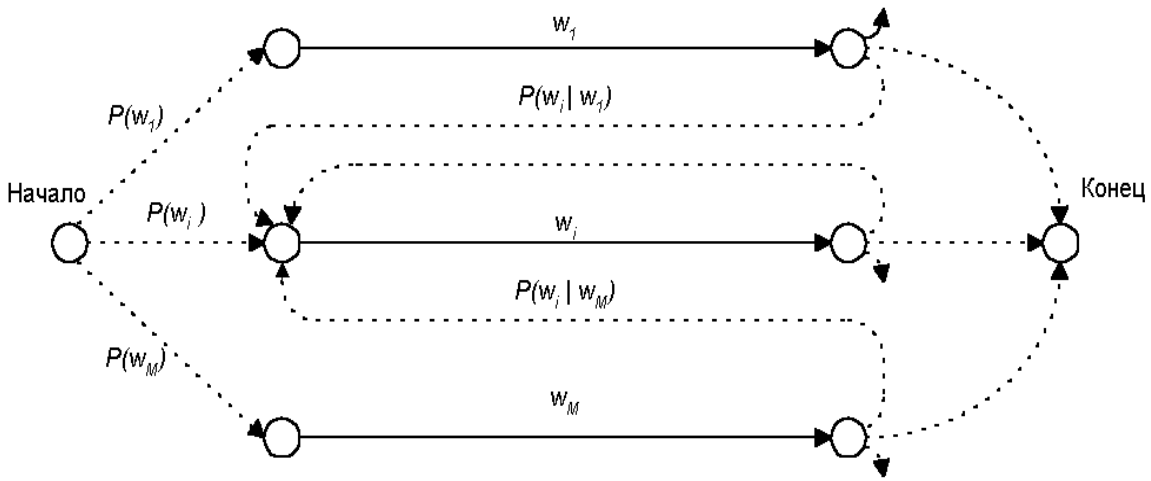


Рисунок 9.27 – Биграмная МЯ

Если каждое слово-состояние биграмной МЯ заменить соответствующей моделью произношения, то можно получить СММ, в которой состояния уже будут соответствовать фонемам. И наконец, если каждую фонему представить в виде модели, соответствующей рисунку 9.26, то можно получить одну большую СММ, в которой каждому состоянию соответствует определенное распределение значений векторов эталонов речевого сигнала. Процесс построения составной модели условно изображен на рисунке 9.28.

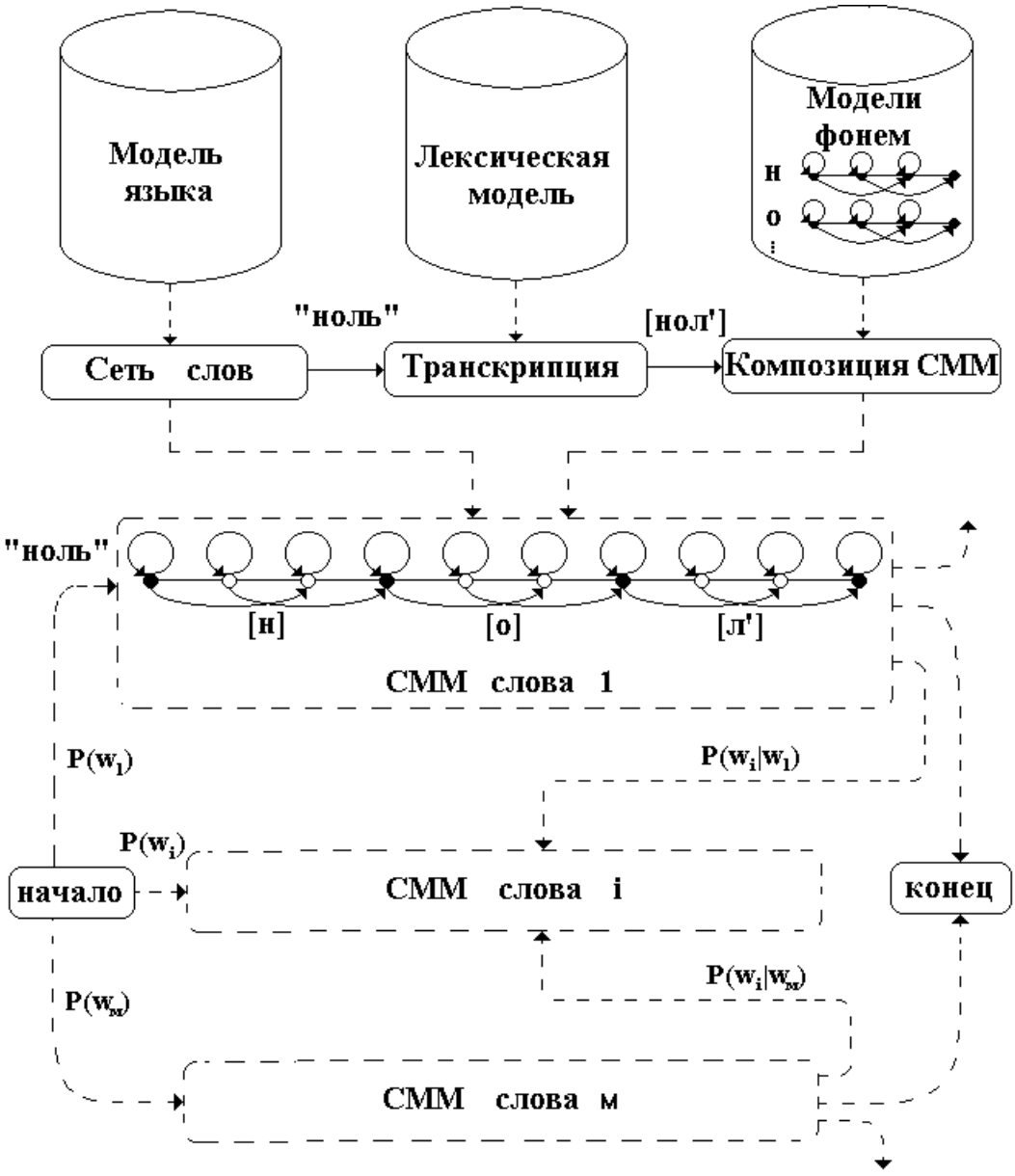


Рисунок 9.28 – Композиция моделей

Используя составную модель, можно приступить к поиску последовательности слов *W**, максимизирующей вероятность $P(W \mid X)$. Для этого необходимо перебирать возможные последовательности слов-состояний W и вычислять для каждой последовательности соответствующие вероятности $P(X, W \mid \lambda)$. Непосредственный поиск, основанный на просмотре всех вариантов последовательностей слов, не эффективен. Более того, при больших размерах словаря, он практически не осуществим. Поэтому в системах АРР применяют специальные методы поиска последовательности *W**.

9.4.7. Алгоритмы поиска

При решении задачи АРР применяют несколько алгоритмов поиска [72,80]:

- алгоритм Витерби;
- A^*-алгоритм;
- алгоритм быстрого сопоставления;
- алгоритм двунаправленного поиска и др.

Рассмотрим алгоритм Витерби. Введем в рассмотрение прямую вероятность

$$\alpha_t(j) = P(X_1^t, q_t = S_j \mid \lambda), \tag{9.13}$$

которая представляет совместную вероятность того, что к моменту времени t модель λ будет находиться в состоянии S_j, и на выходе будет сформирована последовательность $X_1^t = x_1 x_2 \ldots x_t$.

Принимая во внимание свойства СММ, $\alpha_t(j)$ можно определить следующим образом

$$\alpha_t(j) = \sum_{j=1}^{N-1} P(x_t \mid q_t = S_j) P(q_t = S_j \mid q_{t-1} = S_i) P(X_1^{t-1}, q_{t-1} = S_i \mid \lambda).$$

Так как $0 < j < N$ и $1 < t \le T$, то $\alpha_t(j)$ можно представить в виде рекуррентного выражения

$$\alpha_t(j) = \left[\sum_{i=1}^{N-1} \alpha_{t-1}(i) a_{ij} \right] \cdot b_j(x_t), \quad 0 < j < N,\ 1 < t \le T \tag{9.14}$$

с начальными условиями $\alpha_1(0) = 1$ и $\alpha_1(j) = a_{0j} b_j(x_1)$, $0 < j < N$. С учетом конечного состояния S_N вероятность формирования всей последовательности X моделью λ определяется из соотношения

$$P(X \mid \lambda) = \sum_{i=1}^{N-1} \alpha_T(i) a_{iN} \ . \qquad (9.15)$$

Соотношения (9.14) и (9.15) позволяют вычислить полную вероятность формирования последовательности X в модели λ с учетом всех возможных путей (направлений смены состояний). По сравнению с (9.12) вычисление полной вероятности $P(X|\lambda)$ по формуле (9.15) требует анализа лишь N^2T вариантов для СММ, в которых из каждого состояния разрешено N переходов (каждое состояние связано с каждым).

Алгоритм Витерби обеспечивает нахождение в модели λ оптимального пути (последовательности состояний), в наибольшей степени соответствующего заданной последовательности X. Для этого на каждом шаге алгоритма запоминаются состояния, которые обеспечивают максимум совместной вероятности $P(X_1^t, Q_1^t \mid \lambda)$.. Алгоритм Витерби решает задачу восстановления последовательности состояний Q по наблюдаемой последовательности символов X.

Обозначим

$$\delta_t(j) = \max_{\forall Q_1^t (q_t = S_j)} P(X_1^t, Q_1^t \mid \lambda) ,$$

где $\delta_t(j)$ – вероятность формирования последовательности X_1^t при движении вдоль оптимального пути, который в момент времени t завершается состоянием $q_t = S_j$. По аналогии с прямой вероятностью, определяемой рекуррентным соотношением (9.14), перепишем $\delta_t(j)$ в виде [72]:

$$\delta_t(j) = \max_i [\delta_{t-1}(i) a_{ij}] b_j(x_t) .$$

Данное выражение лежит в основе алгоритма Витерби. Для того чтобы восстановить оптимальный путь (оптимальную последовательность состояний), на каждом шаге алгоритма Витерби сохраняются обратные указатели для соответствующих оптимальных состояний S_j. Обозначим массив таких указателей через $\psi_t(j)$. Тогда алгоритм Витерби можно описать следующим образом [72]:

1) инициализация

$$\delta_1(j) = a_{0j} b_j(x_1), \qquad 0 < j < N,$$
$$\psi_1(j) = 0;$$

2) итерации

$$\delta_t(j) = \max_{0<t<N}[\delta_{t-1}(i)a_{ij}]b_j(x_t), \tag{9.16}$$

$$\psi_t(j) = \arg\max_{0<t<N}[\delta_{t-1}(i)a_{ij}], \tag{9.17}$$

$$1< t < T,\ 0 < j < N;$$

3) учет конечного состояния N

$$\delta_t(N) = \max_{0<t<N}[\delta_T(i)a_{iN}],$$

$$\psi_t(N) = \arg\max_{0<t<N}[\delta_T(i)a_{iN}];$$

4) построение обратного пути

$$q_T^* = \psi_T(N),$$

$$q_t^* = \psi_{t+1}(q_{t+1}^*), \qquad t = T-1, T-2, \ldots 1,$$

где q_t^* – состояние СММ, принадлежащее оптимальному пути.

Алгоритм Витерби представляет собой разновидность алгоритма динамического программирования. Наиболее ранние применения динамического программирования к задаче АРР были выполнены в работах Т.Г. Винцюка.

Рассмотрим реализацию алгоритма Витерби применительно к СММ, изображенной на рисунке 9.26. Пусть известна последовательность $X = x_1x_2x_3x_4$, где $x_1 = c_1$, $x_2 = c_3$, $x_3 = c_4$, $x_4 = c_6$ – номера эталонных векторов, получаемые на выходе векторного квантователя. Требуется найти последовательность состояний, которая с наибольшей вероятностью могла бы сформировать заданную последовательность *X*.

Процесс вычислений изображен на рисунке 9.29 в виде характерной решетчатой структуры. Здесь каждый столбец решетки соответствует одной итерации. В самом левом столбце показано только одно состояние q_1, переход в которое выполняется с вероятностью 1,0. Данное состояние обеспечивает формирование символа $[c_1]$ с вероятностью 0,5 (см. рисунок 9.26). Поэтому путь, оканчивающийся состоянием q_1, будет иметь вероятность $\delta_1(1) = 1{,}0 \times 0{,}5 = 0{,}5,$ которая на рисунке 9.29 записана под состоянием q_1. Во втором столбце изображены возможные продолжения путей из первого столбца, позволяющие сформировать последовательность $[c_1, c_3]$. Имеются три таких пути, оканчивающиеся состояниями q_1, q_2, q_3. Вероятности каждого из этих путей вычисляются с помощью выражения (9.16). Продолжения различных путей второго столбца могут приводить к одним и тем же состояниям, изображенным в третьем столбце. Например, в состояние q_2 ведут два пути, а в состояние q_3 три пути. Состояниям q_2 и q_3 столбца три будет предшествовать состояние q_2 столбца два, так как пере-

ходы из этого состояния обеспечивают максимум вероятности для путей, оканчивающихся состояниями q_2 и q_3.

В соответствии с (9.17) предшествующие состояния запоминаются в массиве $\psi_t(j)$. В рассматриваемой ситуации $\psi_3(2) = 2, \psi_3(3) = 2$. Продолжая указанный процесс раскрытия состояний, достигаем конечного состояния с вероятностью 0,00147. Анализируя значение $\psi_T(j)$, получаем последовательность состояний $Q = q_1 q_2 q_3 q_3$, которая с наибольшей вероятностью формирует заданную последовательность $[c_1 c_3 c_4 c_6]$. На рисунке 9.29 оптимальный путь изображен жирной линией.

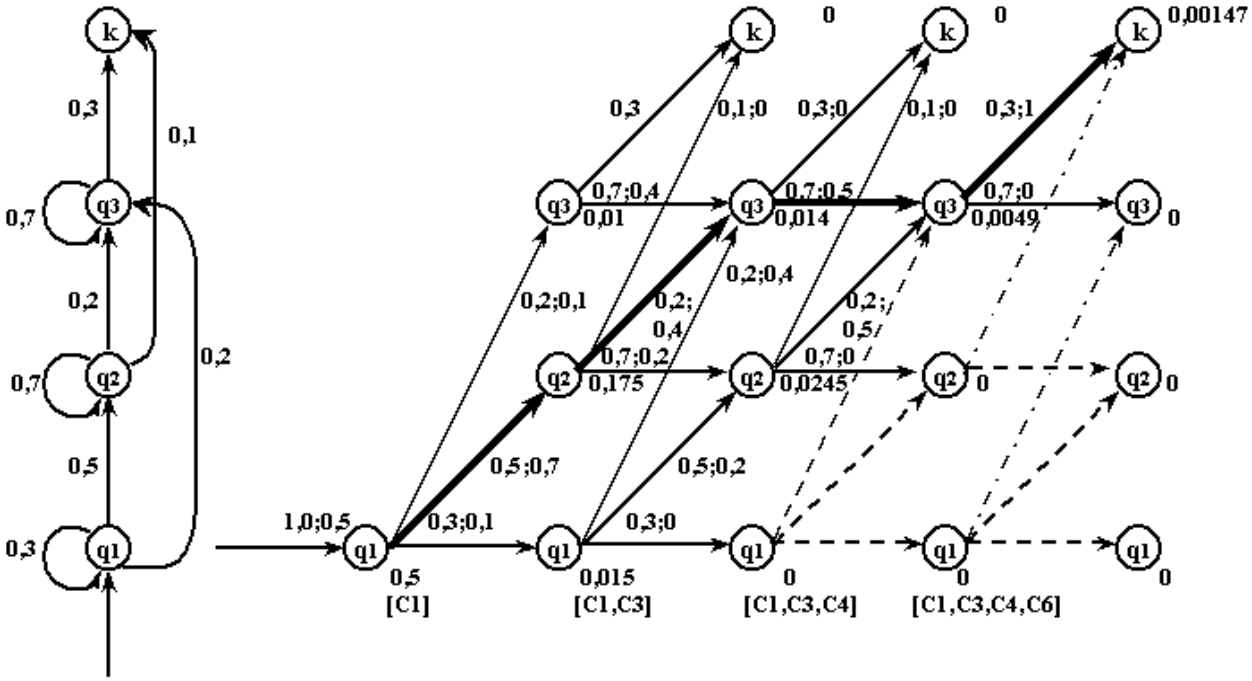


Рисунок 9.29 – Пример реализации алгоритма Витерби

Так как рассматриваемые в системах АРР модели допускают переходы состояний только в направлении слева направо, то общее число вариантов, анализируемых в алгоритме Витерби равно *NT*, что намного меньше по сравнению с N^T, когда учитываются все пути, обеспечивающие формирование заданной последовательности *X* (см. выражение 9.12).

Сложность составной модели возрастает по мере увеличения размера словаря. Для больших словарей размеры поисковых пространств оказываются огромными. Поэтому были разработаны различные модификации алгоритма Витерби, сокращающие поисковые пространства. Один из таких алгоритмов соответствует *лучевому поиску* (см. § 2.2.2). В этом случае на каждом шаге поиска из рассмотрения исключаются состояния, для которых вероятность $P(X_1^t, Q_1^t)$ ниже оптимального значения минус некоторый порог.

Значение порога определяется эмпирически и адаптируется в процессе поиска.

Если размер словаря составляет десятки тысяч слов, то сложность поиска остается высокой. В этой ситуации применяются иные алгоритмы поиска: A^*- алгоритм, алгоритм двунаправленного поиска, алгоритм быстрого сопоставления и др.

При использовании A^*- алгоритма (см. § 2.2.2) на каждом шаге продолжается путь, характеризуемый наибольшей величиной оценочной функции, которая в данном случае представляет собой сумму правдоподобия (логарифм от вероятности) уже пройденного пути (т.е. Q_1^T) и оценки правдоподобия оставшейся части пути (т.е. Q_{t+1}^T) до конечного состояния. Последняя оценка соответствует эвристической составляющей оценочной функции. Если эвристическая оценка правдоподобия оставшейся части пути будет больше или равна реальному значению правдоподобия, то A^*- алгоритм обеспечит поиск оптимального пути. Один из простых способов определения эвристической составляющей основан на ее оценке по оставшейся части исходной последовательности акустических элементов X_{t+1}^T. С этой целью накапливается статистика соответствия X_{t+1}^T последовательности Q_{t+1}^T. Имеются и другие способы определения эвристической составляющей оценочной функции.

9.4.8. Оценивание параметров СММ

Рассмотрим подзадачу оценивания параметров СММ $\lambda = (A, B)$. Данную подзадачу рассматривают как обучение СММ на множестве обучающих данных, представленных парами “речевой сигнал – последовательность слов”. В ходе обучения необходимо определить по заданной последовательности обучающих данных значения матриц A и B таким образом, чтобы обеспечивался максимум вероятности $P(X \mid \lambda)$.

Обучение СММ осуществляется на основе алгоритма Баума-Уэлча [72,74]. Приведем необходимые формулы оценивания параметров СММ в соответствии с алгоритмом Баума-Уэлча без вывода. Обозначим:

$$\gamma_t(i) = \begin{cases} 1, & \textit{если} \quad q_t(i), \\ 0, & \textit{в} \quad \textit{остальных} \quad \textit{случаях}; \end{cases}$$

$$\xi_t(i,j) = \begin{cases} 1, & \textit{если} \quad q_t(i) \quad \textit{и} \quad q_{t+1}(j) \\ 0, & \textit{в} \quad \textit{остальных} \quad \textit{случаях}; \end{cases}$$

$N(i,j) = \sum_{t=1}^{T-1} \xi_t(i,j)$ – количество переходов из состояния *S(i)* в состояние *S(j)*, которое обеспечивает формирование выходного символа;

$K(i) = \sum_{t=1}^{T} \gamma_t(i)$ – количество переходов из состояния *S(i)*.

Тогда при $0 < j, i < N$ элементы матриц $\overline{A}$ и $\overline{B}$ вычисляются по формулам:

$$\overline{a}_{0j} = \gamma_1(j);$$
$$\overline{a}_{ij} = N(i,j)/K(i);$$
$$\overline{a}_{iN} = \gamma_T(i)/K(i);$$
$$\overline{b}_{ij} = \overline{b}_i(c_j) = \sum_{t=1, x_t = c_j}^{T} \gamma_t(i)/K(i), \quad 1 \le j \le M.$$

Приведенные выражения позволяют по заданной модели $\lambda = (A,B)$ построить новую модель $\overline{\lambda} = (\overline{A},\overline{B})$. При этом $P(X \mid \overline{\lambda}) > P(X \mid \lambda)$.

В алгоритме Баума-Уэлча на каждом шаге итерации модель λ замещается моделью $\overline{\lambda}$. Это обеспечивает монотонное увеличение $P(X \mid \lambda)$. Процесс завершается, когда вероятность $P(X \mid \lambda)$ больше не увеличивается, т.е. $P(X \mid \overline{\lambda}) \approx P(X \mid \lambda)$. К сожалению, алгоритм Баума-Уэлча не гарантирует нахождение глобальных экстремумов.

9.4.9. СММ с непрерывными параметрами

До сих пор рассматривался случай дискретных СММ, в которых наблюдаемый процесс соответствует символам фиксированного алфавита $C = \{c_1, c_2, ..., c_M\}$. В этом случае распределение вероятностей выходных символов *i*-го состояния задается вероятностями появления отдельных символов $b_i(c_j)$, где $i \le j \le M$. Это требует квантования непрерывных значений векторов-признаков речевого сигнала. Данный процесс реализуется с помощью векторного квантователя. С одной стороны, это обеспечивает эффективное представление речевого сигнала, а с другой, приводит к неизбежным погрешностям квантования, которые могут ухудшать распознавание речи.

Для того чтобы исключить влияние векторного квантования, применяют СММ, в которых наблюдаемый процесс описывается непрерывными

функциями плотности распределения. Наиболее часто функцию плотности распределения представляют в виде суммы распределений гауссовского типа:

$$b_i(\vec{x}) = \sum_{j=1}^{M} c_{ij} b_{ij}(\vec{x}) = \sum_{j=1}^{M} c_{ij} N(\vec{x}, \vec{\mu}_{ij}, \delta_{ij}), \qquad (9.18)$$

где $\vec{x}$ – вектор признаков речевого сигнала;

i, j – индексы, соответственно, обозначающие номера состояний и номера классов, к которым могут относиться векторы признаков;

c_{ij} – весовой коэффициент;

$b_{ij}(\vec{x})$ – распределение векторов-признаков для *j*-го класса, приписанное *i*-му состоянию;

$N(\vec{x}, \vec{\mu}_{ij}, \delta_{ij})$ – гауссова плотность распределения, характеризуемая вектором средних значений μ_{ij} и ковариационной матрицей δ_{ij}.

Плотность распределения (9.18) достаточно хорошо аппроксимирует различные распределения векторов признаков, встречающиеся на практике. Вместе с тем применение данной функции распределения сопряжено с оценкой большого числа параметров. Чтобы уменьшить число параметров, не теряя основных свойств распределения (9.18), полагают, что ковариационная матрица является диагональной. Это ограничение требует, чтобы компоненты вектора $\vec{x}$ были взаимно независимыми. В этом случае многомерная плотность вероятностей выражается через произведение одномерных плотностей по каждому измерению вектора $\vec{x}$.

Другой способ сокращения числа параметров основан на использовании СММ с “полунепрерывными” параметрами. В этом случае многомерную плотность распределения $b_i(\vec{x})$ признаков аппроксимируют выражением

$$b_i(\vec{x}) = \sum_{j=1}^{M} P(\vec{x} \mid c_j) b_i(c_j). \qquad (9.19)$$

Здесь $b_i(\vec{x})$ представляет собой комбинацию дискретного множества коэффициентов $b_i(c_j) = b_{ij}$, приписываемых каждому классу c_j, определенному в ходе векторного квантования, и непрерывной плотности распределения векторов $\vec{x}$, принадлежащих данному классу.

Параметры плотностей распределения (9.18) и (9.19) определяют в ходе обучения на основе процедуры Баума-Уэлча.

В заключение отметим, что для систем распознавания слитной речи с большим объемом словаря количество оцениваемых параметров может достигать нескольких миллионов. Поэтому для обучения таких систем требуется большие объёмы речевых данных [72].

Лучшие системы распознавания речи обеспечивают распознавание от 80 до 98 % слов. Если речь вводится через хороший микрофон и характеризуется небольшим объемом словаря, ограниченной моделью языка, позволяющей легко предсказать следующее вводимое слово, раздельным произнесением слов, то распознавание речи осуществляется с высокой точностью. Системы АРР пока плохо работают в условиях ввода речевых сигналов посредством телефонных линий при наличии высокого уровня окружающих акустических шумов, когда не ограничивается размер словаря, и речь является слитной.

9.5. Синтез речи по тексту

9.5.1. Основные понятия

Во многих случаях создание СИИ с элементами ЕЯ-общения требует вывода сообщений в речевой форме. На рисунке 9.30 изображена структурная схема интеллектуальной вопросно-ответной системы с речевым интерфейсом. Пользователь вводит запрос в устной форме, который преобразуется в последовательность слов с помощью подсистемы распознавания речи. Интерпретация запроса осуществляется подсистемой понимания высказываний, формирующей представление смысла запроса на внутреннем языке системы. Машина вывода, анализируя содержимое базы знаний и данных, обеспечивает получение ответа на запрос. Подсистема генерации высказываний осуществляет преобразование семантических структур, формируемых машиной вывода, в ЕЯ-текст, представленный в виде предложений. Текст с помощью подсистем синтеза речи преобразуется в речевой сигнал, который непосредственно воспринимается пользователем. К достоинствам речевого интерфейса следует отнести, доступ к информационным ресурсам СИИ с помощью простых средств, например, телефона. Кроме этого, существенно облегчается работа неподготовленных пользователей, так как для общения с системой используется естественная речь.

Рассмотрим особенности функционирования подсистемы синтеза речи по тексту. В этом случае речевое сообщение формируется из отдельных элементов речи естественного или искусственного происхождения. Ранее отмечалось, что минимальной смысловой единицей речи является фонема. Напомним, что одна и та же фонема в речи реализуется в виде

различных звуков. Последовательности различных звуков, соответствующие одной и той же фонеме, называются *аллофонами*.

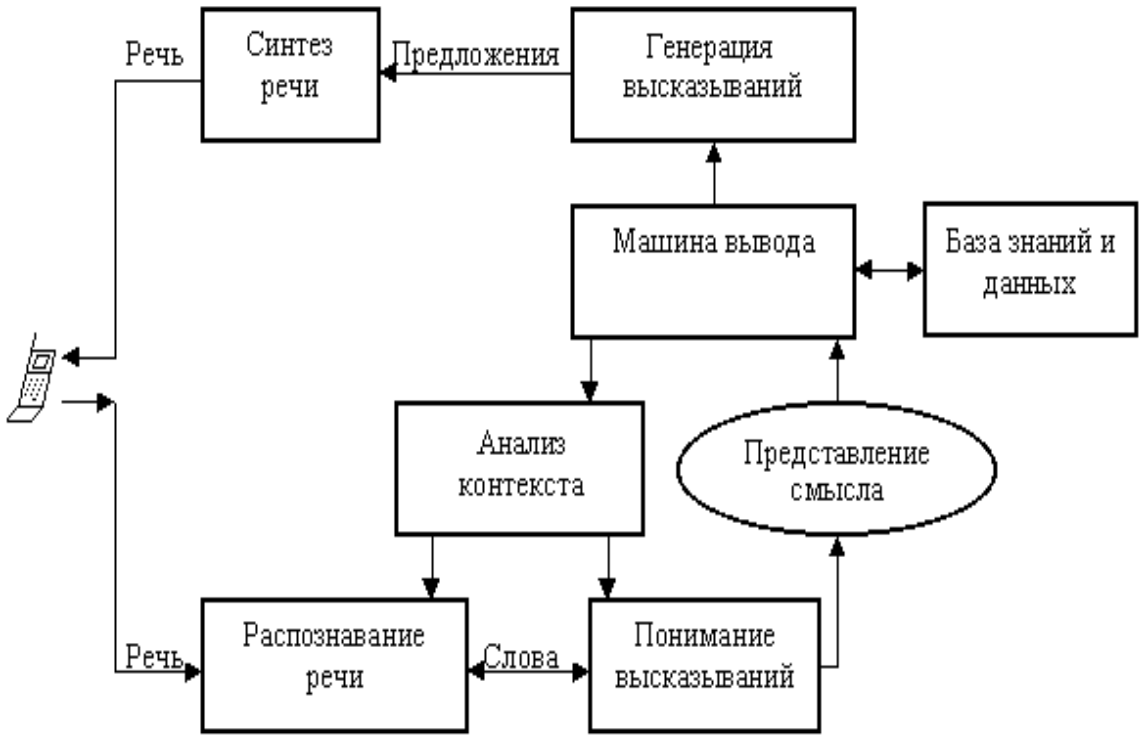


Рисунок 9.30 – Структурная схема СИИ с речевым интерфейсом

В ходе синтеза речевых сообщений используют и иные элементы, формирующую структуру речевого сигнала. Важнейшими из них являются *интонация* и *ударение*, которые определяют *просодию* речи. На уровне акустического сигнала интонация и ударение представляются просодическими характеристиками, к которым относятся [20]: *мелодика* – изменение частоты основного тона; *ритмика* – характеризуется длительностью звуков и пауз; *энергетика* – характеризуется значением интенсивности звука.

В процессе синтеза речи орфографический текст сначала преобразуется в последовательность фонем с указанием значений, необходимых просодических характеристик. С этой целью текст предварительно разбивается по определенным правилам на интонационно-смысловые группы, важнейшей из которых является *акцентная группа*.

Акцентная группа представляет собой одно или несколько слов объединенных общим просодическим контуром (мелодикой, ритмикой, энергетикой), привязанным к единому групповому ударению. Положение ударения определяется в большей степени длительностью ударного гласного и в меньшей степени его интенсивностью. Для уверенного распознавания положения ударения необходимо, чтобы длительность ударного гласного превышала длительность безударного в 1,5 раза [20].

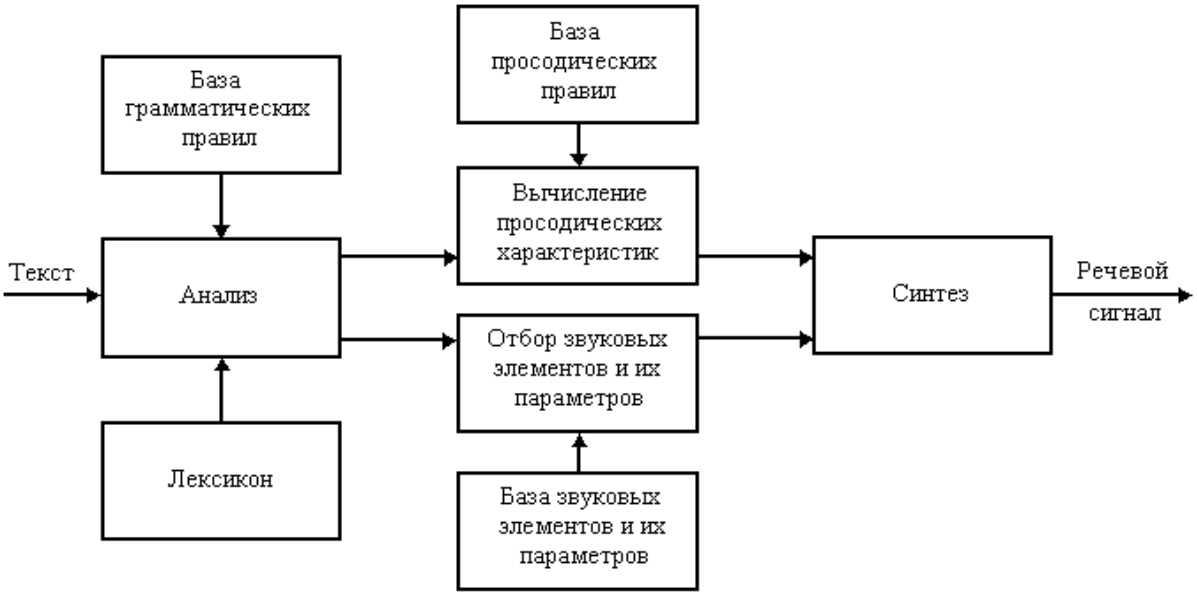


Рисунок 9.31 – Обобщенная схема преобразования текста в речь

На рисунке 9.31 изображена обобщенная схема системы, осуществляющей преобразование текста в речь. В системе предусматривается три этапа обработки. На первом этапе осуществляется морфологический и синтаксический анализ текста с целью получения последовательности фонем и разметки интонационно-смысловых групп предложений. В результате формируется внутреннее фонологическое представления текста. На втором этапе выполняется отбор звуковых элементов (или их параметров), соответствующих фонемному представлению текста, и расчет просодических характеристик: длительностей и интенсивностей звуков, частоты основного тона. Синтез речевого сигнала выполняется на третьем этапе. При этом звуки речи формируются либо из предварительно запомненных звуковых элементов естественного происхождения, либо восстанавливаются на основе параметрической модели речеобразования человека.

9.5.2. Методы синтеза речевых сигналов

В системах синтеза речи по тексту применяются три основных метода синтеза речевых сигналов: формантный синтез, синтез на основе коэффициентов линейного предсказания (ЛПК-синтез) и конкатенативный синтез.

Формантный синтез и ЛПК-синтез опираются на модель речеобразования человека. В этом случае образование звуков речи представляется как результат прохождения сигнала источника возбуждения через модель речевого тракта. При формировании гласных звуков источник возбужде-

ния формирует импульсы, частота следования которых (частота основного тона Fo) непрерывно изменяется в соответствии с мелодикой речи. Модель речевого тракта реализуется в виде цифрового фильтра с управляемыми параметрами. Возбуждение цифрового фильтра импульсным источником приводит к формированию гласных звуков. Спектр гласных звуков характеризуется рядом максимумов, соответствующих резонансным свойствам голосового тракта. Частоты максимумов называют *формантными частотами*. Глухие звуки формируются при прохождении через управляемый цифровой фильтр сигнала источника шума. Параметры фильтра могут определяться либо формантными частотами, либо ЛПК-коэффициентами (см. §9.4.2). В первом случае получают формантный синтезатор речи, а во втором – ЛПК-синтезатор речи.

На рисунке 9.32 изображена структурная схема формантного синтезатора [20]. Здесь цифровые фильтры ЦФ1 и ЦФ2 реализуются путем последовательного соединения звеньев второго порядка, моделирующих форманты. Так как ряд звуков образуется при смешанном характере возбуждения голосового тракта, когда действуют тон и шум, то в рассматриваемой модели цифровой фильтр может возбуждаться сигналами и источника импульсов, и источника шума. Управляющими параметрами модели является:

Арн – амплитуда возбуждения ротового и носового каналов голосового тракта;

Аа – амплитуда аспиративного возбуждения ротового канала (задается при формировании небных согласных).

Аф – амплитуда возбуждения фрикативных формант;

Fo – частота основного тона;

Fi,Bi – частота и ширина полосы *i*-ой форманты, *i*=1,2,3,4.

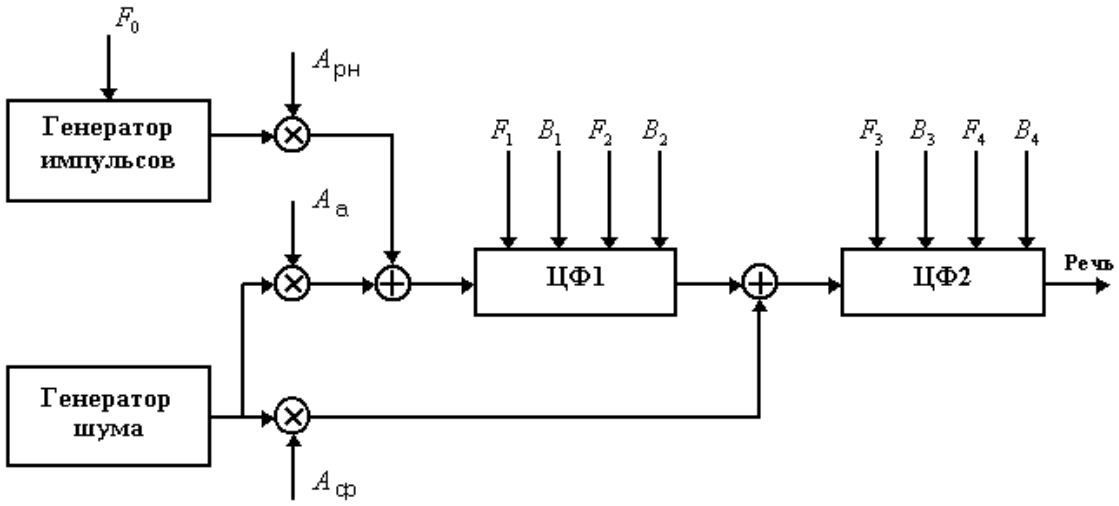


Рисунок 9.32 – Структурная схема формантного синтезатора

При синтезе гласных носовых (‘н’,’м’ и др.) и аспиративных звуков (‘к’,’г’ и др.) задаются значения четырех ротовых или двух носовых формант (F1-F4, B1-B4). При синтезе фрикативных звуков (‘с’,’ш’ и пр.) фильтр ЦФ2 настраивается на частоты первой и второй фрикативных формант (F3-F4, B3-B4). Диапазоны значений указанных параметров приведены в [20]. Формантные синтезаторы при надлежащем управлении параметрами позволяют получать синтезированную речь высокого качества. Однако их применение требует проведения большой подготовительной работы, связанной с определением параметров формант и их взаимосвязи с фонемами.

В ЛПК-синтезаторах речевой тракт моделируется рекурсивным цифровым фильтром с передаточной функцией

$$H(z) = \frac{1}{1 + \sum_{i=1}^{N} a_i z^{i-1}}, \qquad (9.20)$$

где a_i – коэффициенты, определяемые по методу линейного предсказания.

ЛПК-анализ был изначально разработан с целью сжатия и восстановления речевых сигналов. Однако хорошее обоснование ЛПК-модели речеобразования позволили применить ее в задаче синтеза речи по тексту. При этом имеется возможность согласования звуков по энергии, по темпу, по частоте основного тона. В то же время звуки, формируемые с помощью носового канала голосового тракта, плохо синтезируются фильтром вида (9.20), имеющим только полосы. Кроме этого, расположение пиков в спектре синтезируемой речи оказывается весьма чувствительным к гармонической структуре возбуждающих импульсов. Это приводит к ухудшению качества синтезируемой речи.

Лучшие результаты в отношении разборчивости и натуральности звучания речи удается получить с помощью конкатенативного метода синтеза (от англ. concatenate – сцеплять, сглаживать). Наибольшее распространение получили системы дифонного конкатенативного синтеза. *Дифоном* называют участок речи, расположенный между центрами двух соседних фонем. Дифон содержит переходные элементы речи от одной фонемы к другой, что обеспечивает при синтезе речи ее высокую натуральность. Кроме этого, благодаря тому, что границы дифонов соответствуют серединам фонем, где спектр наиболее стабилен, дифоны легко выделяются. Слова речи при этом синтезируются из предварительно запомненных дифонов. Например, речевой сигнал, соответствующий слову “ива”, образуется из дифонов [/и], [ив], [ва], [а/], которые могли быть получены из естественной речи при произнесении слов “ил”([/и]), “пиво”([ив]), “квас”([ва]), “изба”([а/]). Здесь знак “/” обозначает паузу речи. При дифонном синтезе требуется запоминание большого числа дифонов (до нескольких тысяч).

В ходе конкатенативного синтеза возникают две основные проблемы. Во первых, формирование речевого сигнала из сегментов, соответствующих дифонам, требует, устранения разрывов, возникающих на границах сегментов. Во-вторых, необходимо осуществлять такое независимое управление длительностью дифонов и частотой основного тона, чтобы сохранялись спектральные портреты формант.

Если разрывы на границах сегментов устраняются путем обычного сглаживания, то управление длительностью дифонов и частотой основного тона требует специального рассмотрения.

Рассмотрим управление просодическими характеристиками синтезируемой речи во временной области на основе метода синхронизации частоты основного тона путем наложения и смещения речевых участков (TD-PSOLA – time-domain pitch - synchronous overlap-add). В соответствии с этим методом сначала выполняется анализ речевого сигнала. При этом размечается каждый период основного тона и участки речи, соответствующие фрикативным звукам. На рисунке 9.33 изображен пример разметки речевого сигнала. Квазипериоды основного тона обозначены на рисунке буквой “p”, а участки речи, соответствующие фрикативным звукам буквой “u”. Затем формируется последовательность перекрывающихся кратковременных сигналов, образуемых из соседних участков речи, взвешенных с помощью окна Хемминга *w*. Длительность каждого кратковременного сигнала равна двум квазипериодом основного тона. Кратковременный сигнал рассматривается как звонкий, если его правая часть имеет метку “p”, и наоборот, если правая часть имеет метку “u”, то сигнал считается фрикативным.

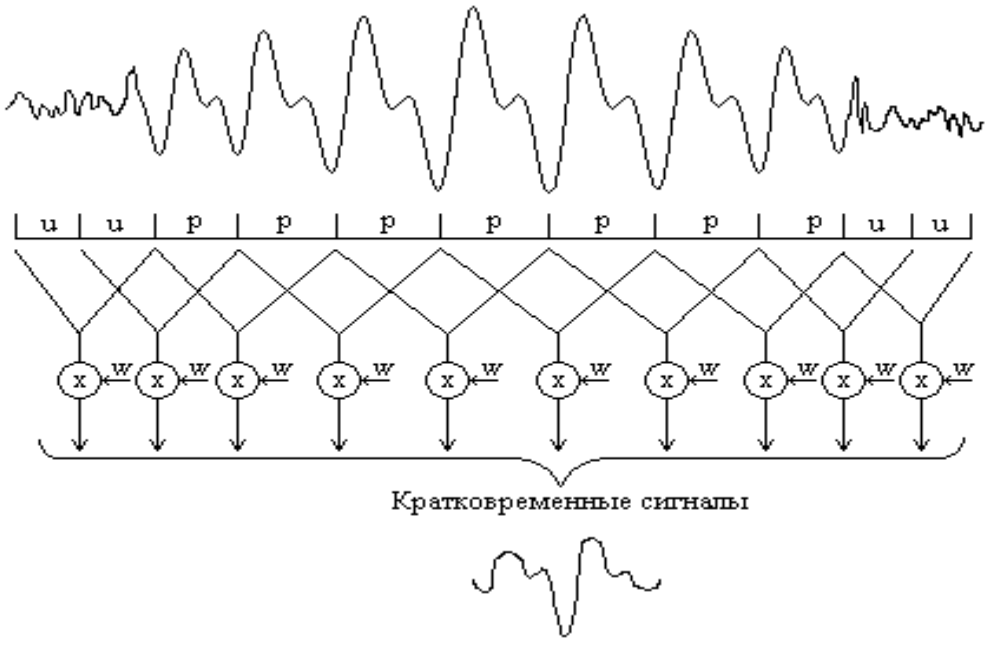


Рисунок 9.33- Формирование кратковременных речевых сигналов

Синтез речевого сигнала заключается в объединении кратковременных сигналов, полученных в ходе анализа. Если расположить все кратковременные сигналы в точном соответствии с их исходным положением, то можно восстановить речевой процесс.

Если требуется изменить частоту основного тона, то перед объединением кратковременных сигналов увеличивают или уменьшают расстояние между их центрами. Так как изменение частоты основного тона может привести к изменению длительности исходного речевого сигнала, то выполняется коррекция длительности.

Управление длительностью состоит в повторении или исключении отдельных сигналов. Повторение кратковременных звонких сигналов не вызывает никаких проблем. Однако простые повторения фрикативных кратковременных сигналов приводят к ухудшению звучания речи. Один из путей решения данной проблемы состоит в “инвертировании” времени каждого нечетного повторения фрикативного кратковременного сигнала. Рассмотренная процедура позволяет перестраивать частоту основного тона в 0,7-1,4 раза при изменении длительности в 0,5-2 раза без заметного ухудшения качества речи [92].

9.5.3. Определение управляющих параметров синтезаторов сигналов

Задача преобразования текста в речь предполагает решение двух подзадач (рисунок 9.31): преобразование орфографического текста в последовательность фонем с выделением интонационно-смысловых групп и разметкой ударений; определение параметров, непосредственно управляющих процессом синтеза речевых сигналов (параметры формант или индексы дифонов, длительности звуковых элементов и частота основного тона).

Преобразование орфографического текста. Орфографический текст, поступающий на вход системы синтеза речи, подвергается разметке. В нём выделяются следующие фонетические элементы: фраза, синтагма, акцентная группа, фонетическое слово, слог.

Фраза – это наиболее крупная фонетическая единица, представляющая законченное по смыслу высказывание, объединенное особой интонацией и отделенное от других таких же единиц паузой. Фраза не совпадает с предложением. В предложении может быть несколько фраз. Например: “Шумит лес, журчит ручей, / Сергей, прикрыв глаза, наслаждался природой / ”. Произнося это предложение, мы делаем паузы (знак ‘/‘), отделяя одну фразу от другой. Каждая фраза интонационно оформлена. *Интонация* проявляется в изменениях частоты основного тона, темпа речи, силы (интенсивности) звучания, внутрифразовых паузах. С помощью интонации

производится членение предложения на семантико-синтаксические единицы – *синтагмы*. Одно и то же предложение в зависимости от цели сообщения может члениться на синтагмы по-разному. Например: “Петр вылетел в Париж” – предложение состоит из одной синтагмы; “Петр вылетел / в Париж” – предложение состоит из двух синтагм, где вторая синтагма уточняет место. Каждой синтагме может присваиваться один из возможных интонационных типов: повествовательный, вопросительный, восклицательный. Повествовательный тип характеризуется повышением тона в начале синтагмы и понижением в конце синтагмы; вопросительный тип характеризуется резким повышением тона в конце синтагмы; интонация восклицательной синтагмы одинаково высокая. Существует также большое число оттенков указанных интонационных типов. Например, повествовательная синтагма внутри предложения характеризуется интонацией незаконченности. Паузы между синтагмами короче, чем между фразами.

В пределах каждой синтагмы выделяют *акцентные группы*, соответствующие одному или нескольким словам, связанным общим групповым ударением. Слова в акцентной группе получают разные по силе ударения. Ударение, характеризуемое большей силой, называется основным. Основное ударение часто получает первое слово акцентной группы. Остальные слова получают слабое ударение. При этом во многих системах синтеза речи выделяют разные уровни относительных ударений, обозначаемые цифрами. Например:

1	3		2
“ Петр	вылетел	в	Париж ”.

Здесь слово с ударением основного уровня обозначено цифрой 1, слово с ударением второго уровня цифрой 2 и т.д.

Необходимо отметить, что назначение уровней ударения словам в значительной степени определяется семантическим контекстом предложения. Так, если необходимо подчеркнуть, что Петр не поехал поездом, а отправился в Париж самолетом, то уровни ударения будут назначены следующим образом:

3	1		2
“ Петр	вылетел	в	Париж ”.

Следует различать понятия лексического и фонетического слова. Фраза имеет столько фонетических слов, сколько ударений. Так как некоторые лексические слова не несут ударений, то нередко фонетических слов оказывается меньше, чем лексических. К безударным словам относятся предлоги, союзы, частицы, которые в ходе разметки предложения “склеивают” с предыдущим или следующим словом. Безударные слова, объеди-

няемые со следующим словом, называют *прокликтиками*, а объединяемые с предыдущим словом – *энкликтиками*. Прокликтиками обычно представлены предлоги, союзы, частицы. Энкликтиками часто бывают частицы и знаменательные слова (пóд ноги, чáс óт часу).

В простых системах разметка текста выполняется на основе анализа знаков пунктуации и несложных правил преобразования орфографического текста в фонетический.

В более совершенных системах выполняется автоматическая разметка текста с помощью методов синтаксического и морфологического анализа. В ходе указанных видов анализа для каждого предложения может строиться дерево грамматического разбора, отражающее как структуру предложения, так и структуру каждого слова. Выполнение синтаксического анализа позволяет:

- корректно определять интонационно смысловые группы, назначать уровни ударений, присваивать группам соответствующие интонационные типы;
- определять правильное произношение аббревиатур и чисел, например, “16 января состоится…” (шестнадцатого января состоится…) или “к 16 января” (к шестнадцатому января).

В ходе синтаксического анализа устанавливается грамматическая функция каждого слова предложения. При этом обычно используется словарь (лексикон), в котором для каждого слова вместе с его орфографическим написанием хранятся следующие сведения: падеж, число, род, фонетическая транскрипция. Наличие производных слов языка требует выполнения морфологического анализа, в ходе которого корректно определяются необходимые грамматические атрибуты слов, не обнаруженных в словаре. В этом случае применяются словари морфем, из которых по определенным правилам образуется слово. Транскрипция слова, не найденного в словаре, также получается с помощью правил.

Таким образом, на выходе блока анализа (рисунок 9.31) формируется фонетическое представление предложений текста с выделенными интонационно-смысловыми группами, их интонационными типами и уровнями ударений. На основе данного представления осуществляется определение просодических характеристик и выполняется поиск звуковых элементов (или их параметров), из которых синтезируется речевой сигнал.

Определение просодических характеристик. Расчет просодических параметров форматного синтезатора детально описан в [20]. Поэтому здесь проиллюстрируем только основные моменты определения параметров конкатенативного синтезатора на примере системы SVOX [92], осуществляющей преобразование текста в речь для немецкого языка.

Рассмотрим в качестве примера фонологическое представление предложения “ein grosses atlantisches Sturmtief verlagert sich heute nach Os-

ten” (большой атлантический циклон перемещается сегодня на восток), формируемое на выходе блока анализа системы SVOX:

#{0}(P)| ain[2]gro:-səs|at-[3]lan-ti-ʃəs[1]
ʃtʊrm-[4]ti:f#{2} (T) fɛr-[3]la:-gərt ziç[2]
hɔy-tə na:x[1]|ɔs-tən#{0}.

Здесь использованы следующие условные обозначения:

#{n} – граница фразы, где n – число; если n=0, то обозначение соответствует паузе между предложениями; если n=1, то паузе внутри предложения; если n>1, то обозначает границу фразы без паузы;

(X) – обозначает интонационный тип; если X=P, то речевой такт (от одной границы до другой) характеризуется интонацией незаконченности; если X=T, то речевой такт характеризуется интонацией, свойственной окончанию повествовательного предложения;

[n] – обозначает уровень ударения, где n=0,1,2,3; 0 – обозначает безударный слог.

Уровни ударения в системе SVOX интерпретируются следующим образом:

[1] → [+f,+d];
[2] перед [1] → [+f,+d];
[2] после [1] → [-f,+d];
[3] → [-f,+d];
[0] → [-f,-d].

Здесь f обозначает ударение, ассоциируемое с повышением частоты основного тона; d – ударение, реализуемое увеличением длительности слога; ‘+’ и ‘-’ обозначают наличие или отсутствие соответствующего признака.

Управление длительностью звуков в системе SVOX выполняется с помощью статистической линейной модели. Входными параметрами модели являются уровни ударений фонологического представления, длительности дифонов, окружение дифонов. Коэффициенты модели определяются в процессе обучения на парах “транскрипция – продолжительность звучания”. Модель предсказывает непосредственно длительность трифонов, которая затем пересчитывается в длительность дифонов или полудифонов. Требуемая длительность дифона получается путем его линейного растяжения по времени.

На этапе определения длительности звуковых элементов также формируются необходимые временные метки, обеспечивающие синхронизацию частоты основного тона. Эти метки проставляются в середине и конце каждого слога.

Ниже показан пример назначения длительностей дифонов и временных меток для следующего фрагмента ‘”es regnet in strömen” (плохо, когда идет дождь):

- фонологическое представление

(T) | εs [2] re: - gnet | in [1] ʃtrø : - mən;

- длительности (мс)

(/ε, 76) , (εs$_a$, 44) (εs$_b$ 36) (sr, 76), (re, 92), (↑e, 14),
(e>, 40), (gn$_a$ 50), (gn$_b$, 22) (nə, 46), (ə>, 62), (↑/, 6),
(t>, 45), (↑/, 18) (/i, 48), (in, 40), (n⌋$_a$, 59) (n⌋$_b$, 43),
(s>45), (↑/, 21), (tr, 77), (rø, 79), (↑ø, 15) (øm, 74) (mə, 45),
(ən, 67), (n/, 99);

- временные метки слогов (мс)

(0, 76, 194), (194, 338, 378), (378, 496, 609),
(609, 675, 774), (774, 1054, 1091), (1091, 1173, 1339).

При записи длительности звуков использованы следующие обозначения:

‘/’ – обозначает паузу;
‘>’ – обозначает переход к короткой паузе;
’↑’ – обозначает продолжение звука.

Некоторые из дифонов разделены на полудифоны (обозначены индексами a и b) для более тонкого управления длительностью.

Управление частотой основного тона в системе SVOX выполняется с помощью нелинейной модели, реализованной в виде рекуррентной нейронной сети Элмана, на вход которой подается фонологическое представление. Для рассмотренного выше примера фонологического представления формируется следующий контур F_0, состоящий из пар “время-частота”:

(0.00, 117.5), (0.04, 117.5), (0.08, 116.1),
(0.12, 113.7),(0.16, 110.8), (0.20, 110.4),
(0.24, 111.0), (0.28, 113.2), … , (1.28, 72.6), (1.32, 72.8).

Здесь время указано в секундах, а частота в герцах. Если необходимо вычислить значение F_0 в промежутке между временными отсчетами, то осуществляется линейная интерполяция.

Структурная схема нейронной сети Элмана изображена на рисунке 9.34. Сеть обеспечивает предсказание частоты основного тона для контура слога по фрагменту фонологического представления, попавшего во входное окно. Полный контур F_0 получается при сдвиге входного окна относительно фонологического представления. Веса связей рассматриваемой

ИНС инициализируются случайными значениями в диапазоне - 0,3 ÷ 0,3. Передаточные функции нейронов описываются униполярной сигмовидной функцией.

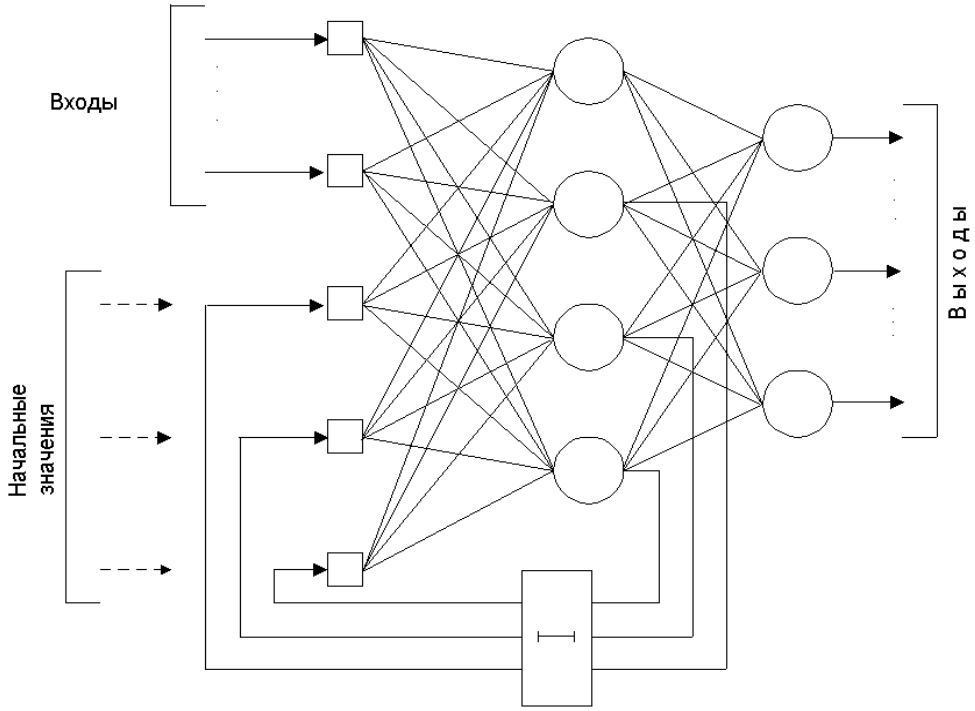


Рисунок 9.34 – ИНС Элмана

Обучение сети выполняется на данных, получаемых в результате прочтения текстов различных стилей. При этом обучающие данные организованы в виде пар “фонологическое представление – контур F_0”. Обучающий контур частоты основного тона восстанавливается по отсчетам F_0, которые вычисляются каждые 15 мс. Для участков речи, соответствующих фрикативным звукам, назначаются виртуальные значения F_0, получаемые с помощью линейной интерполяции. Для обучения сети используется разновидность алгоритма обратного распространения ошибки. В ходе обучения и распознавания каждый символ входного фонологического представления заменяется соответствующими бинарными значениями. На выходе сети формируются 8 значений F_0, представляющих контур F_0 для слога.

Применение ИНС для управления частотой основного тона позволило получить в системе SVOX высокую разборчивость и естественность звучания синтезированной речи.

Вопросы для самопроверки

1. Назовите и объясните назначение основных функций, реализуемых подсистемой ЕЯ-общения.
2. Объясните на простом примере основные этапы анализа и интерпретации предложений подсистемой понимания высказываний.

3. В чем состоит отличие между рациональным и эмпирическим подходами к задаче понимания высказываний?
4. Что понимают под расширенными сетями переходов?
5. Объясните общий принцип грамматического разбора с помощью расширенных сетей переходов.
6. Сформулируйте задачу распознавания частей речи на основе эмпирического подхода.
7. Объясните принцип ЕЯ-доступа к базам данных на основе метода сопоставления с образцом.
8. Какой принцип использован для реализации ЕЯ-доступа в БД LUNAR? Объясните его.
9. Нарисуйте структурную схему ЕЯ-интерфейса БД с промежуточным представлением запроса в логической форме. Объясните функции каждого из блоков.
10. Объясните идею построения ЕЯ-интерфейса БД с возможностью логического вывода.
11. Назовите основные характеристики систем автоматического распознавания речи.
12. Нарисуйте обобщенную структурную схему системы автоматического распознавания речи. Объясните назначение основных модулей.
13. Опишите свойства речевого сигнала.
14. Что называется формантой?
15. Какие операции выполняются в ходе кепстрального анализа?
16. В чем заключается суть ЛПК-анализа?
17. Что понимают под векторным квантованием?
18. Сформулируйте задачу распознавания речи в статистической постановке.
19. Что понимают под биграмной моделью языка?
20. Какие проблемы возникают при построении модели языка? Как их решают?
21. Что понимают под акустической моделью языка?
22. Что представляет собой СММ? Приведите формальное определение СММ.
23. Объясните постановку вычислительной подзадачи, возникающей при использовании СММ.
24. Объясните постановку подзадачи восстановления , возникающей при использовании СММ.
25. Объясните постановку подзадачи оценивания, возникающей при использовании СММ.
26. Объясните на примере решение вычислительной подзадачи СММ.
27. Каким образом выполняется композиция модели языка, лексической модели и модели фонемы?
28. Сформулируйте алгоритм Витерби.
29. Объясните на примере процесс вычислений в соответствии с алгоритмом Витерби.
30. Как осуществляется оценивание параметров СММ? Сформулируйте подход Баума-Уэлча.
31. Какие особенности возникают при использовании СММ с непрерывными параметрами?
32. Приведите структурную схему СИИ с речевым интерфейсом.
33. Что называют аллофоном?
34. Что понимается под просодией речи?

35. Объясните обобщенную схему системы, выполняющей преобразование текста в речь.
36. Объясните принцип формантного синтеза речи.
37. Объясните принцип синтеза речи с помощью ЛПК- коэффициентов.
38. Объясните принцип конкатенативного синтеза речи.
39. Как осуществляется управление просодическими характеристиками в методе TD-PSOLA.
40. Приведите определения следующих понятий, используемых при синтезе речи по тексту: фраза, интонация, синтагма, акцентная группа, основное ударение, прокликтики, энкликтики
41. Объясните на примере системы SVOX, преобразующей текст в речь, принцип определения просодических характеристик.

ГЛАВА 10
КОМПЬЮТЕРНОЕ ЗРЕНИЕ

В настоящей главе рассматриваются основные вопросы, связанные с построением подсистем компьютерного зрения.

После краткого введения в проблематику компьютерного зрения и установления его неразрывной связи с искусственным интеллектом, анализируются особенности системы зрения человека. Одна из мотиваций такого анализа – стремление обнаружить принципы, которые могли бы использоваться в системах компьютерного зрения.

Затем рассматриваются примеры существующих систем компьютерного зрения и выделяются базовые операции, используемые в процессе анализа изображений. Рассмотрение начинается с операции выделения границ. Описываются различные возможности по группировке пикселей, представляющих границы. Один из параграфов посвящён подходу, основанному на выделении областей, который является альтернативой подходу, связанному с выделением границ.

Конечная цель подсистемы компьютерного зрения – формирование символьного описания изображения, т.е. его понимание. С этой целью в главе анализируются ранние попытки интерпретации трехмерных объектов и сцен по контурным рисункам. В заключение осуществляется переход от общих методов компьютерного зрения к системам технического зрения роботов. После краткого обзора методов приводится ряд примеров, показывающих, каким образом общие принципы компьютерного зрения могут быть перенесены на реальные задачи.

10.1. Основные понятия

Исследования в области компьютерного зрения направлены на изучение методов понимания изображений. На основе одного или нескольких двумерных изображений системы компьютерного зрения формируют символьное описание реальности, представляемой изображением. Если

представляемая изображением реальность ─ часть трехмерного мира, то следует говорить о трехмерном компьютерном зрении или *анализе сцен*. Изображения, анализируемые системами компьютерного зрения, могут сильно различаться по своей природе. Двумерные черно-белые и цветные изображения, множественные изображения в случае стерео зрения, диапазонные изображения (получаются с помощью датчиков глубины), трехмерные изображения, формируемые с помощью магнитного резонанса (МР), ─ все это примеры входных изображений для систем компьютерного зрения. Кроме этого, при рассмотрении последовательности изображений учитывается время, которое рассматривается как третье или четвертое измерение.

Компьютерное зрение и искусственный интеллект были связаны друг с другом изначально. Многие из универсальных методов и технологий искусственного интеллекта специально разрабатывались для решения задач из области компьютерного зрения, например, персептрон и искусственные нейронные сети, поиск с распространением ограничений, теория фреймов и др. Поэтому компьютерное зрение называют ключевой областью искусственного интеллекта.

Понимание изображений – это то, что человек выполняет без усилий, часто не осознанно, в течение всего дня. Однако научить компьютер решать эту задачу – чрезвычайно сложное дело. Это дополнительно подтверждает то, что компьютерное зрение относится к области искусственного интеллекта в соответствии с определением ИИ.

Простой пример позволит уточнить, что понимается под термином “компьютерное зрение”. Если с помощью черно-белой камеры зафиксировать простую сцену с многогранным блоком, стоящим на столе, то полученное изображение будет представлять собой матрицу (массив) значений интенсивности, которая соответствует световой энергии, воспринимаемой в течение короткого интервала времени чувствительными элементами камеры (например, приборами с зарядовой связью – ПЗС). Данный массив значений сохраняется в памяти компьютера. Если рассматривать часть значений этого массива (рисунок 10.1), то уловить смысл сцены невозможно. Очевидно, рассмотрение оставшейся части массива значений также не решит проблемы. Компьютер в этой ситуации, конечно, находится тоже не в лучшем положении. Улучшить восприятие можно, если воспользоваться представлением сцены, в котором значение интенсивности интерпретируется как высота некоторого горного ландшафта (рисунок 10.2). Это представление лучше, но смысл изображения всё еще не понятен. И наконец, если представить каждое запомненное значение интенсивности, соответствующей яркостью пикселя на экране компьютера, то можно получить изображение, которое понятно (рисунок 10.3).

162	**162**	**160**	**158**	**156**	**153**	**153**	**151**	**149**	**149**	**149**	**145**	**145**	**145**	**151**	**156**	**169**	**178**
180	**162**	**162**	**162**	**162**	**160**	**156**	**158**	**151**	**149**	**153**	**149**	**147**	**147**	**145**	**151**	**158**	**169**
178	**180**	**162**	**160**	**160**	**162**	**160**	**160**	**158**	**153**	**149**	**151**	**149**	**145**	**145**	**142**	**147**	**156**
167	**173**	**175**	**162**	**162**	**162**	**162**	**162**	**162**	**160**	**158**	**158**	**153**	**151**	**147**	**142**	**140**	**147**
153	**164**	**171**	**171**	**160**	**160**	**162**	**160**	**160**	**160**	**162**	**158**	**158**	**156**	**149**	**147**	**140**	**142**
145	**151**	**162**	**164**	**162**	**160**	**164**	**164**	**162**	**162**	**162**	**160**	**160**	**158**	**158**	**153**	**151**	**145**
142	**147**	**151**	**156**	**162**	**162**	**160**	**162**	**160**	**162**	**160**	**158**	**160**	**160**	**160**	**156**	**153**	**149**

Рисунок 10.1– Матрица значений интенсивности

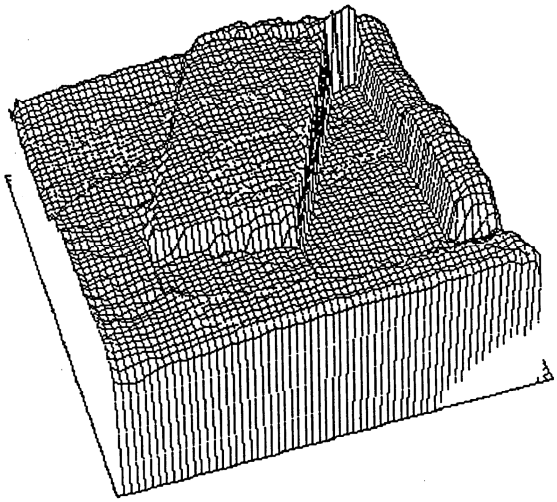


Рисунок 10.2 – Представление интенсивности в виде высоты

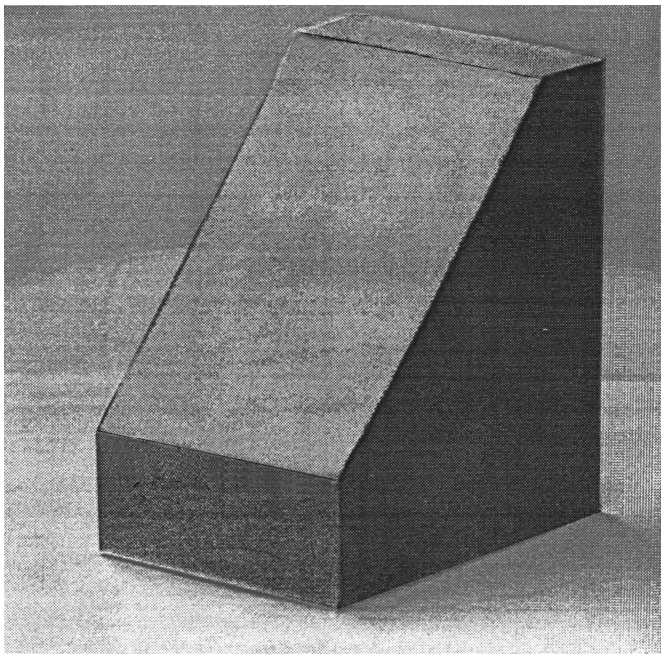


Рисунок 10.3 – Представление интенсивности яркостью пикселя

Система компьютерного зрения должна формировать символьное или словесное описание сцены (рисунок 10.4). Именно такое описание может быть понятным роботу, который должен собрать предметы со стола и перенести их в другое место. Формирование такого символьного описания предполагает наличие априорных знаний об объектах реального мира и соответствующего словаря.

БЛОК В7
ЛЕЖИТ НА СТОРОНЕ А2
ЦЕНТР ТЯЖЕСТИ (16,8 8,0 2,7)
ЛИЦЕВАЯ СТОРОНА ОБРАЩЕНА НА ЮГО-ЗАПАД

Рисунок 10.4 – Символьное описание сцены

Задачи, относящиеся к компьютерному зрению, являются сложными по многим причинам. Отметим некоторые из них.

1. Изображения обычно представляют собой проекции трехмерной реальности на двухмерную плоскость (например, на матрицу ПЗС элементов камеры). В ходе этого процесса безвозвратно теряется информация о сцене, главным образом информация о глубине, и, следовательно, утрачивается информация об отношениях соседства между объектами сцены. Чтобы выполнить восстановление трехмерной реальности с помощью компьютера, необходимо использовать, наряду с данными, представляющими изображение, разумные допущения (эвристики) и априорные знания о сцене. Восстановить трехмерную картину можно (при определенных ограничениях), воспользовавшись двумя камерами и пространственной схемой, изображенной на рисунке 10.5. Здесь используется тот факт, что в схеме с двумя камерами, оптические центры которых находятся на базовом расстоянии b, изображение физической точки P(x,y,z) на каждой из плоскостей изображений смещается на некоторую величину по отношению к точке пересечения оптической оси с плоскостью изображения. Разность указанных смещений называется диспаритетом. Для точек, находящихся на значительном расстоянии от камеры, диспаритет примерно равен нулю. Рассмотренная пространственная схема позволяет находить координаты x, y, z точки P, если она фиксируется обеими камерами. Чтобы восстановить трехмерную сцену (*3D-сцену*), необходимо получить изображения точечных стереопар для соответствующих точек объектов, образующих сцену.

2. Шумы изображения, создаваемые датчиками изображений, также приводят к потерям информации. Это следует учитывать при анализе изображений. Прежде чем выполнить ту или иную операцию по извлечению информации из изображения, необходимо осуществить регуляризацию

изображения, например, сглаживание путем свертки изображения с двумерным окном Гаусса;

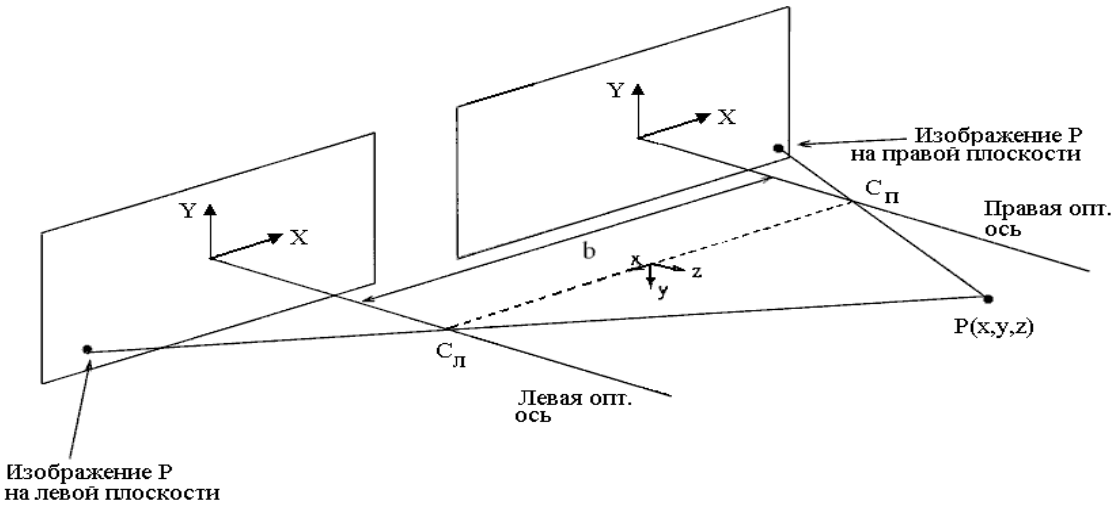


Рисунок 10.5 – Схема получения стереоизображения

3. Чтобы зафиксировать точку изображения с определенной интенсивностью и цветом, применяют различные физические процессы. При этом учитываются многие особенности этих процессов, например: характер излучения (прямое и косвенное), форма и отражающие способности поверхности объекта, тени, атмосферное поглощение и др. Учет всех особенностей не возможен без априорных знаний или эвристик;

4. В области компьютерного зрения требуется быстро обрабатывать большие объемы данных. Обычное изображение имеет размер 512x512 *пикселей (*элементы изображения) с разрешением по интенсивности 8 битов (256 градаций). Система зрения человека может обрабатывать в реальном времени объемы данных, намного превышающие названные. В компьютерном зрении тоже имеются задачи, требующие решения в реальном времени. Например, в робототехнике и различных системах наблюдения. Анализ последовательности изображений (подвижных изображений) – одно из быстро развивающихся направлений компьютерного зрения, требующее больших вычислительных ресурсов.

10.2. Система зрения человека

При разработке систем компьютерного зрения весьма ценными являются сведения о системе зрения человека. Зрение – одно из важнейших чувств человека. Оно незаменимо при ориентации в окружающей

среде, а также при выполнении коммуникативных функций (тексты, диаграммы, картины и т.д.). Поэтому не удивительно, что примерно одна четвертая часть нашего мозга (половина коры головного мозга) относится к системе зрения. Источниками знаний о системе зрения человека являются:

- нейроанатомия (изучает строения мозга посредством препарирования);
- анатомия мозга (изучает дисфункции после повреждений мозга);
- нейрофизиология (изучает электрические сигналы нейронов);
- психофизические эксперименты (реакции на различные стимулы);
- самоанализ (к сожалению, не очень полезно).

Рассмотрим кратко некоторые результаты исследований в указанных областях.

Нейроанатомический и нейрофизиологический подходы. Общая схема глаза человека изображена на рисунке 10.6. [40]. Устройство глаза имеет много общего с оптическими приборами, например, с ПЗС камерой. При этом имеется и много отличий: диафрагма и фокусное расстояние хрусталика глаза изменяются с помощью мышц; изображение проецируется на вогнутую поверхность сетчатки; плотность фоторецепторов на поверхности сетчатки зависит от расстояния до центральной ямки. Имеются и другие существенные отличия.

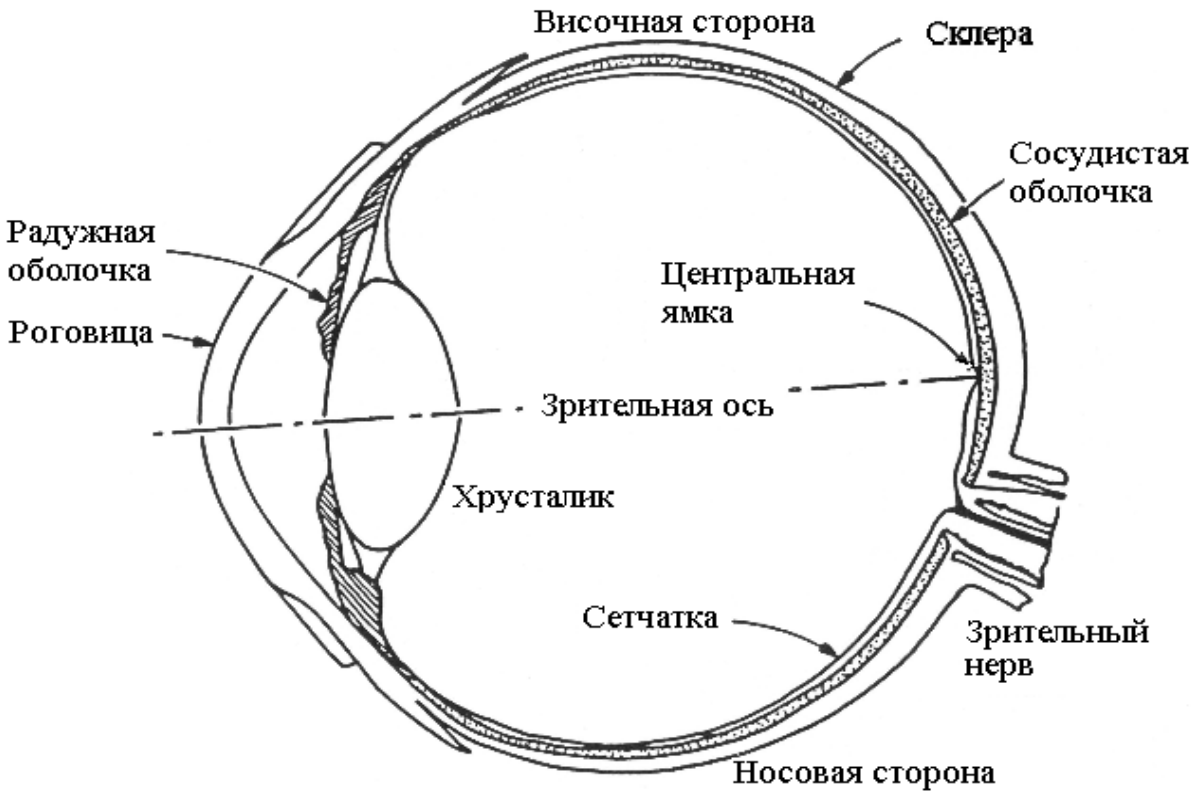


Рисунок 10.6 – Строение глаза человека

Рецепторы глаза. Сетчатка содержит большое количество светочувствительных рецепторов. Рецепторы подразделяются на палочки и колбочки в соответствии с их формой. В глазу человека имеется 120 миллионов палочек и 6 миллионов колбочек. Распределение палочек и колбочек на поверхности сетчатки показано на рисунке 10.7 [40]. Палочки реагируют на низкие, а колбочки на более высокие световые уровни. Имеются три типа колбочек, которые являются чувствительными к различным длинам световых волн λ (рисунок 10.8)[40].

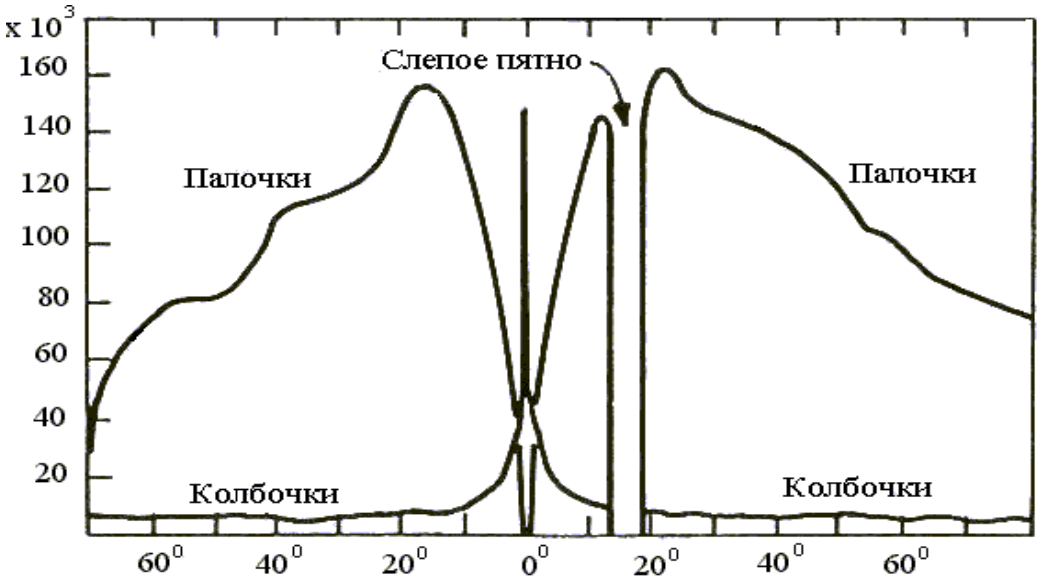


Рисунок 10.7 – Число палочек и колбочек на миллиметре сетчатки в зависимости от угла отклонения от зрительной оси

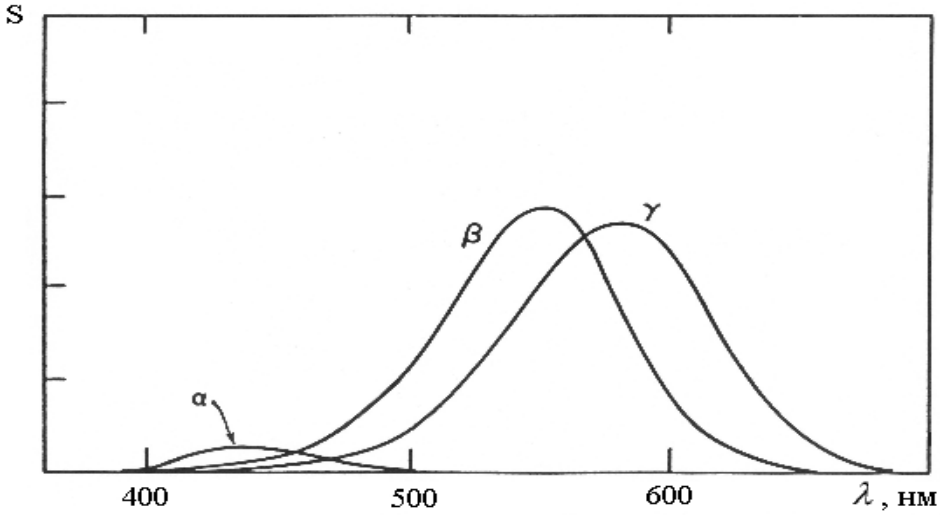


Рисунок 10.8 – Относительная чувствительность (S) колбочек

Предварительная обработка данных в глазу. В сетчатке обнаружена вертикальная и горизонтальная организация (рисунок 10.9). Горизонталь-

ные связи вызывают подавление активности одних рецепторов при возбуждении других рецепторов. Волокна зрительного нерва начинаются у ганглиозных клеток и проходят через слепое пятно глаза. Общее количество волокон приблизительно равно 800 000. В то же время глаз имеет 126 миллионов рецепторов. Очевидно, значительное сжатие визуальных данных происходит уже в глазу.

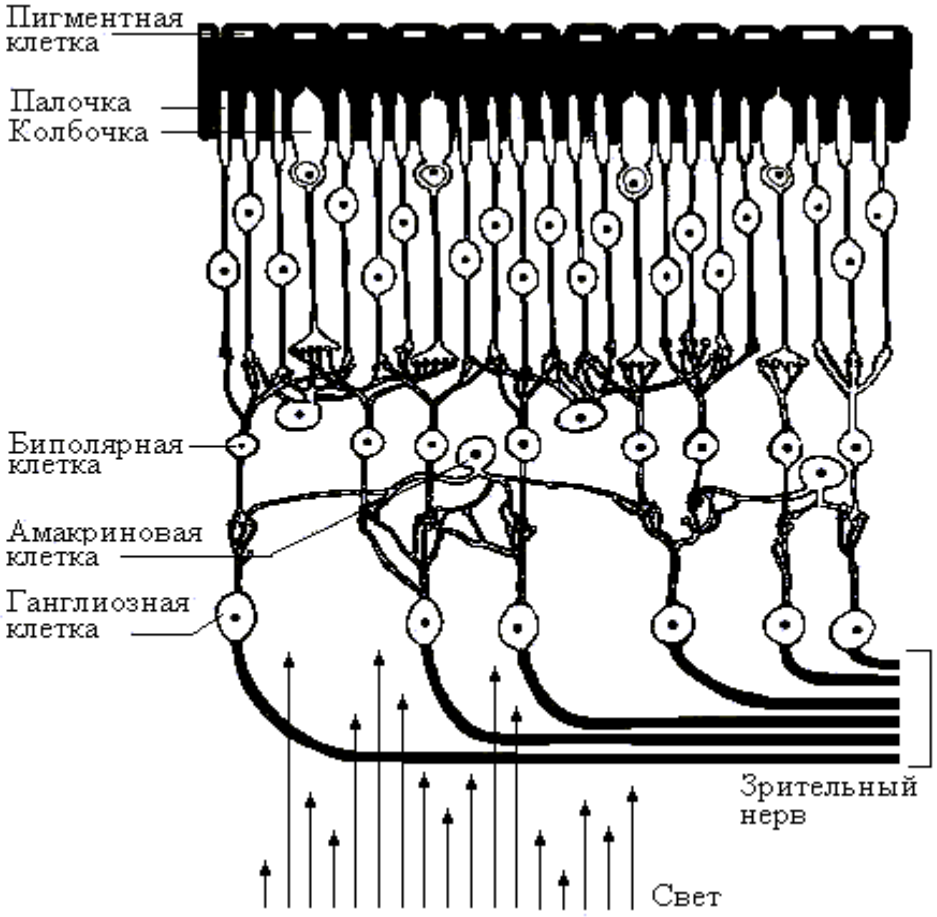


Рисунок 10.9 – Фрагмент сетчатки глаза

Передача сигналов (рисунок 10.10). Сигналы распространяются, начиная с аксонов ганглиозных клеток, и следуют со скоростью примерно 80 м/с. Эти сигналы представляют собой последовательности электрических импульсов. Частота следования импульсов составляет 10-1000 Гц. Даже, когда нет никакого визуального стимула, вдоль аксонов следуют самопроизвольные импульсы с частотой несколько герц. Аксоны связывают одни нейроны с другими нейронами. Зрительный нерв соединяется с хиазмой, где волокна нерва перераспределяются таким образом, чтобы каждое полушарие мозга получило два изображения. Это необходимо для восприятия пространственной информации. Левое полушарие обрабатывает два изображения правой половины сцены, а правое, наоборот, – два изображения левой половины сцены. В конечном итоге сигналы достигают

первичной зрительной коры.

Зрительная кора. Входящие аксоны соединяются со зрительной корой так, чтобы смежные области поля зрения оставались смежными и в зрительной коре с одним исключением: поле зрения разбивается в вертикальном направлении на две части и границы этих двух областей расположены в зрительной коре далеко друг от друга. Масштаб отображения поля зрения в зрительной коре не является постоянным. Области поля зрения с высоким разрешением (соответствуют центральной ямке) занимают большую часть поверхности зрительной коры, чем области с низким разрешением. Вместе с тем структура зрительной коры однородная и, что весьма важно, слоистая. Аксоны достигают самых внешних слоев коры (около черепа). Далее связи идут к следующим слоям в направлении от черепа и т.д. Наряду с вертикальными связями, имеются также и горизонтальные связи, которые могут быть весьма протяженными.

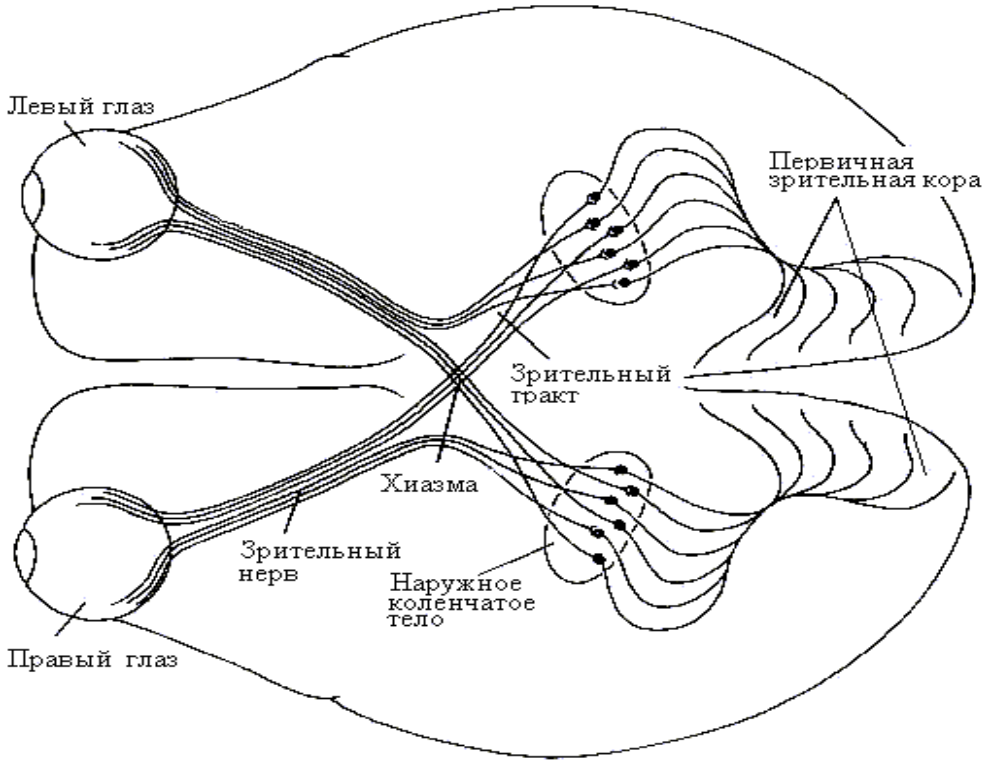


Рисунок 10.10 – Зрительные пути в мозгу человека

Нейроны зрительной коры как детекторы признаков. Если в поле зрения лабораторного животного создавать различные световые стимулы, то электрические сигналы нейронов можно обнаружить с помощью тончайших электродов, которые вводят непосредственно в нейроны или помещают рядом с ними. При объяснении реакции нейронов на световые стимулы используется концепция *рецептивного поля нейрона* [46]. В соответствии с этой концепцией каждый нейрон реагирует на стимулы в определенной области поля зрения.

Ганглиозные клетки сетчатки и нейроны коленчатого тела не обнаруживают направленности: они имеют рецептивные поля концентрической структуры. Имеются два типа таких полей – с *on-центром* и *off-центром*. Оба типа полей показаны на рисунке 10.11. Для поля с on-центром высокая интенсивность света в центре повышает частоту импульсов на выходе нейрона, а высокая интенсивность света в кольцевой области вокруг центра снижает её. При этом наибольшая частота импульсов получается для стимула с ярким пятном в центре и темной зоной вокруг. Постоянная яркость света в пределах всего поля создаёт слабую или нулевую реакцию нейрона. Стимулы с темными пятнами и ярким окружением приводят к максимальной частоте следования импульсов на выходе нейронов с рецептивным полем off-типа.

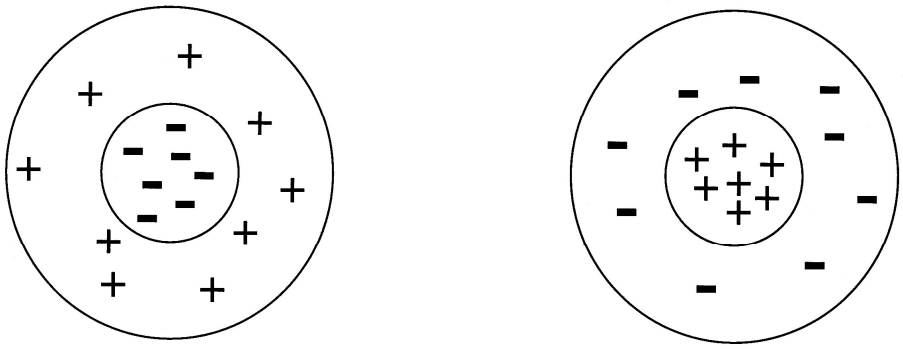


Рисунок 10.11 – Рецептивные поля с off- и on- центрами

Рассмотрим психофизический эксперимент, основанный на использовании сетки Германа и подтверждающий концепцию рецептивных полей с on- и off-центрами (рисунок 10.12).

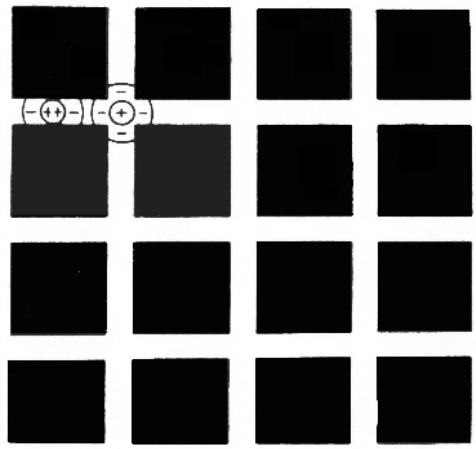


Рисунок 10.12 – Сетка Германа

Рассматривая сетку с соответствующего расстояния, можно заметить на пересечениях белых полос диффузионные пятна, которые кажутся более

темными, чем сами белые полосы. На рисунке изображены два рецептивных поля с on-центром – одно на пересечении белых полос, а другое на некотором расстоянии от места пересечения. Для рецептивного поля, находящегося на пересечении полос, наблюдается большая часть его подавляющей кольцевой области, по сравнению со вторым полем. Это и создаёт темное пятно в центре первого рецептивного поля. Противоположный эффект получается при негативном изображения сетки, т. е. сетки, состоящей из темных полос на белом фоне. В этом случае будут видны несколько более яркие пятна на пересечениях. Здесь эффект объясняется рецептивными полями с off-центрами.

Имеются нейроны с различными размерами рецептивных полей, радиусы которых могут отличаться в десятки раз.

Нейроны с рецептивными полями круговой симметрии найдены не только в сетчатке, но и в первичной зрительной коре. Имеются также нейроны с удлиненными рецептивными полями. Они относятся к классу так называемых простых клеток. Рисунок 10.13 поясняет, каким образом можно получить детектор ярких строк на основе нескольких нейронов с рецептивными полями круговой симметрии. Это, конечно, всего лишь гипотеза.

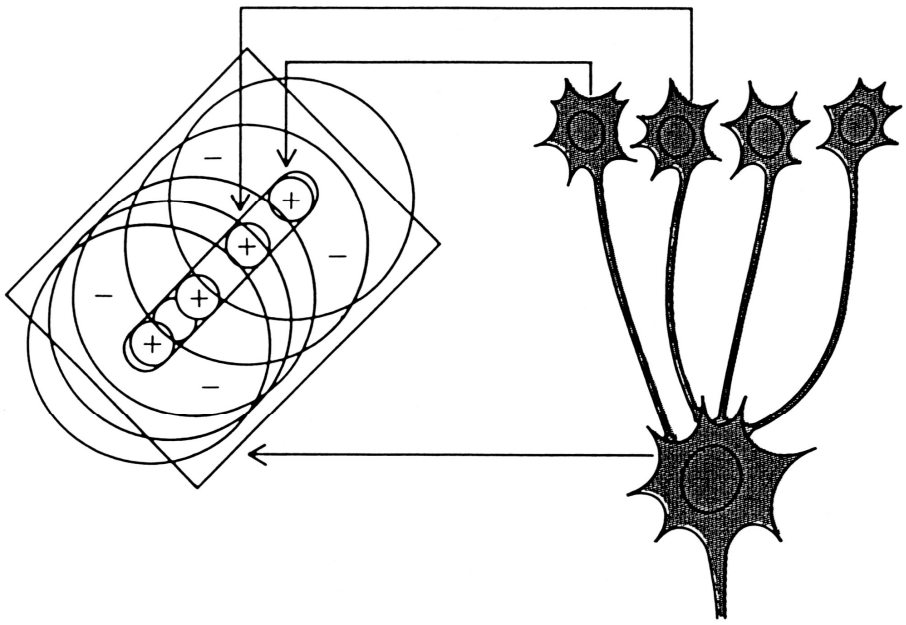


Рисунок 10.13 – Нейронный детектор ярких строк

Чтобы выяснить функции нейронов, через отверстие в черепе лабораторного животного (в большинстве случаев макак из-за подобия их мозга человеческому мозгу) в зрительную кору вводят тончайший электрод и осторожно меняют его позицию. При некоторых положениях электрода обнаруживаются электрические сигналы в ответ на визуальный стимул, создаваемый в определенных точках поля зрения. Хотя такой подход кажется

довольно несовершенным, тем не менее, он дал важные результаты. С его помощью были открыты рецептивные поля, реагирующие на яркость граней и линий, идущих в разных направлениях. Кроме простых клеток, реагирующих на направление граней или линий, имеются так называемые комплексные клетки, которые реагируют на более сложные элементы изображений. Реакция гиперкомплексных клеток зависит от присутствия концов линий в рецептивном поле. Эти клетки также реагируют на углы, имеющиеся на изображениии.

Рисунок 10.14 поясняет поведение гиперкомплексных клеток [69]. Здесь показана реакция нейрона (частота импульсов) на два различных стимула: темную линию, изображенную слева вверху, и ряд линий с выровненными концами, изображенный справа.

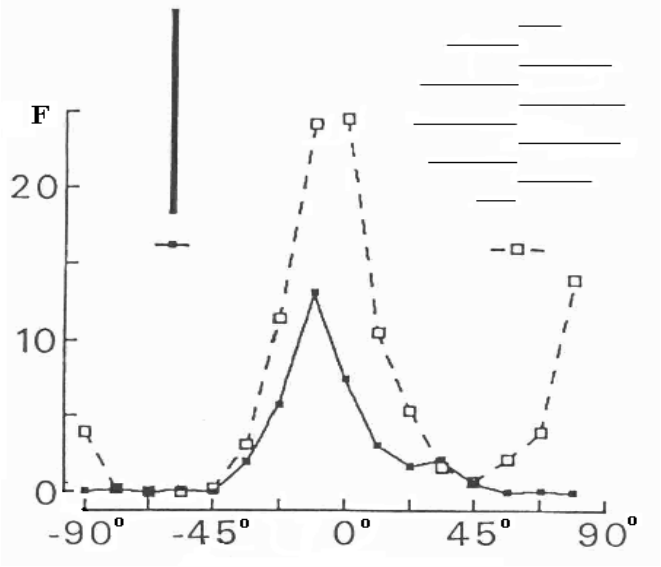


Рисунок 10.14 – Зависимость частоты импульсов от ориентации линий

Данные стимулы демонстрировались лабораторному животному под различными углами поворота. Сплошная кривая на рисунке представляет реакцию для одиночной темной линии, а штриховая – для ряда линий. Из графика следует, что наиболее сильная реакция наблюдается для одиночной линии, расположенной под углом 0 градусов (в некоторой системе отсчета). В случае отклонения этой линии влево или вправо частота снижается. При предъявлении этому же нейрону ряда линий был получен неожиданный результат: очень сильная реакция наблюдалась, когда виртуальная линия, соединяющая концы ряда линий, имела направление близкое к нулю градусов. Это означает, что нейрон реагирует на несуществующую (виртуальную) линию, которая отчетливо ощущается. Безусловно, это оптическая иллюзия, но она имеет материальное основание. Из рисунка также видно, что частота импульсов увеличивается при повороте линий, изображенных справа, на угол +90 или -90 градусов. Это реакция на от-

дельные линии. Следовательно, нейрон не делает никаких отличий между виртуальными и реальными линиями.

Рассмотренные нейрофизиологические представления оказывают огромное влияние на теоретические и прикладные исследования в области компьютерного зрения.

Другие области мозга, связанные с визуальными задачами. Зрительная кора – наиболее исследованная часть мозга, так как она относительно легко доступна. Другие части мозга исследованы в меньшей степени. Предполагается, что они имеют отношение к более высоким уровням анализа зрительной информации.

Психофизические эксперименты и оптические обманы. Некоторые функции системы зрения могут быть исследованы с помощью психофизических экспериментов. Наряду с методами позитронной эмиссионной томографии или функциональным магнитным резонансом (ФМР), посредством которых образы, возникающие в мозгу, могут быть сделаны видимыми, психофизические эксперименты являются единственным средством для изучения сложных явлений, происходящих не в зрительной коре.

Один такой эксперимент был выполнен Р. Шэпардом и Й.Мэтцлером [85]. В этом эксперименте испытуемых просили проанализировать два представления объекта, изображенных на рисунке 10.15 один над другим, и сообщить, можно ли преобразовать один объект в другой трехмерным вращением. Время между представлением изображений и выдачей ответов фиксировалось. Было обнаружено, что это время линейно зависит от трехмерного угла вращения, на который необходимо повернуть объект. Данный факт интерпретируется как доказательство того, что в зрительной системе человека углы вращения представляются аналоговыми величинами. Современные изучения этого эксперимента выполнялись с одновременной фиксацией ФМР изображения мозга, чтобы определить участок мозга, вовлеченный в процесс трансформации изображения. Результаты показывают, что первичная зрительная кора не вовлечена в этот процесс и, скорее, является областью мозга, чувствительной к движению.

Другое явление, которое нуждается в объяснении – постоянство представления внешнего мира, даже в том случае, когда мы меняем положение головы или вращаем её.

Исследование так называемых оптических обманов также сыграло определенную роль. Примерами являются:

- сетка Германа (концепция рецептивного поля, рисунок 10.12);
- треугольник Каница (виртуальные контуры, рисунок 10.16);
- куб Нэккера (рисунок 10.17).

На рисунке 10.16 в дополнение к контурам виртуального треугольника можно видеть несуществующие отличия в яркости. Виртуальный

треугольник кажется более ярким, чем фон, хотя бумага имеет всюду одинаковую степень белизны. Ведутся поиски объяснения этого факта.

Интерпретация куба Нэккера не однозначна. Очень часто наблюдатель испытывает самопроизвольные переходы от одной интерпретации к другой. Этот пример показывает, что зрение является конструктивным. Иными словами, мы часто видим то, что хотим увидеть.

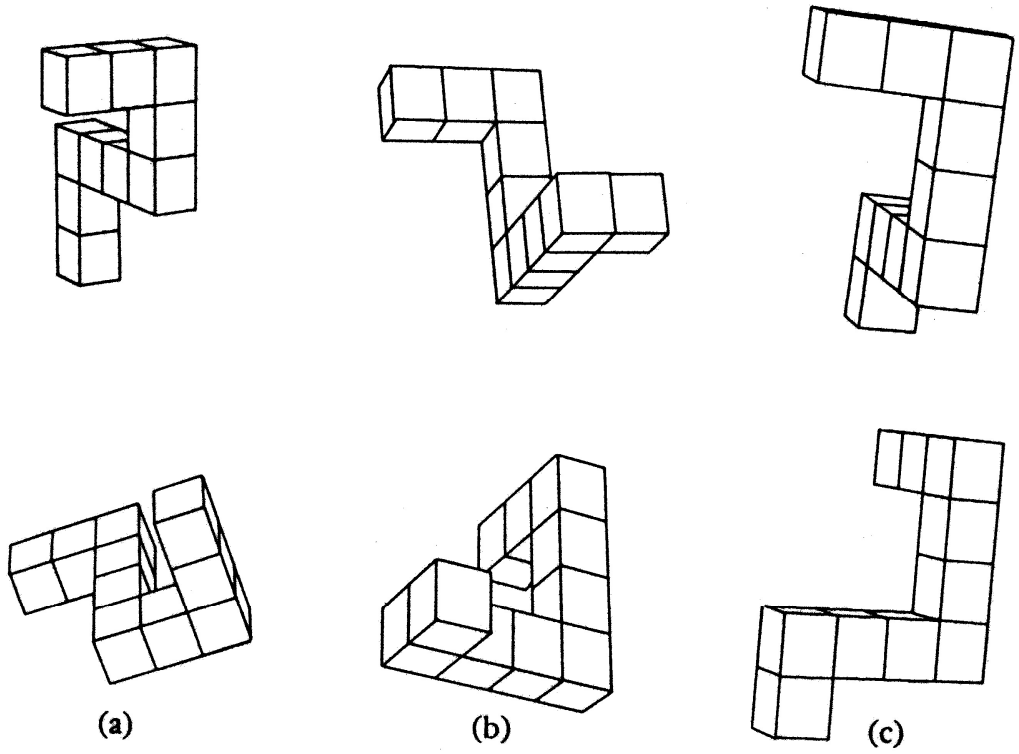


Рисунок 10.15 – Фигуры Шэпарда и Мэтцлера

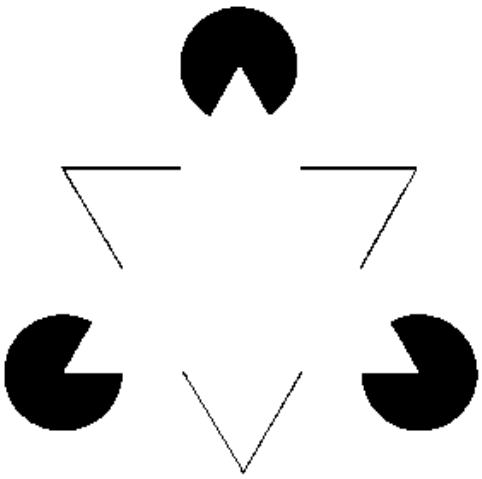


Рисунок 10.16 – Треугольник Каница

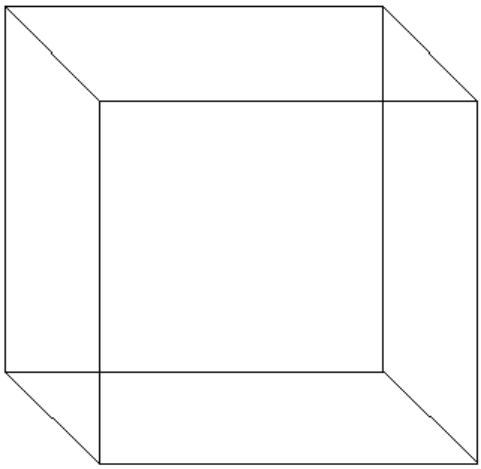


Рисунок 10.17 – Куб Нэккера

10.3. Системы компьютерного зрения

Идея создания универсального решателя задач (GPS) захватила воображение исследователей на начальном этапе развития ИИ. Затем было обнаружено, что для решения задачи требуется большой объём фактических и процедурных знаний. База знаний решателя должна быть настолько большой, что стало очевидной не возможность его реализации, а сама идея создания универсального решателя задач стала рассматриваться как утопическая. Можно задать вопрос: "Действительно ли попытка, направленная на построение универсальной системы компьютерного зрения, тоже является утопической?"

Развитием систем компьютерного зрения, которые не ориентировались бы на решение узких задач, а наоборот, удовлетворяли бы некоторым требованиям общности, занимались около трёх дюжин рабочих групп во всем мире. Хотя работы над некоторыми из таких систем продолжались примерно четверть века, нельзя утверждать, что цель создания универсальной системы была достигнута. Такие системы могут быть весьма полезными в относительно узком контексте. Несмотря на это, вновь и вновь возникают попытки создания подобных систем. Приведем краткое упоминание некоторых систем.

VISIONS. Система была разработана А. Хэнсоном и Э. Райзманом в Массачуссетском университете. Работы по созданию VISIONS были начаты в начале семидесятых годов. Эта система технического зрения, основанная на знаниях, представляла собой среду для универсальной обработки и логических выводов. Вместе с тем система решала задачи в конкретной области. Она выполняла интерпретацию изображений пригородных и дорожных сцен [67]. На основе этой системы был создана коммерческая система KBVision (Knowledge based Vision).

ACRONYM. Система была разработана Р. Бруксом в Стэнфордском университете в середине семидесятых [62]. Эта система предвосхитила важную роль 3-D моделирования. Возможности системы были продемонстрированы при интерпретации изображений аэропортов, полученных с самолетов, а также в робототехнических системах.

SIGMA. Система была разработана Т. Матсуямой и В. Нагао в университетах Сендая и Киото[79]. Работы над этой системой были начаты в конце семидесятых. Назначение системы интерпретация аэроснимков. Система использовала принцип классной доски. В ней предусматривались средства взаимодействия с оператором.

Все эти системы имели много общего: они представляли универсальную среду, в которую встраивались контекстно-зависимые модули. Предполагалось, что на более поздних этапах развития системы будут способны сами выбирать необходимые модули для решения конкретных

задач.

Несмотря на различия указанных систем, существовало определенное согласие относительно компонентов, которые должны были входить в систему компьютерного зрения, а также относительно последовательности обработки визуальных данных. На рисунке 10.18 изображена обобщенная структурная схема системы компьютерного зрения. Она соответствует структурной схеме системы SIGMA. Хотя рассматриваемые ниже системы, ориентированные на решение более узких задач, и будут проще, тем не менее, они обладают общими признаками, которые следует подчеркнуть.

Во-первых, системы являются многоуровневыми. Сложность генерируемых структур данных увеличивается при переходе на более высокие уровни системы. Структуры, сформированные на одном уровне, могут играть роль примитивов на следующих уровнях. В общем, представление структур меняется от уровня к уровню.

Во-вторых, знания, используемые в системах компьютерного зрения, включаются в модели, которые являются одновременно и простыми, и сложными. Модели могут описывать как отдельные объекты сцены, так и всю сцену. Такие модели весьма характерны для систем технического зрения. Они вводятся в систему разными способами, например, с помощью предварительного обучения.

В-третьих, несмотря на существование систем, функционирующих в мире двумерных изображений (например, системы зрения роботов), обычно предусматривается этап получения трехмерных структур по двумерным структурам с помощью различных моделей и процесса сопоставления.

В-четвертых, нет необходимости, чтобы системы работали исключительно в направлении снизу вверх (управление со стороны данных, т.е. от сигналов к символам), хотя во многих системах оно является основным. Возможность возврата к предыдущим шагам обработки часто обеспечивает разрешение неоднозначности. Активизация моделей и генерация гипотез происходит в нисходящем направлении.

В настоящее время возрождается идея построения универсальной системы компьютерного зрения в рамках японского проекта Компьютеризации Реального Мира, который является 10-летней исследовательской программой со сроком завершения в 2001[1)]. В нем реализуется новый подход. Системы компьютерного зрения, которые разрабатываются в настоящее время – это обучающиеся системы. Они не наполняются заранее всеми видами возможно невостребованных в дальнейшем знаний, а обучаются в процессе работы. Это должно привести к построению действительно “тонких” систем компьютерного зрения. В этом случае не полагаются ис-

[1)] http:/www.rwcp.or.jp/home-E.html

ключительно на логический и процедурный подходы, которым ранее отдавалось предпочтение, а опираются на концепции нейроинформатики.

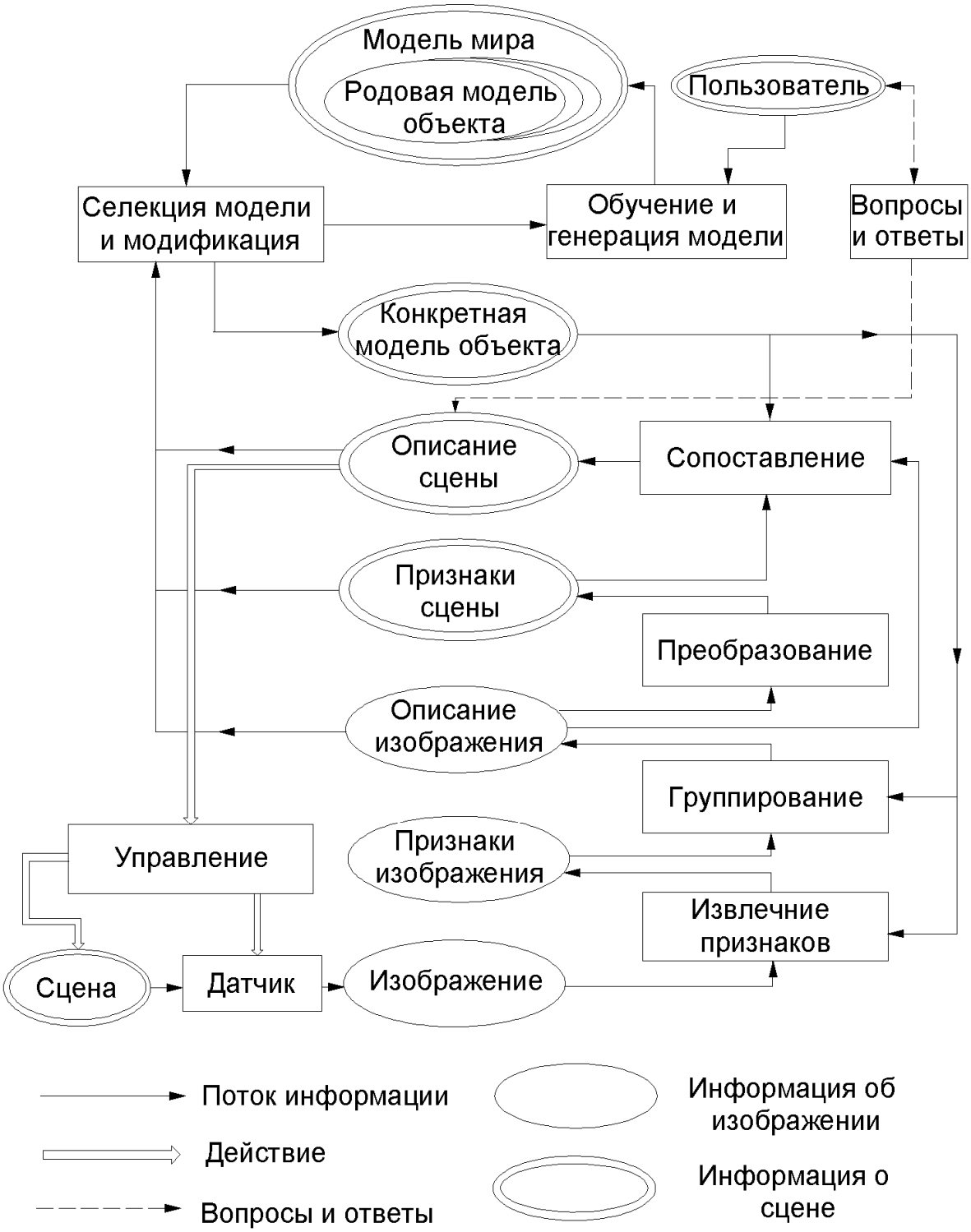


Рисунок 10.18 – Обобщенная схема системы компьютерного зрения

10.4. Выделение граничных элементов

Рассмотрим последовательность шагов преобразований, которые упоминались при описании систем компьютерного зрения. Входом для систем компьютерного зрения, также как и входом для систем зрения в живой природе, являются атомарные части изображения. Изображение представляет собой регулярный массив, в котором сохраняются отношения соседства, т.е. смежные элементы массива соответствуют элементам реальности, видимым в определенном направлении, в котором они воспринимаются как соседние. Это объясняет, почему операторы, выполняемые на первых этапах обработки изображений, применяются к множеству соседних точек. Наиболее известными из них являются дифференциальные операторы выделения границ, а также различные виды фильтров (Габора, wavelet- и т.д.), которые пока обдумываются людьми и о которых природа уже "подумала". В дальнейшем, из-за ограниченности объема учебного пособия, будем говорить в основном об операторах выделения границ и рассматривать их в качестве примеров обобщенных фильтров локальных областей изображений.

Психофизические эксперименты убедительно показали, что те локальные области изображения, которые характеризуются значительным градиентом интенсивности, являются чрезвычайно важными для визуального восприятия человека. Следующие наблюдения подтверждают это:

- нейроны, обнаруживающие границы, изобилуют на первых этапах зрительного пути;
- прерывистые движения глаза сканируют контуры при восприятии объекта (рисунок 10.19);
- человек способен идентифицировать объект на основе простого рисунка в виде контурных линий (технические чертежи, шаржи и т.п.).

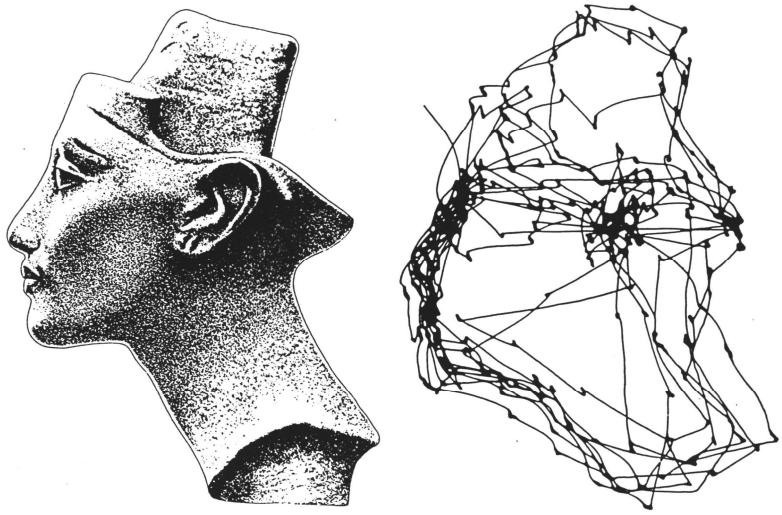


Рисунок 10.19 – Сканирование контуров при восприятии изображения

Границы на изображении обнаруживаются путем вычисления градиента интенсивности, который характеризуется:

- направлением (например, вдоль границы или перпендикулярно по отношению к ней);
- позицией;
- силой.

Контуры объекта обычно видны как некоторые границы на изображении. Поэтому естественным является стремление выделить объект по его границам. Конечно, это не всегда правильно, так как имеются и другие причины формирования границ: изменение отражающей способности или цвета поверхности объекта, тени или искажение пространственной ориентации.

Поэтому целесообразно распознавать границы не сразу, а постепенно, формируя их из меньших элементов, продвигаясь от отдельных точек, представляющих границу, к её сегментам. Вероятно, система зрения человека выполняет обработку изображений таким же пошаговым способом.

Элементом границы является пиксель (англ. edgel – дословно “элемент границы”), для которого утверждается, что он, возможно, принадлежит части контура или сегменту границы. Строго говоря, принадлежность к границе не является непосредственно свойством пикселя, а является свойством точек небольшой окрестности, в которой он расположен. Поэтому это свойство должно проверяться. Элементы границы первоначально обнаруживаются при помощи градиентных (граничных) операторов, например, операторов Собела, Кирша или Канни [6]. Они отличаются степенью подавления шумов и угловой разрешающей способностью. Данные операторы относятся к области обработки изображений и здесь не рассматриваются.

Восстановление границ. Методы восстановления границ используют информацию о геометрическом соседстве точек изображения для присвоения им согласованных меток на основе ненадежных данных, формируемых датчиками границ. В конце постобработки формируется некоторая улучшенная граница изображения, элементы которой первоначально определяются с помощью соответствующего граничного оператора.

Предположим, что для выделения элементов границы используется множество меток, которые фиксируют наличие границы и её направление. Одна из возможностей состоит в использовании кода Фримэна со значениями, лежащими в диапазоне от 0 до 7 (0 обозначает восточное направление, 1 - северо-восточное и т.д.). При этом используется некоторое соглашение о направлении границы и о положении объекта, например, слева от границы. Кроме этого, применяется специальная метка, обозначающая отсутствие границы (рисунок 10.20). Инициализация выполняется так,

чтобы с каждым элементом границы (пикселем) были связаны все 9 меток с суммарной относительной силой (вероятностью), равной 1. Распределение значений меток для каждого элемента определяется выходными значениями датчиков границ, которые настраиваются так, чтобы ни одна метка не получила нулевой вероятности с целью исключения бесконечного итерационного процесса. Начальная нормализация выполняется по отношению к датчику границ, на выходе которого формируется наибольшее значение. Детальные указания о процессе восстановления (релаксации) приведены в [71]

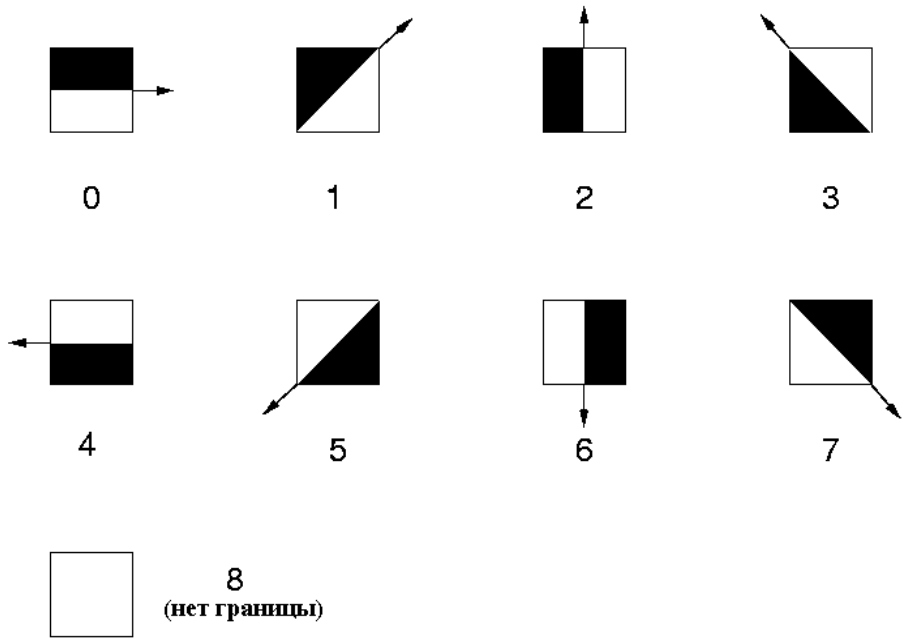


Рисунок 10.20 - Метки пикселей в соответствии с кодом Фримэна

Веса совместимости r_{ij} отражают степень усиления или взаимного ослабления меток двух элементов границ, находящихся на определенных относительных позициях. Данные веса определяются пользователем на основе здравого смысла или методом проб и ошибок. Другая возможность состоит в том, чтобы определять их статистически, в виде частоты появления определенных пар меток на изображении. Функция совместимости $q_i^{(k)}(\lambda)$ определяет, насколько необходимо уменьшить или увеличить текущую вероятность (относительную силу) метки λ для пикселя i в процессе восстановления границы. Возможная формула:

$$q_i^{(k)}(\lambda) = \sum_j d_{ij} [\sum_{\lambda'} r_{ij}(\lambda, \lambda') p_j^{(k)}(\lambda')] .$$

Здесь k – номер шага итерации. Величины r_{ij} и $q_i^{(k)}(\lambda)$ характеризуются

знаком. Массив весов d_{ij} учитывает пространственное положение пикселей i и j. Внутреннее суммирование выполняется по всем меткам пикселя j, характеризуемых вероятностями $p_j^k(\lambda')$ (относительной силой). Внешнее суммирование взвешивает вклады различных пикселей, например, согласно их расстоянию до i-го элемента границы. Обновление значений вероятностей выполняется на основе формулы:

$$p_i^{(k+1)}(\lambda) = \frac{p_i^{(k)}(\lambda)[1 + q_i^{(k)}(\lambda)]}{\sum_{\lambda} p_i^{(k)}(\lambda)[1 + q_i^{(k)}(\lambda)]}.$$

Знаменатель гарантирует, сохранение нормализации в процессе выполнения итераций. Процесс сходится грубо в течение 10 итераций. Результатом является изображение, в котором метки граничных элементов согласованы между собой лучше. Это позволяет улучшить формирование сегментов границ.

Процедура восстановления находит применение не только на первых этапах обработки систем компьютерного зрения. Всюду, где необходимо воссоздавать компоненты целого из элементов, главным образом, когда они являются смежными в пространстве, говорят о восстановлении.

В заключение отметим, что первый этап преобразования информации в системах компьютерного зрения начинается с пикселя, который характеризуется одним атрибутом – уровнем интенсивности цвета. Применение операторов выделения границ обеспечивает нахождение пикселей граничных элементов изображения с более богатым набором атрибутов, т.е. позицией, направлением границы, силой и размером.

Указанные граничные элементы остаются не связанными друг с другом и после процесса восстановления, описанного выше. Но они могут быть использованы как примитивы, чтобы формировать более сложные структуры.

Как уже отмечалось, имеются и другие возможности для выбора примитивов нижнего уровня. Позже рассмотрим другой подход к выбору примитивов – метод наращивания областей.

10.5. От граничных элементов к граничным сегментам

Система зрения человека способна группировать элементы изображения в структуры без априорных знаний содержания изображения. Эта способность называется перцептивным группированием и изучается ког-

нитивной психологией. Рисунок 10.21 иллюстрирует перцептивное группирование.

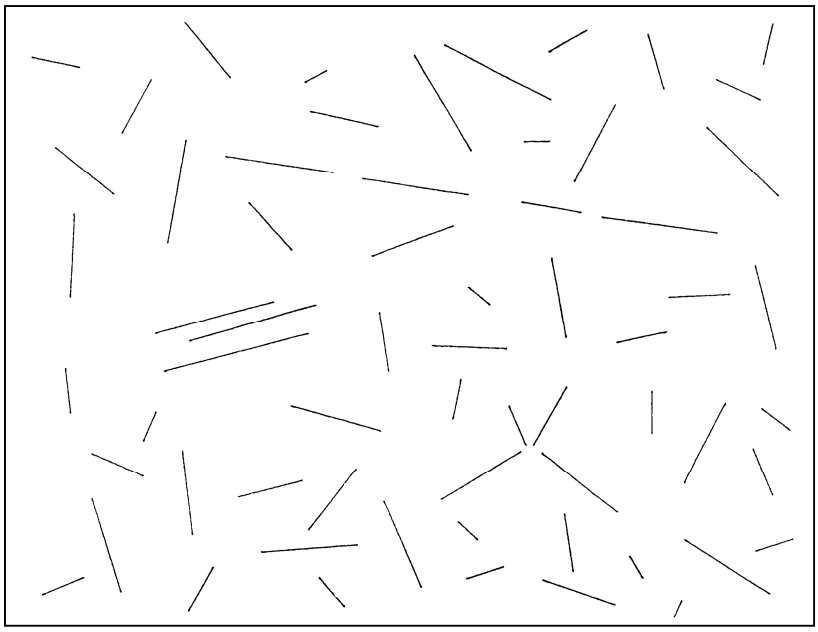


Рисунок 10.21—Перцептивное группирование

Наблюдатель не может воспрепятствовать сам себе в том, чтобы увидеть в случайном расположении линейных отрезков некоторые группы, воспринимаемые его глазами. Существуют три принципа группирования, которые следуют из этого примера, – коллинеарность, параллелизм и геометрическая близость. Благодаря указанным принципам создаётся впечатление, что отдельные элементы изображения образуют нечто целое. Они дают человеку возможность воспринимать структуры в тех случаях, когда соответствующие данные фрагментированы. Было бы весьма полезно, чтобы и системы компьютерного зрения обладали подобным свойством.

10.5.1. Преобразование Хафа

Рассмотрим преобразование Хафа как один из методов группирования, применяемый в системах компьютерного зрения. Предположим, что к изображению уже был применен некоторый граничный оператор, и на изображении сохранены для дальнейшей обработки только те граничные пиксели, для которых абсолютное значение градиента выше некоторого порога. Информация о направлении градиента в данном случае не используется. Сохраненные пиксели маркируются. Вместе с немаркированными пикселями фона они образуют бинарное изображение.

Преобразование Хафа позволяет обнаруживать на изображении кривые заданной формы (прямую линию, круг, эллипс и т.п.) даже, когда они

представлены фрагментами. Самый простой случай – обнаружение прямых линий по коллинеарным точкам, которые могут быть не связаны между собой.

Первоначально Хаф предложил следующую параметрическую формализацию этой задачи

$$y = mx + b ,$$

где *m, b* – параметры восстанавливаемой прямой линии. При этом рассматриваются два пространства: *пространство изображений и пространство параметров* (рисунок 10.22). Прямые линии пространства изображения представляются точками в пространстве параметров, а точки (или семейство прямых, проходящих через эти точки) пространства изображений представляются прямыми в пространстве параметров.

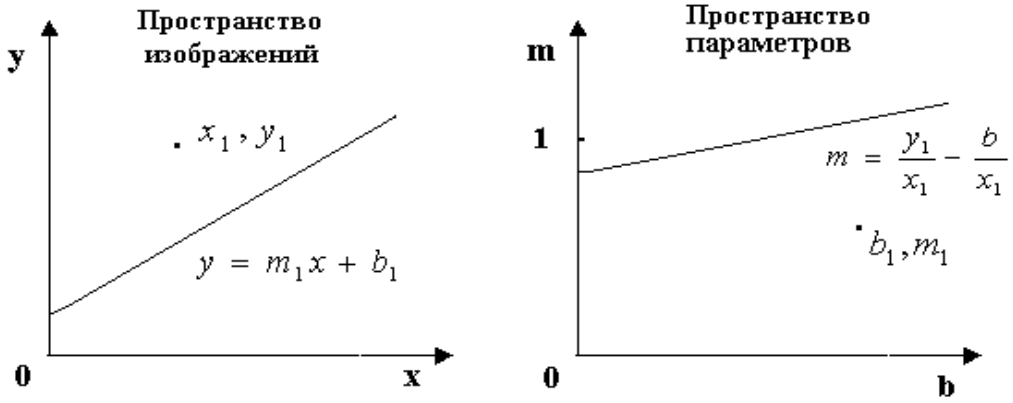


Рисунок 10.22 - Преобразование Хафа для прямых линий

Трудности возникают при построении вертикальных линий, когда $m \to \infty$. Имеется два решения этой проблемы:

- использование двух или более частных пространств параметров, т.е. $-1 \le m \le 1$ и $-1 \le 1/m \le 1$;
- использование для прямой линии другой системы параметров, например, в полярных координатах.

Уравнения прямой линии в полярных координатах (нормальная форма) имеет вид

$$\rho = x \cos\theta + y \sin\theta ,$$

где ρ – расстояние до прямой из начала координат; θ – угол наклона перпендикуляра, опущенного на прямую из начала координат; *x,y* – координаты точки, лежащей на прямой. Недостатком такого представления является потеря дуальности: теперь точке в пространстве изображений соответствует синусоида в пространстве параметров.

Применение преобразования Хафа требует дискретизации пространства параметров. Оно разбивается на аккумулирующие (суммирующие) ячейки подходящего размера. Внесение значений в эти ячейки выполняется в ходе анализа соответствующего пикселя пространства изображений. Пиксель соотносится с кривой в пространстве параметров (она представляет все прямые, проходящие через рассматриваемый пиксель в пространстве изображений). В дальнейшем инкрементируются значения тех аккумулирующих ячеек, через которые проходит кривая. Выполняя указанные действия для всех отмеченных пикселей пространства изображений, можно заметить, что некоторые из аккумулирующих ячеек получают значительные приращения (т.е. за них отдано много "голосов"), а другие – нет. Количество голосов в позициях, где наблюдаются максимумы, можно рассматривать как подтверждение того, что в пространстве изображений имеется прямая линия. Формируемая прямая имеет бесконечную протяженность. Алгоритм можно модифицировать так, чтобы пиксели пространства изображений, которые внесли вклад в обнаружение прямой (т.е. обеспечили формирование пика в пространстве параметров), запоминались. Тогда "формирующие элементы" прямой линии из пространства изображений будут известны, что позволит установить конечные точки прямой.

Рассмотрим пример обнаружения кругов на изображении. Будем опять анализировать два пространства: пространство изображений и пространство параметров, которое в данном случае трехмерное (рисунок 10.23). Уравнение окружности в пространстве изображений имеет вид:

$$(x_1 - x_c)^2 + (y_1 - y_c)^2 = r^2 .$$

В этом случае точке (x_1, y_1) пространства изображений соответствует коническая поверхность в пространстве параметров. Каждая из точек конической поверхности представляет набор параметров, которые определяют круг с центром в соответствующей точке пространства изображений. И наоборот, если рассматривается некоторый круг в пространстве изображений, то он соответствует точке с координатами (x_{c1}, y_{c1}, r_1) в пространстве параметров. Дискретизация пространства параметров выполняется аналогично случаю обнаружения прямых линий.

Необходимо отметить, что в вычислительном отношении преобразование Хафа весьма затратное для случая обнаружения кругов. Если дополнительно определять направление градиента в каждой точке пространства изображений, то можно получить более эффективный алгоритм. Это позволяет ограничить число аккумулирующих ячеек в пространстве параметров, которые требуют модификации.

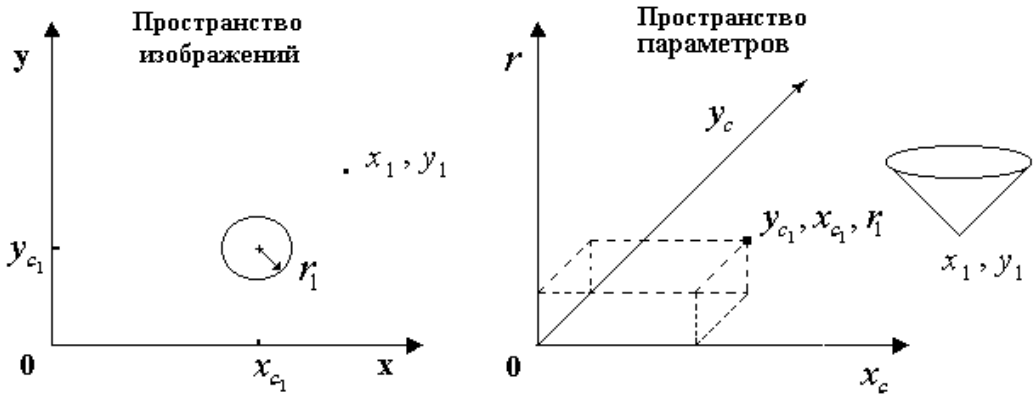


Рисунок 10.23 – Обнаружение кругов с помощью преобразования Хафа

Можно спросить, действительно ли преобразование Хафа помогает воссоздавать сложные структуры по примитивным элементам? Ответ - да. Но способ, которым это выполняется, в корне отличается от биологического подхода. Принципы перцептивного группирования – параллелизм, геометрическая близость, коллинеарность – не играют никакой роли при выполнении преобразования Хафа. Напротив, преобразование Хафа опирается на общее математическое описание, в котором участвуют все точки, принадлежащие некоторой линии. При выполнении преобразования Хафа необходимо указать для какой кривой оно применяется. В то же время система зрения человека не нуждается в подобной информации, чтобы выполнить группировку элементов.

10.5.2. Поиск при выделении контурных сегментов

В 1976г. А. Мартелли рассмотрел применение методов поиска на графах к обнаружению контурных сегментов [78]. По существу, был предложен оптимизационный подход.

Предполагается, что функция градиента вычисляется для каждого пикселя заранее. Основываясь на этой информации, необходимо найти “оптимальный“ контур. Мартелли предложил использовать для этого А*-алгоритм поиска на графах. Как отмечалось ранее, А*- алгоритм – это алгоритм поиска в пространстве состояний, который начинает поиск со стартовой вершины и выбирает наилучший (самый дешевый) путь, ведущий в целевую вершину. Управление в алгоритме осуществляется с помощью оценочной функции, содержащей две составляющие: затраты от стартовой вершины до промежуточной вершины текущего пути; прогнозируемые затраты от промежуточной вершины до целевой вершины. Сформу-

лировать подходящую оценочную функцию бывает весьма трудно, особенно её вторую часть.

Поэтому оценочная функция Мартелли содержит только оценку затрат на пути из стартовой вершины в вершину N, т.е. $\hat{g}(N)$, и не включает затраты на пути из вершины N в целевую вершины – $\hat{h}(N)$. Полагая $\hat{h}(N)=0$, получаем тривиальную версию A*-алгоритма – алгоритм равных цен, свойства которого подобны алгоритму поиска в ширину. Причины, по которым трудно сформулировать нетривиальную эвристическую функцию $\hat{h}(N)$, следующие. Во-первых, выбранная функция оценки накопленных затрат $\hat{g}(N)$ содержит два слагаемых, одно их которых зависит от силы границы (больше сила границы – ниже затраты), а другое – от кривизны контура (выше степень кривизны – больше затраты). Так как заранее ничего не известно о той области изображения, которую предстоит просмотреть на пути к целевой вершине, то трудно оценить предстоящие затраты каким-либо значением, отличным от нуля. Во-вторых, определение стартовой и целевой вершин также затруднительно. Возможность указанная Мартелли: каждый пиксель в первой строке изображения может быть использован как стартовая вершина, а каждый из пикселей в последней строке – как целевая вершина. Дополнительная возможность: в качестве стартовой вершины можно использовать пиксель с максимальным абсолютным значением градиента; в качестве конечной вершины используют тот же самый пиксель, но при достижении его с другой стороны. Данная возможность, конечно, применима только для замкнутых контуров.

Из-за этих трудностей, на практике предпочтение отдают методам с локальными эвристиками, которые, конечно, не могут гарантировать глобальной оптимальности построенного контура. Примером является подход, который использовался П.Руммелем и В. Бойтелем в системе распознавания объектов на основе полутоновых изображений в промышленных условиях [83]. Граничные пиксели непосредственно прослеживаются на изображении с помощью линии-указателя, управляемой направлением градиента. Пиксели результирующего контура представляют последовательность граничных точек, каждая из которых характеризуется позицией и направлением. Стартовая точка контура находится с помощью изотропного оператора Собела. Как только выходное значение оператора (абсолютное значение градиента) превысит некоторый порог, начинается эвристический поиск. С этого момента контур наращивается пиксель за пикселем, посредством тех пикселей, которые не приводят к слишком большому увеличению кривизны. Кроме этого, абсолютное значение градиента в позиции следующего пикселя должно быть выше некоторого порога. Данный метод использует неявные допущения о том, что контуры должны быть относительно гладкими. Если допущения не выполняются, то это оз-

начает, что разрешающая способность изображения недостаточна. Наращивание контура прекращается, когда условия, указанные выше, не выполняются (например, если объект и фон имеют приблизительно равную интенсивность), или когда текущий контур совпадает с контуром, найденным ранее, или границей изображения.

10.5.3. Активные контурные модели

Современной версией метода глобальной оптимизации выделяемого контура является так называемый метод активных контуров, или "снейков" (англ. snake – змейка, боновое заграждение в виде змейки). Активный контур представляет собой кривую на изображении, которая может менять свою форму под воздействием внутренних сил, обусловленных свойствами самой кривой, и внешних сил, направляемых данными изображения. Внешние и внутренние силы определяются так, чтобы форма активного контура соответствовала границам выделяемого объекта. Активные контурные модели ("снейки") являются виртуальными кривыми $r(s)$. С каждой такой кривой связывают значение "энергии". Поиск оптимального контура рассматривается как задача минимизации энергии E_{snake} на множестве кривых:

$$E_{snake} = \int_0^1 E_{\text{int}}(r(s))ds + \int_0^1 E_{image}(r(s))ds + \int_0^1 E_{ext}(r(s))ds .$$

Первый интеграл учитывает длину и плавность кривой; второй имеет малые значения, когда выделяемый контур проходит через области изображения с большими градиентами. Эти два интеграла, по существу, представляют дальнейшее развитие первой составляющей оценочной функции Мартелли. Наличие третьего интеграла позволяет учесть влияние элементов изображения, находящихся геометрически вне "снейка". В интерактивном режиме пользователь может оказывать влияние на выбор желаемой формы восстанавливаемого контура. В не интерактивном режиме система может учитывать цели более высокого порядка (предположения, гипотезы). "Снейки" могут представляться в виде сплайнов. Минимизация энергии выполняется итерационно и является весьма затратной. Узким местом метода активного контурного моделирования является инициализация.

Методы поиска контуров с глобальной оценочной функцией более устойчивы к шумам изображения по сравнению с методами, использующими локальную оценочную функцию. Первые в определенной степени могут соединять разрывы контуров. Для изображений с явными границами и низким уровнем шумов (что часто имеет место в системах техниче-

ского зрения роботов) обычно применяют более эффективные локальные методы.

10.6. Выделение областей изображения

Подходы, основанные на выделении областей изображения, являются дуальными по отношению к подходам, основанным на выделении контуров: контуры ограничивают области, а соприкосновения различных областей между собой создают контуры. Области являются потенциально важными для интерпретации изображений. Отдельные пиксели могут играть роль исходных точек в процессе генерации областей. Пиксель добавляется к области, когда его специфические свойства не слишком отличаются от свойств пикселей, которые уже образуют область, и когда он соседствует с областью. Это довольно неточное определение позволяет выделенные области, т.е. области, в которых интенсивность изменяется плавно, рассматривать как одну. Данный процесс, называемый "слиянием", начинается с уровня пикселей и продолжается до тех пор, пока выполняются следующие условия:

- каждая область представляет собой связное множество пикселей изображения;
- области являются непересекающимися, и вместе они охватывают всё изображение;
- каждая область удовлетворяет некоторому предикату однородности, это касается значений интенсивности, цветов, текстуры, глубины, движения, математического описания поверхности или других свойств;
- если области R_i и R_j являются смежными, то $R_i \cup R_j$ не удовлетворяет предикату однородности.

Противоположным по отношению к процессу "слияния" является процесс "разделения", который начинается с анализа всего изображения и, после проверки однородности некоторой области, выполняется рекурсивно до тех пор, пока это необходимо. Затем смежные области, удовлетворяющие предикату однородности, объединяются.

Рекомендуется использовать такой процесс "разделения и слияния", при котором изображение произвольно разделяется на области среднего размера. Затем области тестируются (с помощью предиката однородности) как с целью выяснения возможности их дальнейшего разделения, так и пригодности для слияния.

Сегментацию изображения на области можно улучшить сопоставлением получаемых результатов с результатами процесса восстановления границ, выполняемого параллельно, и дальнейшей их адекватной адапта-

цией. Такой гибридный процесс реализуется в системе машинного зрения VISIONS.

Гибридная технология сегментации также хорошо подходит для диапазонных изображений. Граничные элементы обычно обнаруживаются при поиске пикселей в окрестностях, которые характеризуются большим диапазоном отличий (разрывы в виде скачка). Другой вид разрывов соответствует неравномерностям скорости нарастания значения (разрывы в виде крыши). Пиксели, приходящиеся на разрыв любого вида, могут объединяться в линии с помощью методов, подобных тем, которые были описаны выше для изображений, представляемых матрицей интенсивности.

Элементы, соответствующие наращиваемой области изображения, обнаруживаются, начиная с некоторых исходных точек, расставляемых регулярно или рассеиваемых беспорядочно с соответствующей плотностью по диапазонному изображению. Исходные точки – маленькие зоны, составленные из нескольких смежных диапазонных пикселей. Такие зоны могут расширяться во все стороны, пока пиксели в зоне удовлетворяют некоторому математическому описанию. В зависимости от характера диапазонной сцены, даже самое простое математическое описание (уравнения плоскостей в 3D пространстве) может приводить к очень интересным результатам. Соответствующий пример будет рассмотрен при описании системы технического зрения робота.

10.7. Интерпретация контурных рисунков

Рассмотрим метод интерпретации трехмерной сцены по двумерному контурному рисунку. Это классический метод искусственного интеллекта, который был разработан независимо Хаффманом и Клоузом (1971) и расширен Вальцом (1975).

В этом случае можно говорить о том, что речь идет о системе компьютерного зрения, однако, по общему признанию, довольно неполной. Вся низкоуровневая часть системы здесь не рассматривается, но предполагается, что она формирует соответствующие признаки, на основе которых выполняются необходимые выводы. Эти признаки (линии) представляются совместно с геометрическими отношениями в форме контурных рисунков.

Чтобы выполнить интерпретацию контурного рисунка, необходимо, с одной стороны, обладать знаниями об окружающей среде и об объектах, присутствующих в ней, а с другой – учитывать весьма сильные ограничения:

- используемые контурные рисунки должны быть корректны (отсутствие разрывов линий, отсутствие свободных концов линий);

- объекты на рисунках должны быть видимы с общих позиций, т.е. малые изменения положения объекта или камеры не должны приводить к качественным изменениям сцены;
- объекты должны быть многогранниками, в каждом углу объекта пересекаются 3 плоскости.

Задача решается методом распространения ограничений. Необходимо строго различать элементы, принадлежащие двумерному изображению, и элементы соответствующей трехмерной сцены. Элементами 2D-изображения являются: линия, соединение, область. К элементам 3D-изображения относятся: край (ребро), вершина, поверхность.

На первом шаге анализа изображения выполняется предварительная классификация линий и соединений в соответствии с их графической формой. Выделяют следующие виды двумерных элементов (рисунок 10.24):

- линия – имеет вид отрезка прямой;
- L-соединение –пересечение двух линий;
- стрелка –пересечение трех линий, наибольший угол между двумя линиями превышает180^0;
- вилка –пересечение трех линий, угол между любыми двумя линиями меньше 180^0;
- Т-соединение – пересечение трех линий, один угол равен 180^0.



Рисунок 10.24 Виды двумерных соединений

На втором шаге осуществляется возможная интерпретация линий и соединений в трехмерном пространстве.

3D-интерпретация линий на рисунке. С целью интерпретации контурных рисунков вводятся следующие метки:

- выпуклые ребра маркируются знаком “+” (поверхности пересекаются так, что ребра образуют угол меньше 180^0 со стороны объекта);
- вогнутые ребра маркируются знаком “-” (поверхности пересекаются так, что ребра образуют угол больше 180^0 со стороны объекта);
- закрывающие ребра маркируются знаком “→“ (поверхности по обе стороны ребра не являются смежными в трехмерном пространстве, направление стрелки может быть выбрано таким, чтобы закрывающая поверхность находилась справа, а частично закрытая поверхность – слева от стрелки).

3D-интерпретация соединений на рисунке. Учитывая сказанное, можно выделить только четыре типа вершин в соответствии их пространственным расположением. Все они могут быть представлены с помощью определенным образом расположенных прямоугольных блоков (рисунок 10.25). Вершины каждого из четырех типов отмечены на рисунке точкой.

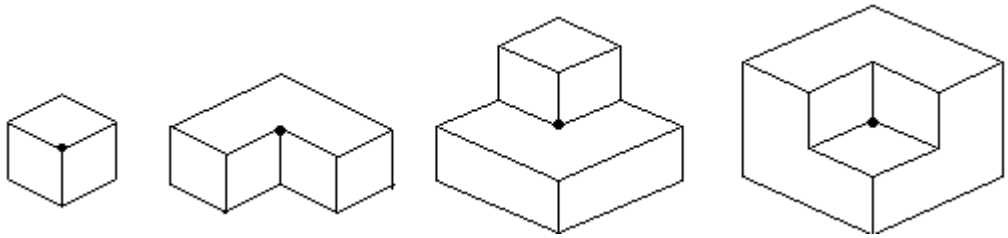


Рисунок 10.25 – Виды вершин

Множество интерпретаций соединений можно получить, исходя из четырех типов пространственного размещения блоков, рассматриваемых под разным углом зрения. Возможные виды интерпретируемых соединений показаны на рисунке 10.26. Чтобы получить интерпретации, изображенные в последней строке рисунка, необходимо наличие второго (закрывающего) объекта.

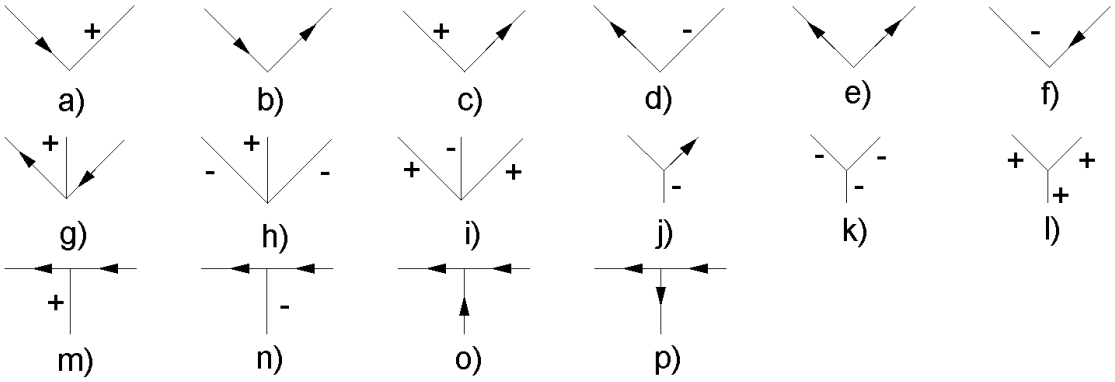


Рисунок 10.26 Виды интерпретируемых соединений

После такой предварительной подготовки можно попытаться выполнить интерпретацию конкретного рисунка, который не должен содержать ошибок (разрывов, свободных концов и т.п.). 3D-интерпретация соединений, присутствующих на рисунке, не может выполняться локально. В начале необходимо для каждого типа соединений (например, для вилки) допустить всевозможные 3D-интерпретации и представить их в форме множества меток. Интерпретация рисунка непротиворечива в том случае, когда линия, связывающая два соединения, имеет одну и ту же метку с ка-

ждой из своих сторон. Хаффман и Клоуз нашли способ непротиворечивой разметки контурных рисунков путем полного перечисления всех возможных комбинаций меток и проверкой совместимости каждого случая. Если непротиворечивая разметка не может быть найдена, то либо не выполняются предварительные условия, либо объект физически не осуществим.

Легко заметить, что проблема разметки соответствует проблеме распространения ограничений. Повысить эффективность поиска можно путем усечения дерева поиска. Это уточнение было предложено Вальцем (1975). Разметка может выполняться последовательно или параллельно. В соответствии с последовательным подходом рассматривается одно соединение за другим. При этом каждый раз исследуется непротиворечивость соседних наборов меток. Рисунок анализируется несколько раз до тех пор, пока не прекратятся замены меток. В случае параллельного алгоритма каждому соединению может быть назначен процессор, который исследует непротиворечивость наборов соседних меток и удаляет недопустимые метки.

Если поиск непротиворечивой разметки заканчивается неудачей, т.е. если алгоритм назначает две различных метки для одной и той же линии, то объект является нереальным (бессмысленным). Также установлено, что не все контурные рисунки, для которых описанным выше методом найдена непротиворечивая разметка, являются реализуемыми.

Полнее извлечь информацию из контурных рисунков можно на основе подхода, разработанного К. Сугихарой [90]. Данный подход позволяет различать реализуемые и не реализуемые объекты.

Сформулируем выводы. Несмотря на сложность, желательно, чтобы методы могли выделять объекты с криволинейными поверхностями. В этом направлении выполняются современные исследования. Проблема состоит в том, что, в отличие от мира многогранников Вальца, имеются закрывающие линии, которые не соответствуют граням тел. Эти закрывающие линии появляются, когда взгляд идет по касательной к изогнутой поверхности.

Дополнительные проблемы возникают, когда используются неполные рисунки или рисунки с ошибками.

Если обнаружена непротиворечивая глобальная разметка контурного рисунка, то это означает, что найдена интерпретация только видимой части сцены, представленной рисунком. Рассмотренный процесс интерпретации не позволяет достичь более глубокого понимания рисунка.

Человек обычно продвигается намного глубже и дополняет полученные результаты предсказанием невидимых частей сцены. При этом он обычно вводит дальнейшие предположения и принципы, например, симметрию и непрерывность.

Таким образом, интерпретация контурных рисунков – все еще актуальная область исследований компьютерного зрения и ИИ.

10.8. Распознавание объектов в системах зрения роботов

10.8.1 Системы зрения роботов и компьютерное зрение

Системы зрения промышленных роботов являются подклассом систем компьютерного зрения, но приоритеты в этом случае расставляются по иному. Ориентация на промышленное применение предопределяет следующие требования к системам технического зрения роботов:

- низкая стоимость оборудования и программного обеспечения;
- высокая скорость распознавания объектов, обеспечивающая синхронность распознавания с технологическим процессом, в который интегрирована система зрения робота (наблюдается расширение этой тенденции);
- высокая надежность распознавания объектов, их позиций и ориентации;
- высокая робастность по отношению к изменениям формы объектов, позиции и ориентации.

С другой стороны, системы зрения роботов характеризуются некоторыми упрощениями:

- малое количество распознаваемых объектов и их моделей, хранящихся в памяти (примерно от 10 до 100);
- модели объектов могут быть заданы довольно точно на основе обучения и/или на основе данных САПР;
- часто объекты имеют простую форму, которая определяется прямыми линиями, плоскими поверхностями и т.п.;
- локальные признаки, которые являются полезными для распознавания, свойственны не только одному объекту, например, просверленные отверстия;
- освещение и фон можно настроить так, чтобы упростить распознавание;
- часто объекты и/или фон могут быть окрашены или помечены;
- предъявлением объектов можно в значительной степени управлять, например, объекты могут отбираться простыми механическими устройствами, что упрощает распознавание.

Системы технического зрения роботов играют все возрастающую роль в следующих производственных задачах:

- упаковка изделий (таблетки, конфеты, колбасы и т.д.);

- осмотр поверхностей (царапины, трещины, цвет), проверка завершенности, обнаружение инородных элементов, проверка размеров;
- позиционирование, настройка, юстировка;
- распознавание, сортировка, схватывание и обработка объектов;
- управление роботами в операциях измельчения, сварки и распыления;
- чтение штрихового кода и текста (для идентификации объектов и сборок);
- сборочные операции;
- передвижение по производственным участкам и складам

Таким образом, можно вывести довольно четкое различие между системами технического зрения роботов, применяемыми в промышленности, и универсальными системами компьютерного зрения. Тем не менее, этапы преобразований, которые встречаются в универсальных системах компьютерного зрения, характерны также и для систем технического зрения роботов, хотя возможно и в сильно упрощенной форме. Эти этапы могут быть обнаружены в каждом из примеров систем, которые будут описаны далее.

10.8.2. 2D-модели

2D-модели, используемые в системах технического зрения роботов, адекватны, если объекты выглядят как плоские или если они имеют только *несколько устойчивых положений* и идентифицируются по особенностям их контура. Методы распознавания классифицируются в соответствии характером используемых признаков. При этом также учитываются различия, связанные с реализацией операции сопоставления.

Методы с глобальными признаками. Модель основана на бинарном изображении силуэта. Признаки включают глобальные свойства силуэта. Порядок следования признаков не имеет значения и их можно объединить в вектор признаков. Поэтому в ходе операции сопоставления, можно применять классические методы распознавания образов. Если объекты соприкасаются или закрывают друг друга, то из-за глобального характера признаков возможно неправильное распознавание.

Методы с локальными (структурными) признаками. Здесь признаки извлекаются из контуров объектов и образуют последовательность, упорядоченную в соответствии с их появлением. Процесс сопоставления подобен сравнению строк на основе расстояния Левенштейна как меры сходства (§ 8.2).

Методы, использующие графы отношений. Признаки, рассматриваемые в этих методах, образуют не линейную последовательность, а фор-

мируют некоторый граф. В этом случае применяются методы сопоставления графов.

10.8.3. 3D-методы

В отличие от методов, рассмотренных выше и использующих только двумерные проекции трехмерных объектов, 3D-методы учитывают форму поверхности объектов в трехмерном пространстве. Форма поверхности может быть получена по нескольким полутоновым изображениям (сложно) или определена непосредственно по точкам с помощью датчика глубины (проще). Современные недорогие датчики глубины работают при структурированном освещении. Они проецируют последовательность специальных образов, состоящих из полос, на сцену. Камера воспринимает изображения деформированных полос, по которым можно определить глубину. Такой подход обобщает традиционный метод, использующий одну световую плоскость. После определения координат (x, y, z) точек, они представляются группами в виде участков поверхности (заплат) адекватной сложности. Этот шаг обеспечивает сжатие данных, а также сглаживание и упрощение запоминания изображения.

Модели объектов, используемые в ходе распознавания, представляются в памяти компьютера в виде участков поверхности. Сопоставление выполняется непосредственным сравнением восстанавливаемой поверхности с соответствующими поверхностными участками модели или сравнением на основе дедуктивных мер. В некоторых задачах можно обойтись полностью без моделей. В этом случае достаточно иметь представление сцены в виде участков поверхностей (заплат), чтобы установить возможность схватывания объекта манипулятором робота.

10.9. Примеры методов распознавания изображений

10.9.1. 2D-методы, использующие глобальные признаки

Несколько версий промышленных систем зрения были разработаны в конце семидесятых годов в Стэнфордском научно-исследовательском институте Г. Эйджином и другими [65].

Проблема состояла в том, чтобы распознавать объекты, находящиеся в некотором рабочем пространстве или на ленте конвейера. При этом объекты не соприкасались друг с другом и не накладывались один на другой. Кроме этого, объекты были плоскими, а если какой-либо объект был

трехмерным, то он мог находиться только в одном из нескольких устойчивых положений.

С учетом с этих предварительных требований *модель объекта*, используемая в процессе распознавания, была чрезвычайно проста: для каждого объекта и для каждого его устойчивого положения определялся вектор-прототип, состоящий из подходящих признаков. Авторы использовали около 30 различных инвариантных признаков, некоторые из которых указаны ниже:

- площадь;
- периметр;
- число отверстий;
- площадь отверстий;
- площадь самых маленьких описанных прямоугольников;
- полуоси самых маленьких описанных эллипсов;
- полуоси эллипса, который (по отношению к моментам) сопоставлен некоторой фигуре;
- самое маленькое и самое большое расстояние контура от центра тяжести.

Обучение. На этапе обучения используется множество изображений объектов в каждом из возможных устойчивых положений (получаемых путем смещения и вращения). При этом накапливается информация о среднем, дисперсии или других статистических характеристиках каждого из признаков. Используются те признаки, которые обладают лучшими дискриминационными характеристиками.

Разработка классификатора. Классификация выполняется одним из двух методов: “ближайших соседей” (расстояние до прототипа) или “двоичного дерева решений”. Это зависит от стоимости определения значений признаков. Если затраты невелики, то применяется метод “ближайших соседей”. Если затраты на определение отдельных признаков значительны, то использование иерархического дерева решений может быть выгоднее, так как в этом случае может оказаться, что некоторые признаки не требуются для классификации. Построение модели, разработка классификатора и обучение выполняются автономно. Поэтому не требуется, чтобы эти шаги выполнялись за минимальное время. После завершения указанных этапов подготовки, выполняется распознавание в реальном времени. Время распознавания (или затраты на распознавание), наоборот, играет важную роль.

Распознавание. Если анализируемая “сцена” содержит несколько объектов, то сначала выполняется разметка компонентов. Затем каждый из компонентов анализируется отдельно с помощью выбранного метода классификации.

10.9.2. 2D-методы, использующие локальные признаки

Модель объекта. Модель конструируется для каждого объекта и для каждого устойчивого положения объекта. Модель состоит из последовательности прямолинейных элементов и дуг, которые Перкинс называет связанными кривыми и которые вместе описывают внешние и внутренние контуры объекта [81]. Кроме того, для каждой из связанных кривых вычисляется список свойств. Модели строятся для "благоприятных" условий, которые означают, что изображение имеет хорошую контрастность, и объекты хорошо отделены от фона.

Извлечение признаков. Извлечению "связанных кривых" предшествует применение оператора градиента к полутоновому изображению (образу). Полученные ансамбли краевых пикселей уточняются и связываются в цепочки, образуя краевой сегмент. Эти краевые сегменты локально аппроксимируются отрезками прямых линий и дугами. Затем последовательности соприкасающихся прямолинейных сегментов и дуг идентифицируются, как связанные кривые, которые весьма эффективно описываются и представляются в памяти. Этот шаг не вызывает никаких проблем для случая построения модели при благоприятных условиях, т. е. при хорошем освещении и одиночных объектах. Каждой связанной кривой приписывается список свойств, который включает в себя полную длину, полное изменение угла от начала до конца, энергию изгиба (вычисляемую через кривизну) и др.

Распознавание. Выделение признаков на стадии распознавания, в принципе, выполняется так же, как и при создании моделей. Имеется одно отличие, состоящее в том, что теперь могут быть "неблагоприятные" условия. Это означает, что связанные кривые могут быть обнаружены на заднем плане, или что связанные кривые – фрагментированы. Кроме этого, при склеивании линейных сегментов и дуг, с целью получения связанных кривых, могут ошибочно использоваться элементы разных объектов. Поэтому необходимо проверять несколько возможных комбинаций. Процедура распознавания должна обеспечивать решение подобных проблем.

Сначала формируется список всевозможных пар связанных кривых модели и изображения. Степень подобия пар может быть определена на основе списка свойств для каждой связанной кривой. При образовании пар их просматривают одну за другой в порядке их ранжирования по критерию подобия. Для каждого соединения, пытаются найти такой поворот изображения связанной кривой, который даст лучшее совпадение с моделью связанной кривой. Это сравнение предпочтительно выполнять в (θ, s)-пространстве, где θ – текущий угол контура, а s – длина дуги (рисунок 10.27).

Изображение связанной кривой (на рисунке 10.27 – “фрагмент”) перемещается вдоль модели. Когда обнаруживается хорошее совпадение с отрезком границы модели, то с целью глобальной верификации выполняется соответствующее вращение фрагмента, которое представляет собой вертикальное перемещение развертки фрагмента в положение сопоставления.

Метод Перкинса может работать с перекрывающимися объектами. Для этого часть границы объекта должна быть видимой (т. е. не скрытой другими объектами), иметь достаточную длину и содержать характерные признаки объекта (углы, круговые дуги, сегменты и т.д.).

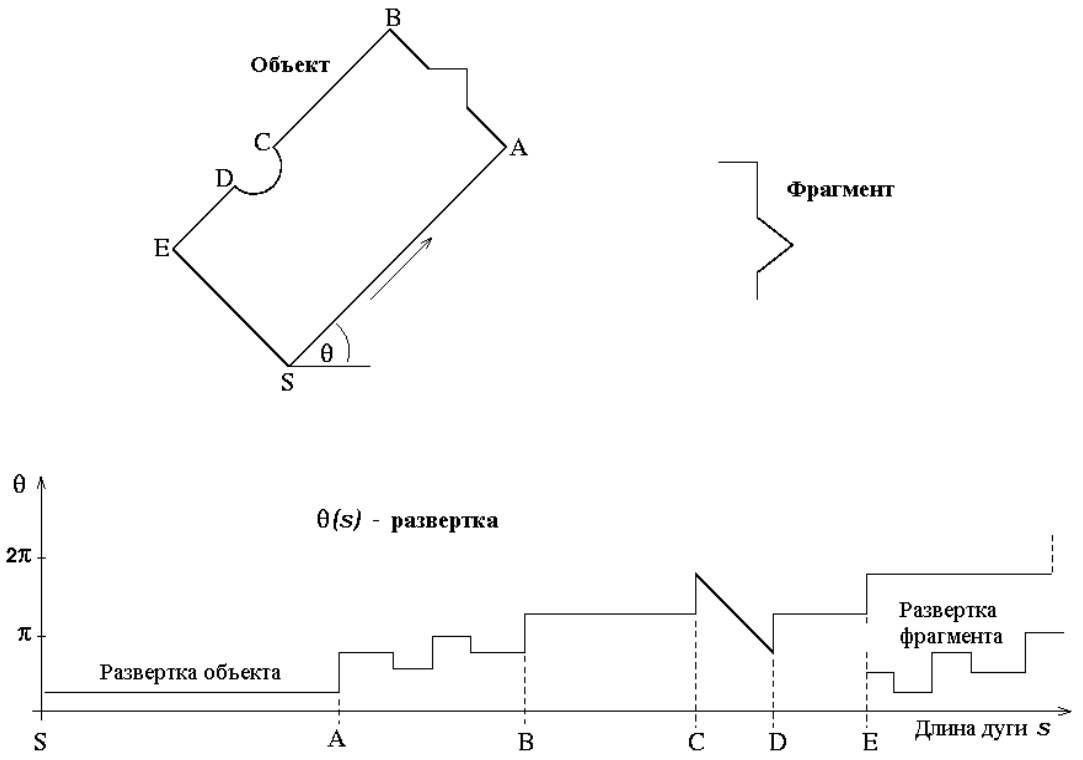


Рисунок 10.27 – Развертка объекта

10.9.3. 2D-метод, использующий граф отношений

Рассмотренный метод Перкинса базировался на использовании последовательности локальных признаков контура объекта. В методе Боллеса и Кэйна [60] используются локально обнаруживаемые внутренние признаки объектов, а также бинарные геометрические отношения между парами признаков. При этом отношения вводятся только для тех признаков, которые близки друг к другу. Это важно с точки зрения распознавания частично видимых объектов.

Боллес и Кэйн продемонстрировали их систему на примере класса объектов, представленного шарнирами, которые изображены на рисунке 10.28 вместе с соответствующими признаками. Использование графов отношений характерно для каждого аспекта метода, например, для модели объекта и её конкретизации.

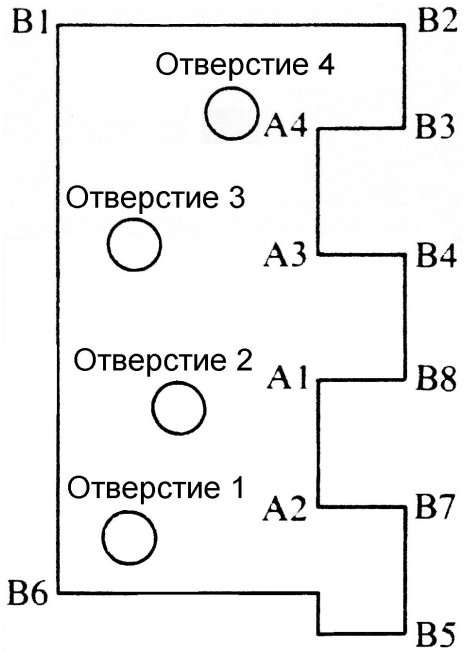


Рисунок 10.28—Схематическое изображение шарнира

Модель объекта. Модель каждого объекта состоит из трех частей. Первая часть представляет собой контур объекта, аппроксимируемый многоугольником. Вторая часть – список локальных признаков, каждый из которых описывается уникальным именем, типом (из множества типов, которые являются подходящими для задачи распознавания), позицией и ориентацией по отношению к системе координат, связанной с объектом, а также соответствующими стандартными отклонениями. Третья часть спецификации объекта позволяет выделять каждое структурно различимое образование в пределах общего класса “шарнир”. Это характерно для каждого определенного типа признака, например, тип “отверстие” охватывает все возможные конкретизации этого признака во всех моделях шарнира, допустимых в некоторой задаче. Каждый возможный признак может выступать в роли “фокусирующего признака”. На рисунке 10.29 изображен фокусирующий признак, подходящий для распознавания части объекта, на примере шарнира. Данный признак вместе с ближайшими признаками и с учетом их взаимного положения позволяет генерировать хорошие гипотезы относительно экземпляра рассматриваемого объекта. С каждым структурно различимым образованием, соответствующим заданному признаку, связано описание в виде подграфа, содержащее достаточные “вторичные

признаки” (и их относительные положения), чтобы определить позицию и ориентацию объекта.

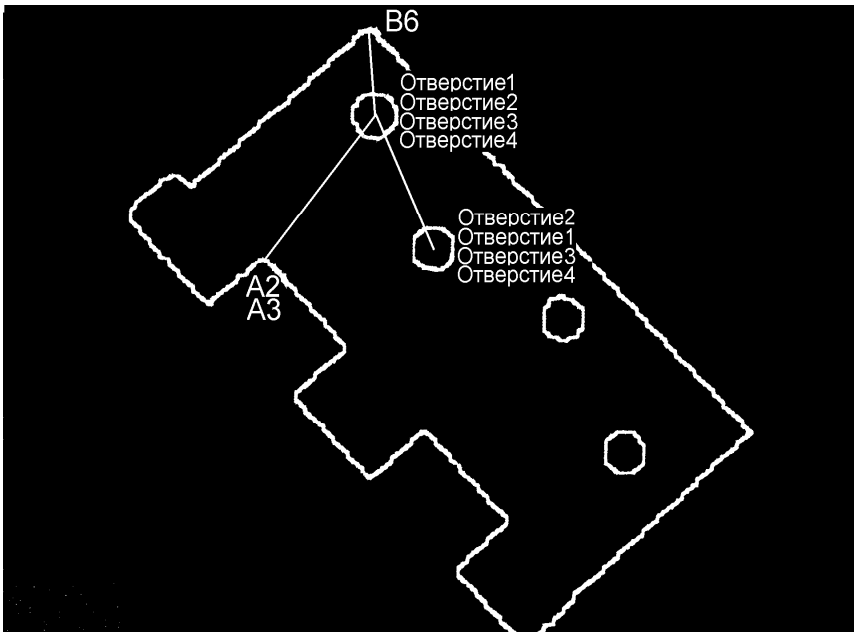


Рисунок 10.29. – Фокусирующий признак

Признаки. Отверстия на бинарном изображении находятся методом поиска областей, углы обнаруживаются с помощью пар соединенных прямолинейных сегментов. Такие соединенные сегменты перемещаются (скользят) по контуру, позволяя обнаруживать углы в тех позициях, где угол между двумя сегментами имеет экстремальное значение.

Стратегия. Распознавание выполняется на основе стратегии, которая разрабатывается заранее. К моменту завершения распознавания информация, содержащаяся неявно в описании объекта, преобразуется в форму, которая является адекватной для процедурного подхода. Для того чтобы начать процесс распознавания, используются знания относительно того, какие из признаков являются более подходящими (в данном случае – отверстия). Информация о координатах отдельных признаков преобразуется в расстояние, информация об ориентации признаков по отношению к системе координат, центрированной относительно объекта, преобразуется в информацию об их относительной ориентации. Некоторые признаки, являющиеся особо значимыми, совместно с другими признаками, близкими к ним, объявляются “фокусирующими признаками”. Фокусирующие и вторичные признаки, близкие к ним, запоминаются в упорядоченных списках, которые определяют порядок обработки.

Распознавание. Известно, что отверстия – наиболее значимые признаки (по крайней мере, для многих индустриальных сцен). Поэтому на изображении сначала отыскиваются отверстия. Как только одно из них найдено, начинается поиск других признаков, которые расположены близко к этому отверстию. В этот момент мы знаем только тип найденных

признаков, но не их отличительные особенности. Система устанавливает и обрабатывает множество меток для каждого из найденных признаков, которые совместимы со сведениями, полученными к этому моменту. Затем формируется список возможных соответствий. Этот список может быть представлен графом, в котором узлы представляют возможные признаки, а ребра объединяют признаки, совместимые друг с другом. На рисунке 10.30 изображен пример графа соответствий признаков объекта. При этом несколько соответствий конкурируют между собой за лучшую гипотезу объекта. Алгоритм находит самый большой кластер соответствий, совместимых друг с другом. Данный алгоритм называют *алгоритмом максимальной группировки*.

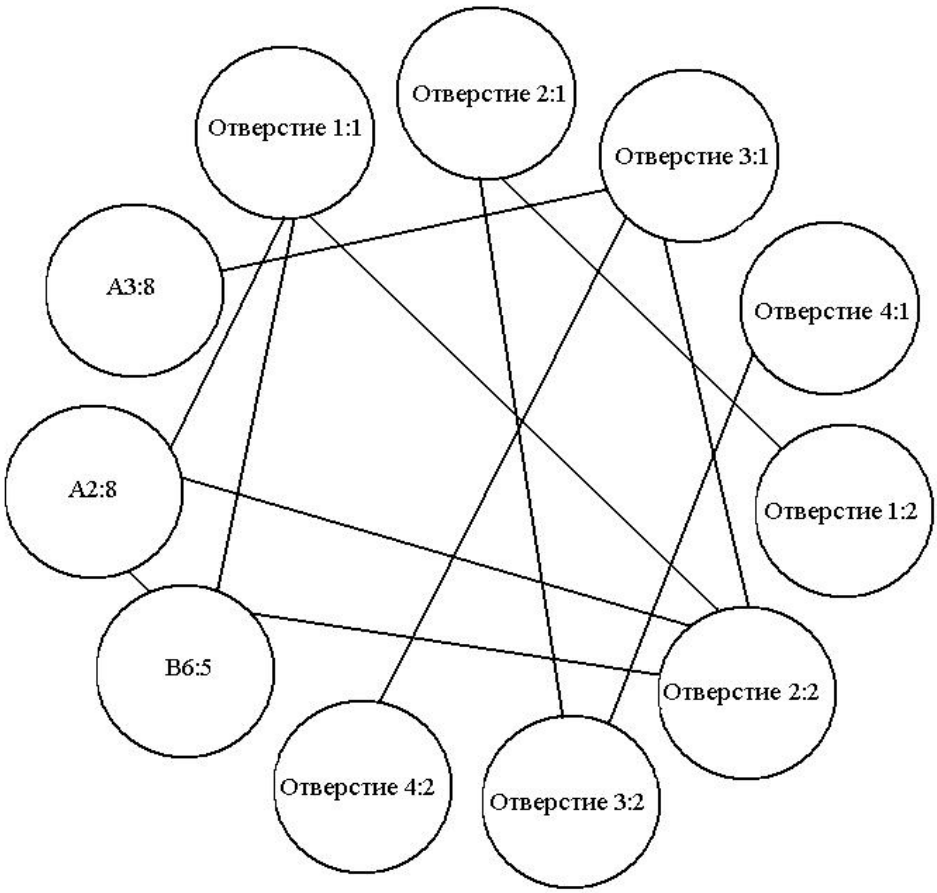


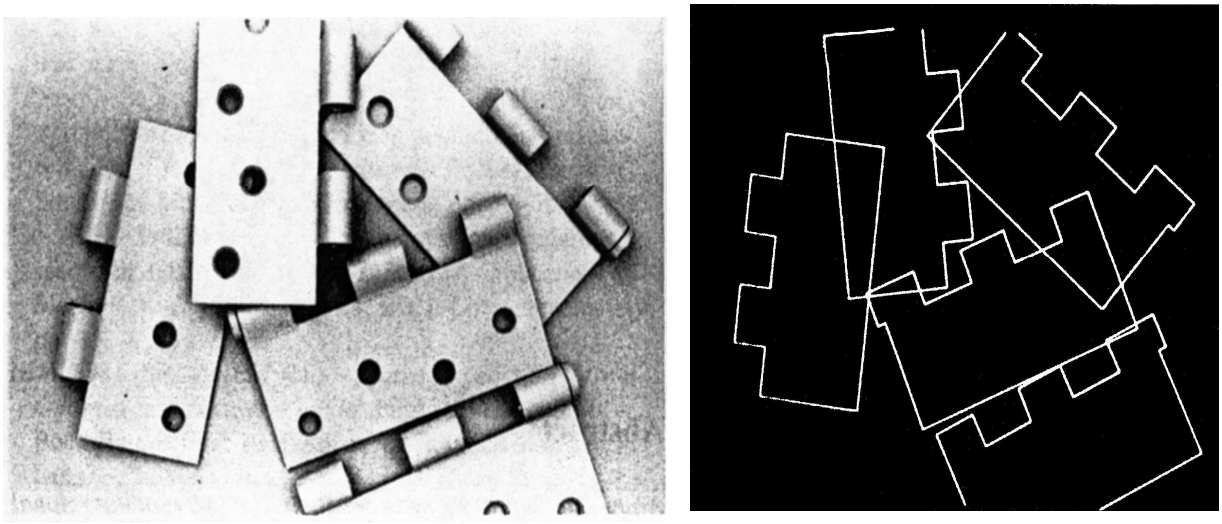
Рисунок 10.30. – Граф соответствий

После этого становится возможной генерация гипотез относительно вида объекта, его ориентации и позиции. Для этого выполняются два проверочных шага:

- дальнейший поиск локальных признаков, позиция и ориентация которых могут быть предсказаны достаточно точно;
- преобразование построенного контура объекта в изображение в позиции, предсказанной в соответствии с гипотезой.

При этом в вертикальном направлении к контуру выполняется небольшое проверочное сечение. Аналогичная процедура используется в методе Перкинса (§10.9.2). Перемещаясь из внутренней области объекта к множеству внешних точек объекта, обнаруживают переход “темное – светлое”. Если имеет место перекрытие объектов, то обнаруживается комбинация “темное – темное”. Обнаружение двух других возможностей (из оставшихся) свидетельствует против гипотезы. После подтверждения гипотезы, используемые локальные признаки отмечаются как объясненные. Они не будут использоваться больше в поиске других объектов.

Метод Боллеса и Кэйна может обрабатывать частично перекрывающиеся объекты при условии, что видимые части объектов, т. е. не скрытые другими объектами, являются достаточно большими и содержат необходимые признаки (отверстия, углы, и т.д.), чтобы сделать попытку сопоставления результативной. На рисунках 9.31,а и 9.31,б показаны примеры успешного распознавания различных экземпляров перекрывающихся шарниров, демонстрирующие потенциальные возможности метода. Результаты распознавания могут в дальнейшем использоваться при управлении манипулятором робота.



а) б)

Рисунок 10.31. – Результаты распознавания

10.9.4. Пример 3D-метода

Как следует из примеров систем, рассмотренных в предыдущих параграфах, методы анализа изображений для роботов, манипулирующих объектами, разрабатывались в течение многих лет. Большинство из методов основано на визуальных данных, т. е. данных, зафиксированных камерами, которые затем интерпретируются с помощью моделей объектов са-

мой различной природы и используются для управления действиями робота, например, схватывания распознанных объектов.

Иным стимулом для развития методов анализа изображений стало появление новых типов датчиков и, следовательно, новых видов данных, например данных о глубине. Рассмотрим один из методов, использующий данные о глубине. В этом случае используется весьма простая модель объекта, и основное внимание уделяется действиям, которые могут выполняться системой. Это ближе к процессам, выполняемым естественным интеллектом. Рассмотрим систему, основанную на использовании данных датчиков глубины, разработанную в Швейцарском Федеральном Технологическом Институте [93] .

Сначала опишем входные данные, на которых базируется метод. Рассмотрим метод световой плоскости, позволяющий получать трехмерные данные. Для этого используется тонкий "световой лист", который проецируется под некоторым углом к сцене и фиксируется камерой с различных направлений. Координаты изображения определяются в точке деформации проекции на основе уравнения прямой линии в трехмерном пространстве, проведенной к этой освещенной точке. Вместе с известным уравнением световой плоскости в трехмерном пространстве это позволяет вычислить x, y и z координаты точки. Без углубления в детали ясно, что этот базовый метод имеет несколько недостатков, которые необходимо устранять. Одна из идей предполагает отказ от использования центра световой плоскости в ходе обработки. Вместо этого лучше использовать границу между тенью и светом, которая определяется намного проще. Имеется большое число усовершенствований основного метода (и устройств). Современные датчики глубины имеют 1024 проекционных направления, реализуемых последовательностью изображений с полосами, проецируемыми одно за другим. Для этого в датчик глубины входит проектор полос, который последовательно во времени проецирует образы с полосами на сцену. Имеется также камера, фиксирующая сцену в направлении соответствующего проектора полосы и регистрирующая образы с полосами последовательно во времени. Для определения глубины изображения требуется ансамбль из n образов с полосами, например n=9 . Каждый образ с полосами характеризуется определенной шириной полос и их положением. Метод очень сложен, так как он использует встроенное кодирование направления проецирования. Для каждой точки сцены код направления можно определить по чередованию черного и белого цветов, которые фиксируются при последовательной подаче образов, что делает возможным решение задачи идентификации различных проекционных плоскостей. Это условие обеспечивает вычисление трехмерных координат точки освещенной проектором датчика глубины и в то же самое время фиксируемой его камерой.

Результаты измерений можно представить в виде трехмерного облака точек, принадлежащих одному и тому же виду объекта. Другой связи между точками не имеется. Данная ситуация такая же, как в случае анализа двумерных изображений, где промежуточный результат – изображение, представляемое совокупностью граничных точек без взаимосвязи между ними. В конечном счете, на основе полученной информации должны быть сформированы общие знания об обычных граничных связях точки в естественных сценах, т. е. предположения о гладкости и углах на ребрах объектов. В двумерном пространстве мы можем перейти от облака точек, соответствующего границе, к “связанным кривым” как в методе Перкинса. Очень похожим способом, когда имеют дело с трехмерными облаками точек, принимают эвристические предположения относительно гладкости и 3D-ребрах, которые позволяет объединять трехмерные точки в поверхностные заплаты. Этому посвящено много исследований. В рассматриваемой системе используется сегментация “облаков” точек на плоские заплаты, что требует минимальных предположений относительно объектов, с которыми имеют дело, и позволит в конце выполнить схватывание объектов. В системе используется парадигма “восстановление и выбор”, предложенная и разработанная А. Леонардисом и др.[75]. Данная парадигма предполагает использование для сегментации метода наращивания областей. Она основывается на множестве решений независимого наращивания исходных точек роста и выбора области в конце этого процесса. При этом используется подходящее пороговое расстояние точек-кандидатов до плоской заплаты, которое уже получено в ходе наращивания области.

На рисунке 10.32 изображена сцена с объектами, отличающимися как по форме, так по свойствам поверхности. Плоские поверхности построены правильно, кривые поверхности – разбиты на множество плоских заплат в соответствии с подходящим допуском. Это позволяет сегментировать сферические и иные поверхности и представлять их в виде множества плоских заплат.

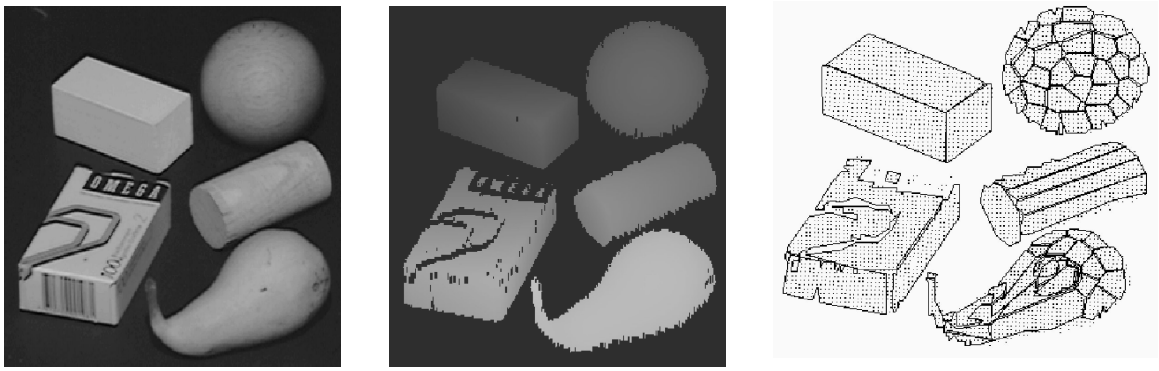


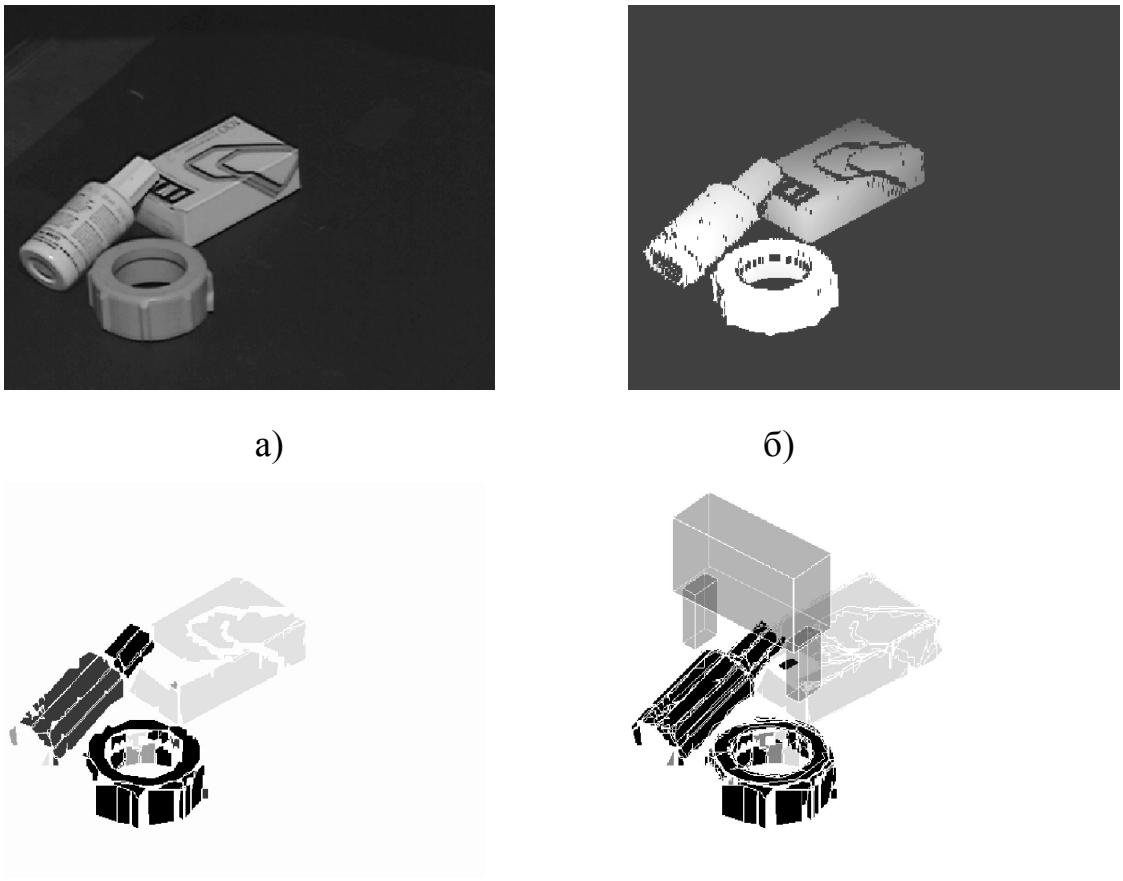
Рисунок 10.32 – Сцена с разнородными объектами

Так как конечная цель системы состоит в удалении объектов из кучи, то вида сцены только с одной стороны явно не достаточно, если не имеется никаких моделей объектов. Поэтому в работе [93] было решено, фиксировать трехмерные детали кучи объектов тремя датчиками глубины, которые размещались симметрично вокруг рабочей области и которые могли "видеть", что находится снизу.

Все датчики глубины были калиброваны во внешней системе координат робота с целью интеграции данных от трех источников. Были получены три диапазонных вида для одной и той же сцены. Сначала множества 3D-точек каждого из видов аппроксимировались набором плоских поверхностных заплат. Затем отдельно для каждого из трех видов выполнялась группировка заплат с целью построения гипотезы объектов данного вида. При этом использовалось предположение о существовании выпуклых ребер объекта между заплатами. Данное предположение подсказывает тот факт, что обе заплаты принадлежат одному объекту, в то время как существование вогнутых ребер объекта – подсказка, что смежные заплаты принадлежат различным объектам. На следующем шаге результаты, полученные для трех отдельных видов, объединяются, т.е. гипотезы объектов, построенные для каждого из трех видов, обобщались с помощью подходящей эвристики. В результате этого шага были построены глобальные гипотезы объектов, т. е. множество заплат, которые, вероятно, принадлежат одному объекту. Глобальные гипотезы объектов обычно подтверждаются на бóльшем числе плоских заплат, чем гипотезы объектов отдельного вида, и они обычно имеют бóльшее пространственное протяжение.

Каждая пара глобальных гипотез (плоских заплат) объекта может быть проанализирована, чтобы установить, имеется ли возможность схватывания объекта манипулятором робота с двумя пальцами. Это означает, что устанавливается, имеются ли две плоские заплаты, которые расположены напротив друг друга, приблизительно параллельны и не слишком смещены относительно центра объекта и т.п. Найденные возможности схватывания затем оцениваются согласно некоторой качественной мере, и лучшие сообщаются роботу, который пытается выполнить схватывание и удалить объект. После этого весь процесс повторяется, начиная со сбора данных, потому что удаление объекта может вызвать изменения в сцене. Таким способом объекты удаляются один за другим, пока куча не исчезнет. Потенциальные возможности метода представлены на рисунке 10.33. Рисунок 10.33,а является изображением с уровнями серого цвета. Рисунок 10.33,б – соответствующее диапазонное изображение, полученное с помощью датчиков глубины. Рисунок 10.33,в представляет отдельные гипотезы объектов, построенные с помощью одного вида. На рисунке 10.33,г изображены глобальные гипотезы объектов совместно с манипулятором, полученные на основе трех отдельных гипотез. Анализ множества гло-

бальных гипотез объекта показывает, что имеется возможность схватывания. При этом манипулятор достаточно удален пространстве и нет риска столкновения.



а) б)

в) г)

Рисунок 10.33 – Распознавание объектов с помощью датчиков глубины

Так как метод базируется на эвристиках, то могут случаться ошибки. Система должна иметь возможность предупреждать столкновения и находить пути их обхода, например, с помощью осторожного перемешивания кучи.

Создание рассмотренной системы было весьма полезно для выяснения проблем, которые возникают, если не использовать модели объектов.

Вопросы для самопроверки

1. Сформулируйте цель исследований в области компьютерного зрения.
2. По каким причинам задачи, относящиеся к области компьютерного зрения, считаются сложными?
3. Объясните схему получения стерео изображений.
4. Объясните строение глаза человека.
5. Какие виды предобработки визуальной информации выполняются в глазу?
6. Объясните схему зрительных путей.
7. Что означает понятие “перцептивное поле”?

8. Объясните зрительный эффект, связанный с сеткой Германа.
9. Приведите краткую характеристику известных универсальных систем компьютерного зрения.
10. Объясните обобщенную схему системы компьютерного зрения.
11. Какие операторы применяются для первоначального выделения границ.
12. Что называют элементом границы?
13. Для чего применяется код Фримэна?
14. Объясните суть метода восстановления границ на основе согласованной разметки граничных элементов.
15. Назовите основные принципы перцептивного группирования, используемые человеком.
16. Сформулируйте преобразование Хафа для прямой линии.
17. Как применить А-алгоритм для выделения контурных сегментов?
18. Объясните суть метода наращивания областей.
19. Сформулируйте основную идею активного моделирования контуров.
20. Как выполняется интерпретация контурных рисунков?
21. Какими отличительными особенностями характеризуются системы зрения роботов?
22. На какие группы подразделяются 2D-методы распознавания изображений в робототехнике?
23. Сформулируйте особенности 2D-методов, использующих глобальные признаки.
24. Сформулируйте особенности 2D-методов, использующих локальные признаки.
25. Сформулируйте особенности 2D-методов, использующих граф отношений.
26. Какая идея лежит в основе построения датчиков глубины?
27. Объясните основные принципы распознавания 3D-объектов.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

REFERENCES

1. Александров В.В. ЭВМ видит мир/ В.В. Александров, М.Д. Горский.– Л.:Машиностроение. Ленингр. отд-ние, 1990. – 129с.

2. Алиев Р.А. Производственные системы с искусственным интеллектом / Р.А. Алиев, Н.М. Абинеев, М.М. Шахназаров. – М.: Радио и связь, 1990. – 264с.

3. Бондарев В.Н. Адаптивное частотно-импульсное моделирование в задачах цифровой обработки сигналов/ В.Н. Бондарев // Вестн. СевГТУ: Сб. науч. тр. – Севастополь, 1999. – Вып. 18 – С.46 – 51.

4. Бондарев В.Н. Среда для интеллектуальной цифровой обработки сигналов/ В.Н. Бондарев, Д.В. Аведов // Вестн. СевГТУ: Сб. науч. тр. – Севастополь, 1997. – Вып. 5 – С.12 – 16.

5. Бондарев В.Н. Информационная технология построения математических моделей объектов/ В.Н. Бондарев, Д.В. Виленчик, С.В. Гольденберг // Электронное моделирование. – 1992. - №1 - С.31 – 40.

6. Бондарев В.Н. Цифровая обработка сигналов: методы и средства / В.Н. Бондарев, Г.Трёстер, В.С. Чернега. – Харьков: Изд-во “Конус”, 2001. – 398с.

7. Братко И. Программирование на языке Пролог для искусственного интеллекта/ И. Братко. - М.:Мир,1990. – 560с.

8. Вагин В.Н. Дедукция и обобщение в системах принятия решений / В.Н. Вагин. – М.: Наука,1988.-384с.

9. Вапник В.Н. Теория распознавания образов (статистические проблемы управления)/ В.Н. Вапник, А.Я. Червоненкис. – М.: Наука, 1974.- 416с.

10. Васильев В.И. Проблема обучения распознаванию образов / В.И. Васильев. – К.: Высш. шк. Головное изд-во,1989. – 64с.

11. Вейценбаум Дж. Возможности вычислительных машин и человеческий разум. От суждений к вычислениям/ Дж. Вейценбаум; Под ред. А.Л. Горелика. – М.: Радио и связь, 1982. – 368с.

12. Гаврилова Т.А. Базы знаний интеллектуальных систем / Т.А. Гаврилова, В.Ф. Хорошевский. – СПб: Питер, 2000. – 384с.

13. Гладун В.П. Процессы формирования новых знаний / В.П. Гладун. – София: СД “Педагог 6”,1994. – 192с.

14. Гладун В.П. Эвристический поиск в сложных средах / В.П. Гладун. – К.: Наук. думка, 1977. – 166с.

15. Горелик А.Л. Методы распознавания: Учеб. пособие для вузов/А.Л. Горелик, В.А. Скрипник. – 3-е изд., перераб. и доп. – М.: Высш. шк.,1989. – 232с.

16. Диалоговые системы и представление знаний: Справочное пособие / Л.В.Кокорева, О.Л. Перевозчикова, Е.Л. Ющенко. – К.: Наук. думка, 1992. – 448с.

17. Доорс Дж. ПРОЛОГ –язык программирования будущего/ Дж. Доорс, А.Р. Рейблен, С. Вацера. – М.: Финансы и статистика,1990. – 142с.

18. Ефимов Е.И. Решатели интеллектуальных задач/ Е.И. Ефимов. – М.: Наука, 1982. – 320с.

19. Информационные технологии в испытаниях сложных объектов: методы и средства / В.И.Скурихин, В.Т.Квачев, Ю.Р.Валькман; Отв. ред. В.Н. Ягипко. – К.: Наук. думка, 1990. – 320с.

20. Искусственный интеллект [В 3-х кн.]. – Кн.1. Системы общения и экспертные системы: Справочник / Под. ред. Э.В. Попова. – М.: Радио и связь, 1990. – 464с.

21. Искусственный интеллект [В 3-х кн.] .– Кн.2. Модели и методы: Справочник / Под. ред. Д.А. Поспелова. – М.: Радио и связь, 1990. – 304с.

22. Классификация и кластер / Ред. Дж. Вэн Райзин; Пер. с англ. П.П. Кольцова; Под ред. Ю.И. Журавлева. – М.: Мир,1980. – 389с.

23. Косарев Ю.А. Естественная форма диалога с ЭВМ / Ю.А. Косарев. – Л.: Машиностроение. Ленингр. отд-ние , 1989. – 143с.

24. Кофман А. Введение в теорию нечетких множеств/ А. Кофман. – М.: Радио и связь, 1982. – 432с.

25. Кузнецов О.П. Дискретная математика для инженера /О.П. Кузнецов, Г.М. Адельсон-Бельский. – М.: Энергоатомиздат,1988. – 480с.

26. Кузин Л.Т. Основы кибернетики [В 2-х т.]. – Т.2. Основы кибернетических моделей: Учеб. пособие для вузов / Л.Т. Кузин – М.: Энергия, 1979. –584с.

27. Логический подход к искусственному интеллекту : от классической логики к логическому программированию / Пер. с франц. А.Тейз , П. Грибомон, Ж. Луи . – М.: Мир ,1990.- 432с.

28. Мичи Д. Компьютер-творец / Д.Мичи, Р. Джонсон; Предисл. Д.А. Поспелова. – М.: Мир, 1987. – 255с.

29. Назаретов В.М. Техническая имитация интеллекта: Учеб. пособие для втузов / В.М. Назаретов, Д.П. Ким; Под ред. И.М. Макарова. – М.: Высш. шк., 1986. – 144с.

30. Нильсон Н. Принципы искусственного интеллекта / Н. Нильсон – М.:Радио и связь,1985. – 376с.

31. Обработка нечеткой информации в системах принятия решений / А.Н. Борисов, А.В. Алексеев , Г.В. Меркурьев . – М.: Радио и связь, 1989. – 304с.

32. Огнев И.В. Интеллектуальные системы ассоциативной памяти/ И.В. Огнев, В.В. Борисов. – М.: Радио и связь 1996. – 176с.

33. Осуга С. Обработка знаний / С. Осуга ;Пер. с япон. – М.: Мир , 1989. –293с.

34. Перспективы развития вычислительной техники [В 11 кн.]: Справ. пособие / Под ред. Ю.М. Смирнова. Кн. 2. Интеллектуализация ЭВМ / Е.С. Кузин, А.Н. Ройтман, И.Б. Фоминых. – М.:Высш. шк., 1989. – 159с.

35. Попов Э.В. Алгоритмические основы интеллектуальных роботов и искусственного интеллекта/ Э.В.Попов , Г.Р.Фридман. –М.: Наука, 1976. – 456с.

36. Поспелов Д.А. Ситуационное управление: теория и практика/ Д.А. Поспелов. – М.: Наука, 1986. – 288с.

37. Представление и исследование знаний / Под ред. Х. Уэно, М. Исидзука; Пер. с япон. – М.: Мир 1989. – 220с.

38. Приобретение знаний / Под ред. С. Осуги, Ю. Саэки ; Пер. с япон. – М.: Мир, 1990. – 304с.

39. Программирование на языке R-Лисп/ А.П. Крюков, А.Я. Родионов, А.Ю. Таранов. – М.: Радио и связь, 1991. –192с.

40. Прэтт У. Цифровая обработка изображений / У. Прэтт;Пер. с англ. – М.: Мир, 1982. – Кн. 1. – 312с.

41. Тарасов В.Б. О системно-организационном подходе в искусственном интеллекте/ В.Б. Тарасов // VI Междунар. конференция "Знания – диалог – решения": Сб. науч. тр. – Ялта,1997.- С.57-70.

42. Тарасов В.Б. Нечеткие модели в обработке изображений: обзор зарубежных достижений/ В.Б.Тарасов, С.Ю.Желтов, А.А. Степанов // Новости искусственного интеллекта. – №3. –1993. – С.40-65

43. Ту Дж. Принципы распознавания образов / Дж. Ту, Р. Гонсалес. – М.: Мир, 1978. – 411с.

44. Файке Р., Нильсон Н. Система STRIPS – новый подход к применению методов доказательства теорем при решении задач / Р. Файке, Н. Нильсон // Интегральные роботы: Сб. ст.;Пер. с англ.; Под ред. Г.Е. Поздняка. - М.: Мир,1973. – 421с.

45. Фу К. Робототехника / К. Фу , Р. Гонсалес, К. Ли; Пер. с англ. – М.: Мир, 1989. – 624с.

46. Хант. Э. Искусственный интеллект / Э.Хант; Пер. с англ.; Под ред. В.Л. Стефанюка. – М.: Мир, 1978. – 558с.

47. Хьюбел Д. Глаз, мозг, зрение / Д. Хьюбел; Пер. с англ. – М.: Мир, 1990. – 239с.

48. Хювёнен Э. Мир Лиспа [В 2-х т.]. – Т.1 Введение в язык Лисп и функциональное программирование / Э.Хювёнен, Й. Сеппянен ; Пер. с финск. – М.: Мир 1990. – 447с.

49. Хювёнен Э. Мир Лиспа [В 2-х т.]. – Т.2 Методы и системы программирования / Э.Хювёнен, Й. Сеппянен; Пер. с финск. – М.: Мир 1990. – 319с.

50. Цыпкин Я.З. Основы теории обучающихся систем/ Я.З. Цыпкин. – М.: Наука ,1970. –252с.

51. Чернега В.С. Искусственные нейронные сети в системе контроля уплотнителей насосного оборудования/ В.С. Чернега, В.Н. Бондарев, Б. Флепп //Вестн. СевГТУ: Сб. науч. тр. – Севастополь , 1999. – Вып. 18. – С.62 – 66.

52. Шевченко А.И. Интеллектуальная система нового поколения / А.И. Шевченко // Искусственный интеллект. – №1.– 1996. – С.15-25

53. Экспертные системы и логическое программирование / Л.А. Бакаев, А.А. Гриценко. – К.: Наук. думка, 1992. – 220с.

54. Элти Дж. Экспертные системы: концепции и примеры / Дж. Элти, М. Кумбе ; Пер. с англ. – М.: Финансы и статистика , 1987. – 191с.

55. Эндрю А. Искусственный интеллект / А. Эндрю; Пер. с англ.; Под ред. и с предисл. Д.А. Поспелова. – М.: Мир, 1985. – 264с.

56. Agin G.J. Computer vision systems for industrial inspection and assembly/ G.J. Agin // Computer. – 1980. – №5. – P.11 – 20.

57. Allen J. AI Growing up/ J.Allen // AI MAGAZINE. – 1998. – V.19. – №4. – P.13 – 23.

58. Andrews P. B. TPS: A theorem proving system for classical type theory/ P.B. Andrews, M. Bishop, I. Sunil // Journal of Automated Reasoning. –1996. – V. 16(3). – P.321-353.

59. Androutsopoulos I. Natural Language Intefaces to Databases: An Infroduction/ I. Androutsopoulos I., G.D. Ritchie, P. Tranisch. – Edinburgh: University of Edinburgh. – 1994. – Research Paper N. 709. –50 p.

60. Bolles R.C. Recognizing and locating partially visible objects. The local-feature-focus method/ R.C.Bolles, R.A.Cain // Int. J. Robotics Res. –1982. – V.3. – N1. –P.57-82.

61. Brill E. An overview of Empirical Natural Language processing/ E.Brill, R.L. Mooney// AI MAGAZINE. – 1997. – V.18. – №4. – P.13 – 23.

62. Brooks R. Symbolic reasoning among 3-dimensional models and 2-dimensional images/ R. Brooks // Artificial Intelligence. – 1981. – V.17. – P.285–349.

63. Chilukuri K.M. Frontiers of expert system: reasoning with limited knowledge / K.M. Chilukuri. – Norwell: Kluwer Academic Publishes, 2000. – 297p.

64. Forgy C.L. RETE: A fast algorithm for many pattern many object pattern match problem/ C.L. Forgy // Artificial Intelligence. – 1982. – №19. – P.17 – 37.

65. Gleason G.J. A modular vision system for sensor-controlled manipulation and inspection/ G.J.Gleason, G.J.Agin // Proceedings of the 9th International Symposium on Industrial Robotics. – Washington D.C.,1979. –P. 57-70.

66. Gordon M.J. HOL: A proof generating system for higher-order logic/ M.J. Gordon // VLSI Specification, Verification and Synthesis/ G. Birtwistle, P.A. Subrahmanyam editors. – Kluwer,1988. – 292p.

67. Hanson A. The VISIONS image understanding system/ A.Hanson, E. Riseman // in Advances in computer vision / edited by Christopher Brown. – New Jersey: Lawrence Erlbaum Associates. – 1989. –P.1 – 115.

68. Hertz J. Introduction to the theory of neural computation/ J. Hertz, A. Krogh, R.G. Palmer. – Redwood City: Addison-Wesley Publishing Company,1996. – 327p.

69. Heydt R. (Von der Heydt R) Illusory contours and cortical neuron responses/ R. Heydt, E. Peterhans, G. Baumgartner // Science. –1984. –N 224 . – P.1260–1262

70. Huet G. A unification algorithm for typed lambda calculus/ G. Huet // Theoretical Computer Science. – 1975. – N 1. – P. 27-57

71. Hummel R.A. On the Foundations of Relaxation Labeling Processes/ R.A. Hummel, S.W. Zucker // Trans. IEEE Pattern Anal. Machine Intell. – 1983. – V.5(3) . – P.267-287

72. Hutter H.P. Comparison of Classic and Hybrid HMM Approaches to Speech Recognition over Telephone Lines/ H.P. Hutter. – Zürich: ETH. – 1997. – 236 p.

73. Kohonen T. Self-organizing Maps/ T.Kohonen . – Berlin: Springer-Verlag,1995. – 363p.

74. Kung S.Y. Digital Neural Networks/ S.Y.Kung. – Engewood Cliffs, New Jersey: PTR Prentice Hall, 1994. – 418p.

75. Leonardis A. Segmentation of Range Images as the Search for Geometric Parametric Models/ A. Leonardis, A. Gupta, R. Bajcsy // International Journal of Computer Vision. – 1995. – V.14. – P.253 –277.

76. Lin C.T. Neural fuzzy systems: a neuro – fuzzy synergism to intelligent systems/ C.T. Lin, C.S.G.Lee. – Upper Saddle Rever, New Jersey: Prentice Hall PTR, 1997. – 786p.

77. Luger G.F. Artificial intelligence: Structures and Strategies for complex problem solving/ G.F.Luger, W.A.Stubblefield – 2-nd ed. – Redwood: The Benjamin Publishing Company Inc.,1993. – 738p.

78. Martelli A. An application of Heuristic Search Methods to Edge and Contour Detection/ A. Martelli // Communication of the ACM. – 1976. – V.19. - №2. – P.73-83

79. Matsuyma T. SIGMA: a knowledge-base aerial image understanding system/ T. Matsuyma, V.S. Hwang. – N. Y.: Plenum Press, 1990. – 278p.

80. Mori R. Statistical Methods for Automatic Speech Recognition/ R. Mori //Speech processing, recognition and artificial neural networks : Proceedings of the 3-d International School on Neural Nets “Eduardo R. Caianiello”. – L.: Springer-Verlag, 1999. – P.165 - 189.

81. Perkins W.A. A model based vision system for industrial parts/ W.A. Perkins // IEEE Transactions On Computers. – 1978. – V.27. – №2 – P.126–143.

82. Pulsed neural networks/ by W. Maas and C.M. Bishop eds.—MIT Press. – 1999. – 408p.

83. Rummel P. Workpiece Recognition and Inspection by a Model-based Scene Analysis System/ P. Rummel, W.Beutel // Pattern Recognition. – 1984. – V.17. – №1. – P.141-148.

84. Russell S.J. Artificial intelligence: a modern approach/ S.J. Russell, P. Norvig. – Upper Saddle River, New Jersey: Prentice-Hall Inc.,1995. – 905p.

85. Shepard R.N. Mental rotation of three-dimensional objects/ R.N. Shepard, J. Metzler // Science. – 1971. – V. 171. – P. 701–703.

86. Shirai Y. Artificial intelligence/ Y. Shirai. – Salisbury: Jojn Wiley & Sons Ltd., 1984. – 173p.

87. Spillman L. Visual perception: the neurophysiological foundations/ L. Spillman, J.S.Werner. – San Diego: Academic Press, 1989. – 466p.

88. Stolcke A. Linguistic Knowledge and Empirical Methods in Speech Recognition/ A. Stolcke // AI MAGAZINE. – 1997. – V.18 – №4 – P.25 – 31.

89. Sycara P. K. Multiagent Systems/ P. K. Sycara // AI MAGAZINE. – 1998. – V.19. – №2. – P.79-93.

90. Sugihara K. Machine interpretation of line drawings/ K. Sugihara. – Cambridge: MIT Presss,1986. –233p.

91. Tanimoto S.I. The elements of artificial intelligence: An Introduction using LISP / S.I. Tanimoto . –Rockwille: Computer Science Press,1987. –530p.

92. Traber C. SVOX: The Implementation of a Text - to - Speech System for German/ C. Traber. – Zurich: ETH, 1995. – 234p.

93. Trobina M. From Planar Patches to Grasps: A 3-D Robot Vision System Handling Unmodeled Objects/ M.Trobina . – Zurich:ETH,1995. – Diss.No. 11326. – 124p.

94. Weld D.S. The role of Intelligent Systems in the National Information Infrastructure/ D.S.Weld, J.Marks, D.G.Bobrow // AI MAGAZINE. – 1995. –V.16. - №3 – P.45-64.

95. Winston P.H. Artificial Intelligence/ Winston P.H. – 3-rd ed. – N.Y.:Addison-Wesley, 1992. – 737p.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

А-алгоритм 52
Агент 108
Агент 21
Адаптивный линейный элемент 441
Аксон 429
Активная контурная модель 583
Акцентная группа 550
Алгоритм
– ID3 205
– ISODATA 420
– RETE 384
– Баума-Уэлча 541
– Витерби 507
– динамического программирования 419
– исключения понятий кандидатов 201
– К средних 420
– максимальной группировки 597
– Митчелла 362
– обратного распространения ошибки 446
– подъема на гору 49
– равных цен 47
Аллофон 519, 544
Анонимная переменная 313
Антецедент 80
Антропоморфизм 14
Ассоциативный список 247
Аттрактор 473

Биграмная модель 527
Бинарное дерево 345
Биологический нейрон, 430

Вершины
– тупиковые 40
– заключительные 40
Вокализованный звук 522
Вопрос
– типа “как” 375
– типа “почему” 375
Вывод
– немонотонный 185
– обратный 100
– правдоподобный 186
– прямой 99
– формулы 79
Выполнимая формула 89
Высказывание 79

Ганглиозная клетка 564
Гарантирующий алгоритм 52
Генерация высказываний 490
Гипотеза компактности 415
Грамматика 484
– СКС-грамматика 508
– контекстно-зависимая 485
– контекстно-свободная 485
– регулярная 485
Граф
– опровержения 137
– решения 40
– состояний 37
– концептуальный 111

Двойная импликация 80
Дедукция 25
Декларативная модель 75
Дендрит 429
Диалог 489

Дизъюнкт Хорна 159
Динамическая переменная 233
Дискурс 490
Дифон 547
Доказательство теорем 22, 23

Законы де Моргана 82, 88
Замкнутая формула 87
Знание 74
Зрительная кора 565
Зрительный нерв 564

Извлечение знаний 379
И-ИЛИ граф 40
Импульсный НЭ 431
Индуктивное формирование понятий 193
Индукция 25, 193
Инженер по знаниям 25, 376
Интеллектуальная система 28
Интенсионал 110
Интерпретация формулы 81, 88
Интонация 549
Искусственные нейронные сети 429
Искусственный интеллект 13, 14, 15

Квантор 87
Кепстральный анализ 523
Кластер 524
Клоз 132
Когнитолог 25
Колбочка 563
Комбинаторный взрыв 45
Композиция
– отношений 181
– подстановок 138
Конкатенативный синтез речи 547
Коннекционистские модели 429
Консеквент 80
Коэффициент уверенности 170
Куб Нэккера 570

Латеральное торможение 436
Лексическая переменная 232
Лингвистическая переменная 177
Линейная
– резолюция 142
– сепарабельная задача 439
Линейный
– пороговый элемент 433
– предиктивный анализ 523
Литерал 132
Логика
– высокого порядка, 93
– лингвистическая 182
– многозначная 182
– нечеткая 181
Логические модели, 75
Логическое следование 82, 89
ЛПК-синтез 545
Лучевой поиск 57
Лямбда -выражение 227

Макровызов 217
Макролитера 266
Макрорасширение 280
Макрос чтения 265
Макросы 280
Марковская цепь 529
Марковские предположения 507
Машина вывода 24, 373
Метапеременная 330
Метаусловие 330
Метод
– анализа конечных значений 153
– резолюций 17
– Перкинса 594
Методы “слепого” поиска 46
Механизм разделения весов 467
Многослойный персептрон 443
Модальный оператор 187
Модели
– представления знаний 23
– Коннекционистские 429
Модель 74
– произношения слова 534
– фонемы 534
– языка 526
Модифицированное дерево доказательства 144
Монотонность 85

Наиболее общий унификатор 139

Недетерминированный предикат 333
Нейрокибернетика 16
Нейроподобный элемент 431
Неокогнитрон 464
Неполнота знаний 185
Непротиворечивость 84
Нечеткая переменная, 177
Нечеткое множество 175
Нечеткое отношение 179

Область интерпретации 88
Оболочка экспертной системы 374
Образ 96, 414
Обратная блокировка вычислений 266
Обучение
– без учителя 437, 451
– с учителем 437
Общезначимость 81
Объект 108
– агрегатный 104
– индивидный 104
– обобщенный 104
Ограничения целостности 58
Оператор
– постфиксный 319
– префиксный 319
– Собела 582
Опровержение 131
Опровержимость 89
Отношение правдоподобия 166, 424
Отношения лингвистические 108
Оценочная функция 49

Падежная рамка108
Палочки 563
Память
– автоассоциативная 467
– ассоциативная 467
– гетероассоциативная 467
Параллельная рекурсия 242
Параметр
– ключевой 229
– необязательный 228
– обязательный 228
– остаточный 228
Переобучение 449
Персептрон 17
Перцептивное группирование 577
Подстановка 138
Подход
– рациональный 494
– эмпирический 494
Поиск
– в глубину 46, 348
– в пространстве версий 362
– в сети фреймов 367
– в ширину 46, 354
– комбинированный 56
– обратный 56
– по пересечению 107
– решений в И/ИЛИ графах 360
– с ограничениями 39, 58
– эвристический 356
– управляемый данными 56
– управляемый целью 56
Полуразрешимая ФС 92
Понимание речи 518
Понимание ЕЯ-высказываний 490, 493
Понимание изображений 558
Понятие 193
– конъюнктивное 193
– охватывающее 200
ППФ исчисления предикатов 87
Правило
– векторного квантования 457
– заключения 84
– подстановки 84
– Уидроу-Хоффа 442
– экзистенциального обобщения, 90
– экзистенциональной конкретизации 90
– дельта-правило обучения 443
– максимального правдоподобия 422
– максимума апостериорной вероятности 422
– модифицированное Кохонена 457
– обобщенное дельта-правило 446
– обучения Кохонена 456
– обучения Хебба 451
– Ойя 452
– Сэнгера 453
– универсальной конкретизации 89
Правильно построенные формулы 78
Предикат 86
Предложение 132
Предложения Хорна, 84

Предметная переменная 86
Предметная постоянная 86
Представление знаний 75
Преобразование Хафа 578
Признак образа 414
Принцип резолюции 136
Приобретение знаний 379
Продукционная модель 24, 76, 94
Прокликтики 551
Пропозициональный граф 40
Просодические характеристики речи 551
Просодия речи 545
Простой персептрон 438
Пространство
– версий 199
– изображений 579
– параметров, 579
– подзадач 22
– признаков 415
– состояний 22, 35
Процедурная модель 75
Процедуры-демоны 119
Псевдофизические логики 93

Равносильность формул 82
Разрешимая ФС 79
Раскрытие вершины 39
Распознавание образов 27
Распознавание речи 517
Распознавания частей речи 506
Распространение ограничений 58
Расстояние
– Левенштейна 417
– Махаланобиса 417
– Хемминга 417
Расширенные сети переходов 495, 497
Редукция 39
Резольвента 137
Рецептивное поле нейрона 566
Решатель задач 34
Решающая функция 416
Родовая функция 276
РСП-анализатор 497

Самоорганизующиеся сети 438
Самоорганизующиеся карты признаков 459
Сведение задач к подзадачам 39
Свертка 181
Свободная переменная 87
Свойство символа 248
Связывание 231
Семантическая сеть 18, 76, 104
Сетевая функция 432
Сетка Германа 570
Сеть
– Nettalk 450
– АРТ 461
– Хэмминга 467
– Элмана 553
– переходов 495
– состязательная 453
– Хопфилда 471
Синапс 429
Синтагма 550
Синтез речи по тексту 544
Система искусственного интеллекта 28
Скрытая марковская модель 530
Слот 117
Случайный поиск 46
Снейк 583
Согласованная разметка 58
Сома 429
Сопоставление 315
Специальные переменная 233
Список
– предшествований 122
– закрытых вершин 45
– открытых вершин 45
Средний риск 422
Статическая переменная 232
Степень разветвления 57
Стохастическая аппроксимация 427
Стратегия
– входных данных 142
– опорного множества 142
– унитарности 141

Таблица истинности 80
Тавтология 81
Теорема о полноте 84, 91
Теория

– адаптивного резонанса 461
– рекурсивных функций 210
– с умолчаниями 188
Терм 86
Тест Тьюринга 14
Топологическая шкала 94
Точечная пара 214
Транзитивность 110
Треугольник Каница 570
Триграмная модель 527

Универсальный решатель задач 16
Универсум Эрбрана 135
Унификатор 138
Условие монотонности 55
Условия применимости 39

Фактор
– достаточности 167
– необходимости 167
Фацет 120
Фонема 519
Форма языка Лисп 217
Формальная система 78
Форманта 523
Формантный синтезатор 546
Формирование знаний 381
Фраза 549
Фрейм 19, 77, 117
Фрейм-пример 118
Фрейм-прототип 118
Фрикативный звук 522
Фундаментальный пример дизъюнкта 135
Функтор 313
Функционал 260
– отображающий 262
– применяющий 262
Функциональная блокировка 261
Функция правдоподобия 422
Функция преобразования в ИНС 432

Цикл с возвратом 333

Чанк 96
Частичное сопоставление 383
Частота основного тона 521
Частота форманты 546

Эвристика 16
Эвристический поиск 49
Эквиваленция 80
Экземпляр фрейма 118
Экспертная система 18, 24, 371
Экстенсионал отношения 110
Элемент границы 575
Элементарная формула 88
Энергетическая функция 472
Энкликтика 551

Учебное издание

Бондарев Владимир Николаевич,
Аде Франк Германович

Искусственный интеллект

Bondarev V.N., Ade F.G.

Artificial Intelligence

Научный редактор: Маригодов В.К., д-р техн.наук, профессор

Сдано в набор 26.05.2002. Подписано в печать с оригинал-макета 17.07.2002г.
Формат 60х90 1/16.Бумага офсетная N2. Печать офсетная. Гарнитура Times.
Усл.печ.л. 38.
Издательство «СевНТУ», Украина, 99053,Севастополь, Студгородок, НМЦ.

Отпечатано ЧП «Торнадо», г. Харьков, ул. Космическая, 21а