



В. Н. Марков

Современное логическое программирование на языке Visual Prolog 7.5



Материалы
на www.bhv.ru



В. Н. Марков

**Современное
логическое
программирование
на языке**

Visual Prolog 7.5

Рекомендовано Федеральным государственным бюджетным образовательным учреждением высшего профессионального образования «Московский государственный технологический университет СТАНКИН» в качестве учебника для студентов высших учебных заведений, обучающихся по направлениям «Прикладная информатика», «Программная инженерия»

Санкт-Петербург

«БХВ-Петербург»

2015

УДК 004.438 Visual Prolog

ББК 32.973.26-018.1

M26

Марков В. Н.

M26 Современное логическое программирование на языке Visual Prolog 7.5:
учебник. — СПб.: БХВ-Петербург, 2015. — 544 с.: ил. —
(Учебная литература для вузов)

ISBN 978-5-9775-3487-1

В учебнике излагается полный набор классических и новейших инструментов логического программирования, а также парадигмы функционального, обобщенного, императивного и объектно-ориентированного программирования, органически вошедшие в Visual Prolog 7.5. Рассматриваются основные способы представления и обработки графов, деревьев и массивов, инструменты профессионального программирования. Приводятся примеры разработки символьных преобразователей, калькуляторов, интерпретаторов языков программирования, игровых моделей и т. п. Книга содержит практикум по программированию и описание основных классов Visual Prolog. Учебник предназначен для изучения дисциплин «Логическое программирование» и «Функциональное и логическое программирование».

*Для студентов, преподавателей
и разработчиков интеллектуальных информационных систем.*

УДК 004.438 Visual Prolog

ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Евгений Рыбаков</i>
Зав. редакцией	<i>Екатерина Капалыгина</i>
Редактор	<i>Григорий Добин</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Дизайн серии	<i>Инны Тачиной</i>
Оформление обложки	<i>Марины Дамбиевой</i>
Фото	<i>Кирилла Сергеева</i>

Подписано в печать 30.04.15.

Формат 70×100¹/16. Печать офсетная. Усл. печ. л. 43,86.

Тираж 700 экз. Заказ №

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

Оглавление

Предисловие	13
О содержании.....	14
Благодарности.....	15
О языке	16
О приоритетах.....	17
Поддержка.....	17
ЧАСТЬ I. ОСНОВНЫЕ ЭЛЕМЕНТЫ ЯЗЫКА VISUAL PROLOG	19
Глава 1. Лексика языка Visual Prolog.....	21
1.1. Алфавит	21
1.2. Комментарии.....	21
1.3. Символы разметки текста	22
1.4. Лексемы	22
1.4.1. Ключевые слова.....	22
1.4.2. Знаки пунктуации	22
1.4.3. Операции	23
1.4.4. Идентификаторы.....	23
1.4.5. Литералы	23
Целые числа.....	24
Вещественные числа.....	24
Символы	25
Строки.....	25
Двоичные данные.....	26
Списки.....	27
Составной терм	27
Глава 2. Термы	28
2.1. Понятие термов.....	28
2.2. Переменные.....	29
2.3. Списки	30
2.4. Операция сопоставления с образцом	31
2.4.1. Сопоставление термов.....	32
2.4.2. Сопоставление списков	33

2.5. Операция унификации.....	34
2.6. Операция «должно унифицироваться»	36
2.7. Неразрушающее присваивание в Visual Prolog.....	37
Глава 3. Операции языка	38
3.1. Арифметические операции и операции сравнения и сопоставления	38
3.2. Целочисленное деление	39
Глава 4. Типы, домены, подтипы	40
4.1. Типы данных	40
4.2. Встроенные домены	41
4.3. Домены-синонимы.....	45
4.4. Подтипы и категориальный полиморфизм	45
4.5. Домены, определяемые пользователем	46
4.5.1. Целочисленные подтипы	46
4.5.2. Вещественные подтипы	47
4.5.3. Списочные домены.....	48
4.5.4. Составные домены.....	48
Глава 5. Константы.....	50
5.1. Объявление пользовательских констант.....	50
5.2. Встроенные константы.....	51
ЧАСТЬ II. ЯЗЫК VISUAL PROLOG.....	53
Глава 6. Предикаты	55
6.1. Понятие предиката	55
6.2. Логические операции над предикатами.....	56
6.3. Логические формулы.....	58
6.4. Правила вывода предикатов	59
6.4.1. Вывод предиката из другого предиката.....	59
6.4.2. Формулы вычисления предикатов	61
6.5. Факты.....	64
6.6. Машина логического вывода.....	65
6.7. Работа механизма поиска с возвратом.....	66
6.8. Чистый Пролог.....	74
Глава 7. Предикаты в Visual Prolog.....	75
7.1. Объявление и определение предикатов	75
7.1.1. Режимы детерминизма	75
7.1.2. Шаблон потоков	77
7.1.3. Определение предикатов.....	78
7.2. Объявление и определение функций.....	80
7.3. Объявление и определение фактов	82
7.3.1. Режимы детерминизма факта	82
7.3.2. Определение фактов	83
7.3.3. Факты-переменные	83
7.4. Операции над разделами базы фактов	87
7.5. Операции над фактами	88

7.6. Встроенные предикаты Visual Prolog	89
7.6.1. Предикаты для работы с фактами базы данных.....	89
7.6.2. Предикаты контроля потока параметров.....	90
7.6.3. Предикаты локализации места выполнения программы в исходном тексте.....	91
7.6.4. Предикаты контроля компиляции исходного текста.....	91
7.6.5. Предикат сравнения термов.....	92
7.6.6. Предикаты преобразования типов.....	92
7.6.7. Предикаты обработки эллипсиса	96
7.6.8. Предикаты получения размера домена	97
7.6.9. Предикат получения размера терма.....	98
7.6.10. Предикаты объявления/проверки домена переменной.....	98
7.6.11. Предикат обработки ошибки	99
7.6.12. Предикаты управления выполнением программы.....	99
Глава 8. Модули.....	101
8.1. Область видимости.....	102
8.2. Структура модуля	102
8.3. Использование модулей в проекте	104
8.3.1. Библиотека констант и доменов	104
8.3.2. Библиотеки предикатов.....	105
8.3.3. Глобальные переменные проекта.....	105
Глава 9. Отсечение и отрицание	108
9.1. Принцип работы отсечения	108
9.1.1. Область видимости отсечения	109
9.1.2. Использование отсечений	109
9.2. Зеленые и красные отсечения	111
9.3. Динамическое отсечение	112
9.4. Отрицание	113
Глава 10. Циклы с откатом	115
10.1. Структура цикла с откатом	115
10.2. Реализация циклов с откатом	117
10.3. Использование изменяемых переменных в циклах с откатом	122
10.4. Циклы с откатом на основе отрицания	125
Глава 11. Рекурсия	127
11.1. Структура рекурсии.....	127
11.2. Реализация рекурсии	130
11.3. Мемоизация.....	138
Глава 12. Ввод/вывод	140
12.1. Ввод/вывод в консольном приложении.....	140
12.1.1. Основные функции ввода с клавиатуры	140
12.1.2. Основные предикаты вывода на экран	141
12.1.3. Использование предиката <i>hasDomain</i>	142
12.2. Файловый ввод/вывод	143
12.3. Потоковый ввод/вывод.....	144
12.3.1. Потоковый ввод/вывод в цикле с откатом.....	146
12.3.2. Потоковый ввод/вывод в рекурсивном цикле	147

12.4. Строковые потоки.....	148
2.4.1. Поток ввода строк.....	149
12.4.2. Поток вывода строк	151
Глава 13. Списки	153
13.1. Представление списков в памяти компьютера.....	153
13.2. Встроенные операции над списками.....	154
13.2.1. Объявление списочных доменов	157
13.3. Реализация очереди и дека.....	157
13.4. Принципы рекурсивной обработки списков	158
13.4.1. Построение списков из элементов	158
13.4.2. Построение реверсивных списков из элементов	159
13.4.3. Сканирование списка	160
13.4.4. Модификация списка.....	161
13.4.5. Синхронная обработка списков.....	161
13.5. Примеры рекурсивной обработки списков.....	162
13.5.1. Определение длины списка.....	162
13.5.2. Построение списка	164
13.5.3. Соединение списков	165
13.5.4. Реверс списка	167
13.6. Ввод/вывод списков целиком	167
13.6.1. Терминальный ввод/вывод списков	167
13.6.2. Файловый ввод/вывод списков.....	168
13.6.3. Потоковый ввод/вывод списков	168
13.7. Поэлементный ввод/вывод списков	169
13.7.1. Поэлементный ввод/вывод списков в цикле с откатом	169
13.7.2. Поэлементный ввод/вывод списков в рекурсивном цикле	170
13.8. Предикат выборки элементов списка.....	171
13.9. Коллектор списков.....	172
13.10. Представление базы фактов списками фактов	174
13.10.1. Когерентность базы фактов и списка фактов	174
13.10.2. Домен фактов внутренней базы данных	175
13.10.3. Создание списка фактов внутренней базы данных	175
13.10.4. Восстановление внутренней базы данных из списка фактов	175
13.10.5. Предикаты преобразования внутренней базы данных в список фактов и обратно	176
Глава 14. Параметрический полиморфизм.....	178
14.1. Полиморфизм параметров предикатов	178
14.2. Полиморфизм параметров доменов	179
14.2.1. Полиморфный списочный домен класса <i>core</i>	182
Глава 15. Эллипсис	183
Глава 16. Предикаты второго порядка и анонимные предикаты.....	185
16.1. Предикатные и функциональные домены	185
16.2. Анонимные предикаты и функции.....	189
16.2.1. Определения анонимных предикатов	190

16.2.2. Использование анонимных предикатов и функций.....	190
16.2.3. Замыкание	195
16.2.4. Каррирование.....	195
16.3. Высокоуровневые предикаты и функции класса <i>list</i>	197
Глава 17. Императивные конструкции	205
17.1. Разрушающее присваивание	205
17.2. Ветвление	206
17.3. Условные выражения	207
17.4. Цикл <i>foreach</i>	207
17.5. Циклы с заданным числом повторений	210
Глава 18. Обработка исключительных ситуаций.....	211
18.1. Явный вызов исключений.....	211
18.2. Обработка исключений	213
Глава 19. Классы	218
19.1. Структура класса	218
19.1.1. Параметры состояния класса и объекта.....	221
19.2. Объекты	221
19.2.1. Создание объектов.....	221
19.2.2. Объектные предикаты	222
19.2.3. Объектные свойства	223
19.2.4. Удаление объектов	225
19.3. Классы	226
19.3.1. Конструкторы	226
19.3.2. Состояние класса	227
19.4. Наследование кода и поддержка интерфейсов.....	228
19.5. Сохранение объектов	230
19.6. Операции над всеми живущими объектами	231
19.7. Примеры использования классов	232
Глава 20. Обобщенное программирование	239
20.1. Обобщенные интерфейсы	239
20.2. Обобщенные классы.....	240
20.3. Обобщенные реализации	241
20.4. Пример обобщенной очереди.....	242
ЧАСТЬ III. СРЕДСТВА ПРОФЕССИОНАЛЬНОГО ПРОГРАММИРОВАНИЯ.....	245
Глава 21. Многопоточность	247
21.1. Основные операции с потоками	247
21.1.1. Создание потоков	247
21.1.2. Завершение потоков	248
21.1.3. Приостановка и возобновление потоков	248
21.1.4. Примеры создания потока	249
21.2. Мониторы.....	251
21.3. Защита.....	253

Глава 22. Доступ к API-функциям Windows.....	256
22.1. Описание типов данных API-функций доменами Visual Prolog	256
22.2. Объявления предикатов для вызова API-функций.....	258
22.3. Использование пакета <i>windowsAPI</i>	260
22.4. Использование API-функций из библиотеки Windows	262
Глава 23. Разработка и использование DLL.....	265
23.1. Создание DLL-проекта.....	266
23.2. Описание экспортируемых предикатов и функций DLL-проекта	267
23.3. Использование DLL-проекта	268
Глава 24. Отладка приложений.....	271
24.1. Отладчик Visual Prolog.....	271
24.2. Просмотр значений переменных средствами языка.....	274
24.3. Контроль стека и кучи.....	275
ЧАСТЬ IV. ПРЕДСТАВЛЕНИЕ И ОБРАБОТКА ДАННЫХ В VISUAL PROLOG.....	277
Глава 25. Графы	279
25.1. Представление ориентированных графов.....	279
25.2. Представление неориентированных графов.....	282
25.3. Поиск «сначала вглубь»	283
25.4. Поиск «сначала вширь».....	285
Глава 26. Деревья	288
26.1. Представление деревьев в Visual Prolog	288
26.2. Операции над деревьями.....	290
26.2.1. Добавление вершины	290
26.2.2. Поиск вершины.....	291
26.2.3. Удаление вершины	291
26.3. Красно-черные деревья	294
Глава 27. Массивы.....	299
27.1. Класс <i>binary</i>	299
27.1.1. Создание массивов <i>binary</i>	299
27.1.2. Основные операции над массивами <i>binary</i>	300
27.2. Класс <i>arrayM</i>	301
27.2.1. Создание одномерных массивов	301
27.2.2. Основные операции над одномерными массивами	301
27.3. Класс <i>array2M</i>	303
27.3.1. Создание двумерных массивов.....	303
27.3.2. Основные операции над двумерными массивами	303
27.4. Класс <i>arrayM_boolean</i>	304
27.4.1. Создание одномерных булевых массивов	304
27.4.2. Основные операции над одномерными булевыми массивами	304
27.5. Способы обработки массивов.....	305

Глава 28. Символьные преобразования	307
28.1. Этапы анализа текстов	307
28.2. Основы анализа текстов на Visual Prolog	308
28.2.1. Простой лексический анализ	308
28.2.2. Простой синтаксический анализ	308
28.3. Анализ математических выражений	309
28.4. Парсер математических выражений с произвольной грамматикой	315
28.5. Символьное дифференцирование выражений.....	322
28.6. Калькулятор	329
Способ 1	329
Способ 2	330
Способ 3	331
28.7. Задания для самостоятельного решения.....	333
Глава 29. Интерпретатор программ	335
29.1. Лексический анализ.....	335
29.2. Синтаксический анализ	341
29.3. Интерпретатор программ	346
29.4. Задания для самостоятельного решения.....	353
Глава 30. Практические рекомендации.....	355
30.1. Выбор способа представления и обработки данных	355
30.2. Управление памятью в Visual Prolog 7	356
30.2.1. Стек вызовов	356
Управляемый детерминизм.....	357
Задание произвольного размера стека вызовов.....	361
30.2.2. Динамическая память	363
Отказ от глобального стека в Visual Prolog 7	364
Многопоточность.....	365
30.2.3. Системная память	365
ЧАСТЬ V. ПРАКТИКУМ ПО ПРОГРАММИРОВАНИЮ.....	367
Глава 31. Введение в Visual Prolog.....	369
31.1. Создание консольного проекта.....	369
31.2. Запуск программы	373
31.3. Расширение области видимости.....	376
31.4. Управление выводом в консоли	376
31.5. Использование классов <i>PFC</i>	377
Глава 32. Поиск с откатом на фактах	382
32.1. Цвета автомобилей	382
32.2. Точки на плоскости	385
32.3. Двенадцать месяцев.....	390
32.4. Зеленые и красные отсечения	390
Глава 33. Поиск с откатом на правилах	392
33.1. Родственные отношения	392
33.2. Моделирование комбинационных схем.....	397

33.3. Фигуры на плоскости	400
33.4. Ребус	405
Глава 34. Рекурсивные правила	408
34.1. Арифметика.....	408
34.2. Ряды	413
34.3. Длинная арифметика	415
34.4. Перевод чисел из одной системы счисления в другую	416
34.5. Обработка строк.....	423
Глава 35. Рекурсивные правила на списках.....	429
35.1. Списки	429
35.2. Математика	434
35.3. Треугольник Паскаля	435
35.4. Длинная арифметика	437
35.5. Преобразователь чисел из одной системы счисления в другую	439
35.6. Монеты	439
35.7. Домино	441
35.8. Ребус	442
35.9. Ребус с произвольными словами.....	444
35.10. Ребус с произвольными словами произвольной длины	446
35.11. Расстановка N ферзей на доске $N \times N$	448
35.12. Кувшины.....	452
35.13. Строки. Баланс скобок	454
35.14. Строки. Транслитерация	455
35.15. Списки. Поиск в ширину подсписка в списке	456
Глава 36. Внутренняя база данных.....	459
36.1. Лексика русского языка	459
36.2. Искусственная жизнь.....	462
36.3. Представление базы данных списком фактов	466
Глава 37. Задачи на графах	468
Глава 38. Задачи на деревьях.....	473
Глава 39. Задачи на массивах	478
39.1. Решето Эратосфена	478
39.2. Сравнение частотных буквей текстов	481
ПРИЛОЖЕНИЯ	485
Приложение 1. Описание свойств и предикатов класса <i>console</i>	487
Свойства	487
Предикаты	487
Приложение 2. Описание предикатов класса <i>file</i>	491
Приложение 3. Описание конструкторов класса <i>inputStream_file</i>	495

Приложение 4. Описание конструкторов класса <i>outputStream_file</i>	497
Приложение 5. Пакет <i>stream</i>	500
Описание предикатов класса <i>inputStream</i>	500
Описание предикатов класса <i>outputStream</i>	501
Описание предикатов класса <i>stream</i>	501
Приложение 6. Меню Visual Prolog	503
Приложение 7. Быстрые клавиши меню Visual Prolog	509
Приложение 8. Панель инструментов Visual Prolog	512
Приложение 9. Описание предикатов класса <i>std</i>	514
Приложение 10. Описание класса <i>string</i>	517
Константа	517
Домены	517
Предикаты	517
Приложение 11. Описание электронного архива, сопровождающего книгу	534
Предметный указатель	535

Главы, помещенные в электронный архив

ЧАСТЬ VI. ПРОСТЫЕ ГРАФИЧЕСКИЕ ПРИЛОЖЕНИЯ

Глава 40. Часы и секундомер	1
40.1. Создание графического приложения	1
40.2. Конструирование формы	3
40.3. Разработка программного кода	7
40.3.1. Программирование часов.....	8
40.3.2. Программирование секундомера	10
Глава 41. Искусственная жизнь	13
41.1. Конструирование формы	13
41.2. Разработка программного кода	17
41.2.1. Константы, домены и факты класса.....	17
41.2.2. Размещение игрового поля на форме	18
41.2.3. Случайное заполнение колонии и сброс	22
41.2.4. Программирование зависимых переключателей	23
41.2.5. Старт/стоп таймера.....	24
41.2.6. Обработчик «тиков» таймера.....	24
41.2.7. Описание ручного режима моделирования	26
41.2.8. Редактирование колонии мышью.....	26
41.2.9. Создание и моделирование колонии	27

Глава 42. Машина Тьюринга.....	29
42.1. Создание формы графического приложения.....	29
42.2. Создание меню.....	30
42.3. Создание панели инструментов.....	35
42.4. Разработка графического интерфейса формы.....	42
42.5. Разработка статического класса <i>TuringMachine</i>	46
42.5.1. Описание предикатов класса <i>TuringMachine</i>	49
42.6. Разработка программного кода для пунктов меню.....	50
42.6.1. Меню <i>Файл</i>	51
42.6.2. Меню <i>Состояние ДМТ</i>	53
42.6.3. Меню <i>Запуск</i>	54
42.6.4. Меню <i>Справка</i>	57
42.7. Испытание проекта.....	58
Приложение 12. Описание графических элементов управления	62

Предисловие

Со времени выхода незабвенной книги А. Н. Адаменко и А. М. Кучукова «Логическое программирование и Visual Prolog» прошло более 10 лет. За это время Visual Prolog включил в свой арсенал принципиально новые для Пролога парадигмы и инструменты программирования. Наиболее значимыми из них являются предикаты-функции, анонимные предикаты, полиморфизм, коллектор списков, факты-переменные и т. п. Кроме того, Visual Prolog приобрел развитые средства ООП (объектно-ориентированного программирования), а также богатый набор средств функционального, обобщенного и императивного программирования, которые органически вошли в язык и расширили его выразительные возможности. Однако произошедшая эволюция языка до сих пор не освещена в учебных и популярных изданиях на русском языке. Эта книга и призвана заполнить образовавшийся пробел.

Книга предназначена для студентов, изучающих логическое программирование, их преподавателей, а также для программистов, желающих ознакомиться с новыми языковыми средствами типизированного языка Visual Prolog. Все базовые парадигмы языка излагаются с использованием консольных программ.

Применительно к логическому языку программирования в книге используются два термина. Когда речь идет о языке Пролог вообще, а по сведениям автора язык Пролог имеет более 40 компиляторов, то используется термин Пролог. Когда описывается заявленная в названии книги версия логического языка, то используется название Visual Prolog.

ВНИМАНИЕ!

Компилировать и запускать на выполнение без изменения можно только те программы, которые имеют раздел `goal`. Фрагменты исходного кода, размещенные в тексте, не компилируйте как самостоятельные программы. Они приведены только для объяснения исходного кода программ.

Основные термины логического программирования приведены в русском и английском вариантах — чтобы читатель мог адекватно воспринимать сообщения компилятора об ошибках и его предупреждения, а также понимать англоязычную справку Visual Prolog.

О содержании

Первая часть книги рассматривает лексику Visual Prolog, понятие терма, как единственной синтаксической сущности языка, и процедуру унификации термов, как единственный способ обработки термов в языке. В эту часть также включены сведения справочного характера об операциях языка, типах и подтипах данных, о доменах.

Вторая, третья и четвертая части книги представляют собой теоретический лекционный материал по дисциплинам «Логическое программирование» и «Функциональное и логическое программирование». Лекционный материал подкреплен множеством простых консольных примеров, готовых к компиляции и выполнению в среде Visual Prolog. При этом:

- *вторая часть* книги посвящена собственно языку Visual Prolog 7.5 и представляет описание основных понятий языка: предикаты, функции, факты, а также раскрывает суть механизмов управления: поиск с возвратом, отсечение и рекурсию. Кроме того, эта часть содержит описание средств метапрограммирования, обобщенного программирования и основных понятий ООП, принятых в Visual Prolog;
- *третья часть* книги предназначена для продвинутых пользователей, желающих использовать такие средства профессионального программирования, как многопоточность, вызов API-функций и разработку DLL;
- *четвертая часть* книги открывает читателю различные способы представления данных и методы их обработки в прикладных задачах. Кроме классического для Пролога представления данных в виде графов и разного рода деревьев, рассматриваются некоторые разновидности массивов с прямым доступом к элементам. Существенное внимание уделено грамматическому разбору текстов с описанием и точным позиционированием возможных синтаксических ошибок. Подробно рассмотрены символные преобразования на примере дифференцирования, а также интерпретатор простых ассемблерных программ и калькулятор математических выражений.

Пятая часть книги содержит богатый набор более сложных примеров консольных программ и представляет собой практикум по программированию в составе девяти практических занятий с примерами и заданиями. Методика практикума предлагает ряд задач, решение которых совершенствуется от занятия к занятию по мере расширения набора применяемых языковых средств. Используя книгу в качестве учебного пособия, лектор может по своему усмотрению распределить эти практические занятия по всему курсу.

Завершают «бумажную» книгу 11 приложений, предметный указатель и список рекомендуемой литературы.

Книгу сопровождает электронный архив, содержащий шестую ее часть в составе глав 40, 41 и 42 и приложения к ним (папка Часть VI), а также ряд готовых проектов (папки Projects и GUI-Projects) и дистрибутив Visual Prolog 7.5 PE.

Электронный архив с материалами к этой книге можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке <ftp://ftp.bhv.ru/9785977534871.zip> или со страницы книги на сайте www.bhv.ru (см. приложение 11).

Шестая часть книги вводит читателя в мир графических приложений на Visual Prolog. В главе 40 демонстрируется реализация часов реального времени и секундомер с использованием простых графических элементов. Глава 41 представляет модель искусственной жизни клеточной колонии на квадратном поле и имеет целью показать операции рисования. Проект, содержащийся в главе 42, является интерпретатором программ для детерминированной машины Тьюринга и содержит текстовый редактор sciLexer, построенный на принципах компонента редактирования Scintilla, а также парсер с точным позиционированием и описанием возможных ошибок и собственно интерпретатор языка программирования машины Тьюринга.

Относительно объемные консольные проекты, описанные в главах 16, 20–23, 25, 26, 28, 29, 34–39, представлены в папке Projects, а в папке GUI-Projects содержатся графические проекты, описанные в *шестой части* книги. Эти проекты имеют исполнимые файлы и все необходимые библиотеки для запуска.

Благодарности

В свое время основополагающий вклад в появление первого и единственного типизированного Пролога внесли Лео Йенсен (Leo Schou-Jensen) и Финн Гронсков (Finn Gronskov), которые ныне являются содиректорами фирмы Prolog Development Center (PDC), Копенгаген, Дания. Благодаря их пионерским разработкам программы на Visual Prolog выполняются непосредственно процессором, без промежуточной интерпретации машиной логического вывода, что в целом повысило надежность, безопасность и скорость выполнения логических программ. Сейчас Visual Prolog по указанным критериям стал непревзойденным среди прочих Прологов. В настоящее время главными идеологами языка Visual Prolog являются сотрудники PDC Томас Линдер Пулс (Thomas Linder Puls), Карстен Келер Холст (Carsten Kehler Holst) и Михаэл Паул Брандт (Michael Paul Brandt). Управление проектом осуществляют уже упомянутый Томас Линдер Пулс.

Хочется отметить тот факт, что непосредственными разработчиками Visual Prolog являются наши соотечественники, которые работают в Санкт-Петербурге в фирме PDC SPB. Следует выразить им искреннюю благодарность за огромный вклад и участие нашей страны в развитии логического программирования. В 1993 году группа под руководством Виктора Анатольевича Юхтенко освоила язык программирования Turbo Prolog и в сотрудничестве с PDC разработала первую версию интегрированной среды разработки (IDE) в среде ОС Windows. В составе группы вместе с В. А. Юхтенко работы начинали Александр Яковлевич Бутовский, Андрей Михайлович Кучуков и Александр Гостинцев. Группа впоследствии преобразовалась в Санкт-Петербургскую компанию PDC SPB, а В. А. Юхтенко в настоящее время является ее директором. С 1994 года в PDC SPB компилятор Visual Prolog и библиотеку runtime разрабатывают, совершенствуют и поддерживают Александр Борисович Доронин и Сергей Викторович Мухин. Они же, кроме того, сейчас обес-

печивают руководство разработкой модулей IDE, отладчика и основных классов PFC. Непосредственно разработку отладчика и модулей IDE осуществляют Денис Геннадьевич Долинин и Александр Альбертович Гукалов. Механизм справочной системы IDE и модифицированная оболочка экспертной системы ESTA появились благодаря Олегу Васильевичу Еремину, а справочная система, в том числе ее Web-версия, появилась благодаря А. М. Кучукову и Елизавете Григорьевне Сафро.

В разное время в PDC SPB работали многие талантливые и увлеченные специалисты, среди которых следует упомянуть Юрия Юрьевича Ильина, Михаила Викторовича Зайченко, Артема Константиновича Ляхова, Александра Львовича Горлова, Бориса Анатольевича Белова, Бориса Владимировича Уласевича, Сергея Миронова, Льва Николаевича Романова и многих других.

О языке

Visual Prolog 7.5 — строго типизированный объектно-ориентированный язык, основанный на парадигме логического программирования. Visual Prolog может быть использован как для преподавания логического программирования в вузах, так и в качестве инструмента для создания крупных 32/64-разрядных коммерческих приложений под платформу Microsoft Windows. Его компилятор написан на нем самом, путем последовательной «раскрутки». И это обычное дело для многих языков программирования. Однако справедливости ради надо заметить, что сканер и парсер сделаны на C++, потому что фирма PDC получает парсер с помощью генератора YACC (Yet Another Compiler Compiler, еще один компилятор компиляторов), все остальное сделано собственными средствами Visual Prolog.

Visual Prolog отличается от других языков уникальным набором составляющих:

- предложения Хорна как основа логического программирования;
- полная поддержка ООП;
- строгая статическая типизация;
- алгебраические типы данных;
- встроенный механизм сопоставления с образцом и унификация;
- управляемый недетерминизм на основе поиска с откатом;
- интегрированная база фактов;
- предикаты и функции высших порядков и анонимные предикаты и функции;
- поддержка императивных конструкций и разрушающего присваивания;
- обработка исключительных ситуаций;
- поддержка обобщенного программирования;
- поддержка многопоточности и синхронизации потоков;
- поддержка параметрического полиморфизма;
- автоматическое управление памятью (garbage collector);

- поддержка линкования с кодом C/C++;
- поддержка прямых вызовов функций API.

Комбинация строгого контроля соответствия типов, отсутствие арифметических манипуляций с указателями и автоматическое управление памятью практически исключают ошибки доступа к памяти. Поэтому Visual Prolog генерирует безопасный и надежный код.

Алгебраические типы данных, базы фактов и сопоставление с образцом, комбинированные с недетерминированным поиском, делают Visual Prolog очень хорошим средством для представления и обработки структурированных данных и знаний.

О приоритетах

Изучение Пролога в университетах, как правило, предваряется изучением императивных языков, и студенты при этом проникаются негласным правилом о том, что процессор выполняет только те действия, которые синтаксически явным способом выражены в программе. Однако в Прологе не все действия выражаются синтаксически — часть управления вычислениями скрыта «между строк». На первых порах это может вызвать некоторые трудности привыкания к новой концепции выполнения программы — механизму поиска решения с откатом, не выражаемому синтаксическими конструкциями. Преодоление этих трудностей заключается не столько в уяснении нового механизма — он предельно прост, сколько в многократном его использовании, чему и уделено внимание в практикуме по программированию, составляющем пятую часть книги.

Бытует шутка о Прологе, как о языке таинственных деклараций, которые, будучи поданы на вход механизма поиска с возвратом, непостижимым образом ведут к чудесному решению задачи. В этой книге Пролог преподнесен как язык, имеющий четкую и понятную процедурную реализацию, и программирование на нем рассматривается с точки зрения ясного понимания программистом того, каким образом и насколько эффективно процессор будет выполнять программу. Кроме того, логическое программирование не рассматривается под тем углом зрения, насколько декларативно программа будет выглядеть со стороны, оправдывая свою принадлежность к семейству декларативных языков, и насколько она логична и чиста. В современном прагматичном мире программистов одна лишь декларативность и чистота логических программ сами по себе ничего не стоят. Поэтому основной упор сделан на эффективности решения задач на языке Visual Prolog по критерию минимального времени разработки проекта и максимальной скорости его выполнения.

Поддержка

Visual Prolog можно скачать по адресу <http://discuss.visual-prolog.com> в двух редакциях:

- Visual Prolog 7.5 CE (Commercial Edition) — Пролог для коммерческого использования, имеет полный набор библиотек для профессионального программирования, платный;

- Visual Prolog 7.5 PE (Personal Edition) — Пролог для учебных целей, имеет сокращенный набор библиотек, все языковые средства доступны для использования, требуется бесплатная регистрация на сайте.

Большинство программ, приведенных в книге, работают в Visual Prolog 7.5 PE. Программы, использующие библиотеки коммерческой редакции Visual Prolog, будут работать только в ней. На все вопросы по материалам книги можно получить ответ на русскоязычном форуме по логическому программированию:

<http://www.hardforum.ru/f141/>

Владеющие английским языком могут также обращаться на форум:

<http://discuss.visual-prolog.com/viewforum.php?f=2>

и в wiki по языку Visual Prolog:

http://wiki.visual-prolog.com/index.php?title>Main_Page

часть I



Основные элементы языка Visual Prolog

Глава 1. Лексика языка Visual Prolog

Глава 2. Термы

Глава 3. Операции языка

Глава 4. Типы, домены, подтипы

Глава 5. Константы

ГЛАВА 1



Лексика языка Visual Prolog

Лексика представляет собой описание неделимых сущностей языка программирования, из которых, как из кирпичиков, строятся все вычислительные конструкции этого языка. Неделимыми они являются потому, что воспринимаются компилятором языка как цельные фрагменты текста, хотя и состоят из последовательности символов алфавита. Такие неделимые сущности называются *лексемами*.

Некоторые лексемы — например, числа, ключевые слова и строки известны многим начинающим программистам. О других лексемах — двоичных данных, списках знают лишь опытные. В этой главе мы рассмотрим только лексемы Visual Prolog, а способы построения из них языковых конструкций будут описаны в следующих главах.

1.1. Алфавит

Алфавит Visual Prolog включает буквы английского и национального (русского) алфавитов, цифры, знаки пунктуации, операторы и ключевые слова. В языке есть различия между идентификаторами, начинающимися с заглавной и строчной буквы.

1.2. Комментарии

Для указания многострочных комментариев используются открывающие символы `/*` и закрывающие символы `*/`. Для указания односторонних комментариев служит знак процента `%`, действие которого распространяется до конца строки.

Пример комментариев:

```
/* Многострочные  
комментарии */  
% Однострочные комментарии
```

1.3. Символы разметки текста

Такими символами являются пробелы, табуляции и символы перехода на новую строку. Комментарии и символы разметки во время компиляции не анализируются.

1.4. Лексемы

В качестве лексем языка Visual Prolog выступают ключевые слова, знаки пунктуации, операции, идентификаторы и литералы.

1.4.1. Ключевые слова

Ключевые слова разделяются на старшие и младшие. Это деление условное и предназначено только для различения цвета ключевых слов при их отображении в редакторе.

К старшим ключевым словам относятся:

class	domains	inherits	predicates
clauses	end	interface	properties
constants	facts	monitor	resolve
constructors	goal	namespace	supports
delegate	implement	open	

Младшие ключевые слова:

align	digits	failure	language	quot
and	div	finally	mod	rem
anyflow	do	foreach	multi	single
as	else	from	nondeterm	then
bitsize	elseif	guard	or	to
catch	erroneous	if	orelse	try
determ	externally	in	procedure	

Все ключевые слова, кроме as и language, зарезервированы, и использовать их в качестве имен, вводимых программистом в исходный текст программы, не допускается. Ключевое слово end всегда комбинируется с другими ключевыми словами:

end class	end implement	end interface
end if	end foreach	end try

1.4.2. Знаки пунктуации

Знаками пунктуации являются:

; ! , . # [] | () { } : :- ::

Многосимвольные знаки :- и :: не должны разделяться пробелами.

1.4.3. Операции

Операции определяют вычисления, которые должны быть выполнены над указанными данными. Все операции бинарные, но плюс и минус могут быть и унарными.

```
+      -      /      *      ^      =      div      mod      quot      rem
<      >      <>     ><     <=     >=     :=      ==
```

Не допускается разрывать пробелами многосимвольные имена операций:

```
<>     ><     <=     >=     :=      ==      div      mod      quot      rem
```

1.4.4. Идентификаторы

В языке различают четыре вида идентификаторов: строчные, заглавные, анонимные и эллипсис.

Строчный идентификатор — последовательность букв, цифр и знаков подчеркивания, начинающаяся со строчной буквы.

Примеры правильных строчных идентификаторов:

```
S      жук001      w_ю_      xMary      й9я8ы
```

Примеры неправильных строчных идентификаторов:

```
8ver      % начинается с цифры
soap#1    % содержит символ #
Soap      % начинается с заглавной буквы
my prog   % содержит пробел
facts     % ключевое слово
```

Заглавный идентификатор — последовательность букв, цифр и знаков подчеркивания, начинающаяся с заглавной буквы или со знака подчеркивания.

Примеры правильных заглавных идентификаторов:

```
S      Жук_001      _Mary      Facts
```

Примеры неправильных заглавных идентификаторов:

```
8ver      % начинается с цифры
Soap$1    % содержит символ $
soap      % начинается со строчной буквы
Му prog   % содержит пробел
```

Анонимный идентификатор — знак подчеркивания: _.

Эллипсис — знак многоточия:

1.4.5. Литералы

В качестве литералов выступают целые и вещественные числа, символы, строки, двоичные данные, списки и составные термы.

Целые числа

Формат целых чисел зависит от системы счисления, в которой представлено число (табл. 1.1).

Таблица 1.1. Представление десятичных, восьмеричных и шестнадцатеричных чисел

Система счисления	Знак (опционально)	Префикс	Допустимые цифры	Примеры
Десятичное целое	±	Нет префикса	0123456789	355 +90 -897
Восьмеричное целое	±	0o	01234567	0o777 +0o121 -0o67
Шестнадцатеричное целое	±	0x	0123456789 AaBbCcDdEeFf	0xABCD +0xc82 -0x3ff5

Вещественные числа

Вещественные числа представляются только в десятичной системе счисления (табл. 1.2, 1.3).

Таблица 1.2. Представление вещественных чисел

Мантисса				Порядок (опционально)		
Знак числа (опционально)	Цифры целой части	Десятич- ный раз- делитель	Цифры дробной части	Префикс показа- теля степени	Знак показа- теля степени	Цифры показателя степени
		опционально				
±	0123456789	.	0123456789	e или E	±	0123456789

Таблица 1.3. Примеры вещественных чисел

Число	Пояснение
-123.789	число без степени
+123.456E-5	+123.456·10 ⁻⁵
-0.123456e+12	-0.123456·10 ¹²

Символы

Символ — любой печатаемый на клавиатуре символ, заключенный в апострофы, или ESC-последовательность, также заключенная в апострофы. *ESC-последовательность (ESCAPE-последовательность)* — это последовательность управляющих символов, начинающаяся с обратной наклонной черты — знака шиллинга \ (иногда этот знак называют *обратным слешем*). Именно обратная наклонная черта и является признаком ESC-последовательности:

- '\t' — табуляция (tabulation);
- '\n' — переход на новую строку (new line);
- '\r' — возврат в начало строки (return);
- '\uxxxx' — Unicode-символ, имеющий шестнадцатеричный код xxxx.

Кодировку Unicode-символов можно посмотреть, например, в приложении MS Word, используя меню **Вставка | Символ**. ESC-последовательность может определять не только действия, но и символы. Примеры символов и ESC-последовательностей представлены в табл. 1.4.

Таблица 1.4. Примеры символов

Символ	Пояснение
'f'	Буква f
'\u0066'	Буква f, заданная двухбайтовым шестнадцатеричным кодом
'Ж'	Буква Ж
'\u0416'	Буква Ж, заданная двухбайтовым шестнадцатеричным кодом
'\n'	Переход на новую строку
'\'	Апостроф '
'\\'	Обратная наклонная черта \
'\"'	Двойная кавычка "
'/'	Наклонная черта /
'\u2194'	Символ ↔
'\u203C'	Символ !!

С помощью Unicode-символов можно задавать символы, которых нет на клавиатуре.

Строки

Строка — один или более символов, заключенных в двойные кавычки. Стока не должна разрываться клавишей <Enter>. Строки могут содержать символы, представляемые ESC-последовательностями. Строки с префиксом @ могут многократно разрываться клавишой <Enter> и, кроме того, выполнение ESC-последова-

тельностей в такой строке подавляется. В табл. 1.5 показаны особенности обработки строк с префиксом @.

Таблица 1.5. Особенности обработки строк при наличии префикса @

Строка	Вывод на экран
"Visual prolog"	Visual prolog
"Visual\tprolog"	Visual prolog
@"Visual prolog"	Visual prolog
@"Visual\tprolog"	Visual\tprolog
"Visual prolog"	Ошибка компиляции
@"Visual prolog"	Visual prolog

Префикс @ удобно использовать, например, при задании в программе пути к файлу, который обычно содержит несколько знаков обратной наклонной черты. С его помощью также удобно определять многострочный текст без многократного ввода ESC-последовательности \n.

Кроме того, префикс @ полезно применять для определения строк, заданных без кавычек и обрамленных парными символами: открывающим и закрывающим. В этом случае компилятор будет проверять наличие закрывающего символа, если был использован открывающий символ. В табл. 1.6 представлены допустимые открывающие и закрывающие символы.

Таблица 1.6. Открывающие и закрывающие парные символы

Парные символы		Парные символы	
Открывающий	Закрывающий	Открывающий	Закрывающий
@()	@)	(
@{	}	@}	{
@[]	@]	[
@<	>	@>	<

Например, выровненный по центру абзац HTML-текста можно задать в виде строки:

```
Str = @"[<p align="center">Некоторый текст</p>]
```

Двоичные данные

Двоичные данные — последовательность байтов, заключенная в квадратные скобки, перед которыми указывается знак \$. Байты могут быть представлены десятич-

ными, восьмеричными или шестнадцатеричными целыми числами в диапазоне от 0 до 255, или арифметическими выражениями, значения которых может быть вычислено во время компиляции (табл. 1.7).

Таблица 1.7. Примеры двоичных данных

Двоичные данные	Вывод на экран
\$ [10, 20, 30]	\$ [0A, 14, 1E]
\$ [10, 0x20, 0o30]	\$ [0A, 20, 18]
\$ [4+4, 2*0x10]	\$ [08, 20]

Двоичные данные используются в Visual Prolog не только для связи с другими языками и для низкоуровневой обработки данных, но и в целях безопасного хранения и обмена данными произвольного типа, а также в качестве массива данных с произвольным доступом к элементам.

Списки

Синтаксически, список представляет собой заключенную в квадратные скобки последовательность элементов одного типа, разделенных запятой. Список может быть также выражен головой списка, роль которого играет первый элемент, и хвостом списка, отделенным от головы вертикальной чертой. Хвост списка является списком. Варианты записи списков приведены в табл. 1.8.

Таблица 1.8. Варианты записи списков

Список	Описание
[]	Пустой список
[10, 20, 30]	Список целых чисел
["123", "abcd"]	Список строк
[10, "abcd"]	Неправильный список
[3 W]	Список, состоящий из головы — числа 3 и хвоста, выраженного переменной W
[3 [45, 12]]	Список, состоящий из головы — числа 3 и хвоста, представленного списком [45, 12]
[[8, 45], [], [90]]	Список, состоящий из трех вложенных списков, один из которых пустой
[[]]	Непустой список, содержащий один пустой список

Составной терм

Составной терм определяется пользователем и служит для представления в программе структур данных. Составные термы описаны в главе 2.

ГЛАВА 2



Термы

2.1. Понятие термов

К *простым* термам относятся числа, символы, строки и идентификаторы, описанные в лексике языка.

Составные термы определяются через простые термы и составные термы, в том числе и через самих себя. Иерархия термов изображена на рис. 2.1.

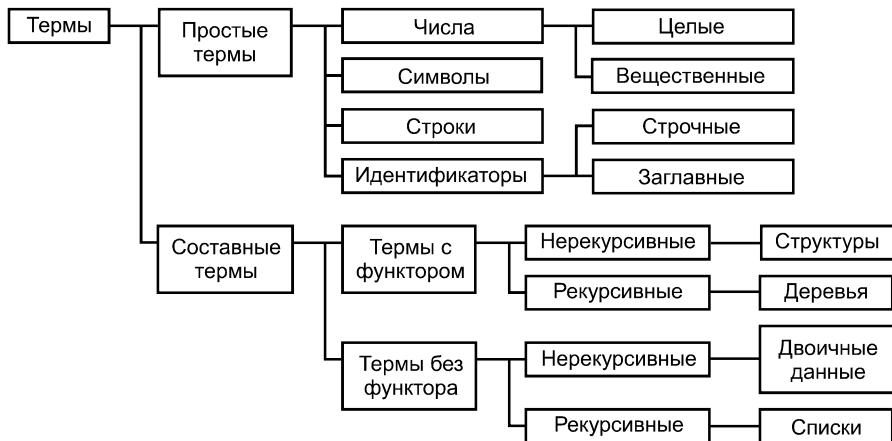


Рис. 2.1. Иерархия термов

Определение составного терма в общем случае рекурсивно. Согласно своему названию такой терм является именованной структурой, составленной из нескольких сущностей, как показано на рис. 2.2.

Имя терма называется *функтором* и обозначается строчным идентификатором. Количество аргументов называется *арностью*. Терм на рис. 2.2 является *N-арным* термом, иногда его называют *N-местным* термом, что есть то же самое. Каждый аргумент, в свою очередь, сам является простым или составным термом. Если один или несколько аргументов терма выражаются этим же термом, то говорят, что такой терм определен *рекурсивно*. В комментариях к логическим программам термы

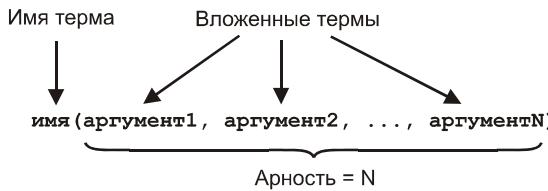


Рис. 2.2. Составной терм

иногда обозначают в виде имени и арности `имя/N`, опуская для краткости аргументы.

Составные термы могут не иметь аргументов. Такие термы содержат пустые скобки и называются *нульарными термами*.

Например, терм `f(999, arm)` имеет функтор `f`, является двухарным, первый аргумент — число `999`, второй аргумент — строчный идентификатор `arm`, терм определен нерекурсивно и имеет сокращенное обозначение `f/2`.

Терм `spice()` является нульарным термом и в документации имеет сокращенное обозначение `spice/0`.

Рекурсивный терм `z(babilon, z(inter,Near,8), 43)` является трехарным. Первый аргумент — строчный идентификатор `babilon`, второй аргумент — *вложенный терм* с этим же функтором и арностью, третий аргумент — число `43`. Вложенный терм `z(inter,Near,8)` содержит в качестве второго аргумента переменную `Near`, т. к. это заглавный идентификатор.

2.2. Переменные

Переменные в Прологе пишутся заглавными идентификаторами и могут быть либо свободными, либо связанными. Пока переменная *не связана* каким-либо значением, она называется *свободной*. Как только свободная переменная *связывается* каким-либо значением, она становится *связанной* и после этого не может изменить своего значения, будто является константой. Такой механизм связывания переменных называется *неразрушающим* присваиванием, в отличие от *разрушающего* присваивания, используемого в императивных языках и допускающего многократное изменение значения переменной.

Термин «переменная» в применении к Прологу достаточно условен и является данью традиции, привнесенной из императивных языков. Переменная в Прологе не меняет свое значение и, будучи свободной, она приобретает его только один раз. Чтобы связанная переменная приобрела другое значение, ее надо «отвязать» от текущего значения с помощью процесса, называемого *откатом*, т. е. произвести своеобразное «развоплощение». Механизм отката будет описан в главе 6. К настоящему времени в теории языков программирования не придуман такой термин, который бы адекватно отражал суть «переменных» Пролога с однократным связыванием. Поэтому мы будем использовать термин «переменная», надеясь на правильное его понимания читателем.

Термы, в которые не входят свободные переменные, называются *основными*. Термы, содержащие свободные переменные, называются *неосновными*.

Терм `f(999, atm)` является основным, а терм `z(inter, Near, 8)` — неосновным, терм `z(babilon, z(inter, Near, 8), 43)` также является неосновным, поскольку один из его аргументов — неосновной терм.

В Прологе используются и *анонимные* переменные — в тех случаях, где согласно синтаксису должна находиться переменная, но ее значение в вычислениях не используется. Анонимные переменные выражаются знаком подчеркивания или идентификатором, начинающимся со знака подчеркивания.

Мы рассмотрели *префиксные* термы, т. е. термы, функтор которых расположен слева от аргументов, перечисленных в скобках. В Прологе используются также *инфiksные* двухарные термы, у которых функтор располагается между аргументами и обозначается не строчным идентификатором, а символом арифметической или логической операции. Такие термы призваны повысить читабельность исходного текста программы. Например, вместо префиксного терма проверки неравенства `>(A, 5)` в Прологе используют инфиксную запись этого терма `A>5`. Такая запись более привычна для человека и, несмотря на отсутствие скобочной нотации и строчного идентификатора в качестве функтора, является допустимым в Прологе термом.

2.3. Списки

Наряду с префиксными и инфиксными термами в Пролог для обозначения односвязных списков введен безфункторный терм, описанный в *главе 1. Список* есть специальный вид двоичного безфункторного рекурсивного терма, у которого первый аргумент — это голова списка, а второй аргумент является хвостом списка. Разделителем между ними служит вертикальная черта. Все элементы списка должны принадлежать к одному типу.

Списочный терм правоассоциативен. Это означает, что список, содержащий три элемента, — например, 10, 11 и 12, с теоретической точки зрения представляет собой терм `[10 | [11 | [12 | []]]]`. Такая запись списка трудна для восприятия, поэтому для упрощения записи и облегчения восприятия списки обозначаются кратко — в виде перечисления через запятую элементов, — например, `[10, 11, 12]`. Подобные упрощения синтаксиса в теории языков программирования называют «*синтаксическим сахаром*». Так, синтаксическим сахаром являются следующие варианты записи этого списка:

```
[10 | [11, 12]]
[10, 11 | [12]]
[10, 11, 12 | []]
```

Далее представлены неправильные записи списков:

- `[1, 2 | 3]` — хвост списка должен быть списком `[3]`, а не элементом 3;
- `[1, "aaa", 7]` — элементы списка должны принадлежать одному типу;
- `[1 | [2 | [3]]]` — список имеет два хвоста, а допускается только один.

Если элементами списка являются тоже списки, то они называются *вложенными списками*, или списками первого уровня вложенности. Список может иметь произвольное число уровней вложенности.

Вертикальная черта, обозначающая разделитель между головой и хвостом, является операцией префикса списка. Префикс списка может быть как конструктором списка, так и селектором списка.

Конструктор собирает новый список из данного ему элемента, служащего головой нового списка, и данного ему списка, служащего хвостом нового списка. Например, если имеется элемент 5 и список [10, 20, 30], то новый список можно построить так: [5 | [10, 20, 30]]. Эту же операцию можно выразить формально:

```
A = 5,  
X = [10, 20, 30],  
NewList = [A | X].
```

В результате:

```
NewList = [5, 10, 20, 30].
```

Селектор разделяет заданный список на первый элемент, т. е. голову списка, и остаток списка, т. е. хвост. Например, если задан список [5, 10, 20, 30], то его можно разделить на голову и хвост, используя две свободные переменные, следующим образом:

```
[A | X] = [5, 10, 20, 30].
```

Результатом будет являться голова 5 и остаток списка [10, 20, 30]:

```
A = 5,  
X = [10, 20, 30].
```

Селектор не применим к пустому списку [], т. к. пустой список не разделяется на голову и хвост. Однако список, содержащий лишь один элемент, разделяется на голову — этот самый единственный элемент, и хвост — пустой список.

И конструктор, и селектор синтаксически обозначаются одинаково — вертикальной чертой. Различие заключается в том, связаны или свободны переменные, между которыми стоит вертикальная черта. Если переменные связаны, то конструируется новый список. Если переменные свободны, то разделяется тот список, который приравнивается к этим свободным переменным, разделенным вертикальной чертой. Такое приравнивание в Прологе выполняет операцию сопоставления с образцом. Эта же операция сопоставления с образцом обрабатывает те случаи сопоставления, когда список содержит как свободные, так и связанные переменные.

2.4. Операция сопоставления с образцом

Сопоставление с образцом (pattern matching) — это операция сопоставления двух термов с целью поиска множества подстановок. Под *подстановкой* понимается замена всех вхождений переменной в терме каким-либо основным термом. *Множество подстановок* — это замена всех переменных терма основными термами.

Общий терм, полученный после замены переменных их значениями, должен быть основным, т. е. не должен содержать свободных переменных. Такое ограничение, наряду с отсутствием проверки вхождения переменной в терм, с которым сопоставляется эта переменная, не только повышает скорость выполнения кода, но и увеличивает его надежность за счет глобального анализа потока данных на этапе компиляции.

2.4.1. Сопоставление термов

Сопоставление термов выполняется за три этапа:

1. Проверка равенства функций сопоставляемых термов.
2. Проверка равенства арности сопоставляемых термов.
3. Поиск множества подстановок.

Если подстановки всех переменных найдены и являются основными термами, то сопоставление с образцом считается успешным, иначе — сопоставление с образцом считается неуспешным, и замена переменных найденными значениями не производится.

Примеры сопоставления с образцом двух термов приведены в табл. 2.1.

Таблица 2.1. Примеры сопоставления с образцом

№	Терм 1	Терм 2	Сопоставление	Подстановки	Общий терм
1	X	24	Успешно	X=24	24 — основной
2	X	f(99)	Успешно	X=f(99)	f(99) — основной
3	ace()	X	Успешно	X=ace()	ace() — основной
4	ace()	ace(88)	Ошибка этапа компиляции	Нет	Нет
5	ace(88)	face(88)	Неуспешно	Нет	Нет
6	X	Y	Ошибка этапа компиляции	Нет	Нет
7	t(X, 45, Y)	t(2, Z, 7)	Успешно	X=2, Y=7, Z=45	t(2, 45, 7) — основной
8	t(X, 45, Y)	t(Z, Z, 8)	Ошибка этапа компиляции	Нет	Нет
9	t(X, 45, Y)	t(Z, 45, 8)	Ошибка этапа компиляции	Нет	Нет
10	t(X, 45, Y)	t(Z, X, W)	Ошибка этапа компиляции	Нет	Нет
11	t(X, 45, Y)	g(2, 45, 7)	Неуспешна	Нет	Нет
12	t(X, 45, Y)	t(X, 45)	Ошибка этапа компиляции	Нет	Нет

Таблица 2.1 (окончание)

№	Терм 1	Терм 2	Сопоставление	Подстановки	Общий терм
13	$t(X, 45, 8)$	$t(2, Z, Z)$	Неуспешно	Нет	Нет
14	X	$s(X)$	Ошибка этапа компиляции	Нет	Нет
15	$ace(88)$	$ace(88)$	Успешно	Нет	$ace(88)$ — основной

В строке 4 табл. 2.1 Visual Prolog допускает использование в программе двух термов с одинаковым именем, но с разной арностью. Такие термы считаются разными, что видно в строках 4 и 12.

Сопоставление двух термов, указанных в строке 8, не будет скомпилировано, т. к. в ходе глобального анализа потока данных компилятор обнаружит, что одной из промежуточных подстановок является подстановка $X=Z$, т. е. связывание двух свободных переменных, что не допускается алгоритмом сопоставления с образцом.

Сопоставление терма с анонимной переменной всегда успешно.

Операция сопоставления с образцом выражается явно знаком равенства — например, $t(X, 45, Y) = t(2, Z, 7)$. С другой стороны, сопоставление с образцом выполняется без явного указания знака равенства — например, при вызове функции с заданными параметрами. Параметры вызова сопоставляются с соответствующими параметрами, заданными в определении функции.

Операция сопоставления с образцом безразлична к расположению свободных переменных относительно знака равенства. Поэтому операции $X=7$ и $7=X$ эквиваленты.

2.4.2. Сопоставление списков

Списки являются безфункциональными термами, поэтому сопоставление списков выполняется аналогично сопоставлению термов — последовательным сопоставлением элементов двух списков между собой. Если в списках есть вложенные списки, то выполняется сопоставление их элементов. В табл. 2.2 приведены примеры сопоставления двух списков.

Таблица 2.2. Примеры сопоставления с образцом для списков

№	Список 1	Список 2	Сопоставление с образцом	Подстановки	Общий терм
1	X	$[10, 20, 30]$	Успешно	$X=[10, 20, 30]$	$[10, 20, 30]$
2	$[X, 20, 30]$	$[10, 20, 30]$	Успешно	$X=10$	$[10, 20, 30]$
3	$[X, 20, 30]$	$[10, Y, 30]$	Успешно	$X=10, Y=20$	$[10, 20, 30]$
4	$[X, 20, 30]$	$[10, Y, 50]$	Неуспешно	Нет	Нет
5	$[X, 20, 30]$	$[X, 20]$	Неуспешно	Нет	Нет

Таблица 2.2 (окончание)

№	Список 1	Список 2	Сопоставление с образцом	Подстановки	Общий терм
6	[X Z]	[10, 20, 30]	Успешно	X=10, Z=[20, 30]	[10, 20, 30]
7	[X Z]	[10]	Успешно	X=10, Z=[]	[10]
8	[X Z]	[]	Неуспешно	Нет	Нет
9	[X, Y, Z]	[X, Y, Z]	Ошибка этапа компиляции	Нет	Нет
10	[X, X, Z]	[10, 10, 30]	Успешно	X=10, Z=30	[10, 10, 30]
11	[X, X]	[10, 10, 30]	Неуспешно	Нет	Нет
12	[X [2, 3]]	[1 Z]	Успешно	X=1, Z=[2, 3]	[1, 2, 3]
13	[1 [X, 3]]	[1 Z]	Ошибка этапа компиляции	Нет	Нет
14	[X [2, 3]]	[1 [Z, 3]]	Успешно	X=1, Z=2	[1, 2, 3]
15	[X [2, 3]]	[1 [Z]]	Неуспешно	Нет	Нет
16	X	[X, 20, 30]	Ошибка этапа компиляции	Нет	Нет
17	X	[1 [2, 3]]	Успешно	X=[1, 2, 3]	[1, 2, 3]
18	[10, 20, 30]	[10, 20, 30]	Успешно	Нет	[10, 20, 30]

Сравнивая Visual Prolog с императивными языками программирования, следует заметить, что сопоставление с образцом в Visual Prolog выполняет одну из двух операций:

- если одна из переменных свободна, а другая переменная связана значением, то операция — например, $28=X$, выполняет присваивание переменной X числа 28;
- если обе переменные связаны значениями, то выполняется проверка условия равенства.

2.5. Операция унификации

Понятие *унификации* (unification) шире понятия сопоставления с образцом. Унификация допускает подстановку вместо одной переменной другой свободной переменной или неосновного терма. Поэтому результатом унификации может являться неосновной терм. В математической логике при унификации производится проверка вхождения переменной в терм, с которым сопоставляется эта переменная. Однако на практике проверку вхождения переменной в терм, с которым эта переменная унифицируется, не выполняет большинство Прологов. Это сделано с целью повышения скорости выполнения программ. Однако в некоторых Прологах — например, в SWI Prolog, есть специальный предикат для принудительной проверки вхождения переменной.

Синтаксически операция сопоставления с образцом и операция унификации одинаковы. Они различаются только тем, является ли результат основным термом или неосновным. В ходе выполнения программы Visual Prolog использует унификацию. Однако конечный результат работы программы в Visual Prolog должен быть только основным термом, и, следовательно, в ходе цепочки унификаций должен получаться основной терм.

Для сравнения унификации и сопоставления с образцом в табл. 2.3 приведены примеры унификации термов из табл. 2.1.

Таблица 2.3. Примеры унификации термов

№	Терм 1	Терм 2	Унификация	Подстановки	Общий терм
1	X	24	Успешна	X=24	24 — основной
2	X	f(99)	Успешна	X=f(99)	f(99) — основной
3	ace()	X	Успешна	X=ace()	ace() — основной
4	ace()	ace(88)	Ошибка этапа компиляции	Нет	Нет
5	ace(88)	face(88)	Неуспешна	Нет	Нет
6	X	Y	Успешна	X=Y, Y=X	X — неосновной Y — неосновной
7	t(X, 45, Y)	t(2, Z, 7)	Успешна	X=2, Y=7, Z=45	t(2, 45, 7) — основной
8	t(X, 45, Y)	t(Z, Z, 8)	Успешна	X=45, Y=8, Z=45	t(45, 45, 8) — основной
9	t(X, 45, Y)	t(Z, 45, 8)	Успешна	X=Z, Y=8, Z=X	t(X, 45, 8) — неосновной t(Z, 45, 8) — неосновной
10	t(X, 45, Y)	t(Z, X, W)	Успешна	X=45, Y=8, Y=W, W=Y	t(45, 45, Y) — неосновной t(45, 45, W) — неосновной
11	t(X, 45, Y)	g(2, 45, 7)	Неуспешна	Нет	Нет
12	t(X, 45, Y)	t(X, 45)	Ошибка этапа компиляции	Нет	Нет
13	t(X, 45, 8)	t(2, Z, Z)	Неуспешна	Нет	Нет
14	X	s(X)	Ошибка этапа компиляции	Нет	Нет
15	ace(88)	ace(88)	Успешна	Нет	ace(88) — основной

Для сравнения унификации и сопоставления с образцом в табл. 2.4 приведены примеры унификации списков из табл. 2.2.

Таблица 2.4. Примеры унификации списков

№	Список 1	Список 2	Унификация	Подстановки	Общий терм
1	X	[10, 20, 30]	Успешна	X=[10, 20, 30]	[10, 20, 30]
2	[X, 20, 30]	[10, 20, 30]	Успешна	X=10	[10, 20, 30]
3	[X, 20, 30]	[10, Y, 30]	Успешна	X=10, Y=20	[10, 20, 30]
4	[X, 20, 30]	[10, Y, 50]	Неуспешна	Нет	Нет
5	[X, 20, 30]	[X, 20]	Неуспешна	Нет	Нет
6	[X Z]	[10, 20, 30]	Успешна	X=10, Z=[20, 30]	[10, 20, 30]
7	[X Z]	[10]	Успешна	X=10, Z=[]	[10]
8	[X Z]	[]	Неуспешна	Нет	Нет
9	[X, Y, Z]	[X, Y, Z]	Успешна	Нет	[X, Y, Z]
10	[X, X, Z]	[10, 10, 30]	Успешна	X=10, Z=30	[10, 10, 30]
11	[X, X]	[10, 10, 30]	Неуспешна	Нет	Нет
12	[X [2, 3]]	[1 Z]	Успешна	X=1, Z=[2, 3]	[1, 2, 3]
13	[1 [X, 3]]	[1 Z]	Успешна	Z=[X, 3]	[1, X, 3]
14	[X [2, 3]]	[1 [Z, 3]]	Успешна	X=1, Z=2	[1, 2, 3]
15	[X [2, 3]]	[1 [Z]]	Неуспешна	Нет	Нет
16	X	[X, 20, 30]	Ошибка этапа компиляции	Нет	Нет
17	X	[1 [2, 3]]	Успешно	X=[1, 2, 3]	[1, 2, 3]
18	[10, 20, 30]	[10, 20, 30]	Успешно	Нет	[10, 20, 30]

2.6. Операция «должно унифицироваться»

Операция «должно унифицироваться» (must unify) выражается двумя знаками равенства == и выполняет унификацию. Различие между унификацией и «должно унифицироваться» состоит в следующем. Если два терма X и Y не унифицируются, то операция «должно унифицироваться» X==Y завершится вызовом обработчика возникшей исключительной ситуации, в то время как операция унификации X=Y вызовет откат назад.

Операция «должно унифицироваться» унифицирует две связанных переменные и не может оперировать со свободными переменными.

2.7. Неразрушающее присваивание в Visual Prolog

Как таковой, оператор неразрушающего присваивания в Visual Prolog отсутствует. Его роль играет унификация. Поскольку переменная в Visual Prolog может находиться только в двух состояниях: быть свободной или связанной, то присвоить значение можно только свободной переменной и только один раз. После этого присвоить ей новое значение, разрушая прежнее, нельзя. Другими словами, в Visual Prolog отсутствует так называемое *разрушающее присваивание*.

Рассмотрим следующий пример, показывающий неправильное использование переменной:

```
X = 17,      % свободная переменная X связывается значением 17  
X = 10,      % ложное сопоставление, т. к. 17 ≠ 10
```

В этом примере свободная переменная X связывается значением 17. Поэтому сопоставление связанной переменной $X = 10$ Visual Prolog воспринимает как $17 = 10$, что является ложью.

Рассмотрим пример неправильного инкремента некоторого значения:

```
X = 17,      % свободная переменная X связывается значением 17  
X = X + 1,   % ложное сопоставление, т. к. 17 ≠ 18
```

Обратите внимание, что с точки зрения математики запись $X = X + 1$ также является тождественно ложным утверждением.

Правильный инкремент некоторого значения выглядит так:

```
X = 17,      % свободная переменная X связывается значением 17  
Y = X + 1,   % свободная переменная Y связывается значением 18
```

Здесь мы ввели новую сущность — свободную переменную Y , с которой и связали значение инкремента. В результате выполнения примера осуществляется подстановка значения 17 вместо переменной X и подстановка значения 18 вместо переменной Y . При этом оба сопоставления истинны.

Так как операция сопоставления является ассоциативной, то инкремент можно записать и в таком виде:

```
17 = X,      % свободная переменная X связывается значением 17  
X + 1 = Y,   % свободная переменная Y связывается значением 18
```

ГЛАВА 3



Операции языка

3.1. Арифметические операции и операции сравнения и сопоставления

В табл. 3.1 в порядке уменьшения приоритета перечислены арифметические операции и операции сравнения. Кроме этих операций Visual Prolog имеет логические операции `and`, `or`, `not`, но они применяются не к булевым значениям, а к значениям истинности предикатов, и будут рассмотрены в главе 6. Операции над булевыми значениями не встроены в язык. Они описаны в классе `boolean` в виде предикатов с именами `logicalNot/1`, `logicalAnd/2`, `logicalOr/2` и `logicalXor/2`.

Таблица 3.1. Операции языка

Операции	Описание операции	Примечание
$^$	Возведение в степень	
$-$	Унарный минус	
\ast , $/$	Умножение и деление	Имеют одинаковый приоритет
<code>div</code> , <code>mod</code>	Целая часть и остаток от целочисленного деления, округленные в меньшую сторону	Не определены для вещественных чисел, имеют одинаковый приоритет
<code>quot</code> , <code>rem</code>	Целая часть и остаток от целочисленного деления, округленные в сторону нуля	
$+$, $-$	Сложение и вычитание	Имеют одинаковый приоритет
$>$, $<$, \neq , \geq , \leq , $=$, $==$	Больше, меньше, неравно, больше или равно, меньше или равно, унификация, должно унифицироваться	Являются предикатами

Операция возведения в степень правоассоциативна. Все другие бинарные операции левоассоциативны. Операции сравнения неассоциативны.

Операция неравенства \neq не является двойственной к операции сопоставления $=$. Неравенство \neq сравнивает два терма, тогда как сопоставление $=$ унифицирует два

терма. Операцией, двойственной к сопоставлению $X=Y$, является операция отрицания $\text{not}(X=Y)$.

В табл. 3.2 показан порядок вычисления значений выражений согласно принятым в Visual Prolog приоритетам операций, а также примеры недопустимых в Visual Prolog выражений.

Таблица 3.2. Порядок вычисления значений выражения

№	Выражение	Порядок вычисления
1	-2^2	$-(2^2)$
2	3^2^2	$3^{(2^2)}$
3	$-2*-3^2+5$	$(((-2) * (-3 ^ (2)))) + 5$
4	$29 \bmod 15 \operatorname{div} 4$	$(29 \bmod 15) \operatorname{div} 4$
5	$8-29 \bmod 15$	$8-(29 \bmod 15)$
6	$15-7 \geq 20 \bmod 7$	$(15-7) \geq (20 \bmod 7)$
7	$3>2 = 7<9$	Синтаксическая ошибка
8	$(X=3) == (6=Z)$	Синтаксическая ошибка

Результат операций сравнения — например: $3>2$ и $7<9$, принимает значение из области {Истина, Ложь}. Но к этим значениям не применима операция унификации. Операция унификации применяется только к термам.

Результат операций унификации $X=3$ и $6=Z$ принимает значение из области {Истина, Ложь}. Но к этим значениям не применима операция «должно унифицироваться». Операция «должно унифицироваться» применяется только к основным термам.

3.2. Целочисленное деление

Операции целочисленного деления `div` и `quot` при положительных результатах работают одинаково, но при отрицательных результатах `div` округляет в сторону бесконечности, в то время как `quot` округляет в сторону нуля. Операции остатка от целочисленного деления: `mod` — остаток для `div`, а `rem` — остаток для `quot`. Пример работы целочисленных операций приведен в табл. 3.3.

Таблица 3.3. Операции целочисленного деления

A	B	A div B	A mod B	A quot B	A rem B
15	7	2	1	2	1
-15	7	-3	6	-2	-1
15	-7	-3	-6	-2	1
-15	-7	2	-1	2	-1

ГЛАВА 4



Типы, домены, подтипы

4.1. Типы данных

Visual Prolog является языком со статической типизацией. Это означает, во-первых, что контроль типов осуществляется во время компиляции и, во-вторых, что совместимость типов должна быть явно указана или наследована при определении типа. Visual Prolog — единственный из Прологов, который имеет статическую типизацию данных. Это дает ему следующие преимущества перед нетипизированными Прологами:

- надежность исполнимого кода. В ходе глобального анализа потока данных компилятор проверяет соответствие типов и не допускает унификации разных термов, операций над термами разных типов, передачи неверных параметров в исполнимом коде;
- скорость исполнимого кода является непревзойденной среди всех Прологов.

Типы Visual Prolog разделяются на типы объектов и типы данных. Типы объектов будут рассмотрены в главе о классах языка Visual Prolog. Здесь же мы рассмотрим типы данных.

Тип данных в Visual Prolog, как собственно и в других языках, определяет множество значений, которое может принимать переменная, а также набор операций над ней и способ хранения значения этой переменной. Кроме этого в Visual Prolog введено понятие домена. В технологии баз данных под доменом понимают множество всех допустимых атомарных значений поля. Это утверждение справедливо и для Visual Prolog, но требует подробного рассмотрения.

Домен — именованное множество значений некоторого типа данных или других доменов, которое имеет смысловое значение, выражаемое его именем.

В Visual Prolog домены определяют множество значений каждого аргумента терма, вне зависимости от того, какие имена переменных будут использованы в программе для обозначения этих аргументов.

Применение доменов позволяет проводить строгий контроль типов и обнаруживать большую часть смысловых и синтаксических ошибок в ходе компиляции.

Например, инвентарный номер тумбочки и количество тумбочек, находящихся в некоторой организации, можно выразить целыми неотрицательными числами, которые в Visual Prolog обозначаются `unsigned`. Инвентарный номер и количество тумбочек имеют совершенно разный смысл. Однако по невнимательности программист может, например, вместо сравнения инвентарных номеров при поиске, ошибочно сравнить инвентарный номер и количество тумбочек. Такая операция сравнения явно не имеет смысла. Чтобы во время компиляции контролировать подобные ошибки, Visual Prolog дает программисту возможность определить множество значений `unsigned` для инвентарного номера и, отдельно от него, другое множество значений `unsigned` для количества тумбочек. Хотя эти два множества по сути своей принадлежат типу `unsigned`, но они имеют разные имена, и, следовательно, домены для инвентарного номера и для количества тумбочек не пересекаются.

Домен в Visual Prolog может быть определен рекурсивно, аналогично рекурсивному терму. Рекурсивное определение доменов используют для задания деревьев произвольного типа.

В программе раздел объявления доменов начинается с ключевого слова `domains` и заканчивается там, где начинается другой раздел программы или встречается конец файла. Специального ключевого слова для обозначения конца раздела объявления доменов в Visual Prolog нет. В программе может быть несколько разделов объявления доменов. Объявлять домены в классе можно и после их использования. Компилятор Visual Prolog многопроходный и поэтому правильно распознает такие коллизии, однако хорошим тоном считается определение доменов до их использования.

4.2. Встроенные домены

В Visual Prolog предопределено 20 базовых доменов, рассмотренных далее.

1. `any` — универсальный домен. Значение этого домена содержит ссылку на библиотеку доменов термов и непосредственно терм. Значения домена `any` получаются только путем преобразования с помощью функции `toAny` из других доменов. Обратное преобразование возможно с помощью встроенной функции `convert/2`. При этом необходимо знать исходный домен. Над значениями этого домена возможны операции сопоставления с образцом и операции сравнения. В табл. 4.1 представлен ряд значений домена `any`.

Таблица 4.1. Значения домена `any`

Значение домена <code>any</code>	Исходный домен	Значение
<code>@any(00442CA4, -56)</code>	<code>integer</code>	<code>-56</code>
<code>@any(00442CC0, 3)</code>	<code>unsigned</code>	<code>3</code>
<code>@any(00442CE8, 23.567)</code>	<code>real</code>	<code>23.567</code>
<code>@any(00442D3C, "qwerty")</code>	<code>string</code>	<code>"qwerty"</code>
<code>@any(00442D6C, 'r')</code>	<code>symbol</code>	<code>'r'</code>
<code>@any(00442D80, \$[0A,0B,0C])</code>	<code>binary</code>	<code>\$[0A,0B,0C]</code>

Таблица 4.1 (окончание)

Значение домена any	Исходный домен	Значение
@any(00442EDC, true())	boolean	true()
@any(0045283C, [3,2,3])	integer_list	[3,2,3]
@any(00452874, ["qwerty","yyy"])	string_list	["qwerty","yyy"]
@any(004528C0, ['r','t'])	symbol_list	['r','t']

Домен `any` введен в язык главным образом для поддержки предикатов с произвольным числом аргументов — эллипсисом. Реализация эллипсиса осуществляется с помощью встроенных функций `toEllipsis/1` и `fromEllipsis/1`.

2. `binary` — последовательность байтов. Значения этого домена используются для хранения двоичных данных. Значение `binary` реализовано как указатель на последовательность байтов, которые представляют содержимое двоичного терма.

Длина терма `binary` находится в четырехбайтовом поле, непосредственно предшествующем этой последовательности байтов. Поле длины содержит следующее значение:

`TotalNumberOfBytesOccupiedByBinary = ByteLen + 4`

где `ByteLen` — это длина терма `binary` и 4 — это количество байтов, занимаемых полем длины. Размер двоичных данных теоретически может превосходить 4 Гбайт.

К значениям домена `binary` применимы только операции присваивания и сравнения (в лексикографическом порядке). Два терма `binary` сравниваются следующим образом:

- если они различны по длине, то большим считается тот терм, который длиннее;
- иначе, термы сравниваются побайтно. Сравнение завершается, когда найдены два различных байта, и результат сравнения этих байтов является результатом сравнения термов;
- два терма `binary` равны, если равна их длина и равны соответствующие друг другу байты.

Синтаксис текста для терма `binary` определяется следующими правилами:

- каждое выражение должно быть вычислено во время компиляции;
- их значения должны быть в диапазоне от 0 до 255.

3. `binaryNonAtomic` — последовательность байтов. Подобно `binary`, но может содержать указатели `pointer`.

Различие между `binary` и `binaryNonAtomic` заключается в обработке их сборщиком мусора. Память `binary` не сканируется сборщиком мусора при поиске указателей, т. к. содержит только неделимые данные: числа, символы, значения `handle`. Память `binaryNonAtomic` сканируется, т. к. содержит указатели, строки, объекты, двоичные данные.

4. `boolean` — булевы значения. Он реализован как обычный составной домен со следующим определением:

```
boolean = false(); true();
```

5. `char` — широкий (двухбайтовый) символ. Значения домена: Unicode-символы, они реализованы как два беззнаковых байта. К значениям этого домена применимы операторы унификации. Сравнение символов осуществляется в лексикографическом порядке путем сравнения кодов этих символов.

6. `compareResult` — этот домен используется, как правило, для определения результата сравнения двух составных термов, к которым неприменимы операции сравнения. Домен определяет три результата сравнения: `less` (меньше), `equal` (равно), `greater` (больше):

```
compareResult = less; equal; greater.
```

7. `integer` — целые числа со знаком. Значения этого домена занимают 4 байта (32 бита). К значениям домена `integer` применяются арифметические операции (`+`, `-`, `/`, `*`, `^`), а также сравнение, унификация, операции `div`, `mod`, `quot` и `rem`. Допустимый числовой диапазон: от -2^{31} до $2^{31}-1$, или от -2 147 483 648 до 2 147 483 647 включительно.

8. `integer64` — целые числа со знаком. Значения этого домена занимают 8 байтов (64 бита). Допустимый числовой диапазон: от -2^{63} до $2^{63}-1$, или от -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 включительно. Набор допустимых операций над `integer64` подобен набору для `integer`.

9. `integerNative` — целые числа со знаком. Диапазон чисел определяется платформой.

10. `factDB` — дескрипторы именованной базы данных. Этот домен имеет следующее скрытое метаобъявление:

```
factDB = struct @factdb(named_internal_database_domain, object).
```

Все имена секций БД, определенные пользователем, являются константами этого домена. Компилятор автоматически строит соответствующие составные термы из таких констант всякий раз, когда это необходимо. Во время выполнения программы первое поле этой структуры содержит адрес, соответствующий дескриптору домена, второе поле содержит либо ноль (для секции `class facts`), либо указатель на объект, т. е. `This` (для объекта `facts sections`).

11. `handle` — значения домена `handle` используются для вызова API-функций Windows и имеют размер 4 байта. Для этого домена не существует операций, и его значения не могут быть конвертированы в другие домены или из других доменов (исключая функцию `uncheckedConvert`). Для домена `handle` имеются встроенные константы `nullHandle` и `invalidHandle`.

12. `pointer` — указатель на адрес памяти. Значение домена `pointer` прямо ссылается на адрес памяти. К значениям этого домена применяются операции равенства. Для домена `pointer` имеется встроенная константа `null`, означающая нулевой адрес, который не является реальным адресом и служит только для обозна-

чения того, что указатель в данный момент не может использоваться для обращения к памяти.

13. `real` — число с плавающей точкой двойной точности (длинное вещественное число). Значения этого домена занимают 8 байтов. Все арифметические операции, а также операции сравнения и унификации применимы для значений домена `real`. Допустимый числовoy диапазон: от $\pm 2.23e-308$ до $\pm 1.7e+308$. Точность представления чисел — не более 16 десятичных разрядов. При необходимости, значения целочисленных доменов автоматически конвертируются в домен `real`.
14. `real32` — число с плавающей точкой одинарной точности (короткое вещественное число). Значения этого домена занимают 4 байта. Все арифметические операции, а также операции сравнения и унификации применимы для значений домена `real32`. Допустимый числовой диапазон: от $\pm 1.8e-38$ до $\pm 3.4e+38$. Точность представления чисел — не более восьми десятичных разрядов.
15. `string` — нуль-законченная последовательность широких (двухбайтовых) символов. Она реализована как указатель на нуль-законченный массив широких символов. К значениям этого домена применима унификация. Сравнение строк осуществляется в лексикографическом порядке. В языке нет средств для обращения к символам строки как к элементам списка, подобно тому, как это сделано в нетипизированных Прологах. Также в языке нет средств для обращения к символам строки как к элементам одномерного массива — по индексу. Однако подобные средства предоставляет класс `string`, входящий в базовые классы Prolog Foundation Classes.
16. `string8` — терм встроенного домена `string8` — это последовательность символов ASCII (однобайтовых). Домен реализован как указатель на нуль-законченный массив ASCII-символов. К значениям этого домена применима унификация. Сравнение строк осуществляется в лексикографическом порядке.
17. `symbol` — нуль-законченная последовательность широких символов. Подобно домену `string`, домен `symbol` — это также последовательность Unicode-символов. Домен реализован как указатель на вход в таблицу символов, которая содержит строки. Операции, которые могут быть применены к символам, те же, что и к строкам.

Символьные последовательности и строки, как правило, взаимозаменяемы, но хранятся различными способами. Символьные последовательности хранятся в таблице, и их адреса, а не сами символы, используются для операций присваивания и сравнения. Это означает, что символьные последовательности обрабатываются очень быстро, и если даже символьная последовательность многократно встречается в программе, это не увеличит размер исполняемого кода, т. к. на самом деле эта последовательность хранится в таблице в единственном экземпляре. В отличие от символьных последовательностей, строки не хранятся в поисковой таблице. Поэтому Visual Prolog проверяет строки посимвольно всякий раз, когда они унифицируются или сравниваются.

18. `unsigned` — целые числа без знака. Значения этого домена занимают 4 байта (32 бита). К значениям домена `unsigned` применяются арифметические операции (+, -, /, *, ^), а также сравнение, унификация, операции `div`, `mod`, `quot` и `rem`. Допустимый числовой диапазон: от 0 до $2^{32} - 1$, или до 4 294 967 295 включительно. Использование унарного минуса для `unsigned` недопустимо.
19. `unsigned64` — целые числа без знака. Значения этого домена занимают 8 байтов (64 бита). Допустимый числовой диапазон: от 0 до $2^{64} - 1$, или до 18 446 744 073 709 551 615 включительно. Использование унарного минуса для `unsigned64` недопустимо. Набор допустимых операций над `unsigned64` подобен набору для `unsigned`.
20. `unsignedNative` — целые числа без знака. Диапазон чисел определяется платформой.

4.3. Домены-синонимы

Домены-синонимы предназначены для создания идентичных по области определения доменов, но используются для указания типа различных по смыслу аргументов.

Рассмотрим следующий код:

```
domains
    адрес = string.
    фамилия = string.
```

Здесь вводятся два новых домена `адрес` и `фамилия`, которые идентичны домену `string` с точки зрения множества значений. Если в каком-либо терме есть аргумент, принадлежащий домену `адрес`, то на месте этого аргумента не может быть использована переменная, принадлежащая домену `фамилия`. Visual Prolog позволяет принудительно конвертировать домен `адрес` в домен `фамилия` и наоборот.

Имена доменов являются строчными идентификаторами. Доменные имена используются при объявлении новых доменов, констант, фактов и предикатов, а также при принудительном преобразовании одних типов данных в другие.

4.4. Подтипы и категориальный полиморфизм

Типы данных организованы в иерархию подтипов. *Подтипы* служат для организации категориального полиморфизма. Категориальный полиморфизм определяет, что категория (тип) может быть представлена любой своей подкатегорией (подтипом). Если в программе некоторая переменная получает значение не своего типа, а подтипа своего типа, то это вполне допускается и называется *категориальным полиморфизмом*.

В оригинальном описании Visual Prolog типы и подтипы называются доменами в тех случаях, когда различия между ними не рассматриваются. Когда же рассматриваются именно различия, то используются термины *тип* и *подтип*. Мы поступим так же.

Подтипы должны явно наследоваться от своих типов. Если такого явного указания при объявлении подтипа нет, то даже если математически подтип и является подмножеством некоторого типа, но он не будет его подтипом, и категориальный полиморфизм в этом случае работать не будет.

Имена доменов (типов, подтипов) должны быть строчными идентификаторами. Рассмотрим следующее объявление доменов:

```
domains
```

```
t1 = [1..17].  
t2 = [5..13].  
t3 = t1 [5..13].  
t4 = t3 [6..13].  
t5 = t4.
```

Тип `t1` содержит целые числа от 1 до 17 включительно. Также и `t2` содержит целые числа от 5 до 13 включительно. Математически `t2` есть подмножество `t1`. Однако `t2` не является подтипов `t1`, т. к. это явно не указано при объявлении `t2`. С другой стороны подтип `t3`, содержащий те же значения, что и `t2`, является подтипов типа `t1`, т. к. это объявлено явно. В свою очередь `t3` имеет подтип `t4`, который отличается от него левой границей. Тип `t5` является подтипов типа `t4`.

Visual Prolog содержит несколько неявных отношений подтип–тип между встроенными доменами — например, тип `integer` является подтипов типа `integer64`. Пользовательские же подтипы должны объявляться явно.

4.5. Домены, определяемые пользователем

Как вы уже увидели, объявление пользовательского домена включает имя определяемого домена, знака равно и указание терма или термов через точку с запятой. Объявление домена заканчивается точкой.

4.5.1. Целочисленные подтипы

В Visual Prolog существует три способа указания пользовательских целочисленных подтипов. Первый способ заключается в указании нижней и/или верхней границы значений подтипа:

```
domains
```

```
возраст = [0..120].      % неявный родительский тип unsigned  
страница = [1..2000].    % неявный родительский тип unsigned  
баланс = [-100..100].    % неявный родительский тип integer  
neg = [..-1]              % неявный родительский тип integer,  
                          % диапазон [-2147483648..-1]  
mm = [88..]                % неявный родительский тип unsigned,  
                          % диапазон [88..4294967295]
```

Левую или правую границу диапазона можно не указывать — тогда эта граница совпадает соответственно с левой или правой границей родительского диапазона.

Обратите внимание, что при таком способе определения домена возраст значения домена занимают в памяти 4 байта, хотя реально требуется всего 7 битов.

Границы допускается определять константами и выражениями, вычисляемыми во время компиляции, например:

```
constants
    lowBound = 0.
    upperBound = 120.
    pix = 50.

domains
    возраст = [lowBound..upperBound]. % неявный родительский тип
                                         % unsigned
    страница = [lowBound+1.. 40*pix]. % неявный родительский тип
                                         % unsigned
    баланс = [-2*pix.. 2*pix].       % неявный родительский тип integer
```

Второй способ предполагает явное указание имени типа, от которого производится подтип:

```
domains
    offset = integer.             % объявление домена-синонима
    countOpt = offset [-1...].    % родительский тип offset, но не integer
    count = countOpt [0...].      % родительский тип countOpt
    size = integer [-1..125].    % родительский тип integer
```

Третий способ позволяет точно указать размер памяти в битах, отводимый под значения домена. Для этого служит ключевое слово `bitsize`, после которого следует указать количество битов памяти, отводимых для хранения значений этого домена:

```
domains
    integer4 = bitsize 4.        % диапазон значений: от -8 до 7
    integer8 = bitsize 8.        % диапазон значений: от -128 до 127
    unsigned4 = bitsize 4 [0...]. % диапазон значений: от 0 до 15
    unsigned8 = bitsize 8 [0...]. % диапазон значений: от 0 до 255
    char8 = unsigned8.          % ASCII-символ
    флаг = bitsize 3 [1..7].    % диапазон значений: от 1 до 7
```

Обратите внимание, что для доменов `integer4` и `integer8` один из отводимых битов играет роль знака, поэтому степень у двойки на единицу меньше битового размера `bitsize`. Кроме того, поскольку способ хранения данных в памяти компьютеров байтовый, то значения типа `флаг`, хотя и объявлены трехбитовыми, но в памяти занимают все равно один байт, меньше не получится.

4.5.2. Вещественные подтипы

Вещественные подтипы образуются из типа `real` посредством уменьшения количества разрядов мантиссы с помощью ключевого слова `digits`, после которого указывается число разрядов мантиссы без незначащих нулей и, при необходимости, указания нижней и/или верхней границы диапазона вещественных чисел:

```

constants
    pi = 3.14159265358979.

domains
    zzz = digits 4.                      % четыре значащих цифры
    zzz_pos = digits 4 [0.0...].          % то же и неотрицательные числа
    aPeriod = digits 16 [-pi..pi].       % угол в радианах
    inCircle = digits 16 [-1.0..1.0].    % значения синуса и косинуса
    sr = digits 16 [...0].              % [-1.79769313486232e+308..0]

```

Максимальное число значащих разрядов вещественных чисел, которое поддерживается компилятором Visual Prolog, равно 16.

4.5.3. Списочные домены

В Visual Prolog списки являются односвязными и содержат элементы только одного домена. Для объявления списка достаточно справа от домена элемента списка поставить знак звездочки: *. Элементом списка может быть не только простой терм (число, строка, символ или строчный идентификатор), но также и составной терм — например: список, нерекурсивный или рекурсивный терм и т. п. Поэтому в Visual Prolog можно объявлять не только одноуровневые списки, но и многоуровневые списки (список списков, список списков списков и т. д.):

```

domains
    ppp = integer*      %домен ppp – список целых чисел
    abc = string**      %домен abc – список списков строк

```

Значением домена ppp может быть список, содержащий целочисленные элементы, например, [-2, 0, 78, -9, 4], а также может быть и пустой список []. Значением домена abc может быть список, содержащий списки строк, — например:

```
[ ["trace", "abc"], [], ["qwerty"] ],
```

а может быть и пустой список: [].

4.5.4. Составные домены

Составные домены могут быть заданы перечислением других доменов: простых, составных и даже указанием самого себя. В последнем случае такой домен будет рекурсивным:

```

domains
    пол = муж; жен.

    человек = персона(string Фамилия, unsigned Возраст, пол).

    number = i(integer); u(unsigned); r(real).

    numberList = number*.

    t = binTree(t LeftTree, integer Node, t RightTree); empty.

    tList = t*.

```

В приведенном примере домен пол составлен из двух нульварных термов муж и жен. Домен человек представляет собой нерекурсивную структуру с именем персона и

тремя аргументами. Первый аргумент здесь — строка, обозначающая фамилию. Это видно по комментарию, который начинается с заглавной буквы и расположен справа от домена. Второй аргумент — беззнаковое целое и предусмотрен для хранения возраста человека. Третий аргумент — это составной домен пол, объявленный ранее. Домен пол может принимать одно из двух значений: или муж, или жен.

Значением домена человек может быть терм:

персона ("Суриков", 18, муж).

Домен number представляет собой перечисление трех термов, аргументами которых служат целые, беззнаковые целые и вещественные числа соответственно. Подобные домены нередко применяются для представления и обработки чисел произвольного типа в калькуляторах, интерпретаторах и т. п.

Домен numberList служит классическим примером списка, по сути составленного из чисел разных типов. Для этого пришлось пойти на своеобразную хитрость и вначале объединить в домене number разнотипные числа, введя для каждого из них свой терм. Для целых чисел — терм i(integer), для беззнаковых целых — терм u(unsigned) и для вещественных — терм r(real).

Значением домена numberList может быть терм:

[u(208), u(3), r(12.45), i(-856)].

Домен t показывает классический пример двоичного дерева binTree, в котором вершины дерева нагружены целыми числами, о чем говорит второй аргумент integer Node. Из каждой вершины может отходить два поддерева: левое и правое, что видно по первому и третьему аргументам LeftTree и RightTree. Если поддерево отсутствует, то вместо него может стоять своеобразная «заглушка», роль которой играет нульварный терм empty.

Значением домена t может быть терм:

binTree(empty, 3, binTree(empty, 25, empty)).

В этом терме корневая вершина со значением 3 имеет только правое поддерево, содержащее также одну вершину со значением 25. Левое поддерево у корневой вершины пустое: empty.

Домен tList представляет собой сложный терм — список деревьев. Значением такого домена может быть список, содержащий, например, приведенное ранее дерево и два пустых дерева:

[binTree(empty, 3, binTree(empty, 11, empty)), empty, empty].

ГЛАВА 5



Константы

5.1. Объявление пользовательских констант

Секция определения констант начинается с ключевого слова `constants`. Определение константы включает ее имя, тип и значение и заканчивается точкой. Константное имя должно начинаться со строчной буквы. Значение константы должно определяться выражением, которое может быть вычислено во время компиляции. Тип константы может быть опущен только для встроенных числовых типов, а также для типов `binary`, `char`, `string`.

Пример объявления констант без явного указания типа:

```
constants
  e = 2.71828182845905.
  my_char = 'a'.
  binaryFileName = "mybin".
  myBinary = #bininclude(binaryFileName).
```

В последней строчке использована директива загрузки `bininclude` двоичных данных из бинарного файла `binaryFileName` и инициализация ими константы `myBinary`.

Тип константы указывается после имени константы через двоеточие.

Пример объявления констант с указанием домена:

```
constants
  true_const : boolean = true.
  чета : пол* = [муж, жен].
domains
  пол = муж; жен.
```

Константа `true_const` объявлена с явным указанием ее типа через двоеточие. Хотя этот тип и встроен в язык, но он не является числовым. Константа `чета` является списком элементов `муж` и `жен`, принадлежащих к типу `пол`.

5.2. Встроенные константы

Visual Prolog имеет ряд встроенных констант:

- compilation_date** — дата компиляции. Здесь YYYY означает четырехзначный год, MM — двузначный месяц, и DD — двузначное число месяца:

```
compilation_date : string = "YYYY-MM-DD".
```

- compilation_time** — время компиляции. Здесь HH означает час, MM — минуты и SS — секунды:

```
compilation_time : string = "HH-MM-SS".
```

- compiler_buildDate** — дата построения компилятора:

```
compiler_buildDate : string = "YYYY-MM-DD HH-MM-SS".
```

- compiler_version** — версия компилятора:

```
compiler_version = 7500.
```

- maxFloatDigits** — определяет максимальное число значащих разрядов вещественных чисел, которое поддерживается компилятором:

```
maxFloatDigits = 16.
```

- null** — специальная константа типа `pointer` с нулевым значением:

```
null : pointer = uncheckedConvert(pointer, 0).
```

- nullHandle** — специальная константа типа `handle` с нулевым значением:

```
nullHandle : handle = uncheckedConvert(handle, 0).
```

- invalidHandle** — специальная константа типа `handle` с неправильным значением (-1):

```
invalidHandle : handle = uncheckedConvert(handle, 0xFFFFFFFF).
```

- platform_bits** — определяет разрядность операционной системы, на которой выполнялась компиляция:

```
platform_bits = 32.
```

или

```
platform_bits = 64.
```

- platform_name** — определяет имя платформы, на которой выполнялась компиляция:

```
platform_name : string = "Windows 32bits".
```

или

```
platform_name : string = "Windows 64bits".
```


часть II



Язык Visual Prolog

- Глава 6.** Предикаты
- Глава 7.** Предикаты в Visual Prolog
- Глава 8.** Модули
- Глава 9.** Отсечение и отрицание
- Глава 10.** Циклы с откатом
- Глава 11.** Рекурсия
- Глава 12.** Ввод/вывод
- Глава 13.** Списки
- Глава 14.** Параметрический полиморфизм
- Глава 15.** Эллипсис
- Глава 16.** Предикаты второго порядка и анонимные предикаты
- Глава 17.** Императивные конструкции
- Глава 18.** Обработка исключительных ситуаций
- Глава 19.** Классы
- Глава 20.** Обобщенное программирование

ГЛАВА 6



Предикаты

Логическое программирование в целом и программирование на языке Visual Prolog в частности базируются на понятиях математической логики. Однако это не означает, что формулы из теории исчисления предикатов первого порядка без изменений будут работать в Прологе. В этой главе раскрывается математическое понятие предиката, рассмотрены операции над предикатами, правила вывода решений с точки зрения логического программирования, а также показан ряд особенностей автоматических вычислений в Прологе и доказательства теорем в математической логике.

6.1. Понятие предиката

Предикат — это n -арная функция $p(t_1, t_2, \dots, t_n)$ с множеством значений {«успех», «неуспех»}, определенная на множестве всех комбинаций своих аргументов $T = T_1 \times T_2 \times \dots \times T_n$. Аргументами предиката являются термы t_1, t_2, \dots, t_n . Таким образом, каждая комбинация значений из декартового произведения T приводит предикат $p(t_1, t_2, \dots, t_n)$ либо к «успеху», либо к «неуспеху».

Если предикат $p(t_1, t_2, \dots, t_n)$ успешен, то переменные, входящие в термы t_1, t_2, \dots, t_n , будут связаны. Если предикат $p(t_1, t_2, \dots, t_n)$ неуспешен, то связывание переменных, входящих в термы t_1, t_2, \dots, t_n , не произойдет.

Простым примером может служить предикат `книга(x, y, z)` для отношения «книга с названием x написана автором y и издана в год z ». Предикат можно понимать как некое утверждение о предмете суждения, которое может быть как успешным (истинным), так и неуспешным (ложным).

Иногда множество значений {«успех», «неуспех»} преподносят как {«истина», «ложь»}, {«true», «false»} или {1, 0}. Это верно, однако здесь надо иметь в виду то, что в Прологе нельзя присвоить переменной X значение истинности предиката напрямую — например: $X = p(t_1, t_2, \dots, t_n)$, и совершать над ней какие-либо операции булевой алгебры. Хотя предикат и является функцией, но значение этой функции читается и интерпретируется машиной логического вывода Пролога, а точнее — *механизмом поиска с возвратом* (англ. backtracking). Значения же `true` и

`false` принимают переменные типа `boolean`. Поэтому, чтобы не путать множество значений предикатов с множеством значений булевых переменных, будем использовать {«успех», «неуспех»} в качестве множества значений истинности предикатов.

ЗАМЕТКА

Справедливости ради здесь стоит сделать оговорку — Visual Prolog имеет встроенную функцию `toBoolean/1`, которая возвращает значение `true` или `false` из области `boolean`, в зависимости от успеха/неуспеха того предиката, вызов которого указан в качестве аргумента этой функции.

Синтаксис предиката $p(t_1, t_2, \dots, t_n)$ такой же, как и синтаксис термов. Отличие между ними заключается только в том, что предикаты имеют *процедурный смысл*, т. е. они определяют последовательность действий, которые приводят либо к «успеху», либо к «неуспеху». Термы же не имеют процедурного смысла, т. к. являются способом представления данных и сами не выполняют никаких действий. Определение правил вычисления предиката мы увидим позже — при рассмотрении импликации.

Декларативный смысл предиката $p(t_1, t_2, \dots, t_n)$ — это отношение p между аргументами t_1, t_2, \dots, t_n . Это отношение может выражать также и некоторое свойство аргументов, действие над аргументами и т. п. К примеру, предикат `вес(Лада, Калина, 800)` выражает свойство вес автомобиля марки Лада модели Калина.

Предикат степень (n, x, y) является истинным, если $y = x^n$.

Предикат называют *тождественно-истинным*, если на любом наборе аргументов он принимает значение «успех». В Visual Prolog введен такой предикат — он нуль-арный и обозначается: `succeed()`. Его реализация очень проста — например, неравенство $1 > 0$ является тождественно-истинным предикатом. Вас не должна смущать инфиксная нотация предиката $1 > 0$, т. к. предикаты условий в Прологе записываются в инфиксной нотации.

Предикат называют *тождественно-ложным*, если на любом наборе аргументов он принимает значение «неуспех». В Visual Prolog введен такой предикат — он нуль-арный и обозначается `fail()`. Его реализация также проста — например: $1 > 1$.

Предикат называют *выполнимым*, если хотя бы на одной комбинации аргументов он принимает значение «успех». Как правило, Пролог-программа содержит определение только выполнимых предикатов.

Заметим, что Visual Prolog допускает запись имен предикатов, термов и переменных в национальном алфавите.

6.2. Логические операции над предикатами

В языке Пролог к предикатам применимы операции отрицания, конъюнкции и дизъюнкции.

Отрицанием предиката `p(X)` называется новый предикат `not(p(X))`, который принимает значение «успех», если вычисление `p(X)` неуспешно, и принимает значение «неуспех», если вычисление `p(X)` успешно.

Предикат в операции `not` не может связывать переменные значениями. Например, если отрицание `not(p(X))` успешно, значит `p(X)` завершился неудачей. А предикат, завершившийся неудачей, не связывает переменные значениями. И наоборот, если предикат `p(X)` завершается удачей и связывает переменную `X` значением, то отрицание `not(p(X))` будет неуспешно, что также ведет к «развязыванию» переменной.

Не следует путать операцию отрицания предиката с логической операцией отрицания переменной булевого типа. Отрицание предиката всего лишь меняет значение предиката в области {«успех», «неуспех»}. Логическое отрицание меняет значение булевой переменной на множестве {«истина», «ложь»} и поэтому позволяет вычислить новое значение переменной.

Конъюнкция двух предикатов `p(X) and q(X)` успешна, если успешны оба предиката, причем Visual Prolog выполняет вначале предикат `p(X)`, и если он успешен, то выполняет предикат `q(X)`. Если предикат `p(X)` неуспешен, то предикат `q(X)` не выполняется. Конъюнкция имеет другое название — «ЛОГИЧЕСКОЕ И».

В Прологе коммутативность относительно конъюнкции предикатов в общем случае изменяет не только процедурный, но и декларативный смысл программы. Например, конъюнкция предикатов `читать(X) and простое(X)` означает чтение переменной `X` и последующую проверку простоты числа `X`. Перемена мест умножаемых предикатов ведет к искажению смысла, т. к. конъюнкция предикатов `простое(X) and читать(X)` будет означать проверку простоты непрочитанного к этому моменту числа `X`. Таким образом, если в некоторой конъюнкции предикатов `p(X) and q(X)` от первого предиката во второй передаются переменные, то, естественно, менять порядок вызова предикатов нельзя.

Visual Prolog разрешает использовать вместо имени операции `and` запятую. Такой «синтаксический сахар» улучшает читабельность программ. Например, конъюнкцию четырех предикатов `p(X) and q(X) and r(X) and s(X)` можно записать в виде `p(X), q(X), r(X), s(X)`.

Дизъюнкция двух предикатов `p(X) or q(X)` успешна, если успешен хотя бы один предикат. Дизъюнкция неуспешна, если все предикаты неуспешны. Visual Prolog выполняет вначале предикат `p(X)`, и если он неуспешен, то выполняет предикат `q(X)`. Дизъюнкция имеет другое название — «ЛОГИЧЕСКОЕ ИЛИ».

В Прологе коммутативность относительно дизъюнкции предикатов в общем случае изменяет процедурный смысл программы, что приводит к изменению вычислительной сложности программы. Например, дизъюнкция предикатов `X>Y or простое(X)` может выполниться быстрее дизъюнкции `простое(X) or X>Y`. Первая дизъюнкция имеет процедурный смысл: «проверить истинность условия `X>Y`, если условие ложно, то проверить простоту числа `X`». Вторая же дизъюнкция имеет другой процедурный смысл: «проверить простоту числа `X`, если число не простое, то проверить истинность условия `X>Y`», что ведет к увеличению времени выполнения, т. к. определение простоты числа достаточно длительная операция по сравнению с проверкой логического условия. При этом декларативный смысл обеих дизъюнкций одинаков: «`X` больше `Y` или `X` — простое».

Visual Prolog разрешает использовать вместо имени операции `or` точку с запятой. Такой «синтаксический сахар» улучшает читабельность программ. Например, дизъюнкцию четырех предикатов `p(X) or q(X) or r(X) or s(X)` можно записать в виде `p(X);q(X);r(X);s(X)`.

Заметим, что в Visual Prolog отсутствует встроенная операция «исключающее или» относительно предикатов, но при необходимости она легко реализуется другими средствами языка — отсечениями решений.

6.3. Логические формулы

Логические формулы строятся с использованием операций отрицания, дизъюнкции и конъюнкции над предикатами. В Visual Prolog допускается применение скобок. Приоритет операции отрицания считается наивысшим, т. к. отрицание использует скобки. Приоритет конъюнкции выше приоритета дизъюнкции. Примеры формул над предикатами представлены в табл. 6.1.

Таблица 6.1. Формулы над предикатами

№	Формула	Порядок вычисления
1	<code>p(X),s(X)</code>	<code>p(X) and s(X)</code>
2	<code>p(X);s(X)</code>	<code>p(X) or s(X)</code>
3	<code>p(X),q(X);s(X),t(X)</code>	<code>(p(X) and q(X)) or (s(X) and t(X))</code>
4	<code>(p(X);q(X)),(s(X);t(X))</code>	<code>(p(X) or q(X)) and (s(X) or t(X))</code>
5	<code>q(X);s(X),t(X)</code>	<code>q(X) or (s(X) and t(X))</code>
6	<code>q(X);not(s(X)),t(X)</code>	<code>q(X) or (not(s(X)) and t(X))</code>
7	<code>not(p(X),s(X))</code>	Синтаксическая ошибка
8	<code>not((p(X),s(X)))</code>	<code>not((p(X) and s(X)))</code>
9	<code>p(X) = s(X)</code>	Синтаксическая ошибка

Обратите внимание на пример в строке 7 табл. 6.1. Аргументом операции отрицания в Visual Prolog может быть только один предикат. Однако если конъюнкцию предикатов `p(X),s(X)` взять в скобки, то мы получим выражение над предикатами, которое уже может служить аргументом оператора `not`. Указанное действие выполнено в строке 8 рассматриваемой таблицы.

Упрощение формул приводит к сокращению вычислений и, как следствие, к оптимизации программ. Например, в формуле:

`p(X),q(X); p(X),t(X)`

предикат `p(X)` можно вынести влево за скобки:

`p(X), (q(X); t(X)) .`

Если предикат `q(X)` в первой формуле закончится неуспехом в ветке `p(X),q(X)`, то Visual Prolog будет повторно вычислять предикат `p(X)` в ветке `p(X),t(X)`. Во второй

формуле предикат $p(x)$ вычисляется один раз, после чего вычисляется дизъюнкция $(q(x) ; t(x))$. Если $q(x)$ заканчивается успехом, то исходная формула считается доказанной. Если $q(x)$ заканчивается неуспехом, то вычисляется $t(x)$.

В формуле:

$q(x), p(x) ; t(x), p(x)$

предикат $p(x)$ можно вынести вправо за скобки:

$(q(x) ; t(x)), p(x)$.

6.4. Правила вывода предикатов

В математической логике правила предназначены для формального вывода одних формул из других. В Прологе же правила предназначены для вывода из формулы *только одного* предиката. Здесь надо заметить, что в математической логике правило вывода одного предиката из формулы имеет специальное название — *дизъюнкт Хорна*. Дизъюнкты Хорна названы по имени логика Альфреда Хорна, впервые описавшего их в 1951 году. Дизъюнкты Хорна используются в логическом программировании, т. к. позволяют существенно повысить эффективность вывода цели. Термин «вывод» в данном контексте можно заменить эквивалентным по смыслу термином «доказательство» или «вычисление».

6.4.1. Вывод предиката из другого предиката

Рассмотрим вначале случай, когда роль исходной формулы для вывода одного предиката играет другой предикат. На языке математической логики правило вывода предиката $p(x)$ определяется с помощью операции импликации $r(x) \rightarrow p(x)$. Суть *импликации* — это логическое следование $p(x)$ из $r(x)$, или, другими словами, импликацию можно выразить правилом «если $r(x)$, то $p(x)$ ». Предикат $r(x)$ играет здесь роль причины, а предикат $p(x)$ — следствия.

Правило $r(x) \rightarrow p(x)$ в математической логике можно представить в форме дизъюнкции $\overline{r(x)} \vee p(x)$ или в эквивалентной записи на Прологе `not(r(X)) or p(X)`. Почему именно такая дизъюнкция выражает импликацию? Объяснение очень простое — импликация имеет два взаимоисключающих правила вывода результата:

- если причина $r(x)$ успешна, тогда и следствие $p(x)$ успешно;
- если причина $r(x)$ неуспешна, то о следствии ничего заключить нельзя.

Первый вариант записывается конъюнкцией $r(x) \wedge p(x)$, второй вариант выражается отрицанием $\overline{r(x)}$. Результат будет получен либо по первому правилу, либо по второму. Поэтому оба правила составляют дизъюнкцию, которую можно упростить по правилу Блейка-Порецкого:

$$\overline{\overline{r(x)}} \vee r(x) \wedge p(x) = \overline{r(x)} \vee p(x)$$

Дизъюнкция $r(x) \vee p(x)$ всегда имеет только один положительный предикат, т. е. предикат без отрицания, — тот самый предикат, доказательство которого и описывает правило. Такой дизъюнкт в математической логике называют дизъюнктом Хорна или предложением Хорна.

В математической логике стрелка импликации может быть направлена в любую сторону — главное, чтобы она указывала на следствие. В языке Visual Prolog следствие, т. е. предикат $p(x)$, всегда располагается слева, а причина, т. е. предикат $r(x)$, — справа. На клавиатуре компьютера отсутствует кнопка для символа стрелки, и вместо стрелки в большинстве Прологов предусмотрено ключевое слово `if`. Поэтому правило математической логики $r(x) \rightarrow p(x)$ в нетипизированном Прологе записывается так:

```
p(X) if r(X).
```

Эту формулу следует читать и понимать так: «Предикат $p(x)$ будет успешен, если успешен предикат $r(X)$ ». В данном контексте «успешен» означает «выведен», «доказан» или «вычислен».

В качестве синтаксического сахара в Прологе вместо ключевого слова `if` можно применить составной символ `:-`, которым, как правило, пользуется большинство программистов на Прологе:

```
p(X) :- r(X).
```

Однако в Visual Prolog необходимо использовать только составной символ `:-`.

Выводимый предикат $p(X)$ называется *головой правила*, а предикат $r(X)$ — *телом правила*. Обратите внимание, что правила в Прологе должны заканчиваться точкой.

Почему для импликации не используется составной символ `<:-`, история умалчивает. Почему для импликации выбран составной символ `:-?` Быть может, потому что дизъюнкт Хорна $p(x) \vee \overline{r(x)}$ содержит знаки дизъюнкции и отрицания, аналог которых на клавиатуре отсутствуют. По этой причине программисты используют подходящие заменители: для дизъюнкции — двоеточие, а для отрицания — минус. Поэтому математическая формула $p(x) \vee \overline{r(x)}$ записывается в Прологе в виде $p(X) :- r(X)$. Следует иметь в виду, что в Прологе имена переменных пишутся только с прописной буквы, в отличие от математической логики, где переменные пишутся со строчной буквы.

Особенностью Пролога является то, что областью видимости переменной является одно правило. Глобальных переменных в Прологе нет. Однако при необходимости их роль играют факты и факты-переменные, которые будут рассмотрены в разд. 6.5 и в главе 7.

Процедурный смысл выполнения Прологом правила $p(X) :- r(X)$ прост. Если предикат $r(X)$ успешен, то считается, что предикат $p(X)$ тоже успешен. Если же предикат $r(X)$ неуспешен, то считается, что предикат $p(X)$ также неуспешен.

Приведенное правило отличается от правила вычисления импликации, принятого в математической логике, где при ложной посылке следствие может быть как лож-

ным, так и истинным, — т. е. истинность следствия при ложной посылке неизвестна. Причина этого отличия кроется в том, что мир логического вывода Пролога замкнут, а мир логического вывода в математической логике открыт. Открытый мир вывода — это мир, правила которого всегда можно изменить или дополнить с целью адекватного описания задачи. Замкнутость вывода в Прологе означает следующее: все, что описано в программе, считается известными сведениями об описываемом мире, известными — значит истинными. Если же каких-либо сведений о задаче в программе нет, то они считаются ложными, независимо от их реальной истинности в описываемом мире.

Предположение о замкнутости мира существенно упрощает логический вывод в Прологе, замещая неоднозначность типа {«успех», «неуспех», «неизвестно»} дуализмом {«успех», «неуспех»}, в котором «неизвестно» рассматривается как неуспех.

Рассмотрим различия замкнутого и открытого миров логического вывода на следующем примере. Пусть есть правило: «если дождь, то урожай». И известно, что «дождь был». В замкнутом мире вывода будет сделано заключение, что урожай будет. В открытом мире вывода заключение такое же — урожай будет, поскольку в обоих мирах из истинной посылки следует истинное заключение. Давайте теперь посмотрим на заключения в случае, когда «дождя не было». В замкнутом мире будет сделано заключение, что урожая не будет. В открытом мире никаких заключений о будущем урожае сделать нельзя. Действительно, ведь в открытом мире вывода можно предположить, что при отсутствии дождя был организован искусственный полив, — тогда урожай будет. Но в закрытом мире вывода Пролога об искусственном поливе никаких явно указанных сведений нет, следовательно, искусственного полива нет.

Рассмотрим пример правила для определения четных чисел:

четное(X) :- X mod 2 = 0.

Здесь предикат $X \bmod 2 = 0$ записан в инфиксной форме, но это вас смущать не должно. В Visual Prolog все математические действия записываются подобно тому, как это делается в математике, но от этого они не перестают быть предикатами. В данном примере предикат равенства нулю остатка от деления числа X на 2 может быть как успешен, так и неуспешен, — в зависимости от значения X .

6.4.2. Формулы вычисления предикатов

В общем случае предикат описывается несколькими правилами. Правила выполняются в том порядке, в котором они записаны в программе, т. е. сверху вниз. Порядок выполнения каждого правила следующий. Сначала аргументы головы унифицируются с аргументами вызова этого предиката. Если унификация успешна, то выполняется тело правила, если оно есть. Если тело правила успешно, то правило считается успешным, следовательно, и предикат считается успешным. В других случаях предикат неуспешен.

В качестве тела правила для вычисления предиката может быть использована формула, содержащая рассмотренные ранее операции отрицания, дизъюнкции и конъ-

юнкции. С помощью логических формул в Прологе описываются все алгоритмические конструкции: последовательные вычисления, ветвления и циклы. *Линейный участок* алгоритма организуется конъюнкцией предикатов, при этом предикаты выполняются последовательно друг за другом. Роль конъюнкта может играть любая правильно записанная формула. *Ветвление* алгоритма организуется дизъюнкцией. Роль дизъюнкта также может играть правильно записанная формула. Количество ответвлений произвольно и равно количеству дизъюнктов. *Циклы* бывают двух типов: циклы с откатом и рекурсивные циклы. *Циклы с откатом* организуются неявно механизмом поиска с откатом, встроенным в Пролог. Поэтому циклы с откатом не имеют специальной синтаксической конструкции в тексте программы. *Рекурсивные циклы* организуются с помощью рекурсии. Рекурсивные циклы будут рассмотрены в главе 11.

Если тело предиката $p(X)$ представлено дизъюнкцией:

```
p(X) :- r(X); s(X); t(X).
```

то каждый дизъюнкт образует правило. Поэтому предикат $p(X)$ может быть записан тремя правилами:

```
p(X) :- r(X).
p(X) :- s(X).
p(X) :- t(X).
```

Такие правила еще называют *предложениями Хорна* или просто *предложениями*. Предложения Хорна с одинаковой головой определяют один предикат. Все предложения одного предиката должны быть сгруппированы.

Рассмотрим пример пятиарного предиката корень/5, выполняющего цепочку последовательных вычислений для нахождения корней квадратного уравнения, имеющего коэффициенты A, B, C:

```
корень(A,B,C,X1,X2) :- V = B*B-4*A*C, % дискриминант
    V >= 0,
    D = sqrt(V), % квадратный корень
    X1 = (-B+D)/2,
    X2 = (-B-D)/2.
```

Здесь тело правила представлено одним дизъюнктом, содержащим конъюнкцию пяти предикатов. Предикат корень/5 будет вычислен и вернет корни X1, X2, если выполнится предикат $V \geq 0$. Однако при $V < 0$ предикат корень/5 будет неуспешен.

Так как при равенстве нулю дискриминанта квадратное уравнение имеет один корень, то с помощью ветвления можно выделить этот случай:

```
корень(A,B,C,X1,X2) :- V = B*B-4*A*C, % дискриминант
    (V > 0,
    D = sqrt(V), % квадратный корень
    X1 = (-B+D)/2,
    X2 = (-B-D)/2; % конец первой ветки
    V = 0, % начало второй ветки
    X1 = -B/2,
    X1 = X2).
```

Здесь тело правила содержит два дизъюнкта, у которых общий член — предикат вычисления дискриминанта — вынесен за скобки. Поэтому дискриминант вычисляется один раз перед ветвлением.

Рассмотренный ранее предикат четное(X) может иметь частное определение — с помощью перечисления конкретных значений X в заданном диапазоне, например: $1 < X < 9$:

```
четное(X) :- X=2; X=4; X=6; X=8.
```

Здесь предикат четное(X) описан одним правилом, тело которого содержит четыре дизъюнкта. Каждый дизъюнкт можно описать отдельным правилом, в результате чего получится четыре правила для предиката четное(X):

```
четное(X) :- X=2.  
четное(X) :- X=4.  
четное(X) :- X=6.  
четное(X) :- X=8.
```

Точка в конце каждого правила играет роль операции дизъюнкции над правилами предиката четное(X). Упростим правила, подставив вместо переменной X ее значение:

```
четное(2).  
четное(4).  
четное(6).  
четное(8).
```

Вот так в Прологе и перечисляются конкретные значения X в заданном диапазоне, например: $1 < X < 9$. Таким способом можно определять данные, значения которых априорно известны, — например, дни недели:

```
день_недели("понедельник").  
день_недели("вторник").  
день_недели("среда").  
день_недели("четверг").  
день_недели("пятница").  
день_недели("суббота").  
день_недели("воскресенье").
```

Другой пример — определение принадлежности переменной X заданному диапазону, например: $0 < X < 12$:

```
диапазон_0_12(X) :- 0 <= X, X <= 12.
```

Тело этого правила содержит конъюнкцию предикатов. Если предикаты $0 <= X$ и $X <= 12$ успешны, то предикат диапазон_0_12(X) также будет успешен. В данном примере границы диапазона указаны в имени предиката. Лучше будет левую и правую границы диапазона указать в качестве аргументов, тогда наш предикат может определять принадлежность числа произвольному диапазону:

```
диапазон(X, Лев, Прав) :- Лев <= X, X <= Прав.
```

Рассмотрим более сложный пример, когда надо определить, что некое число x является четным и входит в заданный диапазон от Лев до Прав включительно. Назовем такой предикат, например, тест(X , Лев, Прав). Тело правила, очевидно, будет содержать проверку принадлежности диапазону и проверку четности:

```
тест(X, Лев, Прав) :- диапазон(X, Лев, Прав), X mod 2 = 0.
```

Обратите внимание, что проверка диапазона осуществляется с помощью ранее описанного предиката. Однако можно сделать и по-другому — проверить принадлежность диапазона непосредственно в теле правила:

```
тест(X, Лев, Прав) :- Лев<=X, X<=Прав, X mod 2 = 0.
```

В заключительном примере о предикатах опишем правила вывода решения для задачи: «определить предикат, который будет успешным в случае, если число x — положительное и четное, или если x — отрицательное и нечетное». Признак положительности и четности записывается в виде конъюнкции:

```
X>0, X mod 2 = 0.
```

Признак отрицательности и нечетности записывается так:

```
X<0, X mod 2 = 1.
```

Заметим, что ноль не является положительным или отрицательным числом. Теперь можно записать правило для нашего предиката, которое назовем тест1:

```
тест1(X) :- X>0, X mod 2 = 0; X<0, X mod 2 = 1.
```

Как можно видеть, тело правила содержит дизъюнкцию двух конъюнкций. Это правило можно эквивалентно представить в виде двух правил, каждое для «своего» дизъюнкта:

```
тест1(X) :- X>0, X mod 2 = 0.
```

```
тест1(X) :- X<0, X mod 2 = 1.
```

6.5. Факты

Факт в Visual Prolog — это предикат, каждое предложение которого является основным термом. Каждый факт записывается в виде головы предложения и не содержит тела. Примерами фактов могут быть, например, дни недели, описанные ранее.

Рассмотрим другой пример — о зависимости урожая от дождя и полива. Программу на Прологе можно записать так:

```
дождь .
```

```
урожай :- дождь; полив .
```

В этом примере предикаты дождь() и полив() нульевые и играют роль высказываний, связанных операций *or*, т. к. для урожая достаточно, чтобы произошло хотя бы что-то одно: или дождь, или полив. Скобки нульевых предикатов можно не указывать. Итак, согласно программе, Прологу известно правило «предсказания» урожая и тот факт, что был дождь. Результат доказательства предиката урожай() будет

положительным. Давайте расширим правило, добавив проверку отсутствия урагана:

дождь .

урожай:- not (ураган) , (дождь ; полив) .

Результат вновь будет положительным. Факт ураган в программе отсутствует, что в замкнутом мире вывода Пролога говорит о том, что урагана нет. Следовательно, предикат not (ураган) будет успешным. Обратите внимание, что отсутствие урагана проверяется и для дождя, и для полива, т. к. дизъюнкция дождя и полива заключена в скобки. Декларативный смысл программы таков: «урожай будет, если нет урагана и есть дождь или полив». Если скобки в этом примере убрать:

дождь .

урожай:- not (ураган) , дождь ; полив .

то отсутствие урагана будет проверяться только для дождя, что может привести к неправильному «предсказанию» урожая. Например, пусть был и дождь, и полив, и ураган:

полив .

дождь .

ураган .

урожай:- not (ураган) , дождь ; полив .

Доказательство предиката not (ураган) будет неуспешным, и Пролог сразу перейдет к доказательству второго дизъюнкта — предиката полив . Факт полив в программе есть, значит, доказательство полива успешно и, следовательно, доказательство урожая также успешно. Причина такого неправильного ответа кроется в отсутствии скобок. Если скобки вернуть на место, то ответ будет правильный — урожая не будет:

полив .

дождь .

ураган .

урожай:- not (ураган) , (дождь ; полив) .

6.6. Машина логического вывода

Цель (или целевое утверждение) логической программы — это предикат, который надо доказать или опровергнуть на основе фактов и правил с помощью машины логического вывода.

Единственным механизмом машины логического вывода Пролога является поиск с возвратом (англ. backtracking). Поиск с возвратом — метод нахождения решений целевого утверждения в логической программе. Этот метод позволяет найти все решения, или некоторую часть из них, или даже одно решение. Понятие backtrack было введено в 1950 году американским математиком Дерриком Генри Лемером. Однако сама процедура поиска с возвратом практически одновременно и независимо была изобретена многими исследователями еще до его формального описания Лемером. Справедливости ради надо заметить, что поиск с возвратом используется

в некоторых других языках программирования и не является «собственностью» Пролога. Самый первый язык, в котором был использован механизм поиска с возвратом, — это SNOBOL, предназначенный для гибкой обработки строк.

Доказательство цели методом поиска с возвратом сводится к последовательному доказательству предикатов, согласно тому порядку, который описывает формула, являющаяся телом целевого утверждения. Если на очередном шаге доказательство какого-либо предиката провести не удается, то осуществляется возврат (откат) к предыдущему предикату с целью поиска альтернативного решения. Правила или факты одного предиката доказываются в порядке их следования в тексте программы: сверху вниз и слева направо. Заметим, что порядок правил существенно влияет на эффективность поиска решения. Поэтому сверху надо располагать самые перспективные правила. Более того, проверки на ранних стадиях поиска позволяют выявить заведомо неподходящие варианты и исключить из перебора целевые поддеревья. Зачастую это также значительно уменьшает время нахождения решения. По своей сути поиск с возвратом реализует стратегию поиска на дереве сначала вглубь.

Для того чтобы правильно совершать откаты, машина логического вывода при вызове предиката, определенного более чем одним правилом, запоминает адрес второго правила в стеке адресов возврата, после чего передает управление первому правилу. Вместе с адресом второго правила в стек помещаются аргументы цели. Адрес и аргументы составляют один стековый фрейм (содержимое стекового фрейма описано в главе 30). Когда необходимо совершить откат назад, из стека извлекается самый верхний фрейм, содержащий адрес правила, и совершается переход к этому правилу. Один шаг отката может «перепрыгнуть» назад сразу несколько предикатов, если их адреса не записаны в стек. Одним из «побочных эффектов» отката является освобождение переменных, связанных после размещения в стеке последнего фрейма. Более того, откат — это *единственный* способ в Прологе освободить переменную от ее значения. После него переменная становится свободной и может быть связана другим значением. Процесс освобождения переменной связан не с тем, что у переменной «отбирают» ее значение, а с тем, что при извлечении фрейма из стека аргументы во фрейме имеют то состояние, которое было до выполнения предикатов, через которые мы «перепрыгнули» во время отката.

При откате на второе правило предиката в стек помещается фрейм, содержащий адрес третьего правила, если таковое существует. Если выполняется последнее правило предиката, то в стек ничего не помещается. В стек также ничего не помещается, когда предикат описан единственным правилом.

6.7. Работа механизма поиска с возвратом

Рассмотрим элементарную задачу — поиск элемента, большего 5, во множестве $Q = \{2, 3, 7, 4\}$. Элементы множества перечислим с помощью предиката `q/1`. Предикат `больше/1` будет искать нужное число во множестве.

```
q(2). q(3). q(7). q(4).
больше(X) :- q(X), X>5.
```

Обратите внимание на то, что правила могут быть записаны в одну строчку. Это особенно удобно при записи правил без тела, какими являются правила предиката `q/1`. Такие правила в Visual Prolog называются *фактами*. Предикат `больше/1` содержит один выходной аргумент — переменную `x`. Пусть целевым запросом к нашей программе является формула:

`больше(X), write(X).`

Предикат `write/1` выведет ответ на экран. В теле предиката `больше(X)` осуществляется вызов предиката `q(X)`. Сопоставление (унификация) цели `q(X)` с фактами `q/1` выполняется по очереди, начиная с первого. Рассмотрим подробно, как это происходит. При унификации цели `q(X)` и факта `q(2)` сначала (рис. 6.1, *a*) в стек задавливается адрес факта `q(3)`, после чего переменная `X` связывается с числом 2. Однако предикат `2>5` неуспешен. Это вызывает откат назад, который происходит посредством выдавливания из стека (рис. 6.1, *b*) последнего помещенного туда адреса — адреса факта `q(3)`, и передачи управления ему. Переменная `X` становится свободной, теряя значение 2.

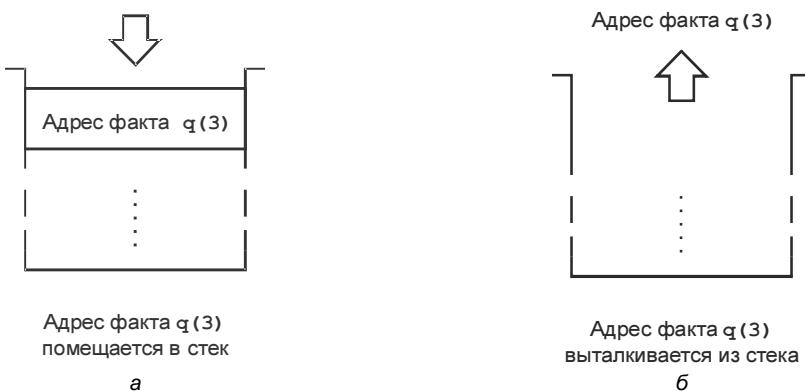


Рис. 6.1. Содержимое стека адресов возврата первой задачи: ситуации *a* и *b*

Повторное доказательство цели `q(X)` приводит к унификации `q(X)` с фактом `q(3)`. Сначала (рис. 6.1, *c*) в стек задавливается адрес третьего факта `q(7)`, после чего переменная `X` связывается с числом 3. Однако предикат `3>5` вновь будет неуспешен. Откат назад выдавливает из стека (рис. 6.1, *d*) адрес факта `q(7)`, задавливает в стек (рис. 6.1, *e*) адрес факта `q(4)` и связывает переменную `X` с числом 7. Предикат `7>5` будет успешен, следовательно, предикат `больше(7)` также будет успешным. Целевой запрос завершится успешно выполнением предиката `write(7)`. Однако следует иметь в виду, что в стеке остался адрес факта `q(4)`.

Рассмотрим другую задачу. Пусть даны два множества чисел: $Q = \{2, 3, 7\}$ и $R = \{5, 8\}$. Необходимо найти такую пару чисел (q, r) из этих множеств, сумма которых больше 10. Далее представлена программа, решающая эту задачу:

`q(2). q(3). q(7).`

`r(5). r(8).`

`пара(X,Y) :- q(X), r(Y), X+Y>10.`

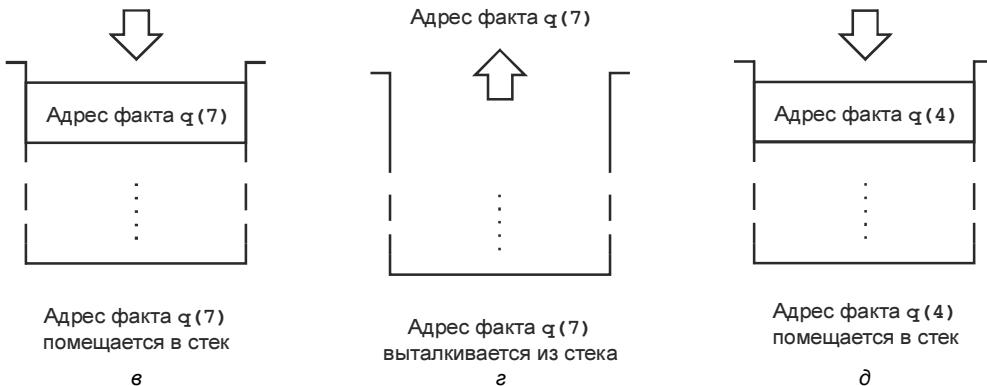


Рис. 6.1. Содержимое стека адресов возврата первой задачи: ситуации *v*, *g* и *d*

Правила каждого предиката должны быть сгруппированы. Это означает, что нельзя смешивать факты *q/1* и *r/1* между собой. Такое смешивание правил является ошибкой.

Пусть предикат пара (*X, Y*) является целью и содержит свободные переменные *X* и *Y*. Выполнение этой цели по шагам представлено далее.

1. Первым выполняется предикат *q(X)*. Он унифицирует предикат *q(X)* и самый первый факт *q(2)*. В стек помещается адрес факта *q(3)*, переменная *X* связывается значением 2. К этому моменту стек имеет состояние, показанное на рис. 6.2, *a*.
2. Далее предикат *r(Y)* унифицируется с первым фактом *r(5)*. В стек помещается адрес факта *r(8)*, переменная *Y* связывается значением 5. Стек переходит в состояние, показанное на рис. 6.2, *b*. Проверяется условие $2+5>10$. Это условие неуспешно, или, еще говорят, ложно.
3. Поэтому механизм поиска с возвратом делает откат назад, выталкивая из стека адрес факта *r(8)*, и получает следующее значение *Y=8*. Стек имеет состояние, показанное на рис. 6.2, *v*. При этом прежнее значение *Y=5* теряется. Проверяется условие $2+8>10$. Это условие неуспешно.
4. Механизм поиска выталкивает из стека адрес факта *q(3)*, помещает в стек адрес факта *q(7)* и связывает переменную *X* с числом 3. Стек имеет состояние, показанное на рис. 6.2, *g*.
5. Предикат *r(Y)* унифицируется с первым фактом *r(5)*. В стек помещается адрес факта *r(8)*, переменная *Y* связывается значением 5. Стек переходит в состояние, показанное на рис. 6.2, *d*. Проверяется условие $3+5>10$. Это условие неуспешно.
6. Механизм поиска с возвратом делает откат назад, выталкивая из стека адрес факта *r(8)*, и получает следующее значение *Y=8*. Стек имеет состояние, показанное на рис. 6.2, *e*. Проверяется условие $3+8>10$. Это условие истинно.

На этом доказательство предиката пара (*X, Y*) завершается успешно со значениями пары (3, 8).

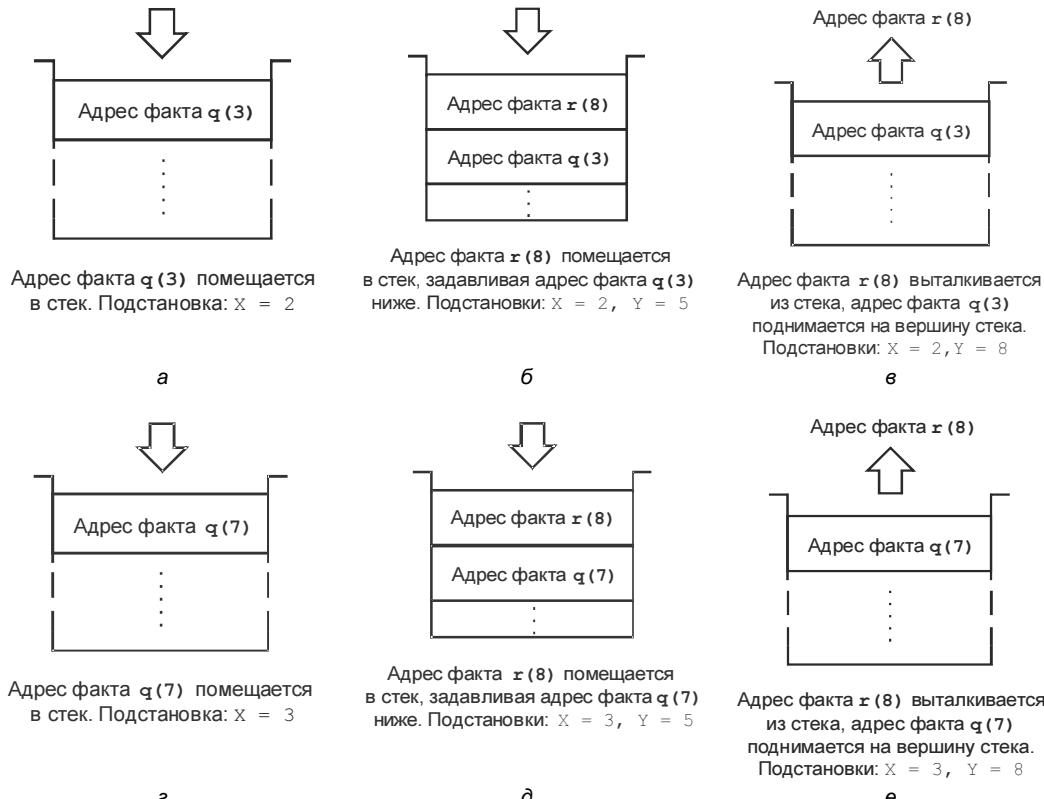


Рис. 6.2. Содержимое стека адресов возврата второй задачи

На рис. 6.3 изображено дерево поиска решения для нашей задачи. Предикаты отмечены круглыми или овальными вершинами. Возвращаемые ими значения находятся в прямоугольниках. Стрелками отмечен пройденный механизмом поиска путь на дереве. Пунктирными линиями показано то поддерево, которое не было покрыто поиском, т. к. после нахождения первой подходящей пары чисел (3,8) поиск завершается.

Вот так происходит поиск с откатом на дереве до нахождения первого успешного решения $X=3$ и $Y=8$. В результате предикат пара (X, Y) заканчивается успешно со значениями аргументов пары $(3, 8)$.

Следует добавить, что дерево поиска решения в Прологе называется И-ИЛИ деревом. Корень дерева расположен сверху, листья внизу. С помощью операции ЛОГИЧЕСКОЕ-И (конъюнкции) дерево растет от корня до листьев. С помощью операции ЛОГИЧЕСКОЕ-ИЛИ (дизъюнкции) дерево разветвляется, из-за чего и образуется множество путей от корня до листьев.

Проводя аналогию с императивными языками программирования, можно указать, что в рассмотренной программе реализован цикл в цикле. Внешний цикл — это перебор значений X из множества Q . Внутренний цикл — это перебор значений Y из множества R .

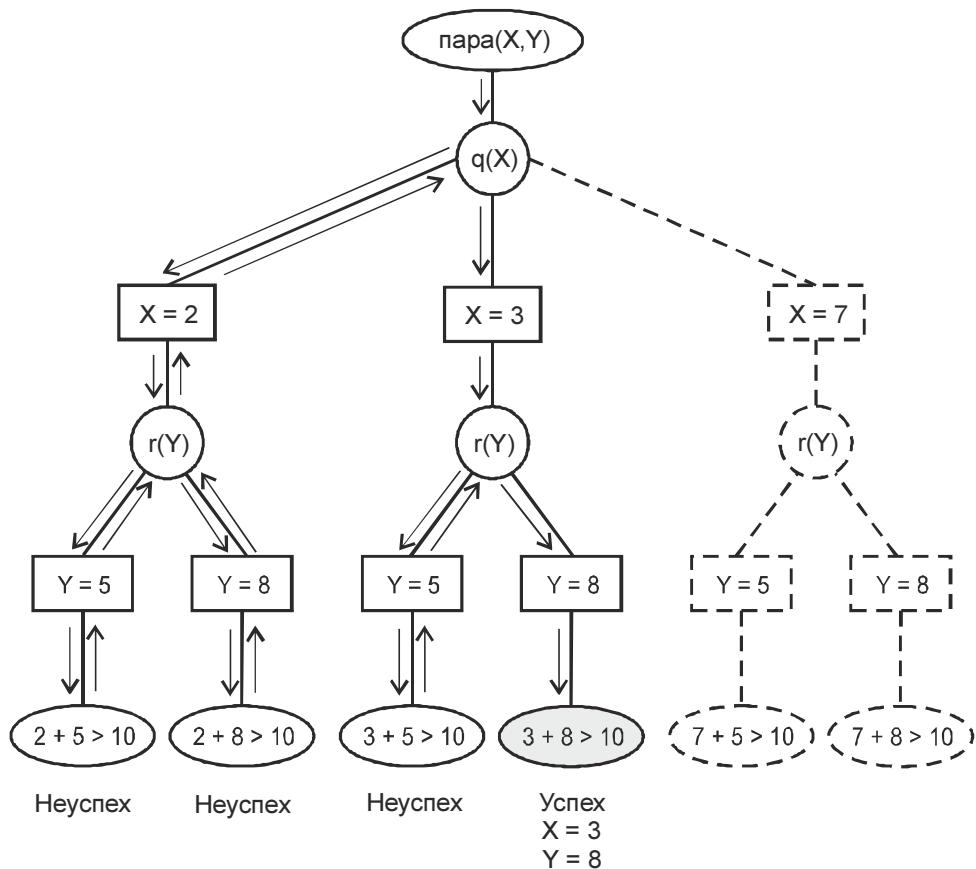


Рис. 6.3. Дерево поиска первого решения

Для уменьшения времени поиска решения в этой задаче следует располагать факты двух множеств в порядке от наибольшего значения к наименьшему:

```

q(7). q(3). q(2).
r(8). r(5).
пара(X,Y) :- q(X), r(Y), X+Y>10.
  
```

В этом случае Пролог найдет решение сразу. Оно будет «висеть» на первой ветке дерева: $X=7$, $Y=8$. На рис. 6.4 изображено дерево поиска, в котором стрелками отмечен путь, ведущий к первому решению. Пунктирными линиями показаны поддеревья, не покрытые поиском с возвратом.

Мы с вами рассмотрели поиск первого удачного решения. А как найти все возможные решения в этой задаче? Очевидно, после нахождения первого решения надо каким-то образом возобновить перебор дальше. Для этого в Visual Prolog есть специальный предикат, который всегда неуспешен. Имя этого предиката `fail`.

Давайте вставим этот предикат в нашу программу сразу после проверки условия $X+Y>10$. А также для фиксации найденных решений выведем их на экран. Предикат `nl` (new line) осуществляет переход на новую строку:

```
q(7). q(3). q(2).
r(8). r(5).
пара(X,Y) :- q(X), r(Y), X+Y>10, write(X,Y), nl, fail.
```

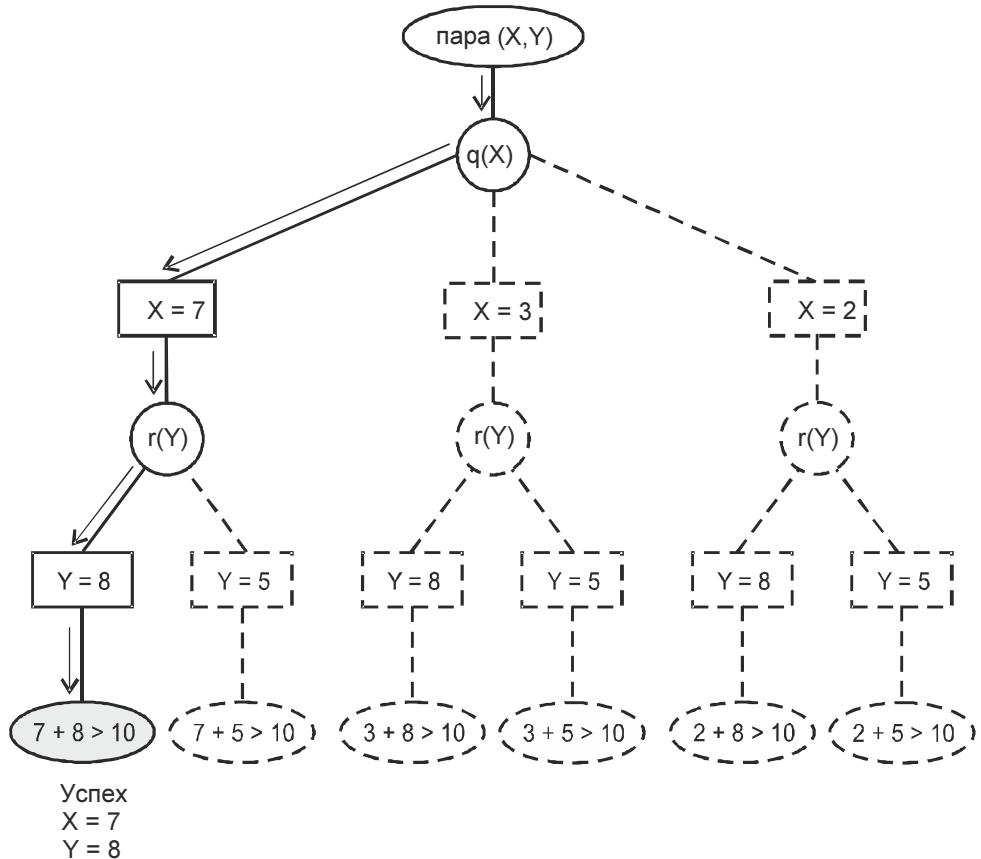


Рис. 6.4. Оптимизированное дерево поиска первого решения

После нахождения первого успешного решения $X=7, Y=8$, оно будет напечатано на экране, курсор будет переведен на новую строку, а предикат `fail` вызовет откат назад — к предикату `r(Y)`. Этот предикат имеет «в запасе» еще одно решение $Y=5$, найдя которое, механизм поиска вновь начнет продвигаться по дереву в прямом направлении до тех пор, пока не произойдет откат назад либо по `fail`, либо по ложному условию $X+Y>10$. Повторяя эти действия, механизм поиска с откатом просмотрит все шесть листьев на дереве. При этом только первые три решения окажутся успешны, и в этих случаях откат назад будет вызываться предикатом `fail`. Остальные три решения окажутся неуспешны, и поэтому сами будут инициировать откат назад. На рис. 6.5 изображено дерево поиска решений для рассмотренного примера. Казалось бы, что все решения найдены и выведены на экран, задача решена. Однако это не так. Предикат `пара(X,Y)` в итоге завершился неуспешком. Дело в том, что

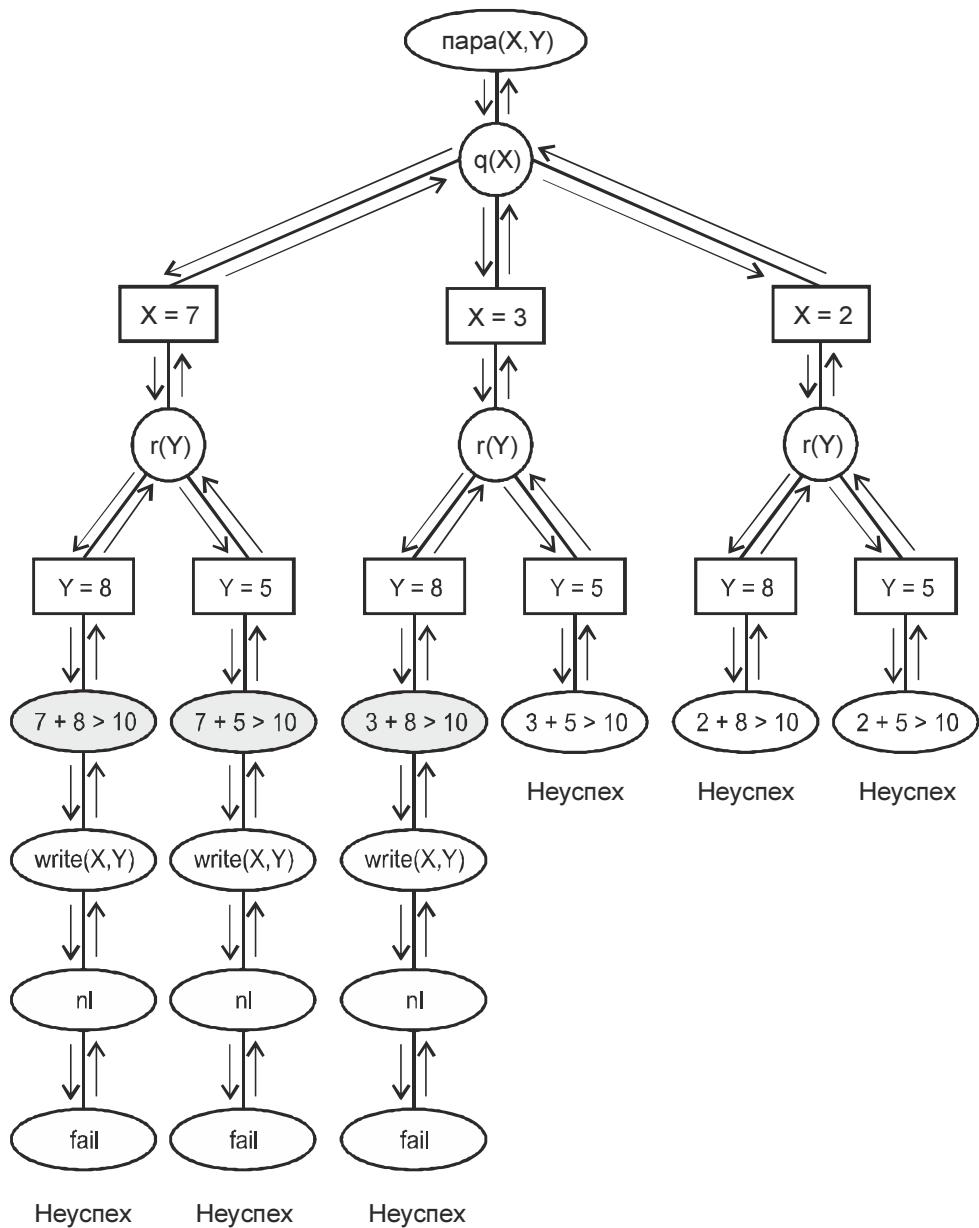


Рис. 6.5. Дерево поиска всех решений

после прохождения по всему дереву механизм поиска, «благодаря» предикату `fail`, откатится в самое начало — к корню, к голове правила `пара(X, Y)` и, не найдя больше альтернативных решений, объявит предикат неуспешным.

Исправить эту досадную ситуацию очень легко. Достаточно к первому и единственному дизъюнкту предиката `пара(X, Y)` добавить второй дизъюнкт, который всегда успешен. Его роль в Visual Prolog играет предикат `succeed`:

`q(7).` `q(3).` `q(2).`

`r(8).` `r(5).`

`пара(X,Y) :- q(X), r(Y), X+Y>10, write(X,Y), nl, fail; succeed.`

В этом случае неуспех первого дизъюнкта вызовет доказательство второго дизъюнкта. На рис. 6.6 изображено дерево поиска решений для примера с двумя дизъюнктами.

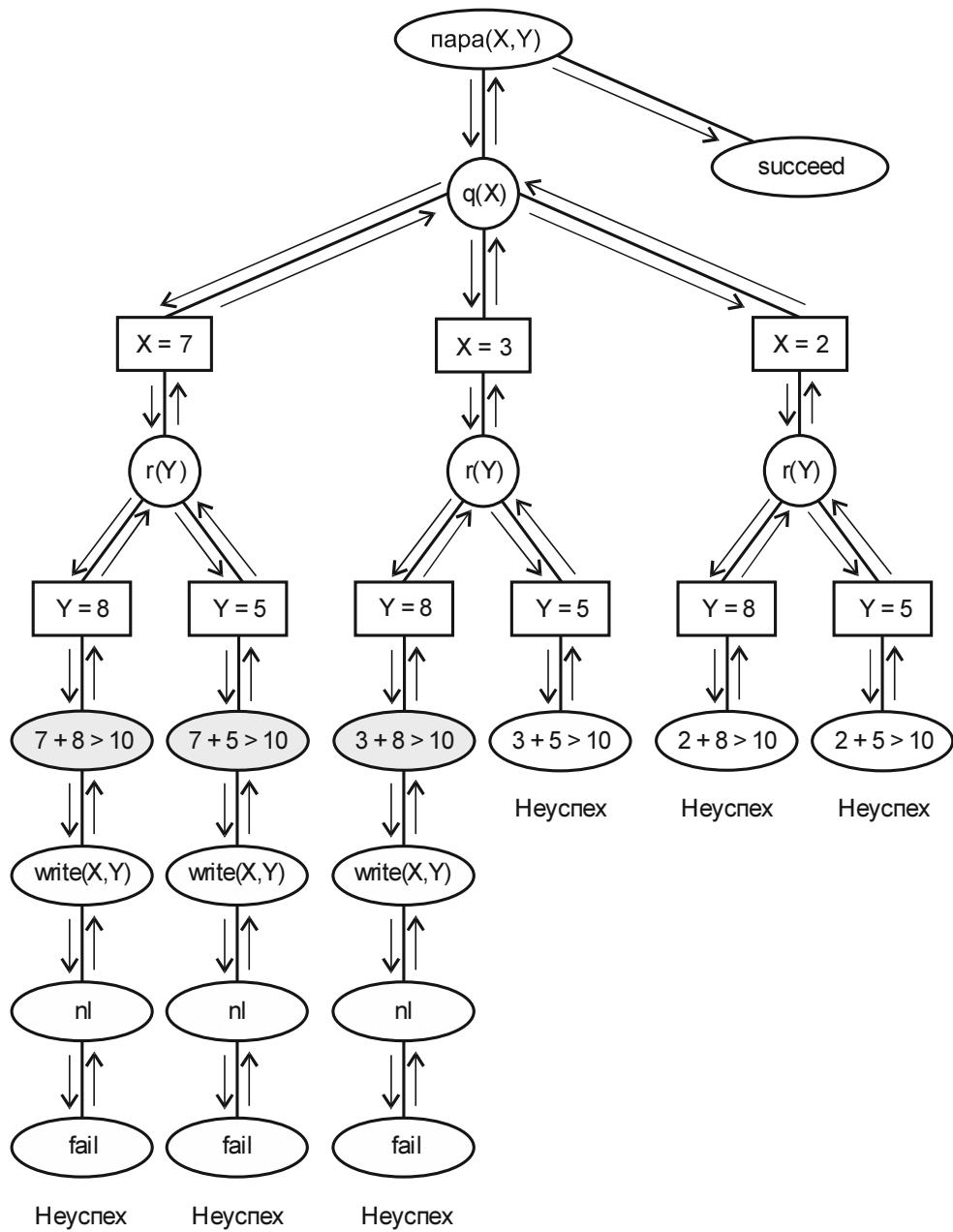


Рис. 6.6. Дерево поиска всех решений с двумя дизъюнктами

6.8. Чистый Пролог

В теории логического программирования рассматривается понятие чистого Пролога. *Чистый Пролог* — это логический язык, содержащий только чистые предикаты. *Чистые предикаты* — это предикаты, значение которых зависит только от их аргументов и не зависит от состояния памяти компьютера. Чистые предикаты не имеют побочных эффектов, т. е. они не читают из памяти какие-либо дополнительные данные и не пишут в нее какие-либо результаты вычислений. В чистом Прологе выполняются все свойства логических операций отрицания, дизъюнкции и конъюнкции. Ранее мы убедились, что свойство коммутативности относительно конъюнкции и дизъюнкции выполняется, и при этом декларативный смысл программы не изменяется. Однако сейчас пора сделать существенную оговорку — коммутативность выполняется только в чистом Прологе.

На практике чистый Пролог не применим, т. к. для реального программирования необходим ряд внелогических предикатов. *Внелогические предикаты* — это предикаты, значение которых существенно зависит от состояния памяти компьютера, или это предикаты, которые изменяют память компьютера. Внелогические предикаты вызывают побочный эффект, т. к. имеют доступ по чтению и записи в память компьютера и поэтому являются «нечистыми».

Visual Prolog имеет множество внелогических предикатов. Классическим примером их являются предикаты ввода и вывода. Предикаты ввода читают информацию из устройств ввода, а предикаты вывода записывают информацию в устройства вывода. Над внелогическими предикатами не выполняется свойство коммутативности относительно конъюнкции и дизъюнкции. Например, конъюнкция простое(X) and write(X) имеет декларативный смысл «вывести на экран число X в случае, если оно простое». А конъюнкция write(X) and простое(X) сначала выводит число X, независимо от того, простое оно или нет, а потом проверяет его простоту.

ГЛАВА 7



Предикаты в Visual Prolog

Предикаты в Visual Prolog объявляются в разделе `predicates`, а предложения предикатов определяются в разделе `clauses`. Ключевое слово `clauses` переводится с английского как «предложения», а произносится — «клозы». Предложения предиката допускается называть *правилами*. Кроме этого в Visual Prolog отдельно выделяют факты. Они объявляются в разделе `facts`, а предложения фактов определяются в разделе `clauses`.

7.1. Объявление и определение предикатов

Объявление предикатов формально выглядит так:

имя : (список_доменов) список_режимов .

Имя предиката, как вы уже знаете, является строчным идентификатором. Список доменов состоит из имен доменов аргументов, разделяемых запятой. Справа от домена может указываться его пояснение с большой буквы, которое компилятор игнорирует. В списке доменов Visual Prolog разрешает указывать эллипсис — произвольное количество аргументов в предикате. Эллипсис указывается многоточием ... и должен быть последним среди всех аргументов предиката.

Список режимов содержит перечень режимов, разделенных пробелами. Каждый режим предиката представляется в следующем виде:

режимы_детерминизма список_шаблонов_потоков

7.1.1. Режимы детерминизма

Режим детерминизма задается одним из ключевых слов:

- `procedure`
- `multi`
- `determ`
- `nondeterm`
- `failure`
- `erroneous`

Опишем эти ключевые слова подробно:

- предикат с режимом детерминизма `procedure` имеет только одно решение и, как следствие, не помещает в стек адрес возврата. Он не может быть неуспешен. Откат назад «перепрыгивает» через такой предикат. Режим `procedure` является режимом по умолчанию. Предикаты с режимом `procedure` аналогичны процедурам и функциям императивных языков программирования. Режим `procedure` другими словами можно назвать *режисмом обязательного выполнения предиката*;
- предикат с режимом детерминизма `multi` имеет множество решений (не менее одного) и помещает в стек адрес возврата в случае, когда «в запасе» осталось хотя бы одно альтернативное решение. Откат назад останавливается на таком предикате для поиска новых решений. Когда получено последнее решение предиката, то в стек ничего не заносится, и следующий откат «перепрыгнет» такой предикат;
- предикат с режимом детерминизма `determ` может быть или успешен, или неуспешен. Он не помещает в стек адрес возврата. Откат назад «перепрыгивает» через такой предикат. Режим `determ` иногда называют детерминированным режимом;
- предикат с режимом детерминизма `nondeterm` может быть как неуспешен, так и иметь множество решений. Откат назад останавливается на таком предикате для получения нового решения. Режим `nondeterm` иногда называют *недетерминированным режимом*;
- предикат с режимом детерминизма `failure` всегда неуспешен и, следовательно, всегда вызывает откат назад и не помещает в стек адрес возврата. Пример — стандартный предикат `fail`. Другой пример — предикат `1<0`. Предикаты с таким режимом используются для принудительного вызова отката в циклах с откатом;
- предикат с режимом детерминизма `erroneous` всегда вызывает исключительную ситуацию и именно для этого он и используется.

Обобщая приведенные описания, можно разделить все предикаты на две группы. Одни предикаты при своем выполнении никогда не оставляют в стеке актуальный адрес возврата. К ним относятся предикаты со следующими режимами детерминизма:

- `procedure`
- `determ`
- `failure`
- `erroneous`

К другой группе относятся предикаты с режимами детерминизма, которые оставляют в стеке актуальный адрес возврата:

- `multi`
- `nondeterm`

Однако здесь следует иметь в виду, что при выполнении последнего правила предикаты второй группы в стек не помещают ничего.

7.1.2. Шаблон потоков

Шаблон потоков в оригинальном описании языка называется *flow pattern*.

Поток параметра или, что то же самое, *поток аргумента* задает направление передачи каждого аргумента предиката. Если при вызове предиката значение аргумента задано, т. е. аргумент является входным, то направление выражается буквой *i* — первой буквой английского слова *input* (вход). Если же при вызове предиката значение аргумента не задано, то оно должно вычисляться в ходе работы предиката. Такой аргумент является выходным, а направление его передачи обозначается английской буквой *o* — первой буквой английского слова *output* (выход).

Шаблон потоков выражается одним из трех способов:

- *Первый* способ записи шаблона потоков — это обрамленный скобками перечень потоков параметров, разделенных запятой. Каждый поток соответствует аргументу предиката.

Если все аргументы входные — например: *(i,i,i)*, то такой шаблон потоков считается шаблоном по умолчанию и его можно опустить.

Поток для эллипсиса указывается троеточием, как, собственно, и сам эллипсис.

В случае, когда аргументом предиката является составной терм, и вам надо указать потоки для отдельных аргументов этого терма, а не всего терма в целом, то в качестве шаблона потока может служить функция терма с потоками всех своих аргументов. Например: *(i,o,tt(i,i,o))*.

Когда аргументом предиката является список, и вам надо указать потоки для отдельных элементов этого списка и, возможно, хвоста списка, то можно использовать явное указание потоков внутри списка — например: *([i,o,i|o])*.

- *Второй* способ записи шаблона потоков — это указание произвольного потока всех параметров предиката с помощью ключевого слова *anyflow*. Такой способ используется только для объявления предикатов внутри раздела реализации класса, т. е. только для локальных предикатов. В этом случае все возможные шаблоны потоков определяются во время компиляции.

Нельзя указывать шаблон потоков без режима детерминизма. Но указывать режим детерминизма без шаблона потоков разрешено, и в этом случае шаблон по умолчанию содержит все входные аргументы.

- *Третий* способ определения шаблона потоков применим для предикатов, имеющих один режим детерминизма, и заключается в указании атрибута *[out]* справа от каждого выходного аргумента. Если атрибут *[out]* отсутствует, то аргумент является входным. Сам шаблон при этом не указывается. Например, объявление предиката:

```
fff:(integer,real,real [out], string [out]) determ.
```

означает, что предикат *fff* имеет режим детерминизма *determ*, первый и второй аргументы входные, третий и четвертый — выходные.

7.1.3. Определение предикатов

Определение предиката в общем случае выглядит так:

```
имя(аргументы) :- тело_предиката.
```

Тело предиката и знак `:`- могут быть опущены. Предикаты с одинаковым именем, но разной арностностью, считаются различными.

Напоминание

Рассматриваемые здесь примеры предназначены для ознакомления без запуска в Visual Prolog.

Пример предиката с режимом `procedure`. Предикат `pr/2` выводит на экран текст, содержащий аргументы, возведенные в указанные степени:

```
predicates
    pr:(real Аргумент1, real Аргумент2) procedure (i,i).
clauses
    pr(X,Y):-write(X^4," + ",Y^3.2).
```

В этом примере первый аргумент возводится в четвертую степень, второй аргумент в степень 3.2. При выводе на экран между вычисленными значениями ставится знак плюс. Комментарии `Аргумент1` и `Аргумент2` являются необязательными пояснениями и во время компиляции пропускаются. Режим `procedure` и шаблон со всеми входными параметрами `(i,i)` являются значениями по умолчанию и их можно опустить. Этот пример в краткой форме записи имеет вид:

```
predicates
    pr:(real,real).
clauses
    pr(X,Y):-write(X^4," + ",Y^3.2).
```

Почему у этого предиката режим `procedure`? Потому что этот предикат имеет одно решение — вывод на экран, предикат `write` сам является процедурой, и других альтернативных решений нет.

Для выполнения предиката надо его вызывать — например: `pr(2.45, 1.8)`. На экран будет выведен текст:

```
36.03000625 + 6.5595193318447
```

Если в этом предикате предусмотреть второй вариант вывода на экран — например, со знаком минус между значениями аргументов, то режим детерминизма будет `multi`:

```
predicates
    pr:(real,real) multi.
clauses
    pr(X,Y):-write(X^4," + ",Y^3.2); write(X^4," - ",Y^3.2).
```

После вывода первого варианта решения машина логического вывода занесет в стек адрес второго правила. Если возникнет откат назад, то будет выполнено второе правило этого предиката: `write(X^4," - ",Y^3.2)`.

Рассмотрим предикат `pr/2`, вычисляющий площадь круга по заданному радиусу `R`. В этом предикате первый аргумент является входным. Второй аргумент является выходным и возвращает площадь круга:

```
predicates
    p:(real,real) procedure (i,o).
clauses
    p(R,S) :- S = 3.14 * R*R.
```

Выходной аргумент можно указать с помощью атрибута `[out]`. При такой записи указания шаблона потоков в скобках не требуется:

```
predicates
    p : (real,real [out]). 
clauses
    p(R,S) :- S = 3.14 * R*R.
```

Следующий пример показывает несколько вариантов режимов детерминизма для одного предиката. Предикат `fact/2` описывает таблично заданные значения факто-риала для первых пяти чисел 0, 1, 2, 3, 4:

```
predicates
    fact:(unsigned Аргумент,unsigned Факториал)      determ (i,i) (i,o)
                                                    nondeterm (o,i)
                                                    multi (o,o).
clauses
    fact(0,1).
    fact(1,1).
    fact(2,2).
    fact(3,6).
    fact(4,24).
```

Режимы детерминизма зависят от шаблонов потоков. Рассмотрим шаблон `(i,i)`, который представляет собой пару заданных чисел. Для такой пары этот предикат будет или успешным, если эта пара описана в предложениях, или неуспешным, если такой пары нет. Можно подумать, что если в предложениях описано много пар чисел, то этот предикат для шаблона `(i,i)` должен быть `multi`. Однако это не так. Все дело в том, что первый аргумент в этой паре не повторяется, и во время компиляции Visual Prolog это определяет. Следовательно, если заданная пара найдена, то второй такой пары точно нет. Поэтому и определен режим `determ`. Те же самые рассуждения справедливы и для шаблона `(i,o)`, при котором для заданного первого аргумента можно найти не более одного значения второго аргумента или вообще не найти ни одного.

К примеру, цель `fact(3,6)` успешна, и второй такой пары чисел 3,6 нет. А цель `fact(3,5)` неуспешна. Для шаблона потока `(i,o)` цель `fact(3,X)` успешна и возвращает значение `X=6`. А цель `fact(8,X)` неуспешна.

Предикат `fact/2` при шаблоне потоков `(o,i)` ведет себя иначе. В предложениях предиката второй аргумент со значением 1 повторяется, и во время компиляции

Visual Prolog это видит. Для заданного второго аргумента можно найти более чем одно значение первого аргумента, а можно не найти ни одного. Следовательно, при шаблоне потока `(o, i)` предикат имеет режим `nondeterm`. К примеру, цель `fact(X, 1)` успешна и вернет `X=0`, кроме того, в стек будет задавлен адрес следующего правила этого предиката. Если произойдет откат, то будет выполнено правило, находящееся по адресу, вытолкнутому из стека, и получено еще одно решение `X=1`. Цель `fact(X, 8)` будет неуспешна, при этом в стек ничего не помещается и вызывается откат назад.

Предикат `fact/2` при шаблоне потоков `(o, o)` может вернуть все пять решений. После нахождения очередного решения механизм поиска с возвратом размещает в стеке адрес правила для получения альтернативного решения. Причем, после последнего, пятого, решения в стек ничего помещено не будет, и поэтому шестой откат перепрыгнет этот предикат.

Кому-то может показаться, что определять режимы предикатов и шаблоны потоков весьма утомительно. Здесь можно предложить простой способ избежать затруднений в этом вопросе на первых порах. Способ заключается в том, что вначале следует описывать правила предикатов в разделе `clauses`, а потом их объявлять в разделе `predicates` без указания режимов и шаблонов потоков. Во время компиляции Visual Prolog осуществит глобальный анализ потока данных и выведет в окне ошибок правильные режимы и шаблоны потоков. Останется только скопировать их в соответствующие строки раздела `predicates`.

7.2. Объявление и определение функций

В оригинальном описании языка Visual Prolog предикаты разделяются на обычные (*ordinary*) и функциональные (*functional*). Для краткости обычные предикаты мы называем предикатами, а функциональные — *функциями*. Отличием функциональных предикатов от обычных является то, что кроме множества определения и множества значений истинности {«успех», «неуспех»}, функциональные предикаты имеют еще и множество значений функции. Значения из этого множества возвращаются функцией в программу — подобно тому, как возвращают свои значения синус или логарифм.

Объявление функций сходно с объявлением предикатов, но дополнительно, после знака стрелки, включает указание домена того значения, которое возвращает функция:

```
имя : (список_доменов) -> домен_функции список_режимов.
```

Visual Prolog допускает использование функции там, где допустимо использование выражения, и разрешает композицию функций.

Определение функции в общем случае выглядит так:

```
имя (аргументы) = возвращаемое_значение :- тело_функции.
```

Здесь имя функции, ее аргументы и тело функции задаются так же, как и при определении предикатов. А возвращаемое значение присваивается голове функции по-

средством знака равенства. Тело функции вместе со знаком `:-` могут быть опущены. В справочной системе Visual Prolog функции обозначаются как обычные предикаты, но с добавлением стрелки: имя/арность \rightarrow .

Напоминание

Рассматриваемые здесь примеры предназначены для ознакомления без запуска в Visual Prolog.

Рассмотрим пример двухарной функции `s/2 \rightarrow` , имеющей режим `procedure` и возвращающей сумму квадратов своих аргументов:

```
domains
    uReal = digits 16 [0...].
predicates
    s:(real,real) -> uReal.
clauses
    s(X,Y) = X*X+Y*Y.
```

В этом примере:

- объявлен тип `uReal`, представляющий собой неотрицательные вещественные числа. Именно такие числа и получаются в результате суммирования квадратов чисел;
- функция `s` имеет режим `procedure` и шаблон потоков (`i,i`);
- возвведение в квадрат `X^2` заменено умножением `X*X`.

Вызов этой функции можно записать так: `D = s(81,-6.9)`, в результате произойдет сопоставление значения этой функции и переменной `D`. Другой вариант — использование функции как аргумента: `write(s(81,-6.9))`.

В следующем примере функция `factorial/1 \rightarrow` описывает таблично заданные значения факториала для первых пяти чисел 0, 1, 2, 3, 4:

```
predicates
    factorial:(unsigned) -> unsigned determ (i) multi (o).
clauses
    factorial(0) = 1.
    factorial(1) = 1.
    factorial(2) = 2.
    factorial(3) = 6.
    factorial(4) = 24.
```

При шаблоне потока (`i`) функция детерминирована, т. к. все перечисленные аргументы функции не повторяются. Аргументы функций в Visual Prolog могут быть выходными.

Цель `X = factorial(3)` сопоставит переменную `X` и значение 6. Эта цель может быть записана и в таком виде: `factorial(3) = X`, но результат будет тем же.

Цель `6 = factorial(Z)` найдет такое значение переменной `Z`, при котором значение функции равно 6. В нашем примере `Z=3`.

7.3. Объявление и определение фактов

Факт — это предикат, каждое предложение которого является основным термом. Предложение факта в Visual Prolog имеет только голову и не имеет тела. Факты объявляются отдельно от остальных предикатов по той причине, что в Visual Prolog такие факты считаются базой данных, которую во время выполнения программы допускается загружать из текстовых файлов и сохранять в файлах, можно также добавлять и удалять факты прямо в оперативной памяти. Такая база данных в Прологе имеет другое название — *база фактов*.

Факты объявляются в разделе: `facts – имя_бд.` Таких разделов в программе может быть сколько угодно. Они отличаются между собой именами. Имя базы фактов является строчным идентификатором. Объявление фактов формально выглядит так:

имя : (список_доменов) режим.

7.3.1. Режимы детерминизма факта

Режим детерминизма фактов задается одним из ключевых слов:

- `single`
- `determ`
- `nondeterm`

Опишем эти ключевые слова подробно:

- ❑ факт с режимом `single` представляется в программе одним и только одним предложением. Термин `single` означает в данном контексте «единственный» или «одиночный». Вызов такого факта или успешен, или неуспешен — в зависимости от результата сопоставления аргументов, и, как следствие, не помещает в стек адрес возврата. Откат «перепрыгивает» через такой факт. Факты с режимом `single` аналогичны переменным в императивных языках, областью видимости которых является класс, в котором они объявлены;
- ❑ факт с режимом `determ` представляется в программе не более чем одним предложением. Вызов такого факта или успешен, или неуспешен — в зависимости от наличия факта в программе и успешности сопоставления аргументов. Адрес возврата в стек не помещается. Откат «перепрыгивает» через такой факт;
- ❑ факт с режимом `nondeterm` представляется в программе произвольным количеством предложений. Вызов такого факта или успешен, или неуспешен — в зависимости от наличия факта в программе и успешности сопоставления аргументов. Адрес возврата помещается в стек. Откат «передоказывает» такой предикат.
Режим `nondeterm` является режимом по умолчанию.

Шаблон потоков параметров фактов не указывается, ибо по определению факта как основного терма, все его аргументы связаны, т. е. являются входными.

7.3.2. Определение фактов

Следующий пример демонстрирует объявление месяцев и дней недели в разделе календарь. Кроме этого в базе фактов предусмотрен раздел праздники, содержащий объявление даты праздников. В разделе clauses приведено определение фактов:

```

domains
    unsigned3 = bitsize 3 [1..].
facts - календарь
    день_недели : (unsigned3 Номер, string Название).
    месяц : (unsigned Номер, string Название, unsigned Продолж).
facts - праздники
    праздник : (unsigned День, unsigned Номер_Месяца)
clauses
    день_недели(1, "Понедельник").
    день_недели(2, "Вторник").
    день_недели(3, "Среда").
    день_недели(4, "Четверг").
    день_недели(5, "Пятница").
    день_недели(6, "Суббота").
    день_недели(7, "Воскресенье").

```

Факты месяц/3 и праздник/2 в программе отсутствуют — режим nondeterm это допускает. При необходимости экземпляры этих фактов можно добавить программно с помощью встроенного предиката assert/1.

В качестве номера дня недели используется введенный в программе домен unsigned3. Можно задействовать и встроенный домен unsigned. Режимом детерминизма факта день_недели/2 является режим nondeterm. В табл. 7.1 представлены результаты выполнения целевых запросов.

Таблица 7.1. Результаты выполнения целевых запросов

Цель	Результат
день_недели(3, X)	X="Среда"
день_недели(X, "Вторник")	X=2
день_недели(4, "Четверг")	Успех
день_недели(4, "Среда")	Неуспех
день_недели(X, Y)	X=1, Y="Понедельник"

7.3.3. Факты-переменные

Факт-переменная — это обычновенный факт с одним аргументом, имеющий режим детерминизма single. Синтаксически он напоминает переменную, допускающую разрушающее присваивание. В объявлении факта-переменной допускается не указывать начальное значение:

```
имя_факта_переменной : домен.
имя_факта_переменной : домен := начальное_значение.
```

В отличие от переменных Пролога, имя факта-переменной является строчным идентификатором, т. е. начинается со строчной буквы. Начальное значение может быть константой, или вычисляемым во время компиляции выражением, или специальным ключевым словом `erroneous`. Ключевое слово `erroneous` переводится как «ошибочный» и используется в тех случаях, когда значение факта-переменной на этапе проектирования программы неизвестно и определяется во время выполнения программы. В случае использования факта-переменной со значением `erroneous` в выражении будет вызвана исключительная ситуация. Для проверки значения факта-переменной существует специальный предикат `isErroneous(факт_переменная)`. Он будет успешен, если факт-переменная имеет значение `erroneous`, иначе — неуспешен.

Составной символ `:=` означает *разрушающее присваивание* и применим только к фактам-переменным. Запись `x:=x+1` означает «взять текущее значение факта-переменной `x`, прибавить к нему единицу и результат записать в факт-переменную `x`, разрушив прежнее значение».

Сравнивая с императивными языками, можно сказать, что факт-переменная является переменной класса. С помощью факта переменной легко организовывать счетчики, аккумуляторы, глобальные переменные для передачи параметров из одного места программы в другое и т. п.

Рассмотрим использование факта-переменной в качестве счетчика. Пусть необходимо подсчитать количество нечетных элементов во множестве `{7, 2, 23, -1}`. Программа будет содержать элементы множества как факты, например: `m/1`. Факт-переменную, играющую роль счетчика, назовем оригинальным именем `i` и инициализируем ее значением `0`. Предикат, осуществляющий перебор элементов и подсчет нечетных чисел, назовем: `счет/1`.

```
facts — множество
m : (integer).
i : unsigned := 0.
predicates
    счет : (unsigned) procedure (o).
clauses
    m(7). m(2). m(23). m(-1).
    счет(_) :- m(X), X mod 2 = 1, i:=i+1, fail.
    счет(i).
```

Целевым запросом для этой программы является предикат `счет(Z)` со свободной переменной `z`. Выполнение начнется с первого предложения этого предиката, в теле которого будет найден первый факт `m(7)`. Поскольку $7 \bmod 2 = 1$, то счетчик `i` будет инкрементирован, а следующий предикат `fail` вызовет откат назад. Предикаты `i := i+1` и `X mod 2 = 1` являются процедурами и не помещают в стек адресов возврата. Следовательно, откат «прыгнет» сразу к недетерминированному предикату `m(X)` и найдет второй факт `m(2)`. Предикат `2 mod 2 = 1` окажется неуспешным,

что вызовет откат назад, и инкремента счетчика не произойдет. Вот в таком порядке и будет реализован перебор всех фактов. После того, как для числа -1 счетчик будет инкрементирован, предикат `fail` вызовет последний откат в первом предложении предиката `счет/1`, и будет вычислено второе предложение `счет(i)`. Здесь вместо факта-переменной `i` будет подставлено ее значение 3, и запрос `счет(Z)` завершится успешно со значением `Z=3`.

Хорошим стилем считается обнуление счетчика после завершения перебора, чтобы следующий запрос также начал счет с нуля. Обнуление можно сделать в запросе — например:

```
счет(Z), i:=0.
```

Если предполагается подсчитывать элементы с накоплением в нескольких разных запросах, то обнулить счетчик следует после всех запросов:

```
счет(Z), счет1(Z), счет2(Z), i:=0.
```

Предикат `счет(Z)` возвращает результат в виде явно заданного параметра `z`. Однако существуют еще, как минимум, два варианта возвращения результата. По первому варианту предикат `счет` можно оформить как функцию:

```
predicates
    счет : () -> unsigned.
clauses
    счет() = i :- m(X), X mod 2 = 1, i:=i+1, fail; succeed.
```

Тогда запросом к программе будет предикат `Z=счет()`.

Согласно второму варианту предикат `счет/1` можно сделать нульярным, т. е. без аргументов, а результат вычислений получать из факта-переменной `i` после выполнения запроса:

```
predicates
    счет : () .
clauses
    счет() :- m(X), X mod 2 = 1, i:=i+1, fail; succeed.
```

Тогда запрос к программе может быть такой:

```
счет(), write(i).
```

Рассмотрим пример с двумя условиями. Пусть необходимо в том же множестве `{7, 2, 23, -1}` подсчитать количество нечетных элементов, меньших 10.

```
facts - множество
    m : (integer).
    i : unsigned := 0.
predicates
    счет : (unsigned) procedure (o).
clauses
    m(7). m(2). m(23). m(-1).
    счет(i):-m(X), X mod 2 = 1, X<10, i:=i+1, fail; succeed.
```

Целевым запросом для этой программы является предикат `счет(Z)` со свободной переменной `Z`. Проверка выполнения двух условий записывается через конъюнкцию: $X \bmod 2 = 1$, $X < 10$. Такую проверку пройдут всего два элемента 7 и -1. Следовательно, ответ `Z=2`.

Рассмотрим пример с двумя независимыми счетчиками. Пусть необходимо в том же множестве $\{7, 2, 23, -1\}$ подсчитать количество нечетных элементов, меньших 10, и отдельно подсчитать количество элементов, принадлежащих диапазону $-5 < X < 10$. Конечно, для каждого счетчика можно написать свой предикат:

```
счетA(i):-m(X), X mod 2 = 1, X<10, i:=i+1, fail; succeed.
```

```
счетB(j):-m(X), X > -5, X<10, j:=j+1, fail; succeed.
```

В этом случае целевой вызов осуществляется конъюнкцией `счетA(X), счетB(Y)`, которая сначала выполнит перебор всех элементов множества $\{7, 2, 23, -1\}$, инкрементируя счетчик `i` при истинности условий в теле предиката `счетA`, и только потом выполнит повторный перебор элементов этого же множества, инкрементируя счетчик `j` при истинности условий в теле предиката `счетB`. Такой двойной перебор элементов множества весьма неэффективен. Для ускорения процедуры подсчета следует при первом переборе элементов множества проверять условия инкремента каждого счетчика. Однако конъюнкцию условий:

```
m(X), X mod 2 = 1, X<10, i:=i+1, X > -5, X<10, j:=j+1, fail
```

мы использовать не можем, т. к. при невыполнении условий инкремента счетчика `i` не будут проверяться условия инкремента счетчика `j`.

Правильное решение здесь заключается в использовании дизъюнкции условий инкремента счетчиков:

```
facts — множество
  m : (integer).
  i : unsigned := 0.
  j : unsigned := 0.

predicates
  счет : (unsigned,unsigned) procedure (o,o).

clauses
  m(7). m(2). m(23). m(-1).
  счет(i,j):-m(X),
    (X mod 2 = 1, X<10, i:=i+1; X>-5,X<10, j:=j+1),
    fail;
    succeed.
```

Целевым запросом является предикат `счет(X,Y)`. Две альтернативные ветки проверки условий заключены в скобки. Предикат `fail`, вынесенный за скобки, выполняет две функции. Во-первых, он инициирует выполнение второй ветви в случае, если первая ветвь завершилась успешно. Во-вторых, он вызывает откат к предикату `m(X)`, инициируя перебор всех элементов множества в случае, если вторая ветвь завершилась успешно. Во всех остальных случаях откат будут вызывать неуспешные проверки условий. В рассматриваемом примере цель `счет(X,Y)` завершится успешно со значениями: `X=2` и `Y=3`.

7.4. Операции над разделами базы фактов

Как было только что показано, база фактов может содержать несколько именованных разделов. База фактов с именем, например, календарь содержит только те факты, которые объявлены в разделе программы facts – календарь. Факты базы могут быть сохранены в файле и загружены из файла в программу. Формат файла, в котором хранятся факты, текстовый. Имя и расширение файла определяются программистом. Если факты расположены в программе, то после компиляции они будут храниться в исполнимом файле и загружаться при его запуске.

Предикаты, выполняющие операции над разделами базы фактов, находятся в классе file. Справедливости ради надо заметить, что пакет stream также содержит аналогичные предикаты, но этот пакет мы рассмотрим позже.

Сохранение фактов раздела facts – календарь в файл "file.txt" осуществляется предикатом `file::save("file.txt", календарь)`. Все факты этого раздела, находящиеся в программе, будут сохранены в файле текущего каталога. При необходимости записать базу фактов в файл, расположенный по другому пути, следует указать полный путь и имя файла. Если такого файла нет, то он будет создан. Если файл есть, то его содержимое будет перезаписано. Каждый факт будет записан в файле в отдельной строке, завершающейся точкой. После сохранения фактов в файле, оригинальная база фактов, конечно же, останется в памяти, и с ней можно продолжать работать. Файл базы фактов можно редактировать в любом текстовом редакторе, не нарушая структуру фактов.

При необходимости можно указать тип текстового файла: Unicode или ANSI. Для этого следует воспользоваться трехарным предикатом:

```
file::save("file.txt", календарь, Unicode)
```

При `Unicode=true()` файл будет содержать факты в кодировке Unicode. При `Unicode=false()` — в кодировке ANSI.

Загрузка фактов раздела facts – календарь из файла "file.txt" осуществляется предикатом `file::consult("file.txt", календарь)`. Следует иметь в виду, что этот предикат *добавляет* факты в память, не удаляя уже находящиеся там факты. Поэтому многократная загрузка фактов без очистки памяти может «наплодить» много дубликатов фактов и, как следствие, быть причиной неправильной работы программы. Файл базы данных может содержать комментарии к фактам. Загружаемый файл базы фактов может быть создан не только предикатом `save`, но и предикатом `write`, или вручную в текстовом редакторе — например, в том же NotePad. Если файл с фактами содержит синтаксические ошибки, то во время загрузки возникнет исключительная ситуация.

Предикат `file::consult("file.txt", календарь, Unicode)` аналогичен предикату `consult/2`, но дополнительно возвращает кодировку файла. Если `Unicode=true()`, то файл содержит факты в кодировке Unicode. Если `Unicode=false()` — то в кодировке ANSI.

Для того чтобы избежать проблем с дублированием фактов при неосторожном использовании предиката `consult`, в классе `file` есть предикаты *перезагрузки фактов*:

```
reconsult(FileName, имя_бд)
и
reconsult(FileName, имя_бд, Unicode)
```

Эти предикаты во всем аналогичны соответствующим предикатам `consult`, за исключением того, что перед загрузкой фактов из файла они удаляют из памяти все текущие факты раздела `имя_бд`.

7.5. Операции над фактами

Во время выполнения программы Visual Prolog позволяет добавлять в программу новые факты и удалять существующие. Предикаты для совершения этих операций встроены в язык и не требуют указания класса.

Добавление факта `месяц(1,"январь",31)` в базу фактов осуществляется предикат `assert(месяц(1,"январь",31))`. Следует иметь в виду, что добавление факта, объявленного с режимом `single`, осуществляет стирание текущего факта и запись нового экземпляра. Добавление нового факта, объявленного с режимом `determ`, успешно, если факта в программе нет. Если факт с режимом `determ` в программе есть, то при добавлении второго экземпляра будет вызвана исключительная ситуация. Добавлять факты, имеющие режим `nondeterm`, можно сколь угодно много.

Предикат `assert/1` добавляет факт в конец группы фактов. Эту же операцию совершает предикат `assertz/1`, полностью аналогичный предикату `assert/1`. Добавление факта в начало группы фактов осуществляется предикатом `asserta/1`. Эти предикаты легко запомнить и различать по последней букве: `a` — первая буква английского алфавита, значит, добавление в начало, `z` — последняя буква английского алфавита, значит, добавление в конец. Добавляя факты в конец или в начало, можно создать базу фактов с желаемым порядком расположения фактов в ней. Это влияет на ход выполнения программы, т. к. поиск фактов Пролог осуществляет сверху вниз. В Прологе не существует операции добавления факта в произвольную позицию базы фактов.

Удаление факта `месяц(1,"январь",31)` осуществляется предикатом `retract(месяц(1,"январь",31))`. Этот предикат удаляет первый сопоставимый факт из базы фактов. Предикат будет неуспешен, если сопоставимого факта в базе нет. Факт `месяц(1,"январь",31)` может содержать анонимные переменные. Например, для удаления месяца, имеющего 31 день, надо использовать предикат `retract(месяц(_,_,31))`, при этом будет удален первый найденный месяц с указанным признаком при просмотре базы фактов сверху вниз. Попытка удаления факта с режимом детерминизма `single` приводит к возникновению исключительной ситуации.

Удаление всех фактов, сопоставимых с `fact`, выполняет предикат `retractall(fact)`. Этот предикат всегда успешен, даже если нет ни одного факта для удаления. На-

пример, для удаления всех месяцев, имеющих 31 день, надо воспользоваться предикатом `retractall(месяц(_, _, 31))`.

Удаление всех фактов раздела имя_бд выполняет предикат `retractFactDb(имя_бд)`. Факты с режимом `single` невидимы для этого предиката.

7.6. Встроенные предикаты Visual Prolog

Visual Prolog имеет ряд встроенных предикатов, описание которых следует далее. Так как эти предикаты встроены в язык, то имя класса не указывается, однако префикс `::` перед именем встроенных предикатов допускается. Этот префикс полезен в той ситуации, когда вы в классе с именем `aaa` определили собственный предикат, имеющий такое же имя, что и некоторый встроенный предикат, — например, `assert`. Тогда вызов встроенного предиката надо предварять префиксом `::assert`. А свой предикат вызывать с указанием класса `aaa::assert`.

7.6.1. Предикаты для работы с фактами базы данных

□ `assert:(Fact)` — предикат `assert(Fact)` вставляет `Fact` во внутреннюю базу данных после последнего сохраненного факта. `Fact` должен быть термом, принадлежащим домену внутренней базы данных. Предикат `assert/1`, примененный к факту `single`, заменяет существующий факт указанным. Предикат `assert/1` действует аналогично предикату `assertz/1`.

Исключение: попытка добавить факт, объявленный как `determ`, если такой факт в БД уже существует.

□ `asserta:(Fact)` — вставляет указанный факт в начало внутренней базы данных. Предикат `asserta(Fact)` вставляет `Fact` во внутреннюю базу данных перед всеми другими сохраненными фактами. `Fact` должен быть термом, принадлежащим домену внутренней базы данных. Предикат `asserta/1`, примененный к факту `single`, заменяет существующий факт указанным (см. также `assert/1` и `assertz/1`).

Исключение: попытка добавить факт, объявленный как `determ`, если такой факт в БД уже существует.

□ `assertz:(Fact)` — предикат `assertz(Fact)` действует точно так же, как и предикат `assert/1`.

□ `retract:(FactTemplate) nondeterm anyflow` — здесь `FactTemplate` должен быть фактом с произвольным уровнем описания структуры терма. Предикат `retract/1` удаляет первый факт, который сопоставляется с термом `FactTemplate` в базе данных. Неуспешен, если ни один факт не связан. Используя цикл с откатом, можно удалить из базы данных все факты, которые сопоставляются с термом `FactTemplate`.

`FactTemplate` может содержать анонимные переменные. Имя анонимной переменной может состоять либо из единственного знака подчеркивания `_`, либо начинаться со знака подчеркивания `_AnyValue`.

Предикат:

```
retract(person("Hans", _Age)),
```

удалит первый факт person, имеющий "Hans" в качестве первого аргумента, и любой второй аргумент. Когда удаляется факт, который объявлен как `determ`, вызов `retract/1` будет детерминированным.

Предикат `retract/1` не может быть использован для удаления фактов `single` или фактов-переменных. Будьте осторожны, вызывая `retract/1` со свободным аргументом `FactTemplate`, когда база данных содержит факты, объявленные как `single`. Если вы удаляете факт `single`, то во время выполнения генерируется ошибка. Предикат `retract/1` неуспешен, когда в базе данных нет фактов, сопоставимых с указанным термом `FactTemplate`.

- `retractall: (FactTerm)` — здесь `FactTemplate` должен быть фактом с произвольным уровнем описания структуры терма. Предикат `retractall/1` удаляет из базы данных все факты, которые сопоставляются с термом `FactTemplate`. Он всегда успешен, даже если нет ни одного факта для удаления. Попытка удаления факта `single` приводит к ошибке во время компиляции.

Невозможно получить какие-либо выходные значения из предиката `retractall/1`. По этой причине переменные при вызове должны быть или связанны, или анонимны. Заметим, что в `FactTemplate` свободные переменные должны быть, безусловно, анонимными и обозначаться только одним знаком подчеркивания. В отличие от предиката `retract/1`, в котором можно использовать условно анонимные переменные — начинающиеся со знака подчеркивания (подобно `_AnyValue`), в предикате `retractall/1` нельзя использовать такие условно анонимные переменные. Шаблон вызова предиката:

```
retractall(FactTemplate)
```

- `retractFactDb: (FactDB)` — удаляет все факты из именованной внутренней базы данных `FactDB`. Заметим, что факты `single` и факты-переменные удалить невозможно — они невидимы для этого предиката. Шаблон вызова предиката:

```
retractFactDb(FactDB)
```

- `isErroneous: (FactVariable) determ` — предикат успешен, если указанная факт-переменная `factVariable` имеет значение `erroneous`, иначе — неуспешен. Шаблон вызова предиката:

```
isErroneous(factVariableName)
```

7.6.2. Предикаты контроля потока параметров

- `bound: (Variable) determ` — предикат `bound(Variable)` успешен, если переменная `Variable` связана, и неуспешен, если она свободна. Предикат `bound` используется для контроля потока переменных и воспринимает переменную `Variable` как связанную, если любая часть этой переменной связана.

- `free:(Variable)` `determ` — предикат `free(Variable)` успешен, если переменная `Variable` свободна, и неуспешен, если она связана. Предикат `free` воспринимает переменную `Variable` как связанную, если любая часть этой переменной связана.

7.6.3. Предикаты локализации места выполнения программы в исходном тексте

- `predicate_fullname:() -> string PredicateFullName` — этот предикат возвращает имя класса и имя `PredicateFullName` того предиката, в теле которого он вызывается. Предикат `predicate_fullname` может быть использован только внутри раздела `clauses`. Использование `predicate_fullname` в других местах программы вызывает ошибку компиляции.
- `predicate_name:() -> string PredicateName` — этот предикат возвращает имя `PredicateName` того предиката, в теле которого он вызывается. Предикат `predicate_name` может быть использован только внутри раздела `clauses`. Использование `predicate_name` в других местах программы вызывает ошибку компиляции.
- `programPoint:() -> programPoint programPoint` — предикат возвращает локализацию места, где он был вызван.

7.6.4. Предикаты контроля компиляции исходного текста

- `class_Name:() -> string ClassName` — этот предикат во время компиляции возвращает строку `ClassName`, которая является именем текущего интерфейса или класса.
- `sourcefile_lineno:() -> unsigned LineNumber` — возвращает номер обрабатываемой компилятором строки исходного кода.
- `sourcefile_name:() -> string FileName` — возвращает имя файла, обрабатываемого компилятором.
- `sourcefile_timestamp:() -> string TimeStamp` — возвращает строку, представляющую дату и время текущего компилируемого файла в формате `YYYY-MM-DD HH:MM:SS`, где:
 - `YYYY` — Год;
 - `MM` — Месяц;
 - `DD` — День;
 - `HH` — Час;
 - `MM` — Минута;
 - `ss` — Секунда.

7.6.5. Предикат сравнения термов

- `compare: (A, A) -> compareResult` — сравнивает два терма одного домена, возвращающее значение домена `compareResult = less; equal; greater.`

7.6.6. Предикаты преобразования типов

- `convert: (Type, Term) -> Converted_Term` — контролируемое преобразование терма. Шаблон вызова функции:

```
ReturnTerm = convert(returnDomain, InputTerm)
```

где:

- `returnDomain` — домен, в который функция `convert/2->` преобразует входной терм `InputTerm`. Домен `returnDomain` должен быть именем встроенного домена, интерфейсного домена, пользователем определенного домена, который является синонимом встроенного домена, числового домена, домена `binary` или `pointer`. Имя домена `returnDomain` должно быть указано во время компиляции и не определяться из переменной;
- `InputTerm` — значение, которое должно быть конвертировано. `InputTerm` может быть любым термом Пролога или выражением. Если `InputTerm` является выражением, то оно вычисляется перед преобразованием;
- `ReturnTerm` — возвращаемый параметр `ReturnTerm` будет принадлежать домену `returnDomain`.

Предикат `convert` выполняет явное преобразование заданного терма `InputTerm`, возвращая новый терм `ReturnTerm`, который принадлежит домену `returnDomain`. Если предикат `convert` не может выполнить требуемое преобразование, он завершается ошибкой. Подобная функциональность обеспечивается и предикатом `tryConvert/2->`, но `tryConvert` завершится неудачей и не генерирует ошибку во время выполнения, если он не может выполнить преобразование.

Допустимые преобразования:

- между числовыми доменами;
- между интерфейсными типами;
- между доменами `string` и `symbol`;
- из `binary` в `pointer`;
- для синонимов упомянутых доменов.

В отличие от этого предикат `uncheckedConvert/2->` выполняет неконтролируемое преобразование между термами любых доменов, имеющих определенный битовый размер.

Предикат `convert/2->` (или `tryConvert/2->`) выполняет контролируемое явное преобразование, когда исходный и целевой домены известны во время компиляции. Результат явного преобразования может быть одним из следующих:

- успешное преобразование в целевой домен;
- преобразование в целевой домен с генерацией контроля совместимости во время выполнения программы;
- преобразование невозможно, вывод ошибки.

Правила контролируемого явного преобразования:

- синонимичные домены конвертируются согласно тем же правилам, которые используются для базовых доменов;
- числовые домены могут быть преобразованы только в числовые домены;
- целочисленные константы представляются анонимным целочисленным доменом: [const..const];
- вещественные константы представляются анонимным вещественным доменом: digits dig [const..const], где dig — число разрядов мантиссы без незначащих нулей;
- значение домена symbol может быть преобразовано в домен string и наоборот;
- значение домена binary может быть преобразовано в домен pointer;
- домены, которые введены для интерфейса, могут быть преобразованы только в интерфейсные домены соответственно правилам, указанным далее;
- все другие домены не могут быть преобразованы.

Преобразование числовых доменов:

- числовой диапазон рассматривается в первую очередь при преобразовании. Если диапазоны источника и цели не пересекаются, то генерируется ошибка. Если диапазоны источника и цели частично пересекаются, то генерируется контроль во время выполнения программы. Также, если один из доменов вещественный, а другой — целочисленный, то целочисленный диапазон конвертируется в вещественный диапазон перед сравнением;
- когда входной терм — вещественный, а выходной — целочисленный, то предикаты convert/2-> и tryConvert/2-> округляют входное значение к ближайшему целому числу, ближайшему к нулю.

Преобразование интерфейсных типов:

- предикат convert/2-> позволяет конвертировать любой объект в любой интерфейсный тип. Действительная правильность такого преобразования контролируется во время выполнения. Где объект создан, его тип сохраняется, поэтому когда объект передается как аргумент, то он, тем не менее, помнит о своем настоящем типе. Этот тип и используется для контролируемого преобразования. Пример:

```
interface x
    supports a, b
end interface x
```

Если объект создан классом, который реализует интерфейс `x`, и когда объект передается как параметр типа `a` в некоторый предикат, то он преобразуется в объект типа `b`.

Исключения:

- ошибка контроля диапазона;
- неподдерживаемый интерфейсный тип.

□ `toAny: (Term) -> any UniversalTypeValue` — преобразует указанный терм `Term` в значение универсального терма `any`. Шаблон вызова функции:

```
UniversalTypeValue = toAny(Term)
```

Этот вызов эквивалентен вызову встроенного предиката `convert/2->`:

```
UniversalTypeValue = convert(any, Term)
```

□ `toBinary: (Term) -> binary Serialized` — преобразует указанный терм `Term` в двоичное представление `binary`. Когда терм `Term` (некоторого домена `domainName`) преобразуется в `binary`, он может быть безопасно сохранен в файл или отправлен по сети в другую программу. Позже, полученное двоичное значение `Serialized` может быть преобразовано назад, в терм Пролога — использованием функции `toTerm/1->` для обратного преобразования. Шаблон вызова функции:

```
Serialized = toBinary(Term)
```

□ `toString: (Term) -> string Serialized` — преобразует указанный терм `Term` в строку. Когда терм `Term` (некоторого домена `domainName`) преобразуется в строку, он может быть безопасно сохранен в файл или отправлен через сеть другой программе. Позже, полученная строка может быть преобразована назад, в терм `Visual Prolog` — использованием функции `toTerm/1->` для обратного преобразования. Шаблон вызова функции:

```
Serialized = toString(Term)
```

□ `toTerm: (string Serialized) -> Term.`

□ `toTerm: (binary Serialized) -> Term.`

□ `toTerm: (Type, string Serialized) -> Term.`

□ `toTerm: (Type, binary Serialized) -> Term.`

Предикат `toTerm` преобразует строковое или двоичное представление указанного терма `Serialized` в представление, соответствующее домену переменной `Term`. Домен может быть установлен точно, или определение домена доверяется компилятору.

Если домен не указан, компилятор должен быть способным определить домен для возвращаемого значения `Term` во время компиляции. Заметим, что бинарная версия предиката `toTerm` выполняет преобразование байт-в-байт и контролирует только общую совместимость переменной `Serialized` с доменом терма `Term`. Программист целиком и полностью отвечает за соответствие бинарных данных

домену, к которому принадлежит выходной терм `Term`. Предикат `toTerm` дополняет предикаты `toBinary/1->` и `toString/1->`. Когда терм `Term` (некоторого домена `domainName`) преобразуется в двоичное или строковое представление `Serialized` (посредством `toBinary/1->` или `toString/1->` соответственно), оно может быть безопасно сохранено в файл или отправлено по сети другой программе. Позже, функция `toTerm/1->` может преобразовать полученное бинарное или строковое значение `Serialized` назад, в терм `Term`. Для правильности обратного преобразования домен переменной `Term` должен быть адекватен исходному домену `domainName`. Шаблон для вызова функции:

```
Term = toTerm(Serialized)      % домен определяется компилятором
Term = toTerm(domainName, Serialized) % точное указание домена
```

Во время компиляции генерируется ошибка, если компилятор не может определить возвращаемый домен. Во время выполнения генерируется исключительная ситуация, когда предикат `toTerm` не может преобразовать строку или двоичные данные в терм указанного домена.

- `tryToTerm: (string Serialized) -> Term.`
- `tryToTerm: (binary Serialized) -> Term.`
- `tryToTerm: (Type, string Serialized) -> Term.`
- `tryToTerm: (Type, binary Serialized) -> Term.`

Предикат `tryToTerm` преобразует строковое или двоичное представление терма `Serialized` в терм `Term` подобно предикату `toTerm`. Разница между предикатами заключается только в том, что `tryToTerm` неуспешен, если он не может преобразовать входной терм в указанный домен, тогда как `toTerm` вызывает исключение.

- `tryConvert: (Type, InputTerm) -> Converted` `determ` — контролирует, может ли входной терм `InputTerm` быть преобразован в указанный домен `Type`, и возвращает результат преобразования `Converted`. Аргументы:

- `returnDomain` — домен, в который предикат `tryConvert/2->` будет пытаться преобразовать входной терм `InputTerm`. Возвращаемый домен `returnDomain` может быть доменом, доступным в текущей области видимости (классе или интерфейсе). Имя домена `returnDomain` должно быть указано во время компиляции, т. е. оно не может быть получено от переменной;
- `InputTerm` — терм, который должен быть преобразован. `InputTerm` может быть любым термом Пролога или выражением. Если `InputTerm` является выражением, то это выражение будет вычислено перед преобразованием;
- `ReturnTerm` — возвращаемый терм `ReturnTerm` будет принадлежать к домену `returnDomain`.

Правила преобразования подобны правилам в предикате `convert/2->`, но предикат `tryConvert/2->` неуспешен тогда, когда предикат `convert/2->` генерирует ошибку преобразования.

Предикат `tryConvert/2->` успешен, если соответствующее преобразование успешно. Иначе он неуспешен. Предикат `tryConvert/2->` пытается выполнить под-

лининое преобразование исходного терма `InputTerm` в значение указанного домена `returnDomain` и будет неуспешен, если требуемое преобразование не может быть выполнено. Когда предикат `tryConvert/2->` успешен, он возвращает терм `ReturnTerm`, преобразованный в указанный домен `returnDomain`.

Допустимые преобразования и правила точного преобразования — см. в предикате `convert/2->`. Шаблон вызова функции:

```
ReturnTerm = tryConvert(returnDomain, InputTerm)
```

□ `uncheckedConvert: (Type, InputTerm) -> Converted` — неконтролируемое преобразование значения в другие типы. Аргументы:

- `returnDomain` — домен, в который предикат `uncheckedConvert` небезопасно преобразует указанный терм `InputTerm`. Домен `returnDomain` может быть любым доменом, доступным в текущей области видимости, терм `ReturnTerm` должен иметь тот же битовый размер, что и входной терм `InputTerm`. Имя домена `returnDomain` должно быть указано во время компиляции, т. е. он не может получаться из переменной;
- `InputTerm` — значение, которое должно быть конвертировано. `InputTerm` может быть любым термом Пролога или выражением. Если входной терм `InputTerm` является выражением, то его значение будет вычислено перед преобразованием;
- `ReturnTerm` — возвращаемый параметр `ReturnTerm` будет иметь тип `returnDomain`.

Предикат `uncheckedConvert` выполняет предварительные вычисления входного терма `InputTerm` (если он является выражением), изменяет текущий тип в тип `returnDomain` и унифицирует с термом `ReturnTerm`. Предикат `uncheckedConvert` не делает контроль во время выполнения программы. Он только контролирует равенство битовых размеров конвертируемых доменов во время компиляции. Почти любой терм может быть конвертирован неконтролируемо в любой другой терм. Неудовлетворительный результат может случиться, если вы попытаетесь использовать переменные, неправильно преобразованные предикатом `uncheckedConvert`.

Будьте чрезвычайно внимательны и осторожны, прибегая к предикату `uncheckedConvert`, — мы настоятельно рекомендуем вам всегда, когда это возможно, использовать предикаты `convert/2->` и `tryConvert/2->`. Но заметим, что, когда объект возвращается СОМ-системой, необходимо конвертировать его предикатом `uncheckedConvert`, т. к. Пролог-программа не имеет информации о действительном типе этого объекта. Шаблон вызова функции:

```
ReturnTerm = uncheckedConvert(returnDomain, InputTerm)
```

7.6.7. Предикаты обработки эллипсиса

□ `fromEllipsis: (...) -> any* AnyTermList` — этот предикат создает список термов универсального типа `any` из `EllipsisBlock`, который обозначается многоточи-

ем ... и содержит переменное число различных параметров. Шаблон для вызова функции:

```
AnyTermList = fromEllipsis(EllipsisBlock)
```

- `toEllipsis: (any* AnyTermList) -> ...` — этот предикат создает `EllipsisBlock`, указанный многоточием ... (т. е. блок, содержащий переменное число различных параметров) из списка термов универсального типа `any`. Такой блок `EllipsisBlock` в качестве аргумента может быть позже передан в предикат, который ожидает переменное число аргументов в позиции эллипсиса (...), — например, в предикат `write/...`. Шаблон вызова функции:

```
EllipsisBlock = toEllipsis(any_term_list)
```

7.6.8. Предикаты получения размера домена

- `digitsOf: (RealDomain) -> unsigned` — возвращает точность указанного вещественного домена в виде числа значащих десятичных разрядов. Входной параметр `RealDomain` — вещественный домен, он должен быть точно указан во время компиляции (т. е. `RealDomain` не может являться значением переменной). Шаблон вызова функции:

```
Precision = digitsof(domainName)
```

- `maxDigits: (RealDomain) -> unsigned MaxDigits` — возвращает точность основного домена, соответствующего указанному вещественному домену `domainName`. Точность домена — максимальное число разрядов `MaxDigits` для параметра `RealDomain`, который должен быть именем вещественного домена `real`. Шаблон вызова функции:

```
MaxDigitsNumber = maxdigits(domainName)
```

- `lowerBound: (NumericDomain) -> LowerBound` — возвращает нижнюю границу указанного числового домена `NumericDomain`. Предикат `lowerBound` выполняется во время компиляции, а не во время выполнения программы. Предикат `lowerBound` возвращает значение нижней границы диапазона `LowerBoundValue` указанного числового домена `NumericDomain`. Возвращаемое значение `LowerBoundValue` принадлежит этому же числовому домену `NumericDomain`. Параметр `NumericDomain` должен быть именем любого числового домена — это имя должно быть явно указано во время компиляции (т. е. `NumericDomain` не может получать значение от переменной). Шаблон для вызова функции:

```
LowerBoundValue = lowerBound(domainName)
```

Если `NumericDomain` не является числовым, то возникнет ошибка во время компиляции.

- `upperBound: (NumericDomain) -> UpperBound` — возвращает значение верхней границы диапазона указанного числового домена. Предикат `upperBound` выполняется во время компиляции и возвращает верхнее значение границы диапазона указанного числового домена `domainName`. Возвращаемое значение `UpperBound` при-

надлежит к этому же домену `domainName`. Параметр `domainName` должен быть именем любого числового домена — это доменное имя должно быть точно указано во время компиляции (т. е. домен `domainName` не может получать значение от переменной). Шаблон вызова этой функции:

```
UpperBound = upperBound(domainName)
```

Если `domainName` не является числовым, то возникнет ошибка во время компиляции.

- `sizeBitsOf:(DomainName) -> unsigned BitSize` — возвращает количество битов, занимаемых в памяти термом указанного домена `DomainName`. Этот предикат во время компиляции получает в качестве входного параметра домен `DomainName` и возвращает размер памяти, занимаемый термом этого домена. Результат измеряется в битах. Для целочисленных доменов `sizeBitsOf/1` вернет число, которое определено для поля размера в доменном объявлении.

Следующее условие всегда истинно для целочисленных доменов `D`:

```
sizeOfDomain(D)*8 - 7 <= sizeBitsOf(D) <= sizeOfDomain(D)*8
```

Шаблон для вызова функции:

```
BitSize = sizeBitsOf(DomainName)
```

- `sizeOfDomain:(DomainName) -> integer ByteSize` — возвращает количество байтов, занимаемых в памяти указанным доменом `DomainName`. Этот предикат во время компиляции получает в качестве входного параметра домен `DomainName` и возвращает размер памяти, занимаемый термом этого домена. Результат измеряется в байтах. Возвращаемое значение `ByteSize` принадлежит целочисленному домену.

Шаблон для вызова функции:

```
ByteSize = sizeOfDomain(DomainName)
```

7.6.9. Предикат получения размера терма

- `sizeOf:(Term) -> integer ByteSize` — возвращает количество байтов, занимаемых в памяти термом `Term`. Функция `sizeOf/1` получает терм в качестве входного параметра и возвращает число байтов `ByteSize`, отводимое в памяти для хранения этого терма `Term`. Шаблон для вызова функции:

```
ByteSize = sizeOf(Term)
```

7.6.10. Предикаты объявления/проверки домена переменной

- `hasDomain:(Type,Variable) procedure`. Предикат `hasDomain` — это объявление того, что переменная `Variable` имеет тип `Type`. Предикат `hasDomain` выполняется во время компиляции, а не во время исполнения программы, и является указан-

ем компилятору, к какому типу отнести переменную в случае, когда тип переменной не выводится при глобальном анализе потока данных. Переменная может быть свободной, связанной или иметь смешанный поток аргументов в случае, если она является структурой. Значение и поток аргументов переменной при этом не изменяется.

- `hasDomain:(Type,Variable) -> Variable` procedure — функция `hasDomain` возвращает свой второй аргумент в том случае, если он принадлежит домену `Type`. Назначение этой функции — убедиться в том, что переменная принадлежит определенному домену. Функция полезна при проверке вводимых в программу данных. При неуспешной проверке вызывается исключение.

7.6.11. Предикат обработки ошибки

- `errorExit:(unsigned ErrorNumber) erroneous` — выводит информацию об ошибке во время исполнения программы с указанным кодом `ErrorNumber`, который может быть использован в конструкции `try-catch-finally`.

7.6.12. Предикаты управления выполнением программы

- `fail:() failure` — предикат `fail/0` завершается неуспехом, следовательно, всегда является причиной отката назад. Предложение, которое завершается неудачей, не может связать значениями выходные аргументы.
- `in` — инфиксный предикат `in/2` имеет два потока параметров: `(i,i)` и `(o,i)`. Когда оба аргумента входные, то предикат проверяет наличие первого аргумента в коллекции, заданной вторым аргументом. Коллекцией может являться список или любой недетерминированный предикат. Например, вызов `4 in [10,4,6,7]` завершится успехом. Когда первый аргумент является выходным, то предикат будет возвращать по одному все элементы коллекции, определяемой вторым аргументом. Например, вызов `X in [10,4,6,7]` вернет по очереди элементы `X=10, X=4, X=6` и `X=7`.
- `not:(SubGoal) determ` — предикат `not/1` успешен, если вычисление подцели `SubGoal` неуспешно. Заметим, что внутри `not/1` подцель `SubGoal` не может связывать переменные значениями, потому что `not(SubGoal)` только успешен, если `SubGoal` завершается неудачей (а неуспешная подцель не связывает свои аргументы). Шаблон для вызова предиката:

```
not (SubGoal)
```

- `orelse` — инфиксный предикат `orelse/2` является детерминированным аналогом дизъюнкции предикатов. Вызов `p1 orelse p2` помещает в стек адрес предиката `p2`, после чего вызывает предикат `p1`. Если `p1` успешен, то адрес предиката `p2` удаляется из стека. Если `p1` завершается неудачей, то вызывается предикат `p2`.

Таким образом, выполнение предиката `p1 orelse p2` не оставляет после себя активных адресов возврата.

- `succeed: ()` — предикат `succeed/0` всегда успешен.
- `toBoolean:(SubGoal) -> boolean Succeed`. Цель этого метапредиката — преобразовать детерминированный вызов предиката или факта в процедуру, которая возвращает значение из домена `boolean`. Результат будет `true`, если детерминированный вызов цели успешен, или `false`, если детерминированный вызов цели неудачен. Шаблон вызова метапредиката:

```
True_or_False = toBoolean(deterministic_call)
```

ГЛАВА 8



Модули

Проект в Visual Prolog представляет собой перечень классов и цель, которая в одном из классов определяет точку входа в программу. Классы в Visual Prolog могут быть двух видов:

- классы, не способные порождать объекты;
- классы, порождающие объекты.

Классы, не способные порождать объекты, называют *модулями* или статическими классами. Такие классы не имеют интерфейса. Классы, порождающие объекты, называют динамическими классами или просто — *классами*. В отличие от модулей они имеют интерфейс. В этой главе мы рассмотрим модули. Классы будут рассмотрены в *главе 19*.

Один или несколько классов могут составлять пакет. При создании и компиляции в проекте нового модуля (т. е. класса без интерфейса) с именем, например, `my_module`, в новом пакете Visual Prolog создает четыре файла:

- `my_module.pack` — описание путей для загрузки заголовочных файлов всех классов пакета и путей для загрузки файлов реализации классов пакета;
- `my_module.ph` — описание путей для загрузки заголовочных файлов открываемых классов и путей для загрузки файлов декларации классов пакета;
- `my_module.cl` — декларация модуля;
- `my_module.pro` — реализация модуля.

При добавлении нового модуля в существующий проект Visual Prolog создает только два файла:

- `my_module.cl` — декларация модуля;
- `my_module.pro` — реализация модуля.

Модули являются основным средством создания библиотек констант, пользовательских доменов и методов, которые могут использоваться другими классами. Кроме того, модули могут иметь свои свойства, доступные по чтению и/или по записи. Однако, в отличие от динамических классов, порождающих объекты, ста-

тический класс и все его сущности существуют в одном экземпляре. Поэтому свойства модуля также существуют в одном экземпляре.

8.1. Область видимости

Для вызова предиката модуля из другого класса необходимо указывать имя класса, в котором определен вызываемый предикат:

```
имя_класса :: имя_предиката(параметры).
```

Имя класса здесь играет роль области, в которой виден предикат. Разделителем между именем класса и именем предиката служит символ, составленный из двух двоеточий ::.

Обращение к константам и доменам класса из другого класса производится аналогично — с указанием области видимости, роль которой играет имя класса.

Например, класс `console` содержит предикат `setConsoleTitle(NewTitle)`, изменяющий название окна консоли. Вызов этого предиката осуществляется с указанием области (класса), в которой виден этот предикат:

```
console :: setConsoleTitle("Moe консольное приложение").
```

8.2. Структура модуля

Декларация модуля содержит:

- определение констант и доменов, видимых извне модуля;
- объявления предикатов и свойств модуля, видимых извне модуля.

Имплементация модуля содержит:

- реализацию предикатов и свойств модуля, объявленных в декларации модуля (видимых извне);
- определение констант и доменов, невидимых извне;
- декларацию и реализацию фактов, свойств и предикатов, невидимых извне.

Все свойства и предикаты, объявленные в декларации модуля, должны иметь реализацию в этом же модуле. Имплементацию модуля еще называют *реализацией модуля*.

Декларация модуля, например, `my_module`, имеет следующую структуру:

```
class my_module
open      % расширение области видимости путем указания классов
constants % раздел определения констант, видимых извне
domains   % раздел определения доменов, видимых извне
properties % раздел объявления свойств, видимых извне
predicates % раздел объявления предикатов, видимых извне
end class my_module % имя модуля — optionalno
```

Имя модуля `my_module` является строчным идентификатором. Неиспользуемые в декларации модуля разделы могут быть опущены. При закрытии декларации модуля его имя можно не указывать.

Открытие классов, указанных в директиве `open`, позволяет использовать в модуле предикаты из открытых классов без указания их области видимости, т. е. имени класса. Однако если вызываемый без указания области видимости предикат определен в нескольких открытых классах, то возникает неоднозначный вызов (англ. *ambiguous*). Для устранения такой неоднозначности при вызове следует явно указать область видимости.

В разделе `properties` объявляются предикаты для чтения и/или записи свойств модуля. Объявление свойства имеет формат:

```
имя_свойства : домен_свойства шаблон_потока.
```

Шаблон потоков параметров указывается одним из вариантов:

- (o) % чтение свойства
- (i) % запись свойства
- (o) (i) % чтение и запись свойства

Если шаблон потока опущен, то полагается, что шаблон (o) (i).

Значения свойств модуля хранятся в `single`-фактах реализации модуля. Если предполагается доступ к свойству только по чтению, то значение свойства может быть реализовано в виде функции. Если значение свойства читается из аппаратного устройства или сторонней программы, то доступ к свойству реализуется с помощью предиката (подробней об этом см. в разд. 19.1). Для обеих операций чтения и записи одного свойства используется только один предикат из раздела `properties`. Причем вызов такого предиката подобен обращению к факту-переменной.

Для чтения и записи свойств, конечно же, можно использовать предикаты отдельно для чтения и отдельно для записи, объявленные в разделе `predicates`, но тогда обращение к свойствам модуля производится в стиле вызова предикатов или функций, а не в стиле обращения к фактам-переменным.

Реализация модуля, например, `my_module`, имеет следующую структуру:

```
implement my_module
open список_классов % расширение области видимости
constants % раздел определения констант, невидимых извне
domains % раздел определения доменов, невидимых извне
class facts - имя_БД % несколько разделов БД, невидимых извне
class properties % раздел объявления свойств модуля, невидимых извне
class predicates % раздел объявления предикатов, невидимых извне
clauses % раздел правил для фактов, свойств и предикатов всего модуля
end implement my_module % optionalno
```

Неиспользуемые в реализации модуля разделы могут быть опущены. При закрытии реализации модуля его имя можно не указывать.

Разделы фактов `class facts` могут быть только в реализации модуля. Причем, нельзя объявить факты в одном классе и пытаться напрямую обращаться к ним из дру-

гого класса. Обращаться к фактам класса извне можно только опосредованно, либо через видимые свойства этого класса, либо через его видимые предикаты.

В разделе `class properties` объявляются невидимые извне предикаты для чтения и/или записи свойств модуля.

Обращение к свойствам модуля в реализации модуля можно осуществлять и без посредников, обращаясь напрямую к `single`-фактам. Однако раздел `class properties` в реализации модуля использовать допускается, и этим при необходимости можно воспользоваться.

8.3. Использование модулей в проекте

С помощью модулей в проектах Visual Prolog можно организовать библиотеки констант, доменов и предикатов, а также реализовать глобальные переменные, видимые во всем проекте.

8.3.1. Библиотека констант и доменов

Модуль может содержать только определения констант и/или доменов. На основе таких модулей в проектах Visual Prolog можно реализовать библиотеку констант и доменов, которые являются общими для некоторых классов проекта или для всего проекта в целом. Никакой функциональности такие модули не несут, но позволяют классам проекта использовать общие константы и домены. Рассмотрим пример модуля `aaa`:

```
class aaa
constants
    color = "green".
    weight = 12.5.
    def : item = n(2).
domains
    item = n(integer); y(integer).
    itemList = item*.
end class aaa
```

Модуль `aaa` содержит определение трех констант, одна из которых принадлежит пользовательскому домену `item`, а также определение двух доменов. Эти константы и домены могут использоваться в других классах — например, в классе `bbb`:

```
class bbb
predicates
    f:(aaa::item) -> integer.
    q:().
end class bbb
implement bbb
clauses
    f(aaa::n(X)) = X+1.
    f(aaa::y(X)) = X-1.
```

```

q():-write(f(aaa::def)),nl,      % 3
          write(aaa::color),nl,    % green
          write(5 + aaa::weight).  % 17.5
end implement bbb

```

8.3.2. Библиотеки предикатов

Модули, содержащие декларацию и реализацию предикатов, можно рассматривать как библиотеку предикатов для обработки данных определенного типа. Предикаты модуля зачастую используют вспомогательные предикаты, которые оформляются как предикаты реализации, т. е. невидимые вне модуля.

В следующем примере показан порядок использования невидимых извне предикатов:

```

class aaa
predicates
  min3:(integer,integer,integer)-> integer.
end class aaa
implement aaa
clauses
  min3(X,Y,Z) = Min :- M = min2(X,Y), Min = min2(M,Z).
class predicates
  min2:(integer,integer) -> integer.
clauses
  min2(X,Y) = X :- X<=Y, !.
  min2(_,Y) = Y.
end implement aaa

```

Видимый извне предикат `min3` определяет минимальное число из трех целых чисел, используя вспомогательный предикат `min2`, который невидим извне. Предикат `min2` определяет минимальное из двух целых чисел. Предикат `min3` можно вызвать из другого класса, указывая его область видимости — например: `aaa::min3(12,3,5)`. Причем вызов можно сделать как в видимом предикате стороннего класса, так и в невидимом предикате.

8.3.3. Глобальные переменные проекта

Модули могут исполнять роль хранилища глобальных переменных проекта. Для этого в декларации модуля объявляются свойства, т. е., по сути, предикаты для чтения/записи глобальных переменных. А в реализации модуля определяются начальные значения `single`-фактов, исполняющих роль глобальных переменных, и собственно правила чтения/записи `single`-фактов. Конечно же, модули могут иметь факты и с другими режимами детерминизма: `determ` и `nondeterm`. Но к таким фактам нельзя обращаться через свойства.

Рассмотрим использование модуля `aaa`, как хранилища глобальной переменной `color` и обращение к ней из класса `bbb`:

```

class aaa
properties
    color : string.
end class aaa
implement aaa
class facts
    color : string := "red".
end implement aaa

class bbb
predicates
    p : () .
end class bbb
implement bbb
clauses
    p():-write(aaa::color),nl,      % "red"
        aaa::color:="green",
        write(aaa::color).          % "green"
end implement bbb

```

Свойство `color` хранит свое значение в одноименном single-факте `color`. Начальным значением свойства `color` является строка `"red"`, которой инициализирована факт-переменная `color`.

Чтение свойства из стороннего класса может выглядеть как чтение факта-переменной:

`X = color.`

Запись свойства из стороннего класса может выглядеть как присваивание факту-переменной нового значения:

`color:="green".`

Для сравнения рассмотрим пример доступа к этому же факту-переменной посредством видимых извне предикатов:

```

class aaa
predicates
    get_color : ()-> string.
    set_color : (string).
end class aaa
implement aaa
class facts
    color: string := "red".
clauses
    get_color() = color.
    set_color(X) :- color:= X.
end implement aaa

```

```
class bbb
predicates
    p : () .
end class bbb
implement bbb
clauses
    p() :- write(aaa::get_color()), nl,      % "red"
           aaa::set_color("green"),
           write(aaa::get_color()).            % " green"
end implement bbb
```

Здесь в модуле aaa объявлены и определены два видимых извне предиката для доступа к факту-переменной color отдельно по чтению get_color/0 и отдельно по записи set_color/1. Чтение значения факта-переменной color осуществляется функцией, например:

```
x = get_color().
```

Запись нового значения факта-переменной color осуществляется вызовом предиката, например:

```
set_color("green").
```

Какой способ выбрать для доступа к фактам-переменным: либо через свойства модуля, либо через видимые извне предикаты, решает программист. Ничто не мешает использовать в проекте Visual Prolog оба способа.

ГЛАВА 9



Отсечение и отрицание

9.1. Принцип работы отсечения

Visual Prolog имеет ряд внелогических средств. Термин *внелогические средства* не означает, что эти средства являются нелогичными или ложными. Он означает, что эти средства лежат вне рамок математической логики. Самыми распространенными в логическом программировании внелогическими средствами являются отсечение и ввод/вывод.

Синтаксически предикат *отсечения* выражается восклицательным знаком `!`. Отсечения используются для повышения эффективности исполнимого кода — как по критерию уменьшения времени выполнения, так и по критерию сокращения размера используемой памяти. Отсечение удаляет из стека все адреса возврата, помещенные туда при выполнении текущего правила:

1. Адрес возврата на следующее правило выполняемого предиката, если таковое имеется.
2. Адреса возврата всех предикатов `nondeterm` и `multi`, которые вызывались в теле правила до отсечения.

Поясним первый пункт на простом примере предиката, содержащего два правила:

```
сравнить(X, Y) :- X > Y, !, write(X, " > ", Y).  
сравнить(X, Y) :- write(X, " <= ", Y).
```

Когда вызывается предикат `сравнить/2`, то вначале Visual Prolog помещает в стек адрес второго правила и только потом начинает выполнять первое правило. Если предикат `X > Y` успешен, то выполняется предикат отсечения, который удаляет из стека адрес второго правила. Благодаря этому второе правило никогда не выполнится, даже при внешнем откате на предикат `сравнить/2`. С помощью отсечений режим детерминизма какого-либо предиката можно сделать `procedure`, даже если предикат описан не одним, а несколькими правилами. При выполнении предиката отсечение некоторых правил сохраняет размер стека и сокращает время выполнения.

Для пояснения второго пункта рассмотрим пример поиска такого числа `X` в базе данных, которое больше заданного числа `Y`:

```

число(5).
число(12).
число(8).
больше(Y) :- число(X), X > Y, !, write(X, " > ", Y).
больше(Y) :- write("Нет числа, большего ", Y).

```

Когда вызывается предикат `больше(8)`, то вначале Visual Prolog помещает в стек адрес второго правила `больше(Y)` и только потом начинает выполнять первое правило. В теле первого правила вызывается факт `число(X)`, при этом Visual Prolog сначала помещает в стек адрес второго факта `число(12)` и только потом выполняет унификацию с первым фактом `число/1`, возвращая `X=5`. Далее, предикат `X > Y` со значениями `5 > 8` будет неуспешен, и Visual Prolog вытолкнет из стека последний помещенный туда адрес, а это адрес второго факта `число(12)`.

Перед выполнением второго факта `число(12)` Visual Prolog поместит в стек адрес третьего факта `число(8)`. Далее Visual Prolog выполняет унификацию со вторым фактом `число/1`, возвращая `X=12`. Предикат `X > Y` со значениями `12 > 8` наконец-то будет успешен, и Visual Prolog выполнит отсечение. При этом из стека будут удалены все адреса, помещенные туда при выполнении предиката `больше/1`:

- адрес третьего факта предиката `число(8)`;
- адрес второго правила предиката `больше/1`.

9.1.1. Область видимости отсечения

Областью видимости отсечения являются все правила предиката, в котором это отсечение используется:

```

aaa() :- pred1_nd(), bbb().
bbb() :- pred2_nd(), !.

```

Здесь предикат `aaa()` вызывает недетерминированный предикат `pred1_nd()`, который размещает в стеке адрес возврата. Затем выполняется предикат `bbb()`. Он вызывает недетерминированный предикат `pred2_nd()`, который также размещает в стеке адрес возврата. Затем в предикате `bbb()` выполняется отсечение. Областью видимости этого отсечения является предикат `bbb()`. Выполнение отсечения приводит к удалению из стека адреса правила предиката `pred2_nd()`. При этом адрес правила предиката `pred1_nd()` в стеке останется, т. к. этот предикат не входит в область видимости отсечения в предикате `bbb()`.

9.1.2. Использование отсечений

Проанализируем предикат сравнения двух чисел:

```

сравнить(X, Y) :- X > Y, write(X, " > ", Y).
сравнить(X, Y) :- X < Y, write(X, " < ", Y).
сравнить(X, Y) :- X = Y, write(X, " = ", Y).

```

Этот предикат написан правильно с декларативной точки зрения. Однако его эффективность оставляет желать лучшего. Выполним вызов: `сравнить(7, 4), fail.`

Первое правило выполнится успешно и выведет на экран сообщение: "7 > 4". А дальше начнутся непроизводительные вычисления. Откат назад выдаст из стека адрес второго правила и, поместив туда адрес третьего правила, попытается выполнить второе. Второе правило будет неуспешным. А затем будет неуспешным и третье правило.

Если присмотреться, то этот предикат имеет три взаимоисключающих условия. Если какое-либо правило успешно, то остальные правила будут априори неуспешны, и их стоило бы отсечь:

```
сравнить(X, Y) :- X > Y, !, write(X, " > ", Y).
```

```
сравнить(X, Y) :- X < Y, !, write(X, " < ", Y).
```

```
сравнить(X, Y) :- X = Y, write(X, " = ", Y).
```

Ставить отсечение в третьем правиле бесполезно. Это не будет являться ошибкой, но отсечение в третьем правиле не нужно по следующим причинам:

- отсечь правила, расположенные ниже, не получится, т. к. третье правило последнее;
- отсечь вызовы тех предикатов, которые расположены левее знака ! и помещают в стек адрес возврата, не получится, так левее расположен только предикат $X = Y$, и он ничего в стек не помещает.

Если даже отсечение в третьем правиле поставить, то оно ничего из стека не удалит.

Третье правило показывает пример явного использования унификации $X = Y$ там, где ее можно выполнить неявно, в голове правила, указав одинаковые имена аргументов: `сравнить(X, X)`. Это не является ошибкой, но лучше будет переписать предикат в виде

```
сравнить(X, Y) :- X > Y, !, write(X, " > ", Y).
```

```
сравнить(X, Y) :- X < Y, !, write(X, " < ", Y).
```

```
сравнить(X, X) :- write(X, " = ", X).
```

Может показаться, что мы усовершенствовали первоначальную версию предиката, и он стал эффективным. Это так, но можно его еще немного «подкрутить». Давайте поменяем местами второе и третье правила, оставив отсечения в первом и втором правилах:

```
сравнить(X, Y) :- X > Y, !, write(X, " > ", Y).
```

```
сравнить(X, X) :- !, write(X, " = ", X).
```

```
сравнить(X, Y) :- X < Y, write(X, " < ", Y).
```

Вроде ничего по существу не изменилось, но сейчас мы можем удалить проверку условия $X < Y$. Основанием этого является следующее рассуждение. Третье правило будет выполняться, если ни первое $X > Y$, ни второе $X = Y$ условия не будут успешны. Однако, учитывая, что все три условия взаимоисключающие, при невыполнении первых двух можно сделать вывод об успешности третьего условия $X < Y$, даже не проверяя его.

Теперь можно записать конечную версию нашего предиката:

```
сравнить(X, Y) :- X > Y, !, write(X, " > ", Y).
сравнить(X, X) :- !, write(X, " = ", X).
сравнить(X, Y) :- write(X, " < ", Y).
```

Он эффективен, т. к., во-первых, выполнение цели `сравнить(7, 4)`, `fail` не вызовет в этом предикате попыток проверки альтернативных условий, и, во-вторых, размер генерируемого кода будет минимальным.

Исходная версия этого предиката имела режим детерминизма `nondeterm`, а сейчас этот предикат стал процедурой.

Важно!

Если логика предиката предполагает единственность решения, то предикат надо писать так, чтобы он был процедурой. Это повышает эффективность логических программ в целом.

9.2. Зеленые и красные отсечения

Зеленое отсечение — это отсечение, удаление которого *не меняет* декларативный смысл предиката.

Рассмотрим предикат определения минимального из двух чисел:

```
мин(X, Y, X) :- X < Y, !.
мин(X, Y, Y) :- X >= Y.
```

Здесь первые два аргумента входные, а третий аргумент — выходной. Цель `мин(2, 5, Z)` на основании первого правила вернет $Z = 2$. А цель `мин(7, 5, Z)` на основании второго правила вернет $Z = 5$.

Используемое в первом правиле отсечение является зеленым. Если убрать это отсечение, то декларативный смысл программы останется прежним, но процедурный смысл изменится, т. к. откат назад после успешного выполнения первого правила передаст управление второму правилу, которое заведомо неуспешно.

Красное отсечение — это отсечение, удаление которого *изменяет* декларативный смысл предиката.

Изменим предыдущий пример, удалив условие во втором правиле, т. к. оба условия взаимоисключающие. Кроме того, в голове второго правила аргумент x надо сделать анонимной переменной, т. к. удалено условие $X \geq Y$, в которое эта переменная входит:

```
мин(X, Y, X) :- X < Y, !.
мин(_, Y, Y).
```

Как можно видеть, удаление явного условия во втором правиле сделало отсечение в первом правиле красным. Ибо теперь удаление отсечения искажает декларативный смысл предиката — он попросту станет неправильно работать:

```
мин(Х, Y, Х) :- Х < Y.
мин(_, Y, Y).
```

Цель `мин(2, 5, Z), fail` по первому правилу вернет $Z = 2$, а после отката вернет неправильный ответ $Z = 5$ согласно второму правилу, в котором нет явной проверки условия минимальности.

Итак, зеленые отсечения повышают скорость выполнения программ и экономят память стека адресов возврата. Красные отсечения эффективнее зеленых, поскольку дополнительно еще и уменьшают размер исполнямого файла. Однако надо быть очень аккуратными, т. к. неправильное использование красных отсечений может исказить правильность программы.

9.3. Динамическое отсечение

Visual Prolog дает возможность управлять содержимым стека адресов возврата с помощью динамического отсечения. Для этого есть две команды из класса `programControl`:

- Адрес = `getBackTrack()` — читает адрес возврата, находящийся в вершине стека, т. е. тот, который был помещен в стек последним;
- `cutBackTrack(Адрес)` — удаляет из верхушки стека все адреса возврата вплоть до заданного адреса Адрес. Этот адрес не удаляется, он становится вершиной стека.

Динамическое отсечение используется в тех случаях, когда надо получить всего одно решение там, где какой-либо предикат при откатах назад возвращает множество решений. Это относится к предикатам с режимами `nondeterm` и `multi`.

Кроме того, с помощью динамического отсечения можно расширить область видимости отсечения на другие предикаты посредством передачи им в качестве параметра текущего адреса, полученного из стека `Адрес = getBackTrack()`.

Решим следующую задачу двумя способами: с использованием отсечения и с использованием динамического отсечения. Пусть определены два множества $A = \{17, 18, 19\}$ и $B = \{2, 3, 5, 7\}$. Необходимо для каждого элемента X из множества A найти один элемент Y из множества B , такой, что $X+Y>20$.

Способ решения № 1. Воспользоваться поиском с возвратом в виде:

```
a(17). a(18). a(19).
b(2). b(3). b(5). b(7).
найти(X, Y) :- a(X), b(Y), X+Y>20, fail();
               succeed().
```

не получится, поскольку такой поиск осуществляет полный перебор всех пар (X, Y) , а по условию задачи перебор должен быть частичным, — ведь для одного элемента из множества A надо найти только один элемент из множества B . Поэтому следует модифицировать указанный поиск так, чтобы он для заданного элемента X находил только один элемент Y . Для этого введем еще один предикат с именем `один_элемент/2`, который и будет выполнять требуемое действие:

```
a(17). a(18). a(19).
b(2). b(3). b(5). b(7).
найти(X, Y) :- a(X), один_элемент(X, Y), fail();
               succeed().
один_элемент(X, Y) :- b(Y), X+Y>20, !.
```

Отсечение в предикате `один_элемент/2` действует только внутри этого предиката, поэтому он не помешает откату назад в теле предиката `найти/2`. Запрос `найти(X, Y), fail()` найдет три решения: $(17, 5)$, $(18, 3)$ и $(19, 2)$.

Способ решения № 2. Воспользуемся динамическим отсечением:

```
a(17). a(18). a(19).
b(2). b(3). b(5). b(7).
найти(X, Y) :- a(X), Addr = getBackTrack(),
               b(Y), X+Y>20, cutBackTrack(Addr), fail();
               succeed().
```

Вызов `a(X)` вернет нам $X=17$. Перед вызовом предиката `b(Y)` в переменную `Addr` помещается адрес возврата на факт `a(18)`. После нахождения элемента $Y=5$, удовлетворяющего условию $X+Y>20$, в стеке будет располагаться еще один адрес — адрес возврата на факт `b(7)`. Однако этот адрес нам совершенно не нужен, т. к. для числа 17 мы уже нашли ему в пару число 5 . Нам нужно перепрыгнуть через предикат `b(Y)` сразу на предикат `a(X)`, чтобы взять новое число 18 из множества A . Для этого мы вызываем `cutBackTrack(Addr)`, который удалит из стека адрес факта `b(7)`, после чего в стеке останется адрес факта `a(18)`. Таким образом, для каждого элемента из множества A будет найдено не более одного элемента из множества B .

Использование динамического отсечения освободило нас от написания дополнительного предиката `один_элемент/2`, который был необходим при первом способе решения.

9.4. Отрицание

Предикат `not/1` является одноарным предикатом *отрицания*. Аргументом этого предиката может быть только предикат, но никак не переменная. Выполнение предиката `not(p(X))` начинается с выполнения предиката `p(X)`. Если предикат `p(X)` успешен, то предикат `not(p(X))` неуспешен и вызывает откат назад. Если предикат `p(X)` неуспешен, то предикат `not(p(X))` успешен.

Не следует путать операцию отрицания значения предиката с логической операцией отрицания переменной булева типа `boolean::logicalNot(Boolean)`. Отрицание предиката, например, `not(p(X))` меняет значение истинности предиката `p(X)` в области {«успех», «неуспех»} на противоположное, и новое значение истинности может «увидеть» только механизм логического вывода, который запустит откат назад, если предикат `not(p(X))` неуспешен. С другой стороны, логическое отрицание булевой переменной `logicalNot(Boolean)` меняет значение этой переменной на множестве {«истина», «ложь»}. И программист может получить новое значение булевой переменной и использовать его в дальнейших вычислениях.

Предикат `not(p(X))` не связывает свободные переменные. Это объясняется тем, что в композиции предикатов `not(p(X))` либо `not`, либо `p(X)` будет неуспешен. А неуспешный предикат не связывает переменные.

Предикат `not(p(X))` не помещает в стек адрес возврата. Если аргумент предиката `not(p(X))` имеет режим `multi` или `nondeterm`, то предикат `not` найдет все варианты решения своего аргумента — предиката `p(X)`, и только когда предикат `p(X)` завершится неуспехом, предикат `not(p(X))` станет успешным. Таким образом, предикат `not` выполняет цикл, перебирая все решения своего аргумента.

Рассмотрим следующий пример:

```
z() :- write("1"); write("2"); write("3").
x() :- write(" A "); write(" B "); write(" C ").
p() :- x(), not(z()); nl, write("end").
```

Здесь предикаты `z()` и `x()` имеют режим `multi`. Запрос `p()` найдет первое решение для `x()`, потом предикат `not(z())` осуществит перебор всех успешных вариантов предиката `z()` и откатится назад, т. к. будет неуспешен. После этого предикат `x()` найдет новое решение, и все повторится сначала. После перебора всех успешных вариантов решения предиката `x()` выполнится второе правило запроса `p()` — вывод строки `end`. На экране мы увидим такой результат:

```
A 123 B 123 C 123
```

```
end
```

В некоторых случаях хотелось бы использовать выражение над предикатами в качестве аргумента предиката `not`. Например, конъюнкцию `x(), z()` ввести под отрицание: `not(x(), z())`. Однако именно так делать нельзя — это является синтаксической ошибкой. Предикат `not` одноарный, а мы указали два аргумента, поскольку конъюнкцию `x(), z()` Visual Prolog будет трактовать как два отдельных аргумента. В данном случае конъюнкцию `x(), z()` надо описать одним предикатом. Проще всего этого сделать, взяв конъюнкцию в скобки: `(x(), z())`. Итого имеем: `not((x(), z()))`. Здесь наружные скобки принадлежат собственно предикату `not`, а внутренние скобки делают из выражения `x(), z()` предикат `(x(), z())`.

В ранее рассмотренном примере предикат `p()` можно определить по-другому — включив вызов `x()` внутрь отрицания `not`:

```
z() :- write("1"); write("2"); write("3").
x() :- write(" A "); write(" B "); write(" C ").
p() :- not((x(), z())); nl, write("end").
```

Результат запроса `p()` останется прежним:

```
A 123 B 123 C 123
```

```
end
```

ГЛАВА 10



Циклы с откатом

Циклы с откатом широко используются в Прологе при работе с фактами внутренних баз данных и с предикатами, имеющими режим детерминизма `multi` или `nondeterm`. С помощью циклов с откатом можно организовать бесконечные циклы, циклы с заданным числом повторений, циклы с выходом по условию.

10.1. Структура цикла с откатом

В общем виде цикл с откатом представляется предикатом:

```
p() :- std::repeat(), тело_цикла(), fail().
```

Здесь тело цикла заключено между предикатом `std::repeat()` и предикатом `fail()`. Тело цикла может описываться выражением над предикатами. При вызове предикатов `repeat/0` и `fail/0` пустые скобки можно не указывать.

Предикат `repeat/0` из класса `std` всегда успешен при откате на него. Он будет успешен бесконечное число раз — столько, сколько раз на него будет совершен откат. Определение предиката `repeat` в классе `std`:

```
repeat().  
repeat() :- repeat().
```

При откате на него этот предикат рекурсивно вызывает сам себя, т. е. при откате голова в правиле `repeat() :- repeat()` вызывает тело `repeat()`, которое будет успешным. Этот предикат определяет начало цикла.

Предикат `fail/0` всегда вызывает откат, т. к. является тождественно ложным предикатом. Этот предикат определяет конец цикла. Тело цикла выполняется от предиката `repeat` до предиката `fail`. После чего откатывается назад, к активному адресу возврата, с которого вычисления продолжаются.

Особенностью описанного цикла является бесконечное число повторов тела цикла и режим детерминизма цикла `failure`. Конечно, легко сделать режим детерминизма `procedure`, но цикл все равно останется бесконечным, и не будет иметь выхода:

```
p() :- std::repeat(), тело_цикла(), fail().
p().
```

Предикаты `repeat` и `fail`, словно теннисные ракетки, будут гонять тело цикла как мячик, — от предиката `repeat` слева направо путем прямого вызова предикатов тела цикла и справа налево посредством отката от `fail` на те предикаты цикла, которые имеют активные адреса возврата, вплоть до предиката `repeat`.

Если тело цикла имеет режим `procedure` или `determ`, то оно будет выполняться при вызове, после предиката `repeat`. В этом случае откат назад, вызванный `fail`, перепрыгнет на `repeat` и начнет новый виток цикла очередным вызовом его тела. Например, такая конструкция будет бесконечное число раз выводить на экран строку ABC:

```
p() :- std::repeat, write("ABC"), fail.
```

Если тело цикла имеет режим `multi` или `nondeterm`, то первый виток будет осуществлен вызовом тела цикла после предиката `repeat`. Остальные витки станут совершаться при откатах. Так будет происходить до тех пор, пока тело цикла не вернет все варианты решений. После этого откат вернется к предикату `repeat`, и Пролог начнет повторение тела цикла заново. Например, такой цикл будет бесконечное число раз выводить по очереди на экран элементы множества $A = \{3, 5, 8\}$:

```
a(3). a(5). a(8).
p() :- std::repeat, a(X), write(X), fail.
```

Какой-либо смысл в рассмотренных бесконечных циклах искать не надо. Главное — усвоить, что показанная структура бесконечного цикла является основой построения цикла, имеющего полезное прикладное поведение. Этим мы сейчас и займемся.

В большинстве случаев циклы должны иметь хотя бы один выход. В циклах с откатом выход может располагаться в начале тела цикла или в его конце. При необходимости можно использовать два выхода из цикла: один в начале, другой в конце.

Для того чтобы сделать выход из цикла в начале его тела, надо использовать вместо предиката `repeat` какое-либо условие продолжения цикла. Пока условие истинно, цикл будет продолжаться. Когда условие станет ложным, то при откате произойдет выход из цикла и будет вызвано второе предложение предиката `p/0`. Такая конструкция обеспечит успешность предиката `p/0` после выхода из цикла:

```
p() :- условие_продолжения(), тело(), fail().
p().
```

Для того чтобы сделать выход из цикла в конце его тела, надо использовать вместо предиката `fail` какое-либо условие завершения цикла. Пока это условие ложно, цикл будет продолжаться. Когда условие станет истинным, произойдет выход из цикла. Обратите внимание, что в этом случае второе предложение предиката `p/0` не обязательно:

```
p() :- std::repeat, тело(), условие_завершения().
```

Точка выхода из цикла зависит от прикладной задачи. При необходимости легко оформить два выхода из цикла, использовав и условие продолжения цикла, и условие его завершения:

```
p() :- условие_продолжения(), тело(), условие_завершения().
```

В Прологе циклы не разделяются на циклы с заданным числом повторений и циклы с предусловием и постусловием, как это делается в императивных языках программирования. Пролог не имеет предопределенных синтаксических конструкций для каждого из них, да и вообще четкая граница между ними в Прологе отсутствует.

Мы рассмотрели циклы с условиями продолжения и завершения. Для реализации циклов с заданным числом повторений надо использовать счетчик. Роль счетчика в Прологе играет факт-переменная. В теле цикла значение факта-переменной надо инкрементировать. Для выхода из цикла достаточно сравнить значение этого счетчика с заданным числом повторений. Если обозначить счетчик фактом-переменной *i*, а число повторений передать циклу в качестве параметра *N*, то цикл с заданным числом повторений можно записать так:

```
p(N) :- i:=0, std::repeat, тело(), i:=i+1, i>=N.
```

Здесь условием выхода из цикла является предикат *i>=N*.

В классе *std* реализованы различные недетерминированные итераторы для циклов с откатом. Их использование упрощает написание таких циклов, поскольку надобность в фактах-переменных отпадает. Например, цикл с заданным числом повторов, рассмотренный ранее, проще организовать с использованием недетерминированной функции *I=fromTo(From, To)*, которая на откатах возвращает новое значение счетчика путем инкремента текущего значения:

```
p(N) :- I=std::fromTo(1,N), тело(), fail.  
p(_).
```

10.2. Реализация циклов с откатом

ПРИМЕР 10.1. Для вывода на экран квадратов первых *N* натуральных чисел можно воспользоваться предикатом *p/1*, аргументом которого является задаваемое число повторений:

```
implement main  
    open core, console  
class facts - aaa  
    i:unsigned := 1.                      % объявление и инициализация счетчика  
class predicates  
    p:(unsigned) nondeterm.  
clauses  
    p(N) :- std::repeat,                 % начало цикла  
            write(i^2), nl,                % вывод квадрата, переход на новую  
            !, i:=i+1.                   % строку
```

```

i:=i+1,                      % инкремент счетчика
i>N.                         % условие завершения цикла
run():- (p(4), !; succeed),   % делаем run() процедурой
                               _ = readline().
end implement main
goal
  console::run(main::run).

```

В этом примере вызов `(p(4), !; succeed)` можно заменить вызовом `(p(4) orelse succeed)` с использованием встроенного предиката `orelse`.

ПРИМЕР 10.2. Рассмотрим подсчет количества фактов в БД. Пусть необходимо подсчитать количество экземпляров недетерминированного факта `a(integer)`. Для этого напишем предикат с одним выходным аргументом, который унифицируем со значением счетчика `i` на выходе из цикла, т. е. когда все факты будут подсчитаны:

```

implement main
  open core, console
class facts - aaa
  i:unsigned := 0.          % объявление и инициализация счетчика
  a:(integer).             % объявление недетерминированного факта
class predicates
  p:(unsigned) procedure (o).
clauses
  a(3). a(5). a(8). a(7). a(1). a(2). % определение фактов
  p(i) :- a(_),                  % очередной факт
          i:=i+1,                 % инкремент счетчика
          fail;                   % принудительный откат назад
          succeed.               % если очередного факта нет, то завершение
                               % успехом
  run():-p(N),                % вызов предиката подсчета фактов a/1
                               write(N),
                               _=readline().
end implement main
goal
  console::run(main::run).

```

Результатом работы программы будет число 6.

ПРИМЕР 10.3. Существуют задачи, связанные с поиском решений, удовлетворяющих только одному условию из заданной дизъюнкции условий. Рассмотрим одну из таких задач. Пусть фактами `dot(integer X, integer Y)` задано множество точек на плоскости. Требуется найти все те точки, которые лежат выше горизонтальной прямой $Y=3$ и правее вертикальной прямой $X=7$.

```

implement main
  open core, console
class facts - aaa
  dot:(integer X, integer Y). % объявление факта с координатами точек

```

```

clauses
    dot(32,12). dot(78,-5). dot(9,54). % определение трех точек
    run():-dot(X,Y), % вызов очередной точки
        (X>7; Y>3), % проверка условия
        write(X," ",Y),nl, % вывод координат
        fail; % откат на следующую точку
        _=readline().
end implement main
goal
    console::run(main)::run).

```

После запуска можно увидеть, что программа работает, но некоторые точки выводит дважды:

```

32 12
32 12
78 -5
9 54
9 54

```

Выводятся повторно только те точки, координаты которых удовлетворяют условию. Точка `dot(32,12)` выводится первый раз благодаря успеху предиката `X>7`, а повторно — благодаря успешности предиката `Y>3`.

Как нам подавить повторные выводы? Нужно каким-то образом запретить проверку координаты `Y`, когда координата `X` удовлетворяет условию.

Если мы используем отсечение после проверки координаты `X` в дизъюнкции `(X>7, !; Y>3)`, то при успехе предиката `X>7` это отсечение удалит из стека адрес возврата на предикат `dot(X,Y)`, из-за чего перебор точек в цикле будет невозможен. Конечно, можно вынести проверку условия в отдельный предикат, и тогда отсечение в этом новом предикате не повлияет на перебор всех точек в цикле. Но новый предикат нужно объявить и описать.

Visual Prolog для этого случая предлагает другое решение — детерминированную дизъюнкцию. Синтаксически она выражается встроенным предикатом `(X>7 orelse Y>3)`. Работа этого предиката описана в разд. 7.6. Суть работы предиката заключается в том, что после его выполнения в стеке не остается активных адресов возврата на этот предикат. То есть, при истинности `X>7` в стеке не останется адреса на предикат `Y>3`. Используя эту детерминированную дизъюнкцию, можно записать окончательный вариант решения задачи:

```

implement main
    open core, console
class facts - aaa
    dot:(integer X,integer Y). % объявление факта с координатами точек
clauses
    dot(32,12). dot(78,-5). dot(9,54). % определение трех точек
    run():-dot(X,Y), % вызов очередной точки
        (X>7 orelse Y>3), % проверка условия
        write(X," ",Y),nl,
        fail;
        _=readline().

```

```

        write(X, " ", Y), nl,           % вывод координат
        fail;                          % откат на следующую точку
        _=readline().
end implement main
goal
    console::run(main:::run).

```

При запуске видно, что повторные решения отсутствуют.

ПРИМЕР 10.4. В задачах на поиск экстремального значения также можно использовать цикл с откатом. Пусть необходимо найти треугольник с максимальным периметром. Треугольник можно образовать, используя точки с заданными координатами, рассматриваемые в качестве вершин треугольника.

```

implement main
    open core, console
class facts - aaa
    s:real := 0.                      % объявление периметра и его
                                         % инициализация
    dot:(integer X,integer Y).        % объявление факта с координатами точек
class predicates
    p:(real) procedure (o).
clauses
    dot(32,12). dot(10,-17). dot(78,-5).
    dot(9,54). dot(-2,33). dot(28,72).      % определение шести точек
    p(_):-dot(X1,Y1),dot(X2,Y2),dot(X3,Y3), % вызов трех точек из БД
          0<>X1*(Y2-Y3)+X2*(Y3-Y1)+X3*(Y1-Y2), % условие "треугольности"
          D1=math::sqrt((X1-X2)^2+(Y1-Y2)^2),   % длина первой стороны
          D2=math::sqrt((X2-X3)^2+(Y2-Y3)^2),   % длина второй стороны
          D3=math::sqrt((X3-X1)^2+(Y3-Y1)^2),   % длина третьей стороны
          D = D1+D2+D3,                         % получение периметра
          D > s,                                % полученный периметр больше наибольшего текущего
          s := D,                                % обновление наибольшего текущего периметра
          fail.                                 % откат для поиска других треугольников
    p(s).        % перебор закончен, возвращение наибольшего периметра
run():-p(N),
      write(N),
      _=readline().
end implement main
goal
    console::run(main:::run).

```

В этом примере условие «треугольности» отсеивает все треугольники, у которых хотя бы две вершины совпадают или все три вершины лежат на одной прямой. Факт-переменная `s` хранит текущий максимальный периметр треугольника. При нахождении треугольника, периметр которого больше `s`, факт-переменная обновляется значением этого периметра. Когда перебор всех треугольников будет завершен, выходной аргумент предиката `p/1` будет связан значением факта-переменной `s`.

Для того чтобы получить не только максимальный периметр, но и координаты вершин треугольника, можно, к примеру, объявить еще один факт и в нем сохранять координаты вершин треугольника тогда, когда обновляется текущий максимальный периметр.

ПРИМЕР 10.5. Логическая задача. За круглым столом Артура сидят 12 рыцарей. Из них каждый враждует со своим соседом, сидящим слева и справа от него. Надо выбрать 5 рыцарей, чтобы освободить заколдованную принцессу. Необходимо найти все способы формирования отряда из пяти рыцарей так, чтобы среди выбранных рыцарей не было врагов.

Для решения этой задачи перенумеруем всех рыцарей и отобразим каждого из них в программе в виде факта с номером рыцаря `p(unsigned)`. Для того чтобы не было повторов выбранных команд рыцарей, наложим такое условие, что номер следующего кандидата в команду должен быть больше номера предыдущего выбранного рыцаря. Выразим условие дружбы двух рыцарей через отношения их номеров. Но мера друзей должны отличаться более чем на единицу. Для рыцарей с номерами 1 и 12, которые являются врагами, потому что за круглым столом они сидят рядом, выделим дополнительное условие, согласно которому разность между номерами друзей должна быть меньше 11.

```

implement main
    open core, console
class facts - aaa
    p: (unsigned).           % объявление факта с номерами рыцарей
class predicates
    набрать:().
    друг: (unsigned,unsigned) determ.
clauses
    p(1). p(2). p(3). p(4). p(5). p(6).      % определение 12-ти рыцарей
    p(7). p(8). p(9). p(10). p(11). p(12).
    набрать () :-
        p(A), p(B), друг(A,B),                  % выбор первых двух рыцарей
        p(B), друг(A,B), друг(B,B),              % выбор третьего рыцаря
        p(C), друг(A,C), друг(B,C), друг(C,B),  % выбор четвертого
                                                % рыцаря
        p(D), друг(A,D), друг(B,D), друг(C,D), друг(G,D),
        writef("% % % % \n",A,B,C,D,G),          % вывод отряда на экран
        fail;                                     % откат для поиска другого отряда
        write("Поиск завершен").
    друг(A,B):-A<B, B-A>1, B-A<11.       % условия дружбы и уникальности
                                                % отряда
    run():-набрать(),
            _ = readline().
end implement main
goal
    console::run(main:::run).

```

Суть решения заключается в том, что при выборе очередного кандидата программа проверяет его дружественность к уже выбранным членам команды. Когда отряд из пяти рыцарей набран, номера рыцарей выводятся на экран, и вызывается принудительный откат для поиска следующего варианта решения. В этой задаче получается 36 вариантов решений.

Такой способ решения логических задач применяется, как правило, для ограниченного объема исходных данных, поскольку проверки дружбы каждого с каждым занимают много вызовов предиката `друг/2`. Для произвольного объема исходных данных обычно используются списки и рекурсия, которые будут рассмотрены в главах 11 и 13.

10.3. Использование изменяемых переменных в циклах с откатом

В Visual Prolog есть пакет `varM`, содержащий четыре класса для создания и работы с изменяемыми переменными. Три класса, `varM_boolean`, `varM_integer` и `varM_unsigned`, имеют дело с переменными типа `boolean`, `integer` и `unsigned` соответственно. А один класс, одноименный с пакетом `varM`, позволяет обрабатывать переменные произвольного типа.

Изменяемые переменные отличаются от переменных Пролога тем, что могут менять свое значение только посредством разрушающего присваивания, поскольку они реализованы фактами-переменными.

Рассмотрим класс `varM_integer`. Для создания и инициализации изменяемой переменной надо воспользоваться одним из двух конструкторов:

```
X = varM_integer::new() % создание неинициализированной переменной X
Y = varM_integer::new(1) % создание переменной Y со значением 1
```

В пакете `varM` предусмотрено свойство `value`, которое возвращает значение переменной. Изменять значение переменной можно двумя способами. Один способ заключается в разрушающем присваивании некоторого значения свойству `value`:

```
Y = varM_integer::new(1),      % создание переменной Y со значением 1
Y:value := Y:value + 1,       % инкремент переменной
Y:value := 10,                % присваивание непосредственного значения
```

Другой способ состоит в использовании предикатов класса `varM_integer`:

```
Y:add(5),        % складывает число 5 с текущим значением переменной Y
Y:sub(5),        % вычитает число 5 из текущего значения переменной Y
Y:max(5),        % присваивает переменной максимальное значение среди
                 % числа 5 и текущего значения переменной
Y:min(5),        % присваивает переменной минимальное значение
                 % среди числа 5 и текущего значения переменной
```

Если в цикле надо создать аккумулятор для данных такого типа, который не является типом `boolean`, `integer` или `unsigned`, то необходимо использовать класс `varM`,

который поддерживает произвольные типы данных. Для создания и инициализации изменяемой переменной произвольного типа надо воспользоваться конструктором, в котором указать тот тип данных, к которому должна принадлежать переменная. Например, для создания изменяемой переменной типа `real` можно воспользоваться конструктором:

```
X = varM{real}::new(1.7) % создание переменной X со значением 1.7
```

Однако в большинстве случаев Visual Prolog во время компиляции сам определяет тип данных, и объявлять его явно не обязательно:

```
X = varM::new(1.7) % создание переменной X со значением 1.7
```

Изменять значение переменной класса `varM` можно только одним способом — разрушающим присваиванием свойству `value` некоторого значения:

```
Y = varM::new(1.75), % создание переменной Y со значением 1.75
```

```
Y:value := Y:value + 0.92, % сложение с числом 0.92
```

```
Y:value := 18.03, % присваивание непосредственного значения
```

С помощью изменяемых переменных можно организовывать счетчики и какие-либо аккумуляторы. Основной способ применения изменяемых переменных состоит в том, что перед началом цикла создается и инициализируется некоторая переменная, а в теле цикла с ней производятся определенные действия. Например, счетчик цикла можно организовать так:

```
p() :- I = varM_unsigned::new(0),
        условие_продолжения(),
        тело(),
        I:add(1),
        условие_завершения().
```

Достоинства и недостатки изменяемых переменных и фактов-переменных представлены в табл. 10.1.

Таблица 10.1. Достоинства и недостатки изменяемых переменных и фактов-переменных

Изменяемая переменная	Факт-переменная
Не требует явного объявления	Требует явного объявления
После использования переменной сборщик мусора освободит занимаемую ею память	Факт-переменная остается в памяти до завершения программы
Область видимости — правило	Область видимости — класс
Для расширения области видимости на весь класс ссылку на изменяемую переменную надо запомнить в явно объявлена факте	Расширять область видимости не требуется

Для сравнения решим задачи из примеров 10.1 и 10.2 с использованием изменяемой переменной.

ПРИМЕР 10.6. Итак, выведем на экран квадраты первых N натуральных чисел:

```

implement main
    open core, console
class predicates
    p:(unsigned) nondeterm.
clauses
    p(N) :- I = varM_unsigned::new(1),      % создаем счетчик
            std::repeat,                      % начало цикла
            write(I:value^2), nl,              % вывод квадрата
            I:add(1),                        % инкремент счетчика
            I:value > N.                   % условие завершения цикла
run():- (p(4), !; succeed),
       _ = readline().
end implement main
goal
    console::run(main:::run).

```

ПРИМЕР 10.7. Подсчитаем количество фактов a/1 в БД:

```

implement main
    open core, console
class facts - aaa
    a:(integer).           % объявление недетерминированного факта
class predicates
    p:(unsigned) procedure (o).
clauses
    a(3). a(5). a(8). a(7). a(1). a(2).      % определяем факты
    p(N) :- I = varM_unsigned::new(0),          % создаем счетчик
            (a(_),                         % проверяем наличие очередного факта
             I:add(1),                      % инкремент счетчика
             fail;                          % принудительный откат назад
             N = I:value).                 % если очередного факта нет, то
                                              % возвращаем ответ
run():- p(N),           % вызов предиката подсчета фактов a/1
       write(N),
       _=readline().
end implement main
goal
    console::run(main:::run).

```

ПРИМЕР 10.8. Подсчитаем не только количество фактов в БД, но и сумму чисел, хранимых в этих фактах:

```

implement main
    open core, console
class facts - aaa
    a:(integer).           % объявление недетерминированного факта
class predicates
    p:(unsigned I, integer Sum) procedure (o,o).

```

```

clauses
    a(3). a(5). a(8). a(7). a(1). a(2).      % определение фактов
    p(N,Sum) :- I = varM_unsigned::new(0),      % создаем счетчик
    S = varM_integer::new(0),                    % создаем накопитель
    (a(V),           % очередной факт
     I:add(1),       % инкремент счетчика
     S:add(V),       % накапливаем сумму
     fail;          % принудительный откат назад
     N = I:value,    % если очередного факта нет, то
     % возвращаем счетчик
     Sum = S:value). % и сумму
run() :- p(N,Sum),
        write(N, " ", Sum),
        _=readline().
end implement main
goal
    console::run(main:::run).

```

Результатом работы программы будет число фактов, равное 6, и сумма, равная 26.

10.4. Циклы с откатом на основе отрицания

Циклы с откатом могут быть реализованы на основе предиката отрицания цели. Если предикат `not(gg())` в качестве своего аргумента содержит недетерминированный предикат `gg()`, то Пролог вначале найдет первое решение предиката `gg()`. Однако предикат `not(gg())` примет значение «неуспешно», что вызовет откат, и Пролог найдет второе решение предиката `gg()`. И так будет продолжаться до тех пор, пока все решения не будут найдены. После чего предикат `gg()` станет неуспешным, а его отрицание `not(gg())` примет значение «успешно», и цикл завершится. Таким образом, предикат `not/1`, по сути, выполняет поиск всех успешных решений своей цели `gg()`. Такое поведение аналогично циклу с откатом.

ПРИМЕР 10.9. Пусть имеется динамическая БД персонала на основе фактов `человек(фамилия, рост, вес)`. Необходимо найти фамилию самого высокого человека. В следующей программе в предикате `высокий/1` ищется такой человек `человек(Ф, Рост, _)`, выше которого никого нет:

```
not((человек(_, Рост1, _), Рост1 > Рост)).
```

Это отрицание завершится успехом в том случае, когда не будет найдено ни одного человека, рост которого `Рост1` будет выше значения `Рост`.

```

implement main
    open core, console
domains
    рост = digits 4 [30.0...]. % в сантиметрах
    вес = digits 4 [1.0...].    % в килограммах
    фамилия = string.

```

```
class facts - люди
    человек: (фамилия, рост, вес) .
class predicates
    высокий: (фамилия) determ (o) .
clauses
    человек("Воронин", 175.5, 78.9) .
    человек("Селиванов", 165.1, 73.4) .
    человек("Ершов", 181.9, 88.2) .
    человек("Панин", 185.3, 92.5) .
    человек("Агеев", 179.5, 84.9) .
высокий(Ф) :- человек(Ф, Рост, _), % выбираем очередного человека из БД
    not( человек(_, Рост1, _), % выбираем человека из БД
        Рост1 > Рост ) , ! . % проверяем, есть ли более высокий
run() :- (высокий(Ф), write(Ф), nl, !;
           succeed), % делаем run() процедурой
           _ = readLine() .
end implement main
goal
    console::run(main::run) .
```

Конечно эффективность такой программы низкая, поскольку максимальный рост ищется с квадратичной вычислительной сложностью относительно числа фактов в БД. Хотя он мог бы быть найден с линейной сложностью. Этот способ организации цикла следует иметь в виду для случая, когда количество альтернатив подцели предиката `not/1` невелико, или когда наилучший алгоритм решения имеет ту самую квадратичную сложность.

ГЛАВА 11



Рекурсия

11.1. Структура рекурсии

Рекурсия является вторым и последним способом организации циклов в Прологе. В самом простом случае рекурсивный предикат описывается набором правил, в котором хотя бы одно правило должно быть рекурсивным и хотя бы одно — нерекурсивным. Нерекурсивное правило играет роль останова рекурсии. Здесь стоит заметить, что рекурсия может и не иметь правила останова, если смысл программы заключается именно в бесконечном цикле. *Рекурсивное правило* — это правило, в теле которого хотя бы один раз осуществляется вызов головы правила. Если не рассматривать аргументы рекурсии, то структуру рекурсивного правила можно выразить предикатом

```
p() :- тело_цикла(), p().
```

Рекурсия с условием завершения содержит как минимум два правила. Первое правило играет роль останова рекурсии, а второе правило представляет собой собственно тело цикла:

```
p() :- условие_завершения(), !.  
p() :- тело_цикла(), p().
```

Здесь тело цикла повторяется до тех пор, пока условие завершения ложно. Как только условие завершения станет истинным, цикл завершается и, благодаря отсечению, из стека удаляется адрес возврата на рекурсивное правило.

Рекурсия с условием продолжения также содержит как минимум два правила. Первое правило содержит тело цикла, а второе правило играет роль останова рекурсии:

```
p() :- условие_продолжения(), тело_цикла(), !, p().  
p().
```

Здесь тело цикла повторяется до тех пор, пока условие продолжения истинно. При каждом повторе выполняется отсечение, благодаря чему из стека удаляется адрес возврата на второе правило. Поэтому размер используемого стека не увеличивается. Как только условие продолжения станет ложным, выполняется второе правило, и цикл завершается.

Рекурсия может содержать несколько рекурсивных правил, каждое из которых имеет собственное тело цикла, а также несколько правил останова цикла:

```
p() :- условие_завершения1(), !.
p() :- условие_продолжения1(), тело1(), !, p().
p() :- условие_завершения2(), !.
p() :- условие_продолжения2(), тело2(), !, p().
p().
```

Рассмотренные рекурсии являются *оптимизированными*. Это означает, что при выполнении рекурсии стек адресов возврата не растет, т. к. выполняются два признака оптимизированной рекурсии:

- все правила, кроме последнего, содержат отсечения;
- рекурсивный вызов является последним в каждом рекурсивном правиле.

Здесь второй пункт иногда называют *хвостовой рекурсией*, т. к. рекурсивный вызов в правиле последний — хвостовой. Оптимизированные рекурсии эффективны по критерию минимального размера используемой памяти и времени выполнения. Они подобны итеративным циклам в императивных языках программирования.

Неоптимизированные рекурсии не имеют хотя бы одного из перечисленных признаков оптимизированной рекурсии. Неоптимизированным рекурсиям свойственны побочные эффекты — некоторые из них полезны, а некоторые — нет. Рассмотрим вначале рекурсии, которые не используют отсечения или используют, но не всюду, где этого требует оптимизированная рекурсия.

Неоптимизированная рекурсия с условием завершения имеет вид:

```
p() :- условие_завершения().
p() :- тело_цикла(), p().
```

При выполнении условия завершения цикл заканчивается, но в стеке остается адрес возврата на второе правило рекурсии. Внешний откат назад на эту рекурсию возобновит ее работу, и тело цикла выполнится еще как минимум один раз. Так можно делать бесконечные циклы. Например, так сделан предикат `repeat` и некоторые недетерминированные функции класса `std`, выполняющие роль итераторов в циклах с откатом.

Неоптимизированная рекурсия с условием продолжения имеет вид:

```
p() :- условие_продолжения(), тело_цикла(), p().
p().
```

При каждом рекурсивном вызове в стек будет помещаться адрес возврата на второе правило рекурсии. Поэтому размер используемого стека станет линейно расти. Это может привести к истощению стека. Поэтому использовать такую рекурсию следует осмотрительно.

Неоптимизированная рекурсия может содержать несколько неоптимизированных рекурсивных правил, каждое из которых имеет собственное тело цикла:

```
p() :- условие_завершения1().  
p() :- условие_продолжения1(), тело1(), p().  
p() :- условие_продолжения2(), тело2(), p().  
p() :- условие_продолжения3(), тело3(), p().
```

При каждом рекурсивном вызове в стек будет помещаться адрес возврата на следующее правило рекурсии. Конечно, размер используемого стека станет расти линейно. Однако положительным свойством такого цикла является его недетерминированность. Это хорошее свойство алгоритма при поиске в пространстве состояний. С помощью подобных циклов язык Пролог легко справляется с поисковыми задачами, особенно из области задач искусственного интеллекта. В этом случае рост используемой памяти является платой за эффективный поиск. Особенностью такого цикла является его способность откатываться назад на нужное количество витков, вплоть до начала цикла, и возобновлять поиск, используя другие правила. Например, такой цикл может со 117-го витка цикла при необходимости откатиться на 116 виток и попытаться выполнить альтернативное тело цикла. Если альтернативы также неуспешны, то Пролог может откатить цикл еще на виток назад, вплоть до первого витка, и возобновить выполнение с использованием других альтернативных правил. С точки зрения императивных языков программирования такое поведение цикла считается как минимум необычным. Но именно такое поведение позволяет писать на Прологе достаточно лаконичные и эффективные программы.

Закончим рассмотрение неоптимизированных рекурсий случаем использования нехвостовых рекурсивных правил.

Нехвостовая рекурсия с условием завершения имеет вид:

```
p() :- условие_завершения(), !.  
p() :- тело1(), p(), тело2().
```

Как можно увидеть, после рекурсивного вызова должен выполниться еще предикат `тело2()`. Перед каждым рекурсивным вызовом в стек будет помещаться адрес предиката `тело2()`. Поэтому такая рекурсия считается неоптимизированной несмотря на то, что в первом правиле рекурсии используется отсечение. Можно даже поставить отсечение перед рекурсивным вызовом, но это не сделает эту рекурсию оптимизированной, т. к. это отсечение будет выполнено до рекурсивного вызова и не воспрепятствует помещению в стек адреса предиката `тело2()`. Стоит заметить, что наличие предиката `тело1()` приведено для общего случая. Его может и не быть вовсе, и при этом рекурсия будет неоптимизированной, поскольку является нехвостовой.

Такого вида рекурсия используется тогда, когда известно, что количество повторов цикла будет невелико. Преимуществом нехвостовых рекурсий является небольшой выигрыш в лаконичности, связанный с тем, что один или несколько аргументов рекурсивного предиката можно не использовать и удалить. К таковым аргументам относятся обычно вспомогательные аккумуляторы, в которых накапливается, например, сумма значений некоторой величины.

Характерным примером нехвостовой рекурсии является предикат вычисления факториала:

```
f(0, 1) :- !.
f(N, F) :- f(N-1, F1), F=N*F1.
```

Оптимизировать эту рекурсию нет никакого смысла, т. к. количество витков цикла из-за быстрого роста факториала невелико, и стек не успевает значительно уменьшиться. А при завершении цикла стек будет освобожден.

Нехвостовая рекурсия с условием продолжения имеет вид:

```
p() :- условие_продолжения(), тело1(), !, p(), тело2().
p().
```

Здесь наличие отсечения перед рекурсивным вызовом удалит из стека адрес возврата на второе правило рекурсии. И это хорошо. Однако затем, во время рекурсивного вызова, в стек все равно будет помещен адрес предиката `тело2()`. Рекурсию такого вида можно использовать тогда, когда известно, что количество повторов цикла будет невелико. Преимуществом ее является отсутствие необходимости использования вспомогательных аккумуляторов. Предикат вычисления факториала для такого вида рекурсии примет вид:

```
f(N, F) :- N>=1, !, f(N-1, F1), F=N*F1.
f(_, 1).
```

В этом разделе мы рассмотрели циклы с условиями продолжения и завершения. Циклы с заданным числом повторений имеют аналогичную структуру, но используют один дополнительный аргумент — счетчик цикла, который на каждом витке цикла декрементируют (инкрементируют).

11.2. Реализация рекурсии

Для правильной реализации рекурсии надо в рекурсивных правилах обеспечить изменение параметра рекурсии до такого значения, которое указано в правиле останова рекурсии.

ПРИМЕР 11.1. Арифметическая прогрессия с шагом 1. Первый аргумент предиката: `ряд(N, K)` на каждом витке цикла инкрементируется до тех пор, пока не превысит значение второго аргумента. По завершении рекурсии на экран будет выведен ряд целых чисел от `N` до `K` включительно.

```
implement main
    open core, console
class predicates
    ряд : (integer From, integer UpTo).
clauses
    ряд(N,K) :- N>K,!.                                % условие останова рекурсии
    ряд(N,K) :- write(N), ряд(N+1,K).                  % рекурсивный вызов
    run() :- ряд(3,7),                                 % вывод: 34567
            _ = readchar().
end implement main
goal
    console::run(main::run).
```

Рекурсивный предикат ряд(N, K) генерирует ряд чисел с шагом 1, начиная со значения N и заканчивая K включительно. Рекурсивный вызов ряд($N+1, K$) осуществляется с новым параметром $N+1$, вычисляя тем самым очередной элемент ряда. Второй параметр K передается без изменений, т. к. именно он определяет значение последнего члена ряда. Условие останова рекурсии успешно тогда, когда первый параметр превысит второй $N > K$. В правиле останова рекурсии использовано отсечение для того, чтобы при откате этот предикат не продолжил вычислять члены ряда, превышающие K . Рекурсия является оптимизированной и выполняется как итеративный цикл, поскольку после рекурсивного вызова ряд($N+1, K$) предикат не содержит ни вызовов других предикатов в рекурсивном правиле, ни других правил.

ПРИМЕР 11.2. Арифметическая прогрессия с шагом D . Первый аргумент предиката: ряд_с_шагом(N, K, D) на каждом витке цикла увеличивается на шаг D до тех пор, пока не превысит значение второго аргумента.

```
implement main
    open core, console
domains
    natural = [1...].
class predicates
    ряд_с_шагом : (integer From, integer UpTo, natural Step).
clauses
    ряд_с_шагом(N,K,_) :- N>K,!.
    ряд_с_шагом(N,K,D) :- write(N), ряд_с_шагом(N+D,K,D).
    run() :- ряд_с_шагом(3,7,2), % вывод: 357
            _ = readchar().
end implement main
goal
    console::run(main:::run).
```

Рекурсивный предикат ряд(N, K, D) генерирует ряд чисел с шагом D , начиная с N и заканчивая величиной, не превышающей K . Рекурсивный вызов ряд($N+D, K, D$) осуществляется с новым параметром $N+D$, вычисляя тем самым очередной элемент ряда. Третий параметр предиката является шагом D , который на каждом витке цикла остается без изменений. Заметьте, что шаг принадлежит к домену natural и должен быть больше нуля, иначе условие останова рекурсии может никогда не выполниться. Рекурсия является оптимизированной и выполняется как итеративный цикл.

ПРИМЕР 11.3. Ряд случайных чисел, содержащий N членов. Предикат ген(N) на каждом витке цикла генерирует псевдослучайное число в диапазоне $0 \leq I < 10$ с помощью функции math::random(10) и декрементирует счетчик цикла N .

```
implement main
    open core, console
class predicates
    ген : (unsigned Counter).
clauses
    ген(N) :- N<1,!.
```

```

ген(N) :- write(math::random(10)), ген(N-1).
run() :- ген(7),                               % вывод семи случайных чисел
         _ = readchar().
end implement main
goal
  console::run(main:::run).

```

Рекурсия является оптимизированной и выполняется как итеративный цикл.

ПРИМЕР 11.4. Усовершенствуем предикат `ген(N)` из предыдущего примера, добавив вычисление суммы ряда случайных чисел. Для этого, во-первых, вынесем функцию `random/1` из предиката `write`, т. к. нам необходимо иметь переменную со значением очередного члена ряда для того, чтобы его складывать с аккумулятором суммы. Во-вторых, добавим вторым аргументом сам аккумулятор суммы. Этот аргумент будет выходным, поэтому в объявлении предиката укажем атрибут `[out]` после типа аргумента.

```

implement main
  open core, console
class predicates
  сумма : (unsigned Counter, unsigned Sum [out]). 
clauses
  сумма(N,0) :- N<1, !.
  сумма(N,S) :-                                % S – сумма N случайных чисел
    I = math::random(10),                         % генерируем случайное число
    write(I),                                     % выводим на экран
    сумма(N-1,S1),                                % S1 – сумма (N-1) случайных чисел
    S = S1+I.                                     % S – сумма N-1 случайных чисел и
                                                % N-го случайного числа
run() :- сумма(3,S),   % генерация и суммирование трех случайных чисел
        nl, write(S),
        _ = readLine().
end implement main
goal
  console::run(main:::run).

```

Идея этой рекурсии основана на том, что для вычисления суммы N членов ряда сначала надо вычислить сумму $N-1$ членов ряда, а потом прибавить к ней N -й член ряда. Поэтому предикат `сумма(N,S)` для вычисления суммы S генерирует N -й член ряда, который записывается в переменную `I`. После этого с помощью рекурсивного вызова `сумма(N-1,S1)` вычисляет сумму $N-1$ членов ряда, которая возвращается в переменной `S1`. На последнем шаге значение суммы N членов ряда вычисляется складыванием суммы $N-1$ членов ряда и N -го члена ряда. Предикат останова рекурсии играет ключевую роль, т. к. именно он определяет то число, с которым начнется сложение членов ряда. Это число — ноль. Как можно заметить, суммирование членов ряда производится, начиная с последнего члена ряда и заканчивая первым. Причем это суммирование происходит на выходе из рекурсии.

В приведенном примере рекурсия является неоптимизированной, т. к. после рекурсивного вызова `сумма(N-1,S1)` должен выполняться предикат `S=S1+I`, поэтому его

адрес помещается в стек, чтобы на выходе из рекурсии вычислить очередное значение аккумулятора суммы.

Рассмотрим работу рекурсии подробно. Пусть на каждом шаге рекурсии генерируются следующие случайные числа: 6, 1 и 7. Предикат `сумма(3, S)` выполняет вычисления, ход которых демонстрируется на рис. 11.1.

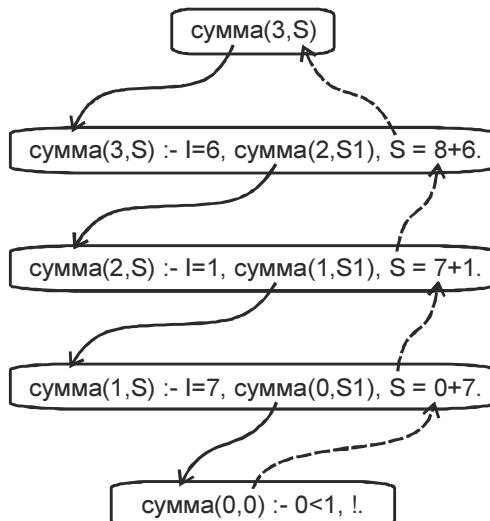


Рис. 11.1. Вычисление суммы на выходе из рекурсии

Рекурсия на первых трех шагах декрементирует счетчик, двигаясь вглубь по стрелкам. На четвертом шаге выполняется предикат останова рекурсии — это является дном рекурсии. При этом будет найдено значение аккумулятора суммы, которое равно нулю. На выходе из рекурсии, обозначенном пунктирными стрелками, с найденным значением аккумулятора будут складываться сгенерированные случайные числа. Сначала к нулю прибавится последнее сгенерированное число $S=0+7$. Потом к этой сумме добавится второе число $S=7+1$. И, наконец, в последнюю очередь добавится первое сгенерированное число $S=8+6$.

Неоптимизированная рекурсия имеет одно достоинство — лаконичность, и как следствие, легкая читабельность. Это объясняется тем, что нет лишних параметров. Недостаток неоптимизированной рекурсии — расход памяти на каждом витке цикла. При достаточно больших исходных данных может наступить переполнение памяти. Поэтому в прикладном программировании не следует использовать неоптимизированную рекурсию без веской причины.

Для того чтобы увидеть, как растет стек в ходе выполнения рекурсии, добавим в нашу рекурсию предикат чтения размера используемого стека и будем на каждом витке цикла выводить его на экран вместе со случайнym числом:

```
implement main
open core, console
```

```

class predicates
    сумма : (unsigned Counter, unsigned Sum [out]) .
clauses
    сумма(N,0) :- N<1,!.
    сумма(N,S) :-                                % S - сумма N случайных чисел
        I = math::random(10),                      % генерируем случайное число
        UsedStack = memory::getUsedStack(),        % получаем размер стека
        write(UsedStack),nl,                        % выводим на экран
        сумма(N-1,S1),                            % S1 - сумма N-1 случайных чисел
        S = S1+I.                                 % S - сумма N-1 случайных чисел
                                                % и N-го случайного числа
run() :-
    сумма(3,S),      % генерация и суммирование трех случайных чисел
    write("Сумма: ",S),
    _ = readLine().
end implement main
goal
    console::run(main:::run).

```

В результате выполнения программы можно увидеть рост стека на 40 байтов на каждом витке цикла:

```

1140
1180
1220
Сумма: 12

```

ПРИМЕР 11.5. Перепишем только что рассмотренную неоптимизированную рекурсию так, чтобы она стала оптимизированной, т. е. не расходовала память при своем выполнении. Для этого добавим в предикат один входной параметр со значением ноль. Этот параметр будет тем самым аккумулятором суммы, значение которого мы получали на дне неоптимизированной рекурсии в примере 11.4.

```

implement main
    open core, console
class predicates
    сумма : (unsigned Counter, unsigned Accum, unsigned Sum [out]) .
clauses
    сумма(N,S,S) :- N<1,!.
    сумма(N,A,S) :-          % A - аккумулятор, S - итоговая сумма
        I=math::random(10),
        write(I),
        сумма(N-1,A+I,S). % сложение аккумулятора A и очередного числа I
run() :-
    сумма(5,0,S),nl, % начальное значение аккумулятора равно нулю
    write(S),
    _ = readLine().
implement main
    open core, console

```

```

class predicates
    сумма : (unsigned Counter, unsigned Sum [out]) .
clauses
    сумма(N,0) :- N<1,! .
    сумма(N,S) :- % S - сумма N случайных чисел
        I = math::random(10), % генерируем случайное число
        UsedStack = memory::getUsedStack(), % получаем размер стека
        write(UsedStack), nl, % выводим на экран
        сумма(N-1,S1), % S1 - сумма N-1 случайных чисел
        S = S1+I. % S - сумма N-1 случайных чисел
                           % и N-го случайного числа
run() :-
    сумма(3,S), % генерация и суммирование трех случайных чисел
    write("Сумма: ",S),
    _ = readLine().
end implement main
goal
    console::run(main:::run).

```

Идея этой рекурсии основана на прибавлении в каждом витке цикла к значению аккумулятора случайного числа. На дне рекурсии значение аккумулятора станет равным искомой сумме. Для того чтобы передать это значение наверх, в точку вызова предиката, надо унифицировать аккумулятор с выходным аргументом, который мы специально «протащили» через всю рекурсию. Эта унификация производится в правиле останова рекурсии. Для этого в нем второй и третий аргументы обозначены одной переменной.

Рассмотрим работу оптимизированной рекурсии подробно. Пусть на каждом шаге рекурсии генерируются случайные числа 6, 1 и 7. Предикат `сумма(3,S)` выполняет вычисления, ход которых отражен на рис. 11.2. Суммирование очередного элемента

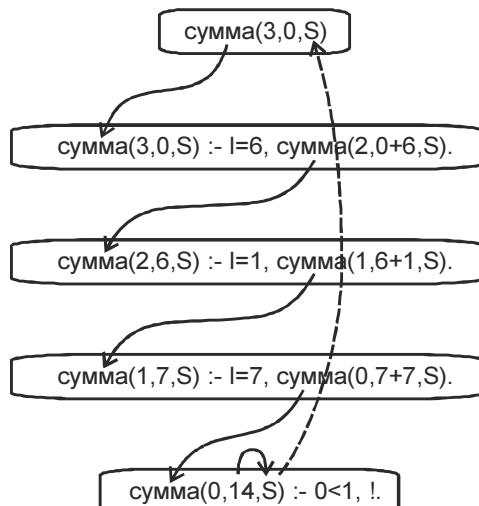


Рис. 11.2. Вычисление суммы до рекурсивного вызова

с текущей суммой производится до рекурсивного вызова. При рекурсивном вызове третий параметр передается без изменений, чтобы с его помощью вытащить со дна рекурсии итоговую сумму. Второй параметр представляет собой аккумулятор текущей суммы. При вызове он равен нулю, а на каждом витке рекурсии к нему прибавляется очередное случайное число, образуя тем самым текущую сумму. Также на каждом витке цикла декрементируется счетчик N до тех пор, пока не станет равным нулю, определяя тем самым окончание рекурсии, т. е. ее дно. Направленные вниз стрелки показывают углубление в рекурсию. На дне рекурсии аккумулятор текущей суммы унифицируется с итоговой суммой. Пунктирная стрелка, направленная вверх, показывает возврат итоговой суммы со дна рекурсии.

ПРИМЕР 11.6. Рассмотрим организацию рекурсивных функций на примере предиката вычисления факториала.

```
implement main
  open core, console
  class predicates
    f:(unsigned) -> unsigned64.
  clauses
    f(0) = 1 :- !.          % условие останова рекурсии: факториал 0 равен 1
    f(N) = N * f(N-1).    % рекурсивный вызов функции
    run() :- write(f(20)),
             _=readline().
end implement main
goal
  console::run(main::run).
```

Здесь использована неоптимизированная рекурсия. Хотя визуально после рекурсивного вызова функции нет явных вызовов других предикатов, но, на самом деле, на выходе из рекурсии Visual Prolog производит основную работу — умножение чисел. Для того чтобы это увидеть, перепишем предикат факториала в иной форме, вынеся умножение в тело предиката:

```
f(0) = 1 :- !.
f(N) = F :- N1=f(N-1), F=N*N1.
```

Этот предикат эквивалентен предыдущему, но явно показывает вычисления, производимые на выходе из рекурсии. Оптимизировать его не имеет смысла, т. к. уже при $N=13$ для типа `unsigned` происходит переполнение разрядной сетки. Максимальное число, факториал которого можно вычислить, для типа `unsigned64` равно 20. А за 20 витков цикла стек не переполнится.

ПРИМЕР 11.7. Рекурсия может использовать не один, а два рекурсивных вызова. Например, n -е число ряда Фибоначчи можно определить по правилу « n -й член ряда равен сумме двух предыдущих». Первые два числа ряда Фибоначчи равны единице.

```
implement main
  open core, console
  class predicates
    fib:(unsigned) -> unsigned.
```

```

clauses
  fib(1) = 1 :- !.                      % первый член ряда равен 1
  fib(2) = 1 :- !.                      % второй член ряда равен 1
  fib(N) = fib(N-1) + fib(N-2).        % двукратный рекурсивный вызов
  run() :-write(fib(12)),
           _=readline().
end implement main
goal
  console::run(main::run).

```

Здесь мы видим два рекурсивных вызова: `fib(N-1)` и `fib(N-2)`. Такая организация рекурсии является неэффективной по двум причинам. Во-первых, растет размер используемого стека, поскольку рекурсия неоптимизированная. Во-вторых, после вычисления `fib(N-1)` происходит повторное вычисление значения `fib(N-2)`, хотя оно было определено во время вычисления `fib(N-1)`. Поэтому на каждом шаге рекурсии происходит, по сути, двойная работа. Эти недостатки приводят к тому, что уже при $N=40$ ощущается задержка при выводе результата.

ПРИМЕР 11.8. Для оптимизации рекурсии необходимо включить в предикат `fib/1` два дополнительных аргумента, которые будут играть роль предпоследнего и последнего членов ряда. Тогда при вызове рекурсии достаточно сложить эти аргументы, чтобы вычислить следующий член ряда. При вызове предиката `fib/3` надо указать первый и второй члены ряда Фибоначчи, которые равны 1. То есть, для нахождения, например, 47-го члена ряда вызов предиката должен быть `fib(47,1,1)`.

```

implement main
  open core, console
class predicates
  fib: (unsigned, unsigned, unsigned) -> unsigned.
clauses
  fib(1, A, _) = A:- !.    % первый член ряда равен второму аргументу
  fib(2, _, B) = B:- !.    % второй член ряда равен третьему аргументу
  fib(N, A, B) = fib(N - 1, B, A + B).
  run() :-write(fib(47,1,1)),
           _=readline().
end implement main
goal
  console::run(main::run).

```

Первый аргумент предиката `fib/3` является номером искомого члена ряда. Второй аргумент — предпоследний член ряда. Третий аргумент — последний член ряда. В правиле:

`fib(N, A, B) = fib(N - 1, B, A + B)`

последний член ряда `B` при рекурсивном вызове становится на место предпоследнего члена, а место последнего члена ряда занимает сумма предпоследнего и последнего членов `A+B`.

Так как все вычисления производятся до рекурсивного вызова, то наш предикат `fib/3` является оптимизированной рекурсией.

ПРИМЕР 11.9. Предикат `fib/3` из примера 11.8 можно упростить, заменив два предиката останова рекурсии одним. Рассмотрим первые члены ряда Фибоначчи, к которым добавлен нулевой член, равный нулю: 0, 1, 1, 2, 3, 5, 8.... Рекурсия для такого ряда с нулевым членом будут выглядеть так:

```
fib(0, A, _) = A:- !.      % нулевой член ряда равен второму аргументу
fib(1, _, B) = B:- !.      % первый член ряда равен третьему аргументу
fib(N, A, B) = fib(N - 1, B, A + B).
```

Для вызова такой рекурсии надо только изменить значение нулевого члена ряда `fib(47, 0, 1)`. Однако, согласно определению ряда Фибоначчи, он не содержит нулевой член, поэтому правило останова для нулевого члена следует удалить. Тогда получается рекурсия с одним правилом останова:

```
implement main
    open core, console
class predicates
    fib: (unsigned, unsigned, unsigned) -> unsigned.
clauses
    fib(1, _, B) = B:- !.
    fib(N,A,B)=fib(N-1,B,A+B).
    run() :-write(fib(47,0,1)),
            _=readline().
end implement main
goal
    console::run(main::run).
```

11.3. Мемоизация

Мемоизация — способ увеличения скорости выполнения программ за счет запоминания результатов промежуточных вычислений. Такой способ работает тогда, когда одни и те же промежуточные результаты неоднократно требуются при дальнейших вычислениях. Мемоизация, как правило, используется в рекурсивных циклах и позволяет исключить повторные вычисления. Иногда мемоизацию называют *табулированием*.

ПРИМЕР 11.10. Введем мемоизацию в предикат вычисления *n*-го члена ряда Фибоначчи, первоначальная версия которого была приведена в примере 11.7 и представляла собой неоптимизированную рекурсию. Для этого объявим недетерминированный факт `ff/2`, первый аргумент которого хранит номер члена ряда, а второй — значение этого члена. После вычисления очередного члена ряда мы его будем сохранять в памяти в виде факта.

```
implement main
    open core, console
class facts
    ff:(unsigned,unsigned).
class predicates
    fib:(unsigned) -> unsigned.
```

```
clauses
```

```
fib(1) = 1 :- !.  
fib(2) = 1 :- !.  
fib(N) = F :- ff(N, F), !.  
fib(N) = F :- F = fib(N-1) + fib(N-2), asserta(ff(N, F)).  
run() :- write(fib(47)),  
       _ = readline().
```

```
end implement main
```

```
goal
```

```
console:::run(main:::run).
```

Хотя рекурсия является формально неоптимизированной, но использование мемоизации позволило устранить временную задержку, проявляющуюся при вычислении членов ряда для $N > 40$ в примере 11.7.

ГЛАВА 12



Ввод/вывод

12.1. Ввод/вывод в консольном приложении

Чтение с клавиатуры зависит от режима, в котором находится консоль. Когда консоль находится в режиме ввода строки, то предикаты чтения возвращают введенную информацию только после нажатия клавиши <Enter>. Если консоль находится в режиме посимвольного ввода, то предикаты чтения возвращают информацию сразу после нажатия клавиши на клавиатуре, не ожидая нажатия клавиши <Enter>.

Для проверки состояния консоли служит предикат `isModeLine`. Этот предикат успешен, если консоль находится в состоянии ввода строки, и неуспешен, если консоль находится в состоянии посимвольного ввода. Для установки режима ввода строки служит предикат `setModeLine(true)`, а для установки режима посимвольного ввода — предикат `setModeLine(false)`. В режиме посимвольного ввода эхо-вывод на экран не производится.

12.1.1. Основные функции ввода с клавиатуры

- `Str = readLine()` — чтение строки по клавише <Enter>, при этом консоль должна находиться в режиме ввода строки.
- `Char = readChar()` — чтение символа. Если консоль находится в режиме ввода строки, то функция возвратит первый из введенных символов, но только после нажатия клавиши <Enter>. Если консоль находится в режиме посимвольного ввода строки, то первый введенный символ будет возвращен немедленно, без нажатия клавиши <Enter>.
- `X = read()` — чтение терма, при этом консоль должна находиться в режиме ввода строки. Терм `x` может быть числом, списком, двоичными данными или принадлежать к домену, объявленному пользователем. Во многих случаях компилятор самостоятельно определяет тип читаемого терма на основании операций, производимых над ним после чтения. Если тип терма компилятором не определен, то в этом случае необходимо перед чтением объявить тип свободной переменной `x` с помощью предиката `hasDomain(Type,X)`.

12.1.2. Основные предикаты вывода на экран

- `nl()` — перевод курсора на новую строку.
- `write(...)` — вывод аргументов. Количество и типы аргументов могут быть произвольными.
- `writef(FormatString,...)` — форматируемый вывод. Первый аргумент — это шаблон выводимой строки. Остальные аргументы — параметры, значения которых подставляются в шаблон. Для этого шаблон должен содержать выводимый текст, в который вставлены знаки процента. Во время вывода знаки процента заменяются значениями параметров. Количество знаков процента в шаблоне должно соответствовать количеству параметров в предикате. После знака процента могут следовать атрибуты вывода:

`%[-][0][ширина][.точность][формат].`

Квадратные скобки здесь просто отделяют один атрибут от другого и в исходном тексте программы не указываются. Атрибуты вывода:

- `[-]` — дефис указывает на выравнивание выводимого параметра влево. При отсутствии дефиса осуществляется выравнивание вправо. Это актуально, когда ширина значения параметра меньше ширины поля;
- `[0]` — знак нуля перед шириной поля означает, что значение параметра будет дополняться нулями слева до тех пор, пока не будет достигнута установленная ширина поля. Если знак нуля следует за дефисом, то знак нуля будет игнорирован;
- `[ширина]` — положительное десятичное число указывает ширину поля вывода параметра. Если количество символов значения параметра меньше ширины поля, то необходимое количество пробелов будет добавлено слева или справа, в зависимости от направления выравнивания;
- `[.точность]` — точка, следом за которой следует беззнаковое десятичное число, указывает на количество выводимых знаков после запятой для вещественного числа или максимальное число выводимых символов для строки. По сути, это округление числа до заданной точности;
- `[формат]` — формат выводимого параметра:
 - `f` — фиксированный десятичный формат для вещественных чисел, например: `123.4` или `0.004321`. Этот формат установлен по умолчанию для чисел типа `real`;
 - `e` — экспоненциальный формат (мантиssa и порядок) вещественного числа, например: `1.234e+002` или `4.321e-003`;
 - `g` — сокращенный формат вещественного числа. Если порядок числа меньше чем `-4` и больше или равен точности представления чисел типа `real`, то при выводе число форматируется как мантиssa и порядок;
 - `d` или `D` — формат целых десятичных чисел со знаком;
 - `u` или `U` — формат беззнаковых целых чисел;

- x или X — формат шестнадцатеричных чисел;
- o или O — формат восьмеричных чисел;
- c — символ;
- B — двоичные данные;
- P — параметр процедуры;
- s — строка.

Далее представлены способы форматированного вывода:

```
X=123.4567,
writeln("%08.2f \n",X),
writeln("%e \n",X),
writeln("%12g \n",X),
writeln("16-е число:%x\t8-е число: %o\t10-е число: %d",128,9,0x0090),
```

Результат вывода на экран:

```
00123.46
1.234567e+002
123.4567
16-е число: 80      8-е число: 11      10-е число: 144
```

Первое число на экране занимает поле шириной 8 символов, дополненное слева нулями, округленное до второго знака после запятой и имеющее фиксированный десятичный формат. Далее указана ESC-последовательность перехода на новую строку \n. Второе число на экране имеет экспоненциальный формат. Третье число занимает поле шириной 12 символов, прижато вправо и имеет фиксированный десятичный формат. Четвертая строка на экране содержит три числа, представленные в разных системах счисления.

12.1.3. Использование предиката *hasDomain*

Предикат `hasDomain(Type, X)` служит для указания типа Type, к которому принадлежит переменная X во время компиляции программы. Этот предикат не имеет какого-либо кода, работающего во время выполнения программы.

Рассмотрим пример, когда компилятор может определить тип вводимых данных:

```
run():-X=read(), Y=read(),
        write(X+Y),
        clearinput(),
        _=readchar().
```

Так как вводимые термы X и Y в дальнейшем складываются, то компилятор делает правильное заключение о том, что вводятся числа. Предикат `=readchar()` ожидает нажатие клавиши <Enter> для того, чтобы задержать закрытие окна консоли после вывода суммы. Предикат `clearinput()` очищает буфер ввода для того, чтобы предикат `=readchar()` правильно работал. Дело в том, что ввод терма Y оставляет в буфе-

ре код возврата каретки, и если буфер не очистить, то предикат `_ = readchar()` прочитает этот код, и окно консоли сразу закроется.

Рассмотрим случай, когда программист должен указать тип вводимых данных. Пусть необходимо сравнить два списка целых чисел. Для этого с помощью предикатов `hasDomain` объявим домен, к которому принадлежат переменные `X` и `Y`, после чего мы можем корректно сравнить списки и вывести наибольший:

```

implement main
    open core, console
domains
    il = integer*.                      % объявление списка целых чисел
clauses
    run():-hasDomain(il,X),            % объявление типа переменной X
        hasDomain(il,Y),              % объявление типа переменной Y
        X=read(), Y=read(),           % ввод двух списков в квадратных скобках
        (X>Y, write(X), !; write(Y)), % вывод максимального списка
        clearinput(),
        _=readline().
end implement main
goal
    console::run(main::run).

```

Если же тип переменных не объявить, то Visual Prolog не сможет корректно определить тип переменных, т. к. предикат сравнения переменных работает практически со всеми типами данных и доменами.

12.2. Файловый ввод/вывод

Класс `file` обеспечивает все стандартные операции с файлами: чтение, запись, удаление, копирование, перемещение, поиск, чтение/запись атрибутов файлов и т. п. При этом, как правило, файловые операции не изменяют содержимое файла. При чтении текстового файла читается весь файл целиком:

```
Text = file::readString(FileName).
```

Чтение двоичного файла целиком:

```
Binary = file::readBinary(FileName).
```

Рассмотрим чтение атрибутов и даты создания файла. Для преобразования даты создания файла, представленной вещественным числом, в аргументы Год, Мес, День, Час, Мин, Сек, служит функция получения даты и времени: `Q=time::newFromGMT/1` и предикат преобразования параметра `Q` в заданный формат даты и времени `Q:getDateAndTime/1`:

```

file::getFileProperties("Abc.zip", Атрибуты, _, Создание, _, _),
write(Атрибуты), nl,
Q=time::newFromGMT(Создание),
Q:getDateAndTime(Год, Мес, День, Час, Мин, Сек),
writeln("%..%..%..%:%:%:", Год, Мес, День, Час, Мин, Сек).

```

Запись текста в файл:

file::writeString (FileName, Text)

Запись двоичных данных в файл:

```
file:::writeBinary(FileName, Binary)
```

Класс `file` описан в [приложении 2](#). Чтение и запись файлов, осуществляемая предикатами этого класса, является самым быстрым способом доступа среди других известных. В этом — преимущество такого вида доступа. А недостатком является сокращение объема свободной оперативной памяти — т. к. весь файл целиком надо размещать в памяти компьютера.

В противовес файловому вводу/выводу, потоковый ввод/вывод дает возможность произвольного доступа к фрагментам файла.

12.3. Потоковый ввод/вывод

Пакет `stream` обеспечивает потоковый доступ к объектам по чтению/записи. Он позволяет позиционировать указатель потока ввода/вывода, читать термы, байты, символы, абзацы, строки заданной длины, а также записывать информацию из переменных программы, адресуемых байтов оперативной памяти, осуществлять форматированный вывод. Кроме этого пакет `stream` дает возможность загружать и сохранять факты именованных разделов динамической БД и управлять режимами потока ввода/вывода. Описание основных классов этого пакета приведено в *приложении 5*.

Для доступа к файлам средствами пакета `stream` необходимо предварительно получить дескрипторы входного потока читаемого файла и выходного потока записи-ваемого файла с помощью конструкторов классов `inputStream_file` и `outputStream_file` пакета `fileSystem`. Эти классы описаны в *приложениях 3 и 4* соответственно.

В пакете `stream` также имеется класс `stdio`, обеспечивающий простой и удобный доступ к потокам ввода/вывода. С помощью его свойств можно получить доступ, например, к стандартному входному потоку (клавиатуре):

```
StandartInput = stdio::inputStream
```

или изменить его на другой поток:

```
stdio::inputStream := MyInputStream
```

Рассмотрим пример нумерации абзацев. Далее приведены декларация факта *i*, играющего роль счетчика, и фрагмент правила для чтения ANSI-файла "aa.txt" по абзацам, их нумерации и вывода нумерованных абзацев в файл "zz.txt" в кодировке Unicode.

```

In = inputStream_file::openFile8("aa.txt"),      % дескриптор ввода
Out = outputStream_file::create("zz.txt"),       % дескриптор вывода
std::repeat(),                                % начало цикла
S=In:readLine(),                             % чтение абзаца
i:=i+1,                                     % инкремент счетчика
Out:writef("% %\n",i,S),                     % вывод номера и строки
In:endOfStream(),                           % конец файла?
In:close(),                                 % закрываем поток ввода
Out:close(),                               % закрываем поток вывода
...

```

Предикат `repeat()` играет роль начала цикла, а предикат проверки конца файла `endOfStream/0` является условием завершения цикла. Пока указатель позиции чтения файла не достигнет конца файла, этот предикат будет неуспешен, вследствие чего возникает откат назад до предиката `repeat()`, который возобновляет следующий виток цикла. В теле цикла функция `readLine/0` читает абзац и передвигает указатель чтения в файле на начало следующего абзаца. Далее инкрементируется счетчик и в выходной поток пишется номер абзаца, через пробел сам абзац и код перехода на новую строку. Когда указатель чтения файла достигнет конца файла, то цикл завершится, и потоки ввода/вывода будут закрыты.

Рассмотрим пример побайтового подсчета контрольной суммы по модулю 2. Для байтового доступа к файлу он открыт как двоичный посредством указания параметра `stream::binary` в конструкторе `inputStream_file::openFile/2`. Сумма байтов накапливается в аккумуляторе, роль которого играет факт-переменная `s`:

```

class facts
  s : unsigned8 := 0.
clauses
...
In = inputStream_file::openFile("www.txt",stream::binary),
hasDomain(unsigned8,B),           % объявление типа переменной B
std::repeat(),
B = In:read(),                  % чтение байта из файла
s := bit::bitXor(s,B),          % сложение с аккумулятором по модулю 2
In:endOfStream(),               % конец файла?
In:close(),                     % закрываем поток ввода
console::write(s),              % выводим контрольную сумму на экран
...

```

Рассмотрим пример чтения фрагмента файла с позиции `Start` длиной `Length`. Здесь следует отметить, что позиция начала чтения указывается в байтах. Поэтому при чтении Unicode-файлов позиция должна быть кратной двум — поскольку в этом случае Visual Prolog читает двухбайтовые Unicode-символы. Позиция первого символа файла принята нулевой. Длина читаемой строки указывается в количестве символов, а не байтов. В приведенном далее примере из Unicode-файла "zzz.txt" читается строка длиной 11 символов. Предикат `setPosition(6)` передвигает указатель чтения на шестой байт, начиная с нулевого. Следовательно, читаемая строка

будет начинаться с четвертого символа, т. к. нулевой и первый байт содержат первый символ, второй и третий байт — второй символ, а четвертый и пятый байт — третий символ:

```
In = inputStream_file::openFile("zzz.txt", stream::unicode()),
In:setPosition(6),
S = In:readString(11),
In:close(),
console::write(S),
```

Если указанная длина читаемой строки выходит за пределы файла, то будет прочитана строка до конца файла.

Достоинством потокового ввода/вывода является возможность обработки больших файлов без существенного сокращения свободной оперативной памяти. Недостаток — некоторое замедление операций ввода/вывода по сравнению с файловым вводом/выводом.

12.3.1. Потоковый ввод/вывод в цикле с откатом

Циклы с откатом используются для чтения файлов в случае, когда требуется обработка данных по мере чтения элементов файла.

ПРИМЕР 12.1. Создадим файл БД со словами русского языка. Источником для этого может послужить любой файл с перечнем русской лексики, взятый из Интернета. Например, ресурс <http://www.speakrus.ru/dict/> содержит богатый выбор словарей для скачивания. Для более простого преобразования текстового файла в файл динамической БД лучше скачать словарь, содержащий в каждой строке одно слово. При запуске программы скачанный файл должен находиться в папке Exe вашего консольного проекта. Пусть имя этого файла Словарь.txt.

ЗАМЕТКА

Файл Словарь.txt расположен в папке Projects\Primer36_1\Exe электронного архива, сопровождающего книгу (см. приложение 11).

Для преобразования текстового файла в БД слов существует несколько путей. Самый простой — прочитать весь текст файла и разделить его на слова в памяти. Однако лучший способ — читать текст не целиком, а по словам или по строкам. Тем самым мы можем обрабатывать файлы достаточно большого размера, не захватывая много памяти. В следующем примере кроме создания БД еще подсчитывается количество слов в БД:

```
implement main
  open core, console
  class facts - слова
    w: (string).           % объявление факта для слов
    i: unsigned :=0.
  class predicates
    r: (inputStream).
```

```

clauses
r(In):-std::repeat,
        S=In:readLine(),
        assert(w(S)),
        i:=i+1,
        In:endOfStream(),!;
        succeed.
run():-In=inputStream_file::openFile8("Словарь.txt"),
       r(In),
       file::save("Словарь.w", слова, false()),
       write("Всего слов = ",i),
       _ = readLine().
end implement main
goal
    console::run(main:::run).

```

Получение нового потока ввода:

```
In=inputStream_file::openFile8("Словарь.txt"),
```

Преобразование содержимого файла в БД:

```
r(In),
```

Сохранение БД в ANSI-файле:

```
file::save("Словарь.w", слова, false()),
```

После выполнения программы в папке Exe должен появиться файл Словарь.w, содержащий БД слов русского языка. В конце файла должен быть факт, указывающий количество слов в словаре. Словарь содержит слова в виде фактов `w(Слово)`. На основе такого словаря можно решать простые задачи, связанные с обработкой русских текстов. Для задач, требующих быстрого доступа к словам, словарь надо создавать в виде дерева или хэш-таблицы. Эти вопросы будут рассмотрены в *главах 25 и 27*.

12.3.2. Потоковый ввод/вывод в рекурсивном цикле

Рекурсивные циклы используются для чтения файлов в случае, когда требуется обработка данных по мере чтения элементов файла и сборка в памяти компьютера нового содержимого файла из полученных элементов.

ПРИМЕР 12.2. Пусть для каждого абзаца текстового файла необходимо получить список чисел в порядке их упоминания в файле. При этом собрать все списки в общий сводный список, сохраняя порядок следования абзацев в тексте.

```

implement main
    open core, console, string
class predicates
    fileToListList : (inputStream, string**) procedure (i,o).
    lineToList : (string, string*) procedure (i,o).

```

```

clauses
fileToListList(In, []):-  

    In:endOfStream(),           % конец файла?  

    !.  

fileToListList(In, [List|Tail]):-  

    S=In:readLine(),            % читаем абзац  

    lineToList(S,List),         % получаем список чисел абзаца  

    fileToListList(In,Tail).  

lineToList(S, [N|Tail]):-  

    fronttoken(S,N,S1),  

    _ = tryToTerm(real,N),!,   % N — число?  

    lineToList(S1,Tail).        % если число, то N в список  

lineToList(S,Tail):-  

    fronttoken(S,_,S1),!,       % иначе пропускаем нечисло  

    lineToList(S1,Tail).  

lineToList(_,[]).  

run():-In = inputStream_file::openFile8("abc.txt"),  

       fileToListList(In,LL),  

       write(LL),  

       _ = readLine().  

end implement main
goal
    console::run(main::run).

```

Если файл abc.txt будет содержать текст:

```

Abc zzz!!!
T 345/67.8 qwerty
zzz $$$ 1.2E-4.

```

то результат выполнения программы может быть, например, таким сводным списком:

```
[[], ["345", "67.8"], ["1.2E-4"]]
```

Он означает, что в первом абзаце текстового файла чисел нет, во втором абзаце два числа 345 и 67.8, а в третьем абзаце одно число, представленное в экспоненциальной форме: 1.2E-4.

12.4. Строковые потоки

Мы рассмотрели файловый и консольный ввод/вывод, которые дают возможность программисту читать текст из файла или консоли порциями: например, по символу — с помощью функции `readchar`, по абзацам — с помощью функции `readLine` или по термам — с помощью функции `read`. Если же текст из файла или консоли прочитан целиком, то его обработку мы делали с помощью предикатов класса `string`.

В этом разделе мы рассмотрим способ порционного чтения текста из строковой переменной так, как будто мы читаем из потока ввода. Такой способ чтения использу-

зуется тогда, когда текст в эту переменную был помещен целиком из файла, консоли или из интерфейсного элемента ввода текста — например: `edit_ctl` GUI-приложения.

Кроме того, мы увидим, как выполнить обратный процесс — последовательно писать фрагменты текста в поток вывода так, как будто мы записываем в файл. После этого достаточно прочитать строку, полученную в результате последовательной записи фрагментов, чтобы получить весь текст целиком. В некоторых случаях такой способ формирования строки проще иных способов.

2.4.1. Поток ввода строк

Поток ввода строк организует конструктор `X = inputStream_string::new(S)`, в котором переменная `S` содержит текст, который нам надо читать порциями. Конструктор возвращает дескриптор потока `X`, имеющий тип `inputStream`. Используя его в качестве объекта, мы можем получать фрагменты текста `S` функциями чтения, получать/устанавливать позицию указателя чтения, получать/устанавливать режим потока (Unicode, ANSI) и т. д. После всех операций чтения поток ввода следует закрыть предикатом `X:close`.

Последовательность операций при использовании потока ввода такова:

```
X = inputStream_string::new(S),  
<блок чтения из потока, например X:readLine(), ...>  
X:close
```

Класс `outputStream_string` пакета `string` содержит предикаты для работы с потоком ввода и потоком вывода строк. Далее описаны яркие представители этого класса:

- `X:save(DBName)` — сохранение фактов именованной БД `DBName` в поток `X`;
- `X:write(...)` — запись переменных в поток `X`;
- `Mode = X:getMode()` — получение режима потока `Mode`. Возможны три значения **режима**: `unicode`, `ansi(codePage)`, `binary`;
- `Position = X:getPosition()` — получение позиции указателя `Position`, с которой будет производиться следующая операция чтения или записи. Позиция первого символа равна нулю;
- `X:setMode(Mode)` — установка нового режима работы `Mode`;
- `X:setPosition(Position)` — установка новой позиции указателя `Position`.

ПРИМЕР 12.3. Пусть имеется текст, который связан с переменной `s`. Необходимо удалить пустые абзацы, а оставшиеся пронумеровать. Для этого создадим поток ввода, который связем с переменной `s`. Получив дескриптор потока ввода `In`, передадим его в предикат `myRead`, который будет выполнять требуемую работу. Кроме того, передадим этому предикату номер, с которого следует нумеровать абзацы. Третьим аргументом предикат вернет нам список пронумерованных непустых абзацев.

```

implement main
    open core,console,string
class predicates
    myRead: (unsigned Номер, inputStream Поток, string* [out]) .
clauses
    myRead(_, In, []) :- In:endOfStream(), !.      % конец потока ввода
    myRead(N, In, [B|L]) :-
        A = In:readLine(),                            % чтение абзаца
        not(isWhiteSpace(A)),                        % если абзац непустой, то
        B = format("%") ~ N, A, !,                  % добавляем номер
        myRead(N+1, In, L).
    myRead(N, In, L) :-
        myRead(N, In, L).                          % если абзац пустой, то читаем следующий
run() :- S = "abc 123\n\t\t\n999 xyz\n",
        In = inputStream_string::new(S),           % создаем поток ввода
        myRead(1, In, L),
        In:close(),                                % закрываем поток ввода
        write(L),
        _ = readchar().
end implement main
goal
    console::run(main:::run) .

```

В результате строка "abc 123\n\t\t\n999 xyz\n" была преобразована в список двух пронумерованных абзацев ["1) abc 123","2) 999 xyz"]. Пустые абзацы и абзацы с табуляциями были игнорированы.

Как мы увидели, чтение строк текста выполняется достаточно просто. Когда же текст состоит из каких-либо термов, то перед чтением с помощью функции `A = X:read()` надо объявить тип переменной `A`, т. е. указать тип терма с помощью предиката `hasDomain(тип, A)`. Во время чтения пробелы, табуляции и символы перевода строки, стоящие перед термом, после терма или между аргументами терма, корректно пропускаются.

ПРИМЕР 12.4. Пусть имеется текст, который содержит сведения о дате и размере поступивших в цех прямоугольных деталей. Формат даты и размера детали задан в виде термов:

```

data: (unsigned Число, unsigned Месяц, unsigned Год)
size: (real X, real Y) .

```

Пары `дата data/3` и `размер size/2` могут следовать друг за другом в одной строке, эта строка также может быть разорвана на несколько строк в любом месте. Требуется собрать в список все термы с размерами деталей `size/2`, а термы дат сохранить во внутренней базе данных по отдельности. Основную работу выполняет предикат `myRead`. Для того чтобы проконтролировать результат, в конце программы выведем список размеров и факты БД на экран. Заметьте, что переменная `S` задана с префиксом `@`, что позволило разделить длинную строку на две подстроки клавишей `<Enter>`.

```

implement main
    open core,console
domains
    ttt = size(real,real).           % размеры X и Y
class facts - dt
    data:(unsigned,unsigned,unsigned).   % (число, месяц, год)
class predicates
    myRead:(inputStream,ttt* [out]). 
clauses
    myRead(In,[]):-In:endOfStream(),!.    % конец потока ввода
    myRead(In,[B|L]):-%
        hasDomain(dt,A),      % переменная A имеет тип dt
        A = In:read(),         % чтение терма data
        hasDomain(ttt,B),     % переменная B имеет тип ttt
        B = In:read(),         % чтение терма size
        assert(A,!),          % сохранение факта data
        myRead(In,L).
run():- % строка для чтения:
    S = @"data(17,04,2008) size(0.98,1.23)
          data(01,06,2013) size(0.88,1.01)",
    In = inputStream_string::new(S),       % создаем поток ввода
    myRead(In,L),
    In:close(),                         % закрываем поток ввода
    (data(A,B,C),                      % очередная дата
     write(data(A,B,C)),nl,            % вывод даты
     fail,
     write(L)),                        % вывод размеров
     _ = readchar().
end implement main
goal
    console::run(main:::run).

```

Результат работы программы демонстрирует, что последовательность дат и размеров соблюдена.

```

data(17,4,2008)
data(1,6,2013)
[size(0.98,1.23),size(0.88,1.01)]

```

12.4.2. Поток вывода строк

Поток вывода строк организует конструктор `X = outputStream_string::new()`. Конструктор возвращает дескриптор потока `X`, который имеет тип `outputStream`. Используя его в качестве объекта, мы можем выполнять действия, аналогичные описанным для потока ввода: выводить фрагменты текста предикатами записи, получать/устанавливать позицию указателя чтения, получать/устанавливать режим потока (Unicode, ANSI) и т. д. После всех операций записи следует прочитать из потока получившуюся строку с помощью функции `X:getString` и закрыть поток

вывода предикатом `X:close`. Последовательность операций при использовании потока вывода такова:

```
X = outputStream_string::new(),
< блок записи в поток, например X:write("abc"),...>
S = X:getString(),
X:close
```

ПРИМЕР 12.5. Пусть имеется текст, который связан с переменной `s`. Необходимо получить новый текст `s1`, в котором пустые абзацы текста `s` удалены, а оставшиеся пронумерованы. Для этого создадим поток ввода, который связем с переменной `s`. Также создадим поток вывода. Получив дескриптор потока ввода `In` и дескриптор потока вывода `Out`, передадим их в предикат `enumeration`, который будет выполнять заданное преобразование текста. Передадим этому предикату также номер, с которого следует нумеровать абзацы.

```
implement main
  open core,console,string
class predicates
  enumeration:(unsigned,inputStream,outputStream).
clauses
  enumeration(_,In,_):-In:endOfStream(),!. % конец потока ввода
  enumeration(N,In,Out):-
    A = In:readLine(),
    not(isWhiteSpace(A)),
    B = format("%") %",N,A),!,
    Out:write(B),Out:nl,
    enumeration(N+1,In,Out).
  enumeration(N,In,Out):-
    enumeration(N,In,Out). % если абзац пустой,
                          % то читаем следующий
run():-S = "abc 123\n\t\t\n999 xyz\n",
  In = inputStream_string::new(S),
  Out = outputStream_string::new(),
  enumeration(1,In,Out),
  In:close(),
  S1= Out:getString(),
  Out:close(),
  write(S1),
  _ = readchar().
end implement main
goal
  console::run(main:::run).
```

В результате строка "abc 123\n\t\t\n999 xyz\n", содержащая четыре абзаца, преобразована в строку "1) abc 123\n2) 999 xyz\n", содержащую два непустых пронумерованных абзаца. При выводе символы '\n' будут интерпретированы как команды переноса на новую строку, поэтому на экране мы увидим текст:

- 1) abc 123
- 2) 999 xyz



ГЛАВА 13

Списки

13.1. Представление списков в памяти компьютера

В Visual Prolog реализованы только односвязные списки. Односвязные списки в памяти представляются линейной последовательностью узлов, в которой каждый узел хранит значение одного элемента списка и адрес следующего. Адрес первого элемента является адресом собственно списка. Последний узел содержит маркер пустого списка. Обход списка возможен только слева направо. На рис. 13.1 изображено представление односвязного списка в памяти.



Рис. 13.1. Представление списка в памяти

В Visual Prolog список может содержать элементы, принадлежащие только одному домену. Размер каждого элемента списка может быть произвольным.

Достоинством списков является быстрое добавление нового элемента в заданную позицию, а также удаление элемента из заданной позиции. Недостаток списков — линейная сложность доступа к элементам списка. Чем больше позиция элемента, тем дольше доступ к нему. Это объясняется тем, что для доступа к n -му элементу надо продвигаться по цепочке элементов, начиная с первого и узнавая на каждом шаге адрес следующего элемента. На $n-1$ шаге мы узнаем адрес n -го элемента, а на n -м узнаем собственно значение n -го элемента.

В противоположность спискам, в массивах размер всех элементов одинаков, поэтому время доступа к элементам массива фиксировано и не зависит от индекса. Однако у массивов есть свой недостаток — добавление или удаление элемента массива по произвольному индексу выполняется с дополнительными накладными расходами. Это объясняется тем, что при добавлении элемента надо «раздвинуть»

массив в заданной позиции, чтобы освободить место для нового элемента. А при удалении элемента из массива надо переместить элементы так, чтобы занять место удаленного элемента. В табл. 13.1 показаны различия между списками и массивами.

Таблица 13.1. Различия между списками и массивами

Операция	Список	Массив
Размер элементов	Произвольный	Фиксированный
Доступ к элементу по его индексу	Линейное время, пропорциональное значению индекса	Константное время
Вставка нового элемента в произвольную позицию	Линейное время, пропорциональное значению индекса	Линейное время, пропорциональное размеру массива
Удаление элемента из заданной позиции	Линейное время, пропорциональное значению индекса	Линейное время, пропорциональное размеру массива

В виде списков удобно представлять упорядоченную совокупность данных, в которую по логике решения задачи предполагается добавление или удаление элементов. Также в виде списков удобно представлять совокупность данных различного размера.

13.2. Встроенные операции над списками

Исторически первым понятие списка ввел Джон Маккарти при разработке языка Lisp. Список является рекурсивной структурой, которая имеет графическое представление, показанное на рис. 13.2 в виде дерева.

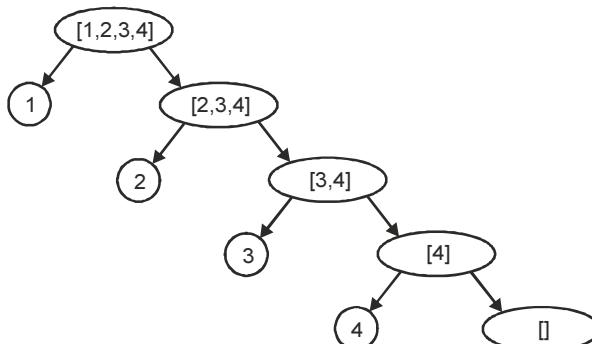


Рис. 13.2. Представление списка в виде дерева

На этом рисунке видно, как список разделяется на голову и хвост. В свою очередь хвост, являясь списком, также разделяется на голову и хвост. В синтаксисе языка Visual Prolog первый элемент списка и хвост списка разделяются вертикальной чертой. Первый элемент называют *головой списка*, и он всегда находится слева

в списке. *Хвост списка* — это исходный список без головы. Хвостом списка может быть и пустой список.

Запись $[A|L]$ означает список, содержащий элемент A в качестве головы и список L в качестве хвоста. Допускается выделять несколько первых элементов списка. Например, запись $[A,B,C|L]$ означает список, в котором A является первым элементом, B — вторым, а C — третьим элементом списка. Хвост списка представлен переменной L. Пустой список [] нельзя разделить на голову и хвост.

Синтаксис списков описан в главе 1. В табл. 13.2 приведены примеры как правильных, так и ошибочных обозначений списков.

Таблица 13.2. Примеры обозначения списков

Обозначение	Пояснение
$[1,2,3 [4]]$	Список $[1,2,3,4]$
$[1,2,3,4 []]$	Список $[1,2,3,4]$
$[1 [2,3,4]]$	Список $[1,2,3,4]$
$[1 [2 [3,4]]]$	Список $[1,2,3,4]$
$[[], []]$	Список содержит два элемента — два пустых списка
$[1,2,3 4]$	Ошибка. Вместо хвоста $[4]$ указан элемент 4
$[[1,2] [3,4]]$	Ошибка. Так списки соединять нельзя
$[1,2 [3] [4]]$	Ошибка. Указано два хвоста
$[[1,2,3] 4]$	Ошибка. Попытка отделить голову справа, а хвост слева
$[1, "ABC", 3]$	Ошибка. Список должен содержать элементы одного типа

На основе описанной конструкции представления списка в виде головы и хвоста в Visual Prolog можно выполнять две операции: конструирование списка и разделение списка.

Операция конструирования списка соединяет заданную голову и хвост, поэтому и называется *конструктором*. Например, для конструирования нового списка из заданного элемента A=2 и списка L=[5,6,7] достаточно указать в новом списке голову A и хвост L:

```
L = [5,6,7],
A = 2,
W = [A|L].
```

Свободная переменная W будет унифицирована со списком [2,5,6,7], который получен путем добавления числа 2 в начало списка [5,6,7]. Здесь важно понять, что для конструирования списка переменная W должна быть свободна. При этом переменные A и L могут быть как связаны, так и свободны. Например, можно сконструировать список, включив в него свободные переменные X и Y:

```
A = 9,
L = [],
W = [A,X,Y|L].
```

Свободная переменная *W* будет унифицирована со списком *[9, X, Y]*. Когда впоследствии переменная, например *Y*, будет связана с числом 6, то переменная *W* станет содержать список *[9, X, 6]*.

При конструировании списка составляющие его головы могут входить в него многократно. Головы надо указывать в заданном порядке:

```
L = [0,1,2],
A = 9,
B = 8,
W = [A,B,B,A|L].
```

Свободная переменная *W* будет унифицирована со списком *[9, 8, 8, 9, 0, 1, 2]*.

Операция разделения списка разделяет заданный список на голову и хвост, поэтому и называется *селектором*. Например, для разделения списка *[8, 9, 0]* достаточно его унифицировать со списком *[A|L]*, в котором голова *A* и хвост *L* являются свободными переменными:

```
[A|L] = [8,9,0].
```

В результате мы получим:

```
A=8, L=[9,0].
```

Разделить список можно на несколько голов и один хвост:

```
[A,B|L] = [8,9,0].
```

В результате мы получим:

```
A=8, B=9, L=[0].
```

На основе унификации списков можно выполнять другие операции над списками, используя различные комбинации свободных и связанных частей списков (табл. 13.3).

Таблица 13.3. Примеры унификации списков

Операция	Пояснения
<i>[1 L] = [1,2,3]</i>	<i>L = [2,3]</i>
<i>[A [2,3]] = [1,2,3]</i>	<i>A = 1</i>
<i>[A, _, B] = [1,2,3]</i>	<i>A = 1, B = 3</i>
<i>[A, B, C L] = [1,2,3]</i>	<i>A = 1, B = 2, C=3, L=[]</i>
<i>[1,B,3] = [A,2,C]</i>	<i>A = 1, B = 2, C=3</i>
<i>[1 L] = [L [2,3]]</i>	Ошибка. Переменная <i>L</i> слева обозначает список, а справа — элемент списка
<i>[A [2,B]] = [1,2,3]</i>	<i>A = 1, B = 3</i>

Таблица 13.3 (окончание)

Операция	Пояснения
$[A, A L] = [5, 5, 3]$	Проверка равенства первых двух элементов списка: $A = 5, L = [3]$
$L = [_, _, _]$	Проверка того, что список содержит ровно три элемента

13.2.1. Объявление списочных доменов

Для обработки списков необходимо объявить их домены. Объявление списочного домена состоит в указании типа данных тех элементов, из которых будет состоять список, и знака звездочки. Две звездочки указывают на то, что будет обрабатываться список списков. Вложенность списков произвольна. Вот примеры различных списочных доменов:

```
domains
  ilist = integer*.      % объявление списка целых чисел
  ill = ilist*.          % объявление списка списков целых чисел
  charl = char*.         % объявление списка символов
  strll = string**.      % объявление списка списков строк
  myBool = истина; ложь; неизвестно.    % это несписочный домен
  blist = myBool*.       % объявление списка элементов из домена myBool
```

13.3. Реализация очереди и дека

Очередь может быть легко реализована в виде базы фактов (см. разд. 20.4). Однако двустороннюю очередь (дек) реализовать базой фактов не получится, т. к. чтение фактов базы начинается с первого факта и заканчивается последним.

Если стек реализуется в виде списка, то очередь можно представить в виде двух списков. В первый список производится постановка элементов в очередь, из второго списка элементы выталкиваются. Второй список реверсивный относительно первого списка. Если второй список опустеет, то тогда надо переместить в него элементы первого списка. На рис. 13.3 показано представление очереди двумя списками. Пусть задана очередь $A_8, A_7, A_6, A_5, A_4, A_3, A_2, A_1$. Элемент A_1 поставлен в очередь первым, элемент A_8 — последним. Для представления этой очереди списками ее надо разделить на две части — например, на левую короткую часть: A_8, A_7 и правую длинную: $A_6, A_5, A_4, A_3, A_2, A_1$. Левая часть очереди представляется списком $[A_8, A_7]$. К этому списку разрешается применять только одну операцию — присоединение головы к списку. Правая часть очереди представляется реверсивным списком $[A_1, A_2, A_3, A_4, A_5, A_6]$. К правой части очереди допускается применять также только одну операцию — отсоединение головы от списка. Когда правая часть очереди опустеет, в нее следует программно перенести элементы из левой части, если таковые имеются.

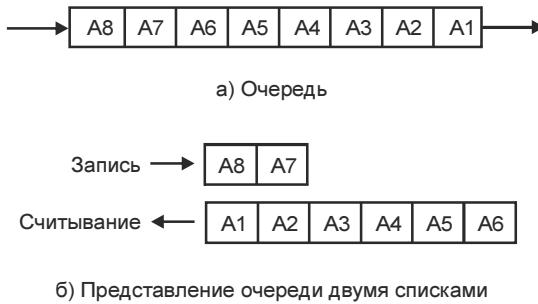


Рис. 13.3. Представление очереди

Для организации двусторонней очереди (дека) достаточно разрешить выполнять операции присоединения и отсоединения головы к обоим спискам.

13.4. Принципы рекурсивной обработки списков

Списки в Visual Prolog являются одним из основных доменов, для обработки которых используется рекурсия. Построение списка, его анализ и модификация производятся рекурсивно.

13.4.1. Построение списков из элементов

Построение списка из элементов может производиться от головы к хвосту. Для этого голову списка надо присоединять к неизвестному хвосту в голове правила, а хвост передавать в рекурсивный вызов для его дальнейшего определения. В этом случае список будет содержать элементы в таком порядке, в котором элементы подставлялись в голову списка на каждом шаге цикла. Схематично такое построение можно описать в следующем виде:

```
ген([]) :- условие_останова(), !.
ген([Элемент|Хвост]) :- получить(Элемент), ген(Хвост).
```

Вызов такого предиката должен содержать свободную переменную ген(Список). На выходе из рекурсии переменная Список будет содержать построенный список, который состоит из элементов, полученных с помощью предиката получить(Элемент).

ПРИМЕР 13.1. Построим список заданного количества случайных чисел.

```
ген(Счетчик, []) :- Счетчик=0, !.
ген(Счетчик, [Элемент|Хвост]) :- Элемент = random(),
    ген(Счетчик-1, Хвост).
```

Здесь используется дополнительный аргумент Счетчик. При каждом рекурсивном вызове он декрементируется. Когда его значение станет равным нулю, цикл останавливается. Для генерации, например, восьми случайных чисел вызов этого предиката должен быть таким: ген(8, Список).

В более лаконичной записи предикат из примера 13.1 выглядит так:

```
ген(0, []) :- !.  
ген(Счетчик, [random() | Хвост]) :- ген(Счетчик-1, Хвост).
```

ПРИМЕР 13.2. Построим список целых чисел от А до Б с шагом 1.

```
ряд(А, Б, []) :- А>Б, !.  
ряд(А, Б, [А|Хвост]) :- ряд(А+1, Б, Хвост).
```

Для генерации списка, содержащего ряд чисел от 4 до 7, вызов этого предиката должен быть таким: ряд(4, 7, Список). Предикат вернет Список = [4, 5, 6, 7]. Вот последовательность рекурсивного построения списка:

```
[4|Хвост]  
[4|[5|Хвост]]  
[4|[5|[6|Хвост]]]  
[4|[5|[6|[7|Хвост]]]]  
[4|[5|[6|[7|[]]]]]
```

Как можно заметить, на каждом витке цикла определяется голова для текущего хвоста. Таким образом, список мы строим слева направо. Когда выполняется условие останова, текущий хвост становится пустым списком.

13.4.2. Построение реверсивных списков из элементов

Для построения списка в направлении от хвоста к голове надо вначале задать хвост. Обычно его роль выполняет пустой список. Потом, на каждом витке цикла нужно добавлять голову к известному хвосту в рекурсивном вызове. На дне рекурсии, когда выполнится условие останова, мы получим готовый список. Для его передачи наверх, в начало рекурсии, надо «протащить» через всю рекурсию свободную переменную, с которой и унифицировать готовый список на дне рекурсии. Схематично такое построение можно описать в следующем виде:

```
ген(Результат, Результат) :- условие_останова(), !.  
ген(Хвост, Результат) :- получить(Элемент),  
    ген([Элемент|Хвост], Результат).
```

При рассмотренном способе генерации списка порядок следования элементов будет инверсным. Вызов такого предиката должен содержать пустой список на месте хвоста и свободную переменную на месте результата: ген([], Список). На выходе рекурсии переменная Список будет содержать построенный список.

Описанные рекурсии являются оптимизированными. Какой из этих двух способов использовать, зависит от логики решения конкретной задачи.

ПРИМЕР 13.3. Построим реверсивно список заданного количества случайных чисел.

```
ген(Счетчик, Результат, Результат) :- Счетчик=0, !.  
ген(Счетчик, Хвост, Результат) :- Элемент = random(),  
    ген(Счетчик-1, [Элемент|Хвост], Результат).
```

Для генерации, например, восьми случайных чисел, вызов этого предиката должен быть таким: `ген(8, [], Список)`.

В более лаконичной записи предикат из примера 13.3 выглядит так:

```
ген(Счетчик, Хвост, Результат) :-  
    ген(Счетчик-1, [random() | Хвост], Результат).
```

ПРИМЕР 13.4. Построим реверсивно список целых чисел от А до Б с шагом 1.

```
ряд(А, Б, Результат, Результат) :- А>Б, !.  
ряд(А, Б, Хвост, Результат) :- ряд(А+1, Б, [А|Хвост], Результат).
```

Вызов предиката `ряд(4, 7, [], Список)` вернет Список = [7, 6, 5, 4]. Вот последовательность рекурсивного построения списка:

```
[4|[]]  
[5|[4|[]]]  
[6|[5|[4|[]]]]  
[7|[6|[5|[4|[]]]]]
```

На каждом витке цикла построенный список дополняется новой головой, т. е. список собирается справа налево. Когда выполняется условие останова, окончательно построенный список передается со дна рекурсии наверх.

13.4.3. Сканирование списка

С помощью сканирования списка определяются заданные свойства его элементов. Сканирование списка заключается в пошаговом отделении его элементов и проверке какого-либо условия или выполнении действий над элементами. Для этого в голове правила от списка отделяется элемент, в теле правила производится какое-либо действие над элементом или проверка какого-либо условия. После этого осуществляется рекурсивный вызов, в который передается оставшийся хвост списка. Рекурсия останавливается, когда список опустеет. Схематично сканирование списка можно описать в следующем виде:

```
скан([Элемент|Хвост]) :- действие(Элемент), скан(Хвост).  
скан([]).
```

Схема сканирования списка практически идентична схеме его построения, за исключением потока параметров. При сканировании список задается, т. е. является входным, а при построении — выходным параметром.

ПРИМЕР 13.5. Проверка четности всех элементов списка.

```
скан([Элемент|Хвост]) :- Элемент mod 2 = 0, скан(Хвост).  
скан([]).
```

Делать отсечение перед рекурсивным вызовом в этом примере не обязательно, т. к. компилятор Visual Prolog определяет, что если разделение списка [Элемент|Хвост] произошло успешно, то список не пуст, и второе правило никогда не выполнится, следовательно, в стек адреса возврата не надо помещать адрес второго правила. Таким образом, рекурсия в примере 13.5 является оптимизированной.

13.4.4. Модификация списка

Модификация списка в Прологе означает построение нового списка из заданного. Предикат модификации списка содержит как минимум два аргумента: один аргумент — заданный список, другой — новый список. Собственно модификация осуществляется путем отделения головы от заданного списка, вычисления значения головы нового списка и подстановки ее в новый список. После этого в рекурсивном вызове указываются только хвосты обоих списков. Когда заданный список опустеет, значит, и хвост нового списка становится пустым. Схематично модификацию списка можно описать в следующем виде:

```
мод([Элемент|Хвост], [НовЭлемент|НовХвост]) :-  
    НовЭлемент = получить(Элемент), мод(Хвост, НовХвост).  
мод([], []).
```

ПРИМЕР 13.6. Построим из заданного списка целых чисел список их кубов.

```
мод([Элемент|Хвост], [НовЭлемент|НовХвост]) :-  
    НовЭлемент = Элемент ^ 3, мод(Хвост, НовХвост).  
мод([], []).
```

Или в более лаконичной записи:

```
мод([Элемент|Хвост], [Элемент ^ 3|НовХвост]) :-  
    мод(Хвост, НовХвост).  
мод([], []).
```

Эта рекурсия также является оптимизированной.

13.4.5. Синхронная обработка списков

Иногда элементы двух списков попарно связаны между собой какой-либо логической зависимостью. Например, список номеров и список фамилий или список фамилий и список дат рождения и т. д. При этом длины связанных списков одинаковы. Для выполнения действия над элементами одного списка в зависимости от значений элементов другого списка производят синхронную обработку двух списков. Такая обработка заключается в одновременном отделении на каждом витке цикла головы одного и головы другого списка. Если голова одного списка удовлетворяет условию, то выполняется какое-либо действие над головой другого списка, и хвосты обоих списков передаются в рекурсивный вызов. Если голова одного списка не удовлетворяет условию, то никакие действия не производятся, а сразу выполняется рекурсивный вызов, в который передаются хвосты списков. Схематично синхронную обработку списков можно описать в следующем виде:

```
синх([Элемент1|Хвост1], [Элемент2|Хвост2]) :-  
    тест(Элемент1), действие(Элемент2), синх(Хвост1, Хвост2).  
синх([_|Хвост1], [_|Хвост2]) :- синх(Хвост1, Хвост2).  
синх([], []).
```

Схема синхронной обработки списков подобна схеме модификации списка, за исключением потока параметров. При модификации списка второй аргумент является выходным, а при синхронной обработке оба списка заданы.

ПРИМЕР 13.7. Пусть задан список фамилий и список соответствующих годов рождения. Необходимо вывести фамилии тех людей, которые родились в диапазоне от 1990 до 1995 года.

```
синх ([Фамилия|Хвост1], [ГодРождения|Хвост2]) :-  
    ГодРождения >= 1990, ГодРождения <= 1995,  
    write(Фамилия), nl, !,  
    синх (Хвост1,Хвост2).  
синх ([_|Хвост1], [_|Хвост2]) :- синх (Хвост1,Хвост2).  
синх ([],[]).
```

13.5. Примеры рекурсивной обработки списков

Рассмотрим часто используемые предикаты, производящие рекурсивную обработку списков. Хотя некоторые из них и описаны в классе *list*, но будет полезно понять, как они устроены.

13.5.1. Определение длины списка

ПРИМЕР 13.8. Для определения длины списка достаточно отрывать на каждом витке цикла голову от списка и инкрементировать счетчик. Первый аргумент предиката *len/3* является списком, длина которого определяется. Второй аргумент — счетчик *I*, который инкрементируется на каждом витке цикла. При вызове предиката этот счетчик равен нулю. На дне рекурсии список станет пустым, а счетчик будет содержать значение длины списка. Третий аргумент — возвращаемый параметр. Он, не изменяясь, передается через весь цикл, чтобы «вытащить» со дна рекурсии длину списка. Рекурсия *len/3* является оптимизированной, хотя в первом правиле и не используется отсечение. В данном случае компилятор Visual Prolog самостоятельно оптимизирует рекурсию на основе анализа первого аргумента. В первом правиле этот аргумент является непустым списком, а во втором правиле — пустым. Следовательно, если выполнилось первое правило, то второе никогда не выполнится, поэтому запоминать в стеке его адрес не надо.

```
implement main  
    open core,console  
class predicates  
    len : (integer*, unsigned, unsigned [out]).  
clauses  
    len([_|L],I,N) :-          % отрываем голову от списка  
        len(L,I+1,N).          % инкрементируем счетчик I  
    len([],N,N).              % останов рекурсии, если список пуст  
run():-len([5,6,7,8],0,N),  
    write(N),                 % N = 4  
    _ = readline().  
end implement main  
goal  
    console::run(main::run).
```

Следующие правила показывают эту же оптимизированную рекурсию, но выполненную неэффективно.

```
len([], N, N).           % останов рекурсии, если список пуст
len([_|L], I, N) :-      % отрываем голову от списка
    len(L, I+1, N).     % инкрементируем счетчик I
```

Неэффективность этой программы заключается в том, что на каждом витке цикла вначале проверяется условие останова, т. е. пустота списка, и только после этого, если список не пуст, выполняется второе правило. Поэтому, какова длина списка, столько раз это условие будет ложно, и только когда список опустеет, оно станет истинным. Это ведет к неоправданному увеличению количества выполняемых операций. Для того чтобы оптимизированную рекурсию сделать еще и эффективной, надо располагать правила останова рекурсии по возможности последними, а рекурсивные правила — первыми, как это сделано в данном примере.

ПРИМЕР 13.9. Так как предикат `len/3` из примера 13.8 имеет один выходной аргумент, то этот предикат выгодно переписать в виде функции `len/2 → Length`, где первый аргумент — список, а второй — счетчик. Значение функции — длина списка.

```
implement main
  open core, console
class predicates
  len : (integer*, unsigned Counter) -> unsigned Length.
clauses
  len([_|L], I) = len(L, I+1). % отрываем голову и инкрементируем
                                % счетчик
  len([], I) = I.             % останов рекурсии, если список пуст
run():-write(len([5,6,7,8],0)), % длина списка 4
            _ = readline().
end implement main
goal
  console::run(main::run).
```

На каждом витке цикла голова отрывается от списка и счетчик инкрементируется. Когда список опустеет, то второй аргумент — счетчик, будет содержать длину списка. Чтобы значение счетчика «вытащить» со дна рекурсии наверх, мы унифицировали функцию с ее рекурсивным вызовом. Поэтому на дне рекурсии достаточно унифицировать счетчик с функцией.

Следующие правила показывают такую же функцию, но неоптимизированную:

```
len([_|L], I) = len(L, I)+1. % отрываем голову и инкрементируем функцию
len([], I) = I.             % останов рекурсии, если список пуст
```

Неоптимизированная функция синтаксически отличается только тем, что операция инкремента вынесена наружу и применяется не к счетчику, а к значению функции. При выполнении такой рекурсии размер стека будет расти. Поэтому здесь надо быть очень внимательным и не допускать по возможности выноса операций, применяемых к аргументам рекурсивной функции, наружу и применять их к самой функции.

13.5.2. Построение списка

ПРИМЕР 13.10. Построим список целых чисел от N до M с шагом 1. Предикат, совершающий такую работу, должен содержать три аргумента: $\text{gen}(N, M, \text{List})$. Первые два — входные, они определяют нижнюю и верхнюю границы списка. Третий аргумент — собственно выходной список.

```
implement main
    open core,console
class predicates
    gen : (integer From, integer UpTo, integer* [out]). 
clauses
    gen(N,M, [N|L]) :- N<=M, !,           % при N<=M присоединяем голову
                      % к списку
    gen(N+1,M,L) .                         % инкрементируем N в рекурсивном
                                             % вызове
    gen(_,_,[]).                           % если N>M, то список является пустым
run() :- gen(-3,2,L),
        write(L),                          % вывод [-3,-2,-1,0,1,2]
        _ = readline().
end implement main
goal
    console::run(main::run).
```

На каждом витке цикла левая граница N служит головой списка, а в рекурсивный вызов передается хвост списка и инкремент N . Когда N превысит M , то цикл останавливается, и хвост списка унифицируется с пустым списком.

Предикат $\text{gen}(N, M, \text{List})$ можно переписать в виде функции:

```
implement main
    open core,console
class predicates
    gen : (integer From, integer UpTo) -> integer* List.
clauses
    gen(N,M) = [N|gen(N+1,M)] :- N<=M, !.
    gen(_,_) = [].
run() :- write(gen(-3,2)),          % вывод [-3,-2,-1,0,1,2]
        _ = readline().
end implement main
goal
    console::run(main::run).
```

ПРИМЕР 13.11. Построим список целых чисел от N до M с шагом D . Для этого предикат должен содержать четыре аргумента: $\text{gen}(N, M, D, \text{List})$. Первые три — входные, они определяют нижнюю и верхнюю границу списка, а также шаг D . Четвертый аргумент — выходной список.

```
implement main
    open core,console
```

```

class predicates
    gen : (integer Min, integer Max, integer Step, integer* [out]) .
clauses
    gen(N,M,D, [N|L]) :- N<=M, !,
        gen(N+D,M,D,L) .
    gen(_,_,_, []) .
    run():-gen(-5,5,2,L),
        write(L),      % вывод [-5,-3,-1,0,1,3,5]
        _ = readline() .
end implement main
goal
    console::run(main::run) .

```

Этот предикат отличается от предиката из примера 13.10 тем, что вместо инкремента переменной N к ней прибавляется шаг D.

ПРИМЕР 13.12. Другой вид построения списков — построение списков заданной длины. Построим список из 8 членов арифметической прогрессии с шагом 5, начиная с первого элемента 2.

```

implement main
    open core,console
class predicates
gen:(integer From, integer Step, unsigned Count,
    integer* [out]) .
clauses
gen(_,_,0, []) :! .
gen(N,D,I, [N|L]) :-gen(N+D,D,I-1,L) .
run():-gen(2,5,8,L),
    write(L),      % вывод [2,7,12,17,22,27,32,37]
    _ = readline() .
end implement main
goal
    console::run(main::run) .

```

13.5.3. Соединение списков

ПРИМЕР 13.13. Соединим два списка в новый список, который содержит сначала элементы первого списка, а следом за ними элементы второго списка. Для этого на каждом шаге цикла будем копировать элемент первого списка в голову выходного списка. Когда первый список опустеет, то второй список окажется хвостом результата.

```

implement main
    open core,console
class predicates
append : (integer* List1, integer* List2, integer* Result)
procedure (i,i,o) .

```

```

clauses
    append([N|L1], L2, [N|L]) :- append(L1, L2, L).
    append([], L, L).
    run() :- append([1,2], [3,4,5], L),
        write(L), nl, % [1,2,3,4,5]
        _ = readline().
end implement main
goal
    console::run(main::run).

```

Других способов соединения списков нет. Если бы мы всегда могли узнать адрес хвоста списка, без вытаскивания из него всех элементов, то соединить списки можно было бы быстро — простым назначением адресу хвоста первого списка значения адреса первого элемента второго списка. Но так как список по определению является рекурсивной структурой, то для определения адреса хвоста первого списка надо в рекурсии пройти по всем элементам, что и демонстрирует пример 13.13.

Примечательной особенностью предиката соединения списков является возможность его использования с различными шаблонами потоков (табл. 13.4 и 13.5).

Таблица 13.4. Варианты использования предиката соединения двух списков `append`

Операция	Режим детерминизма	Шаблон потоков
Найти соединение двух списков	procedure	(i,i,o)
Равно ли соединение двух списков третьему?	determ	(i,i,i)
Попытаться найти второй список при известном первом и третьем списке	determ	(i,o,i)
Попытаться найти первый список при известном втором и третьем списке	nondeterm	(o,i,i)
Найти все возможные значения двух списков, соединение которых задано третьим списком	multi	(o,o,i)

Таблица 13.5. Примеры использования предиката соединения двух списков `append`

Шаблон потоков	Вызов предиката	Результат
(i,i,o)	append([1,2], [3,4,5], L)	L=[1,2,3,4,5]
(i,i,i)	append([1,2], [3,4], [1,2,3,4])	Успех
(i,i,i)	append([1,2], [3,4], [1,2,3])	Неуспех
(i,o,i)	append([1,2], L, [1,2,3,4])	L=[3,4]
(i,o,i)	append([1,2], L, [1,3])	Неуспех
(o,i,i)	append(L, [3,4], [1,2,3,4])	L=[1,2]
(o,i,i)	append(L, [3,4], [1,2,3])	Неуспех

Таблица 13.5 (окончание)

Шаблон потоков	Вызов предиката	Результат
(o,o,i)	append(L1,L2,[1,2,3])	L1=[1,2,3], L2=[] L1=[1,2], L2=[3] L1=[1], L2=[2,3] L1=[], L2=[1,2,3]

13.5.4. Реверс списка

ПРИМЕР 13.14. Задача обращения (реверса) списка на Прологе решается путем построения списка от заданного хвоста к голове. Заданным хвостом при вызове является пустой список. Элементами для такого построения являются элементы исходного списка, которые по одному отсоединяются на каждом витке цикла. Когда исходный список опустеет, то рекурсия будет содержать реверсивный список. Для того чтобы вытащить его со дна рекурсии наверх, надо унифицировать его со свободной переменной, которую следует «протащить» через всю рекурсию до дна.

```

implement main
    open core,console
class predicates
    rev : (integer* List, integer* Tail, integer* Result [out]).
clauses
    rev([N|L1],L2,L) :- rev(L1,[N|L2],L).
    rev([],L,L).
    run() :- rev([1,2,3,4,5],[],L),      % L=[5,4,3,2,1]
            write(L),
            _ = readline().
end implement main
goal
    console::run(main::run).

```

Для реверса списка [1,2,3,4,5] надо сделать вызов `rev([1,2,3,4,5],[],L)`. Первым аргументом является исходный список, вторым — заданный хвост со значением [], в третьем аргументе возвращается результат — обращенный список.

13.6. Ввод/вывод списков целиком

13.6.1. Терминальный ввод/вывод списков

Ввод/вывод списков можно делать целиком или поэлементно — в цикле. Для ввода/вывода всего списка надо объявить необходимый списочный домен для переменной, в которую производится ввод списка. При наборе списка на клавиатуре необходимо заключить список в квадратные скобки — например, так: [1,2,3,4].

ПРИМЕР 13.15. Консольный ввод/вывод списка.

```

implement main
    open core,console
domains
    ilist = integer*.           % объявление списка целых чисел
clauses
    run() :- hasDomain(ilist,X), % объявление домена для переменной X
            X=read(),          % чтение списка целиком
            write(X),nl,        % вывод списка целиком
            clearInput(),       % очистка буфера клавиатуры
            _ = readline().     % ожидание <Enter>
end implement main
goal
    console::run(main::run).

```

Очистка буфера клавиатуры нужна для того, чтобы корректно работал предикат `_ = readline()`. Если очистку не делать, то код клавиши `<Enter>`, находящийся в буфере после операции ввода списка, будет воспринят предикатом ожидания ввода с клавиатуры, и после вывода списка консольное окно сразу закроется.

13.6.2. Файловый ввод/вывод списков

Список можно читать из текстового файла как строку и преобразовывать в список. Например, в файле "file.txt" содержится строка "[1,2,3,4]". Эта строка будет считана в переменную `Text`. После преобразования с помощью предиката `toTerm/2` строка будет преобразована в список `[1,2,3,4]`. Домен `ilist` должен быть определен как список целых чисел `integer*`.

```

Text = file::readString("file.txt"), % чтение списка как строки
List = toTerm(ilist,Text),          % преобразование строки в список

```

13.6.3. Потоковый ввод/вывод списков

Если файл открыть как поток ввода, то списки можно читать из такого файла целиком с помощью предиката чтения терма. Для записи списка в файл необходимо открыть файл как поток вывода:

```

In = inputStream_file::openFile("11.txt"), % дескриптор ввода
Out = outputStream_file::create("22.txt"), % дескриптор вывода
hasDomain(ilist,L),                      % объявление типа переменной
L=In:read(),                            % чтение списка из файла
Out:write([0|L]),                        % запись в файл списка с добавлением нуля
In:close(),                             % закрываем поток ввода
Out:close(),                           % закрываем поток вывода

```

Если файл "11.txt" содержит список `[1,2,3,4]`, то после выполнения программы файл "22.txt" будет содержать список `[0,1,2,3,4]`.

13.7. Поэлементный ввод/вывод списков

13.7.1. Поэлементный ввод/вывод списков в цикле с откатом

Если файл содержит однотипные элементы, разделенные пробелами, табуляцией или кодами клавиши <Enter>, то его содержимое легко собрать в список с реверсивным порядком элементов так, как показано в следующем примере.

ПРИМЕР 13.16. Пусть файл "aa.txt" содержит последовательность целых чисел, которые разделены пробелами 1 2 3 4, в кодировке ANSI-866. Приведенная далее программа собирает эти элементы в реверсивный список [4,3,2,1], который выведет на экран, а также запишет в файл "bb.txt" исходную последовательность элементов, разделенных, например, запятыми: 1,2,3,4,.

```

implement main
  open core,console
domains
  ilist = integer*.      % ilist - список целых чисел
class facts
  list : ilist := [].    % факт-переменная, в которую
                         % собирается список
clauses
run() :-
  In = inputStream_file::openFile("aa.txt",stream::ansi(866)),
  Out = outputStream_file::create("bb.txt"),
  hasDomain(integer,A),    % объявление домена для переменной A
  ( std::repeat(),
    % начало цикла
    A=In:read(),
    % чтение элемента из файла aa.txt
    list:=[A|list],
    % присоединение элемента к списку
    Out:write(A,","),
    % вывод элемента и знака запятой
    % в файл bb.txt
    % конец файла?
    In:endOfStream(),!;
    succeed() ),
    % предикат run/0 должен быть процедурой
    write(list),
    % вывод реверсивного списка на экран
    In:close(),
    % закрываем поток ввода
    Out:close(),
    % закрываем поток вывода
    _ = readline().
end implement main
goal
  console::run(main:::run).

```

Сборка элементов производится в факт-переменную list. Предикат In:endOfStream/0 будет успешен, когда указатель файла достигнет конца файла, иначе будет вызван откат.

13.7.2. Поэлементный ввод/вывод списков в рекурсивном цикле

Собрать список из элементов, хранящихся в файле, можно и с помощью рекурсии. При этом появляется возможность собрать список, как с исходным порядком элементов, так и с реверсивным.

ПРИМЕР 13.17. Здесь приведена программа, которая с помощью рекурсивного предиката `iolist/4` читает целые числа, хранящиеся в файле "aa.txt" в кодировке ANSI-866, и собирает их в реверсивный список. Кроме того, рекурсивный предикат записывает в файл "bb.txt" элементы списка, разделяя их запятыми.

```
implement main
    open core,console
domains
    ilist = integer*.           % ilist - список целых чисел
class predicates
    iolist : (inputStream, outputStream, ilist, ilist [out]). 
clauses
    iolist(In,_,List,List):-In:endOfStream(),!.  % останов рекурсии
    iolist(In,Out,Temp,List) :-
        A=In:read(),             % чтение элемента из файла aa.txt
        Out:write(A,","),
        iolist(In,Out,[A|Temp],List).
run():-
    In = inputStream_file::openFile("aa.txt",stream::ansi(866)),
    Out = outputStream_file::create("bb.txt"),
    iolist(In,Out,[],List),
    write(List),              % вывод списка на экран
    In:close(),               % закрываем поток ввода
    Out:close(),              % закрываем поток вывода
    _ = readline().
end implement main
goal
    console::run(main::run).
```

Если файл "aa.txt" содержит целые числа, разделенные пробелами — например, 1 2 3 4, то программа соберет их в список [4,3,2,1], который выведет на экран, а также запишет в файл "bb.txt" исходную последовательность элементов, разделенных запятыми: 1,2,3,4,.

ПРИМЕР 13.18. Для сборки списка с тем же порядком следования элементов, что и в файле "aa.txt", достаточно переписать рекурсию следующим образом:

```
implement main
    open core,console
domains
    ilist = integer*.           % ilist - список целых чисел
class predicates
    iolist : (inputStream, outputStream, ilist [out]).
```

```

clauses
  iolist(In,_,[]):=In:endOfStream(),!.
  iolist(In,Out,[A|List]):-
    A=In:read(),
    Out:write(A,","),
    iolist(In,Out,List).
run():-
  In = inputStream_file::openFile("aa.txt",stream::ansi(866)),
  Out = outputStream_file::create("bb.txt"),
  iolist(In,Out,List),
  write(List),           % вывод списка на экран
  In:close(),            % закрываем поток ввода
  Out:close(),           % закрываем поток вывода
  _ = readline().
end implement main
goal
  console::run(main:::run).

```

13.8. Предикат выборки элементов списка

Visual Prolog содержит предикат выборки элементов x списка L по одному (см. разд. 7.6). Этот предикат выражается в инфиксной форме записи: $x \text{ in } L$.

В случае, когда переменная x свободная, предикат `in` является недетерминированным и на откатах возвращает через эту переменную все элементы списка L по одному, подобно тому, как это делает функция $x = list::getMember_nd(L)$. Например, вызов:

```
X in [4,6,8], write(X^2," "), fail; succeed.
```

выведет квадраты элементов указанного списка:

```
16   36   64.
```

В случае, когда переменная x связана, предикат `in` является детерминированным и будет успешен, если значение x принадлежит списку. В противном случае он будет неуспешен. Такое поведение предиката `in` эквивалентно поведению предиката `list::isMember(X,L)`. Например, вызов:

```
6 in [4,6,8,6], write("принадлежит"); write("не принадлежит").
```

выведет сообщение: принадлежит.

Предикат `6 in [4,6,8,6]` работает следующим образом. Сначала из списка выбирается первый элемент 4 и унифицируется с элементом 6. Унификация, естественно, неуспешна, поэтому выбирается следующий элемент: 6. В этом случае унификация успешна, и предикат `in` также успешен. При этом в стек не будет помещен адрес возврата, в связи с чем, в случае отката, вторая шестерка в списке `[4,6,8,6]` искастся не будет.

13.9. Коллекtor списков

Visual Prolog содержит средство построения списков, которое называется *коллекtor списков*. Нотация этого средства аналогична тому, что ввел Георг Кантор для определения множества. Кантор определил множество M как совокупность элементов x , обладающих свойством $P(x)$:

$$M = \{x \mid P(x)\}.$$

В Visual Prolog такое определение записывается в виде

```
M = [X || p(X)].
```

Здесь использована двойная вертикальная черта, т. к. одна вертикальная черта занята в операции представления списка в виде головы и хвоста. Свойство $p(X)$ в коллекторе списков должно быть недетерминированным. Его роль могут выполнять предикаты и факты. Рассмотрим варианты составления различных запросов к внутренней базе данных.

ПРИМЕР 13.19. Пусть имеется база данных о фамилиях людей и годах их рождения. Необходимо выбрать из базы данных общие сведения. В теле правила `run` задано два запроса к базе. Обратите внимание, что область видимости имен переменных, используемых в запросах, ограничивается не собственно запросом, а находится в рамках всего тела предиката `run`. Поэтому в каждом запросе имена переменных различны.

```
implement main
  open core,console
class facts
  person:(string Фамилия, unsigned ГодРождения).
clauses
  person("Зуй",1995).
  person("Диев",1990).
  person("Перумов",1997).
  person("Родин",1988).
  person("Варгин",1986).
run() :-
  write([X||person(X,_)]),nl,          % собрать в список все фамилии
  write([Y||person(_,Y)]),nl,          % собрать в список все годы
                                         % рождения
  _ = readline().
end implement main
goal
  console::run(main::run).
```

На экран будет выведено:

```
["Зуй", "Диев", "Перумов", "Родин", "Варгин"]
[1995, 1990, 1997, 1988, 1986]
```

ПРИМЕР 13.20. Выберем из базы данных сведения, удовлетворяющие каким-либо условиям.

```

implement main
    open core,console
class facts
    person:(string Фамилия, unsigned ГодРождения).
clauses
    person("Зуй",1995).
    person("Диев",1990).
    person("Перумов",1997).
    person("Родин",1988).
    person("Варгин",1986).
run():-write([X|person(X,Y), (Y<1990;Y>1995)]),nl,
    write([B|person(A,B),C=frontChar(A),C>='Б',C<='Й']),nl,
    _=readline().
end implement main
goal
    console::run(main:::run).

```

В первом запросе в список собираются фамилии тех людей, которые родились до 1990 года или после 1995 года. Во втором запросе собираются в список годы рождения тех людей, у которых первая буква их фамилии находится между буквой "Б" и буквой "Й" включительно. На экран будет выведено:

```

["Перумов", "Родин", "Варгин"]
[1995, 1990, 1986]

```

Рассмотрим варианты использования коллектора списков для обработки заданных списков.

ПРИМЕР 13.21. Пусть имеется список `[1,3,4,7,8,9]`. Необходимо найти подсписок этого списка, элементы A которого удовлетворяют условию `A>3` и `A<=8`. Кроме того, необходимо составить список всех пар элементов заданного списка, сумма которых равна 10. Для решения этой задачи воспользуемся объявленным в классе `core` кортежем `tuple/2`. С помощью этих кортежей можно обрабатывать не только пары элементов, но и тройки, четверки и т. д. вплоть до 12 элементов.

```

implement main
    open core,console
clauses
    run():-L=[1,3,4,7,8,9],
        write([X || X in L,X>3,X<=8]),nl,
        write([tuple(A,B) || A in L, B in L,A<B, A+B=10]),nl,
        _=readline().
end implement main
goal
    console::run(main:::run).

```

Во втором коллекторе предикат `A<B` не только фильтрует одинаковые элементы списка `L`, но и упорядочивает пары чисел, чтобы не было повторов решений, кото-

рые отличаются только порядком аргументов. В результате на экран будет выведено:

```
[4,7,8]
[tuple(1,9),tuple(3,7)]
```

ПРИМЕР 13.22. Пусть имеются два списка: [1,3,4,7,8,9] и [2,4,6,9,10]. Необходимо найти их пересечение, т. е. построить список, элементы которого принадлежат и первому, и второму спискам. Кроме этого, надо собрать все пары чисел, содержащие число A, из первого списка, и число B из второго списка, для которых выполняется условие делимости A на B без остатка.

```
implement main
  open core,console
clauses
  run():-L=[1,3,4,7,8,9],
    W=[2,4,6,9,10],
    write([X||X in L,Y in W,X=Y]),nl,
    write([tuple(A,B) || A in L, B in W,A>=B, A mod B=0]),nl,
    _=readline().
end implement main
goal
  console:::run(main:::run).
```

На экран будет выведено:

```
[4,9]
[tuple(4,2),tuple(4,4),tuple(8,2),tuple(8,4),tuple(9,9)]
```

13.10. Представление базы фактов списками фактов

Пролог дает доступ к базе фактов по чтению только сверху вниз, т. е. с начала в конец. Доступ по записи доступен как сверху (в начало БД) с помощью предиката asserta, так и снизу (в конец БД) с помощью предиката assertz. Пролог не дает доступ к факту, расположенному в произвольной позиции. Однако в некоторых задачах существует надобность в таком доступе. Рассмотрим один способ чтения, удаления, модификации и добавления фактов в произвольной позиции базы фактов. Для этого факты БД собираются в список фактов, и впоследствии данные хранятся одновременно в двух формах представления: в базе фактов и в списке фактов. Такая организация представления данных соединяет достоинства базы фактов и списка фактов. При обработке данных с помощью цикла с откатом задействуется база фактов. При рекурсивной обработке используется список фактов, который позволяет получить доступ к факту по его индексу.

13.10.1. Когерентность базы фактов и списка фактов

Когерентность фактов — идентичность данных в базе фактов и в списке фактов. После модификации или списка фактов, или базы фактов когерентность нарушается.

ся. Для восстановления когерентности данных устаревший образ данных в хранилище воссоздается из образа, содержащего истинные данные.

Существует два протокола поддержки когерентности. Первый заключается в безотлагательной записи данных в тот образ, который устарел. При больших размерах БД такой протокол требует дополнительных вычислительных затрат. Это связано с необходимостью воссоздания всей базы фактов или всего списка фактов при изменении даже одного факта. Согласно второму протоколу восстановление когерентности данных откладывается до тех пор, пока в этом не возникнет необходимость. К примеру, после рекурсивной обработки списка фактов когерентность можно не восстанавливать, если следующий доступ к данным опять будет через список фактов. В этом случае восстановление когерентности будет отложено до момента, когда потребуется доступ к базе фактов. Таким образом, механизм поддержки когерентности запускается при смене образа данных, к которому происходит доступ.

13.10.2. Домен фактов внутренней базы данных

Пусть объявлена база данных с именем aaa:

```
class facts - aaa
    f(string Имя, string Фамилия, string Отчество).
    s(string Фамилия, string Образование).
```

Имя базы данных aaa является доменом, которому принадлежат все факты этой БД. Список фактов этой БД описывается списочным доменом aaa*.

13.10.3. Создание списка фактов внутренней базы данных

Создать список фактов f/3 можно с помощью коллектора списков:

```
[f(A,B,C) || f(A,B,C)]
```

Для создания списка фактов s/2 надо воспользоваться вторым коллектором:

```
[s(A,B) || s(A,B)]
```

Для сборки фактов БД f/3 и s/2 в один список следует объединить вызов всех фактов в одном коллекторе посредством одной переменной — например, Z:

```
[Z || f(A,B,C), Z=f(A,B,C); s(D,E), Z=s(D,E)]
```

13.10.4. Восстановление внутренней базы данных из списка фактов

При восстановлении БД с именем aaa из списка фактов L следует в первую очередь удалить факты БД с помощью предиката:

```
retractFactDb(aaa)
```

а потом добавить в БД факты из списка L с помощью высокоуровневого предиката:

```
forAll(L, { (X) :- assert(X) })
```

в котором используется анонимный предикат { (X) :- assert(X) }, выполняющий операцию добавления assert(X) для каждого элемента списка L.

13.10.5. Предикаты преобразования внутренней базы данных в список фактов и обратно

В случае, когда список фактов является аргументом некоторого предиката, то домен этого аргумента надо определить явно, с помощью списочного домена.

ПРИМЕР 13.23. Для описанной базы данных с именем aaa объявим функцию tolist, которая возвращает список фактов базы. Этот список принадлежит домену aaa*. Также объявим предикат toDB, воссоздающий БД aaa из списка фактов. Теперь мы можем описать предикаты преобразования внутренней БД:

```
implement main
    open core, console, list
class facts - aaa
    f: (string Фамилия, integer ГодРождения) .
    s: (string Фамилия, string Образование) .
class predicates
    tolist:() -> aaa*.
    toDB:(aaa*).
clauses
    f("Зуев",1981) . f("Камов",1998) . f("Рудь",1954) .
    s("Зуев","среднее") . s("Диев","высшее") .
    tolist() = [Z || f(A,B) , Z=f(A,B) ; s(C,D) , Z=s(C,D)] .
    toDB(L) :- retractFactDb(aaa),
               forAll(L, { (X) :- assert(X) }) .
run():-
    L = tolist(),                               % преобразование БД в список
    write("Список: ",L), nl, nl,
    F = nth(3,L),                                % получение четвертого факта БД
    write("Элемент №4: ",F), nl, nl,
    L1 = filteredMap(L, { (A)=A:-A=f(_,Y), Y<1988 } ),      % фильтрация
    write("Фильтрация: ",L1), nl, nl,
    toDB(L1),                                     % преобразование списка в БД
    assert(s("Таран","высшее")),      % добавление факта в конец БД
    L2 = tolist(),
    write("Итог: ",L2),                           % просмотр списка БД
    _ = readline().
end implement main
goal
    console::run(main::run) .
```

В результате мы получим:

Список: [f ("Зуев", 1981), f ("Камов", 1998), f ("Рудь", 1954),
s ("Зуев", "среднее"), s ("Диев", "высшее")]

Элемент №4: s ("Зуев", "среднее")

Фильтрация: [f ("Зуев", 1981), f ("Рудь", 1954)]

Итог: [f ("Зуев", 1981), f ("Рудь", 1954), s ("Таран", "высшее")]

ГЛАВА 14



Параметрический полиморфизм

14.1. Полиморфизм параметров предикатов

Наряду с категориальным полиморфизмом, который кратко описан в главе 4, в Visual Prolog используется *параметрический полиморфизм*. Параметрический полиморфизм служит для указания в предикатах параметров различного типа.

Пусть у нас есть предикат `isMember/2` определения принадлежности целого числа списку целых чисел. Первый аргумент здесь целое число, второй — список целых чисел. Предикат `isMember/2` является детерминированным, поскольку элемент может и не принадлежать списку:

```
class predicates
    isMember : (integer, integer*) determ.
clauses
    isMember(X, [X|_]) :- !.
    isMember(X, [_|L]) :- isMember(X, L).
```

Для того чтобы предикат работал не только с целыми числами, но и со строками, и с вещественными числами, можно дополнить раздел объявления предиката явным указанием нужных типов. При этом раздел `clauses` останется тем же:

```
class predicates
    isMember : (integer, integer*) determ.
    isMember : (string, string*) determ.
    isMember : (real, real*) determ.
clauses
    isMember(X, [X|_]) :- !.
    isMember(X, [_|L]) :- isMember(X, L).
```

Однако если в следующий раз нам потребуется работать со списком символов или со списком списков строк, мы должны опять расширить раздел объявлений. Чтобы так не поступать, следует использовать параметрический полиморфизм. Для этого в объявлении полиморфного предиката вместо явного указания типа аргумента можно указать формальное имя. Давайте в качестве типа аргумента укажем, к примеру, формальное имя `Elem`. Тогда список будет принадлежать домену `Elem*`. Во время выполнения программы вместо `Elem` программа может использовать любой тип или домен:

```

class predicates
    isMember : (Elem, Elem*) determ.
clauses
    isMember(X, [X|_]) :- !.
    isMember(X, [_|L]) :- isMember(X, L).

```

Областью действия имени `Elem` является объявление предиката, в котором это имя используется. При объявлении другого полиморфного предиката нам ничто не мешает опять указать имя `Elem`, если оно отражает суть параметра.

Достоинством таких полиморфных предикатов является их завершенность. Единожды написав, впоследствии не надо доопределять их объявление новыми типами данных.

14.2. Полиморфизм параметров доменов

Параметрический полиморфизм относится не только к параметрам предикатов — с его помощью можно создавать полиморфные домены. *Полиморфный домен* — это такой домен, в определении которого вместо типов параметров указываются формальные имена типов параметров. Благодаря этому, при использовании такого домена в объявлении какого-либо предиката, типы его аргументов можно связать с типами других полиморфных доменов этого же предиката. Имена типов параметров заключаются в фигурные скобки. Например, полиморфный домен `polymorDom` от двух доменов `A` и `B` записывается так:

```

domains
    polymorDom{A,B} = ddd(A*, B*, A, B).

```

Эту запись следует понимать следующим образом. Если в каком-либо контексте программы вместо формального имени `A` подставляется параметр типа `integer`, а вместо имени `B` подставляется параметр типа `string`, то в указанном контексте полиморфный домен принимает вид:

```
ddd(integer*, string*, integer, string).
```

В другом контексте при подстановке `string` вместо `A` и `char*` вместо `B` этот же домен может принять вид:

```
ddd(string*, char**, string, char*).
```

Типы `A` и `B` могут быть как различны, так и одинаковы. Областью действия имени типа является объявление домена, в котором это имя используется. При объявлении другого полиморфного домена нам ничто не мешает опять указать это же имя.

Для оценки преимуществ полиморфных доменов рассмотрим программу без использования полиморфных доменов и с их использованием. Пусть логика некоторой задачи требует многократного обращения к длине списка целых чисел и к последнему элементу списка. В этом случае будет разумно объявить новый домен `my_list`, в котором хранить список вместе с его длиной и последним элементом. Это нам даст практически мгновенный доступ к длине списка и последнему элементу:

```
domains
    my_list = my_list(integer*, integer Last, unsigned Length);
        empty_list().
```

Все операции будем производить уже с доменом `my_list`, а не со списком. Опишем операции: добавление элемента, получение длины списка и получение последнего элемента списка.

```
class predicates
    add_elem:(my_list, integer Elem) -> my_list.
    get_last:(my_list) -> integer determ.
    get_length:(my_list) -> unsigned determ.
clauses
    add_elem(empty_list,E) = my_list([E],E,1).
    add_elem(my_list(L,Last,Length),E) =
        my_list([E|L],Last,Length+1).
    get_last(my_list(_,Last,_)) = Last.
    get_length(my_list(_,_,Length)) = Length.
```

Для проверки этого предиката можно воспользоваться целью:

```
run():-L = add_elem(empty_list,8),
    L1 = add_elem(L,12),
    L2 = add_elem(L1,9),
    write(L2),nl,                      % my_list([9,12,8],8,3)
    write(get_length(L2)),nl,            % 3
    write(get_last(L2)),                % 8
    _ = readline(),!;
    _ = readline().
```

Эта программа работает со списками целых чисел. При необходимости обработки списков вещественных чисел, строк и т. п. нам понадобится объявить новые домены и предикаты. Дело в том, что тип элемента списка скрыт в определении домена `my_list` и невидим извне. Поэтому этот домен невозможно увязать с другими доменами предиката.

Использование полиморфных доменов позволяет легко решить возникшую проблему добавления новых типов данных. Для этого при объявлении домена в фигурных скобках указывается формальное имя типа. Пусть таковым является имя `A`. Все упоминания ранее используемого типа `integer` нам надо заменить именем `A`:

```
domains
    my_list{A} = my_list(A*, A Last, unsigned Length);
        empty_list().
```

Аналогичные замены типа `integer` на формальное имя `A` надо сделать в объявлении предикатов:

```
class predicates
    add_elem:(my_list{A}, A Elem) -> my_list{A}.
    get_last:(my_list{A}) -> A determ.
    get_length:(my_list{A}) -> unsigned determ.
```

Сделанная нами модификация объявлений домена и предикатов позволяет обрабатывать списки элементов произвольного типа. При этом раздел определения предикатов остался без изменений. Для проверки полиморфного домена можно воспользоваться целью:

```
run() :- L = add_elem(empty_list, 8),
         W = add_elem(empty_list, "qwe"),
         L1 = add_elem(L, 12),
         L2 = add_elem(L1, 9),
         W1 = add_elem(W, "678"),
         write(L2), nl,
         write(W1), nl,
         write(get_length(L2)), nl,      % длина L2 равна 3
         write(get_last(L2)), nl,        % последний элемент 8
         write(get_length(W1)), nl,      % длина W1 равна 2
         write(get_last(W1)), nl,        % последний элемент "qwe"
         _ = readline(), !;
         _ = readline().
```

В этой цели предикаты `add_elem/2`, `get_length/1` и `get_last/1` работают как со списком целых чисел, так и со списком строк.

ПРИМЕР 14.1. Рассмотрим задачу упаковки и распаковки списков. Упаковка двух списков заключается в построении такого списка, каждый i -й элемент которого является парой i -х элементов исходных списков. К примеру, упаковкой двух списков `[1,2,3]` и `['a','б','в']` является список:

```
[tuple(1, 'a'), tuple(2, 'б'), tuple(3, 'в')]
```

Здесь доменом `tuple/2` обозначен кортеж двух элементов. Распаковка такого списка кортежей приводит к получению двух исходных списков: `[1,2,3]` и `['a','б','в']`. Для реализации такой упаковки и распаковки списков надо использовать полиморфный домен `tuple{A,B}`. В этом домене именем A назван тип элементов первого списка, а именем B назван тип элементов второго списка.

Далее приведена программа для упаковки и распаковки списков:

```
implement main
    open core, console
domains
    tuple{A,B} = tuple(A,B).
class predicates
    zip : (A* List1, B* List2) -> tuple{A,B}* determ.
    unzip : (tuple{A, B}*, A* List1 [out], B* List2 [out]). 
clauses
    zip([A|L1], [B|L2]) = [tuple(A,B) | zip(L1,L2)].
    zip([], []) = [].

    unzip([], [], []).
    unzip([tuple(A,B)|L], [A|L1], [B|L2]) :- unzip(L,L1,L2).
```

```

run() :-
    List = zip([1,2,3],['а','б','в']),
    write(List), nl,           % [tuple(1,'а'), tuple(2,'б'), tuple(3,'в')]
    unzip(List, L1, L2),
    write(L1, L2), nl,         % [1,2,3] ['а','б','в']
    _ = readline(), !;
    write("Различная длина списков"),
    _ = readline().
end implement main
goal
    console::run(main:::run).

```

Функция `zip/2` осуществляет упаковку двух списков, а предикат `unzip/2` делает распаковку списка.

14.2.1. Полиморфный списочный домен класса core

В классе `core` объявлен полиморфный списочный домен:

```

domains
    list{Elem} = Elem*.

```

Домен `list{Elem}` удобно использовать в тех случаях, когда применение домена `Elem*` не разрешено синтаксисом языка. В подобных случаях вместо домена `Elem*` следует задействовать домен `core::list{Elem}` или просто `list{Elem}`, если класс `core` открыт.

ПРИМЕР 14.2. Пусть надо преобразовать строку "[3,4,5]", содержащую список целых чисел, в терм типа `integer*`, т. е. собственно в список [3,4,5]. Для этого в языке естьстроенная функция `toTerm` и хотелось бы ею воспользоваться примерно так:

```

Z = toTerm(integer*, "[3,4,5]"),

```

чтобы переменная `Z` содержала не строку, а список чисел [3,4,5], с которым можно проводить списочные операции. Однако вызов этой функции с указанием домена `integer*`, в который требуется преобразовать строку, компилятор не пропустит. Конечно, всегда можно объявить свой домен:

```

domains
    integerList = integer*.

```

и вызвать функцию так:

```

Z = toTerm(integerList, "[3,4,5]"),

```

Однако существует более изящный способ — воспользоваться списочным доменом `list{Elem}` вместо домена `Elem*`. Тогда вызов функции будет выглядеть так:

```

Z = toTerm(list{integer}, "[3,4,5"]),

```

и при этом не надо дополнительно объявлять списочный домен `integerList`, т. к. у нас под рукой всегда есть универсальный списочный домен `list{Elem}`.

ГЛАВА 15



Эллипсис

Термином *эллипсис* в программировании называют умолчание числа параметров. В Visual Prolog эллипсис дает возможность передачи предикату произвольного количества параметров. При этом сами параметры не указываются (умалчиваются), а их позиция обозначается знаком многоточия (...).

Необходимость использования эллипсиса возникает тогда, когда необходимо передать на обработку заранее неизвестное количество параметров произвольных типов. Типы параметров на приемной стороне должны быть известны. Характерным примером использования эллипсиса является предикат `write/...`, который допускает любое количество аргументов. Эллипсисом удобно пользоваться при вызове исключений для передачи произвольного количества различных параметров, описывающих возникшую исключительную ситуацию.

Порядок использования эллипсиса следующий:

1. Преобразовать параметры, передаваемые по умолчанию, в универсальный тип `any` с помощью функции `toAny/1`.
2. Создать эллипсис-блок из списка этих параметров с помощью функции `toEllipsis/1`.
3. Передать эллипсис-блок в предикат, который ожидает переменное число аргументов в позиции эллипсиса (...).
4. Обратно преобразовать эллипсис-блок в список параметров с помощью функции `fromEllipsis/1`.
5. Обратно преобразовать параметры типа `any` в исходные типы с помощью функции `convert/2`.

Функция `toAny/1` преобразует указанный терм `Term` в значение универсального терма `any`:

```
Any_term = toAny(Term) .
```

Функция `toEllipsis/1` используется для создания эллипсис-блока. Например, список из трех элементов типа `any` можно преобразовать так:

```
EllipsisBlock = toEllipsis([Any_term1, Any_term2, Any_term3]) .
```

Функция `fromEllipsis/1` выполняет обратное преобразование эллипсис-блока в список термов универсального типа `any`:

```
AnyTermList = fromEllipsis(EllipsisBlock).
```

ПРИМЕР 15.1. Рассмотрим использование эллипсиса для обработки списка чисел и строк, а во втором случае — только списка чисел. В первом случае числа, передаваемые на нечетных позициях, складываются, а строки, передаваемые на четных позициях, конкатенируются. Во втором случае элементы эллипсиса складываются.

```
implement main
    open core, console, string
class predicates
    el: (integer,...).
    calc: (any*,integer,integer [out]). 
clauses
    el(1,...):-
        [W1,W2,W3,W4|_] = fromEllipsis(...),!,
        write(toTerm(unsigned,W1) + toTerm(unsigned,W3)), nl,
        write(concat(toTerm(string,W2), toTerm(string,W4))). 
    el(_,...):-
        List = fromEllipsis(...),
        calc(List,0,Sum),
        write(Sum). 

    calc([X|L],Acc,Sum) :- calc(L,Acc + toTerm(integer,X),Sum).
    calc([],Sum,Sum).

    run():-W1=toany(53),
        W2=toany("qwe"),
        W3=toany(7),
        W4=toany("rty"),
        W5=toany(-10),
        W6=toany(-30),
        W7=toany(10),
        EllipsisBlock1 = toEllipsis([W1,W2,W3,W4]),
        el(1,EllipsisBlock1), nl,
        EllipsisBlock2 = toEllipsis([W1,W3,W5,W6,W7]),
        el(2,EllipsisBlock2), nl,
        _=readline().
end implement main
goal
    console::run(main:::run).
```

В первом случае на экран будет выведено число 60 и строка `qwert`. Во втором случае будет выведена сумма всех чисел, равная 30:

```
60
```

```
qwert
```

```
30
```



ГЛАВА 16

Предикаты второго порядка и анонимные предикаты

В этой главе мы рассмотрим использование предикатов в качестве аргументов других предикатов. Предикат, имеющий аргументы-предикаты, называется *предикатом второго порядка*. Предикат, имеющий в качестве аргументов предикаты второго порядка, называется *предикатом высшего порядка*. Использование предикатов второго и высших порядков повышает выразительную мощность языка.

16.1. Предикатные и функциональные домены

Исторически первыми в Visual Prolog были введены предикатные домены, позволяющие объявлять предикаты высших порядков. *Предикатный* домен задает сигнатуру предикатов, к которой относятся типы параметров, режимы детерминизма и шаблоны потоков параметров. *Функциональный* домен задает сигнатуру функций, к которой относятся типы параметров, тип возвращаемого функцией значения, режимы детерминизма и шаблоны потоков параметров. Имя предиката/функции не относится к сигнатуре:

```
domains
    ppp = (integer, integer) determ (i,o). % предикатный домен
    fff = (integer) -> integer determ.      % функциональный домен
```

Предикатные домены допускают использование полиморфизма. Передаваемые формальные имена типов данных указываются в фигурных скобках:

```
domains
    ppp{In,Out} = (In, Out) determ (i,o). % предикатный домен
    fff{In,Out} = (In) -> Out determ.      % функциональный домен
```

Использовать предикатные и функциональные домены можно в следующих ситуациях:

- для объявления группы предикатов или функций, имеющих одинаковые режимы детерминизма и шаблоны потоков параметров:

```
predicates
    pred1 : ppp{integer*, integer}.
    pred2 : ppp{string*, string}.
```

```
func1 : fff{integer, integer}.
func2 : fff{real, real}.
```

Достоинством такого объявления является легкость внесения изменений в объявление группы предикатов. Чтобы изменить режимы детерминизма или шаблоны потоков параметров всей группы предикатов, достаточно модифицировать только объявление предикатного домена;

- для объявления предиката, аргументами которого являются другие предикаты:

```
predicates
pred : (A*, ppp{A,B}, B* [out]).
func : (string) -> fff{A,B}.
```

В предикате второго порядка `pred/3` в качестве второго аргумента используется предикат, который имеет тип `ppp{A,B}`. Функция второго порядка `func/1` возвращает функцию, которая имеет тип `fff{A,B}`.

ПРИМЕР 16.1. Рассмотрим описание функции инкремента `my_inc/2`, которая прибавляет единицу к своему аргументу при выполнении различных условий, которые ей передаются через предикаты. При невыполнении этих условий функция не изменяет аргумент. В нашем примере определено два таких предиката: `zone1` и `zone2`. В правиле:

```
my_inc(A, Test) = A+1 :- Test(A), !.
```

параметр `Test` определяет вызов одного из предикатов проверки условия `Test(A)`, который явно передается функции `my_inc/2`, например:

```
write(my_inc(7, zone1)).
```

В этом случае переменная `Test` определяет вызов предиката `zone1`, вследствие чего вызов `Test(A)` является вызовом `zone1(A)`.

```
implement main
    open core, console
domains
    ppp{A} = (A) determ.
class predicates
    my_inc : (integer, ppp{integer}) -> integer.
    zone1 : ppp{integer}.
    zone2 : ppp{integer}.
clauses
    my_inc(A, Test) = A+1 :- Test(A), !.
    my_inc(A, _) = A.
    zone1(A) :- A>0, A<10.
    zone2(A) :- A>-5, A<5, A><0.
    run() :- write(my_inc(7, zone1)), nl,          % 8
            write(my_inc(7, zone2)), nl,          % 7
            _ = readLine().
end implement main
goal
    console::run(main::run).
```

Давайте оценим ту пользу, которую приносит использование предикатов второго порядка на примере одной полезной задачи. Решим ее вначале без предикатов второго порядка, а потом с предикатами второго порядка.

ПРИМЕР 16.2. Пусть имеется предикат `greater3`, который фильтрует список целых чисел и не является предикатом второго порядка. Предикат `greater3` собирает в выходном списке только те элементы входного списка, которые больше числа 3.

```
implement main
    open core,console
class predicates
    greater3 : (integer* In, integer* [out]). 
clauses
    greater3([A|L], [A|W]) :- A>3, !, greater3(L,W).
    greater3([_|L], W) :- greater3(L,W).
    greater3([],[]).
run() :- greater3([1,2,3,4,5],W),
        write(W), nl, % [4,5]
        _ = readLine().
end implement main
goal
    console::run(main:::run).
```

Если возникнет необходимость фильтровать списки по другому условию, то надо переписывать предикат заново с другим условием фильтрации. Однако при использовании предикатов второго порядка ничего переписывать не надо. Для этого следует сразу описать предикат фильтрации списка так, чтобы условие фильтрации не было жестко задано в правиле предиката, а передавалось ему через другой предикат. Предикат `filter/3` в качестве второго аргумента принимает предикат проверки условия, объявленный с типом `condition{A}`:

```
implement main
    open core,console
domains
    condition{A} = (A) determ.
class predicates
    filter : (A*, condition{A}, A* [out]). % фильтрация списка
    test1 : condition{integer}. % первое условие фильтрации
    test2 : condition{real}. % второе условие фильтрации
    test3 : condition{string}. % третье условие фильтрации
clauses
    filter([A|L], Test, [A|W]) :- Test(A), !, % проверка условия
                                    filter(L, Test, W). % условие истинно
    filter([_|L], Test, W) :- filter(L, Test, W). % условие ложно
    filter([], _, []). 

    test1(A) :- A<3, !; A>5.
    test2(A) :- math::abs(math::fraction(A)) > 0.7.
    test3(A) :- string::length(A)<3.
```

```

run() :- filter([1,2,3,4,5,6,7], test1, E),
        write(E), nl,                                % [1,2,6,7]
        filter([0.88, 45.108, 2.76], test2, R),
        write(R), nl,                                % [0.88, 2.76]
        filter(["123","ab","jjjj","9"], test3, T),
        write(T), nl,                                % ["ab", "9"]
        _ = readLine().

end implement main
goal
    console::run(main:::run).

```

В этой программе фильтрация вызывается три раза с различными списками и условиями фильтрации. Первое условие `test1` выбирает целые числа, меньше 3 или больше 5. Второе условие `test2` выбирает вещественные числа, дробная часть которых больше 0.7. Третье условие `test3` выбирает строки, длина которых меньше 3. Предикат фильтрации `filter/3` полиморфный и описан единожды благодаря тому, что является предикатом второго порядка.

ПРИМЕР 16.3. Рассмотрим использование функции в качестве выходного параметра при реализации калькулятора с нестандартными для языков программирования функциями. В этом калькуляторе с помощью строк, играющих роль условных обозначений нестандартных операций, задаются операции, а потом этим операциям передаются параметры, над которыми следует произвести действия.

```

implement main
    open core,console
domains
oper{A,B} = (A,A) -> B.   % объявление типа функции двух аргументов
func{A,B} = (A) -> B.      % объявление типа функции одного аргумента
class predicates
operation:(string) -> oper{integer,real} determ.
function:(string) -> func{integer,integer} determ.
gcd : oper{integer,real}.   % объявление наибольшего
                           % общего делителя
averGeom : oper{integer,real}. % объявление среднего
                               % геометрического
power : func{integer,integer}. % объявление функции  $n^2 - 1$ 
fact : func{integer,integer}. % объявление факториала
clauses
operation("gcd") = gcd.
operation("averGeom") = averGeom.
function("^2") = power.
function("!") = fact.

gcd(A,B) = B :- A mod B = 0,!.
gcd(A,B) = gcd(B,A mod B).           % наибольший общий делитель

```

```

averGeom(X,Y) = math::sqrt(X*Y).      % среднее геометрическое

power(X) = X*X-1.                      % функция  $n^2 - 1$ 
fact(0) = 1 :- !.                      % факториал
fact(N) = N*fact(N-1).

run() :- Q = operation("gcd"),
        write(Q(175,25)),nl,          % задание операции НОД
        write(Q(18,6)),nl,            % 25
        W = operation("averGeom"),   % задание операции сред.геом.
        write(W(3,8)),nl,            % 6
        write(W(1,8)),nl,            % 4.89897948556636
        E = function("^2"),          % задание функции  $n^2 - 1$ 
        write(E(4)),nl,              % 15
        write(E(5)),nl,              % 24
        R = function("!"),
        write(R(4)),nl,              % задание функции факториала
        write(R(5)),nl,              % 24
        write(R(6)),nl,              % 120
        _ = readLine(),!;
        _ = readLine().
end implement main
goal
    console::run(main::run).

```

Необходимая операция задается с помощью вызова функции — например, `Q = operation("gcd")`. При этом переменная `Q` получает предикатное значение наибольшего общего делителя `gcd`. Значение переменной `Q` можно применить к параметрам — например: `18` и `6`, `N = Q(18, 6)`, что эквивалентно непосредственному вызову предиката `N = gcd(18, 6)`.

Необходимость объявления предикатных доменов и предикатов, которые передаются в качестве аргументов другим предикатам, является недостатком такого подхода к высокоуровневому программированию. Во многих случаях использование предикатных и функциональных доменов можно с уверенностью заменить анонимными предикатами и функциями, которые свободны от этого недостатка.

16.2. Анонимные предикаты и функции

Теоретические основы анонимных предикатов и функций восходят к лямбда-исчислению, разработанному американским математиком Алонзо Черчем, для формализации и анализа понятия вычислимости. Исторически первым лямбда-функции в программировании применил Джон Маккарти в языке Lisp.

В языке Visual Prolog лямбда-функции часто называют анонимными функциями или анонимными предикатами. Они позволяют использовать предикаты и функции в качестве аргументов других предикатов. Анонимные предикаты и функции не объявляются и не имеют имени.

16.2.1. Определения анонимных предикатов

Анонимный предикат (функция) — это выражение, которое возвращает предикатное (функциональное) значение. Выражение заключается в фигурные скобки и, по сути, представляет собой определение предиката без указания его имени. Выражение в фигурных скобках возвращает предикатное значение, которое может быть связано с переменной. Используя такую переменную, можно вызвать предикат, или передать его как аргумент другому предикату, или получить его от другого предиката.

В общем случае анонимный предикат определяется в виде

```
P = { (Аргументы) :- Тело}
```

В этом определении аргументы Аргументы являются свободными переменными. Переменная P получает предикатное значение, которое впоследствии может быть применено к аргументам посредством вызова предиката P(Аргументы). Когда настает необходимость вызова предиката P(Аргументы), переменная P синтаксически будет служить именем предиката, а аргументы Аргументы должны быть связанными входными переменными.

В случае, когда анонимный предикат является нульярным, его определение можно записать в сокращенной форме:

```
P = { :- Тело}
```

Переменные анонимного предиката являются локальными и видны только внутри этого предиката. Вернуть результат через выходные переменные анонимный предикат не может, т. к. не имеет таковых.

Анонимные функции определяются в следующем виде:

```
F = { (Аргументы) = Терм :- Тело}
```

```
F = { (Аргументы) = Терм}
```

В случае нульярной функции определение можно записать в сокращенной форме:

```
F = { = Терм :- Тело}
```

```
F = { = Терм}
```

Все аргументы анонимной функции должны быть входными.

16.2.2. Использование анонимных предикатов и функций

Анонимные предикаты и функции выгодно использовать в следующих случаях:

- для многократного выполнения определенных операций над разными наборами данных в теле какого-либо предиката. Операции передаются в этот предикат через аргумент, в качестве которого выступает анонимный предикат;
- для ветвления вычислений. Такое ветвление оформляется в виде анонимного предиката или функции;

- для метаобработки данных. В этом случае сам механизм обработки данных описывается в теле какого-либо предиката, которому передается анонимный предикат или функция, определяющие конкретные операции обработки;
- для интерпретации вычислений с априори заданным перечнем допустимых операций. Выполнение конкретной операции из перечня производится с помощью анонимного предиката.

ПРИМЕР 16.4. Рассмотрим использование анонимной функции для совершения многократных вычислений над разными данными.

run() :-

```

Inc = { (X) = X+1},      % определение анонимной функции инкремента
A = Inc(4),              % вычисление инкремента числа 4
B = Inc(23),             % вычисление инкремента числа 23
C = { (X) = X-1} (4),    % определение и вычисление декремента числа 4
D = Inc(10) - Inc(8),    % вычисление значения выражения
E = Inc(Inc(Inc(2))),   % вычисление композиции функций
writeln("A=%, B=%, C=%, D=%, E=%", A,B,C,D,E),
_ = readLine().

```

Переменная `Inc` содержит ссылку на анонимную функцию инкремента и, следовательно, может быть применена к числовому аргументу так, как будто является функцией. Вызов функции `A = Inc(4)` приведет к тому, что переменная `A` будет связана числом 5. В результате выполнения предиката `run` на экран будет выведено:

`A=5, B=24, C=3, D=2, E=5`

Следует заметить, что определение и вычисление анонимной функции декремента в одном месте:

`C = { (X) = X-1} (4)`

не дает никакого выигрыша, т. к. выполняется единожды, после чего переменная `C` будет хранить число 3, а не предикатное значение декремента. В этом случае лучше вызвать функцию декремента явно `C = 4-1`.

ПРИМЕР 16.5. Рассмотрим использование анонимного предиката для совершения многократных вычислений над разными данными.

```

run() :- Inc = { (X) :- write(X+1), nl},   % определение инкремента
        Inc(4),                      % вывод инкремента числа 4
        Inc(23),                      % вывод инкремента числа 23
_ = readLine().

```

Так как анонимный предикат может иметь только входные аргументы, то нельзя получить значение аргумента через выходную переменную. В связи с этим следующий предикат приведет к ошибке компиляции:

```

run() :- Inc = { (X,Y) :- Y=X+1},          % определение инкремента
        Inc(4,A),                  % вычисление инкремента числа 4
        Inc(23,B),                  % вычисление инкремента числа 23
        writeln("A=%, B=%", A,B),
_ = readLine().

```

ПРИМЕР 16.6. Рассмотрим использование анонимной функции для ветвления вычислений.

```
run() :- F = { (X) = Y :- X>0, !, Y=X+1; X=0, !, Y=X; Y=X-1},      % ветвление
        A = F(4),           % 5
        B = F(-10),         % -11
        C = F(0),           % 0
        writeln("A=%, B=%, C=%", A, B, C),
        _ = readLine().
```

Анонимная функция определяет три ветви. Если входная переменная x больше нуля, то возвращается инкремент переменной, если x равна нулю, то ноль и возвращается. Если переменная x меньше нуля, то возвращается декремент переменной.

ПРИМЕР 16.7. Перепишем пример 16.1 с использованием анонимных предикатов. Вначале заменим предикаты zone1/1 и zone2/1 анонимными предикатами, значения которых присвоим переменным z1 и z2 соответственно.

```
implement main
  open core, console
domains
  ppp{A} = (A) determ.
class predicates
  my_inc : (integer, ppp{integer}) -> integer.
clauses
  my_inc(A, Test) = A+1 :- Test(A), !.
  my_inc(A, _) = A.
  run() :- Z1 = { (A) :- A>0, A<10},
          Z2 = { (A) :- A>-5, A<5, A<>0},
          write(my_inc(7,Z1)), nl,           % 8
          write(my_inc(7,Z2)), nl,           % 7
          _ = readLine().
end implement main
goal
  console::run(main:::run).
```

Хотя мы и заменили предикаты zone1/1 и zone2/1 анонимными предикатами, объявление и использование предикатного домена осталось. Это связано с тем, что предикат my_inc/2 в качестве второго аргумента использует предикаты, тип которых объявлен предикатным доменом ppp{A}. Заменим функцию my_inc/2 анонимной функцией и избавимся от объявления предикатного домена и функции my_inc/2:

```
run() :- Z1 = { (A) :- A>0, A<10},
          Z2 = { (A) :- A>-5, A<5, A<>0},
          Inc = { (A, Test) = B :- Test(A), B=A+1, !; B=A },
          write(Inc(7,Z1)), nl,           % 8
          write(Inc(7,Z2)), nl,           % 7
          _ = readLine().
```

Итого, у нас остался по существу один предикат `run/0`, который функционально эквивалентен программе из примера 16.1. Примечательно то, что анонимная функция, связанная с переменной `Inc`, использует унарный анонимный предикат `Z1` или `Z2`, или любой другой, который может быть определен как анонимный, так и посредством предикатного домена.

ПРИМЕР 16.8. Перепишем пример 16.2 с использованием анонимных предикатов. Для этого заменим предикаты `test1`, `test2` и `test3` анонимными предикатами, значения которых присвоим переменным `F1`, `F2` и `F3` соответственно.

```
implement main
    open core,console
domains
    condition{A} = (A) determ.
class predicates
    filter : (A*, condition{A}, A* [out]) .
clauses
    filter([A|L],Test,[A|W]) :- Test(A),!,
                           filter(L,Test,W).
    filter([_|L],Test,W) :- filter(L,Test,W).
    filter([],_,[]).

    run() :- F1 = { (A) :- A<3,!; A>5},
            F2 = { (A) :- math::abs(math::fraction(A)) > 0.7},
            F3 = { (A) :- string::length(A)<3},
            filter([1,2,3,4,5,6,7], F1, E),
            write(E),nl,                                     % [1,2,6,7]
            filter([0.88, 45.108, 2.76], F2, R),
            write(R),nl,                                     % [0.88, 2.76]
            filter(["123","ab","jjjj","9"], F3, T),
            write(T),nl,                                     % ["ab", "9"]
            _ = readLine().
end implement main
goal
    console::run(main:::run).
```

Использование предикатного домена здесь обязательно, т. к. вторым аргументом в предикате `filter/3` является предикат, тип которого описывается этим предикатным доменом. В случае, когда анонимный предикат используется однократно, его непосредственно можно указывать в качестве аргумента другого предиката. Поэтому в рассматриваемом примере предикат `run/0` можно смело переписать в виде:

```
run() :- filter([1,2,3,4,5,6,7], { (A) :- A<3,!; A>5}, E),
        write(E),nl,                                     % [1,2,6,7]
        filter([0.88, 45.108, 2.76],
              { (A) :- math::abs(math::fraction(A)) > 0.7}, R),
        write(R),nl,                                     % [0.88, 2.76]
```

```

filter(["123","ab","jjjj","9"],
       { (A) :- string::length(A)<3}, T),
write(T),nl,
_ = readLine(). % ["ab", "9"]

```

Избавиться от предикатного домена в этом примере не получится, т. к. предикат фильтрации списков `filter/3` определен рекурсивно, а анонимные предикаты в большинстве случаев не допускают непосредственного рекурсивного определения. Предикат `filter/3` на данном этапе можно использовать в качестве библиотечного предиката. Для его вызова из какой-либо прикладной программы достаточно описать условие фильтрации в виде анонимного предиката. В тех случаях, когда условие фильтрации определяется только рекурсивно, необходимо явно объявить и определить рекурсивный предикат на основе предикатного домена `condition{A}`.

ПРИМЕР 16.9. Рассмотрим реализацию калькулятора с помощью нестандартных для языков программирования функций на основе анонимных функций. Прототипом нашей программы является пример 16.3.

```

implement main
open core,console
domains
oper{A,B} = (A,A) -> B. % объявление типа функции двух аргументов
func{A,B} = (A) -> B.    % объявление типа функции одного аргумента
class predicates
operation:(string) -> oper{integer,real} determ.
function:(string) -> func{integer,integer} determ.
gcd : oper{integer,real}.      % наибольший общий делитель
fact : func{integer,integer}. % факториал
clauses
operation("gcd") = gcd.
function("!") = fact.
gcd(A,B) = B :- A mod B = 0,!.. % наибольший общий делитель
gcd(A,B) = gcd(B,A mod B).
fact(0) = 1 :- !.                % факториал
fact(N) = N*fact(N-1).

run() :-Q = operation("gcd"),   % задание операции НОД
        write(Q(175,25)),nl,     % 25
        write(Q(18,6)),nl,       % 6
        W = {(X,Y) = math::sqrt(X*Y)}, % среднее геометрическое
        write(W(3,8)),nl,        % 4.89897948556636
        write(W(1,8)),nl,        % 2.82842712474619
        E = {(X) = X*X-1},      % задание функции  $n^2 - 1$ 
        write(E(4)),nl,          % 15
        write(E(5)),nl,          % 24
        R = function("!"),       % задание функции факториала

```

```

write(R(4)),nl,          % 24
write(R(5)),nl,          % 120
_ = readLine(),!;
_ = readLine().
end implement main
goal
  console::run(main::run).

```

В программе из примера 16.3 мы смогли заменить анонимными функциями лишь две нерекурсивные функции: функцию вычисления среднего геометрического двух чисел и функцию вычисления значения выражения $n^2 - 1$. Рекурсивную функцию вычисления наибольшего общего делителя и функцию вычисления факториала мы не заменили аналогичными анонимными функциями. Справедливости ради следует заметить, что эти функции все-таки можно описать анонимно, однако для этого надо вводить вспомогательные анонимные функции, что требует достаточно высокой квалификации в рекурсивном программировании и анонимных функциях. Этот вопрос здесь рассматриваться не будет.

16.2.3. Замыкание

Замыкание (англ. closure) — это предикат, захватывающий из окружающего контекста переменную, которая в этом предикате не определена. В практике программирования с помощью замыкания можно получить некоторую функцию от другой функции, связав некоторые ее аргументы фиксированными значениями. В Visual Prolog замыкание можно реализовать как в виде явно объявленного предиката, так и в виде анонимного предиката.

В следующем примере анонимная функция $\{ (X)=X+Y \}$ имеет в своем теле переменную Y , которую она захватывает из окружения. Окружением является внешняя анонимная функция $\{ (Y) = \{ (X)=X+Y \} \}$:

```

run() :- F = { (Y) = { (X)=X+Y} },
        G = F(2),           % X+2
        write(G(3)),nl,     % 3+2=5
        write(G(8)),nl,     % 8+2=10
        _ = readLine().

```

Здесь переменная G является ссылкой на предикатное значение сложения $X+Y$, в котором аргумент Y зафиксирован значением 2. Функция $G(3)$ является суммой числа 3 с фиксированным значением 2.

16.2.4. Каррирование

Каррирование (англ. currying) — это преобразование функции от многих аргументов в функцию, берущую свои аргументы по одному.

Такое преобразование получило свое название в честь американского математика и логика Хаскелла Карри. Каррирование функции позволяет фиксировать некоторые ее аргументы, возвращая функцию от остальных нефиксированных аргументов.

ПРИМЕР 16.10. Рассмотрим описанные действия в Visual Prolog.

```

implement main
    open core,console
domains
    function{A} = (A) -> A. % этот домен возвращается функцией f
class predicates
    f : (integer) -> function{integer}.
clauses
    f(Y) = { (X) = X+Y}.      % объявление функции с замыканием

run() :-G = f(5),          % фиксируем второе слагаемое: G = { (X) = X+5}
        A = G(3),          % фиксируем первое слагаемое числом 3. A = 8
        B = G(2),          % фиксируем первое слагаемое числом 2. B = 7
        C = f(10)(4),       % каррирование одним движением. C = 14
        writeln("A = %, B = %, C = %", A, B, C),nl,
        _ = readLine().
end implement main
goal
    console::run(main:::run).

```

Функция $f(Y)$ возвращает анонимную функцию $\{ (X) = X+Y \}$ от одной переменной X . Значение же переменной Y передается в эту анонимную функцию благодаря захвату переменной Y из контекста правила, в котором эта анонимная функция определена.

Вначале в функции $f(5)$ фиксируется переменная Y значением 5. Поэтому переменной G передается функция от одной свободной переменной. Далее фиксируется вторая переменная значением 3, и полученный результат связывается с переменной A . После этого проводятся аналогичные действия с переменной B . Существует возможность зафиксировать значениями сразу две переменных. Переменная C получает значение 14 именно таким способом.

В этой программе каррированная функция $f(Y)$ объявлена и задана явно. Однако ничего не мешает использовать для этого анонимные функции. При этом объявлять функциональный домен и описывать отдельно каррированную функцию не надо:

```

implement main
    open core,console
clauses
run() :- F = { (Y) = { (X)=X+Y} },      % определение функции с замыканием
        G = F(5),          % фиксация одной переменной G = { (X) = X+5}
        A = G(3),          % A = 8
        B = G(2),          % B = 7
        C = F(10)(4),       % C = 14
        writeln("A = %, B = %, C = %", A, B, C),nl,
        _ = readLine().
end implement main
goal
    console::run(main:::run).

```

Обратите внимание, что каррированная функция определена в виде композиции двух функций. Фиксировать первую переменную каррированной функции можно в краткой форме:

```
implement main
    open core,console
clauses
run() :- F = { (Y) = { (X)=X+Y} } (5),      % фиксация одной переменной
        A = F(3),                      % A = 8
        B = F(2),                      % B = 7
        writeln("A = %, B = %", A, B),nl,
        _ = readLine().
end implement main
goal
    console::run(main:::run).
```

Синтаксис анонимных функций подслащен упрощениями. В случае, когда анонимная функция не имеет аргументов или все ее аргументы анонимны, то указывать аргументы в скобках не обязательно. Такой фрагмент кода:

```
P = { (_ ) :- succeed },
Q = { ( _, _ ) = 0 },
R = { ( _, _, _ ) = _ :- fail }
```

может быть записан короче:

```
P = { :- succeed },
Q = { = 0 },
R = { = _ :- fail }
```

16.3. Высокоуровневые предикаты и функции класса *list*

В этом разделе приводится краткое описание основных высокоуровневых предикатов и функций для обработки списков. Подобные функции широко используются и в других современных языках программирования.

- Предикат тестирования всех элементов списка по заданному условию. Предикат успешен, если все элементы списка удовлетворяют условию, иначе — неуспешен.

```
all([2,4,8],{ (A):-A mod 2 = 0}), write("Ok"),!;
write("No")
```

Результат:

Ok

Вместо анонимного предиката, выражающего условие тестирования, конечно же, можно использовать явно объявленный предикат. Также и в остальных предикатах этого раздела:

```

class predicates
    fff : (integer) determ.
clauses
    fff(A) :- A mod 2 = 0.
    run() :- all([2,4,8],fff), write("all"),!;
              write("no").

```

2. Создание лексического компаратора сравнения элементов списков. Результатом является анонимная функция сравнения списков. Пример показывает традиционное сравнение списков. Первый список меньше второго, т. к. $1 < 3$.

```

F = compareLexically({(A,B)=R:-A>B,R=greater orelse A=B, R=equal
                      orelse R=less}),
write(F([1,2,3,4],[3,4,5])),nl,

```

Результат:

```
less
```

3. Декомпозиция списка разделяет элементы исходного списка на ряд кортежей согласно условиям, задаваемым функцией. Эта функция возвращает номера кортежей. Вторые аргументы кортежей являются собственно подсписками.

```

write(decompose([1,2,3,4,5,6,7,8,9],
               { (A)=Z:-A>5,Z=1 orelse A=5,Z=0 orelse Z=-1})),

```

Результат:

```
[tuple(-1,[4,3,2,1]), tuple(0,[5]), tuple(1,[9,8,7,6])]
```

4. Разность списков на основе анонимной функции сравнения двух элементов. Пример использует традиционное сравнение элементов списков.

```

write(differenceBy({(A,B)=R:-A>B,R=greater orelse A=B, R=equal
                     orelse R=less},[1,2,3,4],[3,4,5,6]))

```

Результат:

```
[1,2]
```

5. Разность списков на основе анонимного детерминированного предиката сравнения двух элементов. Пример использует традиционное сравнение элементов списков.

```

write(differenceEq({(A,B) :- A=B}, [1,2,3,4], [3,4,5,6]))

```

Результат:

```
[1,2]
```

6. Проверка наличия хотя бы одного элемента, удовлетворяющего условию.

```

exists([1,2,3,4,5,6,7,8,9],{ (A) :- A>5}), write("exist"),!;
write("not exist")

```

Результат:

```
exist
```

7. Поиск элемента, удовлетворяющего условию.

```
exists([3,6,8,9], { (A) :- A>5 }) = Elemt, write(Elemt), !;
write("no")
```

Результат:

6

8. Функция фильтрации строит список из элементов исходного списка, удовлетворяющих условию.

```
write(filter([1,2,3,4,5,6,7,8,9], { (C) :- C>5 }))
```

Результат:

[6,7,8,9]

9. Предикат фильтрации строит один список из элементов исходного списка, удовлетворяющих условию, и другой список из элементов, не удовлетворяющих условию.

```
filter([1,2,3,4,5,6,7,8,9], { (C) :- C>5 }, Positive, Negative),
write(Positive," ",Negative)
```

Результат:

[6,7,8,9] [1,2,3,4,5]

10. Отображение списка в новый список на основе преобразования, задаваемого детерминированной функцией.

```
write(filteredMap([4,-7,3,-2], { (A)=A*A :- A>0 }))
```

Результат:

[16,9]

11. Сборка элементов списка на основе неоптимизированной рекурсии, из-за чего на длинных списках происходит переполнение стека. Правоассоциативное применение ко всем элементам списка операции, задаваемой функцией. Третий аргумент участвует в сборке списка наравне с его элементами. Если исходный список пуст, то функция `fold` возвращает свой третий элемент.

```
write(fold([1,2,3], { (A,B)=A+B }, 10)), nl,
write(fold([1,2,3], { (A,B)=A-B }, 10)), nl,
```

Результат:

16 % 1+(2+(3+10))
-8 % 1-(2-(3-10))

12. Сборка элементов списка на основе оптимизированной рекурсии. Левоассоциативное применение ко всем элементам списка операции, задаваемой функцией. Третий аргумент участвует в сборке списка наравне с его элементами. Если исходный список пуст, то функция `foldl` возвращает свой третий элемент.

```
write(foldl([1,2,3], { (A,B)=A+B }, 10)), nl,
write(foldl([1,2,3], { (A,B)=A-B }, 10)), nl,
```

Результат:

```
16          % ((10+1)+2)+3
4           % ((10-1)-2)-3
```

13. Сборка элементов списка на основе оптимизированной рекурсии. Правоассоциативное применение ко всем элементам списка операции, задаваемой функцией. Третий аргумент участвует в сборке списка наравне с его элементами. Если исходный список пуст, то функция `foldr` возвращает свой третий элемент.

```
write(foldr([1,2,3], { (A,B)=A+B}, 10)),nl,
write(foldr([1,2,3], { (A,B)=A-B}, 10)),nl,
```

Результат:

```
16          % 1+(2+(3+10))
-8          % 1-(2-(3-10))
```

14. Выполнение над каждым элементом исходного списка действия, задаваемого предикатом.

```
forall([1,2,3,4,5,6,7,8,9], { (A):-write(A*A," ") })
```

Результат:

```
1 4 9 16 25 36 49 64 81
```

15. Разделение списка на подсписки, содержащие дубликаты. Условие равенства двух элементов задается первым аргументом.

```
write(groupByEq( { (A,B) :- A=B }, [1,4,2,4,3,1,4] ))
```

Результат:

```
[[1,1],[4,4,4],[2],[3]]
```

16. Пересечение списков на основе анонимной функции сравнения двух элементов. Пример использует традиционное сравнение элементов списков.

```
write(intersectionBy( { (A,B) :- A>B, R=greater orelse A=B, R=equal
                           orelse R=less }, [1,2,3,4], [3,4,5,6] ))
```

Результат:

```
[3,4]
```

17. Пересечение списков на основе анонимного детерминированного предиката сравнения двух элементов. Пример использует традиционное сравнение элементов списков.

```
write(intersectionEq( { (A,B) :- A=B }, [1,2,3,4], [3,4,5,6] ))
```

Результат:

```
[3,4]
```

18. Отображение списка в новый список на основе преобразования, задаваемого функцией.

```
write(map([1,2,3,4,5,6,7,8,9], { (A)=A*A }))
```

Результат:

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

19. Поиск максимального элемента списка на основе анонимной функции сравнения двух элементов. Пример использует традиционное сравнение элементов списков.

```
write(maximumBy({(A,B)=R:-A>B,R=greater orelse A=B,R=equal  
orelse R=less},[2,1,4,3]))
```

Результат:

```
4
```

20. Поиск минимального элемента списка на основе анонимной функции сравнения двух элементов. Пример использует традиционное сравнение элементов списков.

```
write(minimumBy({(A,B)=R:-A>B,R=greater orelse A=B,R=equal  
orelse R=less},[2,1,4,3]))
```

Результат:

```
1
```

21. Удаление из списка всех вхождений заданного элемента на основе анонимной функции сравнения двух элементов. Пример использует традиционное сравнение элементов.

```
write(removeAllBy({(A,B)=R:-A>B,R=greater orelse A=B,R=equal orelse  
R=less},[1,2,1,3,3,4],3))
```

Результат:

```
[1,2,1,4]
```

22. Удаление из списка всех вхождений заданного элемента на основе анонимного детерминированного предиката сравнения двух элементов. Пример использует традиционное сравнение элементов.

```
write(removeAllEq({(A,B):- A=B},[1,2,1,3,3,4],3))
```

Результат:

```
[1,2,1,4]
```

23. Удаление из списка первого вхождения заданного элемента на основе анонимной функции сравнения двух элементов. Пример использует традиционное сравнение элементов.

```
write(removeBy({(A,B)=R:-A>B,R=greater orelse A=B,R=equal orelse  
R=less},[1,2,1,3,3,4],1))
```

Результат:

```
[2,1,3,3,4]
```

24. Удаление из списка первого вхождения заданного элемента на основе анонимного детерминированного предиката сравнения двух элементов. Пример использует традиционное сравнение элементов.

```
write(removeEq({(A,B):- A=B},[1,2,1,3,3,4],1))
```

Результат:

[2,1,3,3,4]

25. Удаление из списка последовательностей дубликатов на основе анонимной функции сравнения двух элементов. Применяется, как правило, для сортированных списков. В списке остается последнее вхождение элемента последовательности дубликатов. Вычислительная сложность $O(N)$. Пример использует традиционное сравнение элементов.

```
write(removeConsecutiveDuplicatesBy({(A,B)=R:-A>B,R=greater orelse
                                         A=B,R=equal orelse R=less},[1,3,1,3,3,4]))
```

Результат:

[1,3,1,3,4]

26. Удаление из списка последовательностей дубликатов на основе анонимного детерминированного предиката сравнения двух элементов. Применяется, как правило, для сортированных списков. В списке остается последнее вхождение элемента последовательности дубликатов. Вычислительная сложность $O(N)$. Пример использует традиционное сравнение элементов.

```
write(removeConsecutiveDuplicatesEq({(A,B) :- A=B},[1,3,1,3,3,4]))
```

Результат:

[1,3,1,3,4]

27. Удаление из списка всех дубликатов на основе анонимной функции сравнения двух элементов. В списке остается первое вхождение дубликатов. Вычислительная сложность $O(N \times \ln(N))$. Пример использует традиционное сравнение элементов.

```
write(removeDuplicatesBy({(A,B)=R:-A>B,R=greater orelse
                           A=B,R=equal orelse R=less},[1,3,2,1,3]))
```

Результат:

[1,3,2]

28. Удаление из списка всех дубликатов на основе анонимного детерминированного предиката сравнения двух элементов. В списке остается последнее вхождение дубликатов. Вычислительная сложность $O(N^2)$. Пример использует традиционное сравнение элементов.

```
write(removeDuplicatesEq({(A,B) :- A=B},[1,3,2,1,3]))
```

Результат:

[2,1,3]

29. Сортировка списка термов по возрастанию функциями `sort/1` и `sortBy/2`. В отличие от функции `sort/1`, которая сортирует список термов в лексикографическом порядке, функция `sortBy/2` дает возможность изменить правила сравнения двух термов списка при его сортировке. Для этого в качестве первого аргумента ей передается или собственноручно написанная функция сравнения, или встро-

енная функция сравнения термов с указанием желаемых параметров сравнения. Следующий пример показывает обычную сортировку списка точек L , заданных в декартовой системе своими координатами. Использованы следующие обозначения: L — исходный список точек, L_1 — список точек, сортированный в лексикографическом порядке, L_2 — список точек, сортированный только по координате Y , L_3 — список точек, сортированный по удаленности точки от центра координат.

```
L = [pnt(1,-2),pnt(0,-6),pnt(-3,1)],
L1 = sort(L),
L2 = sortBy({(pnt(_,Y1),pnt(_,Y2))=compare(Y1,Y2)},L),
L3 = sortBy({(pnt(A,B),pnt(X,Y))=compare(A^2+B^2,X^2+Y^2)},L),
write(L1),nl,
write(L2),nl,
write(L3),nl,
```

Результат:

```
[pnt(-3,1),pnt(0,-6),pnt(1,-2)]
[pnt(0,-6),pnt(1,-2),pnt(-3,1)]
[pnt(1,-2),pnt(-3,1),pnt(0,-6)]
```

30. Сортировка списка термов по возрастанию или по убыванию. В функции `sortBy/3` третий параметр указывает направление сортировки. Его значениями могут быть: `ascending` — по возрастанию, `descending` — по убыванию.
31. Получение списка элементов из некоторого терма (числа, строки и т. п.) с помощью детерминированной функции. В приведенном примере первый вызов функции `unfold/2` возвращает список квадратов чисел от 8 до 1, второй вызов возвращает список слов входной строки "qwe 7.8".

```
write(unfold(8,{(I)=tuple(I^2,I-1):-I>0})),nl,
write(unfold("qwe 7.8",{(S)=tuple(T,R):-fronttoken(S,T,R)})),
```

Результат:

```
[64,49,36,25,16,9,4,1]
["qwe","7.8"]
```

32. Объединение списков на основе анонимной функции сравнения двух элементов. Пример использует традиционное сравнение элементов.

```
write(unionBy({(A,B)=R:-A>B,R=greater orelse A=B,R=equal orelse
R=less},[1,2,4],[0,2,3]))
```

Результат:

```
[1,4,0,2,3]
```

33. Объединение списков на основе анонимного детерминированного предиката сравнения двух элементов. Пример использует традиционное сравнение элементов.

```
write(unionEq({(A,B):- A=B},[1,2,4],[0,2,3]))
```

Результат:

```
[1,4,0,2,3]
```

34. Распаковка упакованных списков. Распаковка `unzip` является операцией, обратной упаковке `zip`. Предикат `unzip` поддерживает распаковку от 2 до 12 списков.

```
unzip([tuple(1,'a'), tuple(2,'b'), tuple(3,'c')],A,B),  
write(A," ",B),
```

Результат:

```
[1,2,3]  ['a','b','c']
```

35. Упаковка двух списков в список кортежей `tuple` элементов, занимающих одинаковые позиции в исходных списках. Функция `zip` поддерживает упаковку от 2 до 12 списков.

```
write(zip([1,2,3,4],['a','b','c','d']))
```

Результат:

```
[tuple(1,'a'), tuple(2,'b'), tuple(3,'c'), tuple(4,'d')]
```

36. Недетерминированный выбор из двух списков пары элементов, занимающих одинаковые позиции в этих списках. Функция `zip_nd` поддерживает упаковку от 2 до 12 списков. Пример показывает выбор букв английского алфавита по их номерам.

```
write([B||tuple(A,B)=zip_nd([1,2,3],['a','b','c']),A>1])
```

Результат:

```
['b','c']
```

37. Недетерминированный выбор из двух списков пары элементов, занимающих одинаковые позиции в этих списках. Списки могут иметь разную длину. Хвост более длинного списка игнорируется.

```
write([T||T=zipHead_nd([1,2,3,4,5],['a','b','c'])])
```

Результат:

```
[tuple(1,'a'),tuple(2,'b'),tuple(3,'c')]
```

38. Отображение двух списков в новый список на основе преобразования, задаваемого функцией.

```
write(zipWith({(A,B)=A+B},[1,2,3],[100,200,300]))
```

Результат:

```
[101,202,303]
```

ГЛАВА 17



Императивные конструкции

В Visual Prolog введен ряд конструкций императивных языков. Эти конструкции не являются обязательными элементами языка и имеют эквивалентные аналоги, основанные на классических средствах Пролога. Однако в некоторых случаях они позволяют писать более лаконичный код.

17.1. Разрушающее присваивание

Разрушающее присваивание применяется только к фактам-переменным и к изменяемым переменным класса `varM` и не может быть применено к переменным Visual Prolog. Разрушающее присваивание было рассмотрено в разд. 7.3 и 10.3.

Факты-переменные и основанные на них изменяемые переменные играют роль глобальных переменных, т. е. переменных, видимых в классе, где они объявлены. Интересной особенностью изменяемых переменных в языке Visual Prolog является возможность устанавливать и использовать обработчик события модификации значения такой переменной, что, собственно, присутствует во многих современных языках программирования:

```
run() :-  
    V = varM_integer::new(6), % создание переменной со значением 6  
    Event = V:modified, % получение ссылки Event на событие modified  
    % установка обработчика:  
    Event:addListener({(_):-writeln("Значение изменилось ~n",V:value)}),  
    V:add(2), % изменение значения переменной  
    V:add(0), % значения переменной не изменилось  
    V:add(1), % изменение значения переменной  
    _ = readchar().
```

В результате после трех операций сложения на экран будет выведено всего два сообщения, т. к. сложение с нулем не изменило значение переменной:

Значение изменилось 8

Значение изменилось 9

17.2. Ветвление

Конструкция `if-then-else` в Visual Prolog подобна аналогичной конструкции в императивных языках программирования и может использоваться в трех формах.

1. В сокращенной форме:

```
if Условие then Терм1 end if
```

2. В полной форме:

```
if Условие then Терм1 else Терм2 end if
```

Если предикат(ы) условия успешен, то выполняется предикат(ы) первого терма, иначе — предикат(ы) второго терма. Если блока `else` нет, то в случае ложного условия никакие действия не выполняются, но конструкция `if-then` завершается успешно.

3. Использование вложенного блока `elseif`:

```
if Усл1 then Терм1 elseif Усл2 then Терм2 else Терм3 end if
```

что эквивалентно вложению одного ветвления в другое:

```
if Усл1 then Терм1 else
  if Усл2 then Терм2 else Терм3 end if
end if
```

Предикат(ы) условия может иметь как режим `determ`, так и режимы `multi` или `nondeterm`. Однако, несмотря на возможный недетерминизм, вся конструкция условного ветвления является процедурой, т. к. в случае режима `multi` или `nondeterm` предикат условия вычисляется только до первого успеха, после чего выполняется предикат ветви `then`. Если же ни одного успешного решения не будет найдено, то выполнится предикат ветви `else`.

ЗАМЕТКА

Запятые перед ключевыми словами `then`, `elseif`, `else` и `end if` не ставятся.

Эквивалентный аналог полной формы конструкции `if-then-else` можно записать классическими средствами Пролога: дизъюнкцией и отсечением:

```
(Условие, Терм1; Терм2), !
```

Эквивалентный аналог сокращенной формы конструкции `if-then` имеет вид:

```
(Условие, Терм1; succeed), !
```

Здесь использован тождественно истинный предикат `succeed` вместо неиспользуемой ветви `Терм2` для того, чтобы сохранить процедурный режим условного ветвления.

ПРИМЕР 17.1. Проверка числа на четность с помощью конструкции `if-then`:

```
чет(X) :- if X mod 2 = 0 then write("Число четное"), nl end if.
```

Если число X четное, то на экран будет выведено соответствующее сообщение и курсор переведен на новую строку. В противном случае никаких сообщений не последует, но предикат завершится успешно.

Этот же предикат можно записать без использования конструкции if-then:

```
чет(X) :- X mod 2 = 0, write("Число четное"), nl, !; succeed.
```

ПРИМЕР 17.2. Определение четности наибольшего числа из пары чисел с помощью вложенного ветвления.

```
тест(X,Y) :-  
    if X>Y then  
        if X mod 2 = 0 then  
            write("Наибольшее число - четное")  
        else  
            write("Наибольшее число - нечетное") end if  
    else  
        if Y mod 2 = 0 then  
            write("Наибольшее число - четное")  
        else  
            write("Наибольшее число - нечетное") end if  
    end if.
```

Эквивалентный аналог предиката `тест/2` без использования императивной конструкции вложенного ветвления:

```
тест(X,Y) :-  
    (X>Y, (X mod 2 = 0, write("Наибольшее число - четное");  
            write("Наибольшее число - нечетное"));  
     (Y mod 2 = 0, write("Наибольшее число - четное");  
      write("Наибольшее число - нечетное"))), !.
```

17.3. Условные выражения

Условные выражения if-then-else могут вычислять выражения в зависимости от истинности условия. Каждая ветвь выражения if-then-else должна быть также выражением. Поэтому, по сути, выражение if-then-else является функцией.

Простейшим примером является вычисление минимума двух чисел:

```
Min = if X < Y then X else Y end if
```

В этом предикате вычисляется значение условия $X < Y$. Если условие истинно, то переменная `Min` унифицируется с термом `X`, иначе — с термом `Y`.

17.4. Цикл *foreach*

В Visual Prolog введен цикл `foreach` над выходными аргументами недетерминированных предикатов:

```
foreach Term1 do Term2 end foreach
```

Конструкция цикла `foreach` подобна аналогичной конструкции в императивных языках. `Term1` представляет собой предикат(ы) выборки экземпляра из какой-либо коллекции. Это могут быть, например, элементы списка, факты БД, а в общем случае — решения недетерминированного предиката или предиката с режимом `multi`. Предикат(ы) `Term2` является телом цикла и производит какие-либо действия. Тело цикла должно иметь режим детерминизма `procedure` или `erroneous`.

ЗАМЕТКА

Запятые перед ключевыми словами `do` и `end foreach` не ставятся.

Эквивалентный аналог цикла `foreach` можно выразить классическими средствами Пролога — циклом с откатом:

`Term1, Term2, fail; succeed.`

ПРИМЕР 17.3. Определение четности чисел в базе данных.

```
d(3). d(4). d(5). d(2).
чет() :- foreach d(X) do
    if X mod 2 = 0 then
        writef("Число % - четное\n",X), !
    else
        writef("Число % - нечетное\n",X)
    end if
end foreach.
```

Внутренняя база данных содержит четыре числа. Предикат `чет()` проверяет каждое число на четность и выводит соответствующее сообщение.

Эквивалентный аналог предиката `чет/0` без использования императивных конструкций:

```
d(3). d(4). d(5). d(2).
тест(X) :- X mod 2 = 0, writef("Число % - четное\n",X), !;
            writef("Число % - нечетное\n",X).
чет() :- d(X), тест(X), fail; succeed.
```

Здесь использовался вспомогательный предикат `тест/1`. Этот предикат описывает две ветви условного выполнения, первая из которых имеет отсечение. Отсечение нужно для того, чтобы выполнялась только одна из ветвей. Если не использовать вспомогательный предикат, то мы получим программу с неправильным поведением:

```
d(3). d(4). d(5). d(2).
чет() :- d(X),
        (X mod 2 = 0, writef("Число % - четное\n",X), !;
         writef("Число % - нечетное\n",X)),
        fail,
        succeed.
```

Ошибка заключается в том, что цикл прервется сразу после нахождения первого четного числа. Причина ошибки кроется в отсечении, которое препятствует откату

к предикату `d(X)` после выполнения `fail`. Откат будет произведен к предикату `succeed` и завершению цикла. Для того чтобы избавиться от этой ошибки, был введен вспомогательный предикат `тест/1`.

Чтобы избавиться от вспомогательного предиката `тест/1`, можно использовать конструкцию `if-then-else` в теле предиката `чет()`:

```
d(3). d(4). d(5). d(2).
чет() :- d(X),
        if X mod 2 = 0 then
            writef("Число % - четное\n", X)
        else
            writef("Число % - нечетное\n", X)
        end if,
        fail;
        succeed.
```

Цикл `foreach` может быть вложенным:

```
foreach Term1 do
    foreach Term2 do
        Term3
    end foreach
end foreach
```

Термы `Term1` и `Term2` должны иметь режим `nondeterm` или `multi`. Терм `Term3` должен быть процедурой.

Эквивалентный аналог вложенного цикла `foreach` можно выразить классическими средствами Пролога — циклом с откатом:

```
Term1, Term2, Term3, fail; succeed.
```

ПРИМЕР 17.4. Вывод четных чисел списка списков `LL=[[3,2],[],[6,7]]`.

```
чет(LL) :- foreach L in LL do
            foreach X in L do
                X mod 2 = 0, write(X, "\n"), !; succeed
            end foreach
        end foreach.
```

Внешний цикл делает перебор элементов списка списков. Внутренний цикл перебирает элементы списка и печатает четные. Например, при `LL=[[3,2],[],[6,7]]` на экран будет выведена двойка и шестерка.

Этот пример можно переписать классическими средствами Visual Prolog:

```
чет(LL) :- L in LL,
          X in L,
          X mod 2 = 0, write(X, "\n"), fail;
          succeed.
```

17.5. Циклы с заданным числом повторений

Циклы императивных языков с заданным числом повторений типа `for` в Visual Prolog выражаются не только языковыми средствами, рассмотренными в главе 10, но и на основе недетерминированных функций класса `std`. Эти функции выступают в роли счетчика цикла. Они на откате возвращают числа, являющиеся членами арифметической прогрессии с заданными параметрами: начальное и, возможно, конечное значения, шаг. Описание этого класса приведено в *приложении 9*.

ГЛАВА 18



Обработка исключительных ситуаций

Исключительная ситуация (исключение, exception) — аномальная ситуация, вызванная невыполнением каких-либо условий и требующая реакции со стороны программы с целью исключения возможного фатального ее завершения.

Исключения могут вызываться явно, т. е. непосредственным вызовом в программе при невыполнении каких-либо условий. С другой стороны, исключения могут генерироваться системой в процессе выполнения программы.

18.1. Явный вызов исключений

Для явного вызова исключений в классе exception есть ряд предикатов. Рассмотрим некоторые из них. Исключение может вызываться предикатом:

```
exception::raise_exception(Exception,...)
```

Первый параметр Exception является обязательным и должен иметь вид:

```
core::exception(string,      % имя класса  
                string,      % имя исключения  
                string)     % описание исключения
```

Второй и последующие параметры являются необязательными. Через них обработчику исключения можно передавать значения аргументов, которые привели к возникновению исключения или какую-то другую дополнительную информацию. Перечисленные параметры задаются программистом по критерию точной локализации причины исключения.

ПРИМЕР 18.1. Здесь приведена программа использования исключения в случае, когда входной параметр предиката `ppp` больше 10.

ПРИМЕЧАНИЕ

Вызывать эту и другие программы главы 18 надо с помощью команды меню **Build | Run in window** или комбинации клавиш `<Alt>+<F5>`, ибо при обычном запуске комбинацией клавиш `<Ctrl>+<F5>` после вывода информации об исключении консольное окно сразу закроется, и мы ничего не успеем прочитать.

```

implement main
    open core,console
class predicates
    ppp:(integer).
clauses
    ppp(X) :- if X <= 10 then write("Верно")
        else
            Класс = "Класс AAA",
            Исключ = "Тест данных",
            Опис = "Неверный параметр",
            Арг = string::format ("Параметр = % ",X),
            exception::raise_exception(exception(Класс,Исключ,Опис),Арг)
        end if.
    run() :- ppp(12),
        _ = readchar().
end implement main
goal
    console::run(main::run).

```

При вызове предиката с параметром `ppp(12)` будет вызвано исключение, в результате чего на экран будет выведена следующая информация:

```

Exception: Тест данных (Класс AAA)
Неверный параметр
Predicate name = ppp
SourceCursor = main.pro(14,3)
Arguments = Параметр = 12

```

В классе `exception` есть другие предикаты для вызова исключения. Они позволяют выводить на экран информацию о причинах исключения в более понятной пользователю формулировке. Например, пользовательское исключение

```

raise_definiteUser(string,           % пользовательское сообщение
                   namedValue*)   % список именованных значений

```

позволяет вывести на экран строку с сообщением об исключении, а также ряд аргументов, каждый из которых выражается именованным значением. *Именованное значение* — это терм вида:

```

namedValue(string,      % имя
           value)       % значение

```

Тип `value` описывается доменом:

```

domains
value =
    unsigned(unsigned Value);
    integer(integer Value);
    real(real Value);
    char(char Value);
    string(string Value).

```

ПРИМЕР 18.2. Здесь приведена программа, которая использует пользовательское исключение.

```
implement main
    open core,console
class predicates
    ppp:(integer).
clauses
    ppp(X) :- if X <= 100 then
        writef("Возраст: % лет",X)
        else
            Сообщение =
            "Тест данных %Рез.\nПараметр %Пар больше границы %Гран.",
            Арг1 = namedValue("Пар",integer(X)),
            Арг2 = namedValue("Гран",integer(100)),
            Арг3 = namedValue("Рез",string("не пройден")),
            exception::raise_definiteUser(Сообщение,[Арг1,Арг2,Арг3])
            end if.
    run() :- ppp(105),
            _ = readchar().
end implement main
goal
    console::run(main::run).
```

При вызове предиката `ppp(105)` будет вызвано исключение, в результате чего на экране появится информация, указанная в аргументах предиката `raise_definiteUser`. Вот фрагмент информации, выводимой на экран этим пользовательским исключением:

Тест данных не пройден.

Параметр 105 больше границы 100.

В этом примере при формировании пользовательского сообщения были использованы подстановочные имена `%Пар`, `%Гран`, `%Рез`, значения которых подставлялись из списка именованных значений переменных `Арг1`, `Арг2` и `Арг3`.

Подобными выразительными способностями обладает и предикат пользовательского исключения `raise_user`. Аргументы этого предиката такие же, как и у предиката `raise_definiteUser`.

18.2. Обработка исключений

Причинами генерации исключений системой могут стать, например, такие:

- отсутствие файла при чтении данных;
- переполнение памяти;
- неправильный тип параметров, переданных предикату;
- пустой входной поток;

- деление на ноль;
- исключения, специально вызываемые программистом при невыполнении каких-либо условий в программе, и т. п.

Общая схема обработки исключений следующая. При возникновении исключительной ситуации текущее состояние программы сохраняется в *дескрипторе исключения* (exception descriptor) и управление передается предикату, называемому *обработчик исключений* (exception handler). Обработчик исключений при необходимости читает поля дескриптора, формирует сообщение об ошибке и выводит его на экран.

Для перехвата исключений служит языковая конструкция *try-catch-finally*, позволяющая безопасно выполнять код, содержащий потенциальную угрозу нормальному течению программы. Безопасность гарантируется тем, что при возникновении исключения, причина которого зачастую находится вне программы, эта языковая конструкция позволяет перехватить указатель на исключение и вывести на экран существенную информацию о месте и причине аномального поведения программы. После этого, если позволяет логика программы, вычисления продолжаются.

Языковая конструкция *try-catch-finally* имеет вид:

```
try
  Body          % <- здесь вызов небезопасного кода
  catch Var do % захват исключения
    handler1(Var) % обработчик исключения
  finally
    handler2      % финальный обработчик
  end try
```

Здесь *Body* — код, выполняемый под контролем. Блок *catch* содержит обработчик исключения *handler1(Var)* и переменную *Var*, которая будет связана с указателем на исключение в случае, если оно возникнет. Блок *finally* задает финальный обработчик *handler2*, который будет вызван независимо от того, было исключение или нет. Любой из блоков *catch* или *finally* может отсутствовать, но не оба сразу.

Как *Body*, так и оба обработчика *handler1* и *handler2*, не могут иметь режимы детерминизма *multi* и *nondeterm*.

Логика работы конструкции *try-catch-finally* проста. В блоке *try* выполняется небезопасный код *Body* — т. е. такой, который может завершиться исключением. Он выполняется под контролем.

Если *Body* завершается успешно или дает откат, то выполняется финальный обработчик *handler2*, и вся конструкция *try-catch-finally* будет успешна или даст откат назад соответственно.

Если *Body* прерывается исключением, то это исключение перехватывается, и переменная *Var* связывается с указателем на исключение. Затем выполняется обработчик исключения *handler1*, которому в качестве аргумента можно передать переменную *Var*. Посредством этой переменной обработчик может получить доступ

к полям дескриптора исключения. В случае исключения связывание переменных в коде Body не произойдет. После выполнения handler1 выполняется финальный обработчик handler2.

Финальный обработчик handler2 будет выполнять всегда, независимо от того, как завершилось вычисление Body, — успешно, неуспешно или с исключением.

Существует два упрощенных варианта рассматриваемой конструкции. Первый вариант — без блока catch:

```
try
    Body          % небезопасный код
finally
    handler2      % финальный обработчик
end try
```

второй вариант — без блока finally:

```
try
    Body          % небезопасный код
catch Var do   % захват исключения
    handler1(Var) % обработчик исключения
end try
```

Вот пример безопасного чтения файла:

```
try
    Text = file::readString("111.txt")
catch TraceID do
    exceptionDump::dumpToStdOutput(TraceID),
    Text = "Текст по умолчанию"
end try,
write(Text)
```

Если файл существует и успешно прочитан, то на экран будет выведен его текст. Если при чтении файла возникло исключение, то на экран выведется информация об исключении, а затем — текст по умолчанию.

Класс exceptionDump содержит ряд предикатов для высокогоуровневой обработки текста, предназначенного для вывода информации об исключении. Однако можно обработать исключение собственным обработчиком, читая каждое поле дескриптора отдельно. Дескриптор исключения в Visual Prolog выражается термом:

```
descriptor(
    ProgramPoint,    % место возникновения исключения
    Exception,       % краткое описание ошибки, вызвавшей исключение
    Kind,            % вид исключения: raised или continued
    ExtraInfo,        % информация о возникшем исключении
    GMTTTime,         % информация о времени возникновения исключения
    ThreadId).        % идентификатор потока, в котором произошло исключение
```

Поле ProgramPoint содержит информацию об имени класса, файла, предиката и позицию курсора, где произошло исключение. Поле ExtraInfo содержит список именованных значений, отражающих разнообразную информацию об исключении.

ПРИМЕР 18.3. Здесь приведен пример вывода на экран всех полей дескриптора, а также некоторой другой информации.

```

implement main
    open core, console, exception, exceptionDump
class predicates
    myHandler : (traceID).
clauses
    myHandler(TraceID) :-
        write("TraceID: ", TraceID), nl, nl,
        if Descriptor = getDescriptor_nd(TraceID), ! then
            Descriptor = descriptor( ProgramPoint,
                Exception,
                Kind,
                ExtraInfo,
                GMTTime,
                ThreadId),
            write("ProgramPoint: ", ProgramPoint), nl, nl,
            write("Exception: ", Exception), nl, nl,
            write("Kind: ", Kind), nl, nl,
            write("ExtraInfo: ", ExtraInfo), nl, nl,
            write("GMTTime: ", GMTTime), nl, nl,
            write("ThreadId: ", ThreadId), nl, nl
        end if,
        Code = getErrorCode(TraceId),
        write("Error code: ", Code), nl, nl,
        TraceInfo = getTraceInfo(TraceId),
        TraceInfo = traceinfo(LineInfoList, _, _),
        LineInfoString = formatLineInfoList(LineInfoList),
        write("LineInfoString: ", LineInfoString), nl, nl.

run() :- try
    Text = file::readString("111.txt")
    catch TraceID do
        myHandler(TraceID),
        Text = "Текст по умолчанию"
    end try,
    write(Text).

end implement main
goal
    console::run(main:::run).

```

Вывод на экран всех полей достаточно громоздок и трудночитаем, т. к. вся информация представляется в виде термов. К примеру, в упрощенном виде поля `Exception` и `ExtraInfo` выглядят так:

```
Exception: exception("fileSystem_api",
                     "cannotCreate",
                     "Cannot create or open the specified file")
ExtraInfo: [namedvalue("ErrorDescription",
                      string("Не удается найти указанный файл.\r\n"))]
```

ПРИМЕР 18.4. Информацию, упакованную в термы, можно «вытащить» и использовать для вывода сообщения о причинах исключения. В следующем примере показано, как это можно сделать. В примере осуществляется деление на ноль. В обработчике выводится только краткое сообщение о делении на ноль.

```
implement main
  open core, console, exception
class predicates
  d : (integer) -> real.
  myHandler : (traceID).
clauses
  d(X) = 1/X.

myHandler(TraceID) :-
  TrapInformationObject = getTrapInformationObject(TraceId),
  Message = TrapInformationObject:getPrepareMessage(),
  write(Message), nl.

run() :- try
  write("X=", d(0))
  catch TraceID do
    myHandler(TraceID)
  finally
    write("\nРасчет завершен")
  end try.
end implement main
goal
  console::run(main::run).
```

В результате выполнения получим следующий результат:

```
Exception C000008E: Float divide by zero
Расчет завершен
```

ГЛАВА 19



Классы

В этой главе мы рассмотрим классы, способные порождать объекты. Такие классы содержат интерфейс, декларацию класса и имплементацию класса. Имплементацию класса еще называют *реализацией класса*.

19.1. Структура класса

При добавлении в проект нового класса с именем, например, `my_class`, Visual Prolog создает три файла:

- `my_class.i` — интерфейс класса;
- `my_class.cl` — декларация класса;
- `my_class.pro` — реализация класса.

После компиляции нового класса в его папке можно увидеть пять файлов:

- `my_class.pack` — описание путей для загрузки заголовочных файлов всех классов пакета и путей для загрузки файлов реализаций классов пакета;
- `my_class.i` — интерфейс класса;
- `my_class.ph` — описание путей для загрузки заголовочных файлов открываемых классов и путей для загрузки файлов декларации классов пакета;
- `my_class.cl` — декларация класса;
- `my_class.pro` — реализация класса.

Собрать класс можно и вручную в одном файле.

Интерфейс класса содержит:

- определение констант и доменов интерфейса;
- объявление предикатов и свойств объектов.

Декларация класса содержит:

- определение констант и доменов класса, видимых извне класса;
- объявление предикатов и свойств класса, видимых извне класса.

Имплементация класса содержит:

- объявление фактов, хранящих состояние объекта;
- реализацию предикатов и свойств объектов, объявленных в интерфейсе класса;
- объявление фактов, хранящих состояние класса;
- реализацию предикатов и свойств класса, объявленных в декларации класса;
- определение констант и доменов, невидимых извне;
- декларацию и реализацию фактов, свойств и предикатов, невидимых извне.

Класс можно мысленно представить в виде фабрики. Назначением такой фабрики является производство объектов. Каждый выпускаемый объект имеет свое уникальное состояние, которое можно изменять во время жизни объекта с помощью объектных свойств и предикатов. Это состояние называется *объектным состоянием* (состоянием объекта). Объекты взаимодействуют с внешней средой с помощью предикатов, которые называются *объектными предикатами* (предикатами объекта). Объектные предикаты объявляются в интерфейсе, а их код описывается в реализации класса.

Кроме того, сама фабрика объектов находится в определенном состоянии, которое называется *состоянием класса* (классным состоянием). Фабрика может взаимодействовать с внешней средой с помощью *предикатов класса* (классных предикатов). Используя классные предикаты, можно воздействовать на состояние фабрики и при необходимости изменять свойства производимых объектов. Предикаты класса объявляются в декларации класса, а их код описывается в реализации класса.

Среди предикатов, определяющих работу фабрики, следует выделить предикат, производящий объекты. Этот предикат называется *конструктором*: `new()`. Если конструктор явно не описан, то будет работать конструктор по умолчанию. При необходимости можно объявить и определить несколько конструкторов разной арности. Так делают тогда, когда необходимо производить объекты со свойствами, задаваемыми аргументами конструктора.

Для работы фабрики бывает необходимо использовать вспомогательные константы, домены, предикаты и факты. Они называются *внутренними* константами, доменами, предикатами и фактами класса. Они невидимы извне, объявляются и определяются в реализации класса.

Все описанные здесь элементы класса существуют в одном экземпляре. В отличие от них, состояние каждого произведенного объекта уникально и существует в памяти отдельно от других объектов.

Интерфейс класса, названного, например, `my_interface`, имеет следующую структуру:

```
interface my_interface
open список_классов      % расширение области видимости
constants                 % раздел определения констант объекта
domains                  % раздел определения доменов объекта
properties               % раздел объявления свойств объекта
```

```
predicates % раздел объявления предикатов объекта
end interface my_interface % – optionalno
```

Неиспользуемые разделы могут быть опущены. При закрытии интерфейса его имя можно не указывать. Несколько классов могут иметь один интерфейс. В таком случае порождаемые этими классами объекты будут иметь один тип, которым и будет являться интерфейс классов.

При обращении к константам и доменам интерфейса указывается имя интерфейса — например: `interfaceName::constant`. При обращении к свойствам и предикатам интерфейса указывается объект — например: `object:property`.

Декларация класса, например, `my_class`, который производит объекты с интерфейсом `my_interface`, имеет следующую структуру:

```
class my_class : my_interface
open список_классов % расширение области видимости
constants % раздел определения констант, видимых извне
domains % раздел определения доменов, видимых извне
properties % раздел объявления свойств, видимых извне
constructors % раздел объявления собственных конструкторов
predicates % раздел объявления предикатов, видимых извне
end class my_class % – optionalno
```

Неиспользуемые разделы могут быть опущены. При закрытии декларации класса его имя можно не указывать. Имя класса может совпадать с именем интерфейса класса.

Имплементация класса `my_class`, который производит объекты с интерфейсом `my_interface`, имеет следующую структуру:

```
implement my_class
open список_классов % расширение области видимости
facts % раздел объектных фактов
domains % раздел определения доменов, невидимых извне
constants % раздел определения констант, невидимых извне
class facts – имя_БД % несколько именованных разделов БД,
% невидимых извне
class properties % раздел объявления свойств класса,
% невидимых извне
class predicates % раздел объявления предикатов класса, невидимых
% извне
clauses % раздел предложений для фактов, свойств и предикатов
% всего класса
end implement my_class – optionalno
```

Неиспользуемые в реализации класса разделы могут быть опущены. При закрытии реализации класса его имя можно не указывать.

Разделы фактов `facts` и `class facts` могут присутствовать только в реализации класса. Нельзя объявить факты в одном классе и пытаться напрямую обращаться

к ним из другого класса. Обращаться к фактам класса извне можно только опосредованно: либо через видимые свойства этого класса, либо через видимые предикаты этого класса.

19.1.1. Параметры состояния класса и объекта

Состояние объекта определяется некоторой совокупностью значений параметров, которые он хранит в течение своей жизни до тех пор, пока сборщик мусора не придет за ним со своей большой мусорной корзиной. Каждый объект, порожденный классом, имеет свое уникальное состояние. Будем считать, что состояние класса или объекта изменяется, когда изменяется значение хотя бы одного параметра. Изменение состояния класса или объекта может быть вызвано как извне объекта, так и изнутри. Метод доступа к параметрам состояния зависит от способа их хранения. Можно выделить как минимум три варианта хранения классом или объектом своих параметров:

- в виде фактов — этот вариант удобно использовать для организации доступа, как по чтению/записи, так и для доступа только по чтению или только по записи. Доступ к этим фактам осуществляется посредством свойств класса/объекта или посредством предикатов класса/объекта;
- в виде функций с одним значением, которое, по сути, является константой класса/объекта. Такой вариант можно использовать для доступа к параметру по чтению. Однако это утверждение справедливо только для тех функций, которые не взаимодействуют со сторонними программами или аппаратными устройствами;
- если класс/объект хранит параметры состояния в сторонних программах или аппаратных устройствах с доступом к ним по чтению и/или по записи, то для доступа к этим параметрам можно использовать свойства или предикаты класса/объекта. Характерным примером является использование API-функций для создания графического интерфейса и взаимодействие, например, с аудио- или видеоаппаратурой.

19.2. Объекты

19.2.1. Создание объектов

Объекты создаются посредством вызова конструктора. По умолчанию конструктором является предикат `new()`. Однако ничего не мешает сделать один или несколько собственных конструкторов, объявив их в разделе `constructors`. Явно объявленный конструктор `new()` заменяет одноименный конструктор по умолчанию. По своей сути конструктор — это функция, которая инициализирует внутреннее состояние объекта, т. е. создает в памяти все необходимые объектные факты и присваивает им начальные значения, определенные в имплементации класса. После этого конструктор возвращает уникальную ссылку на объект. С помощью такой ссылки Visual Prolog отличает факты одного объекта от фактов другого объекта.

Рассмотрим класс `my_class`, производящий объекты красного цвета. Пусть класс `my_class` имеет следующее определение:

```
interface my_interface
end interface my_interface

class my_class : my_interface
end class my_class

implement my_class
facts
    color : string := "красный".          % объектный факт color
end implement my_class
```

Создание объекта с помощью конструктора `new()` выглядит так:

```
Объект1 = my_class::new()
```

Здесь переменной `Объект1` присваивается ссылка на созданный объект. Типом этой переменной является интерфейс `my_interface`.

Ничто не мешает создать сразу несколько объектов:

```
Объект1 = my_class::new(),
Объект2 = my_class::new(),
Объект3 = my_class::new()
```

Однако все наши объекты хоть и различаются ссылками, но у каждого внутреннее состояние определяется красным цветом. И изменить это внутреннее состояние пока нечем — нет инструментов. И напрямую осуществить доступ к объектным фактам из других классов посредством предикатов `assert` или `retract` невозможно — это запрещено концепцией программирования классов, принятой в Visual Prolog.

Рассмотрим инструменты, дающие доступ к состоянию объекта. Таковыми могут быть объектные предикаты и/или свойства, которые объявляются в интерфейсе.

19.2.2. Объектные предикаты

Для начала в интерфейсе класса объявим предикаты чтения и записи значения факта `color`. А в разделе имплементации определим эти предикаты:

```
interface my_interface
predicates
    getColor : () -> string.    % получение значения объектного факта
    setColor : (string).        % установка нового значения объектного факта
end interface my_interface

class my_class : my_interface
end class my_class

implement my_class
facts
    color : string := "красный".          % объектный факт color
```

```
clauses
```

```
getColor () = color.          % чтение цвета из объектного факта
setColor(Цвет) :- color := Цвет. % запись цвета в объектный факт
end implement my_class
```

После создания объекта мы можем читать его цвет и даже заменять его другим цветом. При вызове объектных предикатов необходимо через двоеточие указывать, к какому объекту вы обращаетесь. Например, при чтении цвета первого объекта следует использовать вызов:

```
Цвет1 = Объект1:getColor()
```

А при записи нового значения цвета второго объекта следует использовать вызов:

```
Объект2:setColor("зеленый")
```

Таким образом, вначале указывается объект и через одно двоеточие — имя объектного предиката. Вот пример чтения и записи свойства `color` двух объектов:

```
Объект1 = my_class::new(),      % создали первый объект
Объект2 = my_class::new(),      % создали второй объект
Цвет1 = Объект1:getColor(),    % читаем цвет первого объекта
Цвет2 = Объект2:getColor(),    % читаем цвет второго объекта
writeln("% - %", Цвет1, Цвет2), % вывод: "красный - красный"
Объект1:setColor("зеленый"),   % пишем новый цвет первого объекта
Объект2:setColor("синий"),     % пишем новый цвет второго объекта
НовЦвет1 = Объект1:getColor(), % читаем цвет первого объекта
НовЦвет2 = Объект2:getColor(), % читаем цвет второго объекта
writeln("% - %", НовЦвет1, НовЦвет2) % вывод: "зеленый - синий"
```

19.2.3. Объектные свойства

Объектное свойство выражает один параметр и объявляется подобно объявлению факта-переменной, но, в отличие от факта-переменной, свойство не может быть `erroneous`. Однако факт, который описывает свойство, может быть `erroneous`.

И для чтения, и для записи значения некоторого параметра используется только одно свойство из раздела `properties`. Вызов свойства подобен вызову факта-переменной.

Пусть объектное свойство имеет имя `ownColor`. Это свойство надо объявить в интерфейсе класса в разделе `properties`. А в разделе реализации можно объявить однотипную факт-переменную `ownColor` и инициировать ее значением — например, "красный":

```
interface my_interface
properties
    ownColor : string.          % чтение/запись цвета
end interface my_interface

class my_class : my_interface
end class my_class
```

```

implement my_class
facts
    ownColor: string := "красный".           % объектный факт color
end implement my_class

```

Теперь для чтения свойства первого объекта можно использовать стиль фактов-переменных:

```
Цвет1 = Объект1:ownColor()
```

При записи нового значения цвета второго объекта можно использовать вызов:

```
Объект2:ownColor := "Белый"
```

При доступе к фактам можно совершать дополнительные действия или выполнять какие-либо проверки. Для этого свойство и факт-переменная должны иметь различные имена. Пусть объектное свойство имеет имя `ownColor`. А факт-переменная — имя `color`. Свойство `ownColor`, как и положено, объявим в интерфейсе класса. Код опишем в реализации класса. При чтении свойства название цвета будем всегда возвращать заглавными буквами, независимо от того, в каком регистре название цвета было записано:

```

interface my_interface
properties
    ownColor : string.          % чтение/запись цвета
end interface my_interface

class my_class : my_interface
end class my_class

implement my_class
open string
facts
    color : string := "красный".           % объектный факт color
clauses
    ownColor() = toUpperCase(color).      % чтение свойства объекта из факта
    ownColor(Цвет) :- color := Цвет.     % запись свойства объекта в факт
end implement my_class

```

Далее представлен пример чтения и записи свойства `ownColor` двух объектов, в котором красный цвет меняется на зеленый и синий, но при чтении названия цветов приводятся к верхнему регистру:

```

Объект1 = my_class::new(),           % создаем первый объект
Объект2 = my_class::new(),           % создаем второй объект
Цвет1 = Объект1:ownColor,            % читаем цвет первого объекта
Цвет2 = Объект2:ownColor,            % читаем цвет второго объекта
writeln("% - %\n", Цвет1, Цвет2),   % "КРАСНЫЙ - КРАСНЫЙ"
Объект1:ownColor := "зеленый",       % пишем новый цвет первого объекта
Объект2:ownColor := "синий",         % пишем новый цвет второго объекта
НовЦвет1 = Объект1:ownColor,          % читаем цвет первого объекта

```

```

НовЦвет2 = Объект2:ownColor,      % читаем цвет второго объекта
writef("% - \%n", НовЦвет1, НовЦвет2)  % "ЗЕЛЕНЫЙ - СИНИЙ"

```

При реализации свойств, доступных только по чтению, можно использовать функции. Например, свойство `вес` объявлено только по чтению и реализовано в виде нульварной функции `вес()=181.9`. Изменить значение 181.9 в течение всей жизни объекта невозможно. По сути, свойство `вес` является константой:

```

interface my_interface
properties
    вес : real (o).
end interface my_interface

class my_class : my_interface
end class my_class

implement my_class
clauses
    вес()=181.9.
end implement my_class

```

Вот пример чтения свойства `вес`:

```

Объект1 = my_class::new(),      % создаем первый объект
write(Объект1:вес)              % 181.9

```

Преимуществом свойств является их универсальность, которая заключается в использовании свойства, как для чтения, так и для записи некоторого параметра.

19.2.4. Удаление объектов

В Visual Prolog удаление объектов автоматически осуществляется сборщиком мусора (garbage collector). Он удаляет те объекты, на которые нет ссылок. Ссылка на объект существует в следующих случаях.

- Когда выполняется предложение предиката, в котором объект создан, но ссылка на объект не сохранена в факте:

```

clauses
p() :- Объект = my_class::new(),
Объект:pred_1(),
other_class::ppp(Объект),
...
Объект:pred_N().

```

По завершении выполнения предложения предиката считается, что объект будет удален. На самом деле объект может быть удален несколько позже — когда в очередной раз запустится сборщик мусора.

- Когда ссылка на объект сохранена в факте-переменной или в факте:

```

facts
мойОбъект : имя_интерфейса.

```

```

clauses
p() :- Объект = my_class::new(),
       мойОбъект := Объект,
       Объект:pred_1(),
       another_class::ppp(Объект),
       ...
       Объект:pred_N().

```

По завершении выполнения предложения предиката объект будет продолжать жить. Когда надобность в нем отпадет, ссылку на объект можно удалить

мойОбъект := erroneous

или применить предикат `retract/1`, если ссылка на объект была сохранена в факте.

Обратите внимание, что типом ссылки на объект является имя интерфейса класса, в котором этот объект был порожден.

Сборщик мусора можно вызвать принудительно:

`memory::garbageCollect()`

В этом случае все объекты, на которые нет ссылок, будут удалены незамедлительно.

19.3. Классы

19.3.1. Конструкторы

В наших предыдущих примерах создавались объекты только красного цвета. Конечно, этот цвет после создания объекта можно поменять на другой, но хорошо было бы иметь возможность создавать объекты заданного цвета. Для этого можно объявить конструктор `new(Цвет)` с одним аргументом, значение которого и будет определять цвет создаваемого объекта. Этот конструктор следует объявить в декларации класса в разделе `constructors`. А его исходный код описать в реализации класса:

```

interface my_interface
end interface my_interface

class my_class : my_interface
constructors
    new : (string Цвет).           % объявление конструктора
end class my_class

implement my_class
facts
    color : string := "красный".   % объектный факт color
clauses
    new(Цвет) :- color := Цвет.    % определение конструктора
end implement my_class

```

Сейчас мы уже можем создавать объекты, которые сразу будут иметь нужный нам цвет. Конечно, у нас осталась возможность пользоваться и неявным конструктором:

```
Объект1 = my_class::new("белый"),    % Объект1 белого цвета
Объект2 = my_class::new(),           % Объект2 красного цвета
Объект3 = my_class::new("синий")     % Объект3 синего цвета
```

19.3.2. Состояние класса

Дадим возможность нашему классу изменять предопределенный красный цвет, которым наделяется объект при рождении. В качестве состояния класса будем рассматривать цвет класса. Каков цвет класса, таков и цвет производимых объектов в том случае, когда объект создается без указания цвета. Если объект создается конструктором, в котором цвет указан, то цвет класса игнорируется. Для выполнения этой задумки объявим факт-переменную класса — например, classColor, в которой будем хранить цвет класса. Кроме этого нам необходим предикат для изменения цвета класса. Пусть таким предикатом будет setClassColor(Цвет).

И последнее. Так как логика создания объектов изменилась, нам необходимо явно определить конструктор new(), а также переписать код конструктора new(Цвет):

```
interface my_interface
end interface my_interface

class my_class : my_interface
constructors
    new : (string Цвет).           % объявление конструктора
predicates
    setclassColor : (string Цвет)  % предикат изменения цвета класса
end class my_class

implement my_class
facts
    color : string := erroneous.   % объектный факт color
class facts
    classColor : string := "красный". % классный факт classColor
clauses
    new() :- color := classColor.    % определение конструктора new()
    new(Цвет) :- color := Цвет.      % определение конструктора new(Цвет)
    setclassColor(Цвет) :- classColor := Цвет.
end implement my_class
```

Объектный факт color инициализирован значением erroneous (ошибочный). Это делается для того, чтобы при попытке использования этого факта можно было бы с помощью предиката isErroneous(color) проверить, какое значение имеет факт: ошибочное или легитимное.

Явно описанный конструктор new() заменяет неявный конструктор new(), которым мы пользовались ранее. Сейчас можно штамповывать цветные объекты партиями:

```

Объект1 = my_class::new(),           % объект красного цвета
Объект2 = my_class::new(),           % объект красного цвета
my_class::setclassColor("белый"),    % класс стал белым
Объект3 = my_class::new(),           % объект белого цвета
Объект4 = my_class::new(),           % объект белого цвета
Объект5 = my_class::new("синий"),    % объект синего цвета
Объект6 = my_class::new("желтый")   % объект желтого цвета

```

Как можете видеть по двум последним строчкам, у нас осталась возможность производить отдельные объекты с произвольно выбранным цветом.

Для изменения состояния класса вместо классного предиката `setclassColor(Цвет)` можно воспользоваться классным свойством. Синтаксис объявления, определения и использования классного свойства подобен синтаксису объектного свойства.

19.4. Наследование кода и поддержка интерфейсов

Рассмотрим класс, в котором создаются объекты заданного цвета и реализован доступ к цвету по чтению и записи посредством объектных предикатов `getColor()` и `setColor(Цвет)`:

```

interface colorItem
predicates
    getColor : () -> string.    % получение значения объектного факта
    setColor : (string).        % установка значения объектного факта
% установка нового значения объектного факта
end interface my_interface

class item : colorItem
constructors
    new : (string Цвет).          % объявление конструктора
end class item

implement item
facts
    color : string := "красный".    % объектный факт color
clauses
    new(Цвет) :- color := Цвет.
    getColor () = color.          % чтение цвета из объектного факта
    setColor(Цвет) :- color := Цвет. % запись цвета в объектный факт
end implement item

```

Пусть эти объекты требуются нам для игры, в которой их надо расставить на доске в заданные клетки. Клетки доски имеют координаты x,y . Для расстановки цветных объектов на доске по заданным координатам нам стоит написать новый класс, который будет порождать объекты с заданным цветом и координатами с помощью конструктора `new(Цвет, x, y)`.

Для того чтобы не писать этот класс «с чистого листа», мы можем воспользоваться готовым классом `item`, в котором часть требуемой функциональности уже реализована.

В нашем случае для использования класса `item` надо, во-первых, сообщить компилятору о том, что наш новый класс будет использовать объектные предикаты `getColor()` и `setColor(Цвет)`, объявленные в интерфейсе `colorItem`. И, во-вторых, нужно сообщить компилятору о том, что наш новый класс будет использовать исходный код из реализации класса `item`.

Использование стороннего интерфейса определяется ключевым словом `support` (поддержка), после которого указывается имя интерфейса. В нашем случае это: `support colorItem`. Поддержка сторонних интерфейсов касается лишь деклараций объектных свойств и предикатов и поэтому указывается только в интерфейсах классов.

Использование собственно самих предикатов и свойств стороннего класса определяется ключевым словом `inherits` (наследование), после которого указывается имя класса. В нашем случае это: `inherits item`. Наследоваться может лишь код, поэтому наследование указывается только в имплементации класса.

Назовем наш новый класс, производящий цветные объекты по заданным координатам, `factory`. А интерфейс этого класса пусть носит имя `unit`:

```
interface unit
  supports colorItem
  domains
    location = coord(integer X, integer Y).
  predicates
    getCoord : () -> location.
    setCoord : (location).
end interface unit
```

Интерфейс `unit` поддерживает интерфейс `colorItem`, т. е. как бы неявно содержит объявление предикатов `getColor()` и `setColor(Цвет)` для чтения и записи цвета объекта соответственно. Также интерфейс `unit` содержит явную декларацию собственных объектных предикатов `getCoord()` и `setCoord(Координаты)` для чтения и записи координат объекта соответственно. Кроме этого в интерфейсе `unit` объявлен домен `location` для представления координат `x, y`.

Декларация класса `factory` содержит только объявление конструктора:

```
class factory : unit
  constructors
    new : (string Цвет, integer X, integer Y).
end class factory
```

Имплементация класса `factory` наследует предикаты класса `item`, а также определяет предикаты собственного интерфейса:

```
implement factory
  inherits item
```

```

facts
    x : integer.
    y : integer.

clauses
    new(Цвет, X, Y) :-  

        item::new(Цвет),  

        X := X,  

        Y := Y.  

    getCoord() = coord(x,y).  

    setCoord(coord(X,Y)) :- x := X, y := Y.
end implement factory

```

Замечательной особенностью нашего класса `factory` является то, что он позволяет нам обращаться по чтению и записи к тем параметрам состояния объекта, которые не являются для класса `factory` родными, а наследуются из других классов. Примером тому является чтение цвета:

```

Объект = factory::new("Синий",1,-2),
write("Цвет: ",Объект:getColor()),nl

```

Создадим парочку объектов `O1` и `O2`, и поменяем координаты одного из них:

```

O1 = factory::new("Синий",1,-2),
writeln("Цвет: ~nКоординаты: ~n",O1:getColor(),O1:getCoord()),
O2 = factory::new("Красный",0,2),
writeln("Цвет: ~nКоординаты: ~n",O2:getColor(),O2:getCoord()),
O1:setCoord(factory::coord(1,-1)),
writeln("Цвет: ~nКоординаты: ~n",O1:getColor(),O1:getCoord())

```

В результате получим:

```

Цвет: Синий
Координаты: coord(1,-2)
Цвет: Красный
Координаты: coord(0,2)
Цвет: Синий
Координаты: coord(1,-1)

```

Вроде, все работает, но есть одна проблема — по завершении выполнения все объекты уничтожаются сборщиком мусора, т. к. ссылки на объекты мы нигде не сохранили.

19.5. Сохранение объектов

Сохраним ссылки на объекты в фактах с именем `шар`. Заодно, с помощью счетчика будем нумеровать объекты при их создании и сохранять в фактах `шар`, кроме ссылок на объекты, еще и их номера. Для этого объявим в разделе классных фактов имплементации класса `factory` факт-переменную `счетчик` и факт `шар(Номер, Ссылка)` для хранения ссылок на объекты с их номерами:

```
class facts
    счетчик : unsigned := 0.
    шар : (unsigned Номер, unit Ссылка).
```

А также объявим в декларации класса `factory` предикаты для сохранения и удаления ссылки на объект. И нам еще потребуется функция получения объекта по его номеру:

```
predicates
    saveObject : (unit Объект).
    deleteObject : (unit Объект).
    getNumberObject : (unsigned Номер) -> unit determ.
```

Расширим раздел `clauses` имплементации класса `factory` описаниями объявленных предикатов:

```
saveObject(Объект) :- счетчик := счетчик+1,
    asserta(шар(счетчик, Объект)).
deleteObject(Объект) :- retractall(шар(_, Объект)).
getNumberObject(Номер) = Объект :- шар(Номер, Объект), !.
```

Имея описанную функциональность, мы можем создавать объекты и, при необходимости, оставлять жить созданные объекты до тех пор, пока они нам нужны. К примеру, создадим три объекта, сохранив жизнь первому и третьему:

```
Объект1 = factory::new("Синий", 1, -2),
factory::saveObject(Объект1),
Объект2 = factory::new("Красный", 0, 2),
Объект3 = factory::new("Красный", 2, 2),
factory::saveObject(Объект3)
```

Когда настанет время освободить память от объектов с первым и третьим порядковыми номерами, достаточно вызвать код:

```
if O1 = factory::getNumberObject(1)
    then factory::deleteObject(O1) end if,
if O3 = factory::getNumberObject(3)
    then factory::deleteObject(O3) end if
```

19.6. Операции над всеми живущими объектами

Сохранение ссылок на объекты не только продлевает жизнь объектов, но и позволяет выполнять различные операции над всеми живущими объектами. Например, мы можем легко получить список номеров объектов, имеющих заданный цвет, чтобы в дальнейшем совершить над ними какое-либо действие. Для этого надо объявить новую классную функцию:

```
colorNumbers : (string Цвет) -> unsigned* СписокНомеров.
```

И описать ее в имплементации класса `factory`:

```
colorNumbers(Цвет) =  
    [Номер || шар(Номер, Объект), Объект::getColor()=Цвет].
```

Следующий код создает три объекта: синего, черного и красного цвета соответственно, сохраняет им жизнь, потом меняет цвет второго объекта и вызывает функцию получения списка номеров объектов синего цвета:

```
Объект1 = factory::new("Синий", 1, -2),  
factory::saveObject(Объект1),  
Объект2 = factory::new("Черный", 0, 2),  
factory::saveObject(Объект2),  
Объект3 = factory::new("Красный", 2, 2),  
factory::saveObject(Объект3),  
Объект2::setColor("Синий"),           % замена цвета  
L = factory::colorNumbers("Синий"),      % получение списка  
write("Список объектов синего цвета: ", L) % вывод [2,1]
```

В результате на экран будет выведен список `[2,1]`, означающий, что объекты, имеющие второй и первый порядковые номера, имели в момент вызова функции `colorNumbers` синий цвет.

Приведенное здесь описание классов Visual Prolog не является полным, но дает необходимые знания для освоения обобщенного программирования в Visual Prolog, изложенного в главе 20.

19.7. Примеры использования классов

ПРИМЕР 19.1. В качестве примера класса опишем завод, выпускающий объекты-автомобили с цветом кузова, задаваемым при создании автомобиля. Класс- завод назовем именем `factory`. А объекты-автомобили опишем в интерфейсе с именем `car`.

Цвет кузова автомобиля будем хранить в объектном факте `color`. Пусть объекту будет разрешено менять свой цвет с помощью изменения свойства `ownColor`.

В интерфейсе `car` объявим свойство `ownColor`, с помощью которого можно читать или изменять цвет кузова автомобиля:

```
interface car  
properties  
    ownColor : string.          % чтение/запись цвета кузова  
end interface car
```

В декларации класса `factory` объявим конструктор, который будет создавать объекты с указанным цветом кузова:

```
class factory : car  
constructors  
    new : (string Цвет).  
end class factory
```

В реализации класса осталось описать исходный код для объявленных предикатов:

```
implement factory
facts
    color : string.                      % объектный факт color
clauses
    new(Цвет) :- color := Цвет.          % конструктор объекта
    ownColor() = color.                  % чтение свойства объекта из факта
    ownColor(Цвет) :- color := Цвет.    % запись свойства объекта в факт
end implement factory
```

Для того чтобы воспользоваться услугами нашего завода, необходимо из какого-либо стороннего класса вызывать конструкторы для создания автомобиля и при необходимости изменять цвет кузова. Пусть таким сторонним классом будет консольное приложение. В этом примере создаются три автомобиля с различным цветом кузова, после чего цвет кузова первого автомобиля перекрашивается в белый цвет.

```
run() :-
    Авто1 = factory::new("Зеленый"),      % создание Авто1
    writef("Цвет Авто1: %", Авто1:ownColor), nl,
    Авто2 = factory::new("Синий"),         % создание Авто2
    writef("Цвет Авто2: %", Авто2:ownColor), nl,
    Авто3 = factory::new("Голубой"),       % создание Авто3
    writef("Цвет Авто3: %", Авто3:ownColor), nl,
    Авто1:ownColor := "Белый",            % изменение цвета Авто1
    writef("Цвет Авто1: %", Авто1:ownColor).
```

После запуска программы будет выведен следующий результат:

```
Цвет Авто1: Зеленый
Цвет Авто2: Синий
Цвет Авто3: Голубой
Цвет Авто1: Белый
```

ПРИМЕР 19.2. Расширим пример 19.1, введя новый параметр автомобиля — мощность двигателя. Причем мощность двигателя будет задаваться заводом-изготовителем. Для этого введем новый объектный факт `ownPower`, в котором будет храниться значение мощности собственного двигателя объекта. А также введем классный факт `power`, в котором будет храниться значение мощности двигателей, устанавливаемых на производимые в текущее время автомобили. Пусть текущим значением мощности будет 150 л. с.

Кроме этого стоило бы дать объекту возможность читать значение мощности собственного двигателя, а заводу — возможность изменять значение мощности устанавливаемых двигателей. Для этого расширим интерфейс класса предикатом `getOwnPower` для чтения мощности собственного двигателя:

```

interface car
properties
    ownColor : string.                                % цвет кузова
predicates
    getOwnPower : () -> unsigned Мощность.      % мощность своего двигателя
end interface car

```

В декларацию класса добавим предикат `setPower` для изменения мощности устанавливаемых на автомобили двигателей:

```

class factory : car
constructors
    new : (string Цвет).
predicates
    setPower : (unsigned Мощность). % установка нового значения мощности
end class factory

```

В реализации класса изменим конструктор объектов так, чтобы он на каждый создаваемый автомобиль устанавливал двигатель предопределенной мощности. Это предопределенное значение мощности хранится в классном факте `power`. Кроме этого следует добавить объявление объектного факта `ownPower`, классного факта `power` и исходный код для введенных предикатов `getOwnPower` и `setPower`:

```

implement factory
facts
    color : string.                                % объектный факт color
    ownPower : unsigned.                          % объектный факт ownPower
class facts
    power : unsigned := 150. % текущая мощность двигателей - 150 л. с.
clauses
    new(Цвет) :- color := Цвет, ownPower := power. % конструктор
                                                % объекта
    ownColor() = color.                         % чтение свойства
                                                % объекта из факта
    ownColor(Цвет) :- color := Цвет.           % запись свойства объекта в факт
    getOwnPower() = ownPower.                   % чтение собственной мощности
    setPower(Мощность) :- power := Мощность. % установка
                                                % новой мощности
end implement factory

```

Используем описанный класс для создания четырех автомобилей. Первые два создаются с предопределенным значением мощности двигателя 150 л. с. Далее предопределенное значение мощности меняется на 180 л. с. и следующая пара автомобилей создается уже с двигателем новой мощности 180 л. с.:

```

run() :-
    Авт01 = factory::new("Зеленый"), % создание Авт01
    writef("Цвет Авт01: %", Авт01:ownColor), nl,
    writef("Мощность Авт01: % л. с.", Авт01:getOwnPower()), nl, nl,

```

```

Авто2 = factory::new("Синий"),      % создание Авто2
writef("Цвет Авто2: %", Авто2:ownColor),nl,
writef("Мощность Авто2: % л. с.",Авто2:getOwnPower()),nl,nl,
factory::setPower(180),           % устанавливаем мощность 180 л. с.

```

```

Авто3 = factory::new("Красный"),    % создание Авто3
writef("Цвет Авто3: %", Авто3:ownColor),nl,
writef("Мощность Авто3: % л. с.",Авто3:getOwnPower()),nl,nl,

```

```

Авто4 = factory::new("Голубой"),    % создание Авто4
writef("Цвет Авто4: %", Авто4:ownColor),nl,
writef("Мощность Авто4: % л. с.",Авто4:getOwnPower()),nl,nl,

```

```

Авт01:ownColor := "Белый",          % изменение цвета Авт01
writef("Цвет Авт01: %", Авт01:ownColor).

```

После запуска программы будет выведен следующий результат:

```

Цвет Авт01: Зеленый
Мощность Авт01: 150 л. с.

```

```

Цвет Авт02: Синий
Мощность Авт02: 150 л. с.

```

```

Цвет Авт03: Красный
Мощность Авт03: 180 л. с.

```

```

Цвет Авт04: Голубой
Мощность Авт04: 180 л. с.

```

```

Цвет Авт01: Белый

```

ПРИМЕР 19.3. Иногда хотелось бы иметь возможность создать автомобиль по индивидуальному заказу — к примеру, с уникальным значением мощности, отличающейся от предопределенной заводом-изготовителем. Для этого достаточно добавить еще один конструктор, в котором, кроме цвета, указывать еще и желаемую мощность: new(Цвет, Мощность).

Для полноты примера добавим в состояние класса текстовые данные о ценах на запасные части. Также дадим возможность владельцам оформлять рекламации на их автомобили, которые завод-изготовитель сможет при необходимости читать.

В интерфейс класса добавим предикат чтения прайс-листа и предикат создания рекламации на автомобиль с указанием идентификатора, по которому завод сможет отличить данный автомобиль от других. Для этого мы используем тип `object`, представляющий собой уникальный идентификатор объекта:

```

interface car
properties
  ownColor : string.

```

```

predicates
    getOwnPower : () -> unsigned Мощность.
    getInfo : () -> string ПрайсЛист.
    addClaim : (object Авто, string Рекламация).
end interface car

```

Декларация нашего класса обзавелась новым свойством `info`, еще одним конструктором и предикатом чтения рекламации `readClaim`:

```

class factory : car
properties
    info : string.
constructors
    new : (string Цвет).
    new : (string Цвет, unsigned Мощность).
predicates
    setPower : (unsigned Мощность).
    readClaim : (object Авто, string Рекламация) nondeterm (i,o).
end class factory

```

Реализация класса пополнилась объявлением классных фактов `priceList` и `claim`, а также исходным кодом для конструктора `new(Цвет, Мощность)` и для всех новых предикатов:

```

implement factory
facts
    color : string.
    ownPower : unsigned.
class facts
    power : unsigned := 150.
    priceList : string := "Прейскурант...".
    claim : (object Авто, string Рекламация).
clauses
    new(Цвет) :- color := Цвет, ownPower := power.
    new(Цвет, Мощность) :- color := Цвет, ownPower := Мощность.

    ownColor() = color.
    ownColor(Цвет) :- color := Цвет.

    getOwnPower() = ownPower.
    getInfo() = priceList.
    addClaim(Авто, Рекламация) :- asserta(claim(Авто, Рекламация)).

    info() = priceList.
    info(Инф) :- priceList:=Инф.

    setPower(Мощность) :- power := Мощность.

    readClaim(Авто, Рекламация) :- claim(Авто, Рекламация).
end implement factory

```

Посмотрим, как можно пользоваться новой функциональностью. Создадим пару автомобилей с предустановленной на заводе мощностью двигателя 150 л. с. Потом изменим заводскую предустановку мощности на 180 л. с. Затем вызовем конструктор, создающий автомобиль по индивидуальному заказу с мощностью двигателя 210 л. с. Для проверки новой заводской установки мощности создадим четвертый автомобиль. И, наконец, добавим две рекламации и прочитаем прайс-лист:

```
run() :-  
    Авто1 = factory::new("Зеленый"),           % создание Авто1  
    writef("Цвет Авто1: %\n", Авто1:ownColor),  
    writef("Мощность Авто1: % л. с.\n\n", Авто1:getOwnPower()),  
  
    Авто2 = factory::new("Синий"),           % создание Авто2  
    writef("Цвет Авто2: %\n", Авто2:ownColor),  
    writef("Мощность Авто2: % л. с.\n\n", Авто2:getOwnPower()),  
  
    factory::setPower(180),                  % новая мощность 180 л. с.  
  
    Авто3 = factory::new("Красный", 210),     % создание Авто3  
    writef("Цвет Авто3: %\n", Авто3:ownColor),  
    writef("Мощность Авто3: % л. с.\n\n", Авто3:getOwnPower()),  
  
    Авто4 = factory::new("Голубой"),         % создание Авто4  
    writef("Цвет Авто4: %\n", Авто4:ownColor),  
    writef("Мощность Авто4: % л. с.\n\n", Авто4:getOwnPower()),  
  
    Авто1:addClaim(Авто1, "Рекламация об Авто1 ..."),  
    ( factory::readClaim(Авто1, Текст1),  
      writef("Рекламация\nОбъект: %, \nТекст: %", Авто1, Текст1), !;  
      writef("Рекламация о % не найдена\n", Авто1) ),  
  
    Авто2:addClaim(Авто2, "Рекламация об Авто2 ..."),  
    ( factory::readClaim(Авто2, Текст2),  
      writef("Рекламация\nОбъект: %, \nТекст: %", Авто2, Текст2), !;  
      writef("Рекламация о % не найдена\n", Авто2) ),  
  
    writef("\nПрайс лист: % ", factory::info).
```

После запуска программы будет выведен следующий результат:

```
Цвет Авто1: Зеленый  
Мощность Авто1: 150 л. с.
```

```
Цвет Авто2: Синий  
Мощность Авто2: 150 л. с.
```

```
Цвет Авто3: Красный  
Мощность Авто3: 210 л. с.
```

Цвет Авто4: Голубой

Мощность Авто4: 180 л. с.

Рекламация

Объект: 009D7FE0,

Текст: Рекламация об Автол ...

Рекламация

Объект: 009D7FD0,

Текст: Рекламация об Авто2 ...

Прайс лист: Прейскурант...

ГЛАВА 20



Обобщенное программирование

Обобщенное программирование (generic programming) позволяет программисту описывать интерфейсы и классы, имеющие дело с данными различного типа. При описании интерфейса или класса вместо конкретных типов указываются обобщенные типы. При вызове же предиката обобщенного интерфейса или класса ему передаются конкретные типы данных. Обобщенный тип в Visual Prolog обозначается заглавным идентификатором с префиксом @, роль которого играет символ @, именуемый на сленгге программистов «собака».

Обобщенное программирование впервые было применено в языке Ада в 1983 году, а термин был предложен и описан Дэвидом Массером и Александром Александровичем Степановым. В Visual Prolog обобщенное программирование обеспечивает создание обобщенных интерфейсов и классов. С их помощью в программе можно использовать объектные факты, содержащие данные таких типов, которые будут указаны при создании этих объектов.

20.1. Обобщенные интерфейсы

Обобщенный интерфейс строится на основе обычного интерфейса путем добавления к имени интерфейса списка обобщенных типов, которых будет достаточно для построения всех типов данных, используемых в этом интерфейсе. Областью видимости обобщенных типов является весь контекст интерфейса. Синтаксис перечня обобщенных типов аналогичен синтаксису формальных типов полиморфных доменов, рассмотренных в главе 14.

В следующем примере показан обобщенный интерфейс, который включает два обобщенных типа @A и @B. Из них образуются еще два типа @A* и @B*, используемых при объявлении объектных предикатов:

```
interface aaa{@A,@B}
predicates
    pp : (@A,@B,@A*,@B*).
    qq : (@A,@B) -> @A*.
    ww : (@A,@B) -> @B*.
end interface aaa
```

При использовании этого интерфейса вместо обобщенных типов {@A, @B} надо подставить конкретные типы. Например, подставляя вместо @A тип integer, а вместо @B — тип string, мы получим интерфейс:

```
interface aaa{integer,string}
predicates
    pp : (integer, string, integer*, string*).
    qq : (integer, string) -> integer*.
    ww : (integer, string) -> string*.
end interface aaa
```

Подобно этому примеру можно использовать интерфейс с другими типами. Таким образом, описав обобщенный интерфейс один раз, мы имеем целое множество интерфейсов. Без обобщенных интерфейсов в проекте Visual Prolog потребовалось бы явно описывать все требуемые интерфейсы.

Обобщенные интерфейсы могут поддерживать другие обобщенные интерфейсы. Следующие примеры поддержки обобщенных интерфейсов являются правильными:

```
interface xxx{@P} supports yyy{@P} % @P связан
end interface xxx
```

```
interface xxx{@P} supports yyy{integer, @P**} % @P связан
end interface xxx
```

А вот примеры неправильной поддержки обобщенных интерфейсов:

```
interface xxx{@P} supports yyy{@Q} % @P и @Q не связаны
end interface xxx
```

```
interface xxx supports yyy{@P} % @P не связан
end interface xxx
```

20.2. Обобщенные классы

Обобщенные классы, подобно обобщенным интерфейсам, используют список обобщенных типов. Этот список должен иметь те же типы, что и интерфейс. Областью видимости обобщенных типов является как класс, так и его реализация.

Обобщенные классы конструируют объекты обобщенного интерфейсного типа. Для этого обобщенный класс имеет обобщенный конструктор. Конкретный тип конструируемого объекта либо задается программистом явно, либо выводится компилятором из контекста программы.

В следующем примере объявляется обобщенный класс bbb{@A, @B}, производящий объекты интерфейсного типа aaa{@A, @B} с помощью обобщенного конструктора, заданного по умолчанию:

```
class bbb{@A, @B} : aaa{@A, @B}
end class bbb
```

При вызове обобщенного конструктора можно конкретизировать типы данных создаваемого объекта:

```
Объект1 = bbb{real,string}::new(),
Объект2 = bbb{integer,char*}::new(),
Объект3 = bbb::new()
```

Типы данных допускается не указывать. В этом случае Visual Prolog будет пытаться вывести эти типы на основании явно указанных значений, которыми инициализируются объектные факты, или на основании операций, которые выполняются над аргументами объектных фактов.

Для обобщенных классов не допускается:

- использовать одинаковые имена для классов с разной арностью (числом обобщенных типов);
- если класс и интерфейс имеют одинаковое имя, то класс должен создавать объекты этого интерфейса;
- использовать обобщенные типы при объявлении предикатов класса. Эти типы допускается использовать только при объявлении доменов, констант и конструкторов класса.

Следующий пример демонстрирует допустимые и недопустимые декларации:

```
class xxx{@P} : xxx{@P}
domains
    list = @P*.           % допустимо
constants
    empty : @P* = [].    % допустимо
constructors
    new : (@P).          % допустимо
predicates
    p : (@P).            % недопустимо, p - предикат класса, а не объекта
end class xxx
```

20.3. Обобщенные реализации

Обобщенная реализация должна иметь тот же список обобщенных типов, что и класс. Причем, и имена типов, и порядок их перечисления должны быть одинаковыми как в классе, так и в реализации.

В реализации класса обобщенные типы должны использоваться только при объявлении доменов, констант и конструкторов, а также при определении объектных фактов, предикатов и свойств.

Вот пример правильного объявления обобщенного класса и обобщенной реализации:

```
class xxx{@A,@B} : aaa{@A,@B}
...
end class xxx
```

```
implement xxx{@A,@B}
...
end implement xxx
```

Следующий пример показывает допустимые и недопустимые варианты использования обобщенных типов в реализации класса:

```
implement xxx{@P}
domains
    list = @P*.           % допустимо
constants
    empty : @P* = [].     % допустимо
constructors
    new : (@P Init).     % допустимо
predicates
    objectPred : (@P X). % допустимо
class predicates
    classPred : (@P X).  % недопустимо, classPred - предикат класса,
                          % а не объекта
facts
    objectFact : (@P).    % допустимо
class facts
    classFact : (@P).     % недопустимо, classFact - факт класса,
                          % а не объекта
...
end implement xxx
```

20.4. Пример обобщенной очереди

Рассмотрим преимущества обобщенного программирования на примере класса, который создает очередь, а также выполняет операции помещения элемента в очередь и выборки из очереди. Для контраста вначале представим обычный класс, реализующий очередь целых чисел, а потом обобщенный класс, реализующий очередь объектов произвольного типа.

ПРИМЕР 20.1. Здесь представлен класс `queue_integer`, реализующий очередь целых чисел. В декларациях объектных предикатов и факта явно указан тип элементов очереди: `integer`. Операция выборки числа из очереди детерминированная. Если очередь не пуста, то функция `tryGet` вернет число, стоящее первым в очереди, удалив его оттуда. Если очередь пуста, то функция `tryGet` завершится неудачей. Сама очередь организована в виде набора объектных фактов `queue_fact`. При помещении числа в очередь это число добавляется к базе фактов с конца. При выборке числа из базы фактов удаляется самый первый факт, если база не пуста.

```
interface queue_integer
predicates
    insert : (integer).          % помещение числа в очередь
    tryGet : () -> integer determ. % извлечение числа из очереди
end interface queue_integer
```

```

class queue_integer : queue_integer
end class queue_integer

implement queue_integer
facts
    queue_fact : (integer).           % объектный факт
clauses
    insert(Elem) :- assert(queue_fact(Elem)).
    tryGet() = Elem:- retract(queue_fact(Elem)),!.
end implement queue_integer

```

Использовать этот класс можно следующим образом:

```

run() :-
    Q = queue_integer::new(),      % создаем очередь Q
    H = queue_integer::new(),      % создаем очередь H
    Q:insert(4),                  % помещаем в очередь Q число 4
    Q:insert(5),                  % помещаем в очередь Q число 5
    H:insert(9),                  % помещаем в очередь H число 9
    if X=Q:tryGet() then write(X)   % выборка из очереди Q
        else write("Очередь пуста") end if, nl,
    if Y=Q:tryGet() then write(Y)   % выборка из очереди Q
        else write("Очередь пуста") end if, nl,
    if Z=Q:tryGet() then write(Z)   % выборка из очереди Q
        else write("Очередь пуста") end if, nl,
    Q:insert(8),                  % помещаем в очередь Q число 8
    if W=Q:tryGet() then write(W)   % выборка из очереди Q
        else write("Очередь пуста") end if,
    _=readchar().

```

Результат:

```

4
5
Очередь пуста
8

```

Если в проекте, кроме очереди целых чисел, надо иметь очереди, содержащие элементы других типов, то нам потребовалось бы продублировать рассмотренный класс и интерфейс `queue_integer` с другими типами данных. Однако обобщенные классы и интерфейсы позволяют написать такой класс единожды и использовать его для организации очередей с нужными типами данных.

ПРИМЕР 20.2. Здесь представлен обобщенный класс `queue`, реализующий очередь элементов произвольного типа.

```

interface queue{@A}
predicates
    insert : (@A).                 % помещение числа в очередь
    tryGet : () -> @A determ.     % выборка числа из очереди
end interface queue

```

```

class queue{@Elem} : queue{@Elem}
end class queue

implement queue{@Elem}
facts
  queue_fact : (@Elem) .           % объектный факт
clauses
  insert(Value) :- assert(queue_fact(Value)) .
  tryGet() = Value :- retract(queue_fact(Value)), ! .
end implement queue

```

Обратите внимание, что имя обобщенного типа @A, указанное в интерфейсе, может не совпадать с именем, указанным в классе @Elem. На самом деле обобщенный тип интерфейса и обобщенный тип класса связаны друг с другом в строке:

```
class queue{@Elem} : queue{@Elem}
```

Здесь обобщенный тип интерфейса имеет такое же имя, как и обобщенный тип класса — @Elem. И вообще, хорошим тоном считается указывать одинаковые имена обобщенных типов как в интерфейсе, так и в классе.

Использовать обобщенный класс можно следующим образом:

```

run() :-
  G = queue{real}::new(),    % создаем очередь G
  G:insert(14),             % помещаем в очередь G число 14.0
  G:insert(5.8),            % помещаем в очередь G число 5.8
  H = queue::new(),         % создаем очередь H
  H:insert("abc"),          % помещаем в очередь строку "abc"
  H:insert("zz"),           % помещаем в очередь строку "zz"
  if A=G:tryGet() then write(A)      % выборка из очереди G
    else write("Очередь пуста") end if, nl,
  if B=H:tryGet() then write(B)      % выборка из очереди H
    else write("Очередь пуста") end if,
  _=readchar().

```

Результат:

```

14
abc

```

Очередь G создана с явным указанием типа real, который сразу подставляется вместо обобщенного типа @Elem. При создании очереди H тип не указан. В этом случае Visual Prolog определяет тип данных при первом обращении к очереди по записи. Таким обращением является вызов предиката H:insert("abc"), аргумент которого принадлежит к домену string. Следовательно, очередь H является очередью строк.

часть III



Средства профессионального программирования

Глава 21. Многопоточность

Глава 22. Доступ к API-функциям Windows

Глава 23. Разработка и использование DLL

Глава 24. Отладка приложений

ГЛАВА 21



Многопоточность

Visual Prolog имеет средства организации *многопоточности* (multithreading). Эти средства содержатся в классе `multiThread`. В этой главе мы коснемся не всех возможностей работы с потоками (тредами), а только основных способов создания потоков и доступа одновременно выполняемых потоков к разделяемым ресурсам.

21.1. Основные операции с потоками

21.1.1. Создание потоков

Для создания потока средствами Visual Prolog надо определить предикат, который должен выполняться этим потоком, и вызвать конструктор потока, передав ему этот предикат в качестве аргумента. Конструктор создаст поток в очереди задач операционной системы Windows. С точки зрения языка программирования созданный поток рассматривается как объект. Поэтому конструктор вернет ссылку на созданный объект, который, по сути, является потоком. После выполнения предиката поток автоматически удаляется из системной очереди.

В Visual Prolog есть три основных конструктора для создания потока:

- `Object = thread::start(Predicate)` — создает поток в системной очереди задач и размещает в нем код `Predicate`, после чего `Predicate` незамедлительно начинает использовать системные ресурсы вместе с другими активными потоками. Кроме этого конструктор возвращает ссылку на созданный поток `Object`. При создании потока можно указать размер стека, выделяемый потоку:

`Object = thread::start(Predicate, StackSize)` — создает поток в системной очереди задач и выделяет ему стек размером `StackSize` байтов.

- `Object = thread::createSuspended(Predicate)` — создает поток в приостановленном (отложенном) состоянии и размещает в нем код `Predicate`. `Object` — ссылка на созданный поток. Отложенный поток не использует системные ресурсы до тех пор, пока для `Object` не будет выполнен специальный предикат `Object:resume()`. Этот предикат разрешает (возобновляет) работу потока `Object`. При создании потока можно указать размер стека, выделяемый потоку:

`Object = thread::createSuspended(Predicate,StackSize)` — создает поток в приостановленном (отложенном) состоянии и выделяет ему стек размером `StackSize` байтов.

- `Object = thread::attach(ForeignThreadHandle)` — создает в Visual Prolog объект внешнего потока, дескриптор которого `ForeignThreadHandle` известен. Внешний поток — это любой поток, который не создан конструкторами `start` или `createSuspended`.

Аргумент `Predicate` может быть определен как именованный предикат или задан в виде анонимного предиката. Процесс может иметь несколько потоков, одновременно выполняющих один и тот же предикат.

21.1.2. Завершение потоков

Имеются три варианта завершения потока: когда выполнен код потока, по таймеру или принудительно.

- Для определения момента завершения потока служит функция:

```
EventCode = Object:wait()
```

Эта функция ожидает завершения потока столь долго, сколько времени выполняется поток. После успешного завершения потока она возвращает код завершения `EventCode`, равный нулю.

- Потоку можно назначить максимальное время выполнения, по истечении которого поток будет прерван, если только до этого срока он не завершится самостоятельно. Для завершения потока по таймеру служит функция:

```
EventCode = Object:wait(Milliseconds)
```

- Для принудительного завершения потока служит предикат:

```
Object:terminate(ExitCode)
```

Значение `ExitCode` задает программист для того, чтобы определить причину принудительного завершения.

21.1.3. Приостановка и возобновление потоков

Каждый поток имеет счетчик. Этот счетчик автоматически инкрементируется, когда вызывается предикат приостановки потока `Object`:

```
PreviousCount = Object:suspend()
```

но не более чем до 128. Счетчик декрементируется, когда вызывается предикат возобновления потока `Object`:

```
PreviousCount = Object:resume()
```

`PreviousCount` — значение счетчика до его инкремента/декремента предикатами `suspend/resume` соответственно. Выполнение потока продолжится, когда значение счетчика станет равным нулю. Таким образом, счетчик играет роль стека остановок

потока. Сколько раз поток приостановлен, столько же раз он должен быть возобновлен для того, чтобы продолжить свое выполнение. Если применить предикат возобновления потока к уже работающему потоку, то ничего не произойдет, и значение счетчика останется равным нулю.

21.1.4. Примеры создания потока

В Visual Prolog при создании потока ему можно передать входные параметры через переменные или через факты. Получить же результат можно через факты.

ПРИМЕР 21.1. Следующий фрагмент кода демонстрирует правильное создание потока и вывод результатов работы после его завершения. Исходными данными является список чисел, который передается в поток с помощью переменной `s`. Анонимный предикат, который выполняется в потоке, сортирует список `S` и сохраняет результат в факте `s`. Стоит заметить, что в этом примере передать список `[2,3,1]` в поток можно и через факт `s`, предварительно инициировав его значением `[2,3,1]`.

```

implement main
    open core,console
class facts
    s:integer* := [].           % факт для передачи данных в поток
clauses
run() :-
    S=[2,3,1],                  % список для сортировки
    Thread = thread::start({:-s:=list::sort(S)}),  % создание потока
    _ = Thread:wait(),          % ожидание завершения потока
    writeln("s = % \n",S),       % вывод результата
    _ = readchar().
end implement main
goal
    console::run(main::run).

```

Если предикат ожидания завершения потока не использовать (закомментировать), то после запуска потока сразу выполнится предикат вывода результата, и мы на экране увидим пустой список `s`. Дело в том, что предикат сортировки выполняется в своем потоке, параллельно с потоком, в котором выполняется предикат `run()`. Поэтому, когда список будет сортироваться в одном потоке, в другом потоке предикат `writeref("s = % \n", s)` преждевременно выведет результат — пустой список. Для исключения такой коллизии вывод результата надо делать после завершения потока, в котором производятся вычисления.

ПРИМЕР 21.2. Следующий фрагмент кода демонстрирует именованный предикат `ppp`, который будет использован в потоке. Исходные данные передаются в поток с помощью факта `s`, и через него же результат возвращается в основную программу.

```
implement main
    open core,console
class facts
    s:integer* := []. % факт для передачи данных в поток
```

```

class predicates
    ppp : core:::runnable.
clauses
    ppp() :- s:=list:::sort(s).
    run() :- s:=[2,3,1], % список для сортировки
            Thread = thread:::start(ppp), % создание потока
            _ = Thread:wait(), % ожидание завершения потока
            writeln("s = % \n",s), % вывод результата
            _ = readchar().
end implement main
goal
    console:::run(main:::run).

```

ПРИМЕР 21.3. Исходные данные в поток можно передать и с помощью переменной. Именованный предикат следует объявить с входным аргументом ppp:(integer*). Результат возвращается в основную программу опять-таки через факт s.

```

implement main
    open core,console
class facts
    s:integer* := [].
class predicates
    ppp : (integer*).
clauses
    ppp(L) :- s:=list:::sort(L).
    run() :- L= [2,3,1],
            Thread = thread:::start({:-ppp(L)}),
            _ = Thread:wait(),
            writeln("s = % \n",s),
            _ = readchar().
end implement main
goal
    console:::run(main:::run).

```

ПРИМЕР 21.4. Следующий фрагмент кода демонстрирует создание отложенного потока, его возобновление и ожидание завершения.

```

implement main
    open core,console
class facts
    s:integer* := [2,3,1].
class predicates
    ppp : core:::runnable.
clauses
    ppp() :- s:=list:::sort(s).
    run() :-
        T = thread:::createSuspended(ppp), % создание отложенного потока
        _ = T:resume(), % возобновление выполнения потока
        _ = T:wait(), % ожидание завершения потока

```

```

writef("s = % \n",s),
_ = readchar().
end implement main
goal
    console::run(main::run).

```

21.2. Мониторы

Одновременно выполняющиеся потоки зачастую могут совместно работать с общими ресурсами. Или, как говорят, разделяют ресурсы. *Разделяемый ресурс* — это ресурс, доступ к которому разрешен многим процессам. К разделяемым ресурсам относятся аппаратные устройства (как правило, устройства ввода/вывода), файлы и папки, переменные в оперативной памяти. Здесь следует отметить, что Visual Prolog осуществляет безопасный доступ к следующим разделяемым ресурсам:

- факты-переменные;
- факты с режимом детерминизма `single` или `determ`.

Кроме того, есть другая группа разделяемых ресурсов, доступ к которым небезопасен:

- факты с режимом детерминизма `nondeterm`;
- элементы графического пользовательского интерфейса;
- потоки и, следовательно, файловая система;
- программный интерфейс доступа к БД (`odbcConnection/odbcStatement`).

ПРИМЕР 21.5. Продемонстрируем необходимость правильной организации одновременного доступа двух потоков к устройству вывода на экран. Пусть поток `Thread1` выводит сообщение `aaa`, а поток `Thread2` — сообщение `bbb`. Кроме того, пусть потоки выводят на экран информацию о номере потока, а также о начале и конце вывода.

```

implement main
    open core,console
class predicates
    myWrite : (unsigned,...).
clauses
    myWrite(N,...) :- writeln("Поток №%: ",N),
                    write(...),
                    writeln(" Конец потока №% ",N).
    run() :- Thread1 = thread::start({:- myWrite(1,"aaa")}),
            Thread2 = thread::start({:- myWrite(2,"bbb")}),
            _ = Thread1:wait(),
            _ = Thread2:wait(),
            _ = readchar().
end implement main
goal
    console::run(main::run).

```

Может показаться, что на экран будет выведено вначале сообщение:

Поток №1: aaa Конец потока №1

а потом в этой же строке:

Поток №2: bbb Конец потока №2.

На самом деле, запустив несколько раз этот фрагмент кода в консоли, можно увидеть строки вида:

Поток №1: Поток №2: aaabbb Конец потока №1 Конец потока №2

или в обратном порядке:

Поток №2: Поток №1: bbbaaa Конец потока №2 Конец потока №1

и другие виды смешанного вывода. Это не является ошибкой. Дело в том, что два потока выполняются одновременно и конкурируют за доступ к общему ресурсу. Поэтому вывод получается смешанным.

Для исключения такого явления каждому потоку надо захватывать монопольный доступ к ресурсу. Раньше для этого программист использовал *флаг* (mutex). Установленный флаг означает, что некий процесс использует ресурс, и другим процессам доступ к нему запрещен. Сброшенный флаг говорит о том, что ресурс свободен. При доступе какого-либо процесса к общему ресурсу проверяется состояние флага:

- если флаг сброшен, то он устанавливается, запрещая тем самым доступ к ресурсу другим процессам. Когда процесс освобождает ресурс, то флаг сбрасывается, разрешая другим процессам доступ к ресурсу;
- если же флаг установлен, то доступ к ресурсу запрещен, и заинтересованный процесс ожидает освобождения ресурса.

В современных языках программирования, в том числе и в Visual Prolog, никто не обязывает делать все эти проверки и манипуляции с флагом вручную. Вместо этого используют мониторные интерфейсы и классы.

Монитор — языковая конструкция, предназначенная для синхронизации двух или более потоков, которые используют разделяемые ресурсы. Компилятор прозрачно вставляет код блокировки и разблокировки разделяемых ресурсов, который соответствует описанной ранее логике.

Для объявления интерфейса или класса мониторным достаточно объявление этого интерфейса или класса предварить ключевым словом `monitor`. Запуск блокировки/разблокировки будет происходить автоматически при вызове внешних или интерфейсных предикатов и свойств.

ПРИМЕР 21.6. Рассмотрим ранее приведенный пример вывода на экран информации из двух потоков с помощью мониторного класса. Для этого создадим мониторный класс `lockWrite` с одним внешним предикатом `myWrite`. Вызов именно этого предиката и будет запускать блокировку/разблокировку вывода на экран.

```

monitor class lockWrite
predicates
    myWrite : (unsigned,...).
end class lockWrite

implement lockWrite
clauses
    myWrite(N,...) :- stdio::writef("Поток №%: ",N),
                    stdio::write(...),
                    stdio::writef(" Конец потока №% \n",N).
end implement lockWrite

```

Воспользуемся этим мониторным классом для вывода на экран сообщений aaa и bbb двумя параллельными потоками Thread1 и Thread2 соответственно:

```

run() :- Thread1 = thread::start({:- lockWrite::myWrite(1,"aaa")}),
        Thread2 = thread::start({:- lockWrite::myWrite(2,"bbb")}),
        _ = Thread1:wait(),
        _ = Thread2:wait(),
        _ = readchar().

```

Как можно догадаться, результат работы этого кода соответствует желаемому:

Поток №1: aaa Конец потока №1

Поток №2: bbb Конец потока №2

21.3. Защита

Защита (guard) — языковая конструкция, указывающая условие, которое проверяется перед выполнением определенного предиката. Если условие истинно, то предикат выполняется. Если условие ложно, то выполнение предиката задерживается. Эта задержка реализуется бесконечным циклом ожидания до тех пор, пока контекст выполнения предиката не изменится, и условие станет истинным.

Описанное условие является защитой выполняемого предиката от возникновения исключительной ситуации либо каких-либо нежелательных последствий. Например, при чтении файла проверяется наличие файла и правильность формата данных, при доступе к переменной по чтению — факт ее инициализации, при выборке элемента из списка проверяется, что список не пуст, при чтении факта БД проверяется его наличие и т. п.

Защита записывается как специальное предложение, стоящее перед остальными предложениями предиката. Собственно условие представляет собой именованный или анонимный предикат, который записывается после ключевого слова *guard*.

Когда функцию защиты предиката *ppp()* выполняет именованный предикат, например, *aaa()*, то защита выглядит так:

```

predicates
    aaa : () determ.          % объявление предиката-защитника

```

```
clauses
```

```
aaa() :- <проверяемое условие> % предикат-защитник
ppp guard aaa(). % постановка защиты
ppp() :- ... % предложения защищаемого предиката
```

Если же защиту выполняет анонимный предикат, то организация защиты записывается проще:

```
clauses
```

```
ppp guard {:-<проверяемое условие>}. % постановка защиты
ppp() :- ... % предложения защищаемого предиката
```

ПРИМЕР 21.7. Рассмотрим обобщенный класс `queue`, в котором используется защита предиката выборки элемента из очереди в случае, когда очередь пуста. Интерфейс класса является мониторным и содержит два объектных предиката: `enqueue` — постановка элемента в очередь и `dequeue` — выборка элемента из очереди. Реализация класса имеет объявление фактов `value_fact`, в которых и хранятся элементы очереди. Кроме этого реализация содержит определение предикатов постановки и выборки элемента из очереди. Обратите внимание, что строка:

```
dequeue guard { :- value_fact(_,!) }.
```

объявляет защиту предиката `dequeue` посредством проверки условия существования хотя бы одного факта в очереди. Если факт существует, то его значение возвращается функцией `dequeue`, а сам факт удаляется из очереди. Для того чтобы функция `dequeue`, согласно ее объявлению, была процедурой, добавлено последнее предложение, в котором ничего не возвращается из очереди и осуществляется вызов исключительной ситуации с ее описанием и локализацией. Локализацией является указание полного имени предиката, в котором произошло исключение. Для этого служит функция `predicate_fullname()`.

```
monitor interface queue{@Elem}
predicates
enqueue : (@Elem Value).
dequeue : () -> @Elem Value.
end interface queue
```

```
class queue{@Elem} : queue{@Elem}
end class queue
```

```
implement queue{@Elem}
facts
value_fact : (@Elem Value).
clauses
enqueue(V) :- assert(value_fact(V)).
```

```
dequeue guard { :- value_fact(_,!) }.
dequeue() = V :- retract(value_fact(V)),!.
```

```

dequeue() = _ :-  

    exception::raise_errorf(@"Попытка выборки элемента из  

    пустой очереди в предикате %", predicate_fullname()) .  

end implement queue

```

Использование этой очереди показано в следующей цели:

```

run():-Q=queue::new(),           % создание пустой очереди  

    Q:enqueue(15),                % добавление в очередь числа 15  

    Q:enqueue(16),                % добавление в очередь числа 16  

    X=Q:dequeue(), write(X),nl,   % выборка первого элемента 15  

    Y=Q:dequeue(), write(Y),nl,   % выборка второго элемента 16  

    Z=Q:dequeue(), write(Z),nl,   % выборка третьего элемента  

    _ = readchar().

```

После выборки двух элементов очереди программа перейдет в бесконечный цикл ожидания третьего элемента.

ПРИМЕР 21.8. Продемонстрируем пример того, как можно удовлетворить такое бесконечное ожидание из другого вычислительного потока Thread. Работа программы будет выглядеть следующим образом. После создания пустой очереди стартует поток Thread, который вызывает двухсекундную паузу, после чего добавляет в очередь число 17. Это число будет третьим в «хронологии» очереди, т. к. во время двухсекундного сна потока Thread основной поток программы добавит в очередь числа 15 и 16, извлечет их и выведет на экран, после чего попытается извлечь третий элемент, который к тому моменту еще отсутствует. Он появится в очереди две секунды спустя из потока Thread.

```

run():-Q=queue::new(),  

    Thread = thread::start({ :- programControl::sleep(2000),  

                             Q:enqueue(17)}),  

    Q:enqueue(15),  

    Q:enqueue(16),  

    X=Q:dequeue(), write(X),nl,      % выборка первого элемента 15  

    Y=Q:dequeue(), write(Y),nl,      % выборка второго элемента 16  

    Z=Q:dequeue(), write(Z),nl,      % выборка третьего элемента 17  

    _ = Thread:wait(),  

    _ = readchar().

```

ГЛАВА 22



Доступ к API-функциям Windows

Visual Prolog имеет пакет `windowsAPI`, содержащий объявления констант, доменов и функций большой части программных интерфейсов операционной системы Windows. К программным интерфейсам относится и графический интерфейс GUI-приложений и интерфейс с аппаратными ресурсами. Используя этот пакет, можно легко вызывать API-функции (Application Programming Interface, интерфейс прикладных программ) в случае, если вы не можете найти в библиотеке PFC (Prolog Foundation Classes) чего-либо вам необходимого.

Кроме того, Visual Prolog имеет средства для объявления тех функций, которые отсутствуют в пакете `windowsAPI`. Поэтому программист может по описанию API-функции, расположенному на ресурсе подразделения MSDN (Microsoft Developer Network), объявить и использовать вызывающий ее предикат в Visual Prolog. Этот ресурс находится на сайте <http://msdn.microsoft.com>. API-функции, используемые в этой главе для управления консолью, можно посмотреть по адресу:

[http://msdn.microsoft.com/ru-ru/library/windows/desktop/ms682073\(v=vs.85\).aspx](http://msdn.microsoft.com/ru-ru/library/windows/desktop/ms682073(v=vs.85).aspx).

22.1. Описание типов данных API-функций доменами Visual Prolog

Типы данных API-функций в MSDN приведены на языках семейства C, а также на Бейсике. Стандартные типы данных этих языков имеют аналоги в Visual Prolog. Например, булев тип данных `BOOL` можно заменять доменом `booleanInt` языка Visual Prolog. Тип `DWORD` — доменом `unsigned`, `TCHAR` — `char`, `TSTR` — `string`, указатель — `pointer`, дескрипторы `HWND` и другие — `handle` или `fileHandle` и т. д. в зависимости от контекста.

Структуры данных, с которыми работают API-функции, описываются составными доменами Visual Prolog с некоторыми особенностями, обусловленными использованием этих доменов для вызова API-функций.

Первая особенность — выравнивание аргументов домена на заданное количество байтов путем указания ключевого слова `align` (выровнять), за которым следует

число байтов, на которые следует выровнять адреса аргументов. Выравнивание имеет смысл, когда домен содержит более одного аргумента. Выравнивание должно осуществляться на 1, 2 или 4 байта.

Например, координаты точки можно описать в Visual Prolog доменом:

```
domains  
    point = p(integer X, integer Y).
```

Однако для связи с другими языками такое описание не годится. Этот домен надо переписать в виде:

```
domains  
    point = align 4 p(integer X, integer Y).
```

Здесь явно указано выравнивание аргументов на 4 байта.

Вторая особенность — передача API-функции составного домена в виде структуры со встроенными в нее значениями аргументов. Атрибут `[inline]`, расположенный после аргумента, указывает на то, что этот аргумент должен быть встроен.

Например, координаты четырехугольника определяются координатами его левого верхнего угла и правого нижнего угла. В программе Visual Prolog эти координаты легко описать доменом:

```
domains  
    rectangle = rect(point UpperLeft, point LowerRight).
```

Но в таком виде нельзя передавать координаты API-функции. Для их передачи необходимо использовать домен:

```
domains  
    rectangle = rect(point UpperLeft [inline],  
                      point LowerRight [inline]).
```

В структуру также можно встраивать строки фиксированного размера и поля ANSI-строк. Например, можно фиксировать размер строки, описывающей название устройства, величиной 32 символов:

```
domains  
    device = align 4 device(integer Id,  
                           string DeviceName [inline(32)]).
```

Размер строки из двухбайтовых символов домена `string` — 64 байта, а так как размер домена `integer` составляет 4 байта, то размер домена в целом будет 68 байтов. Если же для названия устройства использовать однобайтовые строки `string8`, то размер такого домена — всего 36 байтов:

```
domains  
    device = align 4 device(integer Id,  
                           string8 DeviceName [inline(32)]).
```

Третья особенность — описание домена, имеющего несколько альтернатив с помощью атрибута `[union]`. Этот атрибут служит для имитации структур `union` языков С-семейства:

```
u64var = align 4 u64(unsigned64 Value64);
        u64_struct(unsigned Low32, unsigned High32) [union].
```

22.2. Объявления предикатов для вызова API-функций

Для правильного объявления предикатов, вызывающих API-функции и вообще функции других языков программирования, необходимо использовать так называемое *соглашение о вызовах*, которое определяет порядок передачи аргументов и правила построения имени вызываемой функции из имени вызывающего предиката. Это связано с тем, что порядок передачи аргументов у разных языков программирования различен. Поэтому при объявлении предиката следует указать то соглашение, по правилам которого будут передаваться аргументы. Для этого служат ключевое слово *language* и идущее за ним название соглашения. Соглашение может быть одним из следующих: *thiscall*, *stdcall*, *apicall*, *c*, *prolog*, например:

```
predicates
predicateName : (unsigned, unsigned64) language c.
```

Если соглашение не указано, то по умолчанию принимается соглашение *prolog* языка Visual Prolog.

Соглашение *c* подразумевает соглашение языков C и C++. При этом имя вызываемой функции языка C/C++ получается путем добавления знака подчеркивания *_* к имени вызывающего предиката.

Соглашение *thiscall* следует соглашению стандарта C++ для вызова виртуальных функций. По этому соглашению имя вызываемой функции строится так же, как и по соглашению языка C. Но порядок передачи аргументов иногда может отличаться от соглашения *c*. Соглашение *thiscall* применяется только для объектных предикатов.

Соглашение *stdcall* также следует соглашению стандарта языка C при построении имени вызываемой функции, но правила передачи аргументов другие. Таблица 22.1 показывает правила реализации соглашения *stdcall* для передачи аргументов.

Таблица 22.1. Соглашение о вызовах *stdcall*

Вызов <i>stdcall</i>	Правила реализации
Порядок передачи аргументов	Справа налево
Соглашение о передаче аргументов	Передача по значению с помощью указания атрибута [<i>byVal</i>], за исключением случая, когда передается составной домен, и за исключением предикатов с переменным числом аргументов
Ответственность за освобождение стека	Вызываемая функция выталкивает из стека свои аргументы
Соглашение о построении имени вызываемой функции	Знак подчеркивания <i>_</i> добавляется к имени вызывающего предиката

Таблица 22.1 (окончание)

Вызов stdcall	Правила реализации
Соглашение о трансляции регистра символов	Трансляция регистра символов имени вызывающего предиката не выполняется

В стек предпочтительно передавать непосредственное значение аргумента, нежели использовать указатель `pointer`.

Соглашение `apicall` использует те же правила передачи аргументов и управления стеком, что и `stdcall`. Правила построения имени вызываемой API-функции из имени вызывающего предиката следующие:

- к имени предиката добавляется знак подчеркивания в виде префикса;
- первая буква имени предиката заменяется на заглавную;
- если аргументы API-функции и возвращаемое значение являются символами ANSI, то к имени предиката справа добавляется суффикс `A`, а если символами Unicode (широкими символами, `wide`), то — суффикс `W`. Если API-функция не обменивается символами с предикатом, то к имени предиката не добавляется ни суффикс `A`, ни суффикс `W`;
- к имени предиката добавляется суффикс `@`;
- число байтов, помещенных в стек при вызове API-функции, добавляется к имени предиката в виде суффикса.

Например, объявление предиката:

```
predicates
  predicateName : (unsigned, string) language apicall.
```

указывает, что будут задействованы Unicode-символы, иначе был бы использован тип `string8`. Размер типов `string` и `unsigned` по 4 байта, следовательно, в стек будет задавлено 8 байтов. Поэтому оригинальное имя вызываемой API-функции: `_PredicateNameW@8`.

Для добавления суффикса `A` или `W` к имени предиката надо использовать конструкцию `as decoratedA` или `as decoratedW` соответственно.

В табл. 22.2 приведено описание соглашений `c`, `apicall` и `stdcall`.

Таблица 22.2. Сравнение соглашений `c`, `apicall` и `stdcall`

Имя соглашения	Освобождение стека	Соглашение о трансляции регистра символов имени предиката	Соглашение о декорации имени предиката
<code>c</code>	Вызывающий предикат освобождает стек	Трансляции нет	Знак подчеркивания <code>_</code> добавляется к имени предиката в виде префикса

Таблица 22.2 (окончание)

Имя соглашения	Освобождение стека	Соглашение о трансляции регистра символов имени предиката	Соглашение о декорации имени предиката
thiscall	Вызывающий предикат освобождает стек, исключая аргумент This, который передается в регистре	Трансляции нет	Такое же, как и соглашение с
stdcall	Вызываемая API-функция освобождает стек	Трансляции нет	Знак подчеркивания _ добавляется к имени предиката в виде префикса
apicall	Вызываемая API-функция освобождает стек	Первая буква имени предиката заменяется заглавной	Знак подчеркивания _ добавляется к имени предиката в качестве префикса. Первая буква предиката заменяется заглавной. При необходимости имя предиката декорируется буквами A или W. К имени предиката в качестве суффикса добавляется знак @. К имени предиката в качестве суффикса добавляется число байтов, занимаемых аргументами функции

22.3. Использование пакета windowsAPI

ПРИМЕР 22.1. Так как мы используем в наших примерах консольные приложения, посмотрим, как заполнить каким-либо повторяющимся символом весь экран консоли или некоторую его часть. В классе `console` такой функции нет, однако она есть в классе `console_native` и носит имя `fillConsoleOutputCharacter`. Объявление этой функции расположено в файле `console_native.cl`.

```

predicates
fillConsoleOutputCharacter : (
    fileHandle ConsoleOutputHandle,           % дескриптор выходного потока
    char Character,                         % символ для заполнения
    charCount Length,                      % число повторов
    coord WriteCoord [byVal],              % координаты начальной позиции
    charCount NumberOfCharsWritten [out])   % число выведенных символов
    -> booleanInt Result                 % признак результата
language apicall as decoratedW.        % Unicode-символ

```

Если вывод успешен, то функция возвращает ненулевое значение (или `b_true`), иначе возвращает ноль (или `b_false`). Кроме этого функция возвращает число выведенных символов.

Для заполнения верхних десяти строк символом + достаточно получить дескриптор выходного потока с помощью функции `getStdHandle` класса `console_api` и вызвать функцию `fillConsoleOutputCharacter` с необходимыми параметрами:

```
implement main
    open core, console
clauses
    run() :-
```

% получение дескриптора выходного потока:

```
        OutputHandle = console_api::getStdHandle(
            console_native::stdOutput_Handle),
```

% заполнение 10-ти строк консоли знаком '+':

```
        B = console_native::fillConsoleOutputCharacter(
            OutputHandle,           % дескриптор выходного потока
            '+',                   % выводимый символ
            800,                  % 800 раз, 10 строк по 80 символов
            console_native::coord(0,0), % левый верхний угол
            _N),                  % число выведенных символов неважно
        if B=0 then             % если вывод вызвал исключительную ситуацию
            LastError = exception::getLastError(), % получаем ссылку
                                                % на ошибку
            exception::raise_nativeCallException( % запускаем обработчик
                "fillConsoleOutputCharacter",
                LastError, [])
        end if,
        _ = readline().
end implement main
goal
    console::run(main:::run).
```

ПРИМЕР 22.2. Класс `console_native` имеет функцию `messageBox` для вывода сообщений. Однако в этом классе она объявлена как предикат, а не как функция, и поэтому не возвращает код нажатой кнопки. Исправим эту несправедливость, для чего в нашем консольном классе объявим функцию `messageBox` и передадим ей дескриптор окна консольного приложения `Handle`. Кроме этого при вызове окна сообщения мы можем указать в параметре `Type` флаг, определяющий состав кнопок окна, и флаг, определяющий значок. В нашем примере флаг `mb_yesnocancel` определяет три кнопки: `yes`, `no` и `cancel`. А флаг `mb_iconstop` задает значок, располагаемый в окне сообщения.

```
implement main
    open core, console
class predicates
    messageBox : (handle Hwnd, string Text, string Caption, unsigned Type)
                    -> integer language apicall.
clauses
    run() :-
```

```
        Handle = console_native::getConsoleWindow(),
        messageBox(Handle, "Hello, world!", "Message", mb_yesnocancel | mb_iconstop).
```

```
Type = gui_native::mb_yesnocancel + gui_native::mb_iconstop,
A = messageBox(Handle,"Text","Caption",Type),
(A = 6,write("yes"); A = 7,write("no"); write("cancel")),!,
_=readchar().
end implement main
goal
  console::run(main::run).
```

Название нажатой кнопки легко определяется по возвращаемому функцией коду (табл. 22.3). Возможные для размещения в окне значки показаны в табл. 22.4.

Таблица 22.3. Возвращаемые значения кнопок окна messageBox

Имя кнопки	Возвращаемое значение
OK	1
CANCEL	2
ABORT	3
RETRY	4
IGNORE	5
YES	6
NO	7
TRYAGAIN	10
CONTINUE	11

Таблица 22.4. Возможные значки окна messageBox

Значок	Флаг
	MB_ICONHAND MB_ICONSTOP MB_ICONERROR
	MB_ICONQUESTION
	MB_ICONEXCLAMATION MB_ICONWARNING
	MB_ICONASTERISK MB_ICONINFORMATION

22.4. Использование API-функций из библиотеки Windows

В классе `console` не описана функция `scrollConsoleScreenBuffer`, которая перемещает прямоугольный фрагмент консольного окна в заданные координаты, вырезая его из исходной позиции. При этом координаты курсора не изменяются.

ПРИМЕР 22.3. В MSDN на языке C++ приведено объявление этой функции:

```
BOOL WINAPI ScrollConsoleScreenBuffer(
  _In_      HANDLE hConsoleOutput,
  _In_      const SMALL_RECT *lpScrollRectangle,
  _In_opt_   const SMALL_RECT *lpClipRectangle,
  _In_      COORD dwDestinationOrigin,
  _In_      const CHAR_INFO *lpFill
);
```

Из описания понятно, что все аргументы входные, HANDLE — дескриптор выходного потока, а SMALL_RECT — это структура данных, описывающая координаты прямоугольной области. Ее объявление есть в классе console_native, и нам дополнитель- но объявлять ее не надо. COORD — координаты левого верхнего угла той прямоугольной области, куда размещается вырезаемый прямоугольник. В структуре CHAR_INFO указывается символ, которым будет заполнена исходная позиция прямоугольника и его цветовые атрибуты. Структуры COORD и CHAR_INFO также уже объявлены в классе console_native, поэтому нам не надо объявлять ни одной структуры.

Третий параметр является областью, в которой разрешен перенос прямоугольника из исходной позиции в конечную. Так как наше консольное окно имеет 80 столбцов и 25 строк, нумерация которых начинается с нуля, то пусть третий параметр покрывает все консольное окно. Вырезаемый прямоугольник будет размещен на 30-й позиции 12-й строки.

ЗАМЕТКА

На сленге программистов объявление функции scrollConsoleScreenBuffer называется «оберткой».

```
implement main
    open core, console
class predicates
    scrollConsoleScreenBuffer: (handle,           % дескриптор выходного потока
                                console_native::small_rect,      % исходные координаты
                                console_native::small_rect,      % разрешенная область
                                console_native::coord [byVal],   % конечные координаты
                                console_native::char_info)       % символ и его
                                                    % цветовые атрибуты
    -> booleanInt language apicall as decoratedW.
clauses
run() :-
    OutputHandle = console_api:::getStdHandle(
                    console_native:::stdOutput_handle), % читаем
                                                    % дескриптор
    B=console_native:::fillConsoleOutputCharacter(
        OutputHandle, % дескриптор выходного потока
        '+', 800,      % заполняем знаком плюс 10 строк консольного окна
        console_native:::coord(0,0), % начинаем с левого верхнего угла
        _N),          % игнорируем количество реальных выводов знака +
    if B=0 then     % если вывод вызвал исключительную ситуацию
        LastErr = exception:::getLastErrorCode(), % получаем ссылку
                                                    % на ошибку
        exception:::raise_nativeCallException( % запускаем обработчик
            "fillConsoleOutputCharacter", LastErr, [])
    end if,
```

```

B1=scrollConsoleScreenBuffer(OutputHandle,
    console_native::region(1,1,3,3),           % вырезаемая область
    console_native::region(0,0,79,24),          % все окно
    console_native::coord(30,12),               % координаты размещения
    console_native::char_info('o',8)),          % символ для заполнения
if B1=0 then           % если вывод вызвал исключительную ситуацию
    LastErr1 = exception::getLastError(), % получаем ссылку
                           % на ошибку
    exception::raise_nativeCallException( % запускаем обработчик
        "scrollConsoleScreenBuffer", LastErr1, [])
end if,
_ = readchar().
end implement main
goal
    console::run(main::run).

```

В результате выполнения программы мы получим примерно такую картинку (рис. 22.1).

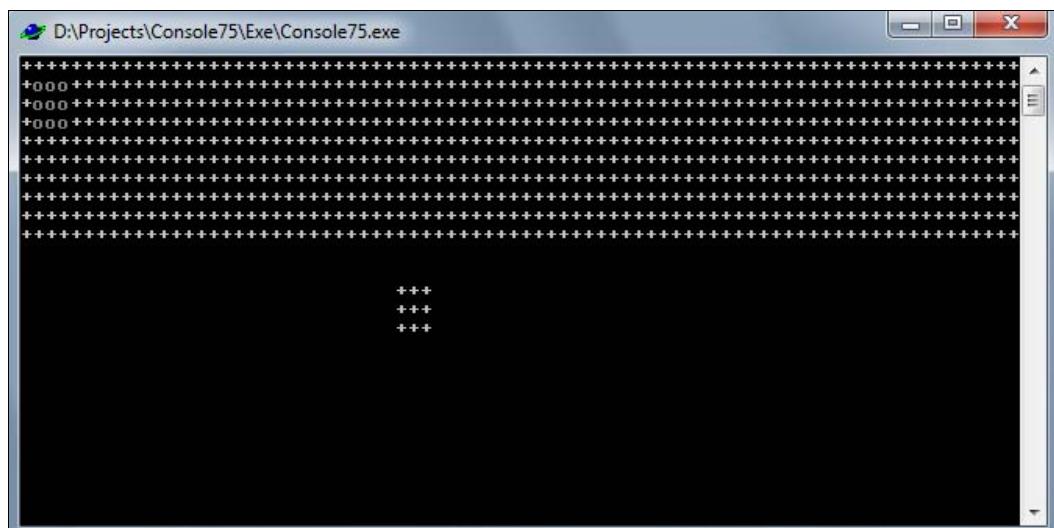


Рис. 22.1. Перемещение прямоугольной области в консольном окне

ГЛАВА 23



Разработка и использование DLL

Коммерческая версия Visual Prolog позволяет создавать динамически связываемые (подключаемые) библиотеки (DLL, dynamic link library). Эти библиотеки могут использоваться как проектами Visual Prolog, так и программами, написанными на других языках программирования. В проектах Visual Prolog могут также задействоваться библиотеки, написанные на других языках. В библиотеку, как правило, помещают те предикаты, которые либо решают отдельную задачу, либо объединяются единой прикладной областью. Подключение библиотек позволяет уменьшить размер используемой памяти, поскольку DLL загружаются только при необходимости и после использования из памяти выгружаются. Кроме того, DLL позволяют уменьшить трудозатраты программирования, т. к. единожды разработанная библиотека может использоваться в различных проектах без изменения.

Существует два способа вызова процедур из DLL: посредством раннего связывания (early binding или static binding) и посредством позднего связывания (late binding или dynamic binding). Под словом «связывание» имеется в виду проверка правильности имени библиотечной процедуры, порядка ее аргументов и их типов в месте вызова. При раннем связывании такая проверка осуществляется во время компиляции. При позднем связывании — во время выполнения программы.

Преимущество использования раннего связывания заключается в повышении надежности разрабатываемого проекта в целом, т. к. все вызовы библиотечных предикатов и функций из основной программы проверяются во время компиляции. Поэтому по возможности следует ориентироваться именно на такой тип связывания.

В этой главе рассматривается разработка и использование DLL с помощью класса mainDll пакета application по технологии позднего связывания. Для разработки DLL по технологии раннего и позднего связывания можно также воспользоваться примерами-заготовками, которые устанавливаются при инсталляции Visual Prolog.

23.1. Создание DLL-проекта

Для построения проекта динамической библиотеки надо открыть диалоговое окно создания нового проекта (рис. 23.1) и указать тип проекта: **DLL**. Кроме этого нужно ввести имя библиотеки — например: `testDll`, а также можно убрать флагок создания файла-манифеста.

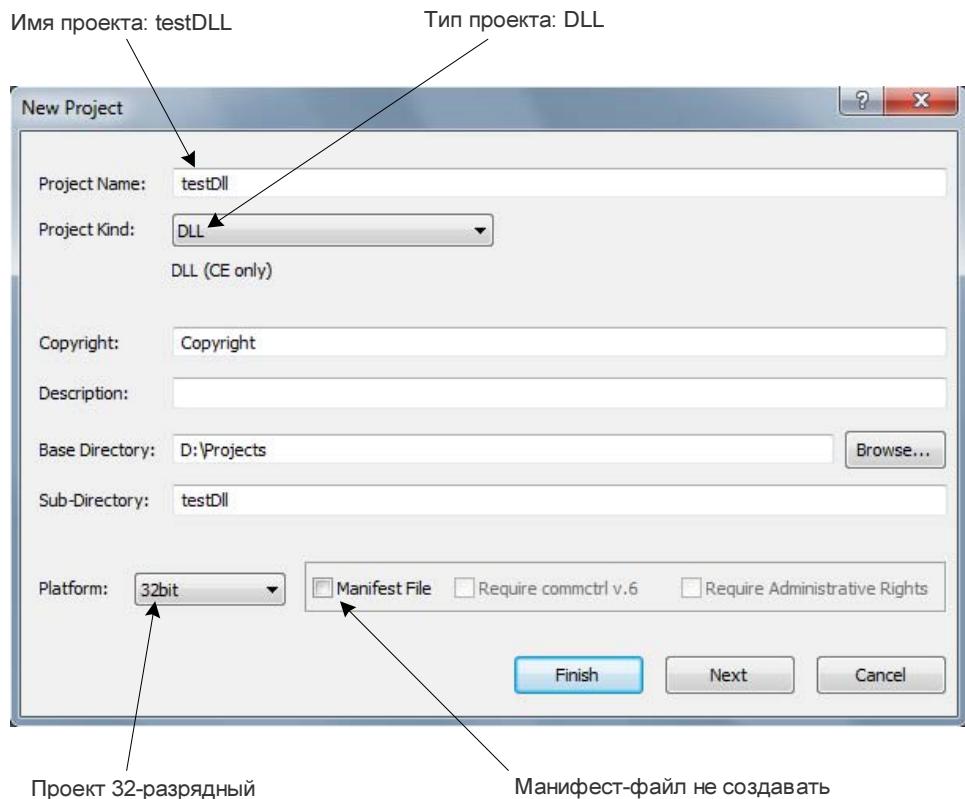


Рис. 23.1. Диалоговое окно создания DLL-проекта

Пути и собственно имя DLL-проекта рекомендуется указывать на латинице. После создания и компиляции проекта мы увидим, что он, кроме всего прочего, содержит модули `export` и `main`:

- `export.cl;`
- `export.pro;`
- `main.cl;`
- `main.pack;`
- `main.ph;`
- `main.pro;`
- `main.version.`

Модуль `export` предназначен для описания экспортруемых предикатов. Экспортруемые предикаты — это те библиотечные предикаты, которые можно вызывать в некотором исполняющем файле. Модуль `main` содержит конструктор и предикаты, определяющие реакцию на события, происходящие с DLL-модулем.

ПРИМЕР 23.1. Разработаем проект, использующий библиотеку DLL, содержащую, к примеру, три экспортруемых предиката, которые производят простые математические операции. Проект должен содержать, во-первых, создание собственно библиотеки DLL, и, во-вторых, разработку консольного приложения, которое будет использовать эту библиотеку.

23.2. Описание экспортруемых предикатов и функций DLL-проекта

В объявлении экспортруемого предиката, кроме собственно объявления предиката, надо указать имя соглашения о передаче значений его аргументов и имя, под которым этот предикат будет вызываться извне. Следует помнить, что экспортруемый предикат должен иметь процедурный режим детерминизма. Как правило, соглашением о порядке передачи аргументов является `stdcall`. Описание этого и других соглашений приведено в главе 22. Экспортруемый предикат в DLL-проекте может быть описан под одним именем (внутреннее имя), а вызываться под другим именем (внешнее имя). Эти имена могут совпадать.

Опишем в нашем DLL-проекте функцию `power`, вычисляющую степени заданного числа. Эта функция будет иметь режим `multi` для того, чтобы на откатах возвращать увеличивающиеся на единицу степени числа. Также опишем предикат `add` для сложения двух целых чисел и функцию `combination` для вычисления количества сочетания из `N` по `M`. Для этого в файле `export.cl` объявим:

```
class export
open core
predicates
    power : (integer) -> integer multi    language stdcall as "power".
    add : (integer,integer,integer [out])   language stdcall as "add".
    combination : (unsigned,unsigned) -> unsigned language stdcall
                                as "cmb".
end class export
```

Здесь функция `power` видна извне как `power`, предикат `add` видим извне под своим же именем, т. к. после ключевого слова `as` указано имя `add`. Для функции `combination` определено внешнее имя `cmb`.

В файле `export.pro` определим наши процедуры:

```
#export export
implement export
    open core
class predicates
    factorial : (unsigned) -> unsigned.
```

```

clauses
power(X) = X.
power(X) = X * power(X).

add(A, B, A+B).

combination(M, N) =
    math::roundToUnsigned(factorial(N) / factorial(M) / factorial(N-M)).

factorial(0) = 1 :- !.
factorial(N) = N * factorial(N-1).
end implement export

```

Обратите внимание, что в разделе `class predicates` объявлена невидимая извне функция `factorial`, которая служит для вычисления числа сочетаний по формуле:

$$\frac{n!}{m!(n-m)!}.$$

Так как значение функции `factorial` объявлено `unsigned`, то для преобразования результата деления, имеющего тип `real`, в тип `unsigned` используется функция преобразования `roundToUnsigned` из класса `math`.

Директива `#export` имя класса указывается в реализации того класса, предикаты которого экспортируются.

После компиляции проекта в папке `Exe` будет создана библиотека `testDll.dll`. Этую библиотеку лучше всего скопировать в ту папку, где расположен исполняемый файл проекта, использующий эту библиотеку. Можно, впрочем, и не копировать, но тогда придется указать путь к библиотеке `testDll.dll` при ее вызове из того проекта, где она используется.

23.3. Использование DLL-проекта

Использование DLL-модуля начинается с его загрузки, и после совершения необходимых вызовов должно завершиться выгрузкой:

```

D1l = useD1l::load("testD1l.dll"),
<здесь располагаются вызовы библиотечных процедур и функций>
D1l:unload(),

```

Для вызова процедуры, содержащейся в DLL-проекте, имеются два способа. Первый способ заключается в том, что для вызова процедуры используется ее внешнее имя. Второй способ вместо имени процедуры использует ее порядковый номер. Процедура, объявленная в файле `export.cl` последней, имеет порядковый номер 1, предпоследняя процедура имеет порядковый номер 2 и т. д.

Чтобы вызвать процедуру, содержащуюся в DLL-проекте, надо:

1. Знать объявление этой процедуры и описать ее в вызывающей программе в разделе доменов в виде предикатного домена.

2. Получить ссылку на процедуру по ее внешнему имени или по ее порядковому номеру с помощью предикатов `getPredicateHandle(Имя)` или `getPredicateHandle_ordinal(Номер)` соответственно.
3. Преобразовать ссылку в предикат с помощью функции `uncheckedConvert`.
4. Вызвать процедуру, используя результат преобразования в качестве имени процедуры.

Далее приводится консольное приложение, использующее процедуры нашей библиотеки `testDll.dll`. Вначале вызывается функция `power`, которая возвращает первую степень числа 2, а потом на откатах: вторую, третью и четвертую степень числа 2. Счетчиком цикла служит изменяемая переменная `N`, инициированная значением 0, а условием выхода является предикат `N:value>=4`.

Предикат `add` и функция `cmb` вызываются двумя способами: по имени и по номеру. Номер предиката `add` равен 2, а номер функции `cmb` равен 1. Стоит заметить, что при добавлении или удалении предикатов из библиотеки номера предикатов изменяются, поэтому надежней использовать вызов по имени, нежели вызов по номеру.

```

implement main
    open core, console
domains
    степень = (unsigned) -> unsigned multi language stdcall.
    сумма = (integer,integer,integer [out]) language stdcall.
    числоКомб = (unsigned,unsigned) -> unsigned language stdcall.
clauses
    run() :-
        Dll = useDll::load("testDll.dll"),      % загрузка библиотеки

        Power = uncheckedConvert(степень,Dll:getPredicateHandle("power")),
        (N = varM_integer::new(0),
        % вызов функции Power по имени:
        write("Вызов по имени:  Power(2) = ",Power(2)),nl,
        N:add(1),N:value>=4,!;
        succeed),

        S = uncheckedConvert(сумма, Dll:getPredicateHandle("add")),
        % вызов предиката add по имени
        S(2,3,Sum),
        write("Вызов по имени:  2+3 = ",Sum),nl,
        S1 = uncheckedConvert(сумма,Dll:getPredicateHandle_ordinal(2)),
        % вызов предиката add по его номеру 2
        S1(2,3,Sum1),
        write("Вызов по номеру:  2+3 = ",Sum1),nl,

        Число = uncheckedConvert(числокомб,Dll:getPredicateHandle("cmb")),
        % вызов функции cmb по имени:
        write("Вызов по имени:  С(2,4) = ",Число(2,4)),nl,

```

```
Числол = uncheckedConvert(числоКомб,
                           Dll:getPredicateHandle_ordinal(1)),
% вызов функции смв по ее номеру 1:
write("Вызов по номеру: C(2,4) = ",Числол(2,4)),nl,
Dll:unload(),    % выгрузка библиотеки
_ = readchar().
end implement main
goal
  console::run(main::run).
```

Результат работы программы:

```
Вызов по имени: Power(2) = 2
Вызов по имени: Power(2) = 4
Вызов по имени: Power(2) = 8
Вызов по имени: Power(2) = 16
Вызов по имени: 2+3 = 5
Вызов по номеру: 2+3 = 5
Вызов по имени: C(2,4) = 6
Вызов по номеру: C(2,4) = 6
```

ГЛАВА 24



Отладка приложений

24.1. Отладчик Visual Prolog

Отладчик Visual Prolog встроен в интегрированную среду разработки приложений. Он работает с 32/64-битными GUI-приложениями, консольными приложениями и DLL-проектами. Отладчик позволяет:

- выполнять программу по шагам;
- просматривать исходный код и его дизассемблированный образ;
- устанавливать точки останова;
- просматривать значения переменных, факты, дамп памяти, содержимое регистров в отдельных окнах;
- просматривать значения переменных наведением на них курсора;
- просматривать стек вызовов предикатов.

В *приложениях 6, 7 и 8* можно найти описание всех инструментов отладчика: пунктов меню, панели инструментов и клавиш быстрого доступа.

Рассмотрим использование отладчика для пошагового выполнения и просмотра значений переменных простейшего предиката вычисления длины списка:

```
implement main
    open core, console
class predicates
    len : (E*,unsigned,unsigned) procedure (i,i,o).
clauses
    len([_|L], I, X) :-  
        I1 = I+1,  
        len(L, I1, X).  
    len([], X, X).  
run():-len([8,9,12],0,X),  
        write(X),  
        _ = readChar().  
end implement main  
goal  
    console::run(main:::run).
```

Чтобы просмотреть работу предиката `len` по шагам, надо установить контрольную точку на первое предложение этого предиката (рис. 24.1). Щелчок по красному кружку на панели инструментов устанавливает/снимает контрольную точку останова в строке, на которой находится курсор. Это означает, что в ходе выполнения программы под отладчиком вычисления будут приостановлены до выполнения программистом следующего шага. При этом в соответствующих окнах отладчика можно просмотреть значения переменных, факты и т. д.

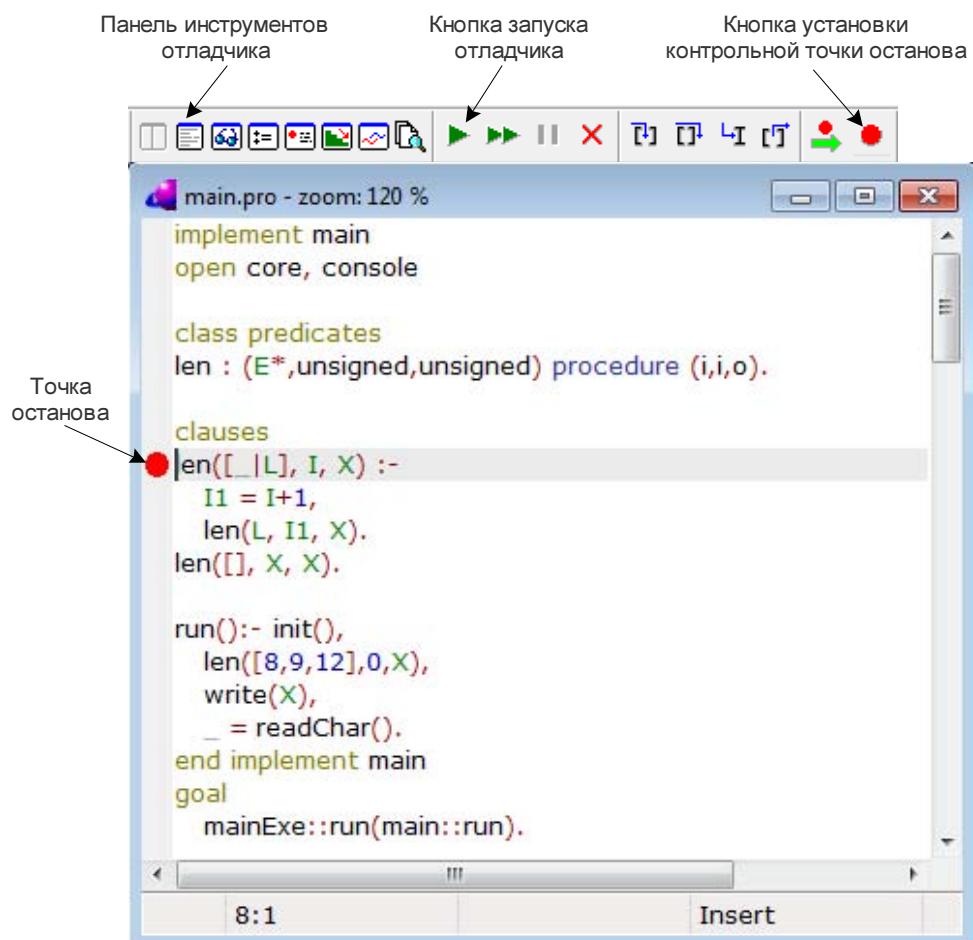


Рис. 24.1. Средства отладчика

Иногда в одной строке программист размещает несколько вызовов предикатов. Здесь следует иметь в виду, что остановка будет совершена только при выполнении того предиката, который в этой строке является самым первым. Если необходимо просмотреть значения переменных всех предикатов, то вызовы этих предикатов должны быть размещены каждый на отдельной строке. Именно так и оформлен предикат `len` на рис. 24.1.

После установки контрольной точки на первом предложении предиката `len` мы можем запустить программу под отладчиком с помощью команды меню **Debug | Run**, или клавиши быстрого вызова **<F5>**, или с помощью соответствующего инструмента на панели (зеленый треугольник).

На рис. 24.2 показан интерфейс отладчика с запущенной программой. Синяя стрелка указывает на текущий шаг выполнения программы — тот предикат, который

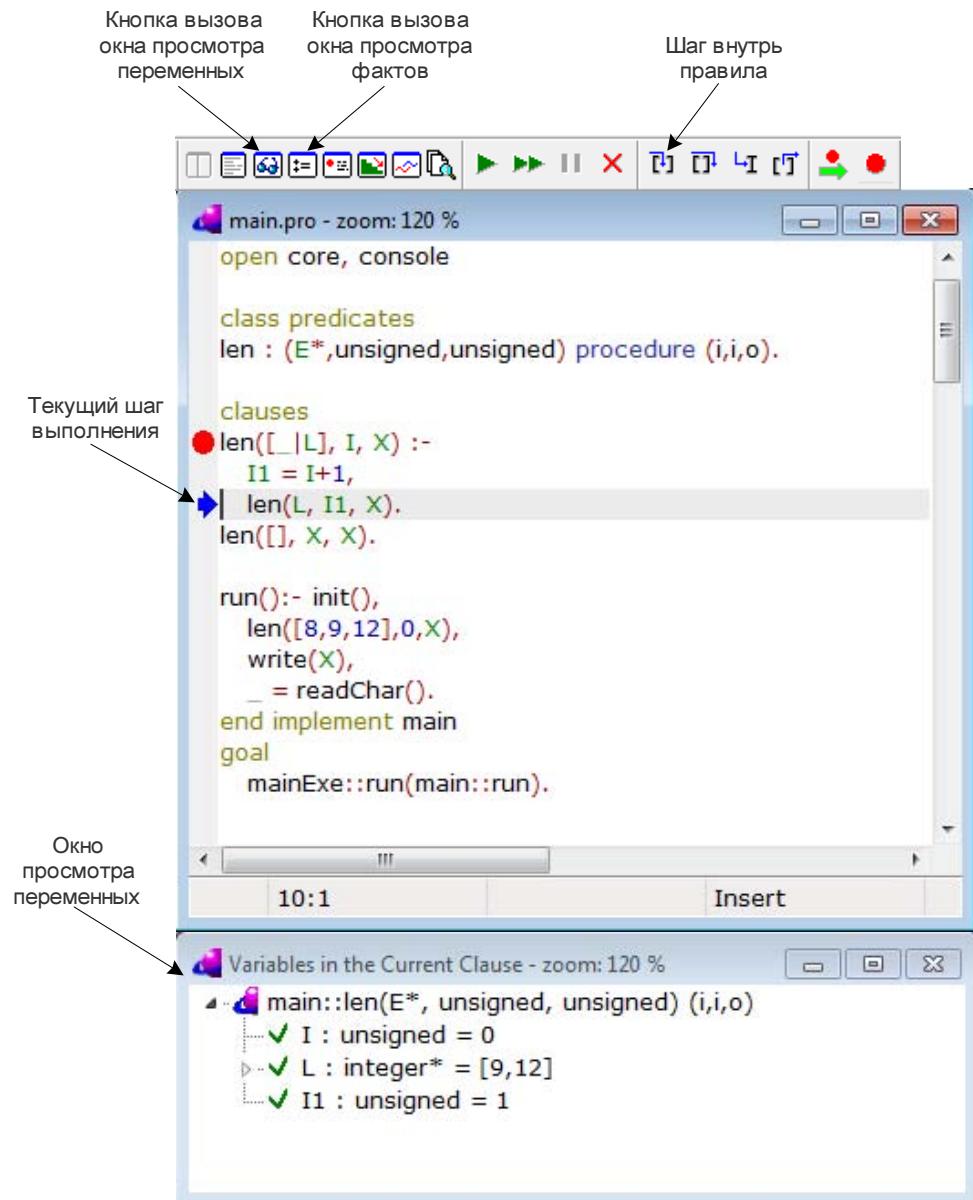


Рис. 24.2. Программа под отладчиком

будет вызван при следующем шаге отладчика. Для просмотра переменных надо вызвать соответствующее окно. В нем мы видим, что от списка отделен первый элемент, т. к. переменная `L` имеет значение `[9,12]`. Переменная `I` равна нулю. Окно просмотра фактов вызывать не будем, т. к. в нашей программе факты не используются.

Сделаем два шага внутрь трассируемого правила. В окне просмотра переменных появилась переменная `I1` со значением 1, т. к. выполнилась операция `I1 = I+1`. Следующий шаг приведет нас ко второму витку цикла, в котором переменная `L` равна `[12]`.

Продолжая так действовать дальше, мы выполним последний, третий, виток цикла и на выходе из цикла перейдем в точку его вызова — в тело предиката `run`. Здесь стоит выполнять предикаты за один шаг, т. к. пошаговое выполнение предиката `write` переместит нас в исходный код класса `console`.

Таким образом, на каждом витке цикла мы получим следующую информацию:

```
1-й виток: L = [9,12], I1 = 1
2-й виток: L = [12],   I1 = 2
3-й виток: L = [],     I1 = 3
```

Здесь мы наблюдаем рост счетчика `I1` по мере уменьшения списка на один элемент на каждом витке цикла. Когда список становится пустым, счетчик содержит правильную длину списка.

Обратите внимание, что в программе для нового значения счетчика служит отдельная переменная `I1`. Это связано с тем, что отладчик выводит только те переменные, которые явно представлены в предикате. Если бы мы использовали функциональную запись вида:

```
len([_|L],I,X) :-  
    len(L,I+1,X).  
len([],X,X).
```

то новые значения счетчика `I+1` мы бы не увидели вовсе, т. к. вместо явной операции `I1=I+1` использовано выражение `I+1`.

24.2. Просмотр значений переменных средствами языка

Во многих случаях для поиска ошибки можно обойтись без запуска отладчика — достаточно вывести значения переменных либо в консоль, если мы работаем с консолью, либо в диалоговое окно, если мы работаем с GUI-приложением.

В нашем примере для просмотра значений переменных `L` и `I1` достаточно выводить их на экран и при необходимости просмотра по шагам использовать функцию ожидания ввода с клавиатуры:

```

len([_|L],I,X) :-  

    I1=I+1, write(L," ",I1), _=readchar(),  

    len(L,I1,X).  

len([],X,X) :- write(X),nl.

```

Автор в большинстве случаев использует именно такой способ просмотра. При этом на экране мы получаем наглядную картину процессов, происходящих в цикле:

```

[9,12] 1  

[12] 2  

[] 3  

3

```

В случае GUI-приложений можно использовать диалоговые окна из коллекции диалогов vpiCommonDialogs:

```

len([_|L],I,X) :-  

    I1=I+1, vpiCommonDialogs::note(toString(tuple(L,I1))),  

    len(L,I1,X).  

len([],X,X) :- vpiCommonDialogs::note(toString(X)).

```

Просмотр значений переменных средствами языка более гибок, т. к. можно указать желаемые условия, при выполнении которых нам нужно увидеть значения переменных. Например, если надо просмотреть значение переменных L и I1 в случае, когда длина списка L меньше 2, то достаточно указать это условие в программе:

```

len([_|L],I,X) :-  

    I1=I+1,  

    (list::length(L)<2, write(L," ",I1), !; succeed),  

    len(L,I1,X).  

len([],X,X) :- write(X),nl.

```

В результате вывод будет осуществляться не на каждом витке цикла, а только на последних витках.

Такой прием особо выгодно использовать в случае, когда надо просмотреть значения переменных в середине или в конце цикла, — например, начиная с тысячного витка цикла или по достижении какого-либо условия. В отладчике это делать грустно, т. к. утомительно «шагать» это расстояние вручную.

Недостатком такого способа просмотра переменных является необходимость вносить дополнительный код, который после отладки надо не забыть удалить.

24.3. Контроль стека и кучи

Когда в программе имеется неоптимизированная рекурсия, то стоит проверить, насколько быстро истощается свободная оперативная память. Для этого надо просмотреть скорость роста используемого стека и кучи, т. е. определить, сколько байтов захватывается при каждом рекурсивном вызове. Это можно сделать с помощью функции получения размера используемого стека `memory::getUsedStack()` и функ-

ции получения размера используемой кучи `memory::getUsedHeap()`. Рост используемого стека хорошо виден на примере вычисления факториала, построенного на базе неоптимизированной рекурсии:

```
implement main
    open core,console,memory
class predicates
    f : (unsigned64) -> unsigned64.
clauses
    f(0) = 1 :- !.
    f(N) = N*f(N-1) :- write(getUsedStack()),nl.
    run():-write(f(4)),
          _ = readchar().
end implement main
goal
    console::run(main::run).
```

На каждом витке рекурсии наша программа захватывает 84 байта. Последнее число 24 является результатом подсчета значения $4!$.

```
1328
1412
1496
1580
24
```

Для вычисления факториала рост стека не страшен, т. к. для вычисления самого большого факториала, который только может уместиться в разрядную сетку процессора, достаточно двадцати витков рекурсии и, следовательно, менее 2 Кбайт, которые после завершения рекурсии будут возвращены системе. Но для тех неоптимизированных рекурсий, которые требуют большого количества рекурсивных вызовов, расход памяти может привести к полному истощению ее свободного остатка.

часть IV



Представление и обработка данных в Visual Prolog

Глава 25. Графы

Глава 26. Деревья

Глава 27. Массивы

Глава 28. Символьные преобразования

Глава 29. Интерпретатор программ

Глава 30. Практические рекомендации

ГЛАВА 25



Графы

25.1. Представление ориентированных графов

Visual Prolog не имеет PFC-классов для представления и обработки графов, так как существует много разновидностей графов и нет никаких сложностей в их представлении. Любые графы в Прологе можно представлять в виде внутренней БД или в виде списков. В простейшем случае дуга с инцидентными ей вершинами представляется фактом БД или элементом списка. Пусть ориентированный граф (орграф) содержит именованные вершины и взвешенные дуги (рис. 25.1).

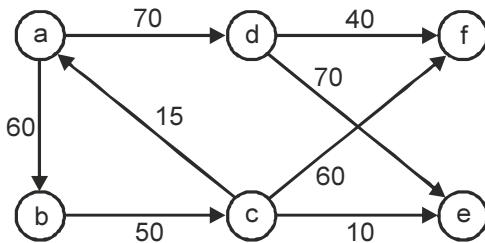


Рис. 25.1. Ориентированный граф

Такой орграф можно описать в виде фактов:

```
facts - орграф
дуга : (char Начало, char Конец, unsigned Вес).
```

Тогда внутренняя БД будет выглядеть так:

```
clauses
дуга('a', 'b', 60).    дуга('a', 'd', 70).
дуга('b', 'c', 50).    дуга('c', 'e', 10).
дуга('c', 'f', 60).    дуга('d', 'e', 70).
дуга('d', 'f', 40).    дуга('c', 'a', 15).
```

В этом описании каждая дуга описывается одним фактом. Например, факт дуга('a', 'b', 60) задает дугу от вершины a до вершины b весом 60.

Этот же граф можно описать в виде списка дуг:

```
[дуга('a','b',60), дуга('a','d',70),
дуга('b','c',50), дуга('c','e',10),
дуга('c','f',60), дуга('d','e',70),
дуга('d','f',40), дуга('e','g',20)]
```

Орграф можно описать также в виде «кустов». Каждый такой куст содержит вершину, имеющую исходящие из нее дуги, и список смежных вершин, достижимых по дугам, исходящим из исходной вершины:

```
domains
пара = π(char Вершина, unsigned Вес).
facts – орграф
куст : (char НачВершина, пара* КонечВершины).
```

Для орграфа, изображенного на рис. 25.1, внутренняя база кустов будет выглядеть так:

```
clauses
куст('a',[π('b',60),π('d',70)]).
куст('b',[π('c',50)]).
куст('d',[π('e',70),π('f',40)]).
куст('c',[π('e',10),π('f',60),π('a',15)]).
```

Такую базу кустов легко построить программно следующим образом: собрать с помощью коллектора списков для каждой вершины графа список смежных вершин. Например, собрать в список L вершины, смежные вершине a , можно так:

```
L = [π(V,D) || дуга('a',V,D)]
```

Описание орграфа в виде кустов хорошо подходит для реализации поиска «сначала вширь» при построении на графах путей, цепей, контуров и т. п.

Упомянутые описания графов эквивалентны, но обладают различными свойствами при решении задач на графах. В случае, когда орграф задан фактами БД, легко осуществлять поисковые задачи на графах, не предполагающие модификацию состава его вершин или веса дуг. При этом поиск смежной вершины и/или инцидентной дуги осуществляет механизм поиска с откатом. Например, для поиска всех вершин, смежных с вершиной c , достаточно вызвать факт `дуга('c',X,W)`. Так как этот факт недетерминированный, то при откатах он вернет все три варианта решений:

```
X = 'e', W = 10;
X = 'f', W = 60;
X = 'a', W = 15.
```

Когда же смысл задачи заключается в модификации графа, то лучше его представлять в виде списка дуг, который передавать в рекурсивный вызов предиката модификации графа. В этом случае поиск смежной вершины осуществляется просмотром вершин в списке, как правило, с помощью предикатов класса `list`.

Ориентированные графы, содержащие петли и/или кратные дуги, описываются аналогично. Например, орграф, представленный на рис. 25.2, описывается в виде дуг:

clauses

```
дуга('a', 'b', 60).    дуга('a', 'd', 70).
дуга('b', 'a', 10).   дуга('b', 'c', 20).
дуга('b', 'c', 50).   дуга('c', 'a', 15).
дуга('c', 'c', 20).   дуга('d', 'd', 90).
```

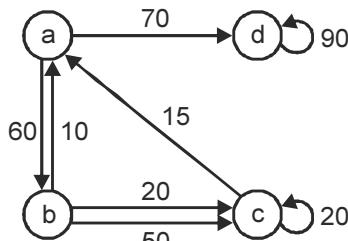


Рис. 25.2. Ориентированный граф с кратными дугами и петлями

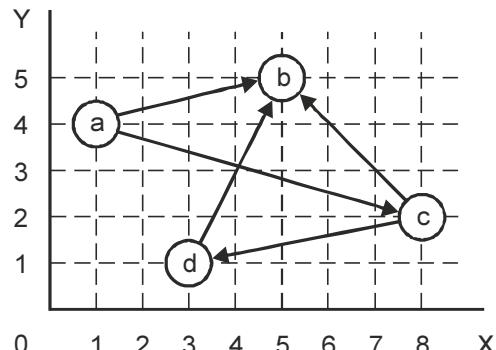


Рис. 25.3. Ориентированный граф на декартовой плоскости

Ориентированные графы могут задаваться на декартовой плоскости координатами вершин и связями между ними. Координаты вершин задаются в базе данных в виде отношения вершина (Имя, X, Y), а связи — в виде отношения дуга (Имя1, Имя2):

```
facts - орграф
вершина : (char Вершина, real X, real Y).
дуга : (char Вершина, char Вершина).
```

На рис. 25.3 изображен ориентированный граф на декартовой плоскости. Этот граф может быть описан фактами:

clauses

```
вершина('a', 1, 4).
вершина('b', 5, 5).
вершина('c', 8, 2).
вершина('d', 3, 1).
дуга('a', 'b').
дуга('a', 'd').
дуга('c', 'b').
дуга('c', 'd').
дуга('d', 'b').
```

Расстояние между вершинами a и b вычисляется по теореме Пифагора:

```
вершина('a', X1, Y1),
вершина('b', X2, Y2),
Dist = math::sqrt((X1-X2)^2 + (Y1-Y2)^2).
```

25.2. Представление неориентированных графов

Неориентированные графы представляются так же, как и ориентированные, — с помощью фактов, описывающих ребра. Но, учитывая, что ребра в неориентированном графе ненаправленные, каждое из них можно представлять либо парой фактов, описывающих встречные дуги между двумя смежными вершинами, либо одним фактом, описывающим дугу, и правилом для замыкания дуги в обратную сторону. Например, неориентированный граф, изображенный на рис. 25.4 и имеющий четыре ребра, может быть описан восемью фактами: по два факта дуга/3 на одно ребро:

```

facts - граф
дуга : (char, char, unsigned).
clauses
дуга('a', 'b', 60).    дуга('b', 'a', 60).
дуга('a', 'c', 15).    дуга('c', 'a', 15).
дуга('a', 'd', 70).    дуга('d', 'a', 70).
дуга('b', 'c', 50).    дуга('c', 'b', 50).

```

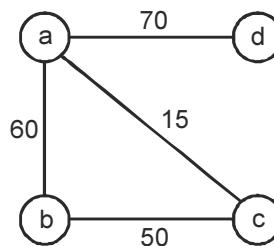


Рис. 25.4. Неориентированный граф

Этот граф можно описать также и четырьмя фактами дуга/3 и предикатом ребро/3, замыкающим дуги в обратную сторону:

```

facts - граф
дуга : (char, char, unsigned).
predicates
ребро : (char, char, unsigned) nondeterm (i,o,o).
clauses
дуга('a', 'b', 60).    дуга('a', 'c', 15).
дуга('a', 'd', 70).    дуга('b', 'c', 50).
ребро(X,Y,W) :- дуга(X,Y,W); дуга(Y,X,W).

```

Здесь предикат ребро/3 объявлен с шаблоном потоков (i,o,o). Это объясняется тем, что в задачах на графах для заданной вершины x (первый аргумент), как правило, ищется смежная ей вершина y (второй аргумент), к которой ведет ребро весом w (третий аргумент). То есть, второй и третий аргументы выходные. Суть предиката замыкания проста — когда необходимо найти вершину y , смежную к заданной вершине x , то предикат ребро(X, Y, W) сначала ищет в базе дугу дуга(X, Y, W),

исходящую из x , а если таковой не находит, то ищет дугу дуга (Y, X, W) , входящую в x . Понятия «исходящая» и «входящая» для ребра неотличимы, т. к. ребра неориентированные.

Второй способ более предпочтительный по сравнению с первым, т. к. ведет к уменьшению базы фактов в два раза.

Неориентированные графы, содержащие петли и/или кратные дуги, а также неориентированные графы на декартовой плоскости, описываются аналогично тому, как показано для ориентированных графов.

25.3. Поиск «сначала вглубь»

ПРИМЕР 25.1. Пусть дан граф, изображенный на рис. 25.4. Найдем все пути между вершинами — например: b и d , рассматривая вес ребер как расстояния между вершинами.

Для этого разработаем недетерминированный предикат поиска одного пути между двумя вершинами и с помощью отката получим все пути, пройдя по всему дереву решений. Пусть начальная и конечная вершины носят имена Старт и Стоп соответственно. Предикат на каждом витке рекурсии для последней пройденной вершины A должен найти одну вершину B , смежную к ней, и расстояние до нее $Длина$. Также предикат должен проверить, что найденная вершина B не принадлежит пройденному маршруту. Кроме того, к уже пройденному расстоянию надо прибавить дистанцию до найденной вершины. При рекурсивном вызове вместо последней вершины A надо указать следующую вершину B . Рекурсия завершится тогда, когда найденная вершина совпадет с вершиной Стоп. Так как рекурсия накапливает пройденный путь в стеке в обратном порядке, то при достижении вершины Стоп надо реверсировать стек, чтобы получить путь в правильном порядке. Аргументами предиката путь/6 при его вызове являются:

путь (Старт,	% Начальная вершина [in]
Стоп,	% Конечная вершина [in]
[Старт],	% Список для хранения пройденных вершин [in]
Путь,	% Список вершин от Старт до Стоп [out]
0,	% Аккумулятор пройденного расстояния [in]
Длина)	% Суммарное расстояние от Старт до Стоп [out]

Для получения всех путей воспользуемся циклом `foreach`, и на каждом витке этого цикла будем выводить на экран очередное решение Путь в виде списка вершин и суммарное расстояние Длина:

```
implement main
    open core, console, list
class facts - граф
    дуга: (char, char, unsigned).
class predicates
    ребро: (char, char, unsigned) nondeterm (i,o,o).
    путь: (char, char, char*, char*, unsigned, unsigned)
        nondeterm (i,i,i,o,i,o).
```

```
clauses
```

```
дуга('a', 'b', 60).    дуга('a', 'c', 15).
дуга('a', 'd', 70).    дуга('b', 'c', 50).
```

```
ребро(A, B, Длина) :- дуга(A, B, Длина); дуга(B, A, Длина).
```

```
путь(B, B, Путь, reverse(Путь), Длина, Длина).
```

```
путь(A, B, Стек, Путь, Аккум, Длина) :-
```

```
ребро(A, A1, Расстояние),           % нашли смежную вершину
not(A1 in Стек),                   % вершина не встречалась
Аккум1 = Аккум + Расстояние,      % пройденное расстояние
путь(A1, B, [A1 | Стек], Путь, Аккум1, Длина).
```

```
run() :- Старт='b',
        Стоп='d',
        foreach путь(Старт, Стоп, [Старт], Путь, 0, Длина) do
            writef("Путь = % \nДлина = % \n\n", Путь, Длина)
        end foreach,
        _ = readLine().
end implement main
goal
    console::run(main::run).
```

В результате программа найдет два пути:

```
Путь = ['b', 'c', 'a', 'd']
Длина = 135
```

```
Путь = ['b', 'a', 'd']
Длина = 130
```

Для сравнения можно привести эту же программу, но работающую не с базой фактов, а со списком фактов. Список фактов строится в теле предиката `run` посредством коллектора списков `[дуга(A, B, D) || дуга(A, B, D)]`:

```
implement main
    open core, console, list
class facts - граф
    дуга:(char, char, unsigned).
class predicates
    путь:(char, char, граф*, char*, char*, unsigned, unsigned)
        nondeterm (i,i,i,i,o,i,o).
    взять:(char,char,unsigned,граф*,граф*) nondeterm (i,o,o,i,o).
clauses
    дуга('a', 'b', 60).    дуга('a', 'c', 15).
    дуга('a', 'd', 70).    дуга('b', 'c', 50).

    путь(B, B, _, Путь, reverse(Путь), Длина, Длина).
    путь(A, B, Граф, Стек, Путь, Аккум, Длина) :-
```

```

взять (A,A1,Расстояние,Граф,Граф1),      % нашли смежную вершину
not(A1 in Стек),                          % вершина не встречалась
Аккум1 = Аккум + Расстояние,            % пройденное расстояние
путь (A1,Б,Граф1, [A1|Стек],Путь,Аккум1,Длина).

```

```

взять (A,Б,Расстояние, [дуга (A,Б,Расстояние) | Граф],Граф) .
взять (A,Б,Расстояние, [дуга (Б,A,Расстояние) | Граф],Граф) .
взять (A,Б,Расстояние, [B|Граф], [B|Граф1]) :-           % возврат
    взять (A,Б,Расстояние,Граф,Граф1) .

```

```

run() :-Старт='b',
Стоп='d',
Список = [дуга (A,B,D) || дуга (A,B,D)],
foreach путь (Старт,Стоп,Список, [Старт],Путь,0,Длина) do
    writef("Путь = % \nДлина = % \n\n",Путь,Длина)
end foreach,
_ = readLine().
end implement main
goal
    console::run(main::run) .

```

Здесь введен новый предикат `взять/5`, который вырезает из списка дуг подходящую дугу. При этом список уменьшается на один факт, благодаря чему по мере построения пути на графе пространство поиска линейно сокращается. Предикат `взять/5` недетерминированный, и поэтому на откате назад перебирает все допустимые варианты.

Результат работы программы:

Путь = ['b', 'a', 'd']

Длина = 130

Путь = ['b', 'c', 'a', 'd']

Длина = 135

25.4. Поиск «сначала вширь»

ПРИМЕР 25.2. Пусть дан граф, изображенный на рис. 25.4. Найдем все пути между вершинами — например: `b` и `d`, рассматривая вес ребер как расстояния между вершинами.

Для этого разработаем процедуру поиска списка всех путей между двумя вершинами. Пусть начальная и конечная вершина носят имена `Старт` и `Стоп` соответственно. Каждый путь представим кортежем `tuple(Путь, Длина)`, где `Путь` является списком вершин, а `Длина` — суммарным расстоянием. Порядок работы предиката пути/4 таков.

1. Когда в списке путей найдется путь, который достиг вершины `Стоп`, то этот путь мы реверсируем и сохраняем в стеке готовых решений. Это делает первое пра-

- вило предиката пути/4. Мы реверсируем путь потому, что список вершин пути строим в обратном порядке.
2. Если вершина A не является конечной для пути [A|Путь], то найдем список путей [A1, A|Путь], продолжающих путь из вершины A ко всем смежным вершинам A1. Этую работу выполняет второе правило предиката пути/4.
 3. Когда список путей опустеет, то стек готовых путей будет списком всех найденных путей, — третье правило предиката пути/4.

4. Для случая, когда какой-либо путь заканчивается вершиной, из которой дальше хода нет, и эта вершина не является стоповой вершиной, то мы исключим этот путь из списка путей с помощью четвертого правила. Это может произойти, когда вершина оказалось тупиковой/висячей в графе, или когда все ее смежные вершины принадлежат этому пути.

Аргументами предиката пути/4 при его вызове являются:

```
путь (Стоп, % Конечная вершина [in]
      [tuple([Старт], 0)], % Начальный список путей [in]
      [], % Стек готовых путей [in]
      Пути) % Список решений [out]
```

Вывод решений в программе выполнен с помощью предиката forall, которому передается список найденных путей и анонимный предикат, печатающий каждый путь вместе с его суммарной длиной:

```
implement main
    open core, console, list
domains
    путь = tuple{char*, unsigned}.
class facts - граф
    дуга: (char, char, unsigned).
class predicates
    ребро: (char, char, unsigned) nondeterm (i,o,o).
    пути: (char, путь*, путь*, путь* [out]).
clauses
    дуга('a', 'b', 60).    дуга('a', 'c', 15).
    дуга('a', 'd', 70).    дуга('b', 'c', 50).

    ребро(A, B, Длина) :- дуга(A, B, Длина); дуга(B, A, Длина).

    пути(Б, [tuple([Б|Путь], Длина) | Пути], Аккум, Пути0) :- !,
        пути(Б, Пути, [tuple(reverse([Б|Путь]), Длина) | Аккум], Пути0).
    пути(Б, [tuple([A|Путь], Длина) | Пути], Аккум, Пути0) :-
        Пути1 = [tuple([A1, A|Путь], Длина+Длина1)] ||
            ребро(A, A1, Длина1), not(A1 in Путь)],
        !, пути(Б, append(Пути, Пути1), Аккум, Пути0).
    пути(_, [], Пути, Пути) :- !.
    пути(Б, [_ | Пути], Аккум, Пути0) :- пути(Б, Пути, Аккум, Пути0).
```

```
run() :-Старт='b',
        Стоп='d',
        пути(Стоп, [tuple([Старт], 0)], [], Пути),
        forall(Пути, { (tuple(Путь, Длина)) :-
                      writeln("Путь = ~nДлина = ~n", Путь, Длина) } ),
        _ = readLine().
end implement main
goal
    console::run(main::run).
```

Результат работы программы:

Путь = ['b', 'c', 'a', 'd']

Длина = 135

Путь = ['b', 'a', 'd']

Длина = 130

ГЛАВА 26



Деревья

Visual Prolog имеет пакет `collection`, содержащий ряд классов для обработки множеств, словарей, массивов, деревьев, отображений, очередей и т. п. Коллекции в пакете разделяются на два типа. Один тип коллекций содержит неизменяемые данные и называется `persistent` (постоянный, неизменяемый). При добавлении или удалении данных из коллекции этого типа мы получим новый экземпляр коллекции. Наряду с ним в программе останется и прежний экземпляр без изменений. Характерным примером такой коллекции являются списки. Когда мы разделяем список на голову и хвост, то в программе остается исходный список, и кроме этого, мы получим голову и хвост. Когда же мы собираем новый список из головы и существующего списка, то получаем новый список, однако в программе остаются и голова, и прежний список. Таким образом, `persistent`-коллекции модифицируются конструктивно.

Другой тип коллекций содержит изменяемые данные и называется `modifiable` (изменяемый). Добавление или удаление данных из коллекции такого типа приводит к изменению этой коллекции, ибо коллекция существует в единственном экземпляре и модифицируется деструктивно. Характерным примером такой коллекции является база фактов. Добавление или удаление факта изменяет коллекцию фактов.

Таким образом, при получении/чтении данных из коллекций между их двумя типами фундаментальных отличий нет. Поэтому получение данных из коллекций организовано через общий интерфейс. Отличия есть только при обновлении коллекций, для чего сделано два интерфейса: один для `persistent`-коллекций, другой для `modifiable`-коллекций.

Visual Prolog имеет класс красно-черных деревьев для создания и обработки деревьев. Однако перед использованием этого класса рассмотрим способы объявления деревьев и реализацию основных операций над ними, которыми можно пользоваться вне рамок красно-черных деревьев.

26.1. Представление деревьев в Visual Prolog

Деревья в Visual Prolog описываются в разделе доменов в виде рекурсивных термов. Терм описывает вершину дерева и имеет столько аргументов, сколько ветвей

выходит из вершины, плюс один или более аргументов для описания самой вершины. Лист дерева описывается нерекурсивным термом и играет роль признака останова рекурсии.

Опишем, к примеру, бинарное дерево в виде домена:

domains

```
дерево = дер(дерево Левое, integer, дерево Правое); лист.
```

Здесь терм дер(дерево, integer, дерево) описывает вершину, хранящую некоторые данные, — в данном случае число integer. Также этот терм содержит левую и правую ветви, которые являются поддеревьями:

```
дер(ЛевоеПоддерево, Вершина, ПравоеПоддерево)
```

Терм лист() описывает лист дерева. Он подставляется вместо левой или правой ветви в терме дер(дерево, integer, дерево), когда этот терм не имеет ветвей или имеет всего одну ветвь.

Представим дерево, изображенное на рис. 26.1. Вершина, содержащая число 6, описывается термом:

```
дер(лист, 6, лист)
```

Вершина с числом 9 описывается термом:

```
дер(дер(лист, 6, лист), 9, лист)
```

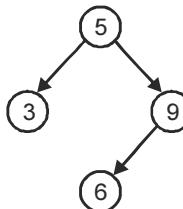


Рис. 26.1. Бинарное дерево

Здесь левое поддерево содержит вершину дер(лист, 6, лист), а правое поддерево является листом.

В целом, дерево выражается термом:

```
дер(дер(лист, 3, лист), 5, дер(дер(лист, 6, лист), 9, лист))
```

Тернарные деревья и деревья большей арности описываются аналогично. Можно описывать деревья, у которых вершины имеют различное число ветвей.

Следующий домен дерево разрешает иметь вершины с одной, двумя и тремя ветвями:

domains

```
дерево = дер1(integer, дерево);
дерево = дер2(integer, дерево, дерево);
дерево = дер3(integer, дерево, дерево, дерево);
лист().
```

Когда арность дерева велика или неизвестна, и у каждой вершины может быть разное число потомков, то такие деревья можно описывать доменом, использующим список поддеревьев:

```
domains
дерево = дер(integer, дерево*); лист.
```

Здесь первый аргумент терма `дер(integer, дерево*)` содержит номер вершины `integer`, второй аргумент `дерево*` является списком поддеревьев.

Используя термы этого домена, опишем дерево, изображенное на рис. 26.1. Вершина, содержащая число 6, описывается термом:

```
дер(6, []).
```

Вершина с числом 9 описывается термом:

```
дер(9, [дер(6, [])]).
```

В целом, дерево выражается термом:

```
дер(5, [дер(3, []), дер(9, [дер(6, [])])]).
```

Терм `лист()` не использовался вовсе. Он нужен только для единственного случая — описать пустое дерево.

26.2. Операции над деревьями

В прикладных задачах вершины деревьев хранят не просто число. Они хранят числовой ключ и какие-либо данные — например, строки. Тогда добавление, поиск и удаление этой строки будет производиться по ключу. Здесь мы рассмотрим самый простой вариант дерева — бинарное упорядоченное дерево, вершины которого хранят только одно число.

26.2.1. Добавление вершины

Операция добавления вершины замещает лист дерева новой вершиной. Например, если в бинарном дереве встречается терм `лист()`, то его можно заменить термом `дер(лист, З, лист)`. Тем самым мы добавим в дерево число 3.

Перед добавлением вершины надо ответить себе на вопросы: куда помещать новую вершину? По каким правилам искать место для новой вершины? Ответов на эти вопросы множество. Все зависит от свойств дерева, которыми мы хотим его наделить. Более того, когда дерево сбалансированное, то в случае, если добавление новой вершины разбалансировало дерево, работает механизм балансировки.

Рассмотрим простой случай бинарного упорядоченного дерева, в которое добавляются числа по следующим правилам. Если добавляемое число меньше числа корневой вершины, то спускаемся в левое поддерево, если больше, то спускаемся в правое поддерево, если добавляемое число равно числу корневой вершины, то ничего не добавляем, т. к. это число уже присутствует в дереве. Эти правила применяются по мере продвижения вниз по дереву. Если на пути встречается лист, то он заменя-

ется вершиной, содержащей добавляемое число. Такое дерево будет содержать числа в единственном экземпляре.

Пусть дерево описано доменом:

domains

```
дерево = дер(дерево, integer, дерево); лист.
```

Объявим предикат добавления вершины:

class predicates

```
добавить: (integer, дерево, дерево [out]). % добавляемое число
                                         % дерево, в которое добавляем число
                                         % новое дерево
```

Реализация предиката по указанным ранее правилам будет выглядеть так:

clauses

```
добавить (Х, лист, дер(лист, Х, лист)) :- !. % заменяем лист
                                                 % новой вершиной
добавить (Х, дер(Лев, Х, Прав), дер(Лев, Х, Прав)) :- !. % число уже есть
добавить (Х, дер(Лев, N, Прав), дер(Дер, N, Прав)) :- 
    X < N, !, добавить (Х, Лев, Дер). % спускаемся в левое поддерево
добавить (Х, дер(Лев, N, Прав), дер(Лев, N, Дер)) :- 
    добавить (Х, Прав, Дер). % спускаемся в правое поддерево
```

26.2.2. Поиск вершины

Операция поиска вершины проверяет, есть ли заданное число в дереве. В самом простом случае предикат поиска будет успешен, когда искомое число найдено в дереве, и неуспешен в противном случае. Правила работы предиката такие же, как и при добавлении вершины, исключая одно правило — собственно замена листа новой вершиной не производится:

class predicates

```
тест: (integer, дерево) determ.
```

clauses

```
тест (Х, дер (_, Х, _)) :- !. % число найдено
тест (Х, дер (Лев, N, _)) :- X < N, !, тест (Х, Лев). % ищем в левом поддереве
тест (Х, дер (_, _, Прав)) :- тест (Х, Прав). % ищем в правом поддереве
```

26.2.3. Удаление вершины

Операция удаления вершины замещает удаляемую вершину другой вершиной дерева или листом. Например, если в бинарном дереве (см. рис. 26.1) надо удалить концевую вершину дер(лист, 3, лист), то на ее место следует поместить терм лист(). Этим самым мы удалим из дерева число 3.

Как можно видеть, удалить вершину, у которой нет потомков, легко. Также легко удалить вершину, у которой всего один потомок, — например, терм дер(лист, 3, Прав) замещается его потомком — правым поддеревом Прав. А что де-

лать, когда удаляемая вершина имеет более одного потомка? Какой из потомков займет место родителя? Ответов на этот вопрос множество. Все зависит от свойств дерева, которые мы должны поддерживать после удаления вершины. Более того, если дерево сбалансированное, то при удалении вершины, наверное, следует запустить механизм балансировки в случае, если удаление вершины разбалансирует дерево. В этой главе мы не будем заниматься балансировкой, а рассмотрим один простой способ удаления вершины в упорядоченном дереве.

Пусть удаление вершины бинарного дерева, которая имеет двух потомков, выполняется по правилу, согласно которому место удаляемой вершины занимает вершина из левого поддерева, содержащая максимальное значение. Так как наше дерево упорядочено, то вершина, замещающая удаляемую, будет последним потомком правой ветви левого поддерева.

Здесь следует заметить, что выбор именно левого поддерева условен. Можно выбрать и правое поддерево, в котором следует искать минимальный элемент, расположенный в левой ветви.

На рис. 26.2 изображено дерево, в котором удаляется вершина с числом 8. Ее место занимает вершина с числом 6. Место же вершины 6 занимает ее левое поддерево, т. е. вершина 4. Никаких других перестроений при этом не производится.

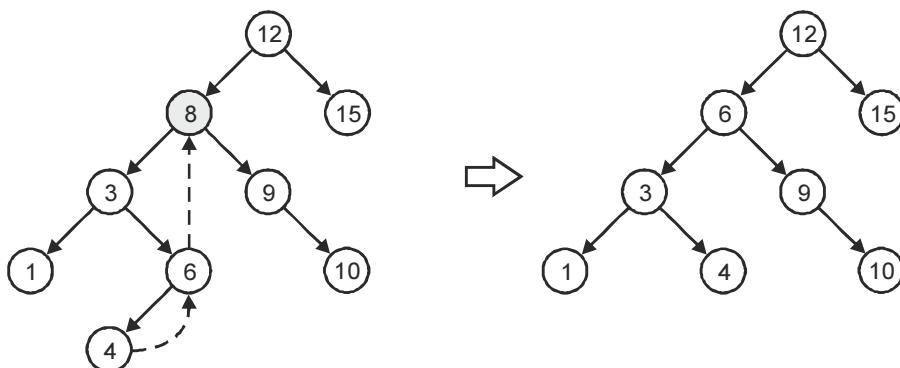


Рис. 26.2. Замещение удаляемой вершины

Описанные правила удаления позволяют сохранить главное свойство нашего дерева — упорядоченность вершин, при которой левый потомок всегда меньше родителя, а правый всегда больше.

Реализация нашего варианта удаления из дерева вершины содержит два предиката. Предикат `удалить/3` занимается поиском вершины по заданному числу `X`. Если такого числа в дереве нет, то предикат завершается неудачей. Если же вершина с таким числом найдена, и она имеет два поддерева `дер(Лев1, X, Прав)`, то вызывается предикат `поднять(Лев, Лев1, Y)` с целью «поднять» из левого поддерева `Лев` потомка с самым большим числом `Y`. При этом возвращается усеченное поддерево `Лев1` и собственно потомок `Y`. Этот потомок образует новый корень поддерева `дер(Лев1, Y, Прав)` вместо удаленной вершины `X`:

```

class predicates
    удалить:(integer, дерево, дерево) determ (i,i,o).
    поднять:(дерево, дерево, integer) determ (i,o,o).

clauses
    удалить (Х, дер (Лев,Х,лист) ,Лев) :-!.          % замещаем левым поддеревом
    удалить (Х, дер (лист,Х,Прав) ,Прав) :-!.          % замещаем правым поддеревом
    удалить (Х, дер (Лев,Х,Прав) ,дер (Лев1, Y,Прав) ) :-
        !, поднять (Лев, Лев1, Y).                  % поднимаем потомка Y
    удалить (Х, дер (Лев,N,Прав) ,дер (Лев1,N,Прав) ) :-
        X<N, !, удалить (Х, Лев, Лев1).            % ищем в левом поддереве
    удалить (Х, дер (Лев,N,Прав) ,дер (Лев,N,Прав1)) :-
        удалить (Х,Прав,Прав1).                      % ищем в правом поддереве

    поднять (дер (Лев,Х,лист) ,Лев,Х) :-!.          % нашли правого потомка
    поднять (дер (Лев,Х,Прав) ,дер (Лев,Х,Прав1),Y) :-
        поднять (Прав,Прав1,Y).                      % ищем в правом поддереве

```

Пример 26.1. Реализация всех трех операций и их использование приведены в следующей программе:

```

implement main
    open core, console
domains
    дерево = дер (дерево, integer, дерево); лист.
class predicates
    добавить:(integer, дерево, дерево [out]). 
    тест:(integer, дерево) determ.
    удалить:(integer, дерево, дерево) determ (i,i,o).
    поднять:(дерево, дерево, integer) determ (i,o,o).

clauses
    добавить (Х,лист,дер (лист,Х,лист) ) :-! .
    добавить (Х, дер (Лев,Х,Прав) ,дер (Лев,Х,Прав) ) :-! .
    добавить (Х, дер (Лев,N,Прав) ,дер (Лев1,N,Прав) ) :-
        X<N, !, добавить (Х, Лев, Лев1).
    добавить (Х, дер (Лев,N,Прав) ,дер (Лев,N,Прав1)) :-
        добавить (Х,Прав,Прав1).

    тест (Х, дер (_,_,_)) :-! .
    тест (Х, дер (Лев,N,_)) :-X<N, !, тест (Х,Лев) .
    тест (Х, дер (_,_,_Прав)) :-тест (Х,Прав) .

    удалить (Х, дер (Лев,Х,лист) ,Лев) :-! .
    удалить (Х, дер (лист,Х,Прав) ,Прав) :-! .
    удалить (Х, дер (Лев,Х,Прав) ,дер (Лев1, Y,Прав) ) :-
        !, поднять (Лев, Лев1, Y).
    удалить (Х, дер (Лев,N,Прав) ,дер (Лев1,N,Прав) ) :-
        X<N, !, удалить (Х, Лев, Лев1).
    удалить (Х, дер (Лев,N,Прав) ,дер (Лев,N,Прав1)) :-
        удалить (Х,Прав,Прав1).

```

```

поднять (дер (Лев,Х,лист) ,Лев,Х) :-! .
поднять (дер (Лев,Х,Прав) ,дер (Лев,Х,Прав1) ,Y) :-
    поднять (Прав,Прав1,Y) .

run () :-Д=дер (лист,8,лист) ,          % инициализация дерева
        добавить (5,Д,Д1) ,                % добавление вершин
        добавить (6,Д1,Д2) ,
        добавить (3,Д2,Д3) ,
        write (Д3) ,nl,
        (тест (6,Д3),write ("есть") ,! ;   % поиск вершины
         write ("нет") ),nl,
        (удалить (5,Д3,Д4),write (Д4) ,! ;  % удаление вершины
         write ("нет") ),nl,
        _ = readLine () .
end implement main
goal
    console::run (main::run) .

```

В результате на экран будет выведено созданное из четырех вершин дерево д3, потом результат поиска вершины 6, и в третьей строчке — дерево, у которого удалили вершину 5:

```

дер (дер (дер (лист () ,3,лист ()) ,5,дер (лист () ,6,лист ()) ),8,лист ())
есть
дер (дер (лист () ,3,дер (лист () ,6,лист ()) ),8,лист ())

```

26.3. Красно-черные деревья

Красно-черные деревья реализованы в классе `redBlackTree`. Они относятся к самобалансирующимся деревьям и предназначены для быстрого выполнения операций поиска, добавления и удаления. Вычислительная сложность этих операций в красно-черных деревьях в худшем случае приближается к $O(\log_2 N)$, что меньше вычислительной сложности подобных операций, проводимых с фактами БД, которая равна $O(N)$. Здесь N — это количество элементов в дереве или количество фактов БД. Уменьшение времени выполнения запросов является достоинством деревьев. Платой за это является усечение выразительности запросов по сравнению с запросами к БД, т. к. запросы к дереву осуществляются только по ключевым данным, а не по всем данным, которые хранятся в вершинах дерева. Рассмотрим эту тему подробнее.

Информация, хранимая в деревьях, представляет собой пару `(ключ, данные)`. Ключ также может являться полем данных, которые выделены в отдельную категорию, рассматриваемую в качестве признака, по которому будет вестись поиск данных в дереве. Если вам надо использовать другой признак для поиска данных, следует создать новое дерево и наполнить его такими парами `(ключ, данные)`, в которых значения ключей описывают нужный вам поисковый признак.

Например, в одном случае это может быть пара:

(адрес, персональные_данные)

в другом

(образование, персональные_данные)

Класс красно-черных деревьев может создавать и обрабатывать деревья двух видов. К первому относятся деревья, позволяющие по одному и тому же ключу записывать и хранить различные данные. Запись по существующему ключу новых данных добавляет их к хранимым данным. Такие деревья будем называть *деревьями с не-уникальными (nonunique) данными*. Ко второму виду относятся деревья, для которых пара *(ключ, данные)* является уникальной парой. При попытке записи новых данных по заданному ключу прежние данные будут удалены. Эти деревья будем называть *деревьями с уникальными (unique) данными*.

Когда в дереве используются нечисловые ключи (строки, символы, составные термы), то при поиске данных в дереве искомый ключ сравнивается с ключами вершин дерева лексикографически. Это означает, что попарно сравниваются коды символов, составляющих ключи: первый — с первым, второй — со вторым и т. д. Если такой способ сравнения нас по каким-либо причинам не устраивает, то при создании нового дерева можно определить собственную функцию сравнения ключей, отличную от лексикографического сравнения. Например, строки можно сравнивать не посимвольно, а по их длинам, или при сравнении строк брать в расчет только первый символ.

Класс красно-черных деревьев содержит функции для создания новых деревьев:

- Дер = empty() — создание nonunique-дерева;
- Дер = emptyCustom(Comparator) — создание nonunique-дерева с указанием функции Comparator для сравнения ключей;
- Дер = emptyUnique() — создание unique-дерева;
- Дер = emptyCustomUnique(Comparator) — создание unique-дерева с указанием функции Comparator для сравнения ключей;
- Дер = emptySameKind(Tree) — создание пустого дерева со свойствами дерева Tree.

Созданные деревья будут пустыми. Для их наполнения, модификации и поиска данных существует ряд предикатов — вот описание основных:

- Дер1 = insert(Дер, Ключ, Данные) — создание нового дерева Дер1 путем добавления в дерево Дер вершины Данные с ключом Ключ;
- Дер1 = delete(Дер, Ключ, Данные) — создание нового дерева Дер1 путем удаления вершины Данные с ключом Ключ из дерева Дер. Если данных с таким ключом нет, то функция возвращает прежнее дерево без изменений;
- tuple(Ключ, Данные) = getAll_nd(Дер) — недетерминированная функция возвращает кортеж tuple(Ключ, Данные);

- Данные = lookUp_nd(Дер, Ключ) — недетерминированная функция возвращает данные по заданному ключу Ключ. Этую функцию выгодно использовать в коллекторе списков для получения списка всех данных по известному ключу;
- tuple(Ключ, Данные) = downFrom_nd(Дер, Ключ0) — недетерминированная функция возвращает в порядке уменьшения значения ключа кортеж tuple(Ключ, Данные), удовлетворяющий условию Ключ <= Ключ0. Этую функцию выгодно использовать в коллекторе списков для получения списка всех кортежей tuple(Ключ, Данные), чей ключ Ключ не больше заданного ключа Ключ0;
- tuple(Ключ, Данные) = upFrom_nd(Дер, Ключ0) — недетерминированная функция возвращает в порядке увеличения значения ключа кортеж tuple(Ключ, Данные), удовлетворяющий условию Ключ >= Ключ0. Этую функцию выгодно использовать в коллекторе списков для получения списка всех кортежей tuple(Ключ, Данные), чей ключ Ключ не меньше заданного ключа Ключ0;
- Данные = tryLookUp(Дер, Ключ) — детерминированная функция возвращает данные, сохраненные по заданному ключу Ключ. Если ключ не найден, то функция неуспешна.

Пример 26.2. Следующая программа демонстрирует создание попunique-дерева, обладающего свойством хранения множества данных по одному ключу. В дерево помещаются фамилии детей и их возраст, который служит ключом. Естественной операцией для такого дерева является выборка фамилий: либо по заданному ключу-возрасту, либо по признаку моложе/старше заданного возраста.

```

implement main
  open core, console
clauses
run():-Дер = redBlackTree::empty(),
  Дер1 = redBlackTree::insert(Дер, 12, "Иванов"),
  Дер2 = redBlackTree::insert(Дер1, 8, "Петров"),
  Дер3 = redBlackTree::insert(Дер2, 4, "Сидоров"),
  write("8 лет и младше: ",
        [T|T=redBlackTree::downFrom_nd(Дер3, 8)]), nl,
  write("8 лет и старше: ",
        [T|T=redBlackTree::upFrom_nd(Дер3, 8)]), nl,
  Дер4 = redBlackTree::insert(Дер3, 8, "Жаров"),
  write("только 8-летние: ",
        [T|T=redBlackTree::lookUp_nd(Дер4, 8)]), nl,
  Дер5 = redBlackTree::delete(Дер4, 8, "Петров"),
  write("всего детей: ",
        [T|T=redBlackTree::getAll_nd(Дер5)]), nl,
  _ = readLine().
end implement main
goal
  console::run(main::run).

```

Результат работы программы:

```
8 лет и младше: [tuple(8, "Петров"), tuple(4, "Сидоров")]
8 лет и старше: [tuple(8, "Петров"), tuple(12, "Иванов")]
только 8-летние: ["Жаров", "Петров"]
всего детей:
[tuple(4, "Сидоров"), tuple(8, "Жаров"), tuple(12, "Иванов")]
```

Пример 26.3. В этой программе создается nonunique-дерево, в котором размещаются сведения об учениках некоторых классов, взявших определенное количество книг в школьной библиотеке. Ключом служит название класса, задаваемое строкой, содержащей номер и букву, например "10б". Хранимыми данными являются фамилия и количество взятых книг. Эти данные хранятся в виде кортежа tuple(Фамилия, Количество).

Для сравнения ключей задан собственныйный компаратор, который сравнивает классы только по числу, отбрасывая букву. Для этого введена вспомогательная функция число(Класс), которая отделяет число от строки Класс и конвертирует его в тип unsigned. Сам компаратор задан анонимной функцией Comparator:

```
Comparator = { (I,J)=R:-  
    число(I)>число(J) , !, R=greater; % класс I старше класса J  
    число(I)<число(J) , !, R=less; % класс I младше класса J  
    R=equal } % одногодки
```

В дерево заносятся сведения о четырех учениках и осуществляется три запроса:

```
implement main  
    open core, console  
class predicates  
    число: (string) -> unsigned determ.  
clauses  
    число(Класс) = toTerm(unsigned, Номер) :-  
        string::frontToken(Класс, Номер, _).  
run () :- Comparator = { (I,J)=R:-число(I)>число(J) , !, R=greater;  
                        число(I)<число(J) , !, R=less;  
                        R=equal },  
        Д = redBlackTree::emptyCustom(Comparator),  
        Д1=redBlackTree::insert(Д, "7б", tuple("Иванов", 6)),  
        Д2=redBlackTree::insert(Д1, "8а", tuple("Петров", 5)),  
        Д3=redBlackTree::insert(Д2, "7а", tuple("Сидоров", 4)),  
        Д4=redBlackTree::insert(Д3, "11в", tuple("Жаров", 5)),  
        write("c 1 по 9 классы: ", [T||T=redBlackTree::downFrom_nd(Д4, "9")]),  
        nl,  
        write("c 8 по 11 классы: ", [T||T=redBlackTree::upFrom_nd(Д4, "8")]),  
        nl,  
        write("семиклассники: ", [T||T=redBlackTree::lookUp_nd(Д4, "7")]),  
        nl,  
        _ = readLine().  
end implement main
```

```
goal  
    console::run(main::run) .
```

Результат работы программы:

```
с 1 по 9 классы: [tuple("8а",tuple("Петров",5)),  
                   tuple("7б",tuple("Иванов",6)),  
                   tuple("7а",tuple("Сидоров",4))]  
с 8 по 11 классы: [tuple("8а",tuple("Петров",5)),  
                   tuple("11в",tuple("Жаров",5))]  
семиклассники: [tuple("Сидоров",4),tuple("Иванов",6)]
```

ГЛАВА 27



Массивы

Visual Prolog не располагает языковыми средствами для создания и обработки массивов, однако имеет классы `binary`, `arrayM`, `array2M` и `arrayM_boolean` для выполнения этих операций. В отличие от списков и деревьев, у которых время доступа к элементу зависит от его расположения в коллекции, названные классы реализуют константный доступ к элементам массива посредством классического принципа вычисления адреса элемента по его индексу/индексам.

Класс `binary` позволяет создавать бинарные массивы и эффективно выполнять операции доступа к элементам массивов по известному адресу. На основе этих массивов можно реализовать массивы любых видов. Именно так и были созданы классы для обработки одно- и двумерных массивов и битовых массивов.

Класс `arrayM` реализует операции с одномерными динамическими массивами, класс `array2M` — с двумерными статическими, класс `arrayM_boolean` — с одномерными динамическими битовыми массивами.

Использование этих классов позволяет значительно повысить скорость выполнения многих операций над списками, заменив их массивами.

27.1. Класс `binary`

27.1.1. Создание массивов `binary`

Класс `binary` позволяет создавать массивы в `Atomic` или `NonAtomic` памяти. Различие между `Atomic` и `NonAtomic` заключается в их обработке сборщиком мусора. Память `Atomic` не сканируется сборщиком мусора при поиске указателей, т. к. содержит только неделимые данные: числа, символы, значения `handle`. Память `NonAtomic` сканируется сборщиком мусора, т. к. может содержать указатели, объекты и т. д. Поэтому массивы `Atomic` будут меньше нагружать процессор во время выполнения программы.

Класс `binary` содержит ряд предикатов для создания бинарных массивов:

- `B = createAtomic(N)` — создание массива `B` размером `N` байт;
- `B = createAtomic(N, Size)` — создание массива `B`, содержащего `N` элементов, каждый размером `Size` байт;

- `B = createAtomicFromIntegerList(integer*)` — создание массива `B` из списка целых чисел;
- `B = createNonAtomic(N)` — создание массива `B` размером `N` байт;
- `B = createNonAtomic(N,Size)` — создание массива `B`, содержащего `N` элементов, каждый размером `Size` байт;
- `B = createCopyAtomic(A)` — создание копии `B` массива `A`;
- `B = createCopyNonAtomic(A)` — создание копии `B` массива `A`.

Создание массива из 100 четырехбайтовых элементов выглядит так:

```
A = createAtomic(100, 4)
```

Поскольку индексация элементов массивов начинается с нуля, то при обращении к элементам массива следует иметь в виду, что самый первый элемент массива будет иметь индекс 0, а последний элемент — индекс 99.

27.1.2. Основные операции над массивами *binary*

- `Value = getIndexed_integer(Binary, Index)` — функция возвращает целое число `Value`, хранящееся в позиции `Index` массива `Binary`;
- `Value = getIndexed_real(Binary, Index)` — функция возвращает вещественное число `Value`, хранящееся в позиции `Index` массива `Binary`;
- `Value = getIndexed_unsigned(Binary, Index)` — функция возвращает целое беззнаковое число `Value`, хранящееся в позиции `Index` массива `Binary`;
- `Value = setIndexed_integer(Binary, Index)` — функция записывает целое число `Value` в позицию `Index` массива `Binary`;
- `Value = setIndexed_real(Binary, Index)` — функция записывает вещественное число `Value` в позицию `Index` массива `Binary`;
- `Value = setIndexed_unsigned(Binary, Index)` — функция записывает целое беззнаковое число `Value` в позицию `Index` массива `Binary`.

Полный перечень операций приведен к справочной системе Visual Prolog.

Массив `binary` по сути является одномерным. Для реализации двумерного массива надо располагать его строки друг за другом в массиве `binary`. Следует помнить, что нумерация позиций начинается с нуля. Для вычисления, например, позиции `Index` элемента двумерного массива размером `m` строк и `n` столбцов, находящегося в j -м столбце i -й строки в одномерном массиве `binary`, надо сложить сумму длин строк, находящихся над нашим элементом, т. е. число $i \times n$ с числом j , определяющим позицию нашего элемента в своей строке.

Далее показано создание массива размером 6 байтов и запись чисел 8 и 2 в первую и последнюю позицию массива соответственно:

```
A = createAtomic(6),
setIndexed_integer8(A, 0, 8),
setIndexed_integer8(A, 5, 2),
write(A),
```

Результатом явится сообщение `$[08,00,00,00,00,02]`. Бинарный массив выводится как список байтов, предваренный знаком доллара. Такой синтаксис бинарных массивов можно использовать при ручном манипулировании подобными массивами.

27.2. Класс `arrayM`

27.2.1. Создание одномерных массивов

Класс `arrayM` содержит ряд конструкторов для создания и инициализации одномерных массивов:

- `A = newAtomic(N)` — создание массива размером N элементов. A — ссылка на массив;
- `A = newAtomicFromList(List)` — создание массива из списка List;
- `A = newNonAtomic(N)` — создание массива размером N элементов;
- `A = newNonAtomicFromList(List)` — создание массива из списка List.

Массив может содержать элементы только одного типа. Тип элементов указывается либо при создании массива, либо при первом обращении к нему по записи. Тип элементов массива определяется во время компиляции, поэтому во время выполнения осуществляется захват памяти под массив и инициализация всех элементов массива нулевыми значениями.

В случае, когда тип указывается при создании массива, — например, массива из пяти символов:

```
A = arrayM{char}::newAtomic(5)
```

то каждый элемент массива будет инициализирован кодом '`\u0000`'. Если это массив чисел, то всем элементам будет присвоено нулевое значение.

Когда тип элементов задается при первой записи в массив, например:

```
A = arrayM::newAtomic(5),  
A:insert(tuple(2,'x')),
```

то будут произведены те же действия, что и в первом случае. Единственное отличие — элементу с индексом 2 будет присвоено значение 'x'.

27.2.2. Основные операции над одномерными массивами

Пусть ссылкой на созданный массив является переменная A. Тогда:

- `A:contains(tuple(I,V))` — детерминированный предикат проверки наличия в массиве A элемента V с индексом I;
- `tuple(I,V) = A:getAll_nd()` — недетерминированная функция возвращает кортеж, содержащий индекс элемента I и сам элемент V;
- `V = A:get(I)` — функция возвращает элемент V по его индексу I. Если индекс выходит за пределы массива, то возникает исключительная ситуация;

- `V = A:getValue_nd()` — недетерминированная функция возвращает элемент `V`;
- `A:set(I,V)` — предикат записывает значение `V` по индексу `I`;
- `A:insert(tuple(I,V))` — предикат записывает значение `V` по индексу `I`;
- `A:insertList(tupleList)` — предикат записывает список кортежей `tupleList` в массив. Список кортежей может содержать несмежные элементы, например: `[tuple(2,'x'), tuple(4,'y')]`;
- `A:add(V)` — предикат увеличивает размер массива на один элемент и записывает в конец массива элемент `V`;
- `A:resize(NewSize)` — предикат устанавливает новый размер массива `NewSize`. При уменьшении размера последние элементы массива теряются, при увеличении массив дополняется справа нулевыми элементами. Остальные элементы массива сохраняют свои значения;
- `A:resizeAt(Position,Count)` — предикат изменяет размер массива посредством добавления или удаления элементов в произвольном месте массива. Если количество добавляемых элементов `Count` положительно, то, начиная с позиции `Position`, вставляется `Count` новых элементов, имеющих нулевое значение. Заметьте, что прежний элемент, имеющий позицию `Position`, будет сдвинут на `Count` элементов. Если количество добавляемых элементов `Count` отрицательно, то, начиная с позиции `Position`, будет удалено `Count` элементов.

Полный перечень операций приведен в справочной системе Visual Prolog в классе `map`.

Используя массивы, можно эффективно выполнять те операции, которые долго выполняются на списках. Например, обмен местами первого и последнего элемента списка длиной n выполняется за n шагов. В массиве же такой обмен выполняется за константное время:

```

A = arrayM::newNonAtomic(5),           % создание массива
A:set(0,5),A:set(1,6),A:set(2,7),      %
A:set(3,8),A:set(4,9),                  % заполнение массива
write([I||I=A:getValue_nd()]),nl,       % вывод исходного массива
First = A:get(0),                      % чтение первого элемента
Last = A:get(4),                       % чтение последнего элемента
A:set(0,Last),                         % запись первого элемента
A:set(4,First),                        % запись последнего элемента
write([I||I=A:getValue_nd()]),nl,       % вывод результата

```

В результате мы получим исходный массив и выходной массив:

`[5,6,7,8,9]`

`[9,6,7,8,5]`

При необходимости множественного доступа к отдельным элементам списка выгодно преобразовать список в одномерный массив с помощью предиката `insertList`, после чего выполнить операции с отдельными элементами массива по-

средством предикатов `get` и `set`. Обратное преобразование можно выполнить с помощью предиката `getAll_nd`.

27.3. Класс `array2M`

27.3.1. Создание двумерных массивов

Класс `array2M` содержит ряд конструкторов для создания и инициализации двумерных массивов:

- `A = newAtomic(N,M)` — создание массива размером N строк и M столбцов;
- `A = newNonAtomic(N,M)` — создание массива размером N строк и M столбцов.

Для создания массива из 10 строк и 12 столбцов достаточно вызвать:

```
A = newAtomic(10,12)
```

Индексация элементов начинается с нуля. Поэтому элемент, находящийся в левом верхнем углу, имеет координаты `tuple(0,0)`, а элемент, находящийся в правом нижнем углу, — координаты `tuple(9,11)`.

27.3.2. Основные операции над двумерными массивами

Пусть ссылкой на созданный массив является переменная `A`. Тогда:

- `A:contains(tuple(tuple(I,J),V))` — детерминированный предикат проверки наличия в массиве A элемента V с координатами `tuple(I,J)`;
- `tuple(tuple(I,J),V) = A:getAll_nd()` — недетерминированная функция возвращает кортеж, содержащий координаты элемента `tuple(I,J)` и сам элемент V;
- `V = A:get(tuple(I,J))` — функция возвращает элемент V по его координатам `tuple(I,J)`;
- `V = A:get(I,J)` — функция возвращает элемент V по его координатам I,J;
- `V = A:getValue_nd()` — недетерминированная функция возвращает элемент V;
- `A:set(tuple(I,J),V)` — предикат записывает значение V по координатам `tuple(I,J)`;
- `A:set(I,J,V)` — предикат записывает значение V по координатам I,J;
- `A:insert(tuple(tuple(I,J),V))` — предикат записывает значение V по координатам `tuple(I,J)`;
- `A:insertList(tupleList)` — предикат записывает список кортежей `tupleList` в массив.

Полный перечень операций приведен к справочной системе Visual Prolog.

Матричные операции не характерны для программ на Прологе, однако Visual Prolog достаточно легко выполняет такие операции. Рассмотрим программу получения матрицы A, состоящей из случайных элементов:

```

A = array2M::newAtomic(2,3),           % инициализация матрицы
foreach I=std::fromTo(0,1), J=std::fromTo(0,2) do % цикл по I,J
    A:set(I,J,math::random(10))          % запись случайного числа
end foreach

```

Здесь недетерминированная функция `std::fromTo(X,Y)` пробегает ряд значений от X до Y. Конъюнкция этих функций на откате пробегает все пары значений (I,J). Предикат `A:set(I,J,math::random(10))` записывает в позицию I,J случайное число, взятое из диапазона от 0 до 9.

Или другой пример — транспонирование матрицы случайных чисел A в матрицу B:

```

B = array2M::newAtomic(3,2),           % инициализация матрицы B
foreach I=std::fromTo(0,1), J=std::fromTo(0,2) do % цикл по I,J
    V = A:get(I,J),                   % чтение из позиции I,J матрицы A
    B:set(J,I,V)                    % запись в позицию J,I матрицы B
end foreach,

```

27.4. Класс `arrayM_boolean`

27.4.1. Создание одномерных булевых массивов

Класс `arrayM_boolean` имеет конструктор для создания и инициализации одномерного булева массива.

`A = new(N)` — создание массива размером N битов `false`.

Элементы булева массива могут принимать значения `true` или `false`. Эти значения в памяти компьютера представляются так: 1 и 0, и хранятся в массиве как отдельные биты, поэтому размер массива невелик.

27.4.2. Основные операции над одномерными булевыми массивами

Операции эти такие же, что и для одномерных массивов. Дополнительно введены три предиката для работы с элементами массива, как с флагами:

- `A:clear(Index)` — предикат сбрасывает бит (присваивает значение `false`), имеющий индекс `Index` в массиве A;
- `A:isSet(Index)` — детерминированный предикат успешен, если бит с индексом `Index` установлен (имеет значение `true`);
- `A:set(Index)` — предикат устанавливает бит (присваивает значение `true`), имеющий индекс `Index` в массиве A.

Полный перечень операций приведен в справочной системе Visual Prolog.

27.5. Способы обработки массивов

Цикл `foreach` позволяет выполнять операции со всеми или некоторыми элементами массива. При этом операция должна иметь процедурный режим детерминизма. Например, следующий цикл инициирует элементы квадратной матрицы `A` размером 3×3 , расположенные выше главной диагонали, случайными значениями из диапазона от 0 до 9:

```
A = array2M::newAtomic(3,3),
foreach I=std::fromTo(0,2), J=std::fromTo(I+1,2)  do
    A:set(I,J,math::random(10))
end foreach
```

Здесь `I` — номер строки, `J` — номер столбца. Конечно, перебор пар индексов можно было бы написать декларативно:

```
A = array2M::newAtomic(3,3),
foreach I=std::fromTo(0,2), J=std::fromTo(0,2), I<J
    A:set(I,J,math::random(10))
end foreach
```

но тогда программа была бы менее эффективной, т. к. вызов двух функций:

```
I=std::fromTo(0,2),
J=std::fromTo(0,2),
```

генерирует все пары `I`, `J`, и только условие `I < J` пропускает лишь те элементы, которые выше главной диагонали.

Для проверки того, что определенные элементы массива удовлетворяют какому-либо условию, лучше воспользоваться циклом, в котором откат вызывается детерминированным предикатом проверки условия выхода из цикла. Например, для проверки того, что элементы квадратной матрицы, расположенные выше главной диагонали, больше элементов, расположенных симметрично им относительно главной диагонали, достаточно в цикле с откатом проверять обратное условие — меньше или равно. Это обратное условие будет инициировать откат в случае, когда testируемые элементы удовлетворяют проверяемому условию, т. е. они больше своих симметричных элементов. Если будет найден элемент, который меньше или равен симметричному элементу, то проверяемое в цикле условие выполнится, цикл завершится и будет выведено сообщение `Неуспех`. Если же такой элемент не будет найден, то цикл завершится откатом, и будет выведено сообщение `Успех`.

```
I = std::fromTo(0,2), J = std::fromTo(I+1,2),
A:get(I,J)<=A:get(J,I), write("Неуспех"),!
write("Успех"),
```

Рассмотрим пример нахождения суммы двух квадратных матриц, заполненных случайными числами. Чтобы не плодить новые предикаты, можно воспользоваться анонимным предикатом `G` для заполнения матриц `A` и `B`, и циклом `foreach` для заполнения результирующей матрицы `C` суммами элементов матриц `A` и `B`:

```

D = 3,                                % размер квадратных матриц
A = array2M::newAtomic(D,D),           % инициализация матрицы A
B = array2M::newAtomic(D,D),           % инициализация матрицы B
G = { (Imax,U):-foreach I = std::fromTo(0,Imax),
      J = std::fromTo(0,Imax) do
      U:set(I,J,math::random(10))
      end foreach},                      % анонимный предикат G
G(D-1,A),                             % заполнение матрицы A случайными числами
G(D-1,B),                             % заполнение матрицы B случайными числами
C = array2M::newAtomic(D,D),           % инициализация матрицы C
foreach I=std::fromTo(0,D-1), J=std::fromTo(0,D-1) do
    Value = A:get(I,J) + B:get(I,J),   % сумма элементов
    C:set(I,J,Value)                  % заполнение матрицы C
end foreach,
write([X|X=A:getValue_nd()],nl),        % вывод матрицы A
write([X|X=B:getValue_nd()],nl),        % вывод матрицы B
write([X|X=C:getValue_nd()],nl),        % вывод матрицы C

```

Приведенные программы можно выполнить с помощью рекурсии. Однако в этом случае исходный код будет больше и не столь очевиден, т. к. рекурсивный обход всех элементов двумерного массива потребует четырех аргументов вместо двух. Первые два аргумента — это пара I, J , а вторые два — это максимальный номер строки и столбца в массиве, в нашем случае это 2, 2.

ГЛАВА 28



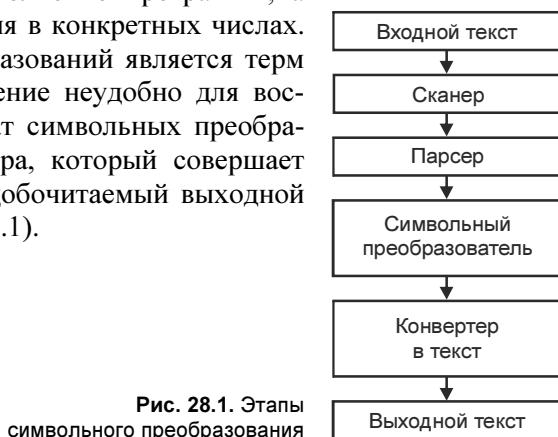
Символьные преобразования

28.1. Этапы анализа текстов

В этой и следующей главах мы рассмотрим обработку текстов, написанных на искусственных языках. Характерным примером искусственных языков являются языки программирования, языки разметки текста, языки запросов к БД, протоколы обмена данными между цифровыми устройствами и т. п.

Первое, что надо сделать с такими текстами, — это провести их лексический и синтаксический анализ. Лексический анализ предшествует синтаксическому анализу и предназначен для разделения входного текста на лексемы языка. Язык входного текста называют *входным языком*. Лексемы языка называют *токенами* (*token*), а сам лексический анализатор — *сканером* (*scanner*). Синтаксический анализ предназначен для построения дерева синтаксического разбора и получения списка термов, выражающих операции над данными, запрограммированные в исходном тексте входной программы. Синтаксический анализатор называют *парсером* (*parser*).

Если входной текст грамматически правилен, то сканер и парсер успешно завершают свою работу и возвращают список термов для выполнения каких-либо действий. Такими действиями могут быть символьные преобразования: интегрирование, дифференцирование, символьное выполнение программы, а также вычисление значения выражения в конкретных числах. Результатом этих символьных преобразований является терм или список термов. Такое представление неудобно для восприятия человеком, поэтому результат символьных преобразований поступает на вход конвертера, который совершает обратное преобразование термов в удобочитаемый выходной текст согласно его грамматике (рис. 28.1).



28.2. Основы анализа текстов на Visual Prolog

28.2.1. Простой лексический анализ

Простейший сканер может быть основан на использовании предиката `fronttoken` из класса `string`:

```
class predicates
    scan: (string, string* [out]) .
clauses
    scan(S, [T|L]) :- fronttoken(S, T, S1), !, scan(S1, L) .
    scan(_, []) .
```

Этому сканеру в первом аргументе передается текст, а через второй аргумент мы получим список лексем. Например, вызов `scan("-12*sin(pi/2+w)", L)` нам вернет список:

```
L = [ "-", "12", "*", "sin", "(" , "pi" , "/" , "2" , "+" , "w" , ")" ]
```

Правила работы предиката `fronttoken` описаны в *приложении 10*. Использование предиката `fronttoken` для распознавания лексем входного языка имеет следующие особенности:

- отрицательные числа разделяются на две лексемы: знак минус и собственно значение числа. В нашем примере число `-12` представлено двумя лексемами `"-"` и `"12"`, а не одной `"-12"`;
- идентификаторами входного языка являются последовательности букв, цифр и знаков подчеркивания, начинающиеся с буквы или со знака подчеркивания. Такие правила для идентификаторов поддерживаются многими языками программирования, но, все же, при необходимости изменить эти правила, их надо описывать в программе дополнительно;
- было бы неплохо для каждой лексемы указывать ее тип: число, константа, переменная или функция и т. п. Например, лексема `"pi"` — это константа, лексема `"w"` — переменная, а лексема `"sin"` — имя функции.

28.2.2. Простой синтаксический анализ

Полученный список лексем служит входными данными для синтаксического анализа, в ходе которого проверяется синтаксическая правильность списка лексем и строится дерево разбора в виде терма или списка термов.

Если в качестве входного текста рассматривать программу на некотором языке программирования, то имя упомянутых термов выражает суть языковой конструкции, а аргументами являются вычисляемые операнды.

Например, оператор присваивания некоторого императивного языка:

```
x = y+10
```

преобразуется парсером в терм вида:

`присв (x, сумма(y, 10))`

Этот терм описывает дерево разбора, изображенное на рис. 28.2.

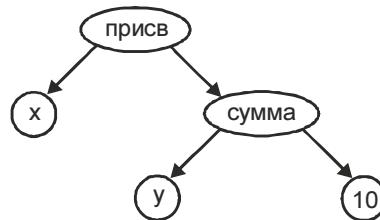


Рис. 28.2. Дерево разбора

Такие термы являются префиксной формой записи языковых конструкций. Слово «префиксная» означает, что оператор преобразования данных является в терме префиксом. В приведенном примере имя присв означает присваивание, причем первому аргументу `x` присваивается значение второго аргумента `сумма(y, 10)`. Имя сумма означает сложение двух аргументов `y` и `10`. Имея такой терм, можно легко выполнить сложение и присваивание. Стоит заметить, что на самом деле структура терма сложнее, т. к. в нем указываются еще и такие параметры, как вид операнда (переменная, константа и т. д.) и тип данных (integer, boolean, real и т. д.). Кроме этого оператор преобразования (присв и сумма) может служить не именем терма, а его первым аргументом, что зачастую упрощает символьные преобразования.

Полученное в виде списка термов внутреннее представление далее подается на вход интерпретатора для исполнения или на вход оптимизатора и компилятора для генерации объектного кода.

28.3. Анализ математических выражений

Здесь мы рассмотрим символьные преобразования математических выражений. Парсер математических выражений призван преобразовать текстовую строку, содержащую некоторое выражение, в терм, соответствующий дереву синтаксического разбора этого выражения.

ПРИМЕР 28.1. Давайте попробуем самостоятельно описать сканер и парсер простых математических выражений, содержащих функции, целые и вещественные числа, а также идентификаторы, играющие роль переменных и констант. Пусть допустимыми операциями в наших выражениях являются унарный минус и бинарные операции сложения, вычитания, умножения, деления и возведения в степень. Приоритеты операций опишем в виде функции:

```

приоритет ("+" )=1 :- !.
приоритет ("-" )=1 :- !.
приоритет ("*")=2 :- !.
приоритет ("/" )=2 :- !.
приоритет ("^")=3.
  
```

Перечень функций зададим предикатом с правилами:

```
функция("abs") :- !.
функция("ln") :- !.
функция("sin") :- !.
функция("cos") :- !.
функция("tan").
```

Пусть функции имеют наивысший приоритет.

Анализ математического выражения, представленного строкой Стока, и его преобразование в терм Выражение опишем в предикате анализ/2:

```
predicates
анализ:(string, выраж [out]) determ.
clauses
анализ(Строка, Выражение) :-
    сканер(Строка, Спис),           % преобразование в список лексем Спис
    парсер(Спис, Ост, Выражение),   % преобразование в терм Выражение
    Ост = [].                      % проверка остатка строки
```

Проверка пустоты остатка строки Ост служит гарантией того, что распознана вся входная строка, а не ее часть. Воспользуемся описанием сканера, приведенным в начале этой главы. А парсер разработаем на основе кода, много лет приводимого в примерах фирмы Prolog Development Center, поставляемых вместе с компилятором Visual Prolog.

Суть синтаксического разбора математических выражений заключается в следующем. Очередные лексемы из списка лексем будем рекурсивно рассматривать как операцию, имеющую наивысший приоритет. Если она таковой не является, то рассмотрим операцию с приоритетом на одну ступень ниже. И так будем действовать до тех пор, пока не найдем правильного сопоставления лексемы и операции. Правильным сопоставлением будет являться такое, при котором приоритет распознанной операции не ниже приоритета той операции, которая следует за ней.

Например, чтобы правильно распознать выражение:

$Z+10-2$

надо убедиться, что после операции сложения следует операция, приоритет которой не выше, чем у сложения. Так как вычитание имеет тот же приоритет, что и сложение, то в первую очередь будет построен терм для сложения:

$+ (Z, 10)$

который будет использоваться в качестве первого аргумента в операции вычитания. Такой анализ операций с равным приоритетом называется *левоассоциативным*. Другими словами, наше выражение будет распознано так, как будто оно записано в виде:

$(Z+10)-2$

В итоге мы должны получить терм вида:

$- (+ (Z, 10), 2)$

Давайте посмотрим, что должно измениться в анализе нашего выражения, если вместо разности указано умножение:

$Z+10*2$

Так как после сложения следует операция умножения, чей приоритет выше, чем у сложения, то в первую очередь будет построен терм для умножения:

$\cdot(10, 2)$

который послужит вторым аргументом в сумме. Поэтому в итоге мы должны получить терм вида:

$+ (Z, \cdot(10, 2))$

Если бы после умножения следовала операция с более высоким приоритетом — например, возведение в степень:

$Z+10*2^3$

то первый терм был бы построен для степени:

$\cdot(2, 3)$

Потом он вошел бы в произведение:

$\cdot(10, \cdot(2, 3))$

И только потом был бы построен терм для операции сложения:

$+ (Z, \cdot(10, \cdot(2, 3)))$

Так как в Visual Prolog нельзя использовать знаки операций в качестве имен термов, то мы поступим следующим образом. Опишем в разделе доменов собственный способ представления выражений:

```
domains
выраж = операция(string, выраж, выраж);
функция(string, выраж);
идентиф(string);
целое(integer);
вещест(real).
```

Здесь операции и функции описываются термами, в которых первый аргумент является строкой, выражающей операцию или имя функции. Тогда сумма $Z+10$ будет преобразована в терм:

$\text{операция}("+" , \text{идентиф}("Z") , \text{целое}(10))$

Внесение знака операции внутрь терма даст нам свободу действий при описании символьных преобразований, т. к. в некоторых ситуациях мы сможем оперировать знаками операций, как переменными.

Строка с функцией, например, синус:

$\sin(2*\pi)$

будет преобразована в терм:

$\text{функция}("sin" , \text{операция}("*" , \text{целое}(2) , \text{идентиф}("pi")))$

Настало время сделать вывод о порядке анализа строки, содержащей математическое выражение. Наивысший приоритет имеет подвыражение, которое может являться:

- выражением в скобках, например: `(z+10)`;
- функцией, например: `ln(z+10)`;
- числом;
- идентификатором, не являющимся именем функции, — например, `sin` не может быть идентификатором, т. к. это имя зарезервировано в качестве имени функции.

Тогда построение термов из лексем имеет следующий порядок анализа признаков:

1. Лексема является унарным минусом.
2. Лексемы являются подвыражением.
3. Лексемы описывают возвведение в степень.
4. Лексемы описывают умножение или деление.
5. Лексемы описывают сложение или вычитание.

В приводимой далее программе использован именно такой порядок анализа. Особенностью кода является то, что парсер начинает разбор выражения, передавая входную строку предикату `плюс`. Предикат `плюс` перед проверкой наличия операций сложения или вычитания вызывает предикат `умнож`. Предикат `умнож` перед проверкой наличия операций умножения или деления вызывает предикат `степень`. Предикат `степень` перед проверкой наличия операции степени проверяет наличие подвыражения путем вызова предиката `подвыраж`:

```
implement main
  open core,console,string
domains
  выраж = операция(string,выраж,выраж) ;
         функция(string,выраж) ;
         идентиф(string);целое(integer);вещест(real).
class predicates
  анализ:(string,выраж [out]) determ.
  сканер:(string,string* [out]). 
  парсер:(string*,string*,выраж) determ (i,o,o).
  степень:(string*,string*,выраж) determ (i,o,o).
  степень1:(string*,string*,выраж,выраж) procedure (i,o,i,o).
  умнож:(string*,string*,выраж) determ (i,o,o).
  умнож1:(string*,string*,выраж,выраж) determ (i,o,i,o).
  плюс:(string*,string*,выраж) determ (i,o,o).
  плюс1:(string*,string*,выраж,выраж) determ (i,o,i,o).
  подвыраж:(string*,string*,выраж) determ (i,o,o).
  приоритет:(string) -> unsigned determ.
  функция:(string) determ.
```

clauses

```
анализ(Строка, Выражение) :- сканер(Строка, Спис),  
парсер(Спис, Ост, Выражение), Ост = [] .
```

```
сканер(Строка, [Лексема | Спис]) :-  
fronttoken(Строка, Лексема, Строка1), !,  
сканер(Строка1, Спис) .  
сканер(_, []) .
```

```
парсер(Спис, Ост, Выраж) :- плюс(Спис, Ост, Выраж) .
```

```
плюс([ "-" | Спис], Ост, Выраж2) :- !, % унарный минус  
умнож(Спис, Ост1, Выраж1),  
плюс1(Ост1, Ост, функция(" - ", Выраж1), Выраж2) .  
плюс(Спис, Ост, Выраж2) :-  
умнож(Спис, Ост1, Выраж1),  
плюс1(Ост1, Ост, Выраж1, Выраж2) .
```

```
плюс1([Опер | Спис], Ост, Выраж1, Выраж3) :-  
приоритет(Опер)=1, !, % операции с приоритетом 1 (+ или -)  
умнож(Спис, Ост1, Выраж2),  
плюс1(Ост1, Ост, операция(Опер, Выраж1, Выраж2), Выраж3) .  
плюс1(Спис, Спис, Выраж, Выраж) .
```

```
умнож(Спис, Ост, Выраж2) :-  
степень(Спис, Ост1, Выраж1),  
умнож1(Ост1, Ост, Выраж1, Выраж2) .
```

```
умнож1([Опер | Спис], Ост, Выраж1, Выраж3) :-  
приоритет(Опер)=2, !, % операции с приоритетом 2 (* или /)  
степень(Спис, Ост1, Выраж2),  
умнож1(Ост1, Ост, операция(Опер, Выраж1, Выраж2), Выраж3) .  
умнож1(Спис, Спис, Выраж, Выраж) .
```

```
степень(Спис, Ост, Выраж2) :-  
подвыраж(Спис, Ост1, Выраж1),  
степень1(Ост1, Ост, Выраж1, Выраж2), ! .
```

```
степень1([Опер | Спис], Ост, Выраж1, операция(Опер, Выраж1, Выраж2)) :-  
приоритет(Опер)=3, % операции с приоритетом 3 (^)  
степень(Спис, Ост, Выраж2), ! .  
степень1(Спис, Спис, Выраж, Выраж) .
```

```
подвыраж([" (" | Спис], Ост, Выраж) :- % выражение в скобках  
парсер(Спис, Ост1, Выраж), Ост1 = [" ") " | Ост], ! .  
подвыраж([Функ, "(" | Спис], Ост, функция(Функ, Выраж)) :-  
функция(Функ), % функция  
парсер(Спис, Ост1, Выраж), Ост1 = [" ") " | Ост], ! .
```

```

подвыраж( [Лексема|Спис] , Спис, целое (Целое) ) :-  

    Целое = tryToTerm(integer, Лексема), !.          % целое число  

подвыраж( [Лексема|Спис] , Спис, веществ (Веществ) ) :-  

    Веществ = tryToTerm(real, Лексема), !.          % вещественное число  

подвыраж( [Имя|Спис] , Спис, идентиф (Имя) ) :-  

    not (функция (Имя) ),                          % не имя функции  

    isname (Имя) .                                % идентификатор

приоритет ("+" )=1:-!.  

приоритет ("-" )=1:-!.  

приоритет ("*" )=2:-!.  

приоритет ("/" )=2:-!.  

приоритет ("^" )=3.

функция ("abs") :-!.  

функция ("ln") :-!.  

функция ("sin") :-!.  

функция ("cos") :-!.  

функция ("tan") .

run() :- Странка = "sin(2*X)^n",  

        Странка1 = toLowerCase(Странка),           % в нижний регистр  

        (анализ(Странка1, Выраж), write(Выраж), !;  

         write("Ошибка")),  

         _ = readchar().  

end implement main  

goal  

    console:::run(main:::run) .

```

Для входной строки "sin(X)^n" результатом работы программы будет терм вида:

```
операция ("^", функция ("sin", идентиф ("x") ), идентиф ("n") )
```

Как можно видеть, в степень возводится синус, а не его аргумент, т. к. функция в нашем парсере имеет приоритет выше, чем степень. Для того чтобы в степень возвести аргумент синуса, надо выражение со степенью взять в скобки:

```
"sin(X^n)".
```

Тогда парсер вернет терм:

```
функция ("sin", операция ("^", идентиф ("x") ), идентиф ("n") ))
```

В случае какой-либо ошибки в выражении программа выведет сообщение **Ошибка**. Способы детального описания и позиционирования ошибки будут рассмотрены в главе 29.

28.4. Парсер математических выражений с произвольной грамматикой

В предыдущем разделе был рассмотрен парсер математических выражений с жестко заданным набором бинарных операций, их приоритетом и ассоциативностью. При необходимости изменить приоритет или ассоциативность операций или добавить новые операции нам пришлось бы модифицировать парсер или вообще переписать его заново.

Этот раздел посвящен разработке универсального парсера математических выражений. Его универсальность заключается в возможности задавать в ходе выполнения программы требуемый набор бинарных операций с приоритетом и ассоциативностью каждой операции, а также определять набор разрешенных имен функций. При этом сам парсер остается без изменений для любого набора операций.

Вначале вспомним необходимые для нас сведения об ассоциативности операций при разборе математических выражений. *Ассоциативность* определяет очередь выполнения операций. Если операция левоассоциативна, то цепочка таких операций вычисляется слева направо. Примером левоассоциативной операции служит сложение. Сумма нескольких слагаемых вычисляется, начиная с левой пары operandов, к сумме которых прибавляется третий operand, потом четвертый и т. д.

$$a+b+c+d = ((a+b)+c)+d.$$

Примером *правоассоциативной* операции является возведение в степень. Вычисление цепочки степеней происходит справа налево:

$$a^b^c^d = a^{(b^{(c^d)})}.$$

ПРИМЕР 28.2. Разработаем универсальный парсер математических выражений, грамматика которых задается в виде фактов. Левую ассоциативность будем обозначать `left`, а правую — `right`. Зададим в базе фактов бинарные операции фактами вида:

`binar(ЗнакОперации, Приоритет, Ассоциативность)`

а функции — фактами вида:

`func(ИмяФункции)`

Вот пример объявления и определения базы фактов, которая задает грамматику математических выражений такого вида, который используется во многих языках программирования, в том числе и в Visual Prolog:

```
domains
    assoc = left; right.
class facts
    binar:(string,unsigned,assoc) .
    func:(string) .
clauses
    binar("+",1,left).
    binar("-",1,left).
```

```

binar("*",2,left).
binar("/",2,left).
binar("^",3,right).
func("abs").
func("ln").
func("sin").
func("cos").
func("tan").

```

Операция возведения в степень в этом примере имеет наивысший приоритет, равный трем, и правую ассоциативность. Остальные операции левоассоциативны. Самые младшие операции — это сложение и вычитание. Набор функций содержит пять разрешенных/поддерживаемых имен.

Будем считать недопустимыми операции с равным приоритетом, но различной ассоциативностью, ибо такие операции могут привести к неоднозначности синтаксического разбора.

Ядро универсального парсера реализовано внешней и внутренней рекурсией. Внешняя рекурсия осуществляется предикатом `parser/4`, а внутренняя — предикатом `parser1/8`.

Далее описана суть работы универсального парсера, после чего следует подробное описание его предикатов.

Внешняя рекурсия передает внутренней рекурсии два операнда `Term` и `Term1`, операцию между ними `Op`, ее приоритет и ассоциативность, а также список лексем `Toks` (от англ. *tokens* — лексемы). Внутренняя рекурсия анализирует полученный список лексем `Toks` и строит структуру выражения, в которую входят указанные операнды `Term` и `Term1` и операция `Op`, а также, возможно, и другие операнды и операции из списка `Toks`. Построение выражения заканчивается:

- либо тогда, когда в списке `Toks` встречается операция с приоритетом ниже, чем приоритет операции `Op`;
- либо тогда, когда операция `Op` левоассоциативна, и в списке `Toks` встречается левоассоциативная операция с таким же приоритетом;
- либо, когда список лексем `Toks` пуст.

При своем завершении внутренняя рекурсия возвращает внешней рекурсии полученное выражение и остаток списка лексем. Если описанные условия завершения рекурсии не выполняются, то сканирование списка `Toks` продолжается.

Внешняя рекурсия, получив от внутренней рекурсии некоторое выражение `Term` и остаток списка лексем `Toks`, отсоединяет от этого списка знак операции `Op` и следующий за ним operand `Term1`, получив после этих действий остаток списка `Toks1`. После чего внешняя рекурсия повторяет вызов внутренней рекурсии, передав ей список `Toks1`, два операнда `Term` и `Term1` и операцию `Op`. И так продолжается до тех пор, пока список лексем не опустеет, либо в этом списке лексем не встретится лексема, которая не описана грамматикой математических выражений.

Описанные варианты правил собраны в табл. 28.1, где рассмотрен разбор списка лексем, начинающегося с выражения $a \text{ Op } b \text{ Op1 } c$, имеющего три операнда a , b , c и две операции Op , Op1 с приоритетами P и $P1$ и ассоциативностью Assoc и Assoc1 соответственно.

Таблица 28.1. Правила анализа выражения $a \text{ Op } b \text{ Op1 } c$

№	Отношения приоритетов	Assoc	Assoc1	Описание
1	$P > P1$	left	left	Рекурсия завершается преобразованием: $a \text{ Op } b \rightarrow \text{операция}(\text{Op}, a, b)$
2		left	right	
3		right	left	
4		right	right	
5	$P = P1$	left	left	
6	$P < P1$	left	left	Вызывается рекурсия <code>parser1</code> для анализа выражения $b \text{ Op1 } c$, которая возвращает некоторый терм <code>операция(...)</code> , после чего еще раз вызывается рекурсия <code>parser1</code> для анализа выражения: $a \text{ Op } \text{операция}(...)$
7		left	right	
8		right	left	
9		right	right	
10	$P = P1$	right	right	

Внутренняя рекурсия `parser1/8` следует логике, описанной в табл. 28.1. Аргументы этого предиката имеют следующее значение:

```
parser1(string*,          % Список лексем
       выраж,            % Операнд1
       string,            % Знак операции Op
       unsigned,          % Приоритет операции Op
       assoc,              % Ассоциативность операции Op
       выраж,            % Операнд2
       выраж,            % Возвращаемое выражение
       string*)           % Возвращаемый остаток списка лексем
```

Определение домена `выраж` аналогично определению из примера 28.1. Первые шесть аргументов входные, два последних — выходные. Внутренняя рекурсия выражается тремя правилами. Первое правило выполняет преобразование, соответствующее вариантам № 1–5 табл. 28.1, второе правило выполняет преобразование, соответствующее вариантам № 6–10, третье правило выполняется тогда, когда список лексем пуст, либо в этом списке встретилась лексема, которая не описана грамматикой математических выражений.

Первое правило имеет вид:

```
parser1([Op1|Toks], Term, Op, Prior, Assoc, Term1,
        операция(Op, Term, Term1), [Op1|Toks]) :-  
    binar(Op1, Prior1, Assoc1),      % Определяем приоритет и ассоциативность
```

```
( Prior > Prior1,           % Варианты № 1 – 4
  Prior = Prior1,
  Assoc = left,
  Assoc1 = left ),!.       % Вариант № 5
```

Входными аргументами являются два операнда Операнд1 и Операнд2 и знак операции с ее приоритетом и ассоциативностью. Возвращаемое выражение:

операция(Op, Term, Term1)

Выходной список лексем [Op1|Toks]. В этот список была возвращена операция Op1 для того, чтобы она не была потеряна при дальнейшем анализе списка лексем.

Второе правило рекурсивное:

```
parser1([Op1|Toks1],Term,Op,Prior,Assoc,Term1,Term0,Rest) :-
  binar(Op1,Prior1,Assoc1),   % Определяем приоритет и ассоциативность
  ( Prior < Prior1,           % Вариант № 6–9
    Prior = Prior1,
    Assoc = right,
    Assoc1 = right ),        % Вариант № 10
  term(Toks1,Term2,Toks2),!,  % Определяем третий operand Term2
  parser1(Toks2,Term1,Op1,Prior1,Assoc1,Term2,Term3,Toks3),
  parser1(Toks3,Term,Op,Prior,Assoc,Term3,Term0,Rest).
```

Возвращаемое выражение: Term0.

Аргумент Term3 в этом выражении был получен с помощью вызова предиката parser1 с аргументами Term1, Term2 и операцией Op1.

Третье правило завершает внутреннюю рекурсию:

```
parser1(Toks,Term,Op,_,_,Term1,операция(Op,Term,Term1),Toks).
```

Внешняя рекурсия имеет следующие аргументы:

```
parser:(string*,          % Список лексем
        выраж,            % Операнд1
        выраж,            % Возвращаемое выражение
        string*)          % Возвращаемый остаток списка лексем
```

В состав внешней рекурсии входит одно рекурсивное правило и одно условие останова рекурсии:

```
parser([Op|Toks],Term,Term0,Rest) :-
  binar(Op,Prior,Assoc),      % Определяем приоритет и ассоциативность
  term(Toks,Term1,Toks1),     % Определяем второй operand Term2
  parser1(Toks1,Term,Op,Prior,Assoc,Term1,Term2,Toks2),!,
  parser(Toks2,Term2,Term0,Rest).
parser(Toks,Term,Term,Toks).
```

В завершение описания хотелось бы пояснить предикат выделения operandов из списка лексем. Под *operandом* будем понимать число, идентификатор переменной, функцию или выражение в скобках. Этот предикат назван *term* и имеет следующее определение:

- Если голова списка лексем преобразуется в целое число, то операнд и есть целое число:

```
term( [Лексема | Спис] , целое (Целое) , Спис) :-  
    Целое = tryToTerm(integer,Лексема) , !.
```

Здесь функция `tryToTerm` пытается преобразовать строку Лексема в целое число Целое. Когда строка Лексема не является целым числом, то функция неуспешна, что вызывает откат назад.

- Если голова списка лексем преобразуется в вещественное число, то операнд и есть вещественное число:

```
term( [Лексема | Спис] , веществ (Веществ) , Спис) :-  
    Веществ = tryToTerm(real,Лексема) , !.
```

- Если голова списка лексем не является именем функции и является допустимым именем Пролога, то операнд есть идентификатор переменной:

```
term( [Имя | Спис] , идентиф (Имя) , Спис) :-  
    not(func(Имя)) , % Не имя функции  
    isname(Имя) . % Идентификатор
```

Здесь следует заметить, что признаки идентификатора переменной схожи во многих языках программирования: *идентификатор* является строкой, содержащей буквы, цифры и знаки подчеркивания, и начинающейся с буквы или знака подчеркивания.

- Если голова списка лексем является именем функции, за которой в круглых скобках следует выражение, то операнд есть функция:

```
term( [Функ, "(" | Спис] , функция (Функ, Выраж) , Ост) :-  
    func(Функ) , % Функция  
    parse(Спис,Выраж,Спис1) , % Выражение  
    Спис1=[")" | Ост] , !. % Правая скобка
```

Для определения выражения в скобках используется вызов самого верхнего предиката `parse(Спис, Выраж, Спис1)`, который по данному списку лексем Спис возвращает выражение Выраж и остаток списка лексем Спис1. От этого остатка Спис1 отсоединяется правая круглая скобка, если таковая есть.

- Если список лексем начинается с левой круглой скобки, за которой следует выражение, заканчивающееся правой круглой скобкой, то операндом является это выражение:

```
term( ["(" | Спис] , Выраж, Ост) :-  
    parse(Спис,Выраж,Спис1) , % Выражение  
    Спис1=[")" | Ост] , !. % Правая скобка
```

Далее приводится имплементация консольного приложения универсального парсера. В ней сканер выполнен в функциональном стиле, а вызываемый предикат для грамматического разбора `analyse/2` использует промежуточный предикат `parse/3` с целью обработки возможного наличия унарного минуса в начале выражения.

Кроме этого предикат `parse/3` выполняет одну промежуточную функцию — отделяет от списка лексем первый operand для того, чтобы передать его внешней рекурсии ядра парсера, т. е. предикату `parser/4`. В предикате `analyse/2` проверяется пустота остатка списка лексем в предположении, что входная строка содержит только математическое выражение. Смысл возвращения нераспознанного остатка списка заключается в том, что парсер математических выражений используют, как правило, в составе парсера более крупных текстов, например, исходных текстов программ. Поэтому после разбора математического выражения такие парсеры должны продолжить анализ исходного текста программы. Остаток исходного текста как раз и находится в том списке лексем, который возвращает парсер математических выражений.

```

implement main
    open core,console,string
domains
выраж = операция(string,выраж,выраж);
функция(string,выраж);
идентиф(string);
целое(integer);
вещест(real);
none().           % Нет выражения (для унарного минуса)
assoc = left;right.
class facts
binar:(string,unsigned,assoc).
func:(string).
class predicates
scan:(string) -> string*.
analyse:(string,выраж) determ (i,o).
parse:(string*,выраж,string*) determ (i,o,o).
parser:(string*,выраж,выраж,string*) determ (i,i,o,o).
parser1:(string*,выраж,string,unsigned,assoc,выраж,
выраж,string*) determ (i,i,i,i,i,i,o,o).
term:(string*,выраж,string*) determ (i,o,o).
clauses
scan(Str)=[T|scan(Str1)]:-fronttoken(Str,T,Str1),!.
scan(_)=[].

analyse(Str,Term0):-parse(scan(Str),Term0,Rest),Rest=[].

parse(["-"]|Toks),Term0,Rest):-
term(Toks,Term,Toks1),
(binar("-",Prior,_),
parser1(Toks1,none,"-",Prior,left,Term,Term1,Toks2),
Term1=операция("-",none,Term2),
parser(Toks2,функция("-",Term2),Term0,Rest);
Term0=функция("-",Term),
Toks1=Rest),!.
```

```

parse(Toks, Term0, Rest) :-  

    term(Toks, Term, Toks1), parser(Toks1, Term, Term0, Rest) .  
  

parser([Op|Toks], Term, Term0, Rest) :-  

    binar(Op, Prior, Assoc),  

    term(Toks, Term1, Toks1),  

    parser1(Toks1, Term, Op, Prior, Assoc, Term1, Term2, Toks2), !,  

    parser(Toks2, Term2, Term0, Rest).  

parser(Toks, Term, Term, Toks) .  
  

parser1([Op1|Toks], Term, Op, Prior, Assoc, Term1,  

        операция(Op, Term, Term1), [Op1|Toks]) :-  

    binar(Op1, Prior1, Assoc1), % Определяем приоритет и ассоциативность  

    ( Prior>Prior1; % Варианты № 1 - 4  

      Prior=Prior1,  

      Assoc=left,  

      Assoc1=left ), !. % Вариант № 5  

parser1([Op1|Toks1], Term, Op, Prior, Assoc, Term1, Term0, Rest) :-  

    binar(Op1, Prior1, Assoc1), % Определяем приоритет  

        % и ассоциативность  

    ( Prior<Prior1; % Вариант № 6-9  

      Prior=Prior1,  

      Assoc=right,  

      Assoc1=right ), !. % Вариант № 10  

    term(Toks1, Term2, Toks2), !, % Определяем третий operand Term2  

    parser1(Toks2, Term1, Op1, Prior1, Assoc1, Term2, Term3, Toks3),  

    parser1(Toks3, Term, Op, Prior, Assoc, Term3, Term0, Rest).  

parser1(Toks, Term, Op, _, _, Term1, операция(Op, Term, Term1), Toks) .  
  

term([Лексема|Спис], целое(Целое), Спис) :-  

    Целое = tryToTerm(integer, Лексема), !. % Целое число  

term([Лексема|Спис], вещественное(Вещественное), Спис) :-  

    Вещественное = tryToTerm(real, Лексема), !. % Вещественное число  

term([Имя|Спис], идентификатор(Имя), Спис) :- % Идентификатор  

    not(func(Имя)),  

    isname(Имя), !.  

term([Функ,"("|Спис], функция(Функ, Выраж), Ост) :- % Функция  

    func(Функ),  

    parse(Спис, Выраж, Ост1), Ост1=[")"]|Ост], !.  

term(["("|Спис], Выраж, Ост) :- % Выражение в скобках  

    parse(Спис, Выраж, Ост1), Ост1=[")"]|Ост] .  
  

run() :-  

    assert(binar("+", 1, left)), % Задаем бинарные операции  

    assert(binar("-", 1, left)),  

    assert(binar("*", 2, left)),  

    assert(binar("/", 2, left)),  

    assert(binar("^", 3, right)),  

    ...

```

```

assert(func("abs")) , % Задаем имена функций
assert(func("ln")) ,
assert(func("sin")) ,
assert(func("cos")) ,
assert(func("tan")) ,

Строка = "-2^3*ln(12)+13.9E-03", % Страна для анализа
(analyse(Строка,Выраж), write(Выраж), !;
write("Ошибка")),
_ = readchar().

end implement main
goal
  console::run(main::run).

```

Результатом работы программы будет терм выражения:

```

операция ("+", операция ("*",
функция ("-", операция ("^", целое (2), целое (3))),
функция ("ln", целое (12))), веществ (0.0139))

```

28.5. Символьное дифференцирование выражений

ПРИМЕР 28.3. Парсер и сканер не играют самостоятельной роли. Они просто преобразуют строку с выражением к виду, удобному для дальнейшей компьютерной обработки. В качестве примера такой обработки давайте рассмотрим символьное дифференцирование, а точнее — напишем программу, которая брала бы производную по указанной переменной от входного выражения, представленного в виде терма. Как вы уже догадались, этот терм является ничем иным, как результатом работы парсера из примера 28.1 или примера 28.2.

Для того чтобы взять производную от выражения по заданной переменной, надо проверить наличие этой переменной в выражении. Для этого воспользуемся предикатом `вхождение`, который рекурсивно проверяет все подтермы входного терма до тех пор, пока не найдет заданную переменную:

```

class predicates
  вхождение:(выраж, выраж) determ.
clauses
  вхождение (Перем, Перем) :- !.      % переменная найдена
  вхождение (функция (_ , Выраж) , Перем) :- 
    вхождение (Выраж, Перем) , !.      % проверяем аргумент функции
  вхождение (операция (_ , Выраж1, Выраж2) , Перем) :- 
    вхождение (Выраж1, Перем) , !;    % проверяем первый аргумент операции
    вхождение (Выраж2, Перем) .       % проверяем второй аргумент операции

```

Когда искомая переменная входит в терм, то предикат `вхождение` будет успешен, иначе — неуспешен.

Правила дифференцирования опишем с помощью предиката производ(Выраж, Перем, Производ). В этом предикате первый аргумент Выраж является дифференцируемым выражением, второй аргумент Перем является переменной дифференцирования, третий аргумент выходной и представляет собой искомую производную. Итак, правила получения производной следующие:

- Если выражение не содержит переменной, по которой берется производная, то производная равна нулю:

производ(Выраж, Перем, целое(0)) :- not(вхождение(Выраж, Перем)), !.

- Если выражением является та самая переменная, по которой берется производная, то производная равна единице:

производ(Выраж, Выраж, целое(1)) :- !.

- Производная суммы/разности равна сумме/разности производных:

производ(операция(Операция, Выраж1, Выраж2), Перем, Производ) :-

(Операция = "+" ; Операция = "-"), !,

производ(Выраж1, Перем, Производ1), % производная первого слагаемого
 производ(Выраж2, Перем, Производ2), % производная второго слагаемого
 Производ = операция(Операция, Производ1, Производ2).

- Производная произведения берется по формуле:

$$(f(x) \cdot g(x))' = f(x)' \cdot g(x) + f(x) \cdot g(x)',$$

которую на Прологе можно записать в виде:

производ(операция("*", Выраж1, Выраж2), Перем, Производ) :-

производ(Выраж1, Перем, Производ1), % производная

% первого выражения

производ(Выраж2, Перем, Производ2), % производная

% второго выражения

A = операция("*", Производ1, Выраж2), % первое произведение

B = операция("*", Выраж1, Производ2), % второе произведение

Производ = операция("+", A, B). % сумма произведений

- Правило для взятия производной частного записывается аналогично. Для сокращения исходного кода это правило в нашем примере не используется.

- Производную степени и показательной функции совместим. Если терм операция("^", Основ, Показ) представляет собой степень $f(x)^n$, то производная равна выражению:

$$n \cdot f(x)^{n-1} \cdot f(x)'.$$

Если терм представляет собой показательную функцию $n^{f(x)}$, то производная равна выражению:

$$n \cdot n^{f(x)} \cdot \ln(n) \cdot f(x)'.$$

```

производ(операция("^^", Основ, Показ), Перем, Производ) :-  

    Вхож1 = if вхождение(Основ, Перем) then true else false end if,  

    Вхож2 = if вхождение(Показ, Перем) then true else false end if,  

    (  

        Вхож1 = true, Вхож2 = false, !,      % производная степени  

        Показ1 = операция("-^", Показ, целое(1)),          % n-1  

        Степень = операция("^^", Основ, Показ1),          % f(x)^n-1  

        Производ1 = операция("*^", Показ, Степень),          % n · f(x)^n-1  

        производ(Основ, Перем, Производ2),          % f(x)'  

        Производ = операция("*^", Производ1, Производ2); % n · f(x)^n-1 · f(x)'  

        Вхож1 = false, Вхож2 = true,      % производная  

                % показательной функции  

        Выраж = операция("^^", Основ, Показ),          % n^{f(x)}  

        Коэф = функция("ln", Основ),          % ln(n)  

        Выраж1 = операция("*^", Выраж, Коэф),          % n^{f(x)} · ln(n)  

        производ(Показ, Перем, Производ1),          % f(x)'  

        Производ = операция("*^", Выраж1, Производ1)    % n^{f(x)} · ln(n) · f(x)'  

    ).
```

7. Производная от $\sin(f(x))$ равна выражению:

$$\cos(f(x)) \cdot f(x)'.$$

```

производ(функция("sin", Выраж), Перем, Производ) :-  

    вхождение(Выраж, Перем), !,  

    производ(Выраж, Перем, Производ1),          % f(x)'  

    Выраж1 = функция("cos", Выраж),          % cos(f(x))  

    Производ = операция("*", Выраж1, Производ1).    % cos(f(x)) · f(x)'
```

Остальные табличные функции рассматривать не будем, ибо приведенных здесь правил достаточно, чтобы остальные записать на Прологе самостоятельно.

На первый взгляд может показаться, что предикаты `вхождение/2` и `производ/3` выполняют всю необходимую работу. Это так, но получаемая производная может иметь такой вид, который желательно бы упростить. Например, производная от выражения $\sin(x)$ будет возвращена предикатом `производ/3` в виде произведения косинуса на единицу $\cos(x) \cdot 1$. Умножение на единицу в этом случае стоило бы удалить.

Для подобного упрощения напишем рекурсивный предикат `упростить(Выраж, УпрощВыраж)`, имеющий восемь правил:

1. Сложение выражения с нулем можно заменить выражением, полученным упрощением исходного выражения:

```
упростить(операция("+" , Выраж1 , Выраж2) , Выраж) :-  

    (Выраж1=целое(0) , !; Выраж1=вещест(0.0)) , % первое слагаемое = 0  

    упростить(Выраж2 , Выраж) , !; % упрощаем второе слагаемое  

    (Выраж2=целое(0) , !; Выраж2=вещест(0.0)) , % второе слагаемое = 0  

    упростить(Выраж1 , Выраж) , !. % упрощаем первое слагаемое
```

2. Умножение выражения на единицу можно заменить выражением, полученным упрощением исходного выражения:

```
упростить(операция("*" , Выраж1 , Выраж2) , Выраж) :-  

    (Выраж1=целое(1) , !; Выраж1=вещест(1.0)) , % первый множитель = 1  

    упростить(Выраж2 , Выраж) , !; % упрощаем второй множитель  

    (Выраж2=целое(1) , !; Выраж2=вещест(1.0)) , % второй множитель = 1  

    упростить(Выраж1 , Выраж) , !. % упрощаем первый множитель
```

3. Умножение выражения на ноль равно нулю:

```
упростить(операция("*" , Выраж1 , Выраж2) , целое(0)) :-  

    (Выраж1=целое(0) , !; Выраж1=вещест(0.0)) , !; % первый множитель = 0  

    (Выраж2=целое(0) , !; Выраж2=вещест(0.0)) , !. % второй множитель = 0
```

4. Натуральный логарифм от числа e равен единице:

```
упростить(функция("ln" , идентиф("e")) , вещест(1.0)) :-! .
```

5. Синус π равен нулю:

```
упростить(функция("sin" , идентиф("pi")) , вещест(0.0)) :-! .
```

6. Для упрощения операции надо вначале упростить аргументы этой операции, а потом упростить саму операцию. При этом надо проверить, что произошло упрощение хотя бы одного аргумента операции, иначе можно попасть в бесконечный цикл:

```
упростить(операция(Операция , Выраж1 , Выраж2) , Выраж) :-  

    упростить(Выраж1 , Выраж3) , % упрощаем первый аргумент  

    упростить(Выраж2 , Выраж4) , % упрощаем второй аргумент  

    (not(Выраж1 = Выраж3) , !; not(Выраж2 = Выраж4)) , !, % упрощение  

    упростить(операция(Операция , Выраж3 , Выраж4) , Выраж) .
```

7. Для упрощения функции надо вначале упростить аргумент этой функции, а потом упростить саму функцию. При этом надо проверить, что произошло упрощение аргумента функции, иначе можно попасть в бесконечный цикл:

```
упростить(функция(Имя , Выраж1) , Выраж) :-  

    упростить(Выраж1 , Выраж2) , % упрощаем аргумент функции  

    not(Выраж1 = Выраж2) , !, % упрощение было  

    упростить(функция(Имя , Выраж2) , Выраж) .
```

8. Если ни одно правило упрощения не выполнилось, то упрощенное выражение равно исходному:

```
упростить(Выраж , Выраж) .
```

Последнее, что нам осталось, — это выполнить обратное преобразование терма в текст, понятный для человека. Для этого напишем функцию

Строка = текст(Приоритет, Выражение) .

Здесь функции текст передается самый низкий приоритет Приоритет, который равен 1, и терм Выражение, который нужно преобразовать в строку Странка. Приоритет нужен для того, чтобы решить вопрос — стоит ли заключать выражение Выражение в скобки или нет. Параметр Приоритет имеет значение приоритета той операции, которая расположена в дереве разбора выше выражения Выражение. Если в вызове функции текст(Приоритет, Выражение) параметр Приоритет больше приоритета операции в выражении Выражение, то это выражение надо заключать в скобки, иначе выражение надо оставить без скобок. К примеру, рассмотрим терм:

```
* (3, +(pi, 45))
```

Приоритет умножения больше приоритета сложения, при этом умножение находится в дереве разбора над суммой +(pi, 45). Следовательно, сумму надо заключить в скобки:

```
3 * (pi + 45)
```

Если этого не сделать, то текстовое представление терма * (3, +(pi, 45)) будет неверным:

```
3 * pi + 45
```

Правила функции текст/2 можно описать следующим образом:

1. Числа и идентификаторы возвращаются функцией в виде непосредственных значений. При этом числа преобразуются в строку с помощью функции `tostring`:

```
текст(_, целое(Число)) = tostring(Число) :-! .
текст(_, вещественное(Число)) = tostring(Число) :-! .
текст(_, идентифик(Имя)) = Имя :-! .
```

2. Терм функции функция(Имя, Выраж) преобразуется в строку, которая собирается из имени функции Имя и текстового представления аргумента текст(1, Выраж), обрамленного скобками:

```
текст(_, функция(Имя, Выраж)) = format("%(%)", Имя, текст(1, Выраж)) :-! .
```

3. Терм операции операция(Опер, Выраж1, Выраж2) преобразуется в строку Текст. При этом преобразование зависит от соотношения приоритета вышестоящей в дереве разбора операции и текущей операции:

```
текст(Приор1, операция(Опер, Выраж1, Выраж2)) = Текст :-  
    Приор = приоритет(Опер),      % приоритет текущей операции  
    (Приор1 > Приор,             % если приоритет вышестоящей операции выше  
     Текст = format("(% % %)",      % то используем скобки  
                    текст(Приор, Выраж1), Опер, текст(Приор, Выраж2)), !;  
    Текст = format("% % %",        % иначе — строим строку без скобок  
                    текст(Приор, Выраж1), Опер, текст(Приор, Выраж2))).
```

Сейчас мы можем представить всю программу символьного дифференцирования:

```

implement main
    open core, console, string
domains
    выраж = операция(string, выраж, выраж) ;
        функция(string, выраж) ;
        идентиф(string) ; целое(integer) ; веществ(real) .
class predicates
    вхождение:(выраж, выраж) determ.
    производ: (выраж, выраж, выраж [out]) determ.
    упростить: (выраж, выраж [out]) .
    текст: (unsigned, выраж) -> string determ.
    приоритет: (string) -> unsigned determ.
clauses
    приоритет ("+" )=1:-! .
    приоритет ("-" )=1:-! .
    приоритет ("*")=2:-! .
    приоритет ("/" )=2:-! .
    приоритет ("^" )=3 .

    вхождение (Перем, Перем) :- ! .
    вхождение (функция (_ , Выраж) , Перем) :- !
        вхождение (Выраж, Перем) , ! .
    вхождение (операция (_ , Выраж1, Выраж2) , Перем) :- !
        вхождение (Выраж1, Перем) , !;
        вхождение (Выраж2, Перем) .

    производ (Выраж, Перем, целое (0) ) :- !
        not (вхождение (Выраж, Перем) ) , ! .
    производ (Выраж, Выраж, целое (1) ) :-! .
    производ (операция (Операция, Выраж1, Выраж2) , Перем, Производ) :- !
        (Операция="+" ; Операция="-" ) , ! ,
        производ (Выраж1, Перем, Производ1) ,
        производ (Выраж2, Перем, Производ2) ,
        Производ = операция (Операция, Производ1, Производ2) .
    производ (операция ("*", Выраж1, Выраж2) , Перем, Производ) :- !
        производ (Выраж1, Перем, Производ1) ,
        производ (Выраж2, Перем, Производ2) ,
        А = операция ("*", Производ1, Выраж2) ,
        Б = операция ("*", Выраж1, Производ2) ,
        Производ = операция ("+", А, Б) .

    производ (операция ("^", Основ, Показ) , Перем, Производ) :- !
        Вхож1 = if вхождение (Основ, Перем) then true else false end if,
        Вхож2 = if вхождение (Показ, Перем) then true else false end if,
        (
        Вхож1 = true, Вхож2 = false, !, % производная степени

```

```

Показ1 = операция ("-", Показ, целое(1)),
Степень = операция ("^", Основ, Показ1),
Производ1 = операция ("*", Показ, Степень),
производ(Основ, Перем, Производ2),
Производ = операция ("*", Производ1, Производ2);
Вхож1 = false, Вхож2 = true, % производная показательной функции
Выраж = операция ("^", Основ, Показ),
Коэффиц = функция ("ln", Основ),
Выраж1 = операция ("*", Выраж, Коэффиц),
производ(Показ, Перем, Производ1),
Производ = операция ("*", Выраж1, Производ1)
).

```

```

производ(функция ("sin", Выраж), Перем, Производ) :-
    входжение (Выраж, Перем), !,
    производ(Выраж, Перем, Производ1),
    Выраж1 = функция ("cos", Выраж),
    Производ = операция ("*", Выраж1, Производ1).

```

```

упростить (операция ("+", Выраж1, Выраж2), Выраж) :-
    (Выраж1=целое(0), !; Выраж1=вещест(0.0)),
    упростить (Выраж2, Выраж), !;
    (Выраж2=целое(0), !; Выраж2=вещест(0.0)),
    упростить (Выраж1, Выраж), !.

```

```

упростить (операция ("*", Выраж1, Выраж2), Выраж) :-
    (Выраж1=целое(1), !; Выраж1=вещест(1.0)),
    упростить (Выраж2, Выраж), !;
    (Выраж2=целое(1), !; Выраж2=вещест(1.0)),
    упростить (Выраж1, Выраж), !.

```

```

упростить (операция ("*", Выраж1, Выраж2), целое(0)) :-
    (Выраж1=целое(0), !; Выраж1=вещест(0.0)), !;
    (Выраж2=целое(0), !; Выраж2=вещест(0.0)), !.

```

```

упростить (функция ("ln", идентиф("e")) , веществ(1.0)) :-! .
упростить (функция ("sin", идентиф("pi")) , веществ(0.0)) :-! .
упростить (операция (Операция, Выраж1, Выраж2), Выраж) :-
```

```

    упростить (Выраж1, Выраж3),
    упростить (Выраж2, Выраж4),
    (not (Выраж1 = Выраж3), !; not (Выраж2 = Выраж4)), !,
    упростить (операция (Операция, Выраж3, Выраж4), Выраж).

```

```

упростить (функция (Имя, Выраж1), Выраж) :-
    упростить (Выраж1, Выраж2), not (Выраж1 = Выраж2), !,
    упростить (функция (Имя, Выраж2), Выраж).

```

```
упростить (Выраж, Выраж).
```

```
текст (_ , целое(Число)) = tostring(Число) :-! .
```

```
текст (_ , веществ(Число)) = tostring(Число) :-! .
```

```
текст (_ , идентиф(Имя)) = Имя :-! .
```

```

текст( _, функция(Имя, Выраж) ) =
    format("%(%)", Имя, текст(1, Выраж)) :-! .
текст(Приор1, операция(Опер, Выраж1, Выраж2)) = Текст :-
    Приор = приоритет(Опер),
    (Приор1 > Приор,
    Текст = format("(% % %)",
        текст(Приор, Выраж1), Опер, текст(Приор, Выраж2)), !;
    Текст = format("% % %",
        текст(Приор, Выраж1), Опер, текст(Приор, Выраж2))).

run() :-
    (Терм = операция("^", функция("sin", идентиф("x")), идентиф("n")),
    производ(Терм, идентиф("x"), Производ),
    упростить(Производ, Производ1),
    write(текст(1, Производ1)), !;
    write("Ошибка")),
    _ = readchar().
end implement main
goal
    console:::run(main:::run).

```

Входной терм в этой программе

операция("^", функция("sin", идентиф("x")), идентиф("n"))

образован парсером из строки "sin(X)^n". Программа использует правило взятия производной от степенной функции, т. к. переменной, по которой берется производная, является x. Программа выведет производную в виде:

$n * \sin(x) ^ (n - 1) * \cos(x)$

При замене переменной дифференцирования x на n будет использоваться правило взятия производной от показательной функции. Программа выведет производную в виде:

$\sin(x) ^ n * \ln(\sin(x))$

28.6. Калькулятор

В этом разделе мы рассмотрим способы вычисления значений арифметических выражений, которые представлены в виде термов. Обработку таких исключительных ситуаций, как, например, деление на ноль или взятие логарифма отрицательного числа, мы сейчас выполнять не будем, оставив это для главы 29. Здесь же мы рассмотрим простые способы вычислений в конкретных значениях.

Способ 1

Первый способ заключается в непосредственном выполнении математических операций. Он реализуется единственной функцией калькулятор/1, аргументом которой является терм, принадлежащий домену выраж:

domains

```
выраж = операция(string, выраж, выраж);
функция(string, выраж);
идентиф(string);
целое(integer);
вещест(real).
```

Пусть эта функция возвращает значения, представленные вещественными числами:

class predicates

```
калькулятор:(выраж) -> real determ.
```

Выражения, представляющие числа, вычислять не надо, достаточно вернуть значение числа:

clauses

```
калькулятор(целое(X)) = X:-!.
калькулятор(вещест(X)) = X:-!.
```

Если выражение обозначает идентификатор переменной, значение которой нам неизвестно, то и вычислять нечего. Однако если идентификатор является именем константы, то функция должна вернуть ее значение. В качестве примера констант используем две константы, определенные в классе math:

```
калькулятор(идентиф("pi")) = math::pi:-!.
калькулятор(идентиф("e")) = math::e:-!.
```

Для выполнения операции необходимо вычислить значения аргументов, а потом производить собственно операцию. Значения аргументов определяются путем вызова этой же функции калькулятор:

```
калькулятор(операция("+", X, Y)) = калькулятор(X)+калькулятор(Y):-!.
калькулятор(операция("-", X, Y)) = калькулятор(X)-калькулятор(Y):-!.
калькулятор(операция("*", X, Y)) = калькулятор(X)*калькулятор(Y):-!.
калькулятор(операция("/", X, Y)) = калькулятор(X)/калькулятор(Y):-!.
калькулятор(операция("^", X, Y)) = калькулятор(X)^калькулятор(Y):-!.
```

При вычислении значения функций поступаем аналогичным образом:

```
калькулятор(функция("abs", X)) = abs(калькулятор(X)):-!.
калькулятор(функция("ln", X)) = ln(калькулятор(X)):-!.
калькулятор(функция("sin", X)) = sin(калькулятор(X)):-!.
калькулятор(функция("cos", X)) = cos(калькулятор(X)):-!.
калькулятор(функция("tan", X)) = tan(калькулятор(X)).
```

Способ 2

Во многих случаях обработку операций и функций производят отдельно, группируя предложения функции калькулятор/1 по отдельным блокам. В некотором смысле это облегчает написание и анализ калькулятора. Поэтому второй способ предполагает использование дополнительных функций: опер — для выполнения операций и функ — для вычисления функций. Обработка чисел и идентификаторов констант остается такой же, как и в первом способе:

```

class predicates
    опер: (string, выраж, выраж) -> real determ.
    функ: (string, выраж) -> real determ.
    калькулятор: (выраж) -> real determ.

clauses
    калькулятор(целое(X)) = X:-!.
    калькулятор(вещест(X)) = X:-!.
    калькулятор(идентиф("pi")) = pi:-!.
    калькулятор(идентиф("e")) = e:-!.
    калькулятор(операция(Опер, X, Y)) = опер(Опер, X, Y):-!.
    калькулятор(функция(Функ, X)) = функ(Функ, X).
    опер("+", X, Y) = калькулятор(X)+калькулятор(Y):-!.
    опер("-", X, Y) = калькулятор(X)-калькулятор(Y):-!.
    опер("*", X, Y) = калькулятор(X)*калькулятор(Y):-!.
    опер("/", X, Y) = калькулятор(X)/калькулятор(Y):-!.
    опер("^", X, Y) = калькулятор(X)^калькулятор(Y).
    функ("abs", X) = abs(калькулятор(X)):-!.
    функ("ln", X) = ln(калькулятор(X)):-!.
    функ("sin", X) = sin(калькулятор(X)):-!.
    функ("cos", X) = cos(калькулятор(X)):-!.
    функ("tan", X) = tan(калькулятор(X)).

```

Способ 3

Отличием третьего способа от предыдущих является использование анонимных функций. Суть способа рассмотрим на примере вычисления значения термов сложения и вычитания:

```

операция("+", X, Y)
операция("-", X, Y)

```

Перед вычислением суммы или разности следует вычислить значения аргументов X и Y . После этого хотелось бы вычислить значение любого из этих термов вот таким образом:

```
калькулятор(операция(Z, X, Y)) = Z(калькулятор(X), калькулятор(Y)):-!.
```

Однако здесь возникает одно препятствие. Переменная Z не может быть именем функции $Z(X, Y)$, т. к. в терме $\text{операция}(Z, X, Y)$ эта переменная принадлежит домену `string`. Поэтому вместо имени Z функции $Z(X, Y)$ следовало бы использовать какую-то функцию преобразования строки — например: "+" или "-" в функцию сложения или вычитания: `опер("+")` или `опер("-")`. Тогда вместо функции $Z(X, Y)$ мы имеем право использовать каррированную функцию `опер(Z)(X, Y)` — например, `опер("+")(X, Y)`. Такую функцию можно задать в виде отображений знаков операций "+" и "-" на соответствующие им анонимные функции:

```

опер["+"] = { (X, Y) = X + Y }:- !.
опер["-"] = { (X, Y) = X - Y }:- !.

```

Тогда для вычисления значения суммы и разности можно использовать единственное правило:

```
калькулятор(операция(Z,X,Y)) = опер(Z)(калькулятор(X),калькулятор(Y)):-!.
```

Для реализации третьего способа нам осталось объявить функцию `опер(Z)`. Мы знаем, что ее входной аргумент принадлежит домену `string`, а результатом является анонимная функция от двух переменных. Поэтому вначале введем в программу новый домен для бинарных операций, которые выполняются нашими анонимными функциями:

```
domains
    доменОпер {In,Out} = (In, In) -> Out.
```

Вот теперь можно объявить нашу функцию:

```
class predicates
    опер: (string) -> доменОпер{real,real} determ.
```

Суть вычисления значений синусов, логарифмов и других функций подобна приведенным здесь рассуждениям.

ПРИМЕР 28.4. Этот пример показывает реализацию третьего способа вычисления значения арифметических выражений, определенных в примере 28.1 пятью арифметическими операциями `+`, `-`, `*`, `/`, `^`, пятью функциями `abs`, `ln`, `sin`, `cos`, `tan` и унарным минусом над множеством вещественных чисел и констант π и e .

Домен `доменФунк` объявлен с режимом `determ`, т. к. среди поддерживаемых функций есть логарифм, определенный в Visual Prolog на множестве беззнаковых вещественных чисел `uReal`. В анонимной функции вычисления логарифма производится попытка преобразования аргумента логарифма к типу `uReal`. В случае неуспеха этой попытки анонимная функция даст откат назад.

```
implement main
    open core,console,math
domains
    выраж = операция(string,выраж,выраж);
    функция(string,выраж);
    идентиф(string);
    целое(integer);
    веществ(real).

    доменОпер{In,Out} = (In, In) -> Out.          % домен операции
    доменФунк{In,Out} = (In) -> Out determ.        % домен функции

class predicates
    опер: (string) -> доменОпер{real,real} determ.
    функция: (string) -> доменФунк{real,real} determ.
    калькулятор: (выраж) -> real determ.

clauses
    опер("+" ) = { (X, Y) = X + Y }:- !.
    опер("-" ) = { (X, Y) = X - Y }:- !.
    опер("*" ) = { (X, Y) = X * Y }:- !.
```

```

опер("//") = { (X, Y) = X / Y}:- !.
опер("^") = { (X, Y) = X^Y}.

функ("-") = { (X) = -X}:- !.           % унарный минус
функ("abs") = { (X) = abs(X)}:- !.
функ("ln") = { (X) = ln(Z):-Z=tryConvert(uReal,X)}:- !.
функ("sin") = { (X) = sin(X)}:- !.
функ("cos") = { (X) = cos(X)}:- !.
функ("tan") = { (X) = tan(X)}.

калькулятор(целое(X)) = X:-!.
калькулятор(вещест(X)) = X:-!.
калькулятор(идентиф("pi")) = pi:-!.
калькулятор(идентиф("e")) = e:-!.
калькулятор(операция(Z,X,Y)) =
    опер(Z)(калькулятор(X),калькулятор(Y)):-!.
калькулятор(функция(F,X)) = функ(F)(калькулятор(X)).

run() :-
    Терм = операция("+",
        функция("sin",операция("//",идентиф("pi"),целое(6))),
        операция("^",идентиф("e"),целое(2))),
    ( write(калькулятор(Терм)),!,
       write("Ошибка")),
       _ = readchar().
end implement main
goal
    console::run(main::run).

```

На вход калькулятора подается терм выражения "sin(pi/6) + e^2". На выходе имеем результат: 7.88905609893068.

28.7. Задания для самостоятельного решения

Задача 28.1. Разработать парсер и калькулятор логических выражений, которые допускают операции, указанные в табл. 28.2.

Таблица 28.2. Логические операции

Операция	Разрешенные обозначения
Логическое сложение	or, +
Сложение по модулю 2	xor
Логическое умножение	and, *
Отрицание	not (Выражение), ! (Выражение)
Эквивалентность	=, ~
Неэквивалентность	!=, !~

Задача 28.2. Разработать универсальный парсер, добавив ассоциативность функций. Например, если имена функций логарифм или синус располагаются слева от своего аргумента, то знак факториала располагается справа.

Задача 28.3. Разработать универсальный парсер, добавив возможность распознавания функций, обозначаемой скобками. Например, вместо функции модуля выражения `abs(V)`, использовать математическую запись вида $|V|$.

Задача 28.4. Разработать программу для упрощения логических выражений.

Задача 28.5. Разработать программу символьного интегрирования.

Задача 28.6. Разработать парсер простых HTML-текстов с выбранным вами набором тегов.

ГЛАВА 29



Интерпретатор программ

Эта глава посвящена анализу и интерпретации простейших ассемблерных программ и рассчитана на читателей, знакомых с основами языка ассемблера. Для упрощения примеров в интерпретатор не включен парсер арифметических выражений, рассмотренный в главе 28.

29.1. Лексический анализ

Наряду с лексическим анализом возложим на наш сканер ассемблерных текстов ряд дополнительных задач:

- Определение и сохранение позиции каждой лексемы в тексте. Это необходимо для формирования сообщения об ошибке в том случае, когда лексема не принадлежит входному языку. Позиция лексемы сохраняется вместе с самой лексемой в терме вида `token` (Лексема, Позиция). Поэтому на выходе сканера мы получим список таких термов, если входной текст правильный. Иначе будет вызвано исключение с сообщением об ошибке.
- Обработка лексических ошибок. Здесь важным моментом является как можно более детальное описание ошибки. Однако следует иметь в виду, что процесс локализации и детализации ошибок довольно кодоемок и неоднозначен. Чем сложнее грамматика входного текста, тем более неоднозначно локализуется и трактуется каждая ошибка.
- Обработка комментариев. Сканер идентифицирует ограничители комментариев, сами же комментарии никак не анализируются, но их длину учитывает при определении позиции лексем.
- Обработка символов перехода на новую строку. Если для грамматики входного языка такой переход существенен, то в выходной список термов `token` (Лексема, Позиция) вставляется нульварный терм вида `endOfLine`. Он не имеет аргументов, т. к. позиция его в тексте не важна, и служит просто маркером перехода на новую строку.

Такие сканеры можно строить различными способами. К первому способу можно отнести посимвольный анализ входного текста. Этот способ трудоемок и кодоемок.

Другой способ может быть основан на использовании полексемного анализа с помощью предиката `fronttoken/3` с дополнительной обработкой тех правил грамматики входного языка, которые не учтены в правилах работы предиката `fronttoken/3`.

Если лексемы входного языка регистронезависимы, то перед анализом лучше привести весь текст к одному регистру. Если язык чувствителен к символам перехода на новую строку, то в этом случае предсканерную и послесканерную обработку можно проводить двумя способами.

Согласно первому способу следует разделить входной текст `S` на список строк `SL` с помощью функции `split_delimiter(Текст, Разделитель)`. Роль разделителя должен играть символ перехода на новую строку. Полученный список строк `SL` передать сканеру, который каждую строку будет анализировать отдельно и на выходе вернет список списков лексем `TTL`, в котором каждая строка будет представлена отдельным списком лексем. При необходимости список списков лексем `TTL` можно объединить в общий список `TL` с помощью функции `appendList`:

```
S = "Некоторый входной текст"
S1 = toLowerCase(S),                      % приведение к нижнему регистру
SL = split_delimiter(S1, "\n"),            % разделение на список строк
TTL = scan(0, SL),                        % вызов сканера
TL = appendList(TTL)                      % получение списка лексем
```

К недостаткам этого способа можно отнести сложность обработки многострочных комментариев. Достоинство — простота сканера.

Второй способ основан на поиске и обработке символов перехода на новую строку внутри сканера. При этом сам сканер несколько усложняется, но в целом эффективность такого способа выше, как при наличии ошибок во входном тексте, так и без таковых:

```
S = "Некоторый входной текст"
S1 = toLowerCase(S),                      % приведение к нижнему регистру
TL = scan(0, S)                          % вызов сканера
```

В обоих способах при вызове функции `scan/2` ей передается номер позиции первого символа входного текста `s`. Это объясняется необходимостью выбора способа позиционирования символов: начинаем счет с нуля или с единицы. В приведенных примерах отсчет начинается с нуля, поэтому самый первый символ текста занимает нулевую позицию. Выбор подхода к позиционированию символов зависит от текстового редактора, с которым работает сканер, и от предпочтений программиста.

ПРИМЕР 29.1. Давайте попробуем написать сканер простейшего варианта языка ассемблера для процессора с 32-битной архитектурой. Пусть набор регистров сокращен до двух: `eax` и `ebx`, а система команд содержит три команды: `mov`, `add` и `inc`. Все лексемы регистронезависимы. Адресация операндов регистрация и непосредственная. Однострочные комментарии начинаются с символа `;`. В дальнейшем набор регистров, состав команд и способы адресации можно расширить. Формальное

описание грамматики нашего языка ассемблера сводится к двум правилам, содержащим инструкции с двумя операндами и с одним операндом:

```
instr ::= opCode2 operand1, operand2
instr ::= opCode1 operand1
```

Нетерминалы языка сводятся к терминалам по правилам:

```
opCode2 ::= mov | add
opCode1 ::= inc
operand1 ::= eax | ebx
operand2 ::= operand1 | number
```

ЗАМЕТКА

Тут приведено формальное описание ассемблера на языке грамматик, а не код на языке Visual Prolog.

Здесь `opCode2` — код операции с двумя operandами, `opCode1` — код операции с одним operandом. Каждая строка ассемблерной программы содержит не более одной команды. Первый operand `operand1` может быть только регистром, второй operand `operand2` может быть как регистром, так и числом. Нетерминал `number` является целым числом со знаком и имеет тривиальное описание, которое здесь не приводится.

Все лексемы языка перечислены в домене `lex`. Домен `tok` определяет способ хранения лексемы вместе с ее позицией в тексте, а также содержит признак конца строки `eol`:

```
domains
lex = eax; ebx;                                % регистры
      mov; add; inc;                            % команды
      imm(integer Number);                     % непосредственный operand
      comma.                                     % запятая
tok = t.lex Token, charCount Position); eol.
```

Программа использует способ разделения входного текста на список строк перед вызовом сканера. Механизм сканирования основан на сравнении каждой лексемы входного текста с терминальными лексемами, заданными грамматикой языка. Лексемы вычленяются слева от входного текста предикатом `fronttoken`. Терминальные лексемы хранятся в специальной базе терминальных лексем. Если лексема входного текста является терминальной, то она добавляется в список распознанных лексем. Если очередная лексема не принадлежит лексемам языка, то вызывается исключение, которому передается неправильная лексема и ее позиция в тексте. Для определения позиции лексемы, найденной предикатом `fronttoken`, к исходной строке применяется функция `search/2` из класса `string`, которая возвращает позицию подстроки в строке.

Терминальные лексемы определены в базе фактов:

```
isOpCode("mov", 2, mov).           % (ИмяОперации, Арность, КодОперации)
isOpCode("add", 2, add).
isOpCode("inc", 1, inc).
```

```
isReg("eax", eax).                                % (ИмяРегистра, Регистр)
isReg("ebx", ebx).
```

Проверку того, что лексема `T` является целым числом, осуществляет функция `tryToTerm(integer,T)`. Когда перед числом следует унарный минус, то предикат `fronttoken` надо применять дважды. Первый раз — для отделения знака минус, второй раз — для отделения собственно числа. Сам сканер описан рекурсивно. В теле рекурсии вызывается функция `getTok/5`, которая определяет принадлежность лексемы языку. Так как лексема языка может строиться более чем из одной лексемы, полученной от предиката `fronttoken`, то функции `getTok/5` передается первая лексема `ПерЛексема`, ее позиция `ПозПерЛексемы` и остаток входного текста `ОстатокТекста`. Сама функция в свою очередь возвращает построенную лексему `Ток`, а также своими четвертым и пятым параметрами возвращает позицию нового остатка текста `ПозНовОстатка` и сам новый остаток текста `НовОстатокТекста`:

```
Tok = getTok(ПерЛексема,
             ПозПерЛексемы,
             ОстатокТекста,
             ПозНовОстатка [out],
             НовОстатокТекста [out])
```

Сообщение об ошибке формируется с использованием строковой константы `scanErr`, нераспознанной лексемы и ее позиции. Функция сканирования `scan` выполняется под наблюдением. Если наступает исключительная ситуация, то она перехватывается обработчиком исключений `myHandler`, в котором выделяется сообщение об ошибке с помощью функции `tryGetExtraInfo`. Это сообщение выводится на экран и процесс сканирования на этом завершается. Само исключение вызывается предикатом `raise_user`, через который и передается сообщение об ошибке обработчику исключений. Далее приведена программа сканера:

```
implement main
  open core, console, string, exception
constants
  scanErr = "Не распознана лексема: ~s в позиции: ~d".
domains
  lex = eax; ebx;                                % регистры
         mov; add; inc;                            % команды
         num(integer Number);                     % целое число
         comma.                                    % запятая
tok = t(lex Token, charcount Position); eol.
class facts
  isOpCode: (string Name, unsigned Arity, lex OpCode).
  isReg: (string Register, lex Register).
class predicates
  scan: (charcount, string*) -> tok** procedure.
  scanLine: (charcount, string, charcount [out]) -> tok* procedure.
  getTok: (string, charcount, string, charcount, string)
        -> tok determ (i, i, i, o, o).
  myHandler: (traceId) .
```

```
clauses
```

```
scan(P,[S|SL]) = [TL|scan(P1,SL)] :- % сканирование текста
    TL=scanLine(P,S,P1),!. % сканирование одной строки
scan(_,[]) = [].
```

```
scanLine(P,S,P0) = [Tok|scanLine(P2,S2,P0)] :-
    frontToken(S,T,S1), % выделение лексемы текста
    T<> ";" , % это не начало комментария
    P1 = search(S,T), % определение позиции лексемы
    Tok=getTok(T,P+P1,S1,P2,S2),!. % построение лексемы
```

```
scanLine(P,S,_)=_-:
    fronttoken(S,T,_),T<> ";" , % если мы здесь, то ошибка в лексеме Т
    P1 = search(S,T), % определение позиции лексемы Т
    Mes = format(scanErr,T,P+P1), % формирование сообщения об ошибке
    raise_user(Mes). % вызов исключения
```

```
scanLine(P,S,P+length(S)+1) = [eol]. % сканирование строки завершено
```

```
getTok(T,P,S,P+length(T),S) = t(Token,P) :-
    (isOpCode(T,_,Token); % это код операции
     isReg(T,Token); % это регистр
     T=",",Token=comma; % это запятая
     N = tryToTerm(integer,T), % это целое положительное число
     Token=num(N)),!.
```

```
getTok("-",P,S,P+P1+1+length(T),S1) = t(num(N),P+P1) :-
    frontToken(S,T,S1), % ожидаем целое число
    P1 = search(S,T), % между минусом и числом могут быть пробелы
    T1 = concat("-",T), % соединение минуса и числа
    N = tryToTerm(integer,T1). % тест отрицательного числа
```

```
isOpCode("mov",2,mov).
isOpCode("add",2,add).
isOpCode("inc",1,inc).
isReg("eax",eax).
isReg("ebx",ebx).
```

```
myHandler(Id):
    if Des = getDescriptor_nd(Id),
        tryGetExtraInfo(Des,"UserMessage")=string(Mes) then
            write("Ошибка сканирования\n",Mes)
        else
            vpiCommonDialogs::error("Ошибка","Неизвестное исключение")
        end if.
```

```
run():-S =
@"mov eax,16 ; это пересылка
mov ebx, -1
add eax,ebx ; это сложение
```

```

inc ebx",
S1 = toLowerCase(S),
SL = split_delimiter(S1, "\n"),
try
  TL = list::appendList(scan(0,SL)),
  list::forAll(TL,{(I):-write(I),nl})      % вывод лексем
catch TraceId do
  myHandler(TraceId)
end try,
_ = readchar().
end implement main
goal
  console::run(main:::run).

```

При входном тексте:

```

mov eax,16 ; это пересылка
mov ebx, -1 ; отрицательное число
add eax,ebx ; это сложение
inc ebx

```

программа полексемно выводит результат:

```

t("mov",0)
t("eax",4)
t(", ",7)
t("16",8)
eol()
t("mov",27)
t("ebx",31)
t(", ",34)
t("-1",36)
eol()
t("add",61)
t("eax",65)
t(", ",68)
t("ebx",69)
eol()
t("inc",88)
t("ebx",92)
eol()

```

Обратите внимание, что лексема "-1" построена из знака минус и числа 1, которые были получены двумя вызовами предиката `fronttoken`.

Второй параметр в каждом терме соответствует позиции лексемы во входном тексте. Если изменить какую-нибудь лексему так, чтобы она не соответствовала языку, — например, вместо второй команды `mov` указать неправильную команду `muv`, то мы получим сообщение об ошибке примерно такого вида:

Ошибка сканирования

Не распознана лексема: `muv` в позиции: 27

Если в текст программы добавить лишнюю лексему, которая принадлежит языку, либо удалить лексему, то сканер такого рода ошибки не обнаружит. Это прерогатива синтаксического анализатора.

29.2. Синтаксический анализ

Visual Prolog в папке примеров Visual Prolog Examples содержит программу Parser Generator, которая по формальному описанию входного языка строит его сканер и парсер на языке Visual Prolog. Однако в этом разделе мы построим парсер своими руками. Действовать будем следующим образом.

При отделении очередной лексемы от списка лексем мы, на основании правил грамматики анализируемого языка, можем предполагать, какая лексема будет следующей. Если следующая лексема совпадает с одной из ожидаемых, продвигаемся по списку лексем далее. При нахождении последней лексемы ассемблерной инструкции мы строим терм этой инструкции и продолжаем анализировать список лексем до тех пор, пока он не опустеет. В случае, когда лексема не совпадает ни с одной из ожидаемых, мы фиксируем все возможные параметры ошибки (позицию ошибки, описание ошибки, ее ближайший контекст и т. п.) и принудительно вызываем исключение с помощью предиката `raise_user`.

При анализе списка лексем возможны четыре исхода анализа лексемы: лексема правильна; лексема неправильна; лексема отсутствует там, где должна быть; лексема присутствует там, где ее быть не должно. В каждом варианте обработка ситуации уникальная:

- лексема соответствует ожидаемой лексеме. Продолжаем анализ списка лексем;
- лексема не соответствует грамматике. Формируем описание той лексемы, которая ожидалась, информацию о позиции ошибки. Вызываем исключение;
- обрыв строки или файла. Формируем описание той лексемы, которая ожидалась, информацию о внезапном окончании строки и позицию. Позицию ошибочной лексемы в строке мы указать не можем ввиду ее отсутствия, но мы можем указать позицию, которая следует после последней правильной лексемы. Вызываем исключение;
- после нахождения последней лексемы ассемблерной инструкции должен следовать признак конца строки или файла, но такой признак отсутствует. Вместо него присутствует лишняя лексема(ы). Формируем информацию о лишней лексеме и ее позиции. Вызываем исключение.

На рис. 29.1 изображен граф переходов парсера (распознавающего автомата) при анализе одной ассемблерной инструкции. Сплошными линиями показаны переходы в случае правильных лексем (вариант 1). Пунктирными линиями показаны переходы к процедурам вызова исключительной ситуации (варианты 2, 3, 4).

Из графа переходов видно, что при анализе каждой лексемы инструкции надо предусмотреть варианты 1, 2 и 3. А после распознавания последней лексемы инструкции — предусмотреть вариант 4.

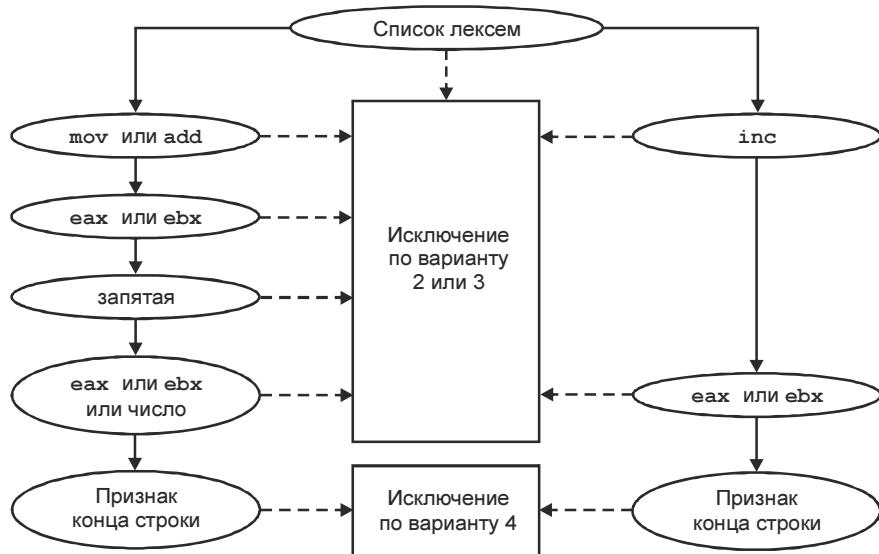


Рис. 29.1. Граф переходов при анализе одной ассемблерной инструкции

Далее представлена проверка того, что голова списка `TL` содержит operand `Oper1`, который должен являться именем регистра. Эта проверка содержит три ветви, т. к. развитие событий может идти по варианту 1, 2 или 3. Номера этих вариантов указаны в комментариях в скобках:

```

TL = [t(Oper1,P1)|TL1], (isReg(_,Oper1);           % это регистр (1)
                           ex(P1,err3));        % это не регистр (2)
TL = [eol|_],ex(P,concat(err6,err3)),            % обрыв строки (3)

```

Здесь `TL` — это список лексем, полученный от сканера. Действия по варианту 1 состоят в следующем. С помощью вызова предиката `isReg(_,Oper1)` мы проверяем, является ли лексема `Oper1` именем регистра. Если предикат `isReg(_,Oper1)` успешен, то лексема является именем регистра, вторая и третья ветви не выполняются, а парсер переходит к проверке следующих лексем.

Если же лексема `Oper1` не является именем регистра, то обработка ситуации идет по варианту 2 — вызывается предикат `ex(P1,err3)`, которому передается позиция неправильной лексемы `P1` и описание ошибки в виде строковой константы `err3`. Указанный предикат вызывает исключение, и на этом синтаксический анализ завершается. Всего в парсере определено шесть строковых констант `err`, описывающих различные ошибки, которые могут быть обнаружены. Эти строковые константы перечислены в примере 29.2.

Обработка по варианту 3 наступает в случае, когда голова списка `TL` является признаком конца строки `eol` (`end of line`), а не лексемой. Это определяется посредством унификации `TL = [eol|_]`. После чего вызывается предикат `ex(P,concat(err6,err3))`, которому передается позиция начала ассемблерной команды `P` и описание ошибки в виде соединения двух строковых констант `concat(err6,err3)`. Указанный предикат вызывает исключение, и на этом синтаксический анализ завершается.

Когда все лексемы ассемблерной инструкции правильны, то проверяется наличие признака конца строки eol:

```
TL3 = [eol|TL4];                                % инструкция распознана (1)
TL3 = [t(_,_P4)|_],ex(P4,err5)                % лишний параметр (4)
```

Если признак eol обнаружен, то считается, что инструкция распознана. Если вместо этого признака мы находим какую-либо лексему, то вызывается предикат ex(P4,err5), которому передается позиция лишней лексемы P4 и описание ошибки в виде строковой константы err5. Указанный предикат вызывает исключение, и на этом синтаксический анализ завершается.

ПРИМЕР 29.2. Здесь представлена программа синтаксического анализа входного текста, которая работает в соответствии с графом переходов, изображенным на рис. 29.1. Эта программа также содержит и сканер, поскольку без него использовать парсер нельзя, ибо входной информацией для парсера является результат работы сканера. В комментариях к программе указаны номера вариантов обработки лексем.

```
implement main
    open core,console,string,exception
constants
    scanErr = "Не распознана лексема: % в позиции: %".
    err1 = " Ожидается код операции".
    err2 = " Ожидается запятая".
    err3 = " Ожидается имя регистра".
    err4 = " Ожидается имя регистра или число".
    err5 = " Лишний параметр".
    err6 = " Слишком мало параметров".
domains
    lex = eax; ebx;                                % регистры
        mov; add; inc;                             % команды
        num(integer Number);                      % целое число
        comma.                                     % запятая
tok = t(lex Token, charcount Position); eol.
oper = reg(lex Register,charcount Position);      % регистр
        imm(integer Number,charcount Position).   % число
instr = instr1(tok,oper);                         % инструкция с одним операндом
        instr2(tok,oper,oper).                     % инструкция с двумя операндами
class facts
    isOpCode:(string Name, unsigned Arity, lex OpCode).
    isReg:(string Register, lex Register).
class predicates
    scan:(charcount,string*) -> tok** procedure.
    scanLine:(charcount,string,charcount [out]) -> tok* procedure.
    getTok:(string,charcount,string,charcount,string)
        -> tok determ (i,i,i,o,o).
    myHandler:(traceId).
```

```

parse:(tok*,instr* [out]).  

ex:(charcount,string) erroneous.  

clauses  

%***** Сканер *****  

scan(P,[S|SL]) = [TL|scan(P1,SL)] :- % сканирование текста  

    TL=scanLine(P,S,P1),!.           % сканирование одной строки  

scan(_,[]) = [].  

scanLine(P,S,P0) = [Tok|scanLine(P2,S2,P0)] :-  

    frontToken(S,T,S1),             % вычленение лексемы текста  

    T<> ";" ,                      % это не начало комментария  

    P1 = search(S,T),              % определение позиции лексемы  

    Tok=getTok(T,P+P1,S1,P2,S2),!. % построение лексемы  

scanLine(P,S,_)=:-  

    fronttoken(S,T,_),T<> ";" , % если мы здесь, то ошибка в лексеме Т  

    P1 = search(S,T),            % определение позиции лексемы Т  

    Mes = format(scanErr,T,P+P1), % формирование сообщения об ошибке  

    raise_user(Mes).            % вызов исключения  

scanLine(P,S,P+length(S)+1) = [eol]. % сканирование строки  

                                % завершено  

getTok(T,P,S,P+length(T),S) = t(Token,P) :-  

    (isOpCode(T,_ ,Token);          % это код операции  

     isReg(T,Token);               % это регистр  

     T==",",Token=comma;          % это запятая  

     N = tryToTerm(integer,T),     % это целое положительное число  

     Token=num(N)),!.  

getTok("-",P,S,P+P1+length(T),S1) = t(num(N),P+P1) :-  

    frontToken(S,T,S1),            % ожидаем целое число  

    P1 = search(S,T),             % между минусом и числом могут быть пробелы  

    T1 = concat("-",T),            % соединение минуса и числа  

    N = tryToTerm(integer,T1).    % тест отрицательного числа  

isOpCode("mov",2,mov).  

isOpCode("add",2,add).  

isOpCode("inc",1,inc).  

isReg("eax",eax).  

isReg("ebx",ebx).  

myHandler(Id):-  

    if Des = getDescriptor_nd(Id),  

        tryGetExtraInfo(Des,"UserMessage") = string(Mes) then  

            write(Mes)  

    else  

        vpiCommonDialogs::error("Ошибка","Неизвестное исключение")  

    end if.

```

```

%***** Парсер *****
parse([t(Code,P) | TL],
      [instr2(t(Code,P), reg(Oper1,P1), Oper2) | L]) :-  

  isOpCode(_,2,Code),                                % команда с двумя операндами  

  (TL= [t(Oper1,P1) | TL1],                          % анализ первого операнда  

   (isReg(_,Oper1);                                % это регистр (1)  

    ex(P1,err3));                                  % это не регистр (2)  

  TL= [eol|_],ex(P,concat(err6,err3))),            % обрыв строки (3)  

  (TL1= [t(Comma,P2) | TL2],                        % анализ запятой  

   (Comma=comma;                                    % это запятая (1)  

    ex(P2,err2));                                  % это не запятая (2)  

  TL1= [eol|_],ex(P,concat(err6,err2))),            % обрыв строки (3)  

  (TL2= [t(T,P3) | TL3],                          % анализ второго операнда  

   ((isReg(_,T),Oper2 = reg(T,P3));                % это регистр (1)  

    T=num(N),Oper2 = imm(N,P3));                  % или число (1)  

   ex(P3,err4));                                  % это не регистр и не число (2)  

  TL2= [eol|_],ex(P,concat(err6,err4))),            % обрыв строки (3)  

  (TL3 = [eol|TL4],                                % инструкция распознана (1)  

   TL3 = [t(_,P4) | _],ex(P4,err5)),!,           % лишний параметр (4)  

  parse(TL4,L).  
  

parse([t(Code,P) | TL],[instr1(t(Code,P), reg(Oper1,P1)) | L]) :-  

  isOpCode(_,1,Code),                                % команда с одним аргументом  

  (TL= [t(Oper1,P1) | TL1],                          % анализ операнда  

   (isReg(_,Oper1);                                % это регистр (1)  

    ex(P1,err3));                                  % это не регистр (2)  

  TL= [eol|_],ex(P,concat(err6,err3))),            % обрыв строки (3)  

  (TL1 = [eol|TL2],                                % команда распознана (1)  

   TL1 = [t(_,P2) | _],ex(P2,err5)),!,           % лишний параметр (4)  

  parse(TL2,L).  
  

parse([eol|TL],L):-!,parse(TL,L).                 % пустую строку пропускаем  

parse([],[]):-!.                                 % вся программа распознана  

parse([t(_,P) | _],_):-ex(P,err1).              % это не код операции  
  

ex(Pos,Err):-  

  S = format(@"%  

Позиция: ~s,~s,~s",[Err,Pos]),  

  raise_user(S).                                % формируем сообщение об ошибке  

                                                % вызываем исключение  
  

run():-S =  

@"mov eax,16 ; это пересылка  

mov ebx, -1 ; отрицательное число  

add eax, ebx ; это сложение  

inc ebx ",  

  S1 = toLowerCase(S),  

  SL = split_delimiter(S1,"`n"),

```

```

try
    TL = list::appendList(scan(0,SL)),           % вызов сканера
    parse(TL,InstrL),                          % вызов парсера
    list::forAll(InstrL,{(Instr):-write(Instr),nl})  % вывод
catch TraceId do
    myHandler(TraceId)
end try,
    _ = readchar().
end implement main
goal
    console::run(main:::run).

```

Результатом работ парсера является список инструкций, представленных в виде термов. Обратите внимание, что позиции всех лексем исходного текста программы сохранены. Они могут потребоваться как для визуализации пошагового выполнения программы, так и для позиционирования возможной ошибки во время выполнения ассемблерной программы с помощью интерпретатора:

```

instr2(t(mov(),0),reg(eax(),4),imm(16,8))
instr2(t(mov(),27),reg(ebx(),31),imm(-1,36))
instr2(t(add(),61),reg(eax(),65),reg(ebx(),70))
instr1(t(inc(),90),reg(ebx(),94))

```

Приведенные сканер и парсер, конечно, не претендуют на эффективность и полноту, но показывают один из способов вычисления и обработки различных ситуаций, возможных при анализе текста по правилам некоторой грамматики. Парсеры настоящих языков программирования гораздо сложнее и, главное, являются многоходовыми. Например, парсер языка Visual Prolog проверяет синтаксическую правильность и собирает информацию более чем за 10 проходов. На каждом проходе по исходному тексту парсер проверяет определенные элементы грамматики, собирает заданную информацию из входного текста и выявляет узкий класс ошибок. Для каждого прохода используются свои правила анализа. В результате всех проходов проверяется соответствие текста заданной грамматике и собирается вся необходимая информация для оптимизатора кода и компилятора.

В случае наличия ошибки в нашей ассемблерной программе — к примеру, в строке 3 потеряна запятая add eax ebx, парсер сформирует сообщение об ошибке:

Ожидается запятая

Позиция: 69

Это сообщение послужит программисту сигналом о том, что в позиции 69 парсер ожидает запятую. Перейти на эту позицию в редакторе исходного кода проблем не составляет, т. к. все современные редакторы имеют такую функцию.

29.3. Интерпретатор программ

Для выполнения ассемблерной программы на вход интерпретатора `interpret(L)` подается результат работы парсера `L`, который представляет собой список инструкций. Инструкции из этого списка выбираются по одной и выполняются с помощью

предиката `command`. Перед выполнением команды в факте-переменной `errDes` сохраняется информация о выполняемой команде в виде имени команды и ее позиции в тексте программы. Факт-переменная `errDes` служит для передачи информации об ошибке обработчику исключений, если такая ситуация возникнет при выполнении команды.

При выполнении команды учитывается арность. Например, инструкции с двумя операндами выполняются с помощью правила:

```
interpret([instr2(Code,Oper1,Oper2)|L]) :-
    Code = t(OpCode,Pos),
    errDes := format(€"Имя команды: %Позиция: %",
                     OpCode,Pos),          % сохранение информации о команде
    command(Code,Oper1,Oper2),      % выполнение инструкции
    printReg,! ,                  % печать регистров и флагов
    interpret(L) .
```

После выполнения каждой инструкции с помощью предиката `printReg` на экран выводится содержимое регистров и флагов. Работа интерпретатора завершается либо когда список инструкций опустеет, либо когда произойдет какая-либо ошибка во время выполнения инструкции.

В случае возникновения ошибки будет вызвано пользовательское исключение `raise_user(errDes)`, и его обработчик `myHandler` сформирует и выведет информацию об ошибке на экран. Однако `myHandler` нам надо усовершенствовать, чтобы он обрабатывал не только пользовательские исключения, но и исключения, вызванные аппаратурой процессора в случае переполнения разрядной сетки при выполнении команд `add` или `inc`. Для этого с помощью функции `tryGetExtraInfo` с параметром `"UserMessage"` надо отдельно проверить наличие пользовательской информации. Если такая информация есть, то это означает, что исключение не аппаратное, а программное, и вызвано оно программистом. Иначе — исключение является аппаратным, и при формировании информации об исключительной ситуации надо воспользоваться не только описанием аппаратного исключения, но и информацией о команде и ее позиции в тексте, которую мы специально сохраним перед выполнением каждой команды в факте-переменной `errDes`:

```
myHandler(Id) :-
    if D = getDescriptor_nd(Id) then
        if tryGetExtraInfo(D,"UserMessage") = string(Mes) then
            write(Mes)
        else
            D = descriptor(_,Excep,_,'_','_'),
            Excep = exception(_,_,Desc),           % описание исключения
            S = format("Описание ошибки: \n% \n%", Desc,errDes),
            write(S)
        end if
    else
        vpiCommonDialogs::error("Неизвестное исключение")
    end if.
```

Для моделирования регистров и флагов в программе предусмотрим соответствующие факты-переменные:

```
domains
    flag = [0..1].
class facts - flags
    zf:flag := 0.          % флаг нуля
    sf:flag := 0.          % флаг знака
class facts - regs
    eaxReg:integer := 0.    % содержимое регистра eax
    ebxReg:integer := 0.    % содержимое регистра ebx
```

Для чтения значений регистров воспользуемся функцией `get`, а для записи значений в регистры — предикатом `set`:

```
get(eax) = eaxReg :- !.           % получение значения регистра eax
get(ebx) = ebxReg :- !.           % получение значения регистра ebx
get(Reg) = _ :- errDes :=format("Ошибка чтения регистра %",Reg),
    raise_user(errDes).
set(eax,X):-eaxReg:=X,! .        % запись значения в регистр eax
set(ebx,X):-ebxReg:=X,! .        % запись значения в регистр ebx
set(Reg,_):-errDes :=format("Ошибка записи в регистр %",Reg),
    raise_user(errDes).
```

Например, выполнение команды `mov eax,ebx` заключается в пересылке содержимого регистра `ebx` в регистр `eax`. Эта команда выполняется двумя действиями. Вначале надо функцией `X=get(ebx)` получить число, хранящееся в регистре `ebx`, а потом предикатом `set(eax,X)` записать новое значение в регистр `eax`. Проще эти два действия сделать одним движением:

```
set(eax,get(ebx))
```

Выполнение команды `add eax,ebx` заключается в сложении содержимого регистров `eax` и `ebx` и записи получившейся суммы в регистр `eax`. Эти действия можно сделать также одним движением:

```
set(eax, get(eax) + get(ebx))
```

Так как команда сложения меняет флаги, то после выполнения этой команды нам надо пересчитать значения флагов:

```
zf := if get(eax)=0 then 1 else 0 end if,
sf := if get(eax)>=0 then 0 else 1 end if
```

ПРИМЕР 29.3. Здесь представлена программа, содержащая сканер, парсер и собственно интерпретатор.

```
implement main
    open core,console,string,exception
constants
    scanErr = "Не распознана лексема: % в позиции: %".
    err1 = " Ожидается код операции".
```

```

err2 = " Ожидается запятая".
err3 = " Ожидается имя регистра".
err4 = " Ожидается имя регистра или число".
err5 = " Лишний параметр".
err6 = " Слишком мало параметров".

domains
lex = eax; ebx;                                % регистры
       mov; add; inc;                            % команды
       num(integer Number);                     % целое число
       comma.                                     % запятая

tok = t(lex Token, charcount Position); eol.    % лексемы после
                                              % сканера
tok1 = t(lex Token, charcount Position).        % лексемы после
                                              % парсера
oper = reg(lex Register,charcount Position);   % регистр
       imm(integer Number,charcount Position).  % число
instr = instr1(tok1,oper);                      % инструкция с одним операндом
       instr2(tok1,oper,oper).                   % инструкция с двумя операндами

class facts
isOpCode:(string Name, unsigned Arity, lex OpCode).
isReg:(string Register, lex Register).

class predicates
scan:(charcount,string*) -> tok** procedure.
scanLine:(charcount,string,charcount [out]) -> tok* procedure.
getTok:(string,charcount,string,charcount,string)
      -> tok determ (i,i,i,o,o).
myHandler:(traceId).
parse:(tok*,instr* [out]).
ex:(charcount,string) erroneous.

clauses

%***** Сканер *****
scan(P,[S|SL]) = [TL|scan(P1,SL)] :-      % сканирование текста
  TL=scanLine(P,S,P1),!.                      % сканирование одной строки
scan([],[]) = [].

scanLine(P,S,P0) = [Tok|scanLine(P2,S2,P0)] :-
  frontToken(S,T,S1),                         % выделение лексемы текста
  T<> ";" ,                                    % это не начало комментария
  P1 = search(S,T),                           % определение позиции
                                              % лексемы
  Tok = getTok(T,P+P1,S1,P2,S2),!.          % построение лексемы
scanLine(P,S,_) = _ :-                          % если мы здесь, то ошибка
  fronttoken(S,T,_),T<> ";" ,                % в лексеме Т
  P1 = search(S,T),                           % определение позиции лексемы Т
  Mes = format(scanErr,T,P+P1),               % формирование сообщения об ошибке
  raise_user(Mes).                           % вызов исключения

```

```

scanLine(P,S,P+length(S)+1) = [eol]. % сканирование строки
                                         % завершено

getTok(T,P,S,P+length(T),S) = t(Token,P) :-  

  (isOpCode(T,_,Token);             % это код операции  

   isReg(T,Token);                 % это регистр  

   T=",", Token=comma;            % это запятая  

   N = tryToTerm(integer,T),       % это целое положительное число  

   Token = num(N)),!.  

getTok("-",P,S,P+P1+1+length(T),S1) = t(num(N),P+P1) :-  

  frontToken(S,T,S1),             % ожидаем целое число  

  P1 = search(S,T),              % между минусом и числом могут быть пробелы  

  T1 = concat("-",T),             % соединение минуса и числа  

  N = tryToTerm(integer,T1).      % тест отрицательного числа

isOpCode("mov",2,mov).  

isOpCode("add",2,add).  

isOpCode("inc",1,inc).  

isReg("eax",eax).  

isReg("ebx",ebx).

myHandler(Id):-  

  if D = getDescriptor_nd(Id) then  

    if tryGetExtraInfo(D,"UserMessage")=string(Mes) then  

      write(Mes)  

    else  

      D = descriptor(_,Excep,_,_,_,_),  

      Excep = exception(_,_,Desc),          % описание ошибки  

      S = format("Описание ошибки: \n% \n%",Desc,errDes),  

      write(S)  

    end if  

  else  

    vpiCommonDialogs::error("Неизвестное исключение")  

  end if.

***** Парсер *****  

parse([t(Code,P)|TL],  

      [instr2(t(Code,P),reg(Oper1,P1),Oper2)|L]) :-  

  isOpCode(_,2,Code),                  % команда с двумя operandами  

  (TL = [t(Oper1,P1)|TL1],           % анализ первого операнда  

   (isReg(_,Oper1);                 % это регистр (1)  

    ex(P1,err3));                  % это не регистр (2)  

   TL = [eol|_],ex(P,concat(err6,err3))), % обрыв строки (3)  

   (TL1 = [t(Comma,P2)|TL2],         % анализ запятой  

    (Comma=comma;                   % это запятая (1)  

     ex(P2,err2));                 % это не запятая (2)

```

```

TL1 = [eol|_],ex(P,concat(err6,err2))), % обрыв строки (3)
(TL2 = [t(T,P3)|TL3], % анализ второго операнда
 (isReg(_,T),Oper2 = reg(T,P3); % это регистр (1)
 T=num(N), Oper2 = imm(N,P3)); % или число (1)
 ex(P3,err4)); % это не регистр и не число (2)
 TL2= [eol|_],ex(P,concat(err6,err4))), % обрыв строки (3)
(TL3 = [eol|TL4]; % инструкция распознана (1)
 TL3 = [t(,_P4)|_],ex(P4,err5)),!, % лишний параметр (4)
parse(TL4,L).

parse([t(Code,P)|TL],[instr1(t(Code,P),reg(Oper1,P1))|L]) :- 
  isOpCode(_,1,Code), % команда с одним аргументом
  (TL = [t(Oper1,P1)|TL1], % анализ операнда
   (isReg(,_Oper1); % это регистр (1)
    ex(P1,err3)); % это не регистр (2)
   TL = [eol|_],ex(P,concat(err6,err3))), % обрыв строки (3)
   (TL1 = [eol|TL2]; % команда распознана (1)
    TL1 = [t(,_P2)|_],ex(P2,err5)),!, % лишний параметр (4)
  parse(TL2,L).

parse([eol|TL],L):-!,parse(TL,L). % пустую строку пропускаем
parse([],[]):-!. % вся программа распознана
parse([t(,_P)|_],_):-ex(P,err1). % это не код операции

ex(Pos,Err):-
  S = format(@%"%
Позиция: % ",Err,Pos), % формируем сообщение об ошибке
  raise_error(S). % вызываем исключение

%***** Интерпретатор *****
domains
flag = [0..1].
class facts - except
errDes:string := "".
class facts - flags
zf:flag := 0. % флаг нуля
sf:flag := 0. % флаг знака
class facts - regs
eaxReg:integer := 0. % начальное содержимое регистра eax
ebxReg:integer := 0. % начальное содержимое регистра ebx
class predicates
interpret:(instr*).
command:(tok1,oper,oper).
command:(tok1,oper).
get:(lex) -> integer.
set:(lex,integer).
printReg:().

```

clauses

```

interpret([instr2(Code,Oper1,Oper2) | L]) :-  

    Code = t(OpCode,Pos),  

    errDes := format(@"Имя команды: % Позиция: %", OpCode, Pos),  

    command(Code,Oper1,Oper2),           % инструкция с двумя операндами  

    printReg,! ,                      % печать регистров и флагов  

    interpret(L).  

interpret([instr1(Code,Oper) | L]) :-  

    Code = t(OpCode,Pos),  

    errDes := format(@"Имя команды: % Позиция: %", OpCode, Pos),  

    command(Code,Oper),                % инструкция с одним операндом  

    printReg,! ,                      % печать регистров и флагов  

    interpret(L).  

interpret([]).  

command(t(mov, _), reg(Reg1, _), reg(Reg2, _)) :-  

    set(Reg1, get(Reg2)), !.  

command(t(mov, _), reg(Reg, _), imm(Num, _)) :-  

    set(Reg, Num), !.  

command(t(add, _), reg(Reg1, _), reg(Reg2, _)) :-  

    set(Reg1, get(Reg1) + get(Reg2)),  

    zf := if get(Reg1)=0 then 1 else 0 end if,  

    sf := if get(Reg1)>=0 then 0 else 1 end if, !.  

command(t(add, _), reg(Reg, _), imm(Num, _)) :-  

    set(Reg, get(Reg) + Num),  

    zf := if get(Reg)=0 then 1 else 0 end if,  

    sf := if get(Reg)>=0 then 0 else 1 end if, !.  

command(t(OpCode, Pos), _, _) :-  

    errDes := format(@"Имя команды: % Позиция: %", OpCode, Pos),  

    raise_user(errDes).  

command(t(inc, _), reg(Reg, _)) :-  

    set(Reg, get(Reg) + 1),  

    zf := if get(Reg)=0 then 1 else 0 end if,  

    sf := if get(Reg)>=0 then 0 else 1 end if, !.  

command(t(OpCode, Pos), _) :-  

    errDes := format(@"Имя команды: % Позиция: %", OpCode, Pos),  

    raise_user(errDes).  

get(eax) = eaxReg :- !.          % чтение значения регистра eax  

get(ebx) = ebxReg :- !.          % чтение значения регистра ebx  

get(Reg) = _ :- errDes :=format("Ошибка чтения регистра %", Reg),  

    raise_user(errDes).  

set(eax,X) :- eaxReg:=X, !.      % запись значения в регистр eax  

set(ebx,X) :- ebxReg:=X, !.      % запись значения в регистр ebx  

set(Reg,_) :- errDes :=format("Ошибка записи в регистр %", Reg),  

    raise_user(errDes).
```

```

printReg() :-
    writef("eax=% \tebx=% \tzf=% \tsf=% \n", eaxReg, ebxReg, zf, sf).

run():-S =
@"mov eax,2147483647 ; это максимальное число
mov ebx, 1
add eax, ebx ; здесь переполнение
inc ebx",
S1 = toLowerCase(S),
SL = split_delimiter(S1, "\n"),
try
    TL = list::appendList(scan(0, SL)),           % вызов сканера
    parse(TL, InstrL),                          % вызов парсера
    interpret(InstrL)                         % вызов интерпретатора
catch TraceId do
    myHandler(TraceId)
end try,
_ = readchar().
end implement main
goal
    console::run(main:::run).

```

Входной ассемблерной программой является текст:

```

mov eax,2147483647 ; это максимальное число
mov ebx, 1
add eax, ebx ; здесь переполнение
inc ebx

```

Выполнение этой программы вызывает ошибку переполнения разрядной сетки, которая происходит из-за того, что в команде `add` с единицей складывается максимальное 32-разрядное знаковое число 2147483647. В результате получается сумма, непредставимая в 32-разрядном регистре `eax`. Интерпретатор корректно обрабатывает эту ошибку и после двух выполненных команд программы выводит сообщение об ошибке, произошедшей в третьей команде:

```

eax=2147483647 ebx=0     zf=0      sf=0
eax=2147483647 ebx=1     zf=0      sf=0

```

Описание ошибки:

Arithmetic overflow

Имя команды: `add()`

Позиция: 55

29.4. Задания для самостоятельного решения

Задача 29.1. В примере 29.2 приведен парсер, сообщающий о первой найденной ошибке. Разработайте парсер ассемблера, выводящий информацию обо всех синтаксических ошибках в программе.

Задача 29.2. Разработайте парсер ассемблера, который бы позиционировал ошибки двумя координатами: номером строки и позицией в строке.

Задача 29.3. Разработайте парсер и калькулятор для вычисления значений математических выражений.

Задача 29.4. Разработайте интерпретатор собственного простого языка программирования. В качестве базы можете выбрать тот язык программирования, который вы уже знаете.

ГЛАВА 30



Практические рекомендации

Если размер исходных данных в задаче невелик, и время решения любым способом мало, то эту главу можно смело пропустить, т. к. она посвящена рекомендациям по решению задач, имеющих большую вычислительную сложность. Мы также приведем здесь некоторые советы о том, как создавать надежные программы на Прологе вообще и Visual Prolog в частности.

30.1. Выбор способа представления и обработки данных

Правильно выбранный способ представления данных во многом определяет успех программиста при решении задачи. Алгоритм обработки данных зависит от способа представления данных и играет вторую роль в достижении цели — быстрой разработке эффективной программы. В табл. 30.1 представлены наиболее распространенные (приемлемые) для указанной задачи способы представления и обработки данных.

Таблица 30.1. Способы представления и обработки данных

Задача	Способ представления данных		Способ обработки данных
	Входные данные	Промежуточные и выходные данные	
Поиск путей, цепей, циклов на графе	Факты (граф)	Списки фактов (пути)	Рекурсия
Преобразование графов	Факты (граф)	Факты (граф)	Цикл с откатом
Операции с фактами БД	Факты (БД)	Факт(ы)	Цикл с откатом
		Списки фактов	Рекурсия
Преобразование БД	Факты (БД)	Факты (БД)	Цикл с откатом
		Списки фактов	Рекурсия
Операции с фактами-переменными: счетчиками, флагами и т. п.	Факты-переменные, изменяемые переменные	Факты-переменные, изменяемые переменные	Цикл с откатом или рекурсия

Таблица 30.1 (окончание)

Задача	Способ представления данных		Способ обработки данных
	Входные данные	Промежуточные и выходные данные	
Операции со списками	Списки	Списки, элементы списков	Рекурсия
Поиск на дереве	Дерево	Вершины, списки вершин	Рекурсия
Преобразование деревьев	Дерево	Дерево	Рекурсия
Обработка текстов	Строка	Строка, строки	Рекурсия
Операции с массивами	Массивы <code>binary</code> , <code>arrayM</code> , <code>array2M</code> и <code>arrayM_boolean</code>	Массив, элементы массива	Цикл с откатом или рекурсия

Общей чертой всех Прологов является наличие двух универсальных правил:

- списки, строки и деревья обрабатываются рекурсивно;
- факты обрабатываются в цикле с откатом.

Конечно, списки, строки и деревья тоже можно обрабатывать с помощью цикла с откатом. Но для этого надо предварительно описать недетерминированный предикат доступа к элементам списка, строки или дерева.

С другой стороны, факты можно обрабатывать рекурсивно, предварительно собрав их в список. Тогда рекурсия будет работать со списком фактов.

Как видите, содержание может меняться, но форма представления и способ обработки данных остаются сцепленными вместе навсегда. Единственным исключением из этих двух правил являются факты-переменные и двоичные данные (`binary`). Указанные сущности легко обрабатываются как рекурсивно, так и с помощью циклов с откатом.

30.2. Управление памятью в Visual Prolog 7

Visual Prolog 7 использует три вида памяти: стек вызовов (Run Stack), динамическую память (Heap) и системную память (System Memory). Каждое из этих трех хранилищ имеет свой механизм управления и преследует свои собственные цели. Стоит заметить, что стек вызовов (Run Stack) иногда называют просто *стеком*, а динамическую память (Heap) — *кучей*.

30.2.1. Стек вызовов

Стек вызовов используется для передачи предикату параметров, локальных переменных и адреса возврата при его вызове. Здесь следует иметь в виду, что в стек заносятся непосредственные значения только чисел и символов. Когда предикату

надо передать данные большого размера — например, списки, строки или термы, то в стек помещаются только указатели на эти данные. Все перечисленные сущности, размещаемые в стеке, называются *стековым фреймом*. При завершении выполнения предиката, имеющего режим, отличный от `nondeterm`, стековый фрейм из стека удаляется. Однако когда завершается выполнение `nondeterm`-предиката, имеющего активный адрес возврата, стековый фрейм не удаляется. Активный адрес возврата — это адрес правила, на которое совершится переход при откате назад. Стековый фрейм не удаляется и тогда, когда адрес возврата активен во вложенном вызове. Таким образом, даже один-единственный активный адрес возврата может держать целую цепочку стековых фреймов, препятствуя их удалению из фрейма. Если адрес возврата удаляется отсечением, то вместе с ним удаляется и вся цепочка стековых фреймов, которую он держал в стеке.

По умолчанию размер стека равен 1 Мбайт. Однако не надо полагать, что весь указанный объем памяти выделяется стеку сразу в момент запуска программы. При запуске исполнимого файла размер свободного стека составляет всего несколько килобайт. Вы это можете проверить, вызвав функцию `memory::getFreeStack()`. В ходе выполнения недетерминированных предикатов в стек помещаются новые стековые фреймы, уменьшая его свободный размер. По мере истощения стека Пролог пополняет его, черпая из системной памяти ровно по одной странице, т. е. по 4 Кбайта. При этом суммарный размер использованного стека ограничен определенной величиной, по умолчанию равной 1 Мбайт. По достижении этой величины любая попытка записи еще одного стекового фрейма вызывает исключительную ситуацию, которая называется *переполнением стека* (*Stack Overflow*).

Эффективное использование стека заключается в оптимизации как можно большего числа рекурсий программы, ибо наиболее вероятный источник истощения памяти компьютера — это использование недетерминированных рекурсивных предикатов. Когда недетерминированный предикат является потенциальной угрозой нормальному завершению программы, необходимо предпринимать какие-либо меры противодействия истощению памяти. Во-первых, надо попытаться ограничить недетерминированный перебор теми или иными эвристиками. Во-вторых, можно контролировать размер занятой части стека с помощью функции `memory::getUsedStack()`, которая возвращает размер в байтах. Указанную функцию надо вызывать в теле неоптимизированной рекурсии и предпринимать какие-либо действия, когда размер использованной части стека становится близок к 1 Мбайту.

Использовать функцию `memory::getFreeStack()` для оценки близости программы к переполнению ее стека нельзя, т. к. эта функция показывает свободный размер текущего размера стека и ничего не говорит о том, может ли стек бытьполнен из системной памяти. Поэтому для контроля стека надо оценивать разность между 1 Мбайтом и величиной использованного стека, возвращаемой функцией `memory::getUsedStack()`.

Управляемый детерминизм

Управляемый детерминизм — это способ изменения режима детерминизма предикатов в зависимости от размера использованных ресурсов. Такими ресурсами являются память и время.

Рассмотрим управляемый детерминизм на примере рекурсии. Рекурсия с управляемым детерминизмом — это рекурсия, которая самостоятельно выбирает режим детерминизма рекурсивных вызовов в зависимости от размера использованного стека. При вызове такая рекурсия, как правило, является неоптимизированной. Однако при превышении некоторой заданной границы объема использованного стека рекурсия начинает использовать оптимизированные рекурсивные вызовы, которые сохраняют стек нетронутым. Когда величина использованного стека опускается ниже заданной границы, то рекурсия возвращается к неоптимизированным рекурсивным вызовам. Такой возврат к недетерминизму происходит после откатов назад на дереве решений.

Рассмотрим пример, показывающий отличие неоптимизированной рекурсии от рекурсии с управляемым детерминизмом.

ПРИМЕР 30.1. Здесь сформулирована задача о кошельке с монетами, из которого надо набрать нужную сумму. В целях демонстрации управляемого стека немного изменим условие задачи. Монетками у нас являются случайные вещественные числа. В программе список монет мы будем генерировать в виде списка случайных чисел. Необходимо выбрать из этого списка такие элементы, сумма которых равна заданному числу s или меньше его не более, чем на величину допуска eps .

Программа создает список случайных вещественных чисел L длиной m . Этот список сортируется по убыванию для того, чтобы заданную сумму можно было собрать, начиная с самых больших чисел в списке.

Предикат `sum` в приведенной далее программе недетерминированно выбирает из списка случайных чисел элементы и суммирует их до тех пор, пока решение не окажется в границах допуска. Недетерминизм перебора заключается в том, что в предикате `sum` предусмотрено два правила выбора элемента из списка. Согласно одному правилу выбирается голова списка, являющаяся максимальным элементом списка. Другое правило пропускает эту голову, чтобы поискать подходящее число среди других элементов списка. В фактах `w` и `n` запоминается первое найденное решение: факт `w` получает список слагаемых, а факт `n` — сумму чисел этого списка. В нашем примере вывод факта `w` закомментирован, поэтому при компиляции сообщение о якобы неиспользуемом факте `w` можно игнорировать.

```

implement main
  open core, console, list, std, math
class facts
  s:real := 40.          % заданная сумма
  m:unsigned := 100.      % количество монет
  eps:real := 1E-3.       % допустимая ошибка
  w:real* := [].         % список монет
  n:real := 0.            % сумма списка монет
class predicates
  sum:(real*,real*,real) nondeterm.    % рекурсия недетерминированная
clauses
  sum(_,W,N) :- s-N<=eps, n:=N, w:=W,!..   % решение найдено

```

```

sum([X|L],W,I) :-      % берем первый элемент
    N=I+X, S>=N,        % требуемую сумму не превысили
    sum(L,[X|W],N).     % неоптимизированный вызов
sum([_|L],W,I) :-        % пропускаем первый элемент
    sum(L,W,I).          % оптимизированный вызов

run() :- L = sort([N||_ = cIterate(m), N = random()]), descending(),
        (sum(L,[],0), !,
         write("Сумма=",N), nl,
         % write("Список монет: ",W), nl,
         write(" Решение в пределах допуска"), !;
        write(" Решение не найдено")),
        _ = readchar().
end implement main
goal
    console::run(main::run).

```

Для $m=100$ монет случайного достоинства можно легко набрать нужную сумму $s=40$ с допуском $\text{eps}=1E-3$:

Сумма=39.9994276519865

Решение в пределах допуска

Однако что будет, если увеличить размерность исходных данных? Зададим в разделе `class facts` нашей программы следующие значения фактов-переменных: $s=40000$, $m=100000$, $\text{eps}=1E-7$. Запустив программу с помощью команды меню **Build | Run in Window**, мы получим сообщение: `Stack overflow`, т. е. переполнение стека:

```
=====
Dump: 2013-10-13 17:02:09
-----
Exception: systemException (runtime_exception)
System exception
error code = 1
ExtraInfo = Exception C00000FD: Stack overflow
```

Причиной этого является недетерминизм рекурсии, который на каждом витке цикла уменьшает размер свободного стека на величину стекового фрейма.

Давайте напишем рекурсию с управляемым детерминизмом. Суть такого управления заключается в следующем. Перед каждым рекурсивным вызовом будем контролировать размер использованного стека. Когда этот размер станет близок к критической величине, то рекурсивный вызов сделаем оптимизированным, чтобы не было истощения свободного стека. Иначе рекурсивный вызов делаем неоптимизированным.

Для надежной работы рекурсии разрешим использовать не более чем 1 миллион байтов. Эта критическая величина меньше 1 Мбайта ровно на 48 576 байтов. Пусть это будет нашим «неприкосновенным запасом».

Далее представлена программа, которая управляет детерминизмом рекурсии, не выпуская размер стека за пределы одного миллиона байтов:

```

implement main
    open core, console, list, std, math, memory
class facts
    s:real := 40000.          % заданная сумма
    m:unsigned := 100000.      % количество монет
    eps:real := 1E-7.          % допустимая ошибка
    w:real* := [].             % список чисел
    n:real := 0.                % сумма списка чисел
class predicates
    sum:(real*, real*, real) nondeterm.
clauses
    sum(_, W, N) :-  

        s - N <= eps, n := N, w := W, !.      % решение найдено
    sum([X|L], W, I) :-  

        sum(L, [X|W], N), !,                 % берем первый элемент
        N = I + X, s >= N,  

        (getUsedStack() > 1000000,  

         !, sum(L, [X|W], N);              % если стек почти заполнен,
         sum(L, [X|W], N)).                % то вызов оптимизированный  

    sum([_|L], W, I) :-  

        sum(L, W, I).                      % иначе — вызов неоптимизированного
                                            % пропускаем первый элемент
                                            % вызов оптимизированного

run() :- L = sort([N||_ = cIterate(m), N = random()]), descending(),
        (sum(L, [], 0), !,  

         write("Сумма=", n), nl,  

         % write("Список монет: ", w), nl,  

         write("Решение в пределах допуска"), nl,  

         write("Размер использованного стека = ", getUsedStack()), !;  

         write("Решение не найдено")),  

         _ = readchar().
end implement main
goal
    console::run(main:::run).

```

Управление детерминизмом рекурсии не замедлило сказаться на результате — стек не переполнился и допустимое решение найдено:

Сумма=39999.9999999856

Решение в пределах допуска

Размер использованного стека = 1280

Обратите внимание, что на выходе из рекурсии мы использовали отсечение. Благодаря этому мы удалили из стека все стековые фреймы, и размер использованного стека вернулся к тому состоянию, которое было до входа в рекурсию, т. е. 1280 байтов. Стек свободен, его можно использовать вновь.

Предложенная программа с управляемым детерминизмом может решать рассматриваемую задачу с исходными данными значительно большей размерности, и при этом мы будем уверены, что переполнения стека не произойдет.

Здесь стоит отметить, что управлять детерминизмом рекурсии можно не только на основе размера стека, но также и по времени выполнения рекурсии. Если отпущенное рекурсии время истекает, то рекурсивный вызов надо оптимизировать. Кроме того, ничто не мешает сделать управление гибридным — как по времени, так и на основе размера свободного стека.

Завершая обсуждение переполнения стека, оценим количество рекурсивных вызовов, превышение которого приведет к переполнению стека в рассматриваемой задаче. Для этого нам надо определить размер стекового фрейма рекурсивного правила предиката `sum/3`. С этой целью в правило останова рекурсии вставим предикат вывода размера использованного стека:

```
sum(_ ,W,N) :-  
    write("Стек: ",getUsedStack()), % вывод размера стека  
    _ = readchar(), % ждем нажатия <Enter>  
    s-N<=eps, n:=N, w:=W, !. % решение найдено
```

После запуска программы нажмем несколько раз клавишу `<Enter>` и по разности между выводимыми значениями определим размер стекового фрейма. Он равен 100 байтам. Именно на такую величину увеличивается размер использованного стека при каждом рекурсивном вызове:

```
Стек: 1396  
Стек: 1496  
Стек: 1596  
Стек: 1696  
Стек: 1796
```

Следовательно, рекурсия заполнит стек приблизительно за 9986 витков, т. к. именно столько раз надо прибавлять по 100 байтов на каждом витке, начиная со значения 1396 байтов, чтобы достичь одного миллиона байтов.

Задание произвольного размера стека вызовов

Необходимый размер стека, выделяемый программе при линковании с помощью PDC Linker, можно задать в файле проекта. Это делается в том случае, когда программиста не удовлетворяет размер стека. Файл проекта имеет расширение `vprpj` и расположен в корне дерева проекта. Его можно открыть для редактирования не только в Блокноте, но и прямо из среды Visual Prolog, используя пункт контекстного меню **Open As Text**. В конце этого файла находятся установки проекта:

```
<setting-map target="Win32">  
  <var-list>  
    <var-map name="Target.Machine" value="x86" />  
  </var-list>  
</setting-map>
```

Добавим строку для установки параметра `StackSize` и зададим ему значение, равное, например, 6 Мбайт:

```
<setting-map target="Win32">  
  <var-list>
```

```

<var-map name="Target.Machine" value="x86" />
<var-map name="Target.StackSize" value="0x600000" />
</var-list>
</setting-map>

```

Обратите внимание, что размер стека задан в шестнадцатеричной форме, что значительно проще, чем в десятичной. В десятичной системе счисления нам пришлось бы использовать величину 6 291 456 байтов, которую предварительно надо было бы вычислить.

Для того чтобы новая установка размера стека вступила в силу, надо построить проект заново с помощью команды меню **Build | ReBuild All**. И только после этого запустить проект на выполнение.

Произвольный размер стека также можно задать не всему проекту, а отдельным потокам, которые создаются и запускаются в проекте (см. главу 21).

Рассмотрим первую программу примера 30.1, в которой значения фактов-переменных имеют те самые значения: $s=40000$, $m=100000$, $\text{eps}=1E-7$, при которых происходило переполнение стека:

```

implement main
    open core, console, list, std, math
class facts
    s:real := 40000.          % заданная сумма
    m:unsigned := 100000.      % количество монет
    eps:real := 1E-7.         % допустимая ошибка
    w:real* := [].            % список монет
    n:real := 0.               % сумма списка монет
class predicates
    sum:(real*,real*,real) nondeterm.    % рекурсия недетерминированная
clauses
    sum(_,W,N) :- s-N<=eps, n:=N, w:=W,! .    % решение найдено
    sum([X|L],W,I) :-              % берем первый элемент
        N=I+X, s>=N,             % требуемую сумму не превысили
        sum(L,[X|W],N) .         % неоптимизированный вызов
    sum([_|L],W,I) :-             % пропускаем первый элемент
        sum(L,W,I) .             % оптимизированный вызов

run() :- L = sort([N||_=cIterate(m),N = random()]), descending(),
        (sum(L,[],0),!,
         write("Сумма=",n),nl,
         % write("Список монет: ",w),nl,
         write(" Решение в пределах допуска"),!;
         write(" Решение не найдено")),
         _ = readchar().
end implement main
goal
    console::run(main:::run).

```

Не забываем установить новое значение размера стека, равное 6 Мбайт. После запуска получаем:

Сумма=39999.9999999355

Решение в пределах допуска

Стек не переполнился, для этих исходных данных как раз хватило 6 Мбайт, решение найдено с помощью полного перебора. Однако здесь следует заметить, что такой способ силового решения задач подходит не всегда. Все-таки лучше находить и использовать эвристики усечения перебора.

30.2.2. Динамическая память

Если в стек помещаются данные небольшого размера — например, числа или символы, то в динамической памяти размещаются данные большого размера: списки, термы (деревья), строки и т. п. В стеке же размещаются только указатели на эти данные. Также в динамической памяти хранятся данные, которые не освобождаются откатом. К таким данным относятся факты и объекты.

Строки сохраняются в специальном разделе динамической памяти: Atomic Heap. Этот раздел динамической памяти обрабатывается очень эффективно, но на данные, хранимые в нем, накладывается одно ограничение — они не могут содержать указатели. Высокая эффективность Atomic Heap обусловлена тем, что во время сборки мусора эти данные не сканируются на предмет наличия указателей. Остальная динамическая память называется NonAtomic Heap. В ней хранятся данные, которые могут содержать указатели, поэтому раздел NonAtomic Heap сканируется на предмет наличия указателей.

Когда программист создает данные большого размера, то они размещаются в динамической памяти и удаляются только сборщиком мусора. Программного способа освобождения памяти от конкретных данных в Visual Prolog не существует. Однако при необходимости можно программно активировать сборщик мусора, вызвав предикат `memory::garbageCollect()`. С другой стороны, сборщик мусора запускается автоматически время от времени. В Visual Prolog существует три варианта захвата динамической памяти для нужд программы:

- сборщик мусора отдает программе свободный фрагмент динамической памяти, если таковой имеется;
- сборщик мусора выделяет программе фрагмент динамической памяти посредством освобождения ее от мусора;
- новая память выделяется программе из системной памяти.

ЭФФЕКТИВНОЕ ИСПОЛЬЗОВАНИЕ ДИНАМИЧЕСКОЙ ПАМЯТИ

Не бойтесь сохранять большие структуры данных в фактах! Это не замедлит вашу программу, т. к. большие данные хранятся в динамической памяти и их не надо копировать из стека в динамическую память и обратно.

Вот пример рекурсивного создания списка целых чисел, представленного в виде факта-переменной `x`:

```

facts
    x:integer* = [].

predicates
    создатьСписок: (unsigned) .

clauses
    создатьСписок(0) :- !.          % останов рекурсии
    создатьСписок(N) :-              %
        x:=[N|x],                  % модификация списка в динамической памяти
        создатьСписок(N-1).

```

При вызове предиката `создатьСписок(1000000)` в динамической памяти миллион раз будет деструктивно обновлен факт `x`, но при этом не произойдет ни одной пересылки данных между глобальным стеком, которого уже нет, начиная с версии Visual Prolog 7, и динамической памятью, а также не произойдет ни одной пересылки данных между стеком вызовов и динамической памятью. Эта программа очень эффективна и мало чем уступит аналогичной программе на C++.

Отказ от глобального стека в Visual Prolog 7

Visual Prolog 7 не имеет глобального стека (GStack) и распределяет все сущности программы непосредственно в динамической памяти (Heap), что дороже с точки зрения ресурсных затрат. Однако общий счет между совместным использованием глобального стека и динамической памяти и использованием только динамической памяти — в пользу второго варианта. Основаниями для такого утверждения служат большой размер памяти в современных компьютерах и отсутствие необходимости в обмене данными между динамической памятью и глобальным стеком.

Отказ от использования глобального стека в Visual Prolog 7 состоялся по следующим причинам:

- ограниченный размер глобального стека;
- не всегда хорошее освобождение памяти глобального стека. Глобальный стек может (и очень часто) содержать мусор, а этот мусор может ссылаться на другой мусор в глобальном стеке, что приводит к большим и необоснованным затратам памяти;
- возможность переполнения глобального стека при неумелом программировании;
- необходимость копирования данных из глобального стека в динамическую память и обратно;
- каждый поток в программе с параллельными потоками команд должен иметь свой собственный глобальный стек. При этом для каждого потока резервируется память (по умолчанию 100 Мбайт), что резко ограничивает число потоков. Это довольно дорогая плата за использование потоков. Однако при создании каждого потока размер глобального стека может быть установлен вручную.

Многопоточность

В многопоточной программе каждый поток имеет свой стек вызовов, но динамическая память одна на всю программу и разделяется всеми потоками программы. Поэтому вопросы синхронизации доступа потоков к стекам не рассматриваются. Вопросы же синхронизации доступа потоков к общей динамической памяти отданы на откуп программисту. Однако следует иметь в виду, что, кроме фактов, все данные в Visual Prolog неизменяемы. Поэтому проблем синхронизации в Visual Prolog нет, т. к. все данные, хранимые в динамической памяти, кроме фактов, имеют доступ лишь по чтению. Только одна проблема синхронизации может возникнуть — когда несколько потоков имеют одновременный доступ по записи к одним и тем же фактам в динамической памяти. Для разрешения этой проблемы служат средства (семафоры, флагки, события и т. п.), предоставляемые программисту пакетом `multithread` и рассмотренные в *главе 21*.

30.2.3. Системная память

Программист получает доступ к системной памяти, когда использует определенные средства, — например, GDI+ или COM. Полученные от этих средств данные должны быть скопированы из системной памяти в динамическую память Visual Prolog, а системная память освобождена. В каждом конкретном случае использования указанных средств получение данных и освобождение памяти от них должно происходить по соглашениям, принятым для этих средств.

часть V



Практикум по программированию

Глава 31. Введение в Visual Prolog

Глава 32. Поиск с откатом на фактах

Глава 33. Поиск с откатом на правилах

Глава 34. Рекурсивные правила

Глава 35. Рекурсивные правила на списках

Глава 36. Внутренняя база данных

Глава 37. Задачи на графах

Глава 38. Задачи на деревьях

Глава 39. Задачи на массивах

ГЛАВА 31



Введение в Visual Prolog

Для вызова среды Visual Prolog необходимо запустить файл vip.exe, находящийся в папке bin, по тому пути, куда вы установили Visual Prolog. По умолчанию Visual Prolog создает себе папку C:\Program Files (x86)\Visual Prolog 7.5 на 64-разрядной платформе MS Windows или C:\Program Files\Visual Prolog 7.5 на 32-разрядной. Также полезно на рабочем столе создать ярлычок для вызова справки C:\Program Files (x86)\Visual Prolog 7.5\doc\vip.chm.

31.1. Создание консольного проекта

Среда разработки приложений системы Visual Prolog включает текстовый редактор на основе компонента Scintilla, различные редакторы ресурсов, систему отслеживания изменений, которая обеспечивает перекомпиляцию только измененных ресурсов и модулей, ряд экспертов исходного кода, оптимизирующий компилятор, набор средств просмотра различных типов информации о проекте и, наконец, отладчик. Полученные приложения являются исполняемыми EXE-программами. В коммерческой версии Visual Prolog возможно создание библиотек DLL, а также выбор разрядности ОС Windows.

При запуске Visual Prolog открывается диалоговое окно с приглашением создать новый проект или открыть ранее созданный (рис. 31.1).

Для создания нового проекта надо воспользоваться кнопкой **New Project** и в открывшемся диалоговом окне создания проекта указать необходимые данные (рис. 31.2).

Выбор платформы с разрядностью **32bit+64bit** означает, что вы сможете выбрать любой из этих вариантов при компиляции проекта. Следует заметить, что на 32-разрядной платформе можно строить приложения для 64-разрядной платформы. Манифест-файл содержит основные данные о проекте и целевой платформе в формате XML.

При изучении языка Visual Prolog мы будем использовать только консольные приложения, чтобы код графического интерфейса не мешал изучению языка самого по себе. Поэтому для нашего проекта pr01 выберем тип проекта **Console application**.



Рис. 31.1. Диалоговое окно при запуске Visual Prolog

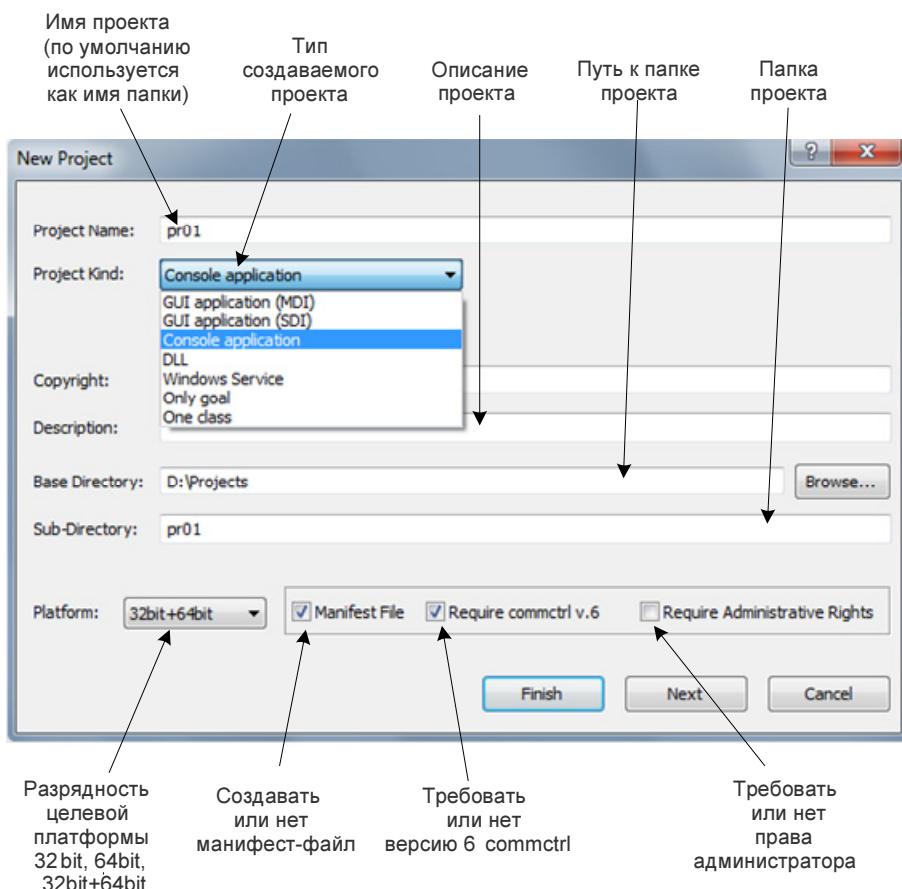


Рис. 31.2. Диалоговое окно создания нового проекта

Создание приложений с графическим интерфейсом рассматривается в главах *части VI*, размещенной в электронном архиве, сопровождающем книгу (см. *приложение 11 с описанием архива*).

После нажатия кнопки **Finish** будет сформировано консольное приложение, и мы наконец-то увидим среду разработки приложений Visual Prolog (рис. 31.3). При нажатии вместо **Finish** другой кнопки — кнопки **Next**, вам будет предложен ряд дополнительных установок для создаваемого проекта.

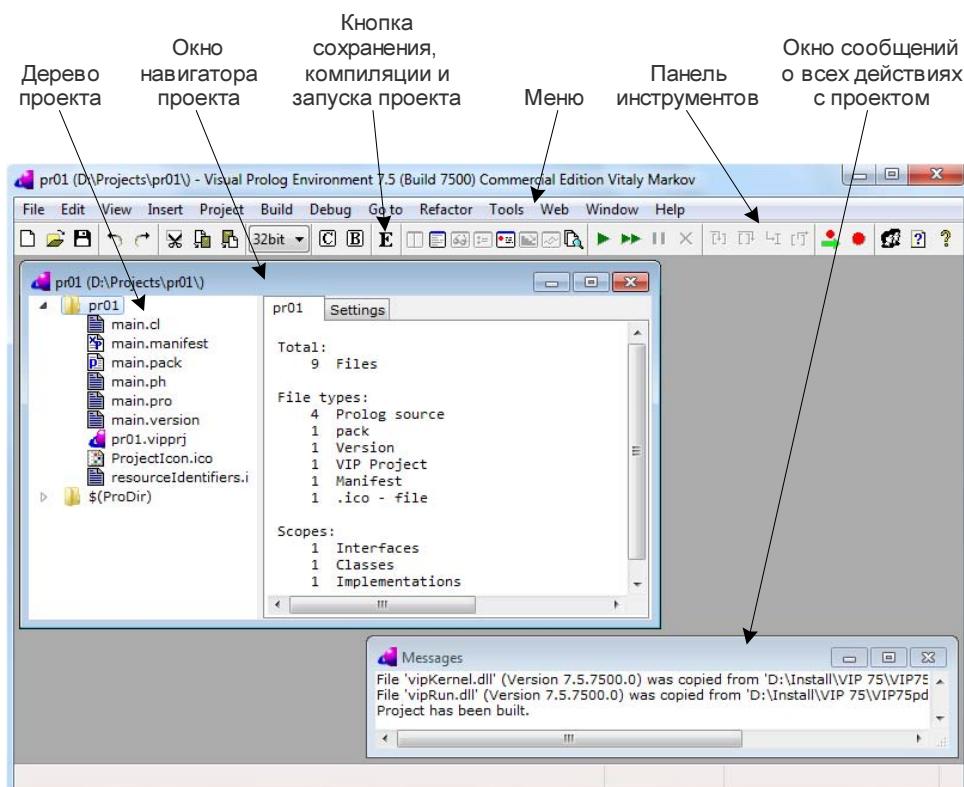


Рис. 31.3. Интерфейс Visual Prolog

Меню и панель инструментов имеют стандартные для большинства языков программирования элементы (описание меню приведено в *приложении 6*, в *приложении 7* описаны клавиши быстрого доступа к пунктам меню, а *приложение 8* содержит описание кнопок панели инструментов).

Компонентами проекта являются библиотеки, описание ресурсов (меню, диалогов, окон, форм, панелей инструментов), объектные файлы, ресурсные файлы (картинки, иконки, курсоры), пакеты.

Независимо от имени проекта основной класс проекта называется `main`. Папка `pr01` дерева проекта (см. рис. 31.3) содержит файлы, описанные в табл. 31.1.

Таблица 31.1. Описание файлов проекта

Файл проекта	Описание файла
main.cl	Декларации класса main
main.manifest	Файл-манифест проекта
main.pack	Пакет проекта
main.ph	Заголовочный файл пакета
main.pro	Реализация (имплементация) класса main
pr01.vipprj	Текстовый файл с описанием проекта
projectIcon.ico	Значок проекта
resourceIdentifiers.i	Ресурсные идентификаторы для курсоров, значков, растровых картинок и т. п.

Открыть любой файл можно двойным щелчком по нему в дереве проекта. При этом в правой панели откроется подробная информация о выбранном файле. В консольных приложениях нас будет интересовать файл main.pro. В реализации класса main, содержащемся в файле main.pro, описан предикат run, который является точкой входа в класс main. В этом классе мы будем создавать свои программы:

```
implement main
    open core
clauses
    run() :- 
        succeed(). % place your own code here
end implement main
goal
    console::runUtf8(main::run).
```

Цель программы описана в разделе goal. В нашем случае целью является предикат console::runUtf8. Он создает и запускает консоль с использованием кодовой страницы utf8, имеющей номер 65001. В этой кодовой странице корректно обрабатываются только англоязычные тексты. Для обработки русских текстов предикат runUtf8 надо заменить одним из следующих:

```
console::run(main::run). % рус Unicode
console::run(main::run, stream::unicode). % рус Unicode
console::run(main::run, stream::ansi(866)). % рус ANSI (866)
console::run8(main::run, 866). % рус ANSI (866)
```

Первые два предиката поддерживают обработку двухбайтовых символов Unicode. Два последних предиката поддерживают однобайтовую кодировку ANSI для указанной кодовой страницы 866. Вы можете указывать другие кодовые страницы. В *приложении 1* приведена подробная информация по всем предикатам run... класса console. Так как большинство программ в этой книге обрабатывают русские тексты, то в качестве целевого предиката вместо console::runUtf8(main::run) мы будем использовать console::run(main::run).

31.2. Запуск программы

Запустить на выполнение программу можно либо кнопкой **E** на панели инструментов, либо с помощью команды меню **Build** (Построить) | **Execute** (Выполнить), либо комбинацией клавиш <Ctrl>+<F5>.

Будьте внимательны

Не нажимайте по привычке клавишу <F5>, которая запускает на выполнение программы в среде многих других языков программирования. В среде Visual Prolog эта клавиша запускает программу в отладчике.

В Прологе для учебных целей (версия Visual Prolog Personal Edition) при компиляции программы вам будет предложено зарегистрировать установленный продукт (рис. 31.4). Регистрация бесплатная. Однако можно и не регистрироваться, нажав кнопку **Cancel**. В таком случае это предложение будет появляться при каждой компиляции.

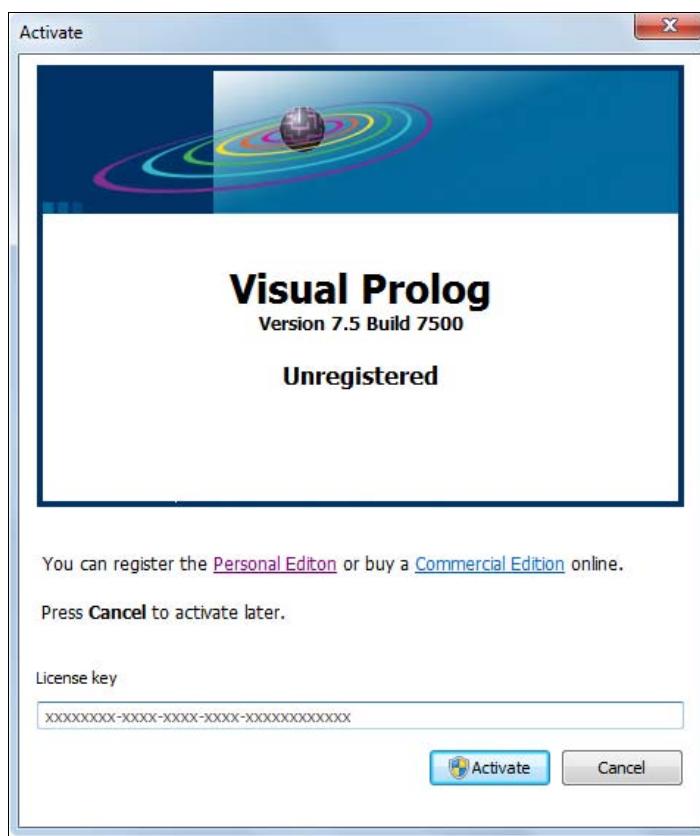


Рис. 31.4. Диалоговое окно активации Visual Prolog

Кроме того, в «персональной» версии при каждом запуске программы будет появляться предупреждение о некоммерческом использовании программы (рис. 31.5). Коммерческая версия Visual Prolog не имеет такой особенности.

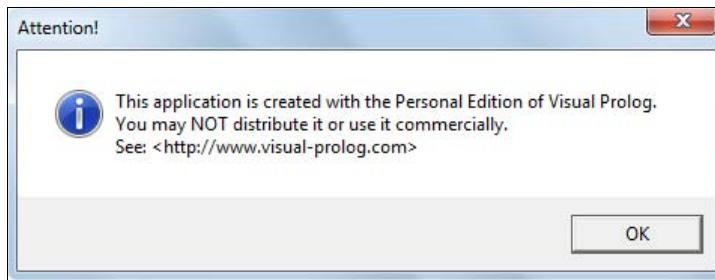


Рис. 31.5. Предупреждение о некоммерческом использовании

Поскольку наша программа пока не обладает вообще никакой функциональностью, то ее запуск приведет лишь к открытию консольного окна и его моментальному закрытию. Все это действие произойдет настолько быстро, что вы увидите лишь как что-то промелькнуло на экране.

При компиляции проекта для 32-разрядной платформы Visual Prolog создаст три папки:

- Deb — папка с информацией для отладчика;
- Exe — папка с исполнимым файлом и необходимыми библиотеками;
- Obj — папка с объектными файлами проекта.

При компиляции проекта для 64-разрядной платформы будут созданы соответственно папки Deb64, Exe64 и Obj64. В любом случае в папке Exe (Exe64) появится исполнимый файл pr01.exe и две библиотеки: vipKernel.dll и vipRun.dll. На данном этапе этих трех файлов достаточно для запуска программы, например, из Проводника.

Добавим функциональность в нашу программу. Пусть она выводит на экран сообщение Привет! ровно на одну секунду. Пауза в одну секунду осуществляется предикатом sleep из класса programControl. Время паузы указывается в миллисекундах.

Для корректного вывода русских сообщений не забудем установить целевой предикат console::run(main::run):

```
implement main
  open core
clauses
  run() :-
    console::write("Привет!"),
    programControl::sleep(1000).      % 1 сек.
end implement main
goal
  console::run(main::run).
```

При компиляции программы Visual Prolog может запросить разрешение на вставку директивы #include в заголовочный файл main.ph с целью включения нового класса в проект (рис. 31.6). Мы ответим кнопкой **Add All**.

Другой способ задержки закрытия консольного окна после выполнения программы заключается в использовании функции readchar, ожидающей нажатие клавиши

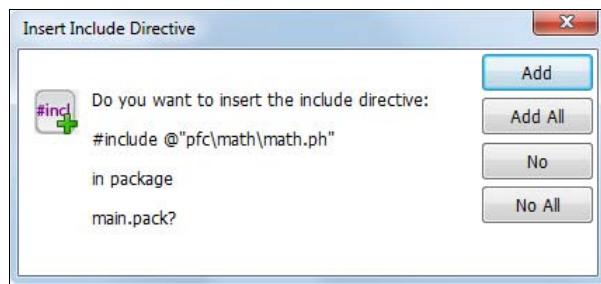


Рис. 31.6. Диалоговое окно с запросом вставки директивы #include

<Enter>. Именно этим способом мы и будем пользоваться в консольных программах:

```
implement main
    open core
clauses
run() :-
    console::write("Привет!"),
    _ = console::readchar().
end implement main
goal
    console::run(main::run).
```

В результате выполнения этой программы мы увидим на экране окно, похожее на рис. 31.7.

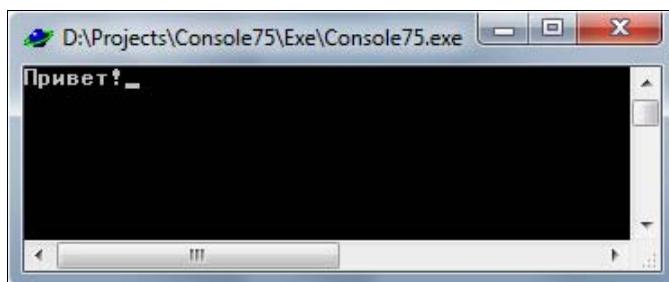


Рис. 31.7. Результат выполнения

Есть еще третий способ задержки закрытия окна — запуск приложения с помощью BAT-файла. Для этого надо воспользоваться командой меню **Build** (Построить) | **Run in Window** (Запустить в окне) или быстрыми клавишами <Alt>+<F5>. При этом в папке Exe создается файл capdos.bat, который запускает приложение и после его завершения вызывает паузу командой pause.

Неизменяемые в программе строки разумно объявлять константами. Такая организация строковых сообщений позволяет легко модифицировать как текст сообщений, так и язык сообщений:

```
implement main
    open core
```

```

constants
s = "Привет!".
clauses
run() :-
    console::write(s),
    _ = console::readchar().
end implement main
goal
    console::run(main:::run).

```

В этом примере тип константы явно не указан, т. к. он легко выводится компилятором из контекста.

31.3. Расширение области видимости

Открыв директивой `open` класс `console`, мы можем избавиться от явного указания имени класса перед предикатами:

```

implement main
    open core, console
clauses
run() :-
    write("Привет!"),
    _ = readchar().
end implement main
goal
    console::run(main:::run).

```

31.4. Управление выводом в консоли

Класс `console` описан в *приложении 1*. Кроме всего прочего, он содержит средства установки курсора в позицию с заданными координатами. Координаты задаются числом строк и столбцов. Консольное окно, как правило, содержит 25 строк и 80 столбцов. Правда, они нумеруются с нуля до 24 и 79 соответственно. Кроме того, класс `console` позволяет устанавливать цвет символов и цвет фона. Например, для вывода приветствия в середине консольного окна красными буквами по ярко-белому фону подойдет такой предикат:

```

implement main
    open core, console
clauses
run() :-
    setLocation(console_native:::coord(37, 12)),
    setTextAttribute(244),
    write("Привет!"),
    _ = readchar().
end implement main

```

```
goal  
    console::run(main::run) .
```

Курсор выставляется на строку 12 и столбец 37. Далее устанавливается атрибут 244. Он складывается из числа 4 (красный цвет символов) и числа 240 (ярко-белый цвет фона).

31.5. Использование классов PFC

Visual Prolog имеет ряд встроенных в компилятор предикатов (они описаны в главе 7). Поскольку эти предикаты встроены, то видимы везде, поэтому в тексте программы они просто предваряются двумя двоеточиями, хотя можно обойтись и без двоеточий — если вы не создали собственный предикат, одноименный встроенному. В противном случае двоеточия необходимы для устранения неоднозначного вызова.

Основная масса предикатов хранится в библиотеке модулей и классов. Эта библиотека устанавливается при инсталляции Visual Prolog и носит название Prolog Foundation Classes (PFC) — основные классы Пролога. Предикаты из этих классов следует вызывать, указывая перед ними имя класса и два двоеточия в качестве разделителя, если только этот класс не открыт директивой `open` и если отсутствует неоднозначность (`ambiguity`). Неоднозначность возникает при использовании предиката, описанного в двух открытых классах. В этом случае Visual Prolog не может определить, из какого класса вызывать предикат. Поэтому имя класса в таком случае надо указывать явно.

Описание некоторых классов приведено в приложениях книги. Для тех, кто владеет английским языком, можно воспользоваться справкой, открываемой с помощью команды меню **Help** (Помощь). Обратите внимание, что выполнив команду меню **Help** (Помощь) | **Reference Language (book)** (Справочная информация по языку), можно ознакомиться с книгой по языку Visual Prolog в формате PDF.

Кроме этого исходный код всех классов доступен для просмотра. В дереве проекта надо распахнуть папку `pfc` и выбрать класс и файл — например, `console.pro` (рис. 31.8). При этом в правой панели будет отображено дерево программных элементов файла, исходный код которых можно просмотреть, щелкнув по ним двойным щелчком.

Знание библиотеки основных классов (PFC) позволяет писать надежный и лаконичный код, сокращая при этом время разработки проекта. Поэтому стоит уделить некоторое время ознакомлению с перечнем классов PFC.

PFC организована в виде совокупности пакетов. Пакет содержит связанные по смыслу интерфейсы и классы, которые обозначены соответствующими значками (табл. 31.2).

Рассмотрим примеры использования некоторых классов.

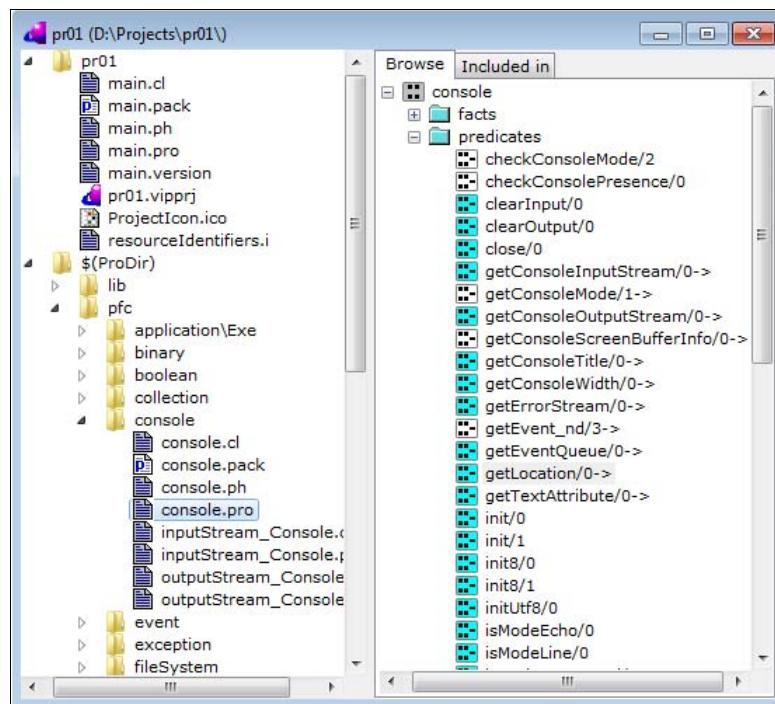


Рис. 31.8. Просмотр классов PFC

Таблица 31.2. Значки интерфейсов, классов и имплементаций

Иконка	Описание
	Интерфейс, располагаемый в файлах с расширением i
	Декларация класса, располагаемая в файлах с расширением cl. Содержит конструкторы и предикаты класса
	Имплементация класса, располагаемая в файлах с расширением pro

ПРИМЕР 31.1. Напишем программу замены в строке всех вхождений одной подстроки другой подстрокой. Исходную строку мы будем вводить с клавиатуры. Заменяемой строкой пусть является строка Prolog, а строка, которой будем заменять, — Пролог. Замена всех вхождений осуществляется предикатом `replaceAll` класса `string`.

```
implement main
    open core, console, string
clauses
run() :-
    setConsoleTitle("Замена подстрок"), % имя окна консоли
    Вход = readLine(), % чтение строки с консоли
    nl, % переход на новую строку
```

```

Вых = replaceAll(Вход, "Prolog", "Пролог"),      % замена
write(Вых),                                     % вывод результата
_ = readchar().                                % ожидание ввода для закрытия консоли
end implement main
goal
    console::run(main:::run).

```

После запуска программы ждет ввода строки с клавиатуры, после чего заменит все вхождения слова Prolog словом Пролог и выведет на экран полученный результат. На рис. 31.9 показан результат работы программы. Верхняя строка вводилась с клавиатуры. Нижняя строка получена с помощью предиката `replaceAll`.

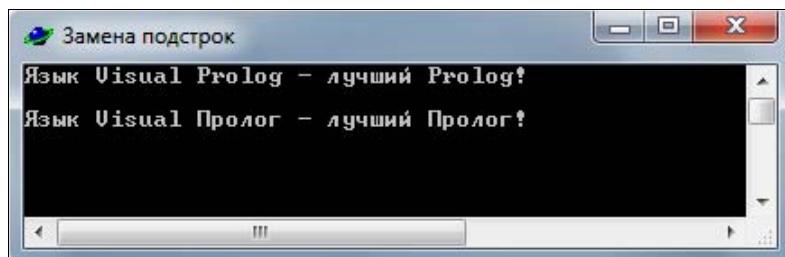


Рис. 31.9. Результат выполнения примера 31.1

Выполните по этому примеру задания 31.1 и 31.2.

Задание 31.1. Разработайте программу вывода на экран значения хэш-функции для строки, введенной с клавиатуры. Необходимый предикат:

`Значение = string::hash(String)` — вычисление значения хэш-функции.

Задание 31.2. Разработайте программу разделения строки на две равные по длине части. Необходимые предикаты:

`string::front(String, Count, First, Last)` — разделяет строку `String` на две строки `First` и `Last`, причем длина строки `First` равна `Count`.

`Длина = string::length()` — функция определения длины строки.

Для определения величины `Count` воспользуйтесь операцией целочисленного деления `Длина div 2`.

Продолжим рассмотрение примеров.

ПРИМЕР 31.2. Определим максимальное из двух чисел, вводимых с клавиатуры.

```

implement main
    open core, console
clauses
run() :-
    setConsoleTitle("Определение максимального числа"),
    write("Введите первое число: "),
    X = read(),                               % ввод X
    write("Введите второе число: "),
    Y = read(),                               % ввод Y

```

```

Max = math::max(X, Y),           % определение максимального числа
setLocation(console_native::coord(25, 10)), % столбец, строка
write("Максимальное число = ", Max),      % вывод рез-та
clearInput(),                   % очистка буфера клавиатуры
_=readchar().
end implement main
goal
    console::run(main:::run).

```

Тип данных для переменных X и Y не объявляется, т. к. Visual Prolog самостоятельно определяет тип как `integer`. Для указания другого типа — например, `real`, необходимо объявить этот тип явно хотя бы для одной переменной. Тогда Visual Prolog догадается, что пользователь будет вводить и обрабатывать вещественные числа.

```

implement main
    open core, console
clauses
run() :-
    setConsoleTitle("Определение максимального числа"),
    write("Введите первое вещественное число: "),
    hasDomain(real, X),           % объявление типа переменной X
    X = read(),                  % ввод X
    write("Введите второе вещественное число: "),
    Y = read(),                  % ввод Y
    Max = math::max(X, Y),        % определение максимального числа
    setLocation(console_native::coord(25, 10)), % столбец, строка
    write("Максимальное число = ", Max),      % вывод рез-та
    clearInput(),                 % очистка буфера клавиатуры
   _=readchar().
end implement main
goal
    console::run(main:::run).

```

Очистка буфера клавиатуры перед закрытием окна делается для корректной работы предиката ожидания нажатия клавиши `<Enter>`. Если очистку буфера не делать, то оставшиеся после ввода чисел символы в буфере клавиатуры считаются предикатом `readchar()`, и консольное окно закрывается без ожидания ввода от пользователя. Чтобы убедиться в этом, попробуйте выполнить программу из примера 31.2, предварительно закомментировав предикат очистки буфера клавиатуры.

Выполните по этому примеру задания 31.3 и 31.4.

Задание 31.3. Разработайте программу определения значения выражения $\max(A, B, C) - \min(B, C)$ от трех целых чисел A , B и C , вводимых с клавиатуры. Необходимые предикаты:

`math::max/3` — функция определения максимального из трех чисел.

`math::min/2` — функция определения минимального из двух чисел.

Задание 31.4. Разработайте программу определения суммы числа, введенного с клавиатуры со случайным числом.

Необходимый предикат:

`X = math::random(N)` — функция возвращает случайное число `X` в диапазоне $0 \leq X < N$.

Продолжим рассмотрение примеров.

ПРИМЕР 31.3. Этот пример показывает способ определения системной информации с использованием классов `systemInformation_api`, `console_native` и `registry`.

```
implement main
    open core, console
clauses
run() :-
    setConsoleTitle("Определение системной информации"),
    ComputerName = systemInformation_api::getComputerName(),
    write("\nИмя компьютера: ", ComputerName),
    UserName = systemInformation_api::getUserName(),
    write("\nИмя пользователя: ", UserName),
    CodePage = console_native::getConsoleCP(),
    write("\nКодовая страница: ", CodePage),
    Software = registry::getSubKeys(registry::currentUser(), "Software"),
    write("\n\nПрограммное обеспечение:\n", Software), nl,
    Hardware = registry::getAllValues(registry::localMachine(),
                                         "Hardware\\Description\\System"),
    write("\n\nАппаратное обеспечение:\n", Hardware),
    _ = readchar().
end implement main
goal
    console::run(main:::run).
```

ПРИМЕР 31.4. Этот пример показывает способ сохранения текста в файле в форматах ANSI и Unicode. Третий аргумент предиката `writeString` задает кодировку содержимого файла. Значение `true` означает кодировку Unicode, значение `false` — кодировку ANSI.

```
implement main
    open core, console
clauses
run() :-
    setConsoleTitle("Файловый ввод-вывод"),
    file::writeString("unicode.txt", "Тест Test", true),
    file::writeString("ansi.txt", "Тест Test", false),
    write("Готово"),
    _ = readchar().
end implement main
goal
    console::run(main:::run).
```

После завершения программы просмотрите размер каждого файла в папке `Exe` проекта и объясните их отличия.

Задание 31.5. Модифицируйте программу из примера 31.3 так, чтобы вся полученная информация сохранялась в файле в формате ANSI.

ГЛАВА 32



Поиск с откатом на фактах

Это занятие является базовым в изучении работы машины логического вывода Пролога, основанной на механизмах унификации и поиска с возвратом (см. главы 6, 7, 10).

32.1. Цвета автомобилей

Пусть имеется множество марок автомобилей {КАМАЗ, Toyota, BMW} и множество возможных цветов кузова автомобиля {зеленый, красный}. Представим эти множества в Visual Prolog и осуществим поиск по различным критериям. Опишем множество автомобилей в виде факта с именем автомобиль/1 с одним аргументом, принадлежащим домену string. Для этого в новом консольном проекте вставим в файл main.pro перед разделом clauses новый раздел с именем class facts, и объявим в нем недетерминированный факт:

```
class facts  
    автомобиль : (string) .
```

Множество автомобилей опишем в разделе clauses:

```
clauses  
    автомобиль ("КАМАЗ") .  
    автомобиль ("Toyota") .  
    автомобиль ("BMW") .
```

Аналогичным образом добавим в файл main.pro объявление и определение факта цвет/1.

Следует заметить, что предложения каждого факта и предиката необходимо группировать, т. е. располагать друг за другом и не смешивать с предложениями другого предиката. При смешивании предложений разных предикатов возникнет ошибка компиляции, и Visual Prolog предложит сгруппировать предложения каждого предиката отдельно от других.

После указанных модификаций файл main.pro должен иметь такой исходный код:

```
implement main  
open core, console
```

```

class facts
    автомобиль : (string).
    цвет : (string).
clauses
    автомобиль ("КАМАЗ").
    автомобиль ("Toyota").
    автомобиль ("BMW").
    цвет ("зеленый").
    цвет ("красный").
run() :-
    succeed().
end implement main
goal
    console::run(main:::run).

```

Сейчас наша программа содержит маленькую базу фактов, но пока не готова отвечать на запросы, т. к. они не описаны в теле предиката `run()`. Рассмотрим некоторые запросы прямо сейчас.

ПРИМЕР 32.1. Осуществим поиск заданного автомобиля в базе фактов. Если автомобиль есть, то выведем соответствующее сообщение, иначе — напишем, что такого автомобиля нет. Таким образом, предикат `run()` будет содержать два предложения:

```

implement main
    open core, console
class facts
    автомобиль : (string).
    цвет : (string).
clauses
    автомобиль ("КАМАЗ").
    автомобиль ("Toyota").
    автомобиль ("BMW").
    цвет ("зеленый").
    цвет ("красный").
run() :-
    автомобиль ("Toyota"), !,
    write("Есть такой автомобиль"),
    _=readchar();
    write("Нет такого автомобиля"),
    _ = readchar().
end implement main
goal
    console::run(main:::run).

```

Запустите программу и исследуйте ее поведение, указывая различные марки автомобиля, даже отсутствующие в программе.

Всплывающее окно предупреждений о неиспользуемом факте `цвет/1` можно игнорировать. Действительно, факт `цвет/1` в программе объявлен и описан двумя пред-

ложениями, но в запросе не упомянут. Visual Prolog считает этот факт лишним, о чем нас и предупреждает. Однако цвет/1 нам потребуется для следующих примеров.

ПРИМЕР 32.2. Для нахождения всех автомобилей с помощью программы из примера 32.1 предикат run должен быть таким:

```
run() :-  
    автомобиль (Х),  
    write("Найден автомобиль: ",Х), nl,  
    fail;  
    write("Конец поиска"),  
    _ = readchar().
```

Запустите программу и исследуйте ее поведение.

Обратите внимание, что для получения всех решений мы убрали отсечение и вставили предикат fail, который принудительно вызывает откат назад.

ПРИМЕР 32.3. Для нахождения всех комбинаций автомобиль-цвет с помощью программы из примера 32.1 предикат run должен быть таким:

```
run() :-  
    автомобиль (Авто), цвет (Цвет),  
    write(Авто, " - ", Цвет), nl,  
    fail;  
    write("Конец перебора"),  
    _ = readchar().
```

Запуск программы приводит к такому решению:

```
КАМАЗ - зеленый  
КАМАЗ - красный  
Toyota - зеленый  
Toyota - красный  
BMW - зеленый  
BMW - красный  
Конец перебора
```

Обратите внимание, что вызов двух недетерминированных предикатов:

```
автомобиль (Авто), цвет (Цвет), fail
```

осуществляет с откатом перебор всех цветов для каждого автомобиля. Это объясняется тем, что с помощью отката Пролог будет перебирать все цвета до тех пор, пока они не закончатся. И только после этого предикат цвет (Цвет) завершится неудачей, что вызовет откат назад, и Пролог найдет новый автомобиль, для которого перебор всех цветов повторится заново. Получается своеобразный цикл в цикле.

Задание 32.1. Найдите все комбинации цвет-автомобиль.

Задание 32.2. Найдите все комбинации автомобиль-автомобиль, в каждой из которых автомобили не должны быть одинаковыми.

32.2. Точки на плоскости

ПРИМЕЧАНИЕ

Все примеры и задачи этого учебного вопроса можно выполнять в новом консольном приложении, однако можно использовать проект, который мы создали при изучении первого учебного вопроса, заменив в нем ненужные факты и цель на новые.

Пусть на плоскости в декартовой системе координат дано множество точек с известными координатами. Необходимо описать эти точки в Visual Prolog и осуществить поиск точек по различным критериям.

Опишем точки в виде фактов `point/2` от двух аргументов — координат. Для этого в разделе `facts` объявим недетерминированный факт:

```
class facts
    point : (integer X, integer Y).
```

Здесь следует иметь в виду, что `X`, `Y` — это не имена переменных, которыми мы будем в программе обозначать координаты точек, а комментарии к аргументам с доменами `integer`. Имена же переменных могут быть любыми, в том числе и `X`, `Y`.

Множество точек зададим в разделе `clauses`. Пусть это множество содержит девять точек:

```
clauses
    point(1,1). point(1,2). point(1,4).
    point(2,1). point(2,2). point(2,4).
    point(3,2). point(4,1). point(4,3).
```

Помните, что предложения можно записывать в одной строке друг за другом? Однако следует знать, что Пролог в ходе вычислений просматривает эти предикаты слева направо и сверху вниз. То есть если в одной строке записано несколько предложений, то просмотр этих предложений будет идти слева направо до последнего предложения в строке. Потом Пролог перейдет к перебору предложений, указанных в следующей строке. Изменить этот порядок просмотра предикатов в Прологе нельзя. Можно только поменять порядок предложений в программе.

Итак, исходный код в файле `main.pro` после указанных модификаций должен быть таким:

```
implement main
    open core, console
class facts
    point : (integer X, integer Y).
clauses
    point(1,1). point(1,2). point(1,4).
    point(2,1). point(2,2). point(2,4).
    point(3,2). point(4,1). point(4,3).
run() :-
    succeed().
end implement main
```

```
goal
    console::run(main:::run) .
```

ПРИМЕР 32.4. Осуществим поиск точки с заданными координатами — например: $X=1$ и $Y=2$. Для этого вставим вызов предиката `point(1,2)` в тело цели и выведем сообщение о найденной точке. Для случая отсутствия искомой точки не забудем предусмотреть второе предложение с выводом соответствующего сообщения.

```
implement main
    open core, console
class facts
    point : (integer X, integer Y) .
clauses
    point(1,1). point(1,2). point(1,4) .
    point(2,1). point(2,2). point(2,4) .
    point(3,2). point(4,1). point(4,3) .
run() :-
    point(1,2), !,
    write("Точка с указанными координатами найдена"),
    _=readchar();
    write("Точка с указанными координатами НЕ найдена"),
    _ = readchar().
end implement main
goal
    console::run(main:::run) .
```

При запуске программы Пролог начнет выполнять предикат `run()`. Так как этот предикат имеет две альтернативные ветви (два предложения), то машина логического вывода войдет в первую ветвь (первое предложение), запомнив в стеке адрес входа во вторую ветвь (второе предложение), с которого она продолжит поиск решения в случае отката в первой ветви.

В первом предложении машина логического вывода Пролога сама будет просматривать предикаты `point/2` до тех пор, пока не встретит предикат с искомыми аргументами $X=1$ и $Y=2$. В этом случае она также запомнит в стеке адрес возврата, с которого она впоследствии сможет продолжить поиск новых решений при откате назад. После этого выведет сообщение о найденном решении и, встретив знак отсечения, удалит все адреса возврата из стека, которые были туда помещены в ходе выполнения предиката `run()`, а их было всего два.

Если же точка с указанными координатами не будет найдена, то предикат `point/2` завершится неудачей, что вызовет откат из первой ветви во вторую — благо адрес возврата в вершине стека как раз указывает на начало второй ветви. Вторая ветвь выведет на экран сообщение о том, что точка не найдена.

ПРИМЕР 32.5. Осуществим поиск точки с координатами, вводимыми с клавиатуры. Для этого используем такой запрос:

```
implement main
    open core, console
```

```

class facts
    point : (integer X, integer Y).
clauses
    point(1,1). point(1,2). point(1,4).
    point(2,1). point(2,2). point(2,4).
    point(3,2). point(4,1). point(4,3).
run() :-
    write("X = "), X=read(),
    write("Y = "), Y=read(),
    clearInput(),
    point(X,Y), !,
    write("Точка с указанными координатами найдена"),
    _=readchar();
    write("Точка с указанными координатами НЕ найдена"),
    _=readchar().
end implement main
goal
    console:::run(main:::run).

```

ПРИМЕР 32.6. Осуществим поиск одной точки с заданной координатой X , и в случае успешного нахождения выведем координату Y :

```

implement main
    open core, console
class facts
    point : (integer X, integer Y).
clauses
    point(1,1). point(1,2). point(1,4).
    point(2,1). point(2,2). point(2,4).
    point(3,2). point(4,1). point(4,3).
run() :-
    write("X = "), X=read(),
    clearInput(),
    point(X,Y), !,
    writeln("Найдена точка с координатами %,%", X, Y),
    _=readchar();
    write("Точка НЕ найдена"),
    _=readchar().
end implement main
goal
    console:::run(main:::run).

```

Обратите внимание, что в программе есть несколько точек с координатой, например, $X=1$. Как найти все эти точки? В Прологе это делается очень просто. В предложении, где осуществляется поиск, надо убрать отсечение и принудительно вызвать откат с помощью предиката `fail()`.

ПРИМЕР 32.7. Осуществим поиск всех точек с заданной координатой X , и в случае успешного нахождения определим координату Y каждой из них:

```

implement main
    open core, console
class facts
    point : (integer X, integer Y).
clauses
    point(1,1). point(1,2). point(1,4).
    point(2,1). point(2,2). point(2,4).
    point(3,2). point(4,1). point(4,3).
run() :-
    write("X = "), X=read(),
    clearInput(),
    point(X,Y),
    writeln("Найдена точка с координатами %,% \n",X,Y),
    fail();
    write("Bce!!!"),
    _=readchar().
end implement main
goal
    console::run(main::run).

```

Предикат `writeln` осуществляет вывод форматированной строки. Вместо знаков процента последовательно подставляются значения переменных, указанных через запятую правее форматированной строки. Знак `\n` является символом перевода на новую строку. Если этот знак не указывать, то весь вывод будет осуществляться в одну строку.

ПРИМЕР 32.8. Осуществим поиск всех точек, находящихся выше горизонтальной прямой $Y=2$:

```

implement main
    open core, console
class facts
    point : (integer X, integer Y).
clauses
    point(1,1). point(1,2). point(1,4).
    point(2,1). point(2,2). point(2,4).
    point(3,2). point(4,1). point(4,3).
run() :-
    point(X,Y), Y>2,
    writeln("Найдена точка с координатами %,% \n",X,Y),
    fail();
    write("Bce!!!"),
    _=readchar().
end implement main
goal
    console::run(main::run).

```

ПРИМЕР 32.9. Осуществим поиск всех точек, принадлежащих прямой $y = x$:

```

implement main
    open core, console

```

```
class facts
    point : (integer X, integer Y) .
clauses
    point(1,1) . point(1,2) . point(1,4) .
    point(2,1) . point(2,2) . point(2,4) .
    point(3,2) . point(4,1) . point(4,3) .
run() :-
    point(X,X),
    writef("Найдена точка с координатами %,% \n",X, X),
    fail();
    write("Все!!!"),
    _=readchar().
end implement main
goal
    console:::run(main:::run) .
```

Обратите внимание, что в этом примере вместо вызова `point(X,Y)`, $X=Y$ мы использовали предикат `point(X,X)`, который сам выполнит проверку равенства координат.

Задание 32.3. Вывести на экран координаты всех точек.

Задание 32.4. Вывести все точки, сумма координат которых равна $x + y = 5$.

Задание 32.5. Вывести все точки, лежащие вне прямоугольника, ограниченного координатами $1 \leq x < 3$, $2 < y \leq 4$.

Задание 32.6. Вывести все точки, имеющие четную абсциссу и нечетную ординату. Подсказка: операция определения остатка целочисленного деления четного числа на 2 возвращает 0, а нечетного — 1. Эта операция обозначается `mod`.

Задание 32.7. Вывести все точки, находящиеся ниже линии $y = 0.6 \cdot x + 0.4$.

Задание 32.8. Вывести все точки, находящиеся выше линии $y = k \cdot x + b$. Значения k и b вводить с клавиатуры.

Задание 32.9. Вывести все точки, принадлежащие квадрату с центром в точке x, y и стороной $a = 2$. Координаты x, y вводить с клавиатуры.

Задание 32.10. Вывести все точки, принадлежащие кругу с центром в точке $(2,2)$ и радиусом $R = 1.8$. Подсказка: для вычисления квадратного корня, квадратов и абсолютного значения используйте библиотеку `math`.

Задание 32.11. Вывести все точки, лежащие вне круга с центром в точке x, y и радиусом R . Значения x, y и R вводить с клавиатуры.

Задание 32.12. Вывести все пары точек, симметричных относительно прямой $y = x$.

32.3. Двенадцать месяцев

Задание 32.13. Объявить недетерминированный факт:

```
месяц(string Имя_Месяца,  
       unsigned Номер_по_порядку,  
       unsigned Количество_дней).
```

Определить в программе все 12 месяцев года. Вывести на экран в одну строку через пробел имена всех четных месяцев и их продолжительность.

Задание 32.14. Вывести имена месяцев с продолжительностью 31 день.

Задание 32.15. Вывести имена месяцев, содержащие не менее пяти символов. Подсказка: для определения длины строки воспользуйтесь предикатом: `string::length/1`.

СПРАВКА

Предикаты класса `string` описаны в *приложении 10*.

Задание 32.16. Вывести суммарную продолжительность летних месяцев.

Задание 32.17. Вывести номера тех месяцев, которые заканчиваются на мягкий знак. Подсказка: для определения наличия мягкого знака в конце строки воспользуйтесь предикатом `string::hasSuffix/2` или функцией `string::lastChar/1` (на ваше усмотрение).

Задание 32.18. Вывести имена месяцев, в имени которых есть буква 'р'. Подсказка: для определения наличия этой буквы воспользуйтесь функцией `string::searchChar/2`.

32.4. Зеленые и красные отсечения

Завершая опыты с механизмом поиска с возвратом, следует иметь в виду, что использовать отсечения и, тем более, удалять их надо осмотрительно. Если вы неосмотрительно поставили отсечение там, где оно не нужно, то самым худшим результатом работы вашей программы будет потеря решений, которые вы отсекли. Если же вы не поставите отсечение там, где оно на самом деле должно быть, то результатом будет ухудшение эффективности программы или даже получение неправильных решений, что гораздо хуже, чем пропуск решений.

Задание 32.19. Написать цель для определения того, содержит ли введенная с клавиатуры строка только строчные буквы. Использовать красное отсечение. Подсказка: предикаты, определяющие регистр символов: `string::isLowerCase(string)` и `string::isUpperCase(string)`. Указанные предикаты истинны, если соответствующее называнию предиката условие истинно. В противном случае они ложны.

Задание 32.20. Написать цель для определения одного из трех вариантов: содержит ли введенная с клавиатуры строка одни лишь буквы, одни лишь десятичные цифры,

или она содержит произвольные символы. Подсказка: воспользуйтесь предикатами `string::hasAlpha(string)` и `string::hasDecimalDigits(string)`. Указанные предикаты истинны, если соответствующее называнию предиката условие истинно. В противном случае они ложны.

Задание 32.21. Написать программу нахождения корней квадратного уравнения, коэффициенты a , b , c которого вводятся с клавиатуры. Программа должна иметь три предложения: для дискриминанта большего нуля, равного нулю и меньшего нуля. В последнем случае выводить решение в виде комплексных чисел.

ГЛАВА 33



Поиск с откатом на правилах

В главе 32 мы использовали только базу фактов и запросы к ней в предикате `run()`. Здесь мы самостоятельно опишем собственные предикаты и научимся их правильно использовать.

33.1. Родственные отношения

Примеры с родственными отношениями являются классикой освоения правил на Прологе. За неимением лучших примеров будем придерживаться упомянутых. Введем новый домен, определяющий пол человека:

```
domains  
    пол = муж; жен.
```

Выразим данные о родственных отношениях недетерминированными фактами: `родитель/2`, `чел/2`.

```
class facts  
    чел : (string Человек, пол Пол).  
    родитель : (string Родитель, string Ребенок).
```

В разделе `clauses` опишем небольшую группу людей, объединенных отношениями `родитель/2`, пол которых характеризуется отношением `чел/2`:

```
clauses  
    чел("Вася", муж).  
    чел("Рома", муж).  
    чел("Гена", муж).  
    чел("Дима", муж).  
    чел("Коля", муж).  
    чел("Лена", жен).  
    чел("Юля", жен).  
    чел("Клава", жен).  
    чел("Уля", жен).  
    чел("Валя", жен).
```

```

родитель ("Вася", "Рома") .
родитель ("Вася", "Дима") .
родитель ("Рома", "Лена") .
родитель ("Гена", "Юля") .
родитель ("Клава", "Рома") .
родитель ("Клава", "Дима") .
родитель ("Лена", "Коля") .
родитель ("Уля", "Гена") .
родитель ("Валя", "Лена") .

```

Здесь стоило бы уточнить, как различать людей в этой базе фактов. Вася в первом предложении предиката родитель и Вася во втором его предложении — это один и тот же человек. Если нам надо описать отношения, в которых есть два разных человека, имеющих одинаковое имя, то следует ввести явные синтаксические отличия в их имена, — например: Вася1 и Вася2.

Предложения можно располагать в одной строке, чтобы одним взглядом охватить больше сущностей программы:

clauses

```

чел ("Вася", муж) .     чел ("Рома", муж) .     чел ("Гена", муж) .
чел ("Дима", муж) .     чел ("Коля", муж) .     чел ("Лена", жен) .
чел ("Юля", жен) .     чел ("Клава", жен) .     чел ("Уля", жен) .
чел ("Валя", жен) .

родитель ("Вася", "Рома") .     родитель ("Вася", "Дима") .
родитель ("Рома", "Лена") .     родитель ("Гена", "Юля") .
родитель ("Клава", "Рома") .     родитель ("Клава", "Дима") .
родитель ("Лена", "Коля") .     родитель ("Уля", "Гена") .
родитель ("Валя", "Лена") .

```

При такой форме записи просмотр предложений Прологом осуществляется слева направо и сверху вниз.

На данном этапе содержимое файла main.pro должно быть таким:

```

implement main
  open core, console
domains
  пол = муж; жен.
class facts
  чел: (string Человек, пол Пол) .
  родитель: (string Родитель, string Ребенок) .
clauses
  чел ("Вася", муж) .     чел ("Рома", муж) .     чел ("Гена", муж) .
  чел ("Дима", муж) .     чел ("Коля", муж) .     чел ("Лена", жен) .
  чел ("Юля", жен) .     чел ("Клава", жен) .     чел ("Уля", жен) .
  чел ("Валя", жен) .

  родитель ("Вася", "Рома") .     родитель ("Вася", "Дима") .
  родитель ("Рома", "Лена") .     родитель ("Гена", "Юля") .
  родитель ("Клава", "Рома") .     родитель ("Клава", "Дима") .

```

```

родитель ("Лена", "Коля") .      родитель ("Уля", "Гена") .
родитель ("Валя", "Лена") .

run() :-
    succeed().
end implement main
goal
    console::run(main:::run).

```

Теперь все готово для определения правил. Правила объявляются в разделе `class predicates` с указанием режима детерминизма и потока параметров.

ПРИМЕР 33.1. Определим правило отец/2 как родителя мужского пола.

```

implement main
    open core, console
domains
    пол = муж; жен.
class facts
    родитель:(string Родитель, string Ребенок) .
    чел:(string Человек, пол Пол) .
class predicates
    отец:(string Отец, string Ребенок) nondeterm (o,o) .
clauses
    отец(Родитель, Ребенок) :- родитель(Родитель, Ребенок),
                                чел(Родитель, муж).

чел("Вася", муж) .     чел("Рома", муж) .     чел("Гена", муж) .
чел("Дима", муж) .     чел("Коля", муж) .     чел("Лена", жен) .
чел("Юля", жен) .     чел("Клава", жен) .     чел("Уля", жен) .
чел("Валя", жен) .

родитель ("Вася", "Рома") .     родитель ("Вася", "Дима") .
родитель ("Рома", "Лена") .     родитель ("Гена", "Юля") .
родитель ("Клава", "Рома") .     родитель ("Клава", "Дима") .
родитель ("Лена", "Коля") .     родитель ("Уля", "Гена") .
родитель ("Валя", "Лена") .

run() :-
    отец(Отец, Ребенок),
    write(Отец, " - ", Ребенок), nl,
    fail;
    write("Конец перебора"),
    _ = readchar().
end implement main
goal
    console::run(main:::run).

```

Задание 33.1. Определите в программе новое правило мать/2 и найдите всех матерей.

ПРИМЕР 33.2. Определим правило брат/2 и найдем всех братьев в базе фактов.

```

implement main
    open core, console
domains
    пол = муж; жен.
class facts
    родитель: (string Родитель, string Ребенок) .
    чел: (string Человек, пол Пол) .
class predicates
    брат: (string Брат, string Человек) nondeterm (o,o) .
clauses
    брат(Брат,Человек) :-  

        чел(Брат, муж),  

        родитель(Родитель, Человек),  

        родитель(Родитель, Брат),  

        not(Человек=Брат).  

    чел("Вася",муж) .     чел ("Рома",муж) .      чел ("Гена",муж) .  

    чел ("Дима",муж) .    чел ("Коля",муж) .      чел ("Лена",жен) .  

    чел ("Юля",жен) .    чел ("Клава",жен) .      чел ("Уля",жен) .  

    чел ("Валя",жен) .  

    родитель ("Вася","Рома") .   родитель ("Вася","Дима") .  

    родитель ("Рома","Лена") .   родитель ("Гена","Юля") .  

    родитель ("Клава","Рома") .  родитель ("Клава","Дима") .  

    родитель ("Лена","Коля") .   родитель ("Уля","Гена") .  

    родитель ("Валя","Лена") .  

run() :-  

    брат(Брат,Человек),  

    write(Брат," - ",Человек), nl,  

    fail;  

    write("Конец перебора"),  

    _ = readchar().  

end implement main
goal
    console::run(main::run) .

```

Результат работы будет содержать дубликаты:

Рома - Дима

Рома - Дима

Дима - Рома

Дима - Рома

Конец перебора

Это связано с тем, что для Димы вначале нашелся брат Рома по отцу, а потом по маме. После этого для Ромы нашелся брат Дима вначале по отцу, потом по маме. Чтобы исключить повторы решений, нам надо ввести новое правило один_предок/2,

которое бы выявляло наличие у двух разных людей хотя бы одного родителя. Если таковой находится, то поиск второго родителя отсекается. Иначе — ищется второй родитель:

```

implement main
    open core, console
domains
    пол = муж, жен.
class facts
    родитель: (string Родитель, string Ребенок) .
    чел: (string Человек, пол Пол) .
class predicates
    брат: (string Брат, string Человек) nondeterm (o,o).
    один_предок: (string Брат, string Человек) determ (i,o).
clauses
    брат(Брат,Человек) :-  

        чел(Брат, муж),  

        один_предок(Брат,Человек) .  

    один_предок(Человек,Брат) :-  

        родитель(Родитель, Человек),  

        родитель(Родитель, Брат),  

        not(Человек=Брат), !.  

    чел("Вася",муж) .      чел ("Рома",муж) .      чел ("Гена",муж) .  

    чел ("Дима",муж) .     чел ("Коля",муж) .     чел ("Лена",жен) .  

    чел ("Юля",жен) .      чел ("Клава",жен) .     чел ("Уля",жен) .  

    чел ("Валя",жен) .  

    родитель ("Вася", "Рома") .      родитель ("Вася", "Дима") .  

    родитель ("Рома", "Лена") .      родитель ("Гена", "Юля") .  

    родитель ("Клава", "Рома") .     родитель ("Клава", "Дима") .  

    родитель ("Лена", "Коля") .      родитель ("Уля", "Гена") .  

    родитель ("Валя", "Лена") .  

run() :-  

    брат(Брат,Человек),  

    write(Брат," - ",Человек), nl,  

    fail;  

    write("Конец перебора"),  

    _ = readchar().  

end implement main
goal
    console::run(main::run) .

```

Теперь программа выведет на экран решения без повторов:

Рома - Дима

Дима - Рома

Конец перебора

Задание 33.2. Определите, зачем использован предикат `not (Человек=Брат)`. Для этого удалите из тела правила этот предикат и проанализируйте результаты выполнения программы.

ЗАМЕТКА

Вместо предиката `not (Человек=Брат)` можно использовать неравенство `Человек<>Брат`.

ПРИМЕР 33.3. Определим правило `дедушка/2` на основе фактов из примера 33.2:

```
class predicates
    дедушка : (string Дедушка, string Чадо) nondeterm anyflow.
clauses
дедушка (Дедушка, Чадо) :-  
    чел (Дедушка, муж),  
    родитель (Дедушка, Родитель),  
    родитель (Родитель, Чадо).
```

Обратите внимание на последовательность предикатов в теле правила. Она является оптимальной. Предикат `чел(Дедушка, муж)` на ранней стадии отсекает от дерева решений ненужные решения.

ЗАМЕТКА

Поток параметров `anyflow` используется в случае, когда в программе прописаны все возможные потоки параметров или когда поток параметров изменяется в зависимости от вызова предиката.

Задание 33.3. Определите в программе новые правила: `сестра/2`, `бабушка/2`, `дядя/2`, `тетя/2`, `двоюродный_брать/2`, `двоюродная_сестра/2`.

Задание 33.4. Определите в программе новые правила: `тесть/2`, `теща/2`, `зять/2`, `сноха/2`.

ПРИМЕР 33.4. Определим правило `сын/1` на основе фактов из примера 33.2:

```
class predicates
    сын: (string Сын) nondeterm anyflow.
clauses
сын (Сын) :- чел (Сын, муж), родитель (_, Сын).
```

При вызове предиката `родитель/2` использована анонимная переменная.

Задание 33.5. Определите в программе новые правила: `дочь/1`, `внук/1`, `внучка/1`, `дед/1`.

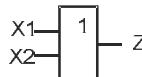
Задание повышенной трудности 33.1. Определите в программе новые правила: `отчим/2`, `мачеха/2`, `сводный_брать/2`, `сводная_сестра/2`.

33.2. Моделирование комбинационных схем

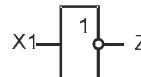
Моделирование электронных схем начинается с определения функциональных элементов и правил их работы. Пусть функциональными элементами являются двухходовые логические элементы И, ИЛИ и инвертор НЕ (рис. 33.1).



а) Элемент И



б) Элемент ИЛИ



в) Элемент НЕ

Рис. 33.1. Обозначения логических элементов

Так как эти элементы работают в двоичной логике, введем новый домен, содержащий только два элемента: ноль и единицу:

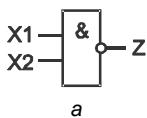
```
domains
  bool = [0..1].
```

Опишем логику работы элементов в виде совокупности фактов, содержащих комбинацию входных сигналов In1 и In2 и соответствующий этой комбинации выходной сигнал Out:

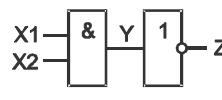
```
class facts
  и: (bool In1, bool In2, bool Out).
  или: (bool In1, bool In2, bool Out).
  не: (bool In, bool Out).

clauses
  и(0,0,0).      и(0,1,0).      и(1,0,0).      и(1,1,1).
  или(0,0,0).    или(0,1,1).    или(1,0,1).    или(1,1,1).
  не(0,1).        не(1,0).
```

А теперь разработаем на Прологе модель элемента И-НЕ на основе элемента И и элемента НЕ. Для этого выход Y элемента И соединим со входом элемента НЕ (рис. 33.2).



а



б

Рис. 33.2. Обозначение двухходового элемента И-НЕ (а) и его модель (б)

Объявим предикат и_не в разделе class predicates:

```
и_не: (bool In1, bool In2, bool Out) nondeterm anyflow.
```

и опишем его правило в разделе clauses:

```
и_не(X1, X2, Z) :- и(X1, X2, Y), не(Y, Z).
```

Обратите внимание, что переменная Y, являющаяся выходной в предикате и(X1, X2, Y), одновременно является входной в предикате не(Y, Z). Это полностью соответствует описанию, приведенному на рис. 33.2, б.

ПРИМЕР 33.5. Проверим работу элемента И-НЕ с помощью программы:

```
implement main
  open core, console
```

```

domains
    bool = [0..1].
class facts
    и:(bool In1,bool In2,bool Out).
    не:(bool In,bool Out).
class predicates
    и_не:(bool In1,bool In2,bool Out) nondeterm anyflow.
clauses
    и(0,0,0).      и(0,1,0).      и(1,0,0).      и(1,1,1).
    не(0,1).        не(1,0).

    и_не(X1,X2,Z) :- и(X1,X2,Y), не(Y,Z).

run() :-
    и_не(X1,X2,Z),
    writef("% % -> %",X1,X2,Z),nl,
    fail();
    write("Bce!!!"),
    _ = readchar().
end implement main
goal
    console:::run(main:::run).

```

Программа печатает на экране таблицу истинности для элемента И-НЕ. Она может работать при любой комбинации входных и выходных данных.

ПРИМЕР 33.6. Для определения выхода Z элемента И-НЕ при входах 0 и 1 достаточно в примере 33.5 изменить предикат `run` следующим образом:

```

run() :-
    и_не(0,1,Z),
    writef("0 1 -> %",Z),nl,
    fail();
    write("Bce!!!"),
    _ = readchar().

```

ПРИМЕР 33.7. Какие должны быть входы X₁, X₂ для того, чтобы выход Z элемента И-НЕ был единицей? Для ответа на вопрос достаточно в предикате `run` примера 33.5 указать требуемый выход, а в качестве входов использовать свободные переменные:

```

run() :-
    и_не(X1,X2,1),
    writef("% % -> 1",X1,X2),nl,
    fail();
    write("Bce!!!"),
    _ = readchar().

```

ПРИМЕР 33.8. Комбинированная задача: какой должен быть вход X₂ элемента И-НЕ для того, чтобы при входе X₁ = 1 выход Z был единицей? Для ее решения на-

до в примере 33.5 указать значения для X_1 и Z , а переменную X_2 оставить свободной, чтобы Пролог сам нашел ее значение:

```
run() :-  
    и_не(1,X2,1),  
    writef("1 % -> 1",X2),nl,  
    fail();  
    write("Все!!!!"),  
    _ = readchar().
```

ПРИМЕР 33.9. Надо проверить, допустима ли такая комбинация входных и выходных значений: $X_1=1, X_2=1, Z=0$. Укажем ее в вызове предиката `и_не(1,1,0)`:

```
run() :-  
    и_не(1,1,0),  
    writef("1 1 -> 0"),nl,  
    fail();  
    write("Все!!!!"),  
    _ = readchar().
```

Задание 33.6. Разработайте и проверьте модель элемента ИЛИ-НЕ.

Задание 33.7. Разработайте и проверьте модель элемента `xor/3` двухвходового сумматора по модулю 2. Его по-другому называют ИСКЛЮЧАЮЩЕЕ ИЛИ. Логика его работы проста: если входы различны, то на выходе — единица, иначе — ноль.

Задание повышенной трудности 33.2. Разработайте и проверьте модель двухразрядного сумматора `sum(X2, X1, Y2, Y1, P, Z2, Z1)`. Здесь складывается двухразрядное число X , представленное разрядами X_2 и X_1 , и двухразрядное число Y , представленное разрядами Y_2 и Y_1 . Получаемый результат: двухразрядное число Z , представленное разрядами Z_2 и Z_1 , и разряд переноса P .

33.3. Фигуры на плоскости

ПРИМЕР 33.10. Найдем все возможные отрезки на множестве из девяти точек, а также их количество. В качестве счетчика используем факт-переменную `i`. Для этого нам надо вызвать факт `point(X1, Y1)` для определения начала отрезка и факт `point(X2, Y2)` для определения конца отрезка:

```
implement main  
    open core, console  
class facts  
    point:(integer X, integer Y).  
    i:unsigned := 0.  
class predicates  
    отрезок:(integer StartX, integer StartY,  
              integer EndX, integer EndY) nondeterm (o,o,o,o).  
clauses  
    point(1,1).    point(1,2).    point(1,4).
```

```

point(2,1).    point(2,2).    point(2,4).
point(3,2).    point(4,1).    point(4,3).

отрезок(X1,Y1,X2,Y2) :- point(X1,Y1), point(X2,Y2), i:=i+1

run() :-
    отрезок(X1,Y1,X2,Y2),
    writeln("Найден отрезок (%,%)-(%,%)\n", X1,Y1,X2,Y2),
    fail();
    writeln("Найдено ~d решений", i),
    _ = readchar().
end implement main
goal
    console::run(main::run).

```

Эта программа найдет 81 отрезок. Почему так много? Потому что каждая точка была соединена с восемью другими точками — это восемь отрезков, а также с самой собой — это девятый отрезок.

Среди этих отрезков будет девять отрезков $[A, A]$, вырожденных в точку, а также для каждого отрезка $[A, B]$ будет найден и его дубликат $[B, A]$, в котором начало и конец переставлены местами.

Для подавления вывода вырожденных в точку отрезков достаточно проверить неравенство координат начала и конца отрезка.

Для подавления отрезков $[B, A]$, являющихся дубликатом отрезка $[A, B]$, надо использовать дополнительную проверку.

Избавимся вначале от отрезков, вырожденных в точку. Для этого определим предикат `равны/4`, который описывает две точки с одинаковыми координатами. Для невырожденных отрезков этот предикат должен быть ложным:

```

run() :-
    отрезок(X1,Y1,X2,Y2),
    writeln("Найден отрезок (%,%)-(%,%)\n", X1,Y1,X2,Y2),
    fail();
    writeln("Найдено % решений", i),
    _ = readchar().
end implement main
goal
    console::run(main::run).

```

На данном этапе количество решений уменьшилось до 72. Сейчас избавимся от отрезков-дубликатов. Для этого опишем предикат фильтр/4, который будет пропускать только те отрезки [A,B], у которых точка A лежит левее точки B. Если эти точки имеют одинаковую координату x, то этот предикат будет пропускать только те отрезки, у которых точка A лежит ниже точки B. Остальные отрезки он будет отфильтровывать:

```

implement main
    open core, console
class facts
    point:(integer X, integer Y).
    i:unsigned := 0.
class predicates
    отрезок : (integer StartX, integer StartY,
                integer EndX, integer EndY) nondeterm (o,o,o,o).
    равны : (integer,integer,integer,integer) determ (i,i,i,i).
    фильтр : (integer,integer,integer,integer) determ (i,i,i,i).
clauses
    point(1,1).    point(1,2).    point(1,4).
    point(2,1).    point(2,2).    point(2,4).
    point(3,2).    point(4,1).    point(4,3).

    отрезок(X1,Y1,X2,Y2) :- point(X1,Y1), point(X2,Y2),
                           not(равны(X1,Y1,X2,Y2)),
                           фильтр(X1,Y1,X2,Y2),
                           i := i+1.

    равны(X,Y,X,Y).

    фильтр(X1,Y1,X2,Y2) :- X1<X2, !; X1=X2, Y1<Y2.

run() :-
    отрезок(X1,Y1,X2,Y2),
    writeln("Найден отрезок (%,%)-(%,%)\n", X1,Y1,X2,Y2),
    fail();
    writeln("Найдено % решений", i),
    _ = readchar().
end implement main
goal
    console::run(main::run).

```

Сейчас наша программа выводит все 36 решений и ни одного лишнего. Однако в программе появился один лишний предикат. Этот предикат — `равны/4`. Дело в том, что логика предиката `фильтр/4` не только фильтрует дубликаты, но и не допускает вырожденных в точку отрезков, т. к. явно проверяет, чтобы точка A лежала строго левее или ниже точки B. Следовательно, предикат `равны/4` можно смело удалить и получить окончательный вариант программы:

```
implement main
    open core, console
class facts
    point:(integer X, integer Y).
    i:unsigned := 0.
class predicates
    отрезок:(integer StartX, integer StartY,
              integer EndX, integer EndY) nondeterm (o,o,o,o).
    фильтр:(integer,integer,integer,integer) determ (i,i,i,i).

clauses
    point(1,1).    point(1,2).    point(1,4).
    point(2,1).    point(2,2).    point(2,4).
    point(3,2).    point(4,1).    point(4,3).

    отрезок(X1,Y1,X2,Y2) :- point(X1,Y1), point(X2,Y2),
                           фильтр(X1,Y1,X2,Y2),
                           i := i+1.
    фильтр(X1,Y1,X2,Y2) :- X1<X2, !; X1=X2, Y1<Y2.

run() :-
    отрезок(X1,Y1,X2,Y2),
    writef("Найден отрезок (%,%)-(%,%)\n", X1,Y1,X2,Y2),
    fail();
    writeln("Найдено ~b решений", i),
    _ = readchar().
end implement main
goal
    console::run(main:::run).
```

ПРИМЕР 33.11. Усовершенствуем предикат `отрезок/4` так, чтобы он возвращал, кроме всего прочего, и длину отрезка:

```
implement main
    open core, console, math
class facts
    point:(integer X, integer Y).
class predicates
    отрезок:(integer StartX, integer StartY,
              integer EndX, integer EndY,
              real Длина) nondeterm (o,o,o,o,o).
    фильтр:(integer,integer,integer,integer) determ (i,i,i,i).
```

```
clauses
```

```
point(1,1).    point(1,2).    point(1,4).
point(2,1).    point(2,2).    point(2,4).
point(3,2).    point(4,1).    point(4,3).
```

```
отрезок(X1,Y1,X2,Y2,D) :- point(X1,Y1), point(X2,Y2),
    фильтр(X1,Y1,X2,Y2),
    D=sqrt(sqr(X1-X2) + sqr(Y1-Y2)).
```

```
фильтр(X1,Y1,X2,Y2) :- X1<X2, !; X1=X2, Y1<Y2.
```

```
run() :-
```

```
отрезок(X1,Y1,X2,Y2,D), 2<=D, D<=3,
writeln("Найден отрезок: (%,%)-(%,%)" длина: %.3 ~n",
        X1,Y1,X2,Y2,D),
fail();
write("Bce!!!"),
_ = readchar().
```

```
end implement main
```

```
goal
```

```
console:::run(main:::run).
```

В этом примере осуществляется вывод только тех отрезков, которые имеют длину, удовлетворяющую условию $2 \leq D \leq 3$. Обратите внимание, каким образом в предикате `writeln` задается вывод длины с тремя знаками после запятой.

Задание 33.8. Разработать программу вывода на экран всех горизонтальных отрезков, невырожденных в точку и не имеющих дубликатов.

Задание 33.9. Найти все невырожденные треугольники с периметром большим 9.

Задание 33.10. Найти все невырожденные равнобедренные треугольники.

Задание 33.11. Найти все квадраты, стороны которых параллельны координатным осям.

Задание 33.12. Найти треугольник с наибольшей площадью.

Задание повышенной сложности 33.3. Найти число треугольников, имеющих наименьший периметр.

Задание повышенной сложности 33.4. Найти параметры k и b для всех прямых $y = k \cdot x + b$, которые проходят не менее чем через три точки, заданные в базе фактов.

Задание повышенной сложности 33.5. Найти прямоугольник минимальной площади, который бы содержал все точки, заданные в базе фактов. Стороны прямоугольника параллельны координатным осям.

33.4. Ребус

ПРИМЕР 33.12. Рассмотрим известную задачу о поиске таких подстановок цифр вместо букв, при которых заданная фраза превращается в правильное арифметическое выражение. Есть только одно ограничение — каждая буква заменяется своей уникальной цифрой.

Такие ребусы можно решать различными способами. Здесь мы используем самый простой способ, основанный на поиске с откатом в базе фактов, описывающих цифры десятичной системы счисления. В главе 35 мы рассмотрим другие способы решения этого ребуса, основанные на списках и рекурсии.

Пусть дана фраза SEND+MORE=MONEY. Она состоит из восьми разных букв, следовательно, из десяти цифр надо выбрать восемь нужных. При выборе цифр из базы фактов следует проверять, чтобы очередная цифра была уникальной, т. е. ранее не выбиралась. Таким образом, при выборе второй цифры надо явно проверить ее неравенство первой цифре, при выборе третьей цифры надо проверить ее неравенство двум предыдущим и т. д. Когда будут выбраны первые семь цифр, то из них можно составить два слагаемых и вычислить сумму. После этого остается проверить, что в этой сумме используются «нужные» цифры. Для этого из семи ранее выбранных цифр и младшей цифры суммы формируется строка и проверяется ее соответствие вычисленной сумме.

Программа использует функцию `format` из открытого класса `string` для формирования строки MONEY из цифр. Функция `toTerm` преобразует строку MONEY в число для сравнения с суммой.

```

implement main
    open core, console, string
class facts
    цифра : (unsigned).      % база цифр
class predicates
    ребус : (unsigned,unsigned,unsigned) nondetterm (o,o,o).
clauses
    цифра(1). цифра(2). цифра(3). цифра(4). цифра(5).
    цифра(6). цифра(7). цифра(8). цифра(9). цифра(0).

    ребус(Число1,Число2,Сумма) :- цифра(S),           % цифра для буквы S
        цифра(E), E<>S,                           % цифра для буквы E
        цифра(N), N<>S, N<>E,                     % цифра для буквы N
        цифра(D), D<>N, D<>S, D<>E,             % цифра для буквы D
        цифра(M), M<>D, M<>N, M<>S, M<>E,       % цифра для буквы M
        цифра(O), O<>M, O<>D, O<>N, O<>S, O<>E, % цифра для буквы O
        цифра(R), R<>O, R<>M, R<>D, R<>N, R<>S, R<>E, % цифра для буквы R
    Число1=1000*S+100*E+10*N+D,                      % слагаемое SEND
    Число2=1000*M+100*O+10*R+E,                      % слагаемое MORE
    Сумма=Число1+Число2,                             % сумма в виде числа
    Сумма mod 10 = Y, Y<>R, Y<>O, Y<>M, Y<>D, Y<>N, Y<>S, Y<>E,

```

```

Слово = format("%у%и%и%и%и",M,O,N,E,Y), % формируем строку
Сумма = toTerm(Слово). % проверяем соответствие строки сумме

run() :-
    ребус(Число1, Число2, Сумма),
    write(Число1, "+", Число2, "=", Сумма), nl,
    fail;
    write("\nПрограмма завершена"),
    _=readchar().
end implement main
goal
    console::run(main:::run).

```

Эта программа находит все 25 решений:

2817+368=3185	2819+368=3187	3712+467=4179
3719+457=4176	3821+468=4289	3829+458=4287
5731+647=6378	5732+647=6379	5849+638=6487
6415+734=7149	6419+724=7143	6524+735=7259
6851+738=7589	6853+728=7581	7316+823=8139
7429+814=8243	7531+825=8356	7534+825=8359
7539+815=8354	7643+826=8469	7649+816=8465
8324+913=9237	8432+914=9346	8542+915=9457
9567+1085=10652		

ПРИМЕР 33.13. Существует другой способ проверки соответствия суммы заданной строке — разложить число на разряды математически, используя операцию деления без остатка `div` и операцию вычисления остатка от деления `mod`, последовательно для каждой цифры, начиная с младшей `Y` и до старшей `M`.

Далее представлен модифицированный предикат `ребус/3`, в котором реализован указанный способ разбиения числа на цифры.

```

ребус(Число1, Число2, Сумма) :- цифра(S),
    цифра(E), E<>S,
    цифра(N), N<>S, N<>E,
    цифра(D), D<>N, D<>S, D<>E,
    цифра(M), M<>D, M<>N, M<>S, M<>E,
    цифра(O), O<>M, O<>D, O<>N, O<>S, O<>E,
    цифра(R), R<>O, R<>M, R<>D, R<>N, R<>S, R<>E,
    Число1=1000*S+100*E+10*N+D,
    Число2=1000*M+100*O+10*R+E,
    Сумма=Число1+Число2,
    Сумма mod 10 = Y, % получение цифры Y
    Y<>R, Y<>O, Y<>M, Y<>D, Y<>N, Y<>S, Y<>E, % цифра Y уникальна
    Сумма div 10 = Oct,
    Oct mod 10 = E, % проверка цифры E
    Oct div 10 = Oct1,
    Oct1 mod 10 = N, % проверка цифры N
    Oct1 div 10 = Oct2,

```

```
Oct2 mod 10 = 0,          % проверка цифры О
Oct2 div 10 = Oct3,
Oct3 mod 10 = M.          % проверка цифры М
```

Задание 33.13. Написать предикат `ребус/3` поразрядного сложения «столбиком» слагаемых `SEND` и `MORE` для получения суммы `MONEY`.

Задание 33.14. Найти решения следующих ребусов:

вагон+вагон = состав

ветка+ветка = дерево

атака+удар+удар = нокаут

дедка+бабка+репка = сказка

Задание 33.15. Из девяти цифр от 1 до 9 составить «магический квадрат». Этот квадрат должен обладать тем свойством, чтобы суммы цифр в каждой его строке, столбце и диагоналях были одинаковы.

ГЛАВА 34



Рекурсивные правила

Рекурсия описана в *главе 11*. Здесь мы рассмотрим практические способы рекурсивной обработки рядов и строк.

34.1. Арифметика

ПРИМЕР 34.1. Найдем сумму всех двузначных чисел, кратных трем или семи. Для этого зададим рекурсивное правило `sum_3_7` (Счетчик, Аккумулятор, Сумма), которое проверит все числа от 10 до 99 по признаку делимости на 3 или 7 без остатка. Если такое число встречается, оно будет складываться с аккумулятором, в котором будет накапливаться искомая сумма.

```
implement main
    open core, console
class predicates
    sum_3_7: (unsigned, unsigned ,unsigned [out]) .
clauses
    sum_3_7(100,S,S) :- !.
    sum_3_7(N,A,S) :- (N mod 3= 0 ; N mod 7= 0), !,
                      sum_3_7(N+1,A+N,S) .
    sum_3_7(N,A,S) :- sum_3_7(N+1,A,S) .

run() :- 
    sum_3_7(10,0,S),
    write(S), nl,
    _ = readline() .
end implement main
goal
    console::run(main::run) .
```

Результат вычислений равен 2183. Далее представлена эта же задача, решенная в функциональном стиле.

```
implement main
    open core, console
```

```

class predicates
    sum_3_7: (unsigned, unsigned) -> unsigned.
clauses
    sum_3_7(100, S) = S:- !.
    sum_3_7(N, A) = sum_3_7(N+1, A+N) :- (N mod 3 = 0 ; N mod 7 = 0), !.
    sum_3_7(N, A) = sum_3_7(N+1, A).

run() :-
    write(sum_3_7(10, 0)),
    _ = readline().
end implement main
goal
    console:::run(main:::run).

```

Задание 34.1. Вывести на экран все двухзначные числа, произведение цифр которых равно 18. Подсказка 1: старший разряд двухзначного числа можно определить выражением $x \text{ div } 10$, младший — выражением $x \text{ mod } 10$, где x — проверяемое число. Подсказка 2: двузначное число можно собирать из цифр. В этом случае выделять старшие и младшие разряды не потребуется.

Задание 34.2. Написать функцию Аккермана. Эта функция часто используется для оценки способности компиляторов языков программирования эффективно выполнять рекурсию. Подсказка: функция Аккермана имеет следующее математическое описание:

```

ack(0, x) = x+1
ack(x, 0) = ack(x-1, 1)
ack(x, y) = ack(x-1, ack(x, y-1))

```

Проверьте с помощью вашего предиката значения функции Аккермана для следующих пар аргументов:

```

ack(3, 6) = 509
ack(3, 7) = 1021
ack(3, 8) = 2045
ack(3, 9) = 4093
ack(3, 10) = 8189

```

Задание 34.3. Разработать предикат для быстрого вычисления n -го числа Фибоначчи. В ряде Фибоначчи каждый следующий член ряда равен сумме двух предыдущих. Первые два члена ряда равны единице. Аргументы этого предиката должны быть такими:

```

quick_fib(integer First,           % первый член ряда
           integer Second,          % второй член ряда
           unsigned Index,          % индекс искомого члена
           integer NthItem).        % искомая величина

```

Работа предиката должна быть основана на том, что каждый следующий член ряда получается путем сложения двух предыдущих, которые хранятся в первом и

втором аргументах соответственно. Индекс при этом декрементируется. Рекурсия останавливается тогда, когда индекс уменьшится до трех. При этом $NthItem=First+Second$, т. к. третий член равен сумме первого и второго.

ПРИМЕР 34.2. Проверим, является ли число n , введенное с клавиатуры, простым. Вначале поступим так, как велит определение простого числа — убедимся, что оно делится без остатка только на себя и на единицу. Для этого проверим по очереди все числа от 2 до $n-1$ в роли делителей числа n . Если хотя бы одно из этих чисел делит n без остатка, то n — составное число. Иначе n является простым.

```
implement main
    open core, console
class predicates
    простое: (unsigned64) determ.
    тест: (unsigned64, unsigned64) determ.
clauses
    простое(Число) :- тест(2, Число).

    тест(Делитель, Число) :- Делитель >= Число, !.           % условие останова
    тест(Делитель, Число) :- Число mod Делитель > 0,          % не делится
                            тест(Делитель+1, Число).

run() :-
    Число = read(),                                % вводим число
    ( простое(Число), write("Простое"), !;      % число простое
      write("Составное") ),                        % число составное
    clearinput,
    _ = readline().
end implement main
goal
    console::run(main:::run).
```

Попробуем усовершенствовать алгоритм, проверяя только нечетные делители. Тем самым мы сократим перебор в два раза. Непосредственно перед этим убедимся в нечетности числа n , если оно не равно 2:

```
implement main
    open core, console
class predicates
    простое: (unsigned64) determ.
    тест: (unsigned64, unsigned64) determ.
clauses
    простое(Число) :- Число = 2, !;
                    Число mod 2 > 0,          % число нечетное
                    тест(3, Число).

    тест(Делитель, Число) :- Делитель >= Число, !.
    тест(Делитель, Число) :- Число mod Делитель <> 0,
                            тест(Делитель+2, Число). % шаг равен 2
```

```

run() :-  

    Число = read(), % вводим число  

    ( простое(Число), write("Простое"), !; % число простое  

      write("Составное") ), % число составное
    clearinput,  

    _ = readline().
end implement main
goal
    console::run(main::run).

```

И, наконец, сократим перебор еще немного, проверяя делители только от 3 до $\lfloor \sqrt{N} \rfloor$, где $\lfloor \sqrt{N} \rfloor$ — округленный до меньшего целого корень квадратный из N . Почему именно так? Потому что если есть делитель, больше $\lfloor \sqrt{N} \rfloor$, то обязательно будет парный ему, который меньше $\lfloor \sqrt{N} \rfloor$, а его мы обнаружим ранее. В граничном случае делитель будет равен $\lfloor \sqrt{N} \rfloor$. Visual Prolog в классе `math` содержит ряд функций округления чисел. Для округления больших чисел можно воспользоваться функцией `roundToInteger64`:

```

implement main
    open core,console,math
class predicates
    простое: (unsigned64) determ.
    тест: (unsigned64,unsigned64,unsigned64) determ.
clauses
    простое(Число) :- Число=2, !;
                    Число mod 2 <>0,
                    Граница = roundToUnsigned64(sqrt(abs(Число))),
                    тест(3,Граница,Число).

    тест(Делитель,Граница,_) :- Делитель>Граница, !.
    тест(Делитель,Граница,Число) :- Число mod Делитель <> 0,
                                    тест(Делитель+2,Граница,Число).

run() :-  

    Число = read(), % вводим число  

    ( простое(Число), write("Простое"), !; % число простое  

      write("Составное") ), % число составное
    clearinput,  

    _ = readline().
end implement main
goal
    console::run(main::run).

```

Задание 34.4. Написать предикат проверки простого числа на основе решета Эратосфена. Суть алгоритма можно посмотреть в Википедии или любом учебнике по теории чисел.

Задание 34.5. Написать программу вычисления и вывода на экран первых ста простых чисел. Первым простым числом является 2.

Задание 34.6. Написать программу вычисления всех простых чисел в диапазоне от 3 до N , где число N вводится с клавиатуры.

ПРИМЕР 34.3. Найдем наименьший общий делитель (НОД) по алгоритму Евклида.

```
implement main
    open core, console
class predicates
    евклид : (integer, integer, integer [out]) .
clauses
    евклид(А, Б, Б) :- А mod Б = 0, !.
    евклид(А, Б, НОД) :- А mod Б = В, !, евклид(В, В, НОД) .

run() :- 
    евклид(12, 27, НОД),
    write("НОД = ", НОД),
    _ = readline() .
end implement main
goal
    console::run(main:::run) .
```

Задание 34.7. Написать предикат определения двух взаимно простых чисел. Подсказка: взаимно простые числа имеют только один общий делитель — единицу.

ПРИМЕР 34.4. Опишем предикат преобразования арабских чисел в римские. Для этого нам потребуются правила соответствия арабских чисел римским, в программе эти правила называются фактами замена. Также нам потребуется функция соединения двух строк concat, которая описана в классе string. Смысл преобразования заключается в последовательном вычитании из заданного числа такого наибольшего арабского числа, взятого из правил замены, которое оставляет полученную разность положительной. При этом римский эквивалент арабского числа присоединяется к результату справа. В связи с этим факты замена должны быть расположены в программе в порядке убывания от наибольших чисел к наименьшим.

```
implement main
    open core, console, string
class facts
    замена: (integer, string) .
class predicates
    араб_рим: (integer, string [out]) nondeterm.
clauses
    араб_рим(0, "") :- !.
    араб_рим(А, concat(R1, R2)) :- замена(А1, R1),
        А2 = А-А1, А2>=0, араб_рим(А2, R2) .

замена(1000, "M") .
замена(900, "CM") .
```

```

замена(500, "D").
замена(400, "CD").
замена(100, "C").
замена(90, "XC").
замена(50, "L").
замена(40, "XL").
замена(10, "X").
замена(9, "IX").
замена(5, "V").
замена(4, "IV").
замена(1, "I").

run() :-  

    (араб_рим(2013,R), write(R), !;           % MMXIII  

     write("Ошибка преобразования")),  

     _ = readline().  

end implement main  

goal  

    console::run(main:::run).

```

Задание 34.8. Написать программу преобразования римских чисел в арабские.
Подсказка: вам потребуется предикат

`front(Source, Count, First [out], Last [out])`

из класса `string`. Он расщепляет строку `Source` на две строки: `First` и `Last`, причем строка `First` будет содержать `Count` символов. Например, вызов предиката

`front("MMXIII", 1, First, Last)`

вернет результат:

`First="M", Last="MXIII".`

После чего останется сложить арабский эквивалент 1000 римского числа `M` с аккумулятором.

34.2. Ряды

ПРИМЕР 34.5. Вычислим сумму первых десяти членов знакопеременного ряда:

$$1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \frac{1}{4!} - \dots = \frac{1}{e}$$

и найдем ее отличие от величины $\frac{1}{e}$. Для этого введем домен:

знак = минус ; плюс.

Переменная этого домена будет менять свое значение на каждом витке цикла и послужит признаком знака для очередного члена ряда. Второй особенностью программы является отсутствие непосредственного вычисления факториала для каждого члена ряда. Вместо этого очередной член ряда вычисляется посредством деления

ния предыдущего члена ряда на значение счетчика. Третья особенность — формирование строки результата с помощью префикса `@`, который в этом примере построчно формирует текст результата:

```

implement main
    open core, console, string
domains
    знак = минус;плюс.
class predicates
    f: (unsigned,          % счетчик
         real,             % аккумулятор
         real,             % последний член ряда
         знак,             % знак
         unsigned)        % оставшееся число членов ряда
        -> real.
clauses
    f(_,Аккум,_,_0) = Аккум :- !.
    f(Счетчик,Аккум,Член,минус,N) =
        f(Счетчик+1,Аккум-Член1,Член1,плюс,N-1):-Член1 = Член /Счетчик,! .
    f(Счетчик,Аккум,Член,плюс,N) =
        f(Счетчик+1,Аккум+Член1,Член1,минус,N-1):-Член1 = Член /Счетчик.

run() :-
    Сумма = f(1,1,1,минус,10),
    Е = 1/math::e,
    writeln(
@"Сумма ряда = %
1/e = %
Разность = %", Сумма,Е,math::abs(Сумма-Е)),
    _ = readline().
end implement main
goal
    console::run(main:::run).

```

Результат работы показывает, что десяти членов ряда недостаточно для точного описания величины $\frac{1}{e}$. Увеличив число членов, например, до 100, можно увидеть достаточно точное значение.

Задание 34.9. Разработать функцию для нахождения суммы ряда:

$$1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \dots = e^x$$

где x — значение переменной. Этот ряд является приближением функции e^x . Проверить работу этой функции ограничившись первыми десятью членами ряда, сравнив ее результаты со значением встроенной функции e^x при $x = 0,7$. Подсказка 1: очередной член ряда можно получить из предыдущего, посредством умножения на x и деления на значение счетчика. Подсказка 2: в Visual Prolog число e

описано в классе `math`. Поэтому обращение к этому числу имеет вид `math::e`. Подсказка 3: в Visual Prolog возведение числа `e` в степень `x` записывается так: `math::e^x`.

Задание 34.10. Разработать функцию для нахождения суммы первых десяти членов ряда:

$$1 - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \dots = \sin(x).$$

34.3. Длинная арифметика

ПРИМЕР 34.6. Опишем операцию сложения целых неотрицательных чисел произвольной разрядности. Такие числа будем представлять строками, содержащими десятичные цифры. Сложение будем делать поразрядно, вычисляя значение переноса и передавая его операции сложения следующих разрядов. Перед сложением длинных чисел их длины надо выровнять путем добавления в начало более короткого числа необходимого количества нулей.

```
implement main
    open core, console, string
domains
    перенос = [0..1].           % значение переноса 0 или 1
class predicates
    сложение:(string, string, string [out]). 
    сложить:(string, string, перенос, charCount, string, string [out]). 
clauses
    сложение(Аргумент1, Аргумент2, Сумма) :- 
        Длина1 = length(Аргумент1),
        Длина2 = length(Аргумент2),
        Длина = math::max(Длина1, Длина2),
        (
        Длина1 > Длина2, !,
        Аргумент22=adjustRight(Аргумент2, Длина, "0"),
        сложить(Аргумент1, Аргумент22, 0, Длина, "", Сумма);
        Длина1 < Длина2, !,
        Аргумент11=adjustRight(Аргумент1, Длина, "0"),
        сложить(Аргумент11, Аргумент2, 0, Длина, "", Сумма);
        сложить(Аргумент1, Аргумент2, 0, Длина, "", Сумма)
    ). 

    сложить(_, _, 0, 0, Сумма, Сумма) :-!. 
    сложить(_, _, 1, 0, Аккум, Сумма) :-!, Сумма=concat("1", Аккум). 
    сложить(Число1, Число2, Перенос, Длина, Аккум, Сумма) :- 
        Длина1 = Длина-1,
        front(Число1, Длина1, Ост1, Цифра1),
        front(Число2, Длина1, Ост2, Цифра2),
        А = toTerm(Цифра1),
        Б = toTerm(Цифра2),
        В = А+Б+Перенос,
```

```

Цифра = В mod 10,
Перенос1 = В div 10,
Аккум1 = concat(toString(Цифра), Аккум),
сложить(Ост1, Ост2, Перенос1, Длина1, Аккум1, Сумма).

run() :-  

    Аргумент1 = "2343124386865890390432032043094385677",
    Аргумент2 = "890675897560760965079450943568734876",
    сложение(Аргумент1, Аргумент2, Сумма),
    write(Сумма),
    _ = readLine().
end implement main
goal
    console::run(main::run).

```

Задание 34.11. Описать операцию сложения целых неотрицательных чисел произвольной разрядности. Сложение производить не по одному разряду, а фрагментами чисел, величина которых представима в разрядной сетке компьютера.

Задание 34.12. Описать операцию вычитания целых неотрицательных чисел произвольной разрядности.

Задание 34.13. Описать операцию умножения целого неотрицательного числа произвольной разрядности на десятичную цифру.

Задание 34.14. Описать операцию целочисленного деления целого неотрицательного числа произвольной разрядности на десятичную цифру с вычислением остатка от деления.

Задание 34.15. Описать операцию умножения целых неотрицательных чисел произвольной разрядности.

Задание 34.16. Описать операцию целочисленного деления целых неотрицательных чисел произвольной разрядности с вычислением остатка от деления.

Числа в длинной арифметике могут задаваться в различных системах счисления. Для совершения операций над такими числами есть два пути. Один путь заключается в повторном описании всех операций для каждой системы счисления. Второй путь состоит в переводе чисел из заданной системы счисления в одну, например, десятичную, и выполнения действий над числами в этой десятичной системе счисления. После этого, при необходимости, результат преобразуется в исходную систему счисления.

34.4. Перевод чисел из одной системы счисления в другую

ПРИМЕР 34.7. Опишем предикат преобразования двоичного беззнакового числа в десятичную систему счисления. Механизм такого преобразования описывается известной формулой:

$$b_n \dots b_1 b_0 = b_n \cdot 2^n + \dots + b_1 \cdot 2^1 + b_0 \cdot 2^0.$$

Здесь $b_n \dots b_1 b_0$ — последовательность разрядов двоичного числа, от старшего разряда к младшему. Длина последовательности не должна превышать разрядности процессора. Например, двоичное число 1111_2 переводится в десятичную систему так:

$$1111_2 = 2^3 \cdot 1 + 2^2 \cdot 1 + 2^1 \cdot 1 + 2^0 \cdot 1 = 8 + 4 + 2 + 1 = 15_{10}.$$

Далее представлена программа, работающая точно по описанному принципу:

```
implement main
    open core, console, string
domains
    степень = [- 1..].
class predicates
    перевод_2_10:(string,unsigned [out]) .
    перевести_2_10:(степень,string,unsigned,unsigned [out]) .
clauses
    перевод_2_10(Биты,Число) :- Длина = length(Биты),
        перевести_2_10(Степень-1,Биты,0,Число) .

    перевести_2_10(Степень,_,Число,Число) :- Степень < 0, ! .
    перевести_2_10(Степень,Биты,Аккум,Число) :-  

        front(Биты,1,Бит,Ост),
        Аккум1 = Аккум + toTerm(Бит) * 2^Степень,
        перевести_2_10(Степень-1,Ост,Аккум1,Число) .

run() :-
    Bin = "1010",
    перевод_2_10(Bin,N),
    write(N), % 10
    _ = readline().
end implement main
goal
    console::run(main::run) .
```

Существует более эффективный алгоритм рассматриваемого преобразования. Он основан на вычислении по той же исходной формуле, но двойки в ней последовательно вынесены за скобки. Например, двоичное число 1111_2 можно перевести в десятичную систему так:

$$1111_2 = 2^1 \cdot (2^1 \cdot (2^1 \cdot 1 + 1) + 1) + 1 = 15_{10}.$$

Вычисления начинаются с самого внутреннего выражения. Эффективность выше за счет того, что на каждом витке рекурсии вычисляется не степень двойки, а производится умножение на 2 и сложение с очередным разрядом:

```
implement main
    open core, console, string
```

```

class predicates
    перевод_2_10:(string,unsigned [out]) determ.
    перевести_2_10:(string,unsigned,unsigned [out]) determ.
clauses
    перевод_2_10(Биты,Число) :- перевести_2_10(Биты,0,Число) .

    перевести_2_10("",X,X):-! .
    перевести_2_10(Биты,A,X) :-
        frontchar(Биты,Бит,Ост),
        I = if Бит='1' then 1 else 0 end if,
        A1=A*2+I,
        перевести_2_10(Ост,A1,X) .

run() :-
    Bin = "1010",
    перевод_2_10(Bin,N),
    write(N),!, _=readline();
    _ = readline().
end implement main
goal
    console::run(main:::run).

```

Задание 34.17. Написать предикат перевода чисел из десятичной системы счисления в двоичную.

Задание 34.18. Написать предикат перевода чисел из десятичной системы счисления в шестнадцатеричную.

ПРИМЕР 34.8. Написать программу перевода вещественных чисел из одной системы счисления в другую. Воспользуемся простым алгоритмом, который организует промежуточный перевод исходного числа в десятичную систему, а из нее — в выходную систему счисления. Таким образом, перевод числа получается двухступенчатым. Например, при переводе вещественного числа $B.6$, представленного в 12-ричной системе счисления, в двоичную систему, будет сделан промежуточный перевод в десятичную систему, в которой оно равно 11.5:

$$B.6_{12} \rightarrow 11.5_{10} \rightarrow 1011.1_2.$$

Промежуточный перевод обусловлен тем, что в качестве цифр систем счисления мы используем символы, коды которых представимы в десятичной системе. Эти коды можно было бы представить и в 16-ричной системе, но суть от этого не меняется — промежуточный перевод использовать приходится все равно.

Чтобы выходное число не содержало бесконечную дробь, введем параметр, ограничивающий количество разрядов дробной части выходного числа. Решение этой задачи сводится к написанию предиката conv/7:

```

conv:(unsigned Base, % основание исходной системы счисления
      string InNumber, % входное число
      unsigned Integer, % целая часть в десятичной системе счисления
      real Frac,         % дробная часть в десятичной системе счисления

```

```

unsigned NewBase,    % основание выходной системы счисления
string OutNumber,   % выходное число
integer FracLen)    % максимальное число разрядов дробной части
                     % выходного числа

```

В этом предикате аргументы `Integer`, `Frac` и `OutNumber` являются выходными. Для вычисления параметров `Integer` и `Frac` нужны два предиката:

- `toDec` — преобразование целой части исходного числа в десятичную систему;
- `toDecFract` — преобразование дробной части исходного числа в десятичную систему.

Кроме этого, для получения `OutNumber` из параметров `Integer` и `Frac` нам потребуются еще два предиката:

- `intTo` — преобразование целого числа из десятичной системы в выходную;
- `fracTo` — преобразование дробной части числа из десятичной системы в выходную.

Предикат `toDec` вычисляет значение выражения в позиционной в-ичной системе счисления. Пусть входное число $A = A_3A_2A_1A_0.A_{-1}A_{-2}A_{-3}A_{-4}$ имеет четыре разряда в целой части $A_3A_2A_1A_0$ и столько же в дробной части $A_{-1}A_{-2}A_{-3}A_{-4}$. Значение целой части в десятичной системе представляет собой выражение:

$$10^3 \cdot A_3 + 10^2 \cdot A_2 + 10^1 \cdot A_1 + 10^0 \cdot A_0,$$

Однако для увеличения скорости вычислений это выражение мы преобразуем к виду, в котором общий множитель 10 вынесен за скобки:

$$10^1 \cdot (10^1 \cdot (10^1 \cdot A_3 + A_2) + A_1) + A_0.$$

Вычисление начинается с самого внутреннего выражения, т. е. с самой старшей цифры числа, которая отделяется от числа предикатом `frontchar`. Увеличение скорости вычислений обусловлено тем, что на каждом витке цикла выполняется всего одно умножение и одно сложение, в отличие от первоначального выражения, в котором для вычисления каждого слагаемого необходимо было вычислять степень, умножать ее на очередной разряд числа и складывать с аккумулятором.

Предикат `toDecFract` вычисляет значение дробной части входного числа. Например, для четырехразрядной дробной части $A_{-1}A_{-2}A_{-3}A_{-4}$ выражение будет таким:

$$A_{-1} \cdot 10^{-1} + A_{-2} \cdot 10^{-2} + A_{-3} \cdot 10^{-3} + A_{-4} \cdot 10^{-4}.$$

Для увеличения скорости вычислений это выражение мы преобразуем к виду, в котором общий множитель 10^{-1} вынесен за скобки:

$$(A_{-1} + (A_{-2} + (A_{-3} + A_{-4} \cdot 10^{-1}) \cdot 10^{-1}) \cdot 10^{-1}) \cdot 10^{-1}.$$

Вычисление начинается с самого внутреннего выражения, т. е. с самой младшей цифры числа, которая отделяется от числа предикатом `lastchar`.

Предикат `intTo` выполняет деление с остатком целого числа. Делитель здесь — основание выходной системы счисления. Остатки от деления представляют собой цифры целого числа в выходной системе счисления. Целочисленное частное на следующем витке цикла становится делимым.

Предикат `fracTo` выполняет аналогичные действия, что и предикат `intTo`, только для дробной части. Так как для вещественных чисел нет операции целочисленного деления, то эта операция выполняется с помощью операции округления `trunc`.

Последний вопрос, который надо решить перед написанием программы, — какой алфавит систем счисления выбрать. Для обозначения цифр от 0 до 9 выберем символы десятичной системы счисления от 0 до 9. Вслед за ними пусть следуют все 26 заглавных букв английского алфавита. Буква A будет десятым символом алфавита, буква B — одиннадцатым и т. д. Буква Z будет 35-м символом алфавита. Далее пусть следуют строчные английские буквы: буква a — 36-й символ алфавита и последняя 61-я буква алфавита — z. Итого мы описали алфавит 62-ричной системы счисления.

Символы от 0 до 9 имеют коды от 48 до 57. Поэтому если из кода символа вычесть число 48, то мы получим числовое значение символа. Буквы от A до Z имеют коды от 65 до 90 и описывают цифры от 10 до 35 нашей 62-ричной системы счисления. Для получения числового значения английской заглавной буквы надо из ее кода вычесть число 55. И, наконец, буквы от a до z имеют коды от 95 до 122 и описывают цифры от 36 до 61. Для получения числового значения английской строчной буквы надо из ее кода вычесть число 61.

Предикат `unicode_digit` определяет числовое значение символа алфавита по его коду. Предикат `digit_symbol` по числовому значению символа алфавита определяет сам символ:

```

implement main
    open core, console, string, math
class predicates
    conv:(unsigned Base, string InNumber, unsigned Integer,
          real Frac, unsigned NewBase, string OutNumber,
          integer FracLen) determ (i,i,o,o,i,o,i).
    toDec:(unsigned Base, string InNumber, unsigned Accum,
           unsigned Integer, real Frac) determ (i,i,i,o,o).
    toDecFract:(unsigned Base, string InNumber, real Accum, real Frac)
               determ (i,i,i,o).
    intTo:(unsigned Integer, unsigned NewBase, string Accum,
           string Integer) determ (i,i,i,o).
    fracTo:(real Frac, unsigned NewBase, string Accum,
            string Fraction, integer FracLen) determ (i,i,i,o,i).
    unicode_digit:(unsigned Unicode,unsigned Digit) determ (i,o).
    digit_symbol:(unsigned Digit,string Symbol) determ (i,o).
clauses
    conv(B,N,Int,Frac,B0,N0,Len):-
        toDec(B,N,0,Int,Frac),           % получение 10-ричного целого и дроби

```

```

intTo(Int,B0,"",Int0),           % перевод целой части числа
fracTo(Frac,B0,"",Frac0,Len),    % перевод дробной части числа
N0 = if Frac0="" ; toTerm(Frac0)=0.0 then % формат. результата
      format("%",Int0)
      else format("%.%",Int0,Frac0) end if.

toDec(_, "", N, N, 0.0) :-!.   % целая часть закончилась, дробь равна 0
toDec(B, N, Int, Int1, Frac1) :-
    frontchar(N, C, R), C = '.', !,   % встретилась десятичная точка
    toDecFract(B, R, 0, Frac),       % перевод дробной части числа
    (Frac >= 1, Frac1 = Frac - 1, Int1 = Int + 1, !;)  % если дробь =1
    Frac1 = Frac, Int1 = Int).
toDec(B, N, A, Int, Frac) :-
    frontchar(N, C, R),             % отделение очередного символа
    U = getCharValue(C),
    unicode_digit(U, D), D < B,    % преобразование кода символа в цифру
    A1 = A * B + D,                % накапливаем целую часть
    toDec(B, R, A1, Int, Frac).    % в аккумуляторе

toDecFract(_, "", Frac, Frac) :-!.   % дробная часть закончилась
toDecFract(B, N, A, Frac) :-
    lastchar(N, R, C),             % отделение символа справа от дробной части
    U = getCharValue(C),
    unicode_digit(U, D), D < B, !,   % преобразование кода символа в цифру
    A1 = (A + D) / B,              % накапливаем в аккумуляторе дробную часть
    toDecFract(B, R, A1, Frac).

intTo(N, B, A, S) :-
    Z = N div B, Z > 0, !,        % число целых оснований
    R = N mod B,                  % остаток
    digit_symbol(R, C),          % получение символа
    A1 = concat(C, A),            % накапливаем целую часть в аккумуляторе
    intTo(Z, B, A1, S).
intTo(N, B, A, S) :-             % если Z=0
    R = N mod B,
    digit_symbol(R, D),
    S = concat(D, A).

fracTo(_, _, Frac, Frac, 0) :-!.   % длина дробной части равна нулю
fracTo(0.0, _, Frac, Frac, _) :-!. % дробная часть равна нулю
fracTo(Frac, B, A, Frac0, Len) :-
    T = Frac * B,                 % деление дробной части на основание системы
    Z = convert(unsigned, trunc(T)), % число целых оснований
    Z < B,
    Frac1 = T - Z,                % остаток дробной части
    digit_symbol(Z, D),           % получение символа
    Frac0 = Frac1 / B,             % дробная часть
    Len = 1.
```

```

A1=concat(A,D),          % накапливаем дробную часть в аккумуляторе
fracTo(Frac1,B,A1,Frac0,Len-1).

unicode_digit(U,D):-
    U>64,U<91,D=U-55,!;   % получение значения буквы от "A" до "Z"
    U>96,U<123,D=U-61,!;   % получение значения буквы от "a" до "z"
    U>47,U<58,D=U-48.     % получение значения цифры от 0 до 9

digit_symbol(D,S):-
    D<10,! ,S=toString(D);      %получение символа цифр от 0 до 9
    D>9,D<36, Unicode=D+55,! , %получение символа букв от "A" до "Z"
    Char=getCharFromValue(Unicode), S=charToString(Char);
    D>35,D<62, Unicode=D+61,  %получение символа букв от "a" до "z"
    Char=getCharFromValue(Unicode), S=charToString(Char).

run():-
    N="1.zzzzz",           % исходное число
    B=62,                  % основание входной системы счисления
    B1 = 2,                 % основание выходной системы счисления
    conv(B,N,Int,Frac,B1,N1,15),
    writeln("Исходное число: % (основание = %)\n",N,B),
    writeln("10-е целое : %\n10-е дробное: %\n",Int,Frac),
    writeln("Результат: % (основание = %)\n",N1,B1),
    _ = readchar(),!;
    _ = readchar().
end implement main
goal
    console::run(main:::run).

```

Результат работы:

```

Исходное число: 1.zzzzz (основание = 62)
10-е целое : 1
10-е дробное: 0.999999998908455
Результат: 1.11111111111111 (основание = 2)

```

Когда в выходной системе счисления невозможно точно описать дробь, представленную во входной системе, то программа использует округление дроби. Например, исходное число 1.zzzzzzzzzz столь близко к двойке, что ни десятичная, ни, тем более, двоичная системы счисления не могут точно описать такую близость, используя ограниченное число разрядов. Поэтому выходное число округляется до двойки, чтобы результат более всего соответствовал величине исходного числа:

```

Исходное число: 1.zzzzzzzzzz (основание = 62)
10-е целое : 2
10-е дробное: 0
Результат: 10 (основание = 2)

```

С другой стороны, если исходное число 1.0000000001 представлено в 62-ричной системе счисления, то оно мало отличается от единицы. И если в десятичной сис-

теме еще можно увидеть разницу между единицей и исходным числом, то в двоичной системе опять будет произведено округление, но уже в меньшую сторону:

```
Исходное число: 1.00000000001 (основание = 62)
10-е целое : 1
10-е дробное: 1.92172577675874e-020
Результат: 1 (основание = 2)
```

Задание 34.19. Написать программу перевода длинных вещественных чисел из одной системы счисления в другую. Промежуточные десятичные числа представлять строкой.

Задание 34.20. Написать программу перевода вещественных чисел из одной системы счисления в другую для алфавита системы счисления, содержащего 100 символов. Справка: добавьте в алфавит системы счисления недостающие символы из национального (русского) алфавита.

34.5. Обработка строк

ПРИМЕР 34.9. Напишем программу определения фраз-палиндромов. Палиндром — последовательность символов, читаемая одинаково слева направо и справа налево.

Для решения этой задачи вначале преобразуем исходную фразу к нижнему регистру и удалим все пробелы. Затем с помощью рекурсивного предиката `симметрия/2` будем читать символы слева и справа и сравнивать между собой. Рекурсия остановится, когда в строке останется либо один символ, либо ни одного.

```
implement main
    open core, console, string
class predicates
    удалить_пробелы : (string, string, string [out]).
    симметрия : (unsigned, string) determ.
clauses
    удалить_пробелы(Фраза, Аккум, Фраза1) :-
        frontToken(Фраза, Слово, Остаток), !,
        удалить_пробелы(Остаток, concat(Аккум, Слово), Фраза1).
    удалить_пробелы(_, Аккум, Аккум).

    симметрия(Длина, _) :- Длина <= 1, !.
    симметрия(Длина, Фраза) :-
        frontChar(Фраза, Первая, Ост),      % отрезаем первую букву
        lastChar(Ост, Ост1, Последняя),     % отрезаем последнюю букву
        Первая = Последняя, !,
        симметрия(Длина-2, Ост1).
```

```
run() :-
    Фраза = "А роза упала на лапу Азора",
    Фраза1 = toLowerCase(Фраза),
```

```

удалить_пробелы(Фраза1, "", Фраза2),
Длина = length(Фраза2) div 2,
( симметрия(Длина, Фраза2), write("Палиндром"), !;
  write("Не палиндром") ),
_ = readline().
end implement main
goal
  console::run(main::run).

```

Задание 34.21. Разработать программу определения палиндрома на основе другого способа — получить реверс исходной строки и сравнить эти строки между собой.

Задание 34.22. Проверить, является ли автобусный шестизначный билет «счастливым». «Счастливый» номер — это номер, у которого сумма первых трех цифр равна сумме последних трех цифр. Подсказка: для выделения первой и второй троек цифр шестизначного номера можно воспользоваться предикатом `string::front`, предварительно переведя шестизначный номер в строку функцией `toString`.

ПРИМЕР 34.10. Напишем программу поиска слова в тексте по максимальному количеству совпадений первых букв. Для решения этой задачи воспользуемся счетчиком совпадающих букв и будем сохранять то слово, у которого значение счетчика наибольшее. Предикат `поиск/5` для каждого очередного слова текста подсчитывает количество совпадений его первых букв с образцом с помощью предиката `число_совпадений/3`. Если это количество больше текущего максимального, то данное слово становится на место бывшего лидера, иначе оно пропускается.

```

implement main
  open core, console, string
class predicates
  поиск: (string,           % что ищем
          string,           % где ищем
          string [out],    % результат поиска
          unsigned,         % счетчик
          string).          % текущий лидер
  число_совпадений : (string, string, unsigned [out]). 
clauses
  поиск(Слово, Текст, Слово1, Число, _) :- 
    fronttoken(Текст, Лексема, Остаток),
    число_совпадений(Слово, Лексема, Число0),
    Число0 > Число, !,
    поиск(Слово, Остаток, Слово1, Число0, Лексема).
  поиск(Слово, Текст, Слово1, Число, Лексема) :- 
    fronttoken(Текст, _, Остаток), !,
    поиск(Слово, Остаток, Слово1, Число, Лексема).
  поиск(_, _, Слово, _, Слово).

  число_совпадений(Слово, Слово1, Число) :- 
    frontchar(Слово, Буква, Остаток),

```

```

frontchar(Слово1, Буква1, Остаток1),
Буква = Буква1, !,
число_совпадений(Остаток, Остаток1, Число1),
Число = Число1 + 1.
число_совпадений(_, _, 0).

run() :-
    Текст = readline(),      % вводим текст, например, adc abe cde ace
    Слово = readline(),      % вводим искомое слово, например, abd
    поиск(Слово, Текст, Слово1, 0, ""),
    write(Слово1),           % вывод: abe
    _ = readline().
end implement main
goal
    console:::run(main:::run).

```

Задание 34.23. Разработать программу поиска слова в тексте по максимальному количеству вхождений букв искомого слова в слово текста в порядке следования букв в слове.

Задание 34.24. Разработать программу поиска максимального по длине и минимального по длине слов в тексте.

ПРИМЕР 34.11. Напишем программу удаления комментариев из исходного текста Prolog-программы. Однострочные комментарии будут удаляться от символа процента и до конца строки. Блочный комментарий будет удаляться от символов начала "/*" до символов конца комментария "*/". Если признак конца блочного комментария не будет найден, то на экран будет выведено соответствующее сообщение, а работа предиката удалить_коммент/3 завершится неудачей.

```

implement main
    open core, console, string
class predicates
    удалить_коммент: (string, string, string [out]) determ.
    удалить_строку: (string, string [out]).
    удалить_строки: (string, string [out]) determ.
clauses
    удалить_коммент(Текст, Аккум, Текст1) :-
        frontchar(Текст, Символ, Остаток),
        Символ = '%', !,                      % начало однострочного комментария
        удалить_строку(Остаток, Остаток1),     % удаляем строку
        удалить_коммент(Остаток1, Аккум, Текст1).

    удалить_коммент(Текст, Аккум, Текст1) :-
        hasPrefix(Текст, "/*", Остаток), !,    % начало блочного комментария
        удалить_строки(Остаток, Остаток1),     % удаляем блок до символов */
        удалить_коммент(Остаток1, Аккум, Текст1).

    удалить_коммент(Текст, Аккум, Текст1) :- % если не комментарий, то
        tryfront(Текст, 1, Символ, Остаток),   % отделяем один символ и
        Аккум1 = concat(Аккум, Символ), !,      % собираем новый текст

```

```

удалить_коммент(Остаток,Аккум1,Текст1).
удалить_коммент("",Текст,Текст).           % останов рекурсии

удалить_строку(Текст,Остаток) :-  

    frontchar(Текст,Символ,Остаток),  

    (Символ='\'n';Символ='\'r'),!.          % конец строки
удалить_строку(Текст,Остаток) :-  

    fronttoken(Текст,_,Текст1),!,  

    удалить_строку(Текст1,Остаток).
удалить_строку(Текст1,Остаток).
удалить_строку(_, "").                     % конец текста

удалить_строки(Текст,Остаток) :-  

    hasPrefix(Текст,"*/",Остаток),!.        % конец блочного комментария
удалить_строки(Текст,Остаток) :-  

    fronttoken(Текст,_,Текст1),             % удаляем блочный комментарий
    удалить_строки(Текст1,Остаток).

run() :-  

    Текст =      % Текст определен как строка с префиксом @  

@"implement main  

/*Это имплементация  

Visual Prolog класса*/  

open core, console, string  

%Перечень открытых классов",  

    (удалить_коммент(Текст,"",Текст1),  

     write(Текст1),!;  

     write("Блочный комментарий не закрыт")),  

     _ = readline().  

end implement main  

goal  

    console::run(main::run).

```

Задание 34.25. Разработать программу удаления всех целых и вещественных чисел из текста.

Задание 34.26. Разработать программу удаления из текста HTML-тегов.

Задание 34.27. Разработать программу подсчета количества цифр, которые встречаются в тексте в числах или как отдельные цифры.

Задание 34.28. Разработать программу расчета частотного словаря текста.

ПРИМЕР 34.12. Разработаем предикат, который будет преобразовывать русский текст в английские буквы. Иногда эта задача, т. е. запись слов одного языка буквами другого языка, называется *транслитерацией*. Основой транслитерации служит база данных замен буквосочетаний одного языка буквосочетаниями другого. Факты базы данных упорядочены по уменьшению длины русских буквосочетаний. Для решения этой задачи надо на каждом витке цикла заменять очередное буквосочетание русского текста буквами английского языка, присоединять эти английские бук-

вы к аккумулятору английского текста и рекурсивно вызывать предикат транслитерации до тех пор, пока русский текст не сократится до пустой строки.

```

implement main
    open core, console, string
class facts
    замена : (string Рус, string Анг) .
class predicates
    транслитер : (string Рус, string Анг [out]) .
clauses
    транслитер("", "") :- !.
    транслитер(Русс, concat(АнгБук, Англ)) :-  

        замена(РусБук, АнгБук),  

        hasPrefix(Русс, РусБук, Русс1), !,  

        транслитер(Русс1, Англ).
    транслитер(Русс, concat(Знак, Англ)) :-  

        front(Русс, 1, Знак, Русс1),  

        транслитер(Русс1, Англ).

замена("кс", "x") .  замена("дж", "j") .

замена("а", "a") .  замена("б", "b") .  замена("в", "v") .
замена("г", "g") .  замена("д", "d") .  замена("е", "e") .
замена("е", "jo") .  замена("ж", "zh") .  замена("з", "z") .
замена("и", "i") .  замена("й", "j") .  замена("к", "k") .
замена("л", "l") .  замена("м", "m") .  замена("н", "n") .
замена("о", "o") .  замена("п", "p") .  замена("р", "r") .
замена("с", "s") .  замена("т", "t") .  замена("у", "u") .
замена("ф", "f") .  замена("х", "kh") .  замена("ц", "ts") .
замена("ч", "ch") .  замена("ш", "sh") .  замена("щ", "sch") .
замена("ъ", "") .  замена("ы", "y") .  замена("ъ", "") .
замена("э", "y") .  замена("ю", "ju") .  замена("я", "ya") .

run() :-
    S="Жезл и скипетр Ксеркса!",
    транслитер(toLowerCase(S), АнглТекст),
    write(АнглТекст),
    _ = readchar().
end implement main
goal
    console::run(main:::run) .

```

Рекурсивный предикат транслитер(РусТекст, АнглТекст) состоит из трех предложений. Первое предложение описывает условие останова рекурсии:

```
транслитер("", "") :- !.
```

Это означает, что английский текст равен пустой строке тогда, когда русский текст равен пустой строке. Если это условие выполняется, то происходит отсечение остальных правил этого предиката. Иначе — правила выполняются.

Второе предложение является рекурсией и описывает тот случай, когда какое-либо буквосочетание из базы данных заменяется префиксом русского текста. Этот префикс заменяется соответствующим английским буквосочетанием.

Третье предложение описывает случай, когда очередной символ русского текста не найден в базе замен. Тогда этот символ без изменения конкатенируется слева к английскому тексту. Третьим предложением обрабатываются знаки пунктуации, пробелы, табуляции, символы перевода в начало строки и на следующую строку.

Задание 34.29. Разработать программу транслитерации русского текста английскими буквами с учетом регистра букв исходного текста.

Задание 34.30. Разработать программу преобразования произвольного русского числительного в число. Например: "сто двадцать пять" → 125. Преобразование ограничить диапазоном чисел от нуля до 999.

Задание 34.31. Разработать программу преобразования произвольного целого числа в числительное. Например: 125 → "сто двадцать пять". Преобразование ограничить диапазоном чисел от нуля до 999. Справка: такая задача имеет самостоятельное название Untaco.

Задание 34.32. Разработать программу вычисления средней длины слов входного текста, вводимого из файла. Длина слов измеряется количеством букв. На примере текстов большого размера определить среднюю длину слов русского, английского и других языков.

Задание 34.33. Разработать программу вычисления средней длины предложений входного текста, вводимого из файла. Длина предложений измеряется количеством слов. Разделителем предложений пусть является точка, восклицательный или вопросительный знаки. На примере текстов большого размера определить среднюю длину предложений русского, английского и других языков.

ГЛАВА 35



Рекурсивные правила на списках

Эта глава освещает практические приемы использования рекурсии для обработки списков.

35.1. Списки

ПРИМЕР 35.1. Разработать предикат циклического сдвига списка вправо на k элементов. Эффективным способом является разделение списка с помощью предиката `list::split` на две части: `Left` и `Right` и соединение их в обратном порядке: `Right` и `Left`. Позиция разделения — разность между длиной списка и величиной сдвига. Интересным моментом является случай, когда величина сдвига больше длины списка. Теоретически список во время сдвига не единожды может вернуться в свое исходное состояние. Мы выйдем из этой ситуации просто — вычислим новую величину сдвига как остаток от целочисленного деления величины сдвига на длину списка.

```
implement main
    open core, console, list
class predicates
    ror:(unsigned Length, E*, E* [out]). % списки полиморфны
clauses
    ror(_,[],[]):=!. % если исходный список пуст, то выходной тоже пуст
    ror(0,L,L):=!. % если сдвиг равен нулю, то списки одинаковы
    ror(K,L,append(Right,Left)):-      % присоединение правого к левому
        Len = length(L),                % длина списка
        K1 = K mod Len,                 % новая величина сдвига
        split(Len-K1,L,Left,Right).     % расщепление в позиции Len-K1

    run() :-
        ror(8,[1,2,3,4,5],L),
        write(L),                      % [3,4,5,1,2]
        _ = readchar().
end implement main
```

```
goal
  console:::run(main:::run).
```

Предикат `ror` выполняется за $2n$ шагов, где n — длина списка.

ПРИМЕР 35.2. Есть некоторое множество монет разного достоинства. Необходимо получить все варианты разделения этих монет на два подмножества так, чтобы сумма в них была одинаковой. Вначале стоит найти общую сумму всех монет — чтобы выяснить, делится она на два без остатка или нет. Если делится, то попытаться набрать ровно половину суммы из исходного списка. Оставшиеся монеты следуют поместить во второе подмножество. Множество монет будем представлять списком целых чисел.

```
implement main
  open core, console, list
class predicates
  p:(unsigned*,unsigned*,unsigned*) nondeterm (i,o,o).
  separ:(integer,unsigned*,unsigned*,unsigned*)
    nondeterm (i,i,o,o)(i,i,[o|o],o).
clauses
  p(L,Q,W) :-
    Sum = list:::fold(L, { (A,B)=A+B}, 0), % вычисление суммы
    Sum mod 2 = 0, % сумма четная
    S = Sum div 2, % вычисление половины суммы
    separ(S,L,Q,W). % разделение списка L на два списка Q и W

  separ(0,L,[],L).
  separ(S,[A|L],[A|Q],W) :- % помещаем монету в список Q
    S1=S-A,S1>=0, % сумма списка Q
    separ(S1,L,Q,W).
  separ(S,[A|L],[B|Q],[A|W]) :- % помещаем монету в список W
    separ(S,L,[B|Q],W), A>B. % условие запрета повторов

  run() :-
    p([3,5,2,4,3,2,1],Q,W),
    write(Q),nl,
    fail;
    _ = readchar().
end implement main
goal
  console:::run(main:::run).
```

ПРИМЕР 35.3. Определить статистику элементов списка, т. е. вычислить, сколько раз каждый элемент встречается в списке. Результат будем собирать в списке кортежей `tuple(E,N)`, где E — элемент исходного списка, а N — число вхождений этого элемента в исходный список. Алгоритм заключается в отделении от списка головы и подсчете числа ее вхождений в хвост списка. Во время этого подсчета все найденные дубликаты мы будем из списка удалять. В случае, когда список содержит

много дубликатов, это приведет к сокращению длины списка на каждом витке цикла и, следовательно, к повышению эффективности программы в целом.

В программе используется высокоуровневый предикат:

```
filter(L, { (C) :- C=E }, Pos, Neg)
```

с помощью которого мы разделяем список на два списка: `Pos` и `Neg`. Длина списка `Pos` является числом вхождений заданного элемента `E` в список. Список `Neg` будет содержать элементы, отличные от `E`.

```
implement main
    open core, console, list
class predicates
    stat:(E*, tuple{E,positive}* [out]) .
clauses
    stat([E|L], [tuple(E,N)|T]) :-
        filter(L, { (C) :- C=E }, Pos, Neg), % разделение списка L
                                                % на Pos и Neg
        N = length(Pos)+1, !,
        stat(Neg, T).
    stat([], []).

run() :-
    stat([5,6,7,6,5], L),
    write(L),
    _ = readchar().
end implement main
goal
    console:::run(main:::run).
```

Недостатком этой программы являются излишние операции, связанные с разделением списка на два списка, `Pos` и `Neg`, и подсчетом длины списка `Pos`. Этого недостатка можно избежать, если предикат `filter` написать вручную с нужной функциональностью. Наш собственный предикат `del(E,L,L1,1,N)` удалит все вхождения заданного элемента `E` в список `L` и возвратит число этих удалений `N`. Кроме того, предикат `del/5` вернет остаток списка `L1`, образовавшийся после удалений:

```
implement main
    open core, console
class predicates
    st:(E*, tuple{E,positive}* [out]) .
    del:(E, E*, E*[out], positive, positive [out]) .
clauses
    st([E|L], [tuple(E,N)|T]) :- del(E, L, L1, 1, N), st(L1, T).
    st([], []).

    del(E, [E|L], L1, A, N) :- !, del(E, L, L1, A+1, N). % удаление, инкремент
    del(E, [X|L], [X|L1], A, N) :- del(E, L, L1, A, N).
    del(_, [], [], N, N).
```

```

run() :-  

    st([5,6,7,6,5],L2),  

    write(L2),nl,  

    _ = readchar().  

end implement main  

goal  

    console::run(main:::run).

```

ПРИМЕР 35.4. Разработать предикат для перестановки k -го и m -го элемента заданного списка. Существует простое решение этой задачи, основанное на использовании предикатов класса `list`. С помощью функции `Item = nth(Index, List)` можно поочередно прочитать k -й и m -й элементы списка, а потом с помощью предиката `setNth(Index, List, Item, NewList)` заменить элементы в указанных позициях.

```

implement main  

    open core, console  

clauses  

run() :-  

    K=1, M=4,  

    List=[0,7,11,2],           % исходный список  

    ItemK = list::nth(K-1,List), % читаем k-й элемент  

    ItemM = list::nth(M-1,List), % читаем m-й элемент  

    list::setNth(K-1,List,ItemM,List1), % m-й элемент на место k-го  

    list::setNth(M-1,List1,ItemK,List2), % k-й элемент на место m-го  

    write(List2),  

    _ = readchar().  

end implement main  

goal  

    console::run(main:::run).

```

Однако такая программа не эффективна, ибо требует $2(k+m)$ шагов решения. Этую задачу можно решить всего за m шагов. Для этого надо описать предикат, который будет собирать новый список из элементов исходного. При встрече k -го элемента его место в новом списке займет свободная переменная, и построение списка продолжится. При встрече m -го элемента его место в новом списке займет k -й элемент, а m -й элемент будет унифицирован с той свободной переменной, которая заняла место k -го элемента.

В этой программе есть только один тонкий момент — где мы будем хранить k -й элемент и ту свободную переменную, которая займет место m -го элемента в новом списке? Ответ прост: k -й элемент мы будем передавать в голове того списка, из которого строим новый список, а свободную переменную будем передавать в голове нового списка. Это видно во втором правиле. k -й элемент обозначен переменной `A`, свободная переменная обозначена переменной `E`. Эти две переменные всегда передаются из исходного и нового списков головы правила в рекурсивный вызов.

Программа не проверяет выход за пределы длины списка индексов k и m , а также ограничивает индексы k и m соотношением $k < m$. Если эти условия не выполняют-

ся, то предикат замены будет неуспешен. Списки в программе объявлены полиморфными:

```

implement main
    open core, console
class predicates
    обмен:(unsigned, unsigned, E*, E*) determ (i,i,i,o) (i,i,i,[o|o]) .
clauses
    обмен(1,2,[A,B|C],[B,A|C]) :- ! .
    обмен(1,M,[A,B|C],[E,B|D]) :- !, обмен(1,M-1,[A|C],[E|D]) .
    обмен(K,M,[A|B],[A|C]) :- обмен(K-1,M-1,B,C) .

run() :- 
    if обмен(1,4,[0,7,11,2],X) then write(X)
        else write("данные некорректны")
    end if,
    _ = readchar().
end implement main
goal
    console::run(main:::run) .

```

Первое правило предиката `обмен` записывает k -й элемент на место m -го, а также унифицирует m -й элемент со свободной переменной, которая установлена на место k -го элемента.

Второе правило описывает запись свободной переменной на место k -го элемента в выходной список. Это правило также обеспечивает передачу свободной переменной до m -го элемента с целью их унификации. Кроме того, оно транспортирует k -й элемент к m -позиции.

Третье правило строит выходной список из элементов входного до тех пор, пока не будет встречен k -й элемент.

Обратите внимание на используемый шаблон потока параметров `(i,i,i,[o|o])`. Он говорит о том, что выходной список мы представляем в виде неизвестной головы и неизвестного хвоста. Этот шаблон описывает поток параметров второго правила предиката, в котором и совершается «тайство» передачи свободной переменной во время продвижения по списку от k -й до m -й позиций.

Задание 35.1. Написать функцию `Elem=nth(N,Elem*)`, возвращающую значение N -го элемента `Elem` списка `Elem*`.

Задание 35.2. Написать предикат `set(Elem,N,L,L1)`, возвращающий новый список `L1`, который образован из входного списка `L` путем замены в нем значения N -го элемента новым значением `Elem`.

Задание 35.3. Написать детерминированную функцию `Elem=last(Elem*)`, возвращающую последний элемент `Elem` списка `Elem*`.

Задание 35.4. Написать предикат `rol(K,Elem*,Elem* [out])`, осуществляющий циклический сдвиг списка `Elem*` влево на K элементов.

Задание 35.5. Написать предикат `max(Elem*, Max)`, который находит максимальный элемент заданного списка.

Задание 35.6. Разработать функцию, вычисляющую среднее арифметическое элементов заданного списка чисел.

Задание 35.7. Разработать функцию, выделяющую из исходного списка подсписок, содержащий элементы исходного списка, начиная с элемента, стоящего в позиции N , и заканчивая элементом, занимающим позицию $N + K$. N и K — аргументы функции.

Задание 35.8. Разработать функцию, возвращающую 0, если в исходном списке из нулей и единиц больше нулей, и 1 — в противном случае.

Задание 35.9. Разработать полиморфный предикат `слияние(L1, L2, L3)`, который производит операцию объединения двух упорядоченных по убыванию списков `L1, L2` в третий, упорядоченный по убыванию список `L3`.

Задание 35.10. Дан список координат точек на плоскости, например: `[[1,3], [2,5], [3,8], [1,1], [1,0], [2,-2]]`. Необходимо вывести координаты самой ближней к центру координат точки и самой дальней от центра.

35.2. Математика

ПРИМЕР 35.5. Найдем произведение тех элементов исходного списка, которые больше нуля. Для этого опишем функцию `mul/2`, которая возвращает `true`, если такие элементы присутствуют в списке. Вторым аргументом функция вернет величину произведения. Если искомых элементов в списке нет, то функция вернет значение `false`.

Функция `mul/2` ищет в списке первый элемент, который больше нуля, и вызывает предикат `mult/3`, который накапливает во втором аргументе произведение.

```
implement main
    open core, console
class predicates
    mul:(integer*,integer) -> boolean determ (i,o).
    mult:(integer*,integer,integer) procedure (i,i,o).
clauses
    mul([X|L],Z)=true():-X>0,! ,mult(L,X,Z).           % элемент найден
    mul([_|L],Z)=mul(L,Z) :-!.                           % поиск элемента
    mul(_,0)=false().                                     % здесь список пуст

    mult([X|L],A,M):-X>0,Y=A*X,! ,mult(L,Y,M);
                    mult(L,A,M).

    mult([],M,M).

run():- L=[-1,-2,3,4,-5],
      ( mul(L,M)=true(), write(M));                      % решение найдено
```

```

        write("Множители не найдены") ), !, % решение не найдено
        _ = readchar() .
end implement main
goal
    console::run(main:::run) .

```

Эту же задачу можно решить, используя функцию `fold` из класса `list`. Она собирает элементы списка в одну величину по правилу, определяемому программистом. Правило зададим анонимной функцией:

```
{ (A,B)=C :- A>0, C=A*B, ! ; C=B }
```

от двух переменных `A` и `B`. Переменная `B` является своеобразным аккумулятором, в который последовательно собираются значения всех элементов списка, представляемых переменной `A`. Поэтому если `A>0`, то анонимная функция имеет значение `A*B`, иначе анонимная функция не изменяется, сохраняя значение `B` для очередного витка цикла. Таким образом, предикат, решающий задачу из примера 35.5, имеет вид:

```

run() :- L=[-1,-2,3,4,-5],
Z=list:::fold(L, { (A,B)=C :- A>0, C=A*B, ! ; C=B }, 1),
write(Z),
_ = readchar() .

```

Задание 35.11. Разработать программу, вычисляющую выражение:

$$\sqrt[n]{x_1^2 + x_2^2 + \dots + x_n^2},$$

где значения x являются элементами списка длины n . Список вводится с клавиатуры в виде терма.

Задание 35.12. Разработать программу для вычисления списка произведений пар чисел (a,b) , занимающих одинаковые позиции в списках `A` и `B`, вводимых с клавиатуры.

Задание 35.13. Разработать программу для нахождения всех комбинаций, сочетаний и перестановок элементов списка. Список вводится с клавиатуры.

35.3. Треугольник Паскаля

ПРИМЕР 35.6. Построим треугольник Паскаля. Первый (верхний) ряд содержит один элемент — единицу. Второй ряд содержит два таких элемента. Числа третьего и последующих рядов являются суммой двух чисел, стоящих над ними.

Пусть начальный элемент треугольника является входным параметром. Функция, вычисляющая новый элемент, пусть также задается как параметр. Программа содержит функцию `map2`, которая, аналогично функции `map` класса `list`, преобразует один список в другой. Особенностью функции `map2` является то, что элементы каждого нового списка строятся на основе функции `F` от двух смежных элементов исходного списка. Функция `map2` содержит два параметра: исходный список и функ-

цию преобразования F. Функция map2 преобразует два элемента исходного списка X и Y в новый элемент F(X, Y) и делает рекурсивный вызов, в который передается исходный список без первого элемента X. Когда в списке остается меньше двух элементов, то функция map2 вернет этот список без изменений.

Предикат pascal/4 содержит четыре аргумента:

- счетчик рядов треугольника;
- пустой список, с которого начнется построение рядов треугольника;
- функцию преобразования, с помощью которой вычисляются числа треугольника Паскаля;
- собственно треугольник Паскаля, представленный списком списков.

Кроме этого программа позволяет задавать любой числовой тип данных для элементов треугольника Паскаля.

```

implement main
    open core, console
domains
    ppp{A} = (unsigned,A*,function{A,A,A}, A** [out]).
class predicates
    pascal : ppp{integer}.
    map2 : (A* List, function{A,A,A}) -> A*.
clauses
    map2([X,Y|List],F) = [F(X,Y)|map2([Y|List],F)] :- !.
    map2(L,_) = L.

    pascal(N,L,F,[1|L1]|T) :- N>0,L1=map2(L,F),!,
                                pascal(N-1,[1|L1],F,T).
    pascal(_,_,[],[]).

run() :-
    pascal(6,[],{(A,B)=A+B},L),           % высота треугольника = 6
    list::forall(L,{(B):-write(B),nl}),    % вывод треугольника
    _ = readchar().
end implement main
goal
    console::run(main:::run).
```

Задание 35.14. Ускорить программу из примера 35.6 используя тот факт, что треугольник Паскаля симметричен относительно вертикальной оси. В связи с этим строить можно только левую половину треугольника, а правую половину выводить в порядке, реверсном относительно левой половины.

Задание 35.15. Разработать программу построения прямоугольника в виде списка списков одинаковой длины. Функция построения очередного ряда должна вычислять его элементы на основе функции от трех смежных элементов предыдущего ряда. Вид функции задайте самостоятельно.

Задание 35.16. Построить функцию преобразования одного списка в другой список той же длины. Функция должна вычислять элементы нового списка на основе функции от n элементов исходного списка.

35.4. Длинная арифметика

Длинная арифметика работает с числами произвольной длины. Такая арифметика используется в тех задачах, где требуется обрабатывать целые числа большой разрядности, для представления которых разрядности одного регистра процессора не хватает. Для совершения операций над длинными числами их разделяют на ряд чисел малой разрядности, можно даже вплоть до цифр, и совершают операции над малыми числами. Потом эти малые числа соединяют в результат большой разрядности.

ПРИМЕР 35.7. Разработаем программу для сложения двух длинных чисел A и B , которые вводятся с клавиатуры в виде строк произвольной длины. Для проверки того, что строки содержат только десятичные цифры, используется предикат `hasDecimalDigits`. Так как обработка чисел начинается с младших разрядов, представим числовую строку в виде реверсивного списка цифр с помощью анонимной функции:

```
Φ = { (Γ) = reverse (map (toCharList (Γ)) , { (Π) = getCharValue (Π) - 48 } ) }
```

которая преобразует исходную строку Γ в список символов посредством функции `toCharList`, после чего список символов преобразуется в список цифр с помощью функции `map`. Для преобразования символа в цифру надо получить код этого символа функцией `getCharValue` и вычесть из него число 48. Это объясняется тем, что коды цифр находятся в диапазоне от 48 до 57. Число 48 является кодом цифры 0, число 49 — кодом цифры 1 и т. д. Завершает рассматриваемое преобразование функция `reverse`, обращающая список цифр для того, чтобы младшие цифры числа стали головой списка.

Вызов предиката `сумма(Φ(A), Φ(B), Сумма)` преобразует введенные с клавиатуры длинные числа в реверсивные списки цифр, после чего выполняет их поразрядное суммирование. Основную работу совершает предикат `сумма1(L1, L2, P, L, S)`. Переменные L_1 и L_2 являются слагаемыми, представленными списками цифр в обратном порядке. Переменная P — перенос в старший разряд, переменная L — список-аккумулятор, в котором по одному разряду собирается сумма, и, наконец, S — возвращаемый параметр, который вытаскивает сумму со дна рекурсии.

Суммирование разрядов X и Y двух чисел и значения переноса P выполняет предикат `сумма_разрядов(X, Y, P, P1, Z)`. Выходными параметрами является сумма Z и новый перенос $P1$.

```
implement main
    open core, console, string, list
domains
    перенос = [0..1].
```

```

class predicates
    сумма:(integer*,integer*,integer*) procedure (i,i,o).
    сумма1:(integer*,integer*,перенос,integer*,integer*)
        determ (i,i,i,i,o).
    сумма_разрядов:(integer,integer,перенос,перенос,integer)
        procedure (i,i,i,o,o).

clauses
    сумма(L1,L2,S):-сумма1(L1,L2,0,[],S),!; S=[].

    сумма1([X|L1],[Y|L2],P,L,S):-!, % X+Y+P
        сумма_разрядов(X,Y,P,P1,Z),
        сумма1(L1,L2,P1,[Z|L],S).

    сумма1([X|L1],[],P,L,S):-!, % X+P
        сумма_разрядов(X,0,P,P1,Z),
        сумма1(L1,[],P1,[Z|L],S).

    сумма1([], [Y|L2], P, L, S):-!, % Y+P
        сумма_разрядов(0,Y,P,P1,Z),
        сумма1([],L2,P1,[Z|L],S).

    сумма1([],[],1,L,[1|L]):-!. % P=1
    сумма1([],[],0,L,L). % P=0

    сумма_разрядов(X,Y,P,1,Z-10):-X+Y+P=Z, Z>9,!. % P=1
    сумма_разрядов(X,Y,P,0,X+Y+P). % P=0

run() :-
    A = readline(), hasDecimalDigits(A), % ввод первого числа
    B = readline(), hasDecimalDigits(B), % ввод второго числа
    Φ={ (Γ)=reverse(map(toCharList(Γ), { (Π)=getCharValue(Π)-48 } )) },
    сумма(Φ(A),Φ(B),Сумма),
    write(concatList(map(Сумма, { (Π)=toString(Π) }))), nl,
    clearinput(),
    _ = readchar(),!;
    _ = readchar().

end implement main
goal
    console::run(main::run).

```

Задание 35.17. Разработать программу для вычитания длинного числа *Б* из длинного числа *А*, которые вводятся с клавиатуры в виде строк произвольной длины.

Задание 35.18. Разработать программу для умножения длинного числа *А* на одноразрядное число *Б*, которые вводятся с клавиатуры.

Задание 35.19. Разработать программу для умножения двух длинных чисел *А* и *Б*, которые вводятся с клавиатуры в виде строк произвольной длины.

Задание 35.20. Разработать программу для деления длинного числа *А* на одноразрядное число *Б*, которые вводятся с клавиатуры.

Задание 35.21. Разработать программу для деления длинного числа А на длинное число Б, которые вводятся с клавиатуры в виде строк произвольной длины.

35.5. Преобразователь чисел из одной системы счисления в другую

Задание 35.22. Разработать программу преобразования чисел из одной системы счисления в другую, в которой алфавиты входной и выходной систем счисления задаются пользователем в виде списка символов:

```
convert(char* InAlpha, string N, char*OutAlpha, string N1),
```

где:

- InAlpha — алфавит системы счисления, в которой представлено число N;
- OutAlpha — алфавит системы счисления, в которую надо перевести число N;
- N1 — выходное число.

35.6. Монеты

ПРИМЕР 35.8. Дан список чисел — например, [50, 40, 15, 5, 3, 2, 1], которые будем понимать как достоинства монет. Количество монет каждого достоинства ничем не ограничено, т. е. мы можем выбрать, например, несколько монет достоинством 40 (копеек). Необходимо набрать заданную сумму из минимального числа монет разного достоинства. Предполагается, что разнообразие монет позволяет составить требуемую сумму. Поиск решений осуществляется предикатом:

вариант (Монеты, Сумма, Вариант, Аккумулятор, Счетчик)

Переменная Монеты является списком монет разного достоинства и требуемой суммой Сумма. Переменная Вариант — выходная и возвращает набор монет в виде списка кортежей tuple (Монета, Количество), где переменная Количество является числом монет одинакового достоинства Монета. Переменная Счетчик возвращает количество монет в этом наборе. В переменной Аккумулятор на каждом витке цикла накапливается количество монет в наборе.

Монеты собираются не по одной, а по группам одинакового достоинства. Для этого служат операции целочисленного деления.

Алгоритм поиска использует факт best (Набор, Количество) для хранения текущего оптимального решения. Если вновь полученное решение лучше текущего оптимального, то новое решение заменяет текущее и становится новым лидером. Когда перебор вариантов завершается, то текущий лидер объявляется оптимальным решением задачи в целом.

```
implement main
    open core, console
domains
    tuple = tuple(integer, integer).
```

```

class facts
    best:(tuple*,integer) determ.
class predicates
    варианты:(integer*,integer,tuple* [out],integer [out]) determ.
    вариант:(integer*,integer,tuple*,integer,integer)
        nondeterm (i,i,o,i,o).
    сохранить:(tuple*,integer) determ.
    печать : (tuple*,integer).

clauses
    варианты(Монеты, Сумма, _, _) :-  

        вариант(Монеты, Сумма, Вариант, 0, Счетчик),  

        сохранить(Вариант, Счетчик),  

        fail.  

    варианты(_, _, Вариант, Счетчик) :- best(Вариант, Счетчик).  
  

    сохранить(Вариант, Счетчик) :-  

        best(Вариант0, Счетчик0), Счетчик < Счетчик0, !,  

        retract(best(Вариант0, Счетчик0)),  

        assert(best(Вариант, Счетчик));  

        not(best(_, _)), assert(best(Вариант, Счетчик)).  
  

    вариант([M|Монеты], Сумма, [tuple(M,N)|Набор], I, Счетчик) :-  

        N = Сумма div M, N > 0,           % взять N монет достоинством M копеек  

        Ост = Сумма mod M,  

        вариант(Монеты, Ост, Набор, I+N, Счетчик).  

    вариант([M|Монеты], Сумма, [tuple(M,N)|Набор], I, Счетчик) :-  

        N = (Сумма div M) - 1, N > 0,           % взять N-1 монет  

        Ост = (Сумма mod M) + M,  

        вариант(Монеты, Ост, Набор, I+N, Счетчик).  

    вариант([_|Монеты], Сумма, Набор, I, Счетчик) :- % эту монету не брать  

        вариант(Монеты, Сумма, Набор, I, Счетчик).  

    вариант(_, 0, [], Счетчик, Счетчик).           % нужная сумма набрана  
  

    печать([tuple(M,Ч)|B], K) :- write(M, "коп. - ", Ч, "шт."), nl, !,  

        печать(B, K).  

    печать(_, K) :- write("Всего - ", K, "шт."), nl.  
  

run() :-  

    варианты([50,40,15,5,3,2,1], 175, Набор, Счетчик),  

    печать(Набор, Счетчик),  

    _ = readchar(), !;  

    _ = readchar().  

end implement main  

goal  

    console::run(main::run).

```

Задание 35.23. Дано множество вещественных чисел. Составить такое подмножество, сумма элементов которого наименее всего отличается от заданного вещественного числа.

35.7. Домино

ПРИМЕР 35.9. Необходимо набрать все возможные цепочки из заданного множества костяшек домино. Воспользуемся стратегией «образовать и проверить». Одну костяшку домино будем представлять доменом $d(n, n)$, где n — это доминочная цифра от 1 до 6. Множество костяшек домино представим в программе в виде списка этих доменов. С помощью предиката образовать ($L, L1$) будем строить из списка L все возможные списки $L1$ и проверять их соответствие правилам совмещения костяшек домино. Вспомогательный предикат $vzять(L, I, L1)$ выбирает из списка L одну из костяшек I и возвращает список оставшихся костяшек $L1$. Также предикат $vzять(L, I, L1)$, беря костяшку из исходного списка костей, может ее переворачивать с помощью правила $vзять([d(A, B) | L], d(B, A), L)$.

```

implement main
    open core, console
domains
    n = [0..6].           % номер на костяшке
    d = d(n,n).          % костяшка домино
class predicates
    vzять: (d*, d, d*) nondeterm (i, o, o).      % взять костяшку
    образовать: (d*, d*) nondeterm (i, o).        % соединить костяшки
                                                % в цепочку
    проверить: (d*) determ.    % проверить правильность
                                % составления цепочки
clauses
    vzять ([X|L], X, L).
    vzять ([d(A,B)|L], d(B,A), L).
    vzять ([X|L], A, [X|L1]) :- vzять (L, A, L1).

    образовать (L, [I|Is]) :- vzять (L, I, L1), образовать (L1, Is).
    образовать ([] , []).

    проверить ([d(_,B), d(B,A) | L]) :- !, проверить ([d(B,A) | L]).
    проверить ([_]) :- !.

run() :-
    L = [d(2,4), d(3,4), d(5,2), d(6,3), d(3,5), d(1,6), d(1,0)],
    образовать (L, L1), проверить (L1),
    write(L1), nl, fail;
    _ = readchar().
end implement main
goal
    console::run(main::run).
```

Конечно, эта программа тратит много времени для генерации всех возможных комбинаций костей. Можно было бы сразу написать «умный» алгоритм, который бы выбирал очередную кость так, чтобы она подошла к уже собранной цепи. В приме-

ре 35.9 мы просто продемонстрировали стратегию «образовать и проверить». По большому счету эта стратегия используется, когда более эффективный алгоритм неизвестен, или для его поиска надо затратить много времени и сил.

Задание 35.24. Разработать программу построения цепочек домино так, чтобы выбор очередной кости домино для присоединения цепи был не случайным, как в примере 35.5, а соответствовал правилам домино.

35.8. Ребус

В разд. 33.4 мы рассматривали простые варианты решения ребусов о сложении чисел, представленных словами. Сейчас мы рассмотрим, каким образом привнесение списков может упростить решение этой задачи.

ПРИМЕР 35.10. Для решения ребуса SEND+MORE=MONEY используем предикат `взять(L, X, L1)`, который вырезает из списка `L` цифру `X` и возвращает остаток списка `L1`. Исходный список цифр `L` будем хранить в константе:

цифры : цифра* = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0].

Достоинством такого подхода является отказ от явной проверки, что выбиралась цифра ранее не встречалась, ибо цифры вырезаются из списка, и остальные цифры будут выбираться из его остатка, следовательно, повторов не будет.

В программе использован домен десятичных цифр:

```
domains
    цифра = [0..9].
```

В принципе, вместо него можно было бы использовать тип `unsigned`, но хорошим тоном считается ограничивать типы данных доменами, чтобы все возможные ошибки алгоритма выявлять на этапе компиляции.

Кроме этого пример демонстрирует объявление и реализацию предиката `ребус` с режимом `failure`. При этом режиме предикат `ребус` никогда не будет успешным. Но это ему не мешает выводить на экран все найденные решения. После перебора всех решений выполнится вторая ветвь предиката `run()`, и программа будет успешно завершена.

```
implement main
    open core, console, string
domains
    цифра = [0..9].
constants
    цифры:цифра* = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0].
class predicates
    взять:(цифра*, цифра, цифра*) nondeterm (i,o,o).
    ребус:() failure.
clauses
    взять ([X|L], X, L).
    взять ([X|L], A, [X|L1]) :- взять (L, A, L1).
```

```

ребус() :-  

    взять(цифры, S, Остаток1), взять(Остаток1, E, Остаток2),  

    взять(Остаток2, N, Остаток3), взять(Остаток3, D, Остаток4),  

    взять(Остаток4, M, Остаток5), взять(Остаток5, O, Остаток6),  

    взять(Остаток6, R, Остаток7), взять(Остаток7, Y, _),  

    Число1=1000*S+100*E+10*N+D,  

    Число2=1000*M+100*O+10*R+E,  

    Сумма=Число1+Число2,  

    Слово = format("%u%u%u%u", M, O, N, E, Y),  

    Сумма = toTerm(Слово),  

    write(Число1, "+", Число2, "=", Сумма), nl, fail.  
  

run() :-  

    ребус();  

    _ = readchar().  

end implement main  

goal  

    console::run(main:::run).

```

ПРИМЕР 35.11. Представленная в предыдущем примере программа опять-таки соответствует неэффективной стратегии «образовать и проверить». Для ускорения поиска всех решений воспользуемся другим подходом. Складывать слова будем столбиком, начиная от младших разрядов к старшим. Именно так человек и поступает, складывая числа карандашом на бумаге.

Поразрядное сложение выполняет предикат:

```
тест(X, Y, Z, P, P1).
```

где X является суммой разрядов Y и Z и переноса P, а P1 — перенос в следующий разряд. Предикат сложения назван тест потому, что он не вычисляет сумму, а проверяет равенство $X=Y+Z+P$, когда нет переноса в следующий разряд, и равенство $X=Y+Z+P-10$, когда перенос есть.

Сложение SEND+MORE=MONEY начинается с младших разрядов и нулевого значения переноса $D+E+0=Y$. Поэтому вначале выбираются цифры только для букв D, E и Y. Они проверяются предикатом тест, который в добавок вычисляет перенос в следующий разряд P. Следующие разряды N, R, E проверяются равенством, в котором учтен перенос P: $N+R+P=E$. И так повторяется до самых старших разрядов.

```

implement main
    open core, console, string
constants
    цифры:integer* = [1,2,3,4,5,6,7,8,9,0].
class predicates
    взять:(integer*,integer,integer*) nondeterm (i,o,o).
    ребус(): failure.
    тест:(integer,integer,integer,integer,integer)
        determ (i,i,i,i,o) (i,i,i,i,i).

```

```
clauses
```

```
взять ([X|L], X, L) .  
взять ([X|L], A, [X|L1]) :- взять (L, A, L1) .
```

```
ребус () :-
```

```
    взять (цифры, D, Остаток1), взять (Остаток1, E, Остаток2),  
    взять (Остаток2, Y, Остаток3), тест (Y, D, E, 0, P1),  
    взять (Остаток3, N, Остаток4), взять (Остаток4, R, Остаток5),  
    тест (E, N, R, P1, P2),  
    взять (Остаток5, O, Остаток6), тест (N, E, O, P2, P3),  
    взять (Остаток6, S, Остаток7), взять (Остаток7, M, _),  
    тест (O, S, M, P3, M),  
    write (S, E, N, D, "+", M, O, R, E, "=", M, O, N, E, Y), nl, fail.
```

```
тест (X, Y, Z, P, 0) :- X = Y + Z + P, !.  
тест (X, Y, Z, P, 1) :- X = Y + Z + P - 10.
```

```
run () :-
```

```
    ребус ();  
    _ = readchar () .
```

```
end implement main
```

```
goal
```

```
    console:::run (main:::run) .
```

При запуске этой программы можно заметить более быстрый перебор по сравнению с программой из примера 35.10.

Задание 35.25. Разработать программу для решения ребуса с вычитанием — например, MONEY-SEND=MORE. Обратите внимание, что вместо переноса надо использовать заем.

35.9. Ребус с произвольными словами

ПРИМЕР 35.12. Разработаем программу, решающую ребусы типа SEND+MORE= =MONEY с двумя слагаемыми и суммой, но с произвольными словами. Для этого воспользуемся поразрядным сложением с переносом слов Слагаемое1 и Слагаемое2 для получения суммы Сумма. Эту работу будет выполнять рекурсивный предикат:

```
ребус (Цифры, Перенос, Униф, Слагаемое1, Слагаемое2, Сумма)
```

В переменной Цифры будем хранить список используемых десятичных цифр. При вызове предиката эта переменная содержит все 10 цифр от 0 до 9. По мере связывания букв, составляющих слова, с цифрами этого списка, последний будет сокращаться.

С помощью переменной Перенос будем передавать значение переноса в старший разряд. Переменная Униф будет содержать цифровые значения букв. Для этого используется список пар вида п(Буква, Цифра). Название переменной напоминает о том факте, что список содержит значения только тех букв, которые связаны циф-

ровыми значениями. При вызове предиката переменная `Униф` равна пустому списку. По мере связывания букв с цифрами этот список будет наполняться. Перед тем как назначить очередной букве цифру, необходимо проверить список `Униф`. Если эта буква в списке есть, то цифровое значение буквы берется из списка. Если буквы нет, то цифровое значение назначается из списка свободных цифр `Цифры`. Описанную работу выполняет предикат:

```
взять(Цифры, Униф, Слово, Ц, Остаток, Униф1, Цифры1)
```

В нем слово `Слово` разделяется на последнюю букву и оставшуюся левую часть слова `Остаток`. В переменной `Ц` возвращается цифровое значение буквы. Также возвращается новый список `Униф1` и остаток списка свободных цифр `Цифры1`.

Предикат `послед` служит для разделения слова на левую часть и последнюю букву. Предикат `униф` ищет цифровое значение буквы в списке `Униф`. Предикат `цифра` берет произвольную цифру из списка свободных цифр и кроме этого возвращает остаток списка свободных цифр. Предикат `сумма` осуществляет проверку сложения с переносом и аналогичен предикату `тест` из предыдущего примера:

```
implement main
    open core, console, string
domains
    пара = π(string,integer).
class predicates
    ребус: (integer*,integer,пара*,string,string,string) nondeterm.
    взять: (integer*,пара*,string,integer,string,пара*,integer*)
        nondeterm (i,i,i,o,o,o,o).
    послед: (string,string,string) determ (i,o,o).
    униф: (string,пара*,integer [out]) nondeterm.
    цифра: (integer*,integer [out],integer* [out]) nondeterm.
    сумма: (integer,integer,integer,integer,integer [out]) determ.
clauses
    ребус(_,0,Униф,"","", "") :- write(Униф), nl.
    ребус(Цифры,Перенос,Униф,Слово1,Слово2,Слово3) :-
        Слово3<>"",
        взять(Цифры,Униф,Слово1,Ц1,С1,Униф1,Цифры1),
        взять(Цифры1,Униф1,Слово2,Ц2,С2,Униф2,Цифры2),
        взять(Цифры2,Униф2,Слово3,Ц3,С3,Униф3,Цифры3),
        сумма(Перенос,Ц1,Ц2,Ц3,Перенос1),
        ребус(Цифры3,Перенос1,Униф3,С1,С2,С3).

    взять(Цифры,Униф,Слово,Ц,С1,Униф,Цифры) :-
        послед(Слово,Б,С1),
        униф(Б,Униф,Ц), !.
    взять(Цифры,Униф,Слово,Ц,С1, [π(Б,Ц) | Униф], Цифры1) :-
        послед(Слово,Б,С1),
        цифра(Цифры,Ц,Цифры1).
    взять(Цифры,Униф,"",0,"",Униф,Цифры) :- !.
```

```

послед(Слово, Б, Слово1) :-  

    Длина=length(Слово),  

    Длина>0,  

    front(Слово, Длина-1, Слово1, Б).  
  

униф(Б, [п(Б, Ц) | _], Ц) :- ! .  

 униф(Б, [_ | Униф], Ц) :- униф(Б, Униф, Ц) .  
  

цифра( [Ц | Цифры] , Ц, Цифры) .  

цифра( [Ц1 | Цифры] , Ц, [Ц1 | Остаток]) :- цифра(Цифры, Ц, Остаток) .  
  

сумма(Перенос, Ц1, Ц2, Ц3, 0) :- Перенос+Ц1+Ц2=Ц3, !.  

сумма(Перенос, Ц1, Ц2, Ц3, 1) :- Перенос+Ц1+Ц2=Ц3+10.  
  

run() :-  

    ребус([1,2,3,4,5,6,7,8,9,0], 0, [], "SEND", "MORE", "MONEY"),  

    fail;  

    _ = readchar().  

end implement main  

goal  

    console:::run(main:::run).

```

Запуск программы с целью SEND+MORE=MONEY приводит к 25 известным решениям в виде списка пар $\text{п}(\text{Буква}, \text{Цифра})$. Программа выводит 25 списков связанных букв. Вот так выглядит первое решение:

```
[п("M", 0), п("S", 7), п("O", 8), п("R", 2), п("N", 3), п("Y", 6),
 п("E", 5), п("D", 1)]
```

Его надо понимать так: буква "M"=0, буква "S"=7 и т. п.

Задание 35.26. Усовершенствовать вывод решения в примере 35.12 так, чтобы вместо списка связанных букв выводилось равенство. Например, первое решение должно выводиться в виде: $2817+368=3185$.

35.10. Ребус с произвольными словами произвольной длины

До сих пор мы решали только те ребусы, у которых количество различных букв не превышало 10. То есть, все расчеты мы вели в десятичной системе счисления. Если количество различных букв превышает 10, то надо вести вычисления в системе счисления, основание которой не меньше числа различных букв.

ПРИМЕР 35.13. Усовершенствуем программу из примера 35.12, заменив в предикате `сумма` десятичную систему счисления системой с заданным основанием. Для этого добавим в предикат `сумма` еще один аргумент `Основание`, через который будем передавать предикату `основание` системы счисления:

```
сумма(Основание, Перенос, Ц1, Ц2, Ц3, Перенос1)
```

Все остальные аргументы остались такими же, как и в примере 35.12.

Кроме этого добавим в предикат ребус еще один аргумент Рез, с помощью которого будем возвращать результат работы предиката со дна рекурсии в точку его вызова. И последнее: так как основание системы счисления может быть любым числом N, то цифры системы счисления будем представлять десятичными числами от нуля до N-1. Конечно, их можно представлять и буквами, но при достаточно большом N даже всех букв может не хватить. Генерация алфавита системы счисления производится коллектором списка с функцией cIterate:

```
Цифры = [Ц | Ц=std::cIterate(Основание)]
```

Для проверки программы решим ребус SEND+MORE=MONEY в десятичной системе счисления, т. к. все 25 решений мы уже знаем:

```
implement main
    open core, console, string
domains
    пара = π(string,integer).
class predicates
    ребус: (integer, integer*, integer, пара*, пара*, string, string, string)
        nondeterm (i, i, i, i, o, i, i, i).
    взять: (integer*, пара*, string, integer, string, пара*, integer*)
        nondeterm (i, i, i, o, o, o, o).
    послед: (string, string, string) determ (i, o, o).
    униф: (string, пара*, integer [out]) nondeterm.
    цифра: (integer*, integer [out], integer* [out]) nondeterm.
    сумма: (integer, integer, integer, integer, integer, integer)
        determ (i, i, i, i, i, o).
clauses
    ребус (_ , _ , 0, Униф, Униф, "", "", "").
    ребус (Основание, Цифры, Перенос, Униф, Рез, Слово1, Слово2, Слово3) :-  

        Слово3>"",
        взять (Цифры, Униф, Слово1, Ц1, С1, Униф1, Цифры1),
        взять (Цифры1, Униф1, Слово2, Ц2, С2, Униф2, Цифры2),
        взять (Цифры2, Униф2, Слово3, Ц3, С3, Униф3, Цифры3),
        сумма (Основание, Перенос, Ц1, Ц2, Ц3, Перенос1),
        ребус (Основание, Цифры3, Перенос1, Униф3, Рез, С1, С2, С3).

    взять (Цифры, Униф, Слово, Ц, С1, Униф, Цифры) :-  

        послед (Слово, Б, С1),
        униф (Б, Униф, Ц), !.
    взять (Цифры, Униф, Слово, Ц, С1, [π (Б, Ц) | Униф], Цифры1) :-  

        послед (Слово, Б, С1),
        цифра (Цифры, Ц, Цифры1).
    взять (Цифры, Униф, "", 0, "", Униф, Цифры) .

    послед (Слово, Б, С1) :-  

        Длина=length (Слово),
```

```

Длина>0,
front(Слово, Длина-1, С1, Б).

униф(Б, [п(Б, Ц) | _], Ц):-! .
 униф(Б, [_ | Униф], Ц):- униф(Б, Униф, Ц) .

цифра([Ц | Цифры], Ц, Цифры) .
цифра([Ц1 | Цифры], Ц, [Ц1 | Остаток]):- цифра(Цифры, Ц, Остаток) .

сумма(_, Перенос, Ц1, Ц2, Ц3, 0):- Перенос+Ц1+Ц2=Ц3, ! .
сумма(Основ, Перенос, Ц1, Ц2, Ц3, 1):- Перенос+Ц1+Ц2=Ц3+Основ.

run():-
    Основание=10,
    Цифры=[Ц| Ц=std::cIterate(Основание)],
    ребус(Основание, Цифры, 0, [], Рез, "SEND", "MORE", "MONEY"),
    write(Рез), nl, fail;
    _ = readchar().
end implement main
goal
    console::run(main:::run).

```

Испытания программы, конечно же, показывают все 25 вариантов решений. В восьмеричной системе счисления этот ребус имеет только три решения, ибо возможностей перебора на восьми цифрах меньше, чем на десяти. Первое восьмеричное решение имеет вид:

```
[п("M", 0), п("S", 5), п("O", 6), п("R", 2), п("N", 1), п("Y", 7),
 п("E", 3), п("D", 4)]
```

или, в более привычной записи: 5314+0623=06137.

Задание 35.27. Модифицировать программу так, чтобы алфавит системы счисления задавался пользователем в виде еще одного входного аргумента предиката ребус.

Задание 35.28. Разработать программу для решения ребуса, в котором складываются не два, а три слова.

Задание 35.29. Разработать программу для решения ребуса, в котором слова не складываются, а умножаются.

35.11. Расстановка N ферзей на доске $N \times N$

Рассмотрим обобщение известной задачи о восьми ферзях, которые надо расставить на шахматной доске так, чтобы они не били друг друга. Пусть размер доски $N \times N$ и количество ферзей N . Необходимо найти все решения этой задачи, а также общее количество решений.

Один из лучших способов заключается в вычеркивании вертикали, горизонтали и двух диагоналей после установки ферзя на клетку. Следующий ферзь следует

устанавливать в одну из тех клеток, которые остались невычеркнутыми. Этот способ дает быстрое сокращение вариантов поиска. Однако от его конкретной реализации зависит эффективность программы в целом.

Пусть клетка шахматного поля задается кортежем (tuple) координат `tuple(X, Y)`. На первый взгляд кажется, что для описания шахматного поля $N \times N$ стоит сгенерировать список таких кортежей длиной N^2 . Тогда после установки ферзя в одну из клеток поля из списка кортежей надо удалить все кортежи, которые задают клетки, лежащие на вертикали, горизонтали и двух диагоналях. Однако такой прямой подход к делу многократно увеличивает требуемые вычислительные ресурсы: время и память.

ПРИМЕР 35.14. Воспользуемся другим подходом, согласно которому мы не будем генерировать список кортежей пустых клеток. Мы будем заполнять список только тех клеток, которые занимают ферзи. Вначале этот список будет пуст, а в конце алгоритма его длина будет равна n . Пустые же клетки доски будут задаваться координатой x , которая в цикле будет декрементироваться от n до 1, и координатой y , которая на каждом витке цикла будет вырезаться из списка оставшихся горизонталей `Rows = [1..n]`. Список горизонталей формируется перед циклом с помощью предиката `fromTo` из класса `std`:

```
Rows = [I || I=std::fromTo(1, N)]
```

Суть подхода заключается в том, что для каждой координаты x будет подбираться из списка `Rows` такая координата y , которые вместе (x, y) определят клетку, не находящуюся под боем.

Остался самый тонкий момент — как определить под боем клетка или нет? Итак, мы имеем клетку с координатами x, y и список клеток T , в которых уже стоят ферзи. Так как координаты x и y в цикле не повторяются, то нам не надо проверять горизонтали и вертикали. Остались диагонали. Из курса геометрии известны условия, которым удовлетворяют координаты точек, лежащих на одной прямой. Две точки, лежащие на одной диагонали (идущей слева направо, снизу вверх), например, точка $(2, 1)$ и точка $(5, 4)$ имеют одинаковую разность соответствующих координат: $5 - 2 = 4 - 1$. Точки, лежащие на других диагоналях (идущих слева направо, сверху вниз), — например, точка $(1, 7)$ и точка $(3, 5)$ имеют одинаковую сумму своих координат: $1 + 7 = 3 + 5$. На основании этих двух признаков можно описать такой тест:

```
test(X, Y, [tuple(A, B) | Tuples]) :- A - X <> B - Y, A + B <> X + Y, !,  
                                test(X, Y, Tuples).  
test(_, _, []).
```

В этом teste для клетки с координатами x, y проверяются все клетки `tuple(A, B)`, на которые уже поставлены ферзи. Если ни один из ферзей не бьет клетку x, y , то на эту клетку выставляется очередной ферзь и список `Rows` лишается числа y . Иначе — эта клетка бракуется, и для координаты x будет выбираться другая координата y из списка `Rows`.

Цикл поиска решения в программе описывается предикатом `f`. В нем аргумент x является номером столбца и в рекурсивном вызове декрементируется. Аргумент

Rows содержит список номеров свободных рядов, т. е. тех, на которых не стоит ферзь. Из этого списка и выбирается координата Y. Третий аргумент T является списком клеток, в которых уже стоят ферзи. Последний аргумент Res — выходной. На дне рекурсии этот аргумент унифицируется со списком T для того, чтобы вернуть решение в точку вызова предиката f. Кроме этого на дне рекурсии инкрементируется счетчик решений:

```
f(X, Rows, T, Res) :-  
    cut(Rows, Y, Rows1),      % вырезаем Y из списка Rows, остаток — Rows1  
    test(X, Y, T),           % под боем ли клетка (X, Y)?  
    f(X-1, Rows1, [tuple(X, Y) | T], Res).  
f(0, _, Res, Res) :- i:=i+1. % решение найдено, инкремент счетчика
```

Для поиска всех решений этой задачи в предикате run организован цикл с откатом:

```
implement main  
open core, console, std  
domains  
c = [0...].          % домен неотрицательных чисел  
class facts  
i : unsigned := 0.   % счетчик решений  
class predicates  
f:(c,c*,tuple{c,c}*,tuple{c,c}*) nondeterm (i,i,i,o).  
cut:(c*,c,c*) nondeterm (i,o,o).  
test:(c,c,tuple{c,c}*) determ.  
clauses  
f(X, Rows, T, Res) :- cut(Rows, Y, Rows1),  
                     test(X, Y, T),  
                     f(X-1, Rows1, [tuple(X, Y) | T], Res).  
f(0, _, Res, Res) :- i:=i+1.  
  
cut([E|L], E, L).  
cut([E1|L], E, [E1|L1]) :- cut(L, E, L1).  
  
test(X, Y, [tuple(A, B) | Tuples]) :- A-X<>B-Y, A+B<>X+Y, !,  
                                     test(X, Y, Tuples).  
test(_, _, []).  
  
run() :-  
    N=8,                      % размер поля и количество ферзей  
    Rows = [I || I=fromTo(1, N)], % список номеров рядов  
    f(N, Rows, [], Res),       % поиск одного решения  
    write(Res), nl,            % вывод решения  
    fail;                     % откат для поиска остальных решений  
    write(i),                 % общее количество решений  
    _ = readchar().  
end implement main  
goal  
console:::run(main:::run).
```

ПРИМЕР 35.15. Нет предела совершенству. Поэтому попробуем усовершенствовать нашу программу с точки зрения лаконичности и эффективности. Во-первых, давайте упростим условия, проверяемые предикатом `test`:

`A-X<>B-Y, A+B<>X+Y`

Для этого неравенство сумм заменим неравенством разностей, сделав соответствующие переносы переменных:

`A-X<>B-Y, A-X<>Y-B`

Сейчас видно, что левые части неравенств одинаковые. Поэтому заменим два неравенства одним:

`abs(A-X) <> abs(Y-B)`

Здесь функция `abs` из класса `math` определяет модуль числа.

Во-вторых, заменим предикат `test` высокоуровневым предикатом `all` из класса `list`, который проверяет то, что все элементы списка удовлетворяют какому-либо условию:

```
implement main
    open core, console, std, math, list
domains
    c = [0...].
class facts
    i:unsigned := 0.
class predicates
    f:(c,c*,tuple{c,c}*,tuple{c,c}*) nondeterm (i,i,i,o).
    cut:(c*,c,c*) nondeterm (i,o,o).
clauses
    f(X,Rows,T,Res) :- cut(Rows,Y,Rows1),
                      all(T,{(tuple(A,B)) :- abs(A-X)<>abs(Y-B)}),
                      f(X-1,Rows1,[tuple(X,Y)|T],Res).
    f(0,_,Res,Res) :- i:=i+1.

    cut([E|L],E,L).
    cut([E1|L],E,[E1|L1]) :- cut(L,E,L1).

    run() :- N=8,
             Rows = [I || I=fromTo(1,N)],
             f(N,Rows,[],Res),
             write(Res),nl,
             fail;
             write(i),
             _ = readchar().
end implement main
goal
    console::run(main::run).
```

Эта программа находит все решения для небольших досок. Если при $N=8$ все 92 решения находятся практически мгновенно, то для доски с $N=13$ на поиск всех решений будет затрачено уже несколько секунд. Дальнейшее совершенствование алгоритма связано с нахождением и использованием эвристик, ограничивающих перебор.

Задание 35.30. Усовершенствовать программу из примера 35.15 для четных N . Для этого использовать эвристику, гласящую, что на досках с четным размером стороны главная (черная) и дополнительная (белая) диагонали должны оставаться всегда пустыми.

Задание 35.31. Разработать программу поиска всех таких расстановок N ладей на доске $N \times N$, в которых они не бьют друг друга.

Задание 35.32. Разработать программу поиска пути коня на доске $N \times N$, проходящего через все клетки доски и побывавшего в каждой не более одного раза.

Задание 35.33. Разработать программу поиска цикла коня на доске $N \times N$, проходящего через все клетки доски и побывавшего в каждой не более одного раза. Цикл отличается от пути тем, что конь должен вернуться в ту клетку, из которой вышел.

35.12. Кувшины

ПРИМЕР 35.16. Даны два кувшина емкостью A и B , причем $A > B$. Кувшины можно наполнять из бесконечной емкости и выливать обратно. Жидкость можно переливать из одного кувшина в другой. Найдем все возможные количества жидкости, которые можно получить выполнением указанных операций.

Рассмотрим один способ решения, который состоит в последовательном наполнении большего кувшина и получения разных остатков посредством переливания жидкости в меньший кувшин и его опорожнения в том случае, когда он становится полным. Алгоритм описывается четырьмя рекурсивными правилами предиката `searchAll/4`:

- если меньший кувшин неполный, то наполнить его из большего;
- если в большем кувшине жидкости мало, то вылить все в меньший кувшин;
- если меньший кувшин полный, то опорожнить его;
- если больший кувшин пустой, то наполнить его.

Условием останова рекурсии является либо нахождение всех решений или появление признака зацикливания. Число всех решений равно $A-2$, т. к. числа B и A мы исключим из множества возможных решений. Признаком зацикливания примем получение в большем кувшине объема воды, равного объему меньшего кувшина.

Получение всех $A-2$ решений возможно тогда, когда числа A и B являются взаимно простыми. В противном случае будет получена только часть решений, и алгоритм остановится по признаку зацикливания.

Очередным решением будет являться количество жидкости в большем кувшине, отличное от чисел 0, B, A и отличающееся от ранее найденных решений. Список решений накапливается в ходе углубления в рекурсию.

Объемы кувшинов будем вводить с клавиатуры. При выводе на экран последовательности переливаний в скобках (A,B) будем указывать количество жидкости в кувшинах до и после переливания. Причем первый аргумент кортежа — это количество жидкости в большем кувшине, второй аргумент — количество жидкости в меньшем кувшине.

```

implement main
    open core, console, list
class facts
    a:unsigned := erroneous. % больший кувшин
    b:unsigned := erroneous. % меньший кувшин
class predicates
    searchAll:(unsigned,unsigned,unsigned,unsigned*) determ.
    test:(unsigned,unsigned*,unsigned* [out]). 
clauses
    searchAll(_,A,_ ,L) :- length(L)=Z,
        ( Z = a-2 ,
        writef("Найдены все % решений: % \n",a-2,L);
        A = b,
        writef("Поиск завершен. Найдено % решений: % \n",Z,L)),! .
    searchAll(I,A,B,L) :- B<b, A>b-B, ! , A1=A-b+B,
        writef("% . (% ,%) Наполнить меньший из большего (% ,%) \n",I,A,B,A1,b),
        test(A1,L,L1),
        searchAll(I+1,A1,b,L1) .
    searchAll(I,A,0,L) :- A<b, ! ,
        writef("% . (% ,0) Перелить все из большего в меньший (0,% )\n",I,A,A),
        searchAll(I+1,0,A,L) .
    searchAll(I,0,B,L) :- ! ,
        writef("% . (0,% ) Наполнить больший кувшин (% ,%) \n",I,B,a,B),
        searchAll(I+1,a,B,L) .
    searchAll(I,A,b,L) :-
        writef("% . (% ,%) Меньший кувшин опорожнить (% ,0)\n",I,A,b,A),
        searchAll(I+1,A,0,L) .

    test(A,L,L) :- (A=0;A=a;A=b),!. % 0,a,b - не решения
    test(A,[A|L],[A|L]) :- !. % это решение найдено ранее
    test(A,[B|L],[B|L1]) :- test(A,L,L1) .
    test(A,[],[A]) :- writef("*** Новое решение: % ***\n",A) .

run() :-
    write("Емкость первого кувшина: "), a:=read(),
    write("Емкость второго кувшина: "), b:=read(),
    ( (a>b; a<b, X=b, b:=a, a:=X) , searchAll(1,a,0,[]),
    write("Емкости должны быть разными") ),!,
```

```

clearInput(),
_ = readchar().
end implement main
goal
  console:::run(main:::run).

```

Задание 35.34. Разработать программу для нахождения всех возможных решений задачи о кувшинах согласно другому алгоритму, который заключается в последовательном наполнении меньшего кувшина и получения разных остатков посредством переливания жидкости в больший кувшин и его опорожнения в том случае, когда он становится полным. Сравнить результаты с результатами предыдущего примера.

Задание 35.35. Разработать программу нахождения последовательности переливаний для получения заданного количества жидкости. Емкости кувшинов и требуемое количество жидкости вводятся с клавиатуры.

Задание 35.36. Разработать программу нахождения последовательности переливаний для получения заданного количества жидкости за минимальное число шагов. Емкости кувшинов и требуемое количество жидкости вводятся с клавиатуры.

35.13. Строки. Баланс скобок

ПРИМЕР 35.17. В произвольном тексте определить баланс круглых скобок. Это означает, что каждая левая скобка должна где-то правее иметь свою правую пару. Способ решения основан на изменении счетчика в зависимости от очередного символа строки. Левая скобка инкрементирует счетчик, правая — декрементирует, остальные символы не изменяют значение счетчика. Если счетчик становится отрицательным, то баланс нарушен, т. к. правых скобок обнаружено больше, чем левых. Если по достижении последнего символа строки счетчик равен нулю, то баланс соблюден, иначе — баланс нарушен, т. к. левых скобок в тексте больше, чем правых.

```

implement main
  open core, console, string
class predicates
  b:(string, integer) determ.
clauses
  b("",0):-!.          % баланс есть
  b(_,_):-I<0,fail.    % баланс нарушен
  b(S,I):-frontchar(S,C,R), % отделяем очередной символ строки
    ( C='(',!,b(R,I+1); % если скобка левая, то инкремент счетчика
    C=')',!,b(R,I-1); % если скобка правая, то декремент счетчика
    b(R,I) ).           % не скобка

run():-
  if b("qwe(rty(111))",0) then write("да")
    else write("нет") end if,
    _ = readchar().
end implement main

```

```
goal
  console::run(main::run) .
```

Задание 35.37. В произвольном тексте определить баланс круглых, квадратных и фигурных скобок. Подсказка: способ со счетчиком тут не годится. Здесь надо использовать стек в виде списка, в котором запоминать последовательность скобок для того, чтобы сравнивать очередной символ строки с первым элементом этого списка. Если очередной символ совпадает с ожидаемой правой скобкой, то освобождать список от этой скобки. Баланс скобок соблюдается, если программа завершится с пустым списком скобок.

35.14. Строки. Транслитерация

ПРИМЕР 35.18. Преобразовать произвольный русский текст так, чтобы все русские буквы были заменены соответствующими по произношению английскими буквами или буквосочетаниями. Правила замены русской буквы английским буквосочетанием описаны с помощью фактов замена/2. Вначале мы произведем преобразование текста в список символов с помощью функции `toCharList` из класса `string`. Далее зададим анонимную функцию `F`, которая русскую букву заменяет английским эквивалентом, а любой другой символ текста оставляет как есть. Замену всех символов текста произведем в коллекторе списка `L1`, в котором посредством предиката `V in L` будем получать очередные символы `V` из списка `L`, а с помощью анонимной функции `F` получать замену очередного символа `V` новым символом `F(V)`. На завершающем этапе с помощью функции `concatList` соберем результирующую строку из списка `L1`.

```
implement main
  open core, console, string, list
  class facts
    замена: (char Рyc, string Анг) .
  clauses
    замена('а', "а") .  замена('б', "b") .  замена('в', "v") .
    замена('г', "g") .  замена('д', "d") .  замена('е', "e") .
    замена('ё', "jo") .  замена('ж', "zh") .  замена('з', "z") .
    замена('и', "i") .  замена('й', "j") .  замена('к', "k") .
    замена('л', "l") .  замена('м', "m") .  замена('н', "n") .
    замена('о', "o") .  замена('п', "p") .  замена('р', "r") .
    замена('с', "s") .  замена('т', "t") .  замена('у', "u") .
    замена('ф', "f") .  замена('х', "kh") .  замена('ц', "ts") .
    замена('ч', "ch") .  замена('ш', "sh") .  замена('щ', "sch") .
    замена('ъ', "") .  замена('ы', "y") .  замена('ъ', "") .
    замена('э', "y") .  замена('ю', "ju") .  замена('я', "ya") .

  run() :-
    S = "жезл и скипетр!",
    L=toCharList(toLowerCase(S)),      % получение списка символов
    F={ (V)=Q :- замена(V,Q), !; Q=charToString(V) },
```

```

L1=[F(V) || V in L],
        %получение выходного списка
write(concatList(L1)),
        % получение строки из списка
_ = readchar().
end implement main
goal
    console::run(main:::run).

```

Задание 35.38. Разработать программу транслитерации, используя предикат `map`.

Задание 35.39. Разработать программу транслитерации, в которой предусмотреть правила замены русских буквосочетаний английскими буквами — например, русское буквосочетание «кс» можно заменить английской буквой «х», а русское буквосочетание «дж» — английской буквой «г».

35.15. Списки.

Поиск в ширину подсписка в списке

На первый взгляд, поиск подсписка может показаться тривиальной задачей. Что может быть проще проверки того, что подсписок является префиксом списка. Если подсписок не префикс, то можно провести проверку со следующей позиции списка:

```

find(F,L,N,N) :- prefix(F,L).
find(F,[_|L],I,N) :- find(F,L,I+1,N).

```

```

prefix([],_).
prefix([A|F],[A|L]) :- prefix(F,L).

```

Здесь предикат `find(F,L,I,N)` осуществляет поиск подсписка `F`, начиная с позиции `I` списка `L`. Предикат `prefix(F,L)` проверяет то, что список `F` является префиксом списка `L`.

Однако за кажущейся простотой скрывается неэффективность алгоритма. В наихудшем случае такому алгоритму потребуется $M \times N$ шагов, где M — длина списка, N — длина подсписка.

Существует ряд более эффективных алгоритмов. Рассмотрим один из них, суть которого заключается в следующем. Пусть первый элемент списка совпадает с первым элементом подсписка. Тогда берем второй элемент списка и сравниваем его не только со вторым элементом подсписка, но также и с первым элементом подсписка. Если оба сравнения удачны, то у нас будет уже два кандидата на решение. Действуя так дальше, мы получим множество кандидатов, которые будем хранить в списке. Конечно, некоторые кандидаты, а может, и все, будут удаляться из этого списка в случае, когда их элементы не совпадают с образцом. Если какой-либо из подсписков становится пустым, то это означает, что он найден в списке. Такой алгоритм параллельного поиска завершается в худшем случае за M шагов, не считая накладных расходов, связанных с поиском каждого элемента списка среди всех текущих кандидатов.

ПРИМЕР 35.19. Рассмотрим реализацию описанного алгоритма. Пусть предикат $p(L, F, N)$ определяет позицию N подсписка F в списке L . Предикат $f(L, F, FF, I, N)$ описывает внешний цикл. Первое правило этого предиката определяет наличие пустого списка в списке кандидатов. При этом $I=N$ является позицией конца подсписка в списке. Для определения позиции начала подсписка следует из N вычесть длину подсписка. Этим занимается предикат p .

Кроме того, предикат $f(L, F, FF, I, N)$ для каждого очередного элемента A списка L осуществляет проверку его наличия в голове каждого элемента списка FF . При этом в конец списка FF добавляется исходный подсписок F . Таким образом, в списке FF кандидаты на решение упорядочены так, что самый последний элемент является полным подсписком, а самый первый элемент — это подсписок, который дальше всех проверяется. Поэтому именно в начале списка FF и может находиться пустой список, что является признаком очередного найденного решения.

```

implement main
  open core, console, list
class predicates
  p: (E*, E*, positive) nondeterm (i, i, o).
  f: (E*, E*, E**, positive, positive) nondeterm (i, i, i, i, o).
  g: (E, E**, E** [out]). 
clauses
  p(L, F, N-Len) :- Len = length(F), f(L, F, [], 1, N).

  f(_, _, [[ ]| _], N, N).          % решение найдено
  f([A| L], F, FF, I, N) :-        % отделяем голову от списка
    g(A, append(FF, [F]), FF1),   % добавляем подсписок F в конец списка FF
    f(L, F, FF1, I+1, N).          % инкремент счетчика текущей позиции

  g(A, [[ A | F ] | FF], [F | FF1]) :-!, g(A, FF, FF1). % тест кандидата успешен
  g(A, [[ ] | FF], FF1) :-!, g(A, FF, FF1).      % удаление пустого списка из FF
  g(A, [ _ | FF], FF1) :-!, g(A, FF, FF1).      % удаление неуспешного кандидата
  g(_, [], []).                         % список кандидатов пуст

run() :-
  p([1,1,7,1,1,1,3], [1,1], N),
  write(N, " "),
  fail;
  _ = readchar().
end implement main
goal
  console:::run(main:::run).
```

Для исходного списка $[1,1,7,1,1,1,3]$ и искомого списка $[1,1]$ программа выведет три позиции вхождения: 1, 4 и 5. Эффективность этой программы проявляется особенно на тех примерах, в которых длина искомого списка велика, и/или начало искомого списка часто повторяется в том списке, в котором производится поиск.

Задание 35.40. Разработать программу поиска позиций всех палиндромов в заданном списке. Палиндром — это список, который равен своему реверсивному списку. Например, в список [2,1,3,1,1,3,5] входит три палиндрома. Первый палиндром [1,3,1] начинается со второй позиции, второй палиндром [3,1,1,3] начинается с третьей позиции и третий палиндром [1,1] начинается с четвертой позиции. Обратите внимание, что третий палиндром входит в состав второго палиндрома.

Задание 35.41. Разработать программу поиска позиции самого длинного палиндрома в заданном списке. Если таких палиндромов несколько, вывести позиции всех.



ГЛАВА 36

Внутренняя база данных

В Прологе реализован механизм доступа к фактам базы данных реляционного типа, загруженным из текстового файла целиком в память. Поэтому такая БД и называется *внутренней*, чтобы отличать ее от внешних баз данных, — т. е. тех, которые расположены на носителе и доступ к которым организован через тот или иной программный интерфейс. Внутренние базы данных выгодно использовать в приложениях, работающих на локальной машине и оперирующих с количеством фактов, не превышающим сотни тысяч.

Внутренние базы данных хранятся в текстовых файлах и могут быть модифицированы как программно, так и вручную в любом текстовом редакторе.

Операции по работе с внутренней базой данных описаны в главе 7.

36.1. Лексика русского языка

ПРИМЕР 36.1. Создадим файл БД со словами русского языка. Источником для этого может послужить любой файл с перечнем русских слов, взятый из Интернета, например, по ссылкам:

<http://www.speakrus.ru/dict/>

<http://www.artint.ru/projects/frqlist.php>

В папке Primer36_1\Exe электронного архива, сопровождающего книгу (см. *приложение 11*), имеется файл Словарь.txt. Используя этот файл, скачанный из указанных ресурсов, мы будем строить дальнейшие проекты.

Лучший способ преобразования текстового файла в БД — это чтение файла по строкам и сохранение каждого слова строки в виде факта БД. Таким способом можно обрабатывать файлы достаточно больших размеров, не используя при этом много памяти.

Объявим факт `w(string)`, в котором будем сохранять каждое слово. Также объявим счетчик слов в виде факта-переменной `i`. Откроем в программе файл Словарь.txt для чтения. В цикле `r(In)` будем построчно читать содержимое файла и сохранять каждую строку, содержащую только буквы, в факте `w(string)`, инкрементируя при этом

счетчик слов *i*. Когда будет достигнут конец файла, т. е. предикат `In:endOfStream()` завершится успехом, то всю базу фактов мы сохраним с помощью предиката `file::save("Словарь.w", слова, false())` в файле Словарь.w в кодировке ANSI.

```

implement main
    open core, console, string
class facts - слова
    w:(string).           % факт для сохранения слов
    i:unsigned :=0.         % счетчик слов
class predicates
    r:(inputStream).
clauses
    r(In):-In:endOfStream(),!.          % останов цикла, если конец файла
    r(In):-S=In:readLine(),             % чтение строки из потока In
        if hasAlpha(S) then            % слово содержит только буквы
            assert(w(S)),              % сохранение слова в виде факта БД
            i:=i+1                      % инкремент счетчика слов
        end if,
        r(In).                         % рекурсивное продолжение чтения
run():-
    In = inputStream_file::openFile8("Словарь.txt"),      % новый поток
    r(In),                                              % преобразование входного текста в БД
    file::save("Словарь.w", слова, false()),   % сохраняем БД в файл
    write("Всего слов = ",i),
    _ = readLine().
end implement main
goal
    console::run(main:::run).

```

Убедитесь, что после выполнения программы в папке Exe появился файл Словарь.w, содержащий слова русского языка в кодировке ANSI. В конце файла должен идти факт, указывающий количество слов в словаре.

ПРИМЕР 36.2. На основе полученной БД слов Словарь.w составим линейный кроссворд, длина которого вводится с клавиатуры. Линейный кроссворд — это список слов, у которых каждое следующее слово начинается на последнюю букву предыдущего. Например: яблоко-окурок-кот-телевизор-ребро-оглобля. Длина этого кроссворда равна шести.

Для составления кроссворда напишем предикат `cross(I,A,L)`, где *I* — счетчик цикла, задающий длину кроссворда; *A* — аккумулятор слов, представляющий собой список слов кроссворда в обратном порядке; *L* — готовый список слов кроссворда в прямом порядке *I*. Первые два аргумента являются входными, третий — выходной.

Перед вызовом предиката `cross/3` необходимо определить слово, которое послужит первым словом кроссворда. Кроме этого наложим ограничения на слова кроссворда:

- длина слов должна быть больше 2, чтобы исключить короткие предлоги;
- повторы слов в кроссворде недопустимы.

Для проверки того, что последняя буква текущего слова равна первой букве следующего слова, воспользуемся предикатами из класса `string`:

- `frontchar(S, _, P)` — предикат получения первой буквы `P` слова `S`;
- `P = lastChar(S)` — функция получения последней буквы `P` слова `S`.

Далее представлена программа составления линейного кроссворда длиной 5 слов. Перед запуском убедитесь, что файл БД Словарь.w расположен в папке Exe проекта.

```

implement main
    open core, console, string
class facts - слова
    w: (string).           % слово
    i:unsigned :=0.         % счетчик слов
class predicates
    cross:(unsigned,string*,string*) nondeterm (i,i,o).
clauses
    cross(1,A,list:::reverse(A)).   % останов рекурсии
    cross(I,[S|A],L):-I>0,          % пока счетчик больше нуля
        lastChar(S,_,P),            % P – последняя буква текущего слова
        w(S1),                      % следующее слово
        length(S1)>2,              % длина следующего слова больше 2
        frontChar(S1) = P,           % смежные буквы слов совпадают
        not(S1 in [S|A]),           % повторов слов быть не должно
        cross(I-1,[S1,S|A],L).      % рекурсивное продолжение
run():-file::consult("Словарь.w", слова),
    w(S), length(S)>2,             % длина первого слова кроссворда больше 2
    cross(5,[S],L),                % вызов предиката составления кроссворда
    write(L),nl,fail;              % вывод кроссворда и откат
    _ = readLine().
end implement main
goal
    console::run(main::run).

```

Приведенная программа выводит все возможные варианты линейных кроссвордов, которые можно составить на имеющейся в файле Словарь.w базе слов русского языка.

Задание 36.1. На основе полученной БД слов Словарь.w составить линейный замкнутый кроссворд, в котором первая буква первого слова должна совпадать с последней буквой последнего слова. Длина кроссворда вводится с клавиатуры.

Задание 36.2. На основе полученной БД слов Словарь.w составить линейный замкнутый кроссворд из слов одинаковой длины, в котором первая буква первого слова должна совпадать с последней буквой последнего слова. Длина кроссворда и длина слова вводятся с клавиатуры.

Задание 36.3. На основе полученной БД создать новую БД слов, в которой должны быть только те слова русского языка, которые можно написать английскими буквами. Подсказка: опишите фактами буквы русского алфавита, которые по начертанию

совпадают с буквами английского алфавита. Сохраните в новую БД только те слова из БД Словарь.w, которые состоят из этих заданных букв.

Задание 36.4. На основе полученной БД слов Словарь.w разработать программу для заполнения квадрата, содержащего $n \times n$ клеток словами одинаковой длины n . В каждую клетку помещается одна буква. Например, для $n = 5$, квадрат должен быть заполнен пятью горизонтально расположеными словами и пятью вертикально расположенными словами. Все слова должны быть разными.

Задание 36.5. На основе полученной БД слов Словарь.w разработать программу для составления кроссвордов, которые представляются пересечением слов, расположенных горизонтально и вертикально. Схема расположения слов задается программно на ваше усмотрение.

36.2. Искусственная жизнь

ПРИМЕР 36.3. Модель «Искусственная жизнь» моделирует жизнь поколений гипотетической колонии живых клеток, которые выживают, размножаются или погибают в соответствии со следующими правилами. Клетка выживает, если она имеет двух или трех соседей из восьми возможных. Если у клетки только один сосед или вовсе ни одного, она погибает в изоляции. Если клетка имеет четырех или более соседей, она погибает от перенаселения. В любой пустой позиции, у которой ровно три соседа, в следующем поколении появляется новая клетка.

Ввод исходного состояния колонии осуществляется с клавиатуры. Форму клеточного организма можно задавать не только прямоугольной, но любой произвольной формы и даже в виде нескольких несвязанных форм. Внутри каждой формы могут быть полости произвольного размера. Форма клеточного организма ограничена размерами окна консольного приложения. Живая клетка вводится латинской буквой <x>, место, где может появиться клетка, вводится латинской буквой <o>, пустые полости внутри фигуры и выемки снаружи вводятся пробелами. Признаком конца ввода колонии является ввод пустой строки. Далее каждый шаг жизни клеточного организма происходит по нажатию клавиши <Enter>.

Внизу организма печатается номер шага. На каждом шаге в консольное окно выводится очередное состояние клеточного организма и номер шага. Программа определяет состояние, когда организм перешел в устойчивое состояние и перестал видоизменяться, и сообщает об этом. Также программа определяет смерть организма, т. е. состояние, при котором все клетки мертвы.

Далее представлена программа, сдобренная обширными комментариями:

```
implement main
    open core, console, string
class facts
    % клетка поля в памяти:
    cell: (unsigned Шаг, unsigned16 X, unsigned16 Y, char Клетка).
    % номер строки для вывода текущих сообщений:
    m:unsigned16 := erroneous.
```

```

% флаг изменения клеточного поля на очередном шаге:
change:boolean := erroneous.

class predicates
    % ввод строк клеточного поля, Y - номер строки:
    zone:(unsigned16 Y) procedure.

    % разделение строки на символы и сохранение их в БД:
    row:(unsigned16,unsigned16,string) procedure.

    % сохранение в БД очередного символа строки в виде клетки:
    f:(unsigned16 X,unsigned16 Y,char) procedure.

    % жизнь колонии:
    life:(unsigned) procedure (i).

    % жизнь клетки:
    life_cell:(unsigned,unsigned16,unsigned16,char,unsigned) procedure.

    % количество соседей клетки (X,Y):
    neighbours:(unsigned,unsigned16 X,unsigned16 Y) -> unsigned
        procedure (i,i,i).

    % клетка (X,Y) жива или ее нет:
    n:(unsigned ЕстьНет,unsigned16 X,unsigned16 Y) -> unsigned
        procedure (i,i,i).

    % вывод в окно клеточного организма на шаге N:
    print:(unsigned N) procedure (i).

clauses
    zone(Y):-L = readline(),L>"",row(0,Y,L),!,zone(Y+1);
        m:=Y.      % факт m хранит номер последней введенной строки

    row(X,Y,L):-frontchar(L,C,R),f(X,Y,C),!,row(X+1,Y,R);
        succeed().

    f(_,_, ' '):-!.                                % если пробел, то клетки нет
    f(X,Y,C):-assert(cell(0,X+1,Y+1,C)).       % иначе - сохраняем клетку

    % модификация всех клеток в памяти:
    life(N):-cell(N,X,Y,C),life_cell(N,X,Y,C,neighbours(N,X,Y)),fail.

    % вывод нового поколения клеток в окно:
    life(N):-change=true(),
        print(N+1),                                % организм изменился
        retractall(cell(N,_,_,_)),                 % удаление предыдущего организма
        setLocation(console_native::coord(0,m)),   % курсор на посл.строку
        write("Шаг ",N+1),                         % вывод номера шага жизни
        _ = readchar(),                            % ожидание ввода от пользователя
        change:=false(!),                         % сброс флага перед следующим шагом
        life(N+1).                                % рекурсия

    % организм не изменился:
    life(N):-cell(N+1,_,_,'_x'),!,             % живые есть
        setLocation(console_native::coord(0,m+1)),
        write("Устой-ть").

```

```

% организм не изменился, и живых клеток нет:
life(_):-setLocation(console_native::coord(0,m+1)),
write("Все вымерли") .

% рождение новой клетки на пустом месте, если есть три соседа:
life_cell(N,X,Y,'o',3):-assert(cell(N+1,X,Y,'x')),
change:=true(),! .

% продолжение жизни клетки:
life_cell(N,X,Y,'x',NB):-NB>1,NB<4,
assert(cell(N+1,X,Y,'x')),! .

% смерть клетки
life_cell(N,X,Y,'x',_):-assert(cell(N+1,X,Y,'o')),
change:=true(),! .

% живой клетки не было, и не появилось:
life_cell(N,X,Y,C,_):-assert(cell(N+1,X,Y,C)).

% подсчет числа соседей:
neighbours(N,X,Y) = n(N,X-1,Y-1)+n(N,X,Y-1)+n(N,X+1,Y-1)+  

n(N,X-1,Y)+n(N,X+1,Y)+n(N,X-1,Y+1)+  

n(N,X,Y+1)+n(N,X+1,Y+1).

n(N,X,Y)=1:-cell(N,X,Y,'x'),!.      % клетка (X,Y) жива
n(_,_,_)=0.                          % клетка мертвa или отсутствует

% вывод колонии на экран:
print(N):-cell(N,X,Y,C),
setLocation(console_native::coord(X-1,Y-1)),
write(C),fail;
succeed().

run():-
    zone(0),life(0),
    _ = readchar().
end implement main
goal
    console::run(main::run).

```

После запуска программа ожидает ввода колонии. С помощью пробелов, латинских букв x и o и клавиши <Enter> введем клеточную колонию, подобную, например, такой:

```

оххххх
ххооохоохо
хooooooooохххх
хooooooooоххххх
оохooooooooоо
ххооххоо

```

После ввода пустой строки программа построит модель в памяти компьютера и выведет на экран колонию на первом шаге жизни. Нажимая клавишу <Enter>, мы будем последовательно переводить колонию в новые состояния. На восьмом шаге наступает устойчивое состояние, которое определяется программой, и моделирование развития колонии на этом прекращается:

```
ооооххо
оооооххоох
ооооооооххоооо
оооооооооххооооо
ооооооооооооооооо
оооооооооооооооооо
```

Шаг 8

Устой-ть

Есть также много интересных осцилляторов. Простейший осциллятор представляет собой вертикаль на поле 3×3:

```
охо
охо
охо
```

Жизнь такого организма представляет собой бесконечную смену двух состояний: вертикаль — горизонталь.

Задание 36.6. На основе примера 36.3 исследовать влияние модификации констант правил на жизнь популяции.

Задание 36.7. Разработать модель распространения инфекции в клеточном организме по следующим правилам. Инфицированная клетка каждый промежуток времени с вероятностью 50 % заражает соседнюю клетку. Клетка болеет 6 промежутков времени, затем выздоравливает и 4 промежутка времени остается иммунной.

Задание 36.8. Разработать программу «Естественный отбор», моделирующую жизнь поколений двух враждующих колоний живых клеток на общей территории, которые выживают, размножаются или погибают в соответствии со следующей стратегией. Клетки умирают либо от перенаселения, либо от большого количества чужих клеток, находящихся по соседству. Клетки продолжают жить, если нет перенаселения и чужих клеток рядом мало или нет. Клетки рождаются, когда рядом немного своих клеток и совсем нет чужих. Колонии могут иметь различные константы своей стратегии. Количественные критерии развития колоний задайте самостоятельно.

Задание 36.9. Разработать программу «Симбиоз», моделирующую жизнь поколений двух колоний живых клеток на общей территории, которые выживают, размножаются или погибают согласно правилам, которые вы придумайте сами в соответствии со следующей стратегией. Клетки умирают либо от перенаселения, либо от малого количества чужих клеток, находящихся по соседству. Клетки продолжают жить, если нет перенаселения и чужих клеток рядом достаточно. Клетки рождаются, когда рядом немного своих клеток и немного чужих. Колонии могут иметь различные константы своей стратегии. Количественные критерии развития колоний задайте самостоятельно.

Задание 36.10. Разработать программу «Гармония и равновесие», моделирующую жизнь хищников, травоядных и травы на общей территории. Хищники передвигаются быстро и при встрече пожирают травоядных. Хищники видят на расстояние нескольких клеток только впереди себя и могут целенаправленно бежать к травоядному. Если по прошествии нескольких промежутков времени еда не найдена, то хищник погибает от голода. Травоядные передвигаются медленно и едят подножный корм — траву. Травоядные видят недалеко, впереди себя и в стороны и могут целенаправленно двигаться к траве. Травоядное умирает от голода, если не найдет траву через несколько промежутков времени. Калорийность белковой пищи гораздо выше растительной, поэтому хищники живут без еды дольше травоядных. Трава прорастает на новом месте, если рядом уже есть трава. Трава не умирает сама по себе и может быть только съедена. Количественные критерии развития колоний задайте самостоятельно. Опытным путем определите такие количественные критерии и соотношение хищников и травоядных, при которых наступает равновесие в модели.

36.3. Представление базы данных списком фактов

Для объявления предиката, который работает со списком фактов, надо описать домен, к которому принадлежит список фактов. Основой для описания такого домена является имя базы фактов. Поэтому соответствующая база фактов должна быть именованной. Имя базы фактов является тем доменом, к которому принадлежат все факты этой базы. Для описания домена списка фактов достаточно указать имя базы данных со звездой.

ПРИМЕР 36.4. Пусть имеется база фактов, содержащая название наиболее продаваемого литературного произведения каждого русского автора. Факты базы упорядочены в алфавитном порядке фамилий авторов. Создадим механизм для модификации базы в случае, когда надо обновить число продаж в некоторых фактах или некоторые произведения заменить другими, с большим числом продаж. Причем сортировка по алфавиту должна сохраняться.

Пусть база фактов содержит факты типа:

```
book(string Автор, string Название, unsigned Число_продаж)
```

и носит имя `bestseller`. Определим функцию `tolist`, которая возвращает список фактов базы. Возвращаемый функцией `tolist` список принадлежит домену `bestseller*`, т. к. имя базы фактов — `bestseller`. Функция `tolist` определена с помощью коллектора списков `[book(A, B, C) | book(A, B, C)]`.

Также определим предикат `toDB`, который обновляет базу из списка фактов. Входной список фактов предиката `toDB` принадлежит домену `bestseller*`. Предикат сначала удаляет все факты базы, чтобы не было дубликатов, а потом восстанавливает базу из списка фактов с помощью предиката:

```
forall(L, { (X) :- assert(X) } ).
```

Получить индекс факта, содержащего, например, произведение Лермонтова, из списка L можно с помощью предиката класса list:

```
I = tryGetIndexEq({(A,book(A, _, _))}, "Лермонтов", L).
```

Записать в позицию I списка L новый факт book("Лермонтов", "Бородино", 70996) вместо прежнего можно с помощью предиката:

```
setNth(I, L, book("Лермонтов", "Бородино", 70996), L1),
```

где L1 — выходной список.

Далее приведена программа, заменяющая в базе фактов одно произведение Лермонтова другим, с другим числом продаж:

```
implement main
    open core, console, list
class facts - bestseller
    book: (string Автор, string Название, unsigned Число_продаж).
class predicates
    tolist: () -> bestseller*.
    toDB: (bestseller*).
clauses
    book("Достоевский", "Бесы", 19775).
    book("Лермонтов", "Демон", 22613).
    book("Пушкин", "Руслан и Людмила", 38104).
    book("Толстой", "Война и мир", 74398).

    tolist() = [book(A,B,C) || book(A,B,C)]. % собираем факты в список
    toDB(L):- retractFactDb(bestseller), % удаляем все факты базы
            forAll(L, { (X):- assert(X)}). % воссоздаем базу из списка
run():-
    L = tolist(), % получить список фактов
    write(L), nl,
    I = tryGetIndexEq({(A,book(A, _, _))}, "Лермонтов", L),
    setNth(I, L, book("Лермонтов", "Бородино", 70996), L1),
    toDB(L1), % обновить базу фактов
    write(tolist()), nl, !,
    _ = readchar(); % просмотреть базу фактов
    _ = readchar().
end implement main
goal
    console::run(main::run).
```

Задание 36.11. Разработать программу, реализующую механизм поддержки когерентности базы фактов и списка фактов согласно протоколу неотложенной записи.

Задание 36.12. Разработать программу, реализующую механизм поддержки когерентности базы фактов и списка фактов согласно протоколу отложенной записи.

ГЛАВА 37



Задачи на графах

Теоретический материал этого практического занятия рассмотрен в главе 25.

Задание 37.1. Разработать программу поиска всех путей на неориентированном графе от одной вершины до другой. Вершины неориентированного графа должны быть заданы координатами x, y декартовой системы. Расстояние между вершинами вычисляется по известной формуле — через корень квадратный из суммы квадратов разностей координат. Подсказка: ребро между двумя вершинами с координатами x, y и x_1, y_1 можно описать так:

ребро (X, Y, X_1, Y_1)

или так:

ребро (вершина (X, Y), вершина (X_1, Y_1))

предварительно описав домен вершина (integer X , integer Y).

ПРИМЕР 37.1. Разработаем программу поиска минимального пути на графике, используя алгоритм поиска «сначала вглубь», встроенный в машину вывода Пролога.

Наилучшее текущее решение сохраняется в двух фактах-переменных базы данных `минПуть` и `минДлина` соответственно. При запуске программы эти факты не определены, поэтому они получают значения только при нахождении первого решения. До этого момента факты-переменные не могут участвовать в операциях сравнения. Предикат `isErroneous` дает возможность проверить, были ли уже инициализированы факты-переменные. Если инициализации не было, то этот предикат успешен.

С помощью отката в предикате `run` поочередно находятся остальные пути. Длина каждого сравнивается с длиной текущего минимального пути `минДлина`. Если она меньше, чем `минДлина`, то в фактах-переменных `минПуть` и `минДлина` запоминается новое решение. Сравнение с `минПуть` производится не только в конце построения очередного пути, но и в ходе его построения. Поэтому если текущая длина очередного пути превышает значение `минПуть`, то построение этого пути прекращается, и ищется другой путь на дереве решений. После проверки всех путей на графике факты `минПуть` и `минДлина` будут содержать минимальный путь и его длину соответственно.

`implement main`

`open core, console, list`

```

class facts - граф
дуга: (char Вершина1, char Вершина2, unsigned Длина) .
class facts
минПуть:char* := erroneous.
минДлина:unsigned := erroneous.
class predicates
ребро: (char,char,unsigned) nondeterm (i,i,o) (i,o,o).
путь: (char,char,char*,unsigned) nondeterm.

clauses
дуга('a','b',60).    дуга('a','c',15).
дуга('a','d',70).    дуга('b','c',50).

ребро (A,Б,Длина) :- дуга (A,Б,Длина); дуга (Б,A,Длина) .

путь (A,Б,Путь,Аккум) :-
    ребро (A,Б,Расстояние),           % конец пути найден
    (isErroneous (минДлина),          % это первое решение
     минДлина:=Аккум+Расстояние,
     минПуть:=reverse ([Б|Путь]);   % сохраняем длину первого пути
     минПуть:=reverse ([Б|Путь]);   % сохраняем первый путь
     % это не первое решение
     Аккум1 = Аккум+Расстояние,      % находим длину нового пути
     минДлина>Аккум1,               % новый путь короче
     минДлина:=Аккум1,               % сохраняем длину нового пути
     минПуть:=reverse ([Б|Путь])).  % сохраняем новый путь

путь (A,Б,Стек,Аккум) :-
    ребро (A,A1,Расстояние),         % находим промежуточное ребро
    not (A1 in Стек),                % вершина A1 встретилась впервые
    Аккум1=Аккум+Расстояние,        % находим длину текущего пути
    (isErroneous (минДлина);        % если первого решения еще нет
     минДлина>Аккум1),             % или мин. длина не превышена
    путь (A1,Б, [A1|Стек],Аккум1). % то двигаемся по графу дальше

run() :-
    write("Начальная вершина - "),
    Старт=readchar(),clearInput(),
    write("Конечная вершина - "),
    Стоп=readchar(),clearInput(),
    путь (Старт,Стоп,[Старт],0),
    fail;
    write(минПуть," ",минДлина),
    _ = readLine().

end implement main
goal
    console::run(main:::run).

```

Введя при запуске программы символы `d` и `c` в качестве начальной и конечной вершин, мы увидим минимальный путь и его длину:

Начальная вершина - `d`

Конечная вершина - `c`

`['d', 'a', 'c'] 85`

Задание 37.2. Написать программу для поиска минимального пути на неориентированном графе, координаты вершин которого задаются в декартовой системе.

Задание 37.3. Написать программу для поиска максимального пути на неориентированном графе по стратегии «сначала вширь».

Задание 37.4. Написать программу для поиска всех цепей заданной длины N от заданной вершины на неориентированном графе. Цепь длины N — это путь на графе, содержащий N смежных вершин.

Задание 37.5. Написать программу для поиска минимального пути коммивояжера на полном графе. Путь коммивояжера — это путь, выходящий из стартовой вершины, проходящий по всем вершинам ровно один раз и возвращающийся в стартовую вершину. Полный граф — это граф, у которого любые две вершины являются смежными, т. е. между ними есть ребро.

Задание 37.6. Найти минимальный путь из точки с координатами x,y в точку с координатами x_1,y_1 на поле, заданном в декартовой системе координат своим левым верхним углом с координатами A,B и правым нижним углом с координатами C,D . Предусмотреть на поле препятствия в точках со случайными координатами. Эти препятствия описать фактами вида:

`domains`

`препятствие: (integer X, integer Y).`

ПРИМЕР 37.2. Найти такое минимальное множество вершин неориентированного графа, для которых все остальные вершины являются смежными (соседними).

Воспользуемся методом «образовать и проверить». Его идея заключается в том, что мы будем строить подмножества вершин графа и проверять, являются ли остальные вершины графа смежными к вершинам нашего подмножества. Если являются, то подмножество вершин и есть искомое решение. Для того чтобы первое найденное подмножество было минимальным, мы будем строить подмножества, начиная с тех, которые содержат одну вершину, потом две вершины и т. д.

Для ускорения вычислений представим граф фактами `tuple(unsigned,unsigned*)`, каждый из которых выражает куст — т. е. все множество ребер `unsigned*`, инцидентных вершине, указанной первым аргументом. Например, факт `tuple(3,[1,12,7])` выражает соединение вершины 3 с вершинами 1, 12 и 7.

Алгоритм решения нашей задачи включает построение списка вершин графа и сохранение для каждой вершины `N` списка смежных вершин `L` в виде факта `tuple(N,L)`. На этом заканчивается этап подготовки данных. Далее, с помощью недетерминированной функции `SL = subList(N,LN)`, мы будем строить подсписки `SL` списка вершин `LN`, длина которых `N` будет монотонно возрастать, начиная с единицы.

ници. Монотонное увеличение параметра `N` осуществляется функцией `N=std::countFrom(1)`.

Для проверки каждого построенного подсписка служит предикат `test(SL, LN)`. Этот предикат удаляет из исходного списка вершин графа `LN` те вершины, которые являются смежными к вершинам генерированного подсписка `SL`. Сами смежные вершины находятся в базе фактов `tuple`. Если после удаления всех смежных вершин список `LN` опустеет, то считается, что проверка завершилась успешно, и подсписок `SL` является искомым минимальным множеством вершин, для которых все остальные вершины графа являются смежными.

```

implement main
    open core, console, list
class facts
    дуга:(unsigned,unsigned).      % граф
    tuple:(unsigned,unsigned*).   % (вершина, соседи)
class predicates
    ребро:(unsigned,unsigned) nondeterm (o,o) (i,o).
    test:(unsigned*,unsigned*) nondeterm.
    subList:(integer, unsigned*) -> unsigned* nondeterm.
clauses
    дуга(1,5). дуга(1,2). дуга(1,3). дуга(1,4). дуга(4,6). дуга(6,7).
    дуга(6,8). дуга(7,9). дуга(2,3). дуга(3,4). дуга(8,5).

    ребро(X,Y):-дуга(X,Y);дуга(Y,X).

    test([],[]):-!.
    test([A|SL],LN):-tuple(A,AL),LN1=difference(LN,[A|AL]),
                    test(SL,LN1).

    subList(1,[E|_]) = [E].
    subList(N,[_|List]) = subList(N,List).
    subList(N,[Elem|List]) = [Elem|subList(N-1,List)].

run():-
    LN=list::removeDuplicates([N| ребро(N,_)]),           % список вершин
    list::forAll(LN,{(N):-
        L=[M||ребро(N,M)],assert(tuple(N,L))}),       % служебная БД
    N=std::countFrom(1),                                % длина подсписка
    SL = subList(N,LN),                                 % получение подсписка
    test(SL,LN),                                       % проверка подсписка
    write(SL),                                         % вывод подсписка
    _ = readLine(),!
    write("end"),
    _ = readLine().
end implement main
goal
    console::run(main::run).

```

Запуск программы показывает, что искомое множество содержит три элемента: 3, 8 и 9.

Задание 37.7. Найти такое минимальное множество вершин неориентированного графа, для которых все остальные вершины находятся на расстоянии N . Длину ребер положить равной единице.

Задание 37.8. Определить, является ли граф планарным. Подсказка: планарный граф — это граф, не содержащий подграф K_5 или подграф $K_{3,3}$. Граф K_5 — это полный граф, содержащий пять вершин (рис. 37.1). Граф $K_{3,3}$ — это граф, в котором каждая вершина из одной тройки вершин соединена с каждой вершиной из другой тройки вершин (рис. 37.2).

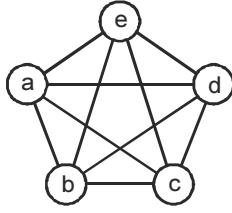


Рис. 37.1. Граф K_5

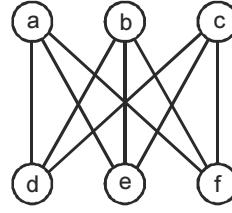


Рис. 37.2. Граф $K_{3,3}$

Задание 37.9. Данна карта, содержащая N городов, связанных дорогами каждый с каждым. Нужно соединить все города телефонной связью так, чтобы общая длина телефонных линий была минимальной. Другими словами: найти оствовное дерево минимальной длины на неориентированном полном графе, содержащем N вершин.



ГЛАВА 38

Задачи на деревьях

Теоретический материал этого практического занятия рассмотрен в главе 26.

ПРИМЕР 38.1. Найти самую длинную ветку в бинарном дереве.

Рассмотрим простой алгоритм, в котором определяется самая длинная ветка в левом поддереве, а потом самая длинная ветка в правом поддереве. После этого рекурсия углубляется в то поддерево, которое имеет самую длинную ветку.

```
implement main
    open core, console
domains
    tree = t(integer,tree,tree);n.           % бинарное дерево
    branch = integer*.                      % ветвь дерева как список вершин
class predicates
    depth:(tree,branch,integer) procedure (i,o,o).
clauses
    depth(t(Верш,Лев,Прав), [Верш|Ветка], Г):-!, 
        depth(Лев, Ветка1, Г1),             % глубина левого дерева
        depth(Прав, Ветка2, Г2),             % глубина правого дерева
        (Г1>=Г2, !, Ветка=Ветка1, Г=Г1+1;   % углубляемся влево
        Ветка=Ветка2, Г=Г2+1).              % углубляемся вправо
    depth(n, [], 0).                      % лист дерева имеет глубину 0
run():-
    Дерево = t(20,t(12,t(10,n,n),t(17,n,n)),t(25,t(23,t(22,n,n),n),n)),
    depth(Дерево, Ветка, Глубина),
    write(Ветка, Глубина),
    _ = readLine().
end implement main
goal
    console::run(main::run).
```

Недостаток этого алгоритма состоит в многократном прохождении ветвей дерева. Самая длинная ветка будет пройдена столько раз, какова ее длина.

Другой алгоритм основан на поиске «сначала вглубь» и поэтому проходит каждую ветвь один раз. В конце каждой ветви он сравнивает ее длину с фактом-переменной `d`, в которой хранится максимальная на данный момент длина. Если текущая ветвь длиннее, то факты-переменные `d` и `b` обновляются:

```

implement main
    open core, console, list
domains
    tree = t(integer,tree,tree);n.
    branch = integer*.
class facts
    d:unsigned := 0.          % длина ветви
    b:branch := [].           % ветвь
class predicates
    depth:(tree,branch,unsigned) nondet.
clauses
    depth(t(Верш,Лев,_),Ветка,Г):-depth(Лев,[Верш|Ветка],Г+1).
    depth(t(Верш,_,Прав),Ветка,Г):- depth(Прав,[Верш|Ветка],Г+1).
    depth(n,Ветка,Г):-Г>d,d:=Г,b:=Ветка.          % обновление фактов

run() :-
    Дерево = t(20,t(12,t(10,n,n),t(17,n,n)),t(25,t(23,t(22,n,n),n),n)),
    depth(Дерево,[],0),fail;
    writeln("Ветка=~{, Глубина=~{", reverse(b),d),
    _ = readLine().
end implement main
goal
    console:::run(main:::run).

```

Задание 38.1. Найти самую длинную ветку в бинарном дереве с использованием поиска «сначала вширь».

Задание 38.2. Найти разность между самой длинной и самой короткой ветвями бинарного дерева.

Задание 38.3. Определить, является ли неориентированный граф, заданный базой фактов ребро(вершина,вершина), деревом. Найти такой корень дерева, при котором дерево имеет наименьшую разность между самой длинной и самой короткой ветвями.

Задание 38.4. Определить, является ли бинарное дерево строгим бинарным деревом, т. е. таким, в котором узел, не являющийся листом, имеет не пустые правые и левые поддеревья.

ПРИМЕР 38.2. Дан список элементов произвольной природы. Список не упорядочен и содержит дубликаты элементов. Необходимо определить количество вхождений каждого элемента в список.

Для решения этой задачи используем красно-черное дерево, работающее с уникальными элементами. Ключом для доступа будет являться элемент списка, а хра-

нимым данным — счетчик числа вхождений элемента в список. Предикат `toTree` размещает в дереве все элементы списка. При размещении нового элемента его счетчик устанавливается в единицу. Когда размещаемый элемент присутствует в дереве, то его счетчик инкрементируется.

Данные из дерева собираются с помощью недетерминированной функции `getAll_nd` в список кортежей, содержащих элемент и значение его счетчика `tuple(Элемент, Счетчик)`.

```

implement main
    open core, console, redBlackTree
class predicates
    toTree:(E*,tree{E, unsigned},tree{E, unsigned} [out]) .
clauses
    toTree([A|L],T,T0) :-
        (I = tryLookUp(T,A),           % элемент A есть в дереве
         T1 = insert(T,A,I+1),        % инкремент счетчика элемента
         T1 = insert(T,A,1) ),!,     % элемент A отсутствует, счетчик = 1
        toTree(L,T1,T0).
    toTree([],T,T).

run() :-
    L = [7,8,10,8,7,8,10,8],
    T = emptyUnique(),
    toTree(L,T,T1),
    write([Tuple||Tuple = getAll_nd(T1)]),nl,
    L1 = ["wow","eh","h","eh","wow"],
    T2 = emptyUnique(),
    toTree(L1,T2,T3),
    write([Tuple||Tuple = getAll_nd(T3)]),nl,
    _ = readLine().
end implement main
goal
    console::run(main::run).
```

Программа работает с полиморфными списками. Вначале обрабатывается список целых чисел `[7,8,10,8,7,8,10,8]`, затем список строк `["wow", "eh", "h", "eh", "wow"]`. Результат:

```
[tuple(7,2),tuple(8,4),tuple(10,2)]
[tuple("wow",2),tuple("eh",2),tuple("h",1)]
```

Задание 38.5. Дан список элементов произвольной природы. Список не упорядочен и содержит дубликаты элементов. Найти элемент, имеющий наибольшее число вхождений в список. Если их несколько, то найти их все.

ПРИМЕР 38.3. Даны два списка элементов произвольной природы. Длина списков одинакова. Списки не упорядочены и содержат дубликаты элементов. Необходимо подтвердить или опровергнуть утверждение о том, что число вхождений каждого элемента в первом списке такое же, как и во втором списке.

Для решения этой задачи воспользуемся красно-черным деревом, работающим с уникальными элементами. Ключом для доступа будет являться элемент списка, а хранимым данным — счетчик числа вхождений элемента в список. Предикат `toTree` размещает в дереве все элементы первого списка. При размещении нового элемента его счетчик устанавливается в единицу. Когда размещаемый элемент существует в дереве, то его счетчик инкрементируется.

Предикат `remove` удаляет из дерева элементы, входящие во второй список. Когда счетчик удаляемого элемента больше 1, то он декрементируется, когда счетчик удаляемого элемента равен 1, то элемент полностью удаляется из дерева. Если удаляемый элемент отсутствует в дереве, то предикат `remove` будет неуспешен. Когда второй список опустеет, то производится проверка того, что дерево тоже пусто. Если проверка неуспешна, то предикат `remove` также будет неуспешен.

```

implement main
    open core, console, redBlackTree
class predicates
    toTree: (E*, tree{E, unsigned}, tree{E, unsigned} [out]) .
    remove: (E*, tree{E, unsigned}) determ.
clauses
    toTree([A|L], T, T0) :-
        (I = tryLookUp(T, A),           % элемент A есть в дереве
         T1 = insert(T, A, I+1),       % инкремент счетчика элемента
         T1 = insert(T, A, 1) ), !,   % элемент A отсутствует, счетчик = 1
        toTree(L, T1, T0).
    toTree([], T, T).

    remove([A|L], T) :-
        I = tryLookUp(T, A),           % элемент A есть в дереве
        (I>1,
         T1 = insert(T, A, I-1), !; % декремент счетчика элемента
         T1 = delete(T, A, I)),     % удаление последнего элемента
        remove(L, T1).
    remove([], T) :- isEmpty(T).    % проверка пустоты дерева

run() :-
    L = [7,8,10,8,7,8,10,8],          % первый список
    T = emptyUnique(),               %
    toTree(L, T, T1),                % элементы первого списка помещаем в дерево
    L1 = [10,7,8,7,10,8,8,8],          % второй список
    (remove(L1, T1), write("да"));    % удаляем элементы из дерева
    write("нет")), !
    _ = readLine().
end implement main
goal
    console::run(main:::run).

```

В результате программа ответит "да". Если бы элементы списков были только числами или символами, то вместо дерева можно было бы использовать одномерный

массив. Тогда программа работала бы быстрее из-за константного времени доступа к элементам. Но так как элементы списков могут быть и строками, объектами, деревьями и т. п., то мы можем использовать только деревья. Поэтому программа будет работать несколько медленнее, поскольку время доступа ко всем элементам дерева растет не быстрее, чем функция $n \cdot \log_2(n)$, где n — размер списка.

Предикаты программы полиморфны, поэтому она работает, например, и со строками. Для проверки этого достаточно использовать следующий предикат `run`:

```
run() :-  
    L = ["7", "8", "10", "8", "7", "8", "10", "8"],  
    T = emptyUnique(),  
    toTree(L, T, T1),  
    L1 = ["10", "7", "8", "7", "10", "8", "8", "8"],  
    (remove(L1, T1), write("да"));  
    write("нет")), !,  
    _ = readLine().
```

В результате программа ответит "да".

Задание 38.6. Дан список элементов произвольной природы. Разделить его, если это возможно, на два таких подсписка у которых число вхождений каждого элемента в первом подсписке такое же, как и во втором подсписке.

Задание 38.7. Дан произвольный текст. Получить частотный словарь этого текста.

ГЛАВА 39



Задачи на массивах

Теоретический материал этого практического занятия рассмотрен в главе 27.

39.1. Решето Эратосфена

ПРИМЕР 39.1. Решето Эратосфена используют для поиска простых чисел. Рассмотрим заполнение решета Эратосфена в заданном диапазоне чисел тремя способами. Первый способ организован предикатом `эраторсфен` и основан на использовании массива `binary`. Второй способ организован предикатом `эраторсфен1` и использует битовый массив `arrayM_boolean`. Третий способ организован предикатом `эраторсфен2` и основан на одномерном массиве `arrayM`. Сравним эти способы по времени выполнения.

При создании массива `binary` он заполняется нулевыми байтами. Индекс байта является значением числа. Места расположения составных чисел будем отмечать единицами. После заполнения массива некоторые байты останутся нулевыми, что будет свидетельствовать о том, что индексы этих байтов являются простыми числами.

С помощью функций `getIndexed_integer8` и `setIndexed_integer8` класса `binary` будем читать и писать значения элементов массива. Перебор чисел на всем заданном диапазоне организуем с помощью внешнего цикла с откатом, управляемого недетерминированной функцией `fromTo(2, Len)` класса `std`. Заполнять решето единицами с заданным шагом будем с помощью внутреннего цикла с откатом, управляемого недетерминированной функцией `fromToInStep` класса `std`.

Обработка массивов `arrayM_boolean` и `arrayM` выполняется аналогично. Способы решения задачи сравнимы по времени их работы. Измерение времени можно производить по-разному. В коммерческой версии `Visual Prolog` для этого лучше использовать класс `testsupport`, т. к. он наиболее точно измеряет малые промежутки времени:

```
T = testSupport::getProcessCurrentTime(),
<измеряемый процесс>
testSupport::getProcessIntervalTime(T,D,H,M,S),
writeln("%02u дней %02u часов %02u мин. %2.2f сек.", D,H,M,S),
```

Левая подстрока `%02u` означает, что вместо знака процента будет подставлено значение переменной `D`, т. е. количество дней. Дни станут отображаться двумя цифрами, при этом левая цифра будет нулем, если количество дней выражается одной цифрой. Буква `u` показывает, что количество дней надо понимать как целое число без знака. Так как переменная `D` у нас является целым числом, то эту букву можно опустить. Правая подстрока `%02.2f` означает, что целую часть секунд следует выводить двумя цифрами, а дробную часть надо округлить, оставив два знака после запятой. Буква `f` показывает, что секунды являются вещественным числом.

В «персональной» версии Visual Prolog для измерения времени можно использовать пакет `time`:

```
T=time::new(),
<измеряемый процесс>
T1=time::new(),
T : getInterval(T1) : getIntervalDetailed(D,H,M,S),
writeln("%02u дней %02u часов %02u мин. %2.2f сек.",D,H,M,S),
```

Далее представлена программа получения решета Эратосфена размером 5 млн чисел тремя способами для персональной версии Visual Prolog.

```
implement main
    open core, console, math, std, binary
domains
    b = [0..1].           % значения для массива arrayM
class facts
    i:unsigned := 0.      % счетчик простых чисел для binary
    j:unsigned := 0.      % счетчик простых чисел для arrayM_boolean
    k:unsigned := 0.      % счетчик простых чисел для arrayM{b}
class predicates
    эратосфен: (binary,byteCount).
    эратосфен1: (arrayM_boolean,byteCount).
    эратосфен2: (arrayM{b},byteCount).
clauses
    эратосфен(A,Len) :-
        Q=round(sqrt(convert(ureal,Len))),          % вложенный цикл от 2 до Len
        B=fromTo(2,Len),                            % если В-простое число
        0=getIndexed_integer8(A,B-1),                % то инкремент счетчика
        i:=i+1,                                     % внутр. цикл от 2 до корня из Len
        B<=Q,                                       % внутренний цикл с шагом В
        I=fromToInStep(2*B,Len,B),                  % заполнение решета единицами
        setIndexed_integer8(A,I-1,1),
        fail;                                         % оба цикла работают на откате от одного fail
        succeed.

    эратосфен1(A,Len) :-
        Q=round(sqrt(convert(ureal,Len))),          % вложенный цикл от 2 до Len
        B=fromTo(2,Len),                            % если В-простое число
        false=A:get(B-1),
```

```

j:=j+1,                                % то инкремент счетчика
B<=Q,                                    % внутр. цикл от 2 до корня из Len
I=fromToInStep(2*B,Len,B),               % внутр. цикл с шагом B
A:set(I-1),                             % заполнение решета
fail;                                     % оба цикла работают на откате от одного fail
succeed.

```

`эрратосфен2(A,Len) :-`

```

Q=round(sqrt(convert(ureal,Len))),          %
B=fromTo(2,Len),                           % внешний цикл от 2 до Len
0=A:get(B-1),                            % если B-простое число
k:=k+1,                                   % то инкремент счетчика
B<=Q,                                    % внутр. цикл от 2 до корня из Len
I=fromToInStep(2*B,Len,B),               % внутр. цикл с шагом B
A:set(I-1,1),                            % заполнение решета единицами
fail;                                     % оба цикла работают на откате от одного fail
succeed.

```

`run() :-`

`Size = 5000000,` % размер решета Эратосфена

`T=time::new(),` % создаем объект времени
 `A=binary::createAtomic(Size),` % создаем массив binary
 `эратосфен(A,Size),` % заполнение решета Эратосфена

`T1=time::new(),` % создаем объект времени
 `B=arrayM_boolean::new(Size),` % создаем массив arrayM_boolean
 `эратосфен1(B,Size),` % заполнение решета Эратосфена

`T2=time::new(),` % создаем объект времени
 `C=arrayM::newAtomic(Size),` % создаем массив arrayM
 `эратосфен2(C,Size),` % заполнение решета Эратосфена
 `T3=time::new(),`

```

T : getInterval(T1) : getIntervalDetailed(_,_,M,S),
writeln("Простых чисел : % \t%2u мин.\t%2.2f сек.",i,M,S),nl,
T1 : getInterval(T2) : getIntervalDetailed(_,_,M1,S1),
writeln("Простых чисел : % \t%2u мин.\t%2.2f сек.",j,M1,S1),nl,
T2 : getInterval(T3) : getIntervalDetailed(_,_,M2,S2),
writeln("Простых чисел : % \t%2u мин.\t%2.2f сек.",k,M2,S2),nl,
_ = readchar().

```

`end implement main`

`goal`

`console::run(main::run).`

Результаты работы программы показывают превосходство бинарных массивов над остальными:

Простых чисел : 348513 0 мин. 1.88 сек.

Простых чисел : 348513 0 мин. 3.32 сек.

Простых чисел : 348513 0 мин. 2.29 сек.

Это объясняется тем, что бинарные массивы имеют меньше накладных расходов при обращении к элементам памяти. Больше всего накладных расходов у массива arrayM_boolean, т. к. для доступа к отдельным битам приходится выполнять битовые операции над байтами.

Задание 39.1. Разработать программу для поиска первой тысячи простых чисел.

Задание 39.2. Разработать программу для поиска простых чисел в диапазоне от a до b . Величины a до b вводятся с клавиатуры.

Задание 39.3. Разработать программу для поиска простых чисел на основе решета Сундара.

Задание 39.4. Разработать программу для поиска простых чисел на основе решета Аткина.

39.2. Сравнение частотных буквей текстов

ПРИМЕР 39.2. Даны две строки одинаковой длины. Определить, равны или нет частотные буквари этих строк. Наше решение будет основано на использовании одномерного массива счетчиков символов. Каждый символ имеет свой уникальный Unicode, который будет являться индексом счетчика в нашем массиве. Отбросим первые 32 кода, т. к. многие из них не имеют своего символьного отображения на экране. Суть алгоритма заключается в инкременте счетчиков при посимвольном сканировании первой строки и последующего декремента счетчиков при посимвольном сканировании второй строки. Если после этих операций массив будет содержать все нули, то число вхождений каждого символа в этих строках одинаково. Для ускорения алгоритма перед каждой операцией декремента сделаем проверку того, что счетчик больше нуля.

Для оценки эффективности различных массивов решим эту задачу с использованием двух массивов: `binary` и `arrayM`. Каждый из массивов имеет размер 65 000, что практически полностью покрывает весь диапазон двухбайтовой кодировки символов Unicode.

Предикат `toBin` инкрементирует счетчики массива `binary`, отделяя по одному символу от первой строки. Предикат `fromBin` декрементирует счетчики массива `binary`, отделяя по одному символу от второй строки. Аналогичные операции совершают предикаты `toArray` и `fromArray` с массивом `arrayM`.

Проверку массива `binary` выполняет предикат `testBin`. Если массив `binary` содержит только нули, то предикат успешен. Аналогичную операцию выполняет предикат `testArr` с массивом `arrayM`.

Для проверки правильности работы алгоритма в предикате `run` предусмотрены исходные данные:

```
Str1 = "123qwe3",
Str2 = "3qwe123",
```

В программе они закомментированы, т. к. для оценки времени работы используются две строки длиной 3 миллиона символов каждая. Первая строка содержит случайные символы с кодами от 32 до 65 032. Вторая строка является реверсом первой.

Факт-переменная `flag` используется для возвращения результата работы алгоритма. Стока "да" означает идентичность частотных буквей, строка "Нет" говорит об их различии.

```
implement main
    open core, console, binary, math, string
class facts
    flag:string := erroneous.
class predicates
    toBin:(string,binary) determ.
    fromBin:(string,binary) determ.
    testBin:(positive,binary) determ.
    toArray:(string,arrayM{unsigned}) determ.
    fromArray:(string,arrayM{unsigned}) determ.
    testArr:(arrayM{unsigned}) determ.
clauses
    toBin(W,Bin):-frontchar(W,Char,L),
        Unicode = uncheckedConvert(unsigned16,Char),
        Index = tryConvert(positive,Unicode),
        Count = getIndexed_unsigned(Bin,Index-32),
        setIndexed_unsigned(Bin,Index-32,Count+1),!,
        toBin(L,Bin).
    toBin("",_).

    fromBin(W,Bin):-frontchar(W,Char,L),
        Unicode = uncheckedConvert(unsigned16,Char),
        Index = tryConvert(positive,Unicode),
        Count = getIndexed_unsigned(Bin,Index-32),Count>0,
        setIndexed_unsigned(Bin,Index-32,Count-1),!,
        fromBin(L,Bin).
    fromBin("",_).

    testBin(0,Bin):-!,0=getIndexed_unsigned(Bin,0).
    testBin(Index,Bin):-0=getIndexed_unsigned(Bin,Index),
        testBin(Index-1,Bin).

    toArray(W,Array):-frontchar(W,Char,L),
        Unicode = uncheckedConvert(unsigned16,Char),
        Index = tryConvert(positive,Unicode),
        Count = Array:get(Index-32),
        Array:set(Index-32,Count+1),!,toArray(L,Array).
    toArray("",_).
```

```

fromArray(W,Array) :- frontchar(W,Char,L),
    Unicode = uncheckedConvert(unsigned16,Char),
    Index = tryConvert(positive,Unicode),
    Count = Array:get(Index-32), Count>0,
    Array:set(Index-32,Count-1), !, fromArray(L,Array).
fromArray("",_).

testArr(Array):-0<>Array:getValue_nd(),!,fail; succeed.

run() :-
    N = 3000000,
    L1 = [getCharFromValue(tryConvert(unsigned16,
        random(65000)+32)) || _ = std::fromTo(1, N)],
    L2 = list::reverse(L1),
    Str1 = string::createFromCharList(L1),
    Str2 = string::createFromCharList(L2),
%   Str1 = "123qwe3",
%   Str2 = "3qwe123",
    Size = 65000,

    T=time::new(),
    Bin = binary::createAtomic(Size,4),
    ( toBin(Str1,Bin), fromBin(Str2,Bin), testBin(Size-33,Bin),
      flag:="Да",!; flag:="Нет" ),
    T1=time::new(),
    T : getInterval(T1) : getIntervalDetailed(_,_,M,S),
    writeln("% \t%2u мин.\t%2.2f сек.",flag,M,S),nl,

    T2=time::new(),
    Array = arrayM{unsigned}::newAtomic(Size),
    ( toArray(Str1,Array), fromArray(Str2,Array), testArr(Array),
      flag:="Да",!; flag:="Нет" ),
    T3=time::new(),
    T2 : getInterval(T3) : getIntervalDetailed(_,_,M1,S1),
    writeln("% \t%2u мин.\t%2.2f сек.",flag,M1,S1),nl,

    _ = readchar().
end implement main
goal
    console::run(main:::run).

```

Программа возвращает результат:

Да	0 мин. 1.17 сек.
Да	0 мин. 1.52 сек.

который говорит о том, что массив `binary` является наиболее эффективным, ибо не использует никаких промежуточных преобразований.

Описанный подход к решению является эффективным только потому, что символы строки имеют свой уникальный номер — Unicode. Если бы, например, вместо строки символов был дан список строк или вещественных чисел, то такой подход был бы затруднителен, т. к. пришлось бы искать функцию для отображения строк или вещественных чисел в уникальный целочисленный номер. Поэтому для элементов, не имеющих своего «естественного» уникального номера, подобные задачи решаются с использованием дерева вместо массива. Если такое дерево сбалансировано, то время доступа к листу дерева пропорционально логарифму количества листьев, что всегда дольше времени доступа к элементам массива.

Задание 39.5. Дано множество строк одинаковой длины. Найти подмножество строк с одинаковым частотным буквавром. Причем это подмножество должно содержать максимальное количество строк.

Задание 39.6. Решить обратную задачу. Дан частотный букварь текста. Построить текст из слов априори заданного набора.

Задание 39.7. На основе больших русских текстов найти частотный букварь русского языка и определить первые десять наиболее часто используемых букв. Букварь должен содержать только 33 русские буквы и не должен включать символ пробела, знаки пунктуации, цифры и т. п.

приложения



- Приложение 1.** Описание свойств и предикатов класса *console*
- Приложение 2.** Описание предикатов класса *file*
- Приложение 3.** Описание конструкторов класса *inputStream_file*
- Приложение 4.** Описание конструкторов класса *outputStream_file*
- Приложение 5.** Пакет *stream*
- Приложение 6.** Меню Visual Prolog
- Приложение 7.** Быстрые клавиши меню Visual Prolog
- Приложение 8.** Панель инструментов Visual Prolog
- Приложение 9.** Описание предикатов класса *std*
- Приложение 10.** Описание класса *string*
- Приложение 11.** Описание электронного архива, сопровождающего книгу

ПРИЛОЖЕНИЕ 1

Описание свойств и предикатов класса *console*

Свойства

1. mode — устанавливает консольные потоки в режим stream::mode.
2. codePage — устанавливает консольные потоки в кодовую страницу codePage.

Предикаты

1. clearInput — очистка буфера ввода (буфера клавиатуры).
2. clearOutput — очистка всего окна консольного приложения.
3. close — отсоединение программы от ее консоли.
4. InputStream = getConsoleInputStream — возвращает входной поток InputStream, связанный с активной консолью.
5. OutputStream = getConsoleOutputStream — возвращает выходной поток OutputStream, связанный с активной консолью.
6. Title = getConsoleTitle() — возвращает имя окна консоли.
7. Width = getConsoleWidth — возвращает ширину окна как максимальное число символов в строке.
8. ErrorStream = getErrorStream() — возвращает поток ошибок.
9. EventList = getEventQueue() — возвращает без ожидания список событий в порядке их поступления. Возможные события:

```
event =
key(
    unsigned     Клавиша нажата,
    unsigned     Число повторов,
    unsigned     Виртуальный код клавиши,
    unsigned     Виртуальный скан-код,
    char        Символ Unicode,
    unsigned     Состояние клавиатуры);
```

```

mouse(
    unsigned Координата X,
    unsigned Координата Y,
    unsigned Нажатая кнопка,
    unsigned Состояние кнопки управления,
    unsigned Флаги событий);

size(
    unsigned Координата X,
    unsigned Координата Y);

menu;
focus.

```

10. `Pos = getLocation()` — возвращает позицию курсора (столбец, строка), т. е. позицию следующего печатаемого символа. Позиция — это структура `console_native::coord(X, Y)`.

Пример: `console_native::coord(X, Y) = getLocation();`

Окно консольного Windows-приложения работает в текстовом режиме и имеет 25 строк, нумеруемых от 0 до 24, и 80 столбцов, нумеруемых от 0 до 79. Позиция левого верхнего угла экрана имеет координаты `coord(0, 0)`.

11. `Attribute = getTextAttribute()` — чтение атрибута (код цвета фона и код цвета символа) в текущей позиции курсора. Таблицу кодов цвета символа и фона см. в описании предиката `setTextAttribute/1` далее.
12. `init()` — инициализация консольных потоков в режиме `stream::unicode`. Потоки регистрируются в классе `stdio`.
13. `init(Mode)` — инициализация консольных потоков в режиме `Mode`. Возможные значения режима `Mode`:

```

stream::unicode;
stream::ansi(codePage);
stream::binary.

```

В качестве кодовой страницы `codePage` для русского текста можно указывать 866.

14. `init8()` — инициализация консольных потоков в кодовой странице ANSI с номером 3. Потоки регистрируются в классе `stdio`.
15. `init8(CodePage)` — инициализация консольных потоков в кодовой странице ANSI с номером `CodePage`. В качестве номера кодовой страницы `CodePage` для русского текста можно указывать 866.
16. `initUtf8()` — инициализация консольных потоков в режиме `utf8`. Константа `utf8` определяет кодовую страницу 65001.
17. `isModeEcho()` — истинен, если есть эхо-вывод в окно.
18. `isModeLine()` — истинен, если есть режим ввода строки только по клавише `<Enter>`.

19. `isModeProcessed()` — истинен, если есть режим обработки символов (`backspace`, `tab`, `bell`, `carriage return` и `linefeed`) без помещения их в буфер.
20. `isModeWrap()` — истинен, если есть режим переноса строки.
21. `nl()` — перевод курсора в начало следующей строки.
22. `Term = read()` — чтение терма. Домен терма определяется компилятором автоматически из контекста программы. Если такой контекст отсутствует, домен можно определить с помощью предиката `hasDomain(Домен, Терм)`.
23. `Char = readChar()` — чтение символа.
24. `EventList = readEvents()` — чтение списка событий в режиме ожидания. Как правило, возвращается список, содержащий одно событие. Список событий такой же, как и при использовании предиката `getEventQueue()`.
25. `String = readLine()` — чтение строки `String`.
26. `run(Runnable)` — вызов предиката `Runnable` в режиме `stream::unicode`. Предикат `Runnable` должен вызываться в секции `goal` или использоваться для создания новых потоков.
27. `run(Runnable, Mode)` — вызов предиката `Runnable` в режиме `Mode`:
`stream::unicode;`
`stream::ansi(codePage);`
`stream::binary.`
В качестве кодовой страницы `codePage` для русского текста можно указывать 866.
28. `run(Runnable, Mode, RunMode)` — вызов предиката `Runnable` в режиме `Mode`. Режим запуска потока `RunMode` может принимать значения:
`exe_api::ole;`
`exe_api::multithread;`
`exe_api::noCom.`
29. `run8(Runnable)` — вызов предиката `Runnable` в режиме ANSI используя кодовую страницу 3.
30. `run8(Runnable, CodePage)` — вызов предиката `Runnable` с использованием кодовой страницы `CodePage`. Для обработки русских текстов можно указывать 866.
31. `runUtf8(Runnable)` — вызов предиката `Runnable` с использованием кодовой страницы `utf8`. Константа `utf8` имеет значение 65001.
32. `setConsoleTitle("Имя окна")` — установка имени окна.
33. `setLocation(console_native::coord(X, Y))` — установка положения курсора для вывода следующего символа.
34. `setModeEcho(true/false)` — установка/запрет режима эхо-вывода.
35. `setModeLine(true/false)` — установка/запрет режима ввода строки только по клавише `<Enter>`.

36. `setModeProcessed(true/false)` — установка/запрет режима обработки символов.
37. `setModeWrap(true/false)` — установка/запрет режима переноса строки.
38. `setTextAttribute(Attribute)` — установка атрибута (цвета фона и символа). Для получения атрибута надо сложить код цвета символа и код цвета фона (табл. П1.1).

Таблица П1.1. Коды цвета символа и фона

Код цвета символа	Цвет символа	Код цвета фона	Цвет фона
0	Черный	0	Черный
1	Синий	16	Синий
2	Зеленый	32	Зеленый
3	Бирюзовый	48	Бирюзовый
4	Красный	64	Красный
5	Сиреневый	80	Сиреневый
6	Коричневый	96	Коричневый
7	Белый	112	Белый
8	Серый	128	Серый
9	Ярко-Синий	144	Ярко-Синий
10	Ярко-Зеленый	160	Ярко-Зеленый
11	Ярко-Бирюзовый	176	Ярко-Бирюзовый
12	Ярко-Красный	192	Ярко-Красный
13	Ярко-Сиреневый	208	Ярко-Сиреневый
14	Желтый	224	Желтый
15	Ярко-Белый	240	Ярко-Белый

39. `setTextAttribute(Color, Intencity)` — установка цвета символа и его интенсивности.
40. `write(...)` — вывод произвольного количества аргументов.
41. `writef(FormatString, ...)` — форматируемый вывод (см. главу 12).

ПРИЛОЖЕНИЕ 2

Описание предикатов класса *file*

1. `appendString(Filename, String,IsUnicodeFile)` — добавление в конец файла строки `String`. Нулевой символ конца строки в файл не пишется. Если файл по указанному пути отсутствует, он будет создан. Параметр `IsUnicodeFile` указывает кодировку строки при записи в файл. Для Unicode параметр `IsUnicodeFile = true()`, для ANSI параметр `IsUnicodeFile = false()`.
2. `consult(FileName, имя_бд)` — добавление фактов именованного раздела БД из текстового файла.
3. `consult(FileName, имя_бд,IsUnicodeFile [out])` — добавление фактов именованного раздела БД из текстового файла. Возвращает кодировку содержимого файла.
4. `copy(FileName, NewFileName)` — копирование файла. Путь в имени может быть полным или относительным. Если файл `NewFileName` уже существует, то запись будет поверх него.
5. `copy(FileName, NewFileName, Overwrite)` — копирование файла. Путь в имени может быть полным или относительным. Если файл `NewFileName` уже существует, то параметр `Overwrite` указывает что делать:

```
Overwrite =
    overwrite();           % Запись поверх
    skip();                % Не копировать
    append();              % Добавить в конец файла
    errorIfExist().        % Завершить по ошибке
```

6. `UniqueName = createUniqueName(DirectoryPath, Prefix)` — создание файла с уникальным именем `UniqueName`. Имя образуется конкатенацией пути `DirectoryPath`, префикса `Prefix`, уникальной шестнадцатеричной строки и строки для расширения `".tmp"`. Префикс образует начальную часть имени файла. Предикат использует первые три символа префикса. Такие уникальные имена обычно используются для временных файлов.
7. `delete(FileName)` — удаление файла.
8. `existExactFile(FileName)` — предикат успешен, если файл существует, иначе — неуспешен. Имя файла не может содержать знаки подстановки: `?` и `*`.

9. `existFile(FileName)` — предикат успешен, если файл существует, иначе — неуспешен. Имя файла может содержать знаки подстановки: ? и *.
10. `getFileProperties/6` — получение свойств файла. Шесть параметров предиката имеют следующее назначение:
 - `FileName` — имя файла;
 - Атрибуты `[out]` — атрибуты файла (см. далее);
 - Размер `[out]` — размер файла;
 - Создание `[out]` — дата создания файла;
 - Последнее_чтение `[out]` — дата последнего чтения файла;
 - Последнее_изменение `[out]` — дата последнего изменения файла.

Атрибуты файла представляются списком свойств:

```
Атрибуты =
  readOnly();
  system();
  hidden();
  archive();
  compressed();
  encrypted();
  temporary();
  deleteOnClose();
  offline() .
```

Выходные параметры `Создание`, `Последнее_чтение` и `Последнее_изменение` имеют числовое представление. Для преобразования их в дату и время, т. е. в значения года, месяца, числа, часа, минут и секунд, следует воспользоваться предикатами класса `time`. Например, для получения даты и времени создания файла можно воспользоваться следующим фрагментом кода:

```
getFileProperties(FileName, _, _, Создание, _, _),
Q = time::newFromGMT(Создание),
Q:getDateAndTime(Год, Мес, День, Час, Мин, Сек),
writef("%.%.% %:%:%", Год, Мес, День, Час, Мин, Сек),
```

11. `PhysicalName = getPhysicalName(FileName)` — получение имени файла в таких же регистрах символов имени, в каких они записаны на диске.
12. `isUnicode(FileName)` — предикат успешен, если содержимое файла в кодировке Unicode.
13. `move(FileName, Destination)` — перемещение и/или переименование файла. Если на диске существует файл, куда перемещается исходный файл, то запись будет происходить поверх.
14. `move(FileName, Destination, Overwrite)` — перемещение и/или переименование файла. Если на диске существует файл, куда перемещается исходный файл, то параметр `Overwrite` указывает что делать.

15. `moveToRecycleBin(File Name)` — перемещение файла в корзину.
16. `Binary = readBinary(File Name)` — чтение содержимого двоичного файла.
17. `Binary = readBinary(File Name, Offset, Size)` — чтение фрагмента двоичного файла размером `Size`, начиная с позиции `Offset`.
18. `Text = readString(File Name)` — чтение содержимого текстового файла целиком. Файл должен содержать текст Unicode.
19. `Text = readString(File Name, IsUnicode [out])` — чтение содержимого текстового файла целиком. Файл может содержать текст ANSI или Unicode. Текст ANSI при чтении будет преобразован в Unicode на основании текущей кодовой страницы.
20. `Text = readStringFromHandle(File Handle, IsUnicode [out])` — чтение текста из `File Handle`.
21. `reconsult(File Name, имя_бд)` — обновление именованного раздела БД. Удаляет факты из раздела `имя_бд` и загружает факты из файла.
22. `reconsult(File Name, имя_бд,IsUnicodeFile [out])` — обновление именованного раздела БД. Удаляет факты из раздела `имя_бд` и загружает факты из файла. Возвращает кодировку содержимого файла.
23. `save(File Name, имя_бд)` — сохранение на диске именованного раздела БД. В первой строке файла содержится ключевое слово `clauses`, после которого следуют факты, завершающиеся символом точки. В каждой строке — один факт.
24. `save(File Name, имя_бд,IsUnicodeFile [out])` — сохранение на диске именованного раздела БД. В первой строке файла содержится ключевое слово `clauses`, после которого следуют факты, завершающиеся символом точки. В каждой строке — один факт. Параметр `IsUnicodeFile` определяет кодировку фактов в файле.
25. `saveUtf8(File Name, имя_бд)` — сохранение на диске именованного раздела БД в кодировке `utf8`.
26. `setAttributes(Full Name, Attributes, IsTurnedOn)` — изменение атрибутов указанного файла. Атрибуты `Attributes` файла представляются списком свойств. Если параметр `IsTurnedOn` имеет значение `true()`, то устанавливаются те свойства, которые указаны в списке `Attributes`. Неуказанные в списке атрибуты игнорируются.
27. `setFileTime/4` — установка даты и времени создания, последнего доступа и последнего изменения файла. Четыре параметра предиката имеют следующее назначение:
 - `FileName` — имя файла;
 - `ДатаСоздания` — дата создания файла;
 - `ДатаПоследнегоДоступа` — дата последнего доступа к файлу;
 - `ДатаПоследнейЗаписи` — дата последней записи в файл.

28. `LastChange = tryGetLastChange(Filename)` — попытка получения даты последнего изменения указанного файла. Предикат завершится неудачей, если указанный файл не будет найден.
29. `FullName = trySearchFileInPath(Filename)` — попытка поиска указанного файла в текущей папке, системной папке или папке PATH. Предикат завершится неудачей, если указанный файл не будет найден.
30. `writeBinary(Filename, Binary)` — запись двоичных данных `Binary` в указанный файл. Если указанного файла нет, он будет создан. Если файл существует, запись будет производиться поверх него.
31. `writeBinary(Filename, Offset, Binary)` — запись двоичных данных `Binary` в указанный файл `Filename`, начиная с позиции `Offset`.
32. `writeString(Filename, Text)` — запись текста `Text` в файл `Filename`. Если указанного файла нет, он будет создан. Если файл существует, запись будет производиться поверх него. Символ нуля, завершающий текст, в файл не записывается.
33. `writeString(Filename, Text,IsUnicodeFile)` — запись текста `Text` в файл `Filename`. Если указанного файла нет, он будет создан. Если файл существует, запись будет производиться поверх него. Параметр `IsUnicodeFile` определяет кодировку содержимого файла ANSI или Unicode.
34. `writeString_CodePage(Filename, Text, CodePage)` — запись текста `Text` в файл `Filename` в указанной кодовой кодировке `CodePage`.
35. `writeStringUtf8(Filename, Text)` — запись текста `Text` в файл `Filename` в кодировке utf8. Если указанного файла нет, он будет создан. Если файл существует, запись будет производиться поверх него.
36. `writeStringUtf8_bom(Filename, Text)` — запись текста `Text` в файл `Filename` в кодировке utf8. В начало файла записывается маркер порядка байтов кодировки utf8. Если указанного файла нет, он будет создан. Если файл существует, запись будет производиться поверх него.

ПРИЛОЖЕНИЕ 3

Описание конструкторов класса *inputStream_file*

1. `openFile(File Name)` — открывает файл File Name для чтения и устанавливает указатель потока чтения в позицию 0 (начало файла).
2. `openFile(File Name, Mode)` — открывает файл File Name для чтения, устанавливает режим Mode и устанавливает указатель потока чтения в позицию 0 (начало файла). Параметр Mode может иметь следующие значения:

```
Mode = stream::unicode();           % по умолчанию
      stream::ansi(codePage);
      stream::binary().
```

Номера распространенных кодовых страниц CodePage:

<code>codePage = 1250</code>	% ANSI центрально-европейская
<code>codePage = 1251</code>	% ANSI кириллица
<code>codePage = 855</code>	% OEM кириллица традиционная
<code>codePage = 866</code>	% OEM русская

Режимы для чтения/записи двоичных данных `binary` и текста в кодировке `Unicode` и `ANSI` имеют некоторые различия. При чтении/записи двоичных данных не производится преобразование и они записываются или считываются в таком виде, в каком представлены в памяти или на носителе. При операциях с текстом данные преобразуются. Например, символ конца строки, представляемый в текстовом файле двумя байтами 10 и 13, конвертируется в '`\n`'.

3. `openFile(File Name, Mode, Access)` — открывает файл File Name для чтения, устанавливает указатель потока чтения в позицию 0 (начало файла), устанавливает параметр Mode и разрешенный доступ к файлу Access.

```
Access = fileSystem_api::permitRead();        % Только чтение
         fileSystem_api::permitReadWrite();    % Чтение и запись
         fileSystem_api::permitNone().       % Произвольный доступ
```

4. `openFile(File Name, Mode, Access, BufferSize)` — открывает файл File Name для чтения, устанавливает указатель потока чтения в позицию 0 (начало файла), устанавливает режим ввода данных Mode, а также разрешенный доступ к файлу

Access. Для чтения используется буфер. Размер буфера `BufferSize` должен быть четным и больше 2.

5. `openFile8(File Name)` — открывает файл `FileName` для чтения и устанавливает указатель потока чтения в позицию 0 (начало файла). Содержимое файла обрабатывается как текст ANSI.
6. `openFileUtf8(File Name)` — открывает `utf8` файл `FileName` для чтения и устанавливает указатель потока чтения в позицию 0 (начало файла). Содержимое файла обрабатывается как текст в кодировке `utf8`.

ПРИЛОЖЕНИЕ 4

Описание конструкторов класса *outputStream_file*

1. append(FileName) — открывает файл FileName для записи и устанавливает указатель потока вывода в конец файла.
2. append(FileName, Mode) — открывает файл FileName для записи и устанавливает указатель потока вывода в конец файла. Параметр Mode может иметь следующие значения:

```
Mode = stream::unicode();  
       stream::ansi(codePage);  
       stream::binary();
```

Номера распространенных кодовых страниц codePage:

```
codePage = 1250    % ANSI центрально-европейская  
codePage = 1251    % ANSI кириллица  
codePage = 855     % OEM кириллица традиционная  
codePage = 866     % OEM русская
```

Режимы для чтения/записи двоичных данных binary и текста unicode и ansi имеют некоторые различия. При чтении/записи двоичных данных не производится преобразование и они записываются или считываются в таком виде, в каком представлены в памяти или на носителе. При операциях с текстом данные преобразуются. Например, символ конца строки, представляемый в текстовом файле двумя байтами 10 и 13, конвертируется в '\n'.

3. append8(FileName) — открывает ANSI-файл FileName для записи и устанавливает указатель потока вывода в конец файла.
4. appendUtf8(FileName) — открывает UTF8-файл FileName для записи и устанавливает указатель потока вывода в конец файла.
5. create(FileName) — создает файл FileName для вывода текста в кодировке Unicode.
6. create(FileName, Mode) — создает файл FileName для вывода текста в кодировке Unicode с режимом Mode.
7. create(FileName, Mode, Access) — создает файл FileName для вывода текста в кодировке Unicode с режимом Mode и устанавливает доступ к файлу Access.

```

Access =
    fileSystem_api::permitRead();           % Только чтение
    fileSystem_api::permitWrite();          % Только запись
    fileSystem_api::permitReadWrite();      % Чтение и запись
    fileSystem_api::permitNone().          % Произвольный доступ

```

8. `create(FileName, Mode, Access, Attributes)` — создает файл `FileName` для вывода текста в кодировке Unicode с режимом `Mode` и устанавливает доступ к файлу `Access`. Если такой файл уже существует, то его содержимое будет потеряно. Атрибуты файла представляются списком свойств:

```

Attributes =
    readOnly();
    system();
    hidden();
    archive();
    compressed();
    encrypted();
    temporary();
    deleteOnClose();
    offline().

```

Список атрибутов может быть пустым.

9. `create8(FileName)` — создает файл `FileName` для вывода текста в кодировке ANSI.
10. `create8(FileName, Access)` — создает файл `FileName` для вывода текста в кодировке ANSI и устанавливает доступ к файлу `Access`.
11. `createUtf8(FileName)` — создает файл `FileName` для вывода текста в кодировке UTF8.
12. `createUtf8(FileName, Access)` — создает файл `FileName` для вывода текста в кодировке UTF8 и устанавливает доступ к файлу `Access`.
13. `createUtf8_bom(FileName)` — создает файл `FileName` для вывода текста в кодировке UTF8. В начало файла записывается маркер порядка байтов.
14. `createUtf8_bom(FileName, Access)` — создает файл `FileName` для вывода текста в кодировке UTF8 и устанавливает доступ к файлу `Access`. В начало файла записывается маркер порядка байтов.
15. `newHandle(Handle, Mode)` — создает новый поток из `Handle` с режимом `Mode`. По умолчанию `Mode` равно `unicode`.
16. `newHandleFile(FileHandle, Mode)` — создает новый файловый поток из `FileHandle` с режимом `Mode` и устанавливает указатель в начало файла. По умолчанию `Mode` равно `unicode`.
17. `openFile(FileName)` — открывает существующий файл `FileName` для вывода текста в кодировке Unicode и устанавливает указатель потока вывода в начало файла после маркера порядка байтов ВОМ, если таковой присутствует. Страницы процессы имеют доступ к этому файлу только по чтению.

18. `openFile(FileName, Mode)` — открывает существующий файл `FileName` для вывода текста в кодировке Unicode с режимом `Mode`. Устанавливает указатель потока вывода в начало файла.
19. `openFile(FileName, Mode, Access)` — открывает существующий файл `FileName` для вывода текста в кодировке Unicode с режимом `Mode`. Устанавливает указатель потока вывода в начало файла и доступ к файлу `Access`
20. `openFile8(FileName)` — открывает существующий ANSI-файл `FileName` для вывода и устанавливает указатель потока вывода в начало файла.
21. `openFileUtf8(FileName)` — открывает существующий UTF8-файл `FileName` для вывода и устанавливает указатель потока вывода в начало файла после маркера порядка байтов BOM, если таковой присутствует.

ПРИЛОЖЕНИЕ 5

Пакет *stream*

В это приложение включено описание предикатов трех классов пакета *stream*: *inputStream*, *outputStream* и *stream*.

Описание предикатов класса *inputStream*

1. `consult(Имя_бд)` — загружает факты именованного раздела `Имя_бд`.
2. `endOfStream()` — проверяет позицию указателя потока чтения. Предикат успешен, когда указатель находится в конце потока чтения, иначе — неуспешен.
3. `Term = read()` — читает терм из входного потока. При необходимости явное определение домена терма должно производиться предикатом `hasDomain/2`.
4. `Binary = readBytes(Size)` — читает двоичные данные длиной `Size` байтов из входного потока. Если во время чтения встречается конец потока, то чтение прекращается и возвращается такое количество байтов, какое реально прочитано.
5. `Char = readChar()` — читает символ `Char` из входного потока.
6. `Line = readLine()` — читает абзац `Line` из входного потока. Индикатором конца абзаца является символ перехода на новую строку '`\n`' или нулевой символ (для нуль-завершенных строк).
7. `Line = readLineCRLF()` — читает абзац `Line` из входного потока. Индикатором конца абзаца является символ перехода на новую строку '`\n`' или нулевой символ (для нуль-завершенных строк) или конец потока. Функция добавляет символ '`\n`' в конец абзаца `Line`, если этого символа там нет.
8. `String = readString(Size)` — читает строку длиной `Size` символов из входного потока. Если во время чтения встречается конец потока, то чтение прекращается и возвращается реально прочитанная строка.
9. `reconsult(Имя_бд)` — загружает факты именованного раздела `Имя_бд`, удаляя прежние факты этого раздела.

10. `repeatToEndOfStream()` — предикат успешен, пока поток ввода данных не пустой. Используется для организации цикла чтения, основанного на откате.
11. `ungetChar()` — возвращает последний прочитанный символ в поток ввода.

Описание предикатов класса *outputStream*

1. `flush()` — принудительный вывод содержимого буфера в выходной поток. Этот предикат полезен в двух случаях. Во-первых, когда выходной поток направлен в последовательный порт и возникает необходимость послать данные в порт до того, как буфер будет заполнен. Во-вторых, обеспечить получение другим процессом последних данных из потока.
2. `nl()` — выводит символы перехода на новую строку в выходной поток.
3. `save(Имя_бд)` — сохраняет факты именованного раздела `Имя_бд`.
4. `write(...)` — выводит значения указанных переменных в выходной поток.
5. `writeBytes(Pointer, Size)` — выводит `Size` байтов из оперативной памяти, начиная с адреса, на который указывает `Pointer`.
6. `writef(FormatString, ...)` — форматированный вывод (см. главу 12).
7. `writeQuoted(...)` — выводит значения указанных переменных в выходной поток. Если переменные содержат строки или символы, то значения этих переменных при выводе будут обрамляться в кавычки или апострофы.

Описание предикатов класса *stream*

1. `close()` — закрывает поток.
2. `Enabled = getCrLfConversion()` — возвращает значение, которое указывает на правила преобразования кодов перевода на новую строку при чтении/записи текста. Если `Enabled = true()`, то перед записью в поток осуществляется преобразование LF → CRLF, и после чтения из потока осуществляется преобразование CRLF → LF. Если `Enabled = false()`, то преобразование не осуществляется. Здесь:
 - CR (carriage return) — возврат каретки, обозначаемый кодом 13;
 - LF (line feed) — перевод строки, обозначаемый кодом 10.
3. `Mode = getMode()` — получение режима потока.

```
Mode = stream::unicode();  
       stream::ansi(codePage);  
       stream::binary().
```

Параметр `CodePage` указывает кодовую страницу для символов ANSI.

4. `Position = getPosition()` — получение позиции в потоке, с которой будет производиться операция чтения или записи.

5. `setCRLFconversion(Enabled)` — установка правила преобразования кодов перехода на новую строку при чтении/записи текста. Если `Enabled = true()`, то перед записью в поток осуществляется преобразование `LF → CRLF`, и после чтения из потока осуществляется преобразование `CRLF → LF`. Если `Enabled = false()`, то преобразование не осуществляется.
6. `setMode(Mode)` — установка режима потока.
7. `setPosition(Position)` — установка позиции в потоке, с которой будет производиться операция чтения или записи.

ПРИЛОЖЕНИЕ 6

Меню Visual Prolog

В этом приложении приводится описание пунктов меню графической среды разработки Visual Prolog (табл. П6.1).

Таблица П6.1. Описание меню графической среды разработки Visual Prolog

Пункты меню	Назначение
File	
New in New Package...	Создать новый элемент проекта в отдельном каталоге
New in Existing Package...	Создать новый элемент проекта в каталоге проекта
Add...	Добавить в проект новый модуль
Open...	Открыть файл в окне редактора
Close	Закрыть активное окно
Save	Сохранить текст активного окна
Save As...	Сохранить текст активного окна в указанном файле
Print....	Печать текста активного окна
Exit	Выход из среды Visual Prolog
Edit	
Undo	Отмена последнего действия в редакторе
Redo	Вернуть отмену
Cut	Вырезать выделенный текст в буфер обмена
Copy	Копировать выделенный текст в буфер обмена
Paste	Вставить из буфера обмена
Delete	Удалить выделенный текст
Copy Position	Копировать описание и позицию ошибки в буфер обмена
Find...	Искать текст в активном окне
Find Previous	Искать текст назад

Таблица П6.1 (продолжение)

Пункты меню	Назначение
Find Next	Искать текст дальше
Find in Files...	Искать текст в файлах
Replace...	Заменить
Upper Case	Перевести выделенный текст в верхний регистр
Lower Case	Перевести выделенный текст в нижний регистр
Toggle Case	Обратить регистр букв выделенного текста
Comment Lines	Закомментировать строки выделенного текста
Uncomment Lines	Раскомментировать строки выделенного текста
Select All	Выделить весь текст активного окна
Toggle Bookmark	Установить/Снять закладку
View	
Project Window	Открыть окно проекта
Error/Warnings Window	Открыть окно сообщений об ошибках
Messages	Открыть окно сообщений компилятора
Text Windows	Сделать активным одно из окон исходного текста проекта
Resource Windows	Открыть окно ресурсов
Code Expert...	Открыть или сделать активным окно эксперта кода
Run Stack	Открыть окно стека вызовов
Variables for Current Clause	Открыть окно переменных текущего предложения при отладке
Watch Window	Открыть окно просмотра в отладчике
Facts	Открыть окно фактов при отладке
Threads	Открыть окно потоков (тредов) при отладке
Modules	Открыть окно загруженных модулей при отладке
Exceptions	Открыть окно исключений
Breakpoints	Открыть окно точек останова при отладке
Bookmarks	Открыть окно закладок
Last Search Results Window	Просмотр результатов последнего поиска
TODO	Просмотр всех TODO комментариев в файлах исходного кода
Disassembly	Открыть окно дизассемблера при отладке
Disassembly from Current Line	Дизассемблировать текущую строку
Registers	Открыть окно регистров при отладке
Memory Dump	Открыть окно с дампом памяти при отладке
Zoom In	Увеличить размер шрифта

Таблица П6.1 (продолжение)

Пункты меню	Назначение
Zoom Out	Уменьшить размер шрифта
Zoom to Normal	Нормальный (№ 8) размер шрифта
Insert	
Qualification...	Вставить класс как область видимости
New GUID	Вставить новый глобальный уникальный идентификатор (см. MSDN)
RGB Value...	Вставить код цвета
Font Name...	Вставить имя и размер шрифта
File Name...	Вставить имя файла
Directory Name...	Вставить имя каталога
#include...	Вставить директиву <code>include</code> для файла заголовка пакета (Ph-файла)
#bininclude...	Вставить директиву <code>bininclude</code> для двоичного файла
Date Stamp	Вставить закомментированную текущую дату
Project	
New...	Создать новый проект
Open...	Открыть проект
Close	Закрыть проект
Save	Сохранить проект
Settings...	Открыть окно установок проекта
Build	
Compile	Компилировать проект
Build	Построить исполнимый файл проекта
ReBuild All	Построить заново проект
Stop Building	Остановить построение проекта
Execute	Построить проект и запустить на выполнение
Run in Window	Построить проект и запустить на выполнение в консольном окне
Build All Platforms	Построить исполнимый файл для всех платформ
Rebuild All Platforms	Построить заново проект для всех платформ
32bit Platform	32-битная целевая платформа
64bit Platform	64-битная целевая платформа
Script Preview	Скрипт построения проекта

Таблица П6.1 (продолжение)

Пункты меню	Назначение
Debug	
Run	Запустить проект в отладчике
Run Skipping Soft Breakpoints	Запустить проект в отладчике с игнорированием программных точек останова
Stop Debugging	Завершить отладку
Break Program	Останов программы в отладчике (пауза)
Restart	Перезапуск проекта в отладчике
Attach Process	Присоединить процесс до запуска отладчика
Step Over	Выполнение предиката целиком за один шаг отладки
Step Into	Пошаговое выполнение внутри предиката
Step out of	Выполнение предиката за один шаг
Run to Cursor	Выполнение до позиции курсора
Go to Executing Predicate Source	Вернуться в окно отладки к выполняемому предикату
Go to Variable	Вернуться в окно просмотра переменных
Break on Exception	Остановка программы при возникновении исключительной ситуации. После этого в окне стека можно найти предикат, вызвавший исключение
Toggle Breakpoint	Установить/удалить точку останова
Remove All Breakpoints	Удаление всех точек останова
Go To	
Source Browser	Отобразить окно поиска в проекте
Go to Declaration	Перейти к объявлению предиката
Go to Definition	Перейти к предложениям предиката
Go to Usage	Показать в отдельном окне все вызовы предиката в проекте. Предикат должен быть отмечен указателем курсора
Go to Related Files...	Перемещение по всем окнам с исходным текстом
Go to Line Number...	Перейти к строке №
Go to Position on Clipboard	Перейти к позиции, хранящейся в буфере обмена
Go to Next Bookmark	Перейти к следующей закладке
Go to Previous Bookmark	Перейти к предыдущей закладке
Go to Next Error (Warning)	Перейти к следующей ошибке (предупреждению)
Go to Previous Error (Warning)	Перейти к предыдущей ошибке (предупреждению)
Go Back	Вернуться назад
Locate in Project Tree	Локализовать в дереве проекта

Таблица П6.1 (продолжение)

Пункты меню	Назначение
Refactor	
Optimal Set of Includes	Оптимизировать набор включаемых в проект файлов
View Unused Items	Вывести на экран неиспользуемые пакеты и папки проекта
Move	Переместить выделенные в дереве проекта папки или пакеты
Rename	Переименовать выделенные в дереве проекта папки или пакеты
Namespace Handling...	Установить/удалить пространство имен Namespace во всех классах (интерфейсах) в указанной папке
Tools	
Memory Profiler...	Построение профиля выполнения программы, в случае использования в программе профилировщика
Configure Tools...	Назначение собственных быстрых клавиш для использования других приложений и библиотек в среде Visual Prolog
Source Control Repository...	Хранилище элементов управления
IDE Variables	Назначение переменных окружения среды Visual Prolog
Options...	Опции для настройки среды Visual Prolog
Web	
Tutorials	Открывает описание Visual Prolog по адресу: http://wiki.visual-prolog.com/index.php?title=Category:Tutorials
Wiki	Википедия по Visual Prolog по адресу: Main_Page">http://wiki.visual-prolog.com/index.php?title>Main_Page
Discussion Forum	Англоязычный форум по Visual Prolog по адресу: http://discuss.visual-prolog.com/
Check for Updates	Проверка наличия новых версий Visual Prolog
My Registration	Регистрация Visual Prolog по адресу: http://order.pdc.dk/vip/
Visual Prolog	Перейти на сайт Visual Prolog: http://www.visual-prolog.com/
Prolog Development Center	Открывает сайт Prolog Development Center: http://www.pdc.dk/
Window	
Cascade	Расположить все окна уступом вправо-вниз. Работает при разблокированных окнах
Horizontal Tile	Расположить окна друг под другом, вытянув их горизонтально

Таблица П6.1 (окончание)

Пункты меню	Назначение
Vertical Tile	Расположить окна друг под другом, вытянув их вертикально
Arrange Icons	Расположить по значкам
Lock All Editor Windows together	Заблокировать/разблокировать все окна открытых файлов в одном окне редактора
Close All Editor Windows Except the Active	Закрыть все окна редактора, кроме активного окна
Prune Editor Windows	Оставить открытыми те 10 окон редактора, которые посещались в последнюю очередь. Остальные закрыть
Help	
Visual Prolog Help	Открывает справку по Visual Prolog
Context Help	Открывает справку для выделенного контекста (слова)
Language Reference (wiki)	Открывает описание языка по адресу: http://wiki.visual-prolog.com/index.php?title=Language_Reference
Language Reference (book)	Загрузка руководства по Visual Prolog в формате PDF
Getting Started	Открывает раздел справки «Быстрое начало в Visual Prolog»
Send Bug Report...	Отправление разработчику отчета об ошибке, найденной в Visual Prolog
Manage License...	Активация/деактивация лицензии
Install Examples...	Инсталляция библиотеки примеров
Restore File Associations	Создать/восстановить ассоциативную связь с файлами проекта по их расширению
About	Информация о версии Visual Prolog и лицензии

ПРИЛОЖЕНИЕ 7

Быстрые клавиши меню Visual Prolog

В этом приложении приведено описание комбинаций клавиш быстрого доступа к пунктам меню графической среды Visual Prolog (табл. П7.1).

Таблица П7.1. Описание клавиш быстрого доступа к пунктам меню

Функция	Клавиши
Редактирование	
Копировать в буфер обмена	<Ctrl>+<C> или <Ctrl>+<Ins>
Вырезать в буфер обмена	<Ctrl>+<X> или <Shift>+
Вставить из буфера обмена	<Ctrl>+<V> или <Shift>+<Ins>
Отменить последнее действие в редакторе	<Ctrl>+<Z>
Вернуть отмену	<Ctrl>+<Y> или <Ctrl>+<Shift>+<Z>
Выделить все	<Ctrl>+<A>
Выделить текущее слово	<Ctrl>+<W>
Активировать или вызвать Эксперт кода	<Ctrl>+<Shift>+<W>
Удалить строку	<Ctrl>+<Shift>+<L>
Удалить от курсора до конца строки	<Ctrl>+<Shift>+<E>
Удалить от курсора до конца слова	<Ctrl>+
Вставить предикат из списка предикатов	<Ctrl>+<Shift>+<I>
Вызвать диалоговое окно замены текста	<Ctrl>+<H>
Привести к нижнему регистру выделенный текст	<Ctrl>+<U>
Привести к верхнему регистру выделенный текст	<Ctrl>+<Shift>+<U>
Обратить регистр букв выделенного текста	<Ctrl>+<Alt>+<U>
Увеличить размер шрифта	<Ctrl>+<+> или <Ctrl>+<Scroll вверх>
Уменьшить размер шрифта	<Ctrl>+<-> или <Ctrl>+<Scroll вниз>

Таблица П7.1 (продолжение)

Функция	Клавиши
Нормальный размер шрифта (100 %)	<Ctrl>+<0> (ноль)
Закомментировать строку	<Ctrl>+<Alt>+<%>
Снять комментарий строки	<Ctrl>+<Alt>+<Shift>+<%>
Вставить закомментированную текущую дату	<Ctrl>+<Shift>+<Y>
Выбор области видимости предиката из списка классов	<Ctrl>+<Shift>+<S>
Установить/Снять закладку	<Ctrl>+<K>
Навигация	
Найти текст в активном окне	<Ctrl>+<F>
Найти текст в файлах	<Ctrl>+<Shift>+<F>
Искать дальше	<F3>
Локализовать в дереве проекта	<Ctrl>+<T>
Перейти к предложениям выделенного предиката	<F12> или <Ctrl>+<Shift>+<C>
Перейти к объявлению выделенного предиката	<Ctrl>+<F12> или <Ctrl>+<Shift>+D
Перейти к строке с номером	<Ctrl>+<F2>
Перейти к позиции следующей ошибки	<F8>
Перейти к позиции ошибки, хранящейся в буфере обмена	<Shift>+<F2>
Вернуться к позиции предыдущей ошибки	<Shift>+<F8>
Вызвать диалоговое окно навигации окон проекта	<Ctrl>+<Tab> или <Ctrl>+<F6>
Вернуться назад	<Ctrl>+<Shift>+<F8>
Компиляция	
Построить исполнимый файл проекта	<Ctrl>+<Shift>+
Остановить построение проекта	<Ctrl>+<Break>
Построить только ресурсы	<Alt>+<F8>
Компилировать	<Ctrl>+<F7>
Построить проект заново	<Ctrl>+<Shift>+<Alt>+
Запустить проект	<Ctrl>+<F5>
Запустить проект в консольном окне	<Alt>+<F5>
Отладка	
Запуск проекта в отладчике	<F5>
Остановить отладчик	<Shift>+<F5>
Свойства точки останова	<Alt>+<F9>
Обновить окно отладки	<Ctrl>+<R>
Отобразить окно регистров	<Ctrl>+<Alt>+<G>
Отобразить окно точек останова	<Ctrl>+<Alt>+

Таблица П7.1 (окончание)

Функция	Клавиши
Отобразить окно стека вызовов	<Ctrl>+<Alt>+<C>
Отобразить окно дизассемблера	<Ctrl>+<Alt>+<D>
Отобразить окно просмотра переменных	<Ctrl>+<Alt>+<V>
Отобразить окно дампа памяти	<Ctrl>+<Alt>+<M>
Отобразить окно потоков	<Ctrl>+<Alt>+<H>
Перейти по адресу	<Ctrl>+<G>
Вернуться в окно отладки к выполняемому предикату	<Ctrl>+<E>
Перейти к окну фактов при отладке	<Ctrl>+<Alt>+<F>
Включить/выключить точку отладки	<Ctrl>+<F9>
Отобразить окно просмотра	<Ctrl>+<Alt>+<W>
Установить/удалить точку отладки	<F9>
Пошаговое выполнение внутри предиката	<F11>
Завершить выполнение предиката	<Shift>+<F11>
Выполнение предиката за один шаг отладки	<F10>
Выполнение до позиции курсора	<Ctrl>+<F10>
Проект	
Создать новый проект	<Ctrl>+<Shift>+<N>
Открыть проект	<Ctrl>+<Shift>+<O>
Добавить модуль в проект	<Ctrl>+<Shift>+<A>
Создать элемент проекта в новом пакете	<Ctrl>+<N>
Создать элемент проекта в существующем пакете	<Ctrl>+<Alt>+<N>
Среда разработки	
Закрыть все окна редактора	<Ctrl>+<Q>
Закрыть неактивные окна редактора	<Ctrl>+<Shift>+<Q>
Закрыть активное окно редактора	<Alt>+<F4>
Отобразить окно проекта	<Ctrl>+<Alt>+<P>
Отобразить окно установок проекта	<Alt>+<F7>
Сохранить файл	<F2>
Сохранить проект	<Ctrl>+<S>
Отобразить окно поиска в проекте	<Ctrl>+
Отобразить окно ошибок	<Ctrl>+<Alt>+<E>
Справка	<F1>
Отобразить окно вывода	<Ctrl>+<Alt>+<O>
Печать	<Ctrl>+<P>

ПРИЛОЖЕНИЕ 8

Панель инструментов Visual Prolog

В этом приложении приведено описание кнопок панели инструментов графической среды Visual Prolog (табл. П8.1). Кнопки панели инструментов дублируют основные пункты меню Visual Prolog.

Таблица П8.1. Описание кнопок панели инструментов Visual Prolog

Кнопка	Назначение
	Создать новый элемент проекта: класс, окно, диалог, значок и т. п.
	Открыть файл в текстовом редакторе
	Сохранить в файле содержимое окна текстового редактора
	Отменить последнее действие в редакторе
	Вернуть отмену
	Вырезать в буфер обмена
	Копировать в буфер обмена
	Вставить из буфера обмена
	Выбрать разрядность платформы (32 или 64 бита)
	Компилировать модуль
	Построить проект
	Построить проект и запустить на выполнение
	Отобразить окно проекта Project Window
	Отобразить окно сообщений Message Window
	Отобразить окно переменных в режиме отладки
	Отобразить окно дерева фактов в режиме отладки

Таблица П8.1 (окончание)

Кнопка	Назначение
	Отобразить окно контрольных точек останова
	Отобразить окно стека вызовов предикатов
	Отобразить окно потоков (тредов)
	Поиск в файлах
	Запуск проекта в отладчике
	Запуск проекта в отладчике, игнорируя точки останова
	Останов проекта в отладчике (пауза)
	Завершение отладки
	Выполнение шага внутрь предиката
	Выполнение предиката целиком за один шаг
	Выполнение до позиции курсора
	Выход из предиката за один шаг
	Остановка программы при возникновении исключительной ситуации. После этого в окне стека можно найти предикат, вызвавший исключение
	Установка контрольной точки останова в любой строке исходного кода
	Форум: http://discuss.visual-prolog.com/
	Руководство: http://wiki.visual-prolog.com/index.php?title=Category:Tutorials
	Контекстная справка в http://wiki.visual-prolog.com

ПРИЛОЖЕНИЕ 9

Описание предикатов класса *std*

Это приложение содержит описание предикатов класса *std*, которые, по сути, являются итераторами для циклов.

1. $I = \text{between}(\text{From}, \text{To})$ — функция возвращает значения I в диапазоне от From до To . Когда $\text{From} < \text{To}$, то значения I увеличиваются с шагом 1, иначе — уменьшаются.

Пример использования:

```
L = [I || I = between(5,2)]
```

Результат:

```
[5,4,3,2]
```

2. $I = \text{betweenInStep}(\text{From}, \text{To}, \text{Step})$ — функция возвращает значения I в диапазоне от From до To с шагом Step . Когда $\text{From} < \text{To}$, то значения I увеличиваются с шагом Step , иначе — уменьшаются.

Пример использования:

```
L = [I || I = betweenInStep(9,4,2)]
```

Результат:

```
[9,7,5]
```

3. $I = \text{cIterate}(N)$ — функция возвращает значения I в диапазоне от 0 до $N-1$.

Пример использования:

```
L = [I || I = cIterate(4)]
```

Результат:

```
[0,1,2,3]
```

4. $I = \text{countDownFrom}(N)$ — функция делает обратный отсчет, начиная от N . По достижении наименьшего числа, для 32-разрядного *integer* это -2147483648 , функция начинает обратный отсчет, начиная с наибольшего положительного числа.

Пример использования:

```
foreach I = countDownFrom(-2147483647) do
    write(I), _=readchar()
end foreach
```

Результат:

```
-2147483647
-2147483648
2147483647
2147483646
```

5. $I = \text{countDownFrom}(N, Step)$ — функция делает обратный отсчет, начиная от N с шагом $Step$. По достижении отрицательного числа, для 32-разрядного `integer` это -2147483648 , функция начинает обратный отсчет, начиная с наибольшего положительного числа.

Пример использования:

```
foreach I = countDownFromInStep(-2147483646, 2) do
    write(I), _=readchar()
end foreach
```

Результат:

```
-2147483646
-2147483648
2147483646
2147483644
```

6. $I = \text{countFrom}(N)$ — функция делает отсчет, начиная с N . По достижении наибольшего положительного числа, для 32-разрядного `integer` это 2147483647 , функция начинает отсчет, начиная с наименьшего отрицательного числа.

Пример использования:

```
foreach I = countFrom(2147483646) do
    write(I), _=readchar()
end foreach
```

Результат:

```
2147483646
2147483647
-2147483648
-2147483647
```

7. $I = \text{countFromInStep}(N, Step)$ — функция делает отсчет, начиная с N с шагом $Step$. По достижении наибольшего положительного числа, для 32-разрядного `integer` это 2147483647 , функция начинает отсчет, начиная с наименьшего отрицательного числа.

Пример использования:

```
foreach I = countFromInStep(2147483645, 2) do
    write(I), _=readchar()
end foreach
```

Результат:

```
2147483645  
2147483647  
-2147483647  
-2147483645
```

8. $I = \text{downTo}(\text{From}, \text{To})$ — функция делает обратный отсчет, начиная от From и заканчивая To, с шагом 1.

Пример использования:

```
L = [I | I = downTo(5, 2)]
```

Результат:

```
[5, 4, 3, 2]
```

9. $I = \text{downToInStep}(\text{From}, \text{To}, \text{Step})$ — функция делает обратный отсчет, начиная от From и заканчивая To, с шагом Step.

Пример использования:

```
L = [I | I = downToInStep(9, 4, 2)]
```

Результат:

```
[9, 7, 5]
```

10. $I = \text{fromTo}(\text{From}, \text{To})$ — функция делает отсчет, начиная от From и заканчивая To, с шагом 1.

Пример использования:

```
L = [I | I = fromTo(2, 5)]
```

Результат:

```
[2, 3, 4, 5]
```

11. $I = \text{fromToInStep}(\text{From}, \text{To}, \text{Step})$ — функция делает отсчет, начиная от From и заканчивая To, с шагом Step.

Пример использования:

```
L = [I | I = fromToInStep(4, 9, 2)]
```

Результат:

```
[4, 6, 8]
```

12. `repeat()` — предикат успешен бесконечное число раз. Пояснение работы этого предиката изложено в главе 10.

ПРИЛОЖЕНИЕ 10

Описание класса *string*

В этом приложении приведено описание констант, доменов и предикатов класса *string*.

Константа

Конец строки: eos = '\u0000'

Домены

Виды размещения строки в заданном поле:

```
adjustBehaviour =
    expand();           % поле расширить до длины строки
    cutRear();          % обрезать строку справа под размер поля
    cutOpposite().     % обрезать строку со стороны, противоположной прижиму
adjustSide =
    left();            % прижим строки влево
    right().           % прижим строки вправо
```

Виды чувствительности к регистру:

```
caseSensitivity =
    caseSensitive();    % чувствительность к регистру
    caseInsensitive(); % нечувствительность к регистру
    casePreserve().    % сохранение регистра каждого позиции символов
```

Предикаты

1. Str = adjust(Src, FieldSize, adjustSide) — функция размещения строки Src в поле размером FieldSize.

Пример успешного вызова функции:

```
"a      " = adjust("a",5,left) - строка короче поля, прижим влево
"      a" = adjust("a",5,right) - строка короче поля, прижим вправо
```

"tes" = adjust("test.",3,left) — строка длиннее поля, прижим влево
 "tes" = adjust("test.",3,right) — строка длиннее поля, прижим вправо

2. Str = adjust(Src, FieldSize, Padding, adjustSide) — функция размещения строки Src в поле размером FieldSize. Padding — строка для дополнения строки до размеров поля.

Пример успешного вызова функции:

"a****" = adjust("a",5,"*",left) — строка короче поля, прижим влево
 "****a" = adjust("a",5,"*",right) — строка короче поля, прижим вправо
 "tes" = adjust("test.",3," ",left) — строка длиннее поля, прижим влево
 "tes" = adjust("test.",3," ",right) — строка длиннее, прижим вправо

3. Str = adjust(Src, FieldSize, Padding, adjustBehaviour, adjustSide) — функция размещения строки Src в поле размером FieldSize. Padding — строка для дополнения строки до размеров поля. adjustBehaviour — способ обрезания строки.

Пример успешного вызова функции:

"test string." = adjust("test string.",3," ",expand,left)
 "tes" = adjust("test string.",3," ",cutOpposite,left)
 "ng." = adjust("test string.",3," ",cutOpposite,right)
 "a****" = adjust("a",5,"*",expand,left)
 "****a" = adjust("a",5,"*",expand,right)

4. Str = adjustLeft(Src, FieldSize) — функция размещения строки Src в поле размером FieldSize с прижимом влево.

Пример успешного вызова функции:

"a " = adjustLeft("a",5) — строка короче поля
 "" = adjustLeft("test string.",0) — строка длиннее поля
 "tes" = adjustLeft("test string.",3) — строка длиннее поля

5. Str = adjustLeft(Src, FieldSize, Padding) — функция размещения строки Src в поле размером FieldSize с прижимом влево.

Пример успешного вызова функции:

"a****" = adjustLeft("a",5,"*")

6. Str = adjustLeft(Src, FieldSize, Padding, adjustBehaviour) — функция размещения строки Src в поле размером FieldSize с прижимом влево.

Пример успешного вызова функции:

"tes" = adjustLeft("test.",3,"",cutRear) — строку обрезать справа
 "test." = adjustLeft("test.",3,"",expand) — строку не укорачивать

7. Str = adjustRight(Src, FieldSize) — функция размещения строки Src в поле размером FieldSize с прижимом вправо.

Пример успешного вызова функции:

"" = adjustRight("test string.",0) — строка длиннее поля
 "tes" = adjustRight("test string.", 3) — строка длиннее поля

8. Str = adjustRight(Src, FieldSize, Padding) — функция размещения строки Src в поле размером FieldSize с прижимом вправо.
9. Str = adjustRight(Src, FieldSize, Padding, adjustBehaviour) — функция размещения строки Src в поле размером FieldSize с прижимом вправо.

Пример успешного вызова функции:

```
"tes" = adjustRight("test string.", 3, "", cutRear)  
"test." = adjustRight("test.", 3, "", expand)  
"ng." = adjustRight("test string.", 3, "", cutOpposite)
```

10. StringList = arrayToList(ArrayOfStrings, Count) — функция преобразует массив строк ArrayOfStrings, представленный в стиле языка C, в список строк StringList. StringList — список строк. ArrayOfStrings — указатель на массив строк в стиле языка C. Count — количество строк в массиве.

11. CharInLowerCase = charLower(Char) — функция преобразует символ Char в нижний регистр CharInLowerCase.

```
'a' = charLower('A')
```

12. CharAsString = charToString(Char) — функция преобразует символ Char в строку CharAsString, содержащую один этот символ.

```
"A" = charToString('A')
```

13. CharInUpperCase = charUpper(Char) — функция преобразует символ Char в верхний регистр CharInUpperCase.

```
'A' = charLower('a')
```

14. CompareResult = compareIgnoreCase(String1, String2, Locale) — функция сравнения строк, нечувствительная к регистру. Результат сравнения:

```
CompareResult = greater(); equal(); less().
```

Пример успешного вызова функции:

```
equal = compareIgnoreCase("STRING", "string",  
                         locale_native::locale_system_default)  
equal = compareIgnoreCase("STRING", "string")  
less = compareIgnoreCase("STRING", "STRINGGGG")  
greater = compareIgnoreCase("STRING", "STR")
```

15. Функция соединения нескольких строк в одну. Количество соединяемых строк от двух до шести:

```
String = concat(Str1, Str2)  
String = concat(Str1, Str2, Str3)  
String = concat(Str1, Str2, Str3, Str4)  
String = concat(Str1, Str2, Str3, Str4, Str5)  
String = concat(Str1, Str2, Str3, Str4, Str5, Str6)
```

Пример успешного вызова функции:

```
"04.03.2013" = concat("04", ".", "03", ".", "2013")
```

16. `String = concatList(StringList)` — функция соединения списка строк в одну строку `String`.

Пример успешного вызова функции:

```
"04.03.2013" = concatList(["04", ".", "03", ".", "2013"])
"" = concatList([])
```

17. `String = concatWithDelimiter(StringList,Delimiter)` — функция соединения списка строк в одну строку с вставкой разделятеля `Delimiter`.

Пример успешного вызова функции:

```
"пи**пи**3.14" = concatWithDelimiter(["pi","пи","3.14"], "***")
"пипи3.1415" = concatWithDelimiter(["pi","пи","3.14"], "")
"" = concatWithDelimiter([], "***")
```

18. `String = create(Count)` — функция создает строку, содержащую `Count` пробелов.

Пример успешного вызова функции:

```
"      " = create(3)
```

19. `String = create(CharCount, StringItem)` — функция создает строку `String` длиной `CharCount` символов путем повторения строки `StringItem`.

Пример успешного вызова функции:

```
"ABABA" = create(5, "AB")
"" = create(0, "AB")
```

20. `String = createCopy(String)` — функция создает копию строки `String`.

Пример успешного вызова функции:

```
"Some string" = createCopy("Some string")
```

21. `String = createFromCharList(CharList)` — функция создает строку `String` из списка символов `CharList`.

Пример успешного вызова функции:

```
"Prolog" = createFromCharList(['P', 'r', 'o', 'l', 'o', 'g'])
```

22. `equalIgnoreCase(String1, String2)` — предикат успешен, если строки одинаковы при нечувствительном к регистру сравнении. Верхний и нижний регистры буквы определяются для того языка, который выбран пользователем при локализации Windows.

Пример успешного вызова предиката:

```
equalIgnoreCase ("PROLOG", "Prolog")
```

23. `equalIgnoreCase(String1, String2, LanguageID)` — предикат успешен, если строки одинаковы при нечувствительном к регистру сравнении. Параметр `LanguageID` определяет язык, для которого будут определяться верхний и нижний регистры буквы.

Пример успешного вызова предиката:

```
equalIgnoreCase("PROLOG", "Prolog", locale_native::locale_user_default)
equalIgnoreCase("PROL", "Prol", locale_native::locale_system_default)
```

24. `String = format(FormatString, ...Parameters)` — функция создает строку `String` по шаблону, определенному строкой `FormatString`. Описание шаблона приведено в главе 12 при рассмотрении предиката `writef`.
25. `StringLast = fromPosition(String, Position)` — функция возвращает правую часть `StringLast` строки `String`, начиная с позиции `Position`. Счет позиций начинается с нуля.

Пример успешного вызова функции:

```
"Prolog" = fromPosition("Prolog", 0)
"olog" = fromPosition("Prolog", 2)
```

Пример генерирует исключительную ситуацию:

```
StringLast = fromPosition("Prolog", 10)
```

26. `front(String, CharCount, First [out], Last [out])` — предикат разделяет строку `String` на две строки: `First` и `Last`. Длина строки `First` должна быть равна `CharCount` символов. Когда параметр `CharCount` превышает длину строки `String`, то вызывается исключительная ситуация.

Пример успешного вызова предиката:

```
front("Prolog", 2, First, Last)
First="Pr"    Last="olog"
```

Пример генерирует исключительную ситуацию:

```
front("Prolog", 12, First, Last)
```

27. `Char = frontChar(String)` — функция возвращает первый символ строки или символ конца строки '\u0000' (eos), если строка `String` пустая.

```
'A' = frontChar("ABCD")
'\u0000' = frontChar("")
```

28. `frontChar(String, Char [out], LastString [out])` — предикат возвращает первый символ `Char` и остаток строки `LastString`.

```
frontChar("Prolog", Char, String)
Char='P'    Last="olog"
```

29. `frontToken(String, Token [out], RestString [out])` — предикат возвращает первую лексему `Token` и остаток строки `RestString`. Если строка содержит только пробелы, табуляции и коды перевода строки, или пуста, то предикат будет неуспешен. Порядок отделения лексемы:

- удаление пробелов, табуляций и кодов перевода строки до встречи первого печатаемого символа;

- определение типа первого символа (буква; цифра; знак подчеркивания; префикс десятичного числа или восьмеричного или шестнадцатеричного; десятичное число; другие символы, включая "+" и "-");
- к первому символу добавляются все символы, принадлежащие к определенному в пункте 2 типу. Они составляют лексему Token;
- если первый символ — цифра, то следующие символы рассматриваются как символы числа соответствующего типа ($3.1415e+5$ — десятичного, $0xFFFFFFF$ — шестнадцатеричного и т. д.);
- когда тип очередного символа меняется, то он и оставшиеся символы возвращаются в виде остатка строки RestString.

Пример успешного вызова предиката:

```
frontToken("\t\t\t3.1415e+5abcd", Token, Rest),
  Token = "3.1415e+5", Rest = "abcd"
frontToken("\t \t \t abc3.1415e+5", Token, Rest),
  Token = "abc3", Rest = ".1415e+5"
frontToken("-3.1415e+5", Token, Rest),
  Token = "-", Rest = "3.1415e+5"
frontToken("0xFFFFF", Token, Rest),
  Token = "0", Rest = "xFFFF"
```

30. `getBinaryPrefix(Number, Base, Prefix)` — предикат возвращает множитель `Base` и префикс `Prefix` для заданного количества информации, выраженного вещественным числом `Number`, по правилам стандарта памяти JEDEC 100B.01. Согласно этим правилам количество информации `Number` может выражаться с помощью префиксов К, М, Г и т. д., которые понимаются как множители 2^{10} , 2^{20} и 2^{30} соответственно. Например, вызов `getBinaryPrefix(123392, Base, Prefix)` вернет `Base = 1024` и `Prefix = к`. Следовательно, число 123392 можно записать в килобайтах, вычислив значение выражения $123392/1024$, с использованием приставки КВ, т. е. 120.5 КВ.
31. `Char = getCharFromValue(Code)` — функция возвращает символ `Char`, имеющий 16-разрядный код `Code`.


```
'A' = getCharFromValue(65)
```
32. `Code = getCharValue(Char)` — функция возвращает 16-разрядный код `Code` заданного символа `Char`.


```
65 = getCharValue('A')
```
33. `getMetricPrefix(Number, Base, Prefix)` — предикат форматирует вещественное число `Number` согласно международной системе измерения СИ, возвращая при этом множитель `Base` и префикс `Prefix`. Например, вызов `getMetricPrefix(1234, Base, Prefix)` вернет `Base = 1000` и `Prefix = к`. Следовательно, число 1234 можно записать в метрической системе, вычислив значение выражения $1234/1000$, с использованием приставки К, т. е. 123.4 К. С другой стороны, вызов `getMetricPrefix(0.0012, Base, Prefix)` вернет `Base = 0.001` и

Prefix = m. Следовательно, число 0.0012 можно записать в метрической системе, вычислив значение выражения $0.0012/0.001$, с использованием приставки m, т. е. 1.2 m.

34. `hasAlpha(String)` — предикат успешен, если строка `String` содержит только буквы.

Пример успешного вызова предиката:

```
hasAlpha("Visual Prolog")
```

Пример неуспешного вызова предиката:

```
hasAlpha("Visual Prolog 7.4")
hasAlpha("Visual Prolog!")
```

35. `hasDecimalDigits(String)` — предикат успешен, если строка `String` содержит только десятичные цифры.

Пример успешного вызова предиката:

```
hasDecimalDigits("1234567890")
```

Пример неуспешного вызова предиката:

```
hasDecimalDigits("-1.23")
hasDecimalDigits("Visual Prolog 7.4")
```

36. `HashValue = hash(String)` — функция возвращает значение хэш-функции от строки `String`.

```
2728594564 = hash("Prolog")
1099903006 = hash("pi")
2634797358 = hash(" ")
82610334    = hash("")
```

37. `hasPrefix(String, Prefix, Rest [out])` — предикат успешен, если строка `String` содержит префикс `Prefix`. При этом предикат возвращает остаток строки `Rest`. Предикат неуспешен, если строка `String` не содержит префикса `Prefix`.

Пример успешного вызова предиката:

```
hasPrefix("Visual Prolog", "Vi", Rest)
Rest = "sual Prolog"
```

Пример неуспешного вызова предиката:

```
hasPrefix("Visual Prolog", "Pro", Rest)
```

38. `hasPrefixIgnoreCase(String, Prefix, Rest [out])` — предикат успешен, если строка `String` содержит префикс `Prefix`. Сравнение нечувствительно к регистру.

Пример успешного вызова предиката:

```
hasPrefixIgnoreCase("Visual Prolog", "VIS", Rest)
Rest = "ual Prolog"
```

39. `hasSuffix(String, Suffix)` — предикат успешен, если строка `String` содержит суффикс `Suffix`.

40. `hasSuffix(String, Suffix, Rest [out])` — предикат успешен, если строка `String` содержит суффикс `Suffix`. При этом предикат возвращает префикс строки `Rest`. Предикат неуспешен, если строка `String` не содержит суффикс `Suffix`.

Пример успешного вызова предиката:

```
hasSuffix("Visual Prolog", "log", Rest)  
    Rest = "Visual Pro"
```

Пример неуспешного вызова предиката:

```
hasSuffix("Visual Prolog", "Pro", Rest)
```

41. `isClassIdentifier(ClassName, MemberName)` — предикат успешен, если пара `ClassName::MemberName` является синтаксически правильным идентификатором класса.

Пример успешного вызова предиката:

```
isClassIdentifier("the_Name1", "the_Name1")  
isClassIdentifier("class", "member")  
isClassIdentifier("", "the_Name1")
```

Пример неуспешного вызова предиката:

```
isClassIdentifier("1Name", "class")  
isClassIdentifier("NoClass", "NoMember")  
isClassIdentifier("class", "")
```

42. `isLowerCase(String)` — предикат успешен, если строка `String` не содержит заглавные буквы (буквы в верхнем регистре).

Пример успешного вызова предиката:

```
isLowerCase("prolog")  
isLowerCase("12345")  
isLowerCase("")
```

Пример неуспешного вызова предиката:

```
isLowerCase("Prolog")
```

43. `isName(String)` — предикат успешен, если строка `String` представляет правильный идентификатор в соответствии с синтаксисом Visual Prolog.

Пример успешного вызова предиката:

```
isName("Some_string")  
isName("_1File")  
isName("aaa_1_bbb_2_ccc")
```

Пример неуспешного вызова предиката:

```
isName("file.txt")  
isName("abc def")  
isName("2file")
```

44. `isObjectIdentifier(String1, String2)` — предикат успешен, если пара `ObjectName:MemberName` является синтаксически правильным объектным идентификатором.

Пример успешного вызова предиката:

```
isObjectIdentifier("The_Name1", "the_Name1")
isObjectIdentifier("Object", "member")
```

Пример неуспешного вызова предиката:

```
isObjectIdentifier("1Name", "class")
isObjectIdentifier("noObject", "NoMember")
isObjectIdentifier("_C", "_the_Name1")
```

45. `isUpperCase(String)` — предикат успешен, если строка `String` не содержит строчные буквы (буквы в нижнем регистре).

Пример успешного вызова предиката:

```
isUpperCase("PROLOG")
isUpperCase("12345")
isUpperCase("")
```

Пример неуспешного вызова предиката:

```
isUpperCase("Prolog")
```

46. `isValidToRead(StrPtr)` — предикат используется для тестирования прав доступа к байтам памяти, занятым строкой, указателем на которую является `StrPtr`. Предикат неуспешен, если нет прав доступа по чтению ко всем символам строки.

47. `isWhiteSpace(Char)` — предикат успешен, если символ `Char` представляет собой пробел ' ', символ табуляции '\t', символ перевода на новую строку '\n' и другие непечатаемые символы.

48. `isWhiteSpace(String)` — предикат успешен, если строка `String` содержит только пробелы, табуляции или коды перевода на новую строку '\u000A' и другие непечатаемые символы.

Пример успешного вызова предиката:

```
isWhiteSpace("")
isWhiteSpace(" ")
isWhiteSpace("\n \t \u000D")
isWhiteSpace(' ')
isWhiteSpace('\n')
```

Пример неуспешного вызова предиката:

```
isWhiteSpace("1 \n")
isWhiteSpace('n')
```

49. `keywordString(String, Boolean)` — предикат успешен, если строка `String` является ключевым словом языка Visual Prolog. Если флаг `Boolean` истинен, то ключевое слово старшее, иначе — младшее.

Пример успешного вызова предиката:

```
keywordString("clauses", Boolean)
    Boolean = true
keywordString("if", Boolean)
    Boolean = false
```

Пример неуспешного вызова предиката:

```
keywordString("abc", Boolean)
```

50. `LastChar = lastChar(String)` — функция возвращает последний символ `LastChar` строки `String`.

Пример успешного вызова предиката:

```
'a' = lastChar("a")
'D' = lastChar("ABCD")
'\u0000' = lastChar("")
```

51. `lastChar(String, Prefix, LastChar)` — предикат разделяет строку `String` на последний символ `LastChar` и начало строки `Prefix`. Предикат неуспешен, если строка `String` пустая.

Пример успешного вызова предиката:

```
lastChar("ABCD", Prefix, Char)
    Char = 'D', Prefix = "ABC"
lastChar("a", Prefix, Char),
    Char = 'a', Prefix = ""
lastChar("qwe ", Prefix, Char),
    Char = ' ', Prefix = "qwe"
```

Пример неуспешного вызова предиката:

```
lastChar("", String, Char)
```

52. `Length = length(String)` — функция возвращает длину строки `String`. Под длиной строки понимается количество символов, а не байтов. Последний символ нуль-терминальных строк '`\u0000`' является признаком конца строки и не учитывается.

Пример успешного вызова функции:

```
5 = length("12345")
0 = length("")
```

53. `listToArray(StringList,ArrayOfStrings,Count)` — предикат преобразует список строк `StringList` в массив строк языка С. `Count` — число строк в массиве. `ArrayOfStrings` — указатель на массив строк.

54. `LastString = rear(String,CharCount)` — функция возвращает суффикс `LastString`, содержащий последние `CharCount` символов строки `String`.

Пример вызова функции:

```
"" = rear("abcd",0)
"A" = rear("A",5)
```

```
"dart test " = rear("Standart test ",10)
"" = rear("",0)
```

55. `NewString = replaceAll(String,What,With)` — функция заменяет все вхождения строки `What` строкой `With` в исходной строке `String`. Результатом является новая строка `NewString`. Размеры строк `What` и `With` могут различаться. Поиск нечувствительный к регистру.

Пример вызова функции:

```
"asd**qwe**zxc" = replaceAll("asd qwe zxc"," ","**")
"Q 1 le" = replaceAll("Q QW qwe","qw","1")
"" = replaceAll","", "qw", "1")
```

56. `NewString = replaceAll(String,What,With,Case)` — функция заменяет все вхождения строки `What` строкой `With` в исходной строке `String`. Учет регистра определяется параметром `Case`. Результатом является новая строка `NewString`. Размеры строк `What` и `With` могут различаться. Параметр `Case` может иметь одно из трех значений:

- `caseSensitive()` — с учетом регистра;
- `caseInsensitive()` — без учета регистра;
- `casePreserve()` — поиск без учета регистра, но сохранение регистра каждого позиции символов новой строки таким, каким он был в исходной строке.

Пример вызова функции:

```
"Aa AaW qwe"=replaceAll("Q QW qwe","Q","Aa",caseSensitive())
"aa AaW aaWe" = replaceAll("q QW qwe","Q","Aa",casePreserve())
```

57. `NewString = replaceAll(String,What,With,Case,Number)` — функция заменяет все вхождения строки `What` строкой `With` в исходной строке `String` и возвращает количество произведенных замен `Number`. Учет регистра определяется параметром `Case`. Результатом является новая строка `NewString`. Размеры строк `What` и `With` могут различаться.

Пример вызова функции:

```
S = replaceAll("Q QW qwe","Q","Aa",caseSensitive(),Number)
S = "Aa AaW qwe", Number = 2
S = replaceAll("Prolog","qw","1",caseSensitive(),Number)
S = "Prolog", Number = 0
```

58. `NewString = replaceFirst(String,What,With,Case)` — функция заменяет первое вхождение строки `What` строкой `With` в исходной строке `String`.

59. `NewString = replaceFirst(String,What,With,Case)` — функция заменяет первое вхождение строки `What` строкой `With` в исходной строке `String`. Учет регистра определяется параметром `Case`. Результатом является новая строка `NewString`. Размеры строк `What` и `With` могут различаться.

Пример вызова функции:

```
S = replaceFirst("1Abc 2abC 3aBc","abC","ttt",caseSensitive)
S = "1Abc 2ttt 3aBc"
```

```
S = replaceFirst("1Abc 2abC 3aBc", "abC", "ttt", caseInsensitive)
S = "1ttt 2abC 3aBc"
S = replaceFirst("1Abc 2abC 3aBc", "abC", "ttt", casePreserve)
S = "1Ttt 2abC 3aBc"
```

60. `NewString = replacePart(String,Position,Length,With)` — функция заменяет подстроку в строке `String`, начинающуюся с позиции `Position` длиной `Length` символов, строкой `With`. Длина строки `With` может не совпадать с длиной `Length`. Если заменяемая строка выходит за пределы исходной строки, то вызывается исключительная ситуация.

Пример вызова функции:

```
"qAAAAAArty" = replacePart("qwerty",1,2,"AAAAAA")
"qAty" = replacePart("qwerty",1,3,"A")
```

61. `Position = search(String,LookFor)` — функция возвращает позицию `Position` подстроки `LookFor` в строке `String`. Функция неуспешна, если подстрока не найдена. Поиск чувствителен к регистру.

Пример успешного вызова функции:

```
7 = search("Visual prolog","pro")
```

Пример неуспешного вызова функции:

```
Position = search("Visual prolog","PRO")
```

62. `Position = search(String,LookFor,Case)` — функция возвращает позицию `Position` подстроки `LookFor` в строке `String`. Функция неуспешна, если подстрока не найдена. Чувствительность к регистру определяется параметром `Case`.

Пример успешного вызова функции:

```
3 = search("PROLOG prolog","log",caseInsensitive)
10 = search("PROLOG prolog","log",caseSensitive)
```

Пример неуспешного вызова функции:

```
Position = search("prolog","visual",caseInsensitive)
```

Вызов функции завершается исключительной ситуацией:

```
Position = search("a","",caseInsensitive)
```

63. `Position = searchChar(String,Char)` — функция возвращает позицию `Position` символа `Char` в строке `String`. Функция неуспешна, если символ не найден. Поиск чувствителен к регистру.

Пример успешного вызова функции:

```
4 = searchChar("Visual prolog",'a')
```

Пример неуспешного вызова функции:

```
Position = searchChar("Visual prolog",'A')
```

64. `Position = searchChar(String, Char, Case)` — функция возвращает позицию символа `Char` в строке `String`. Функция неуспешна если символ не найден. Чувствительность к регистру определяется параметром `Case`.

Пример успешного вызова функции:

```
3 = searchChar("Test", 't', caseSensitive)  
0 = searchChar("Test", 't', caseInsensitive)
```

Пример неуспешного вызова функции:

```
Position = searchChar("abc", 'A', caseSensitive)  
Position = searchChar("", ' ', caseInsensitive)
```

65. `Position = searchLast(String, LookFor, Case)` — функция возвращает позицию последнего вхождения подстроки `LookFor` в строку `String`. Функция неуспешна, если подстрока не найдена. Чувствительность к регистру определяется параметром `Case`.

Пример успешного вызова функции:

```
3 = searchLast("PROLOG prolog", "LOG", caseSensitive)  
10 = searchLast("PROLOG Prolog", "LOG", caseInsensitive)
```

Пример неуспешного вызова функции:

```
Position = searchLast("Prolog prolog", "LOG", caseSensitive)  
Position = searchLast("PROLOG Prolog", "prog", caseInsensitive)
```

Вызов функции завершается исключительной ситуацией:

```
Position = searchLast("a", "", caseInsensitive)
```

66. `StringList = split(String, Separators)` — функция расщепляет строку `String` на подстроки и возвращает список этих подстрок. Разделителем является тот символ строки `String`, который принадлежит строке символов-разделителей `Separators`. Сравнение символов чувствительно к регистру. Функция используется тогда, когда разделителями строки являются одиночные символы.

Пример успешного вызова функции:

```
["A", "CD", "234"] = split("ABCD1234", "1B")  
["ABCD", "234"] = split("ABCD1234", "1b")
```

67. `StringList = split_delimiter(String, Separator)` — функция расщепляет строку `String` на подстроки и возвращает список этих подстрок. Разделителем является строка `Separator`. Если строка `String` начинается (заканчивается) разделителем, то первая (последняя) подстрока в списке подстрок будет пустой. Если строка `String` не содержит разделителей, то выходной список будет содержать один элемент — исходную строку. Функция используется тогда, когда разделителем строки является подстрока.

Пример успешного вызова предиката:

```
["A", "123", "3.14"] = split_delimiter("A**123**3.14", "***")  
["ABCD", ""] = split_delimiter("ABCD&", "&")
```

```
[ "", "", "", "" ] = split_delimiter("****", "*")
[ "" ] = split_delimiter("", "")
```

Пример вызывает исключительную ситуацию:

```
L = split_delimiter("ABCD", "")
```

68. `StringList = split_length(String,Separators)` — функция расщепляет строку `String` на подстроки и возвращает список этих подстрок. Сепаратор `Separators` является списком длин подстрок. Длина каждой подстроки в списке `StringList` равна соответствующему значению в списке `Separators`. Если сумма значений в списке `Separators` превышает длину строки `String`, то возникает исключительная ситуация.

Пример успешного вызова предиката:

```
[ "hip", " ", "hop" ] = split_length("hip hop", [3,1,3])
[ "AB", "0", "", "3.14" ] = split_length("AB03.14", [2,1,0,4])
```

Пример генерирует исключительную ситуацию:

```
L = split_length("AB1233.1415", [4,3,4,8])
```

69. `splitStringBySeparators(String,Separators,Head,Separator,Rest)` — предикат расщепляет строку `String` на две части одним из символов строки `Separators`. Предикат возвращает часть строки до символа-разделителя `Head`, сам символ-разделитель `Separator` и часть строки после символа-разделителя `Rest`. Функция используется тогда, когда разделителями строки являются одиночные символы.

Пример успешного вызова предиката:

```
splitStringBySeparators("prolog","qwerty",Head,Separator,Rest)
  Head = "p", Separator = 'r', Rest = "olog"
splitStringBySeparators("prolog","Zpx",Head,Separator,Rest)
  Head = "", Separator = 'p', Rest = "rolog"
```

Пример неуспешного вызова предиката:

```
splitStringBySeparators("prolog","ZX",Head,Separator,Rest)
```

70. `Char = subChar(String,Position)` — функция возвращает символ `Char`, который занимает позицию `Position` в строке `String`. Первый символ строки `String` имеет нулевую позицию. Если позиция `Position` выходит за границы строки `String`, то возникает исключительная ситуация.

Пример успешного вызова функции:

```
'r' = subchar("Prolog",1)
```

Пример генерирует исключительную ситуацию:

```
Char = subchar("Prolog",10)
```

71. `SubString = subString(String,Position,Length)` — функция возвращает подстроку `SubString`, которая начинается с позиции `Position` строки `String` и имеет длину `Length` символов. Первый символ строки `String` имеет нулевую позицию.

Если позиция начала или конца подстроки выходит за границы строки *String*, то возникает исключительная ситуация.

Пример успешного вызова функции:

```
"rol" = subString("Prolog",1,3)
```

Пример генерирует исключительную ситуацию:

```
SubString = subString("Prolog",2,10)
```

72. *String* = toBinaryPrefixForm(*Precision*,*Value*) — функция возвращает строку *String*, представляющую вещественное число *Value* с точностью *Precision* знаков после запятой. Например, вызов *S* = toBinaryPrefixForm(4,123392) вернет строку *S* = "120.5000 K".

73. *CharList* = toCharList(*String*) — функция расщепляет строку *String* на список символов *CharList*.

Пример успешного вызова функции:

```
['T','e','c','t',' ',' ','1'] = toCharList("Text 1")
[] = toCharList("")
```

74. *NewString* = toLowerCase(*String*) — функция переводит буквы строки *String* в нижний регистр.

```
"this is a prolog." = toLowerCase("This is a PROLOG.")
```

75. *String* = toMetricPrefixForm(*Precision*,*Value*) — функция возвращает строку *String*, представляющую вещественное число *Value* с точностью *Precision* знаков после запятой. Например, вызов *S* = toMetricPrefixForm(8,10/7) вернет строку *S* = "1.42857143". С другой стороны, вызов *S* = toMetricPrefixForm(8,1/7) вернет строку *S* = "142.85714286 м".

76. *NewString* = toUpperCase(*String*) — функция переводит буквы строки *String* в верхний регистр.

```
"THIS IS A PROLOG." = toUpperCase("This is a PROLOG.")
```

77. *NewString* = trim(*String*) — функция удаляет пробелы, табуляции и символы перевода на новую строку, обрамляющие строку *String*.

Пример успешного вызова функции:

```
"Begin."= trim(" Begin. ")
"Test string."= trim(" \n \t Test string. \n \t ")
"" = trim(" ")
"" = trim("")
```

78. *NewString* = trimFront(*String*) — функция удаляет пробелы и табуляции и символы перевода на новую строку, находящиеся в начале строки *String*.

Пример успешного вызова функции:

```
"Begin. " = trimFront(" Begin. ")
"Test string. " = trimFront(" \t \n Test string. ")
"" = trimFront(" ")
"" = trimFront("")
```

79. `NewString = trimInner(String)` — функция заменяет группы пробелов и табуляций, находящиеся внутри строки `String`, ровно одним пробелом. Символы перевода на новую строку, находящиеся внутри строки `String`, не удаляются. Пробелы и табуляции, обрамляющие строку, остаются без изменений.

Пример успешного вызова функции:

```
" Test\n string. " = trimInner(" Test \t\t\t \n string. ")
"" = trimInner("")
" " = trimInner(" ")
" \t \t \t " = trimInner(" \t \t \t ")
```

80. `NewString = trimRear(String)` — функция удаляет пробелы, табуляции и символы перевода на новую строку, находящиеся в конце строки `String`.

Пример успешного вызова функции:

```
" Begin." = trimRear(" Begin. ")
" This is a string." = trimRear(" This is a string. \t \n ")
"" = trimRear(" ")
"" = trimRear("")
```

81. `String = truncate(Source, CharCount)` — функция возвращает префикс входной строки `Source` длиной `CharCount`. Если `CharCount` превосходит длину строки, то возвращается исходная строка.

82. `tryFront(String, CharCount, First, Last)` — предикат расщепляет строку `String` на две строки: `First` и `Last`. Длина строки `First` должна быть `CharCount` символов. Предикат неуспешен, когда параметр `CharCount` превышает длину строки `String`.

Пример успешного вызова предиката:

```
tryfront("Prolog", 3, First, Last)
    First = "Pro",    Last = "log"
tryfront("Prolog", 0, First, Last)
    First = "",    Last = "Prolog"
tryfront("", 0, First, Last)
    First = "",    Last = ""
```

Пример неуспешного вызова предиката:

```
tryfront("Prolog", 10, First, Last)
```

83. `Char = tryFrontChar(String)` — функция возвращает первый символ `Char` строки `String`.

Пример успешного вызова функции:

```
'P' = tryFrontChar("Prolog")
```

Пример неуспешного вызова функции:

```
Char = tryFrontChar("")
```

84. `Char = tryLastChar(String)` — функция возвращает последний символ `Char` строки `String`.

Пример успешного вызова функции:

```
'g' = tryLastChar("Prolog")
```

Пример неуспешного вызова функции:

```
Char = tryLastChar("")
```

85. `NewString = tryReplacePart(String, Position, CharCount, With)` — функция заменяет подстроку в строке `String`, начинающуюся с позиции `Position` длиной `CharCount` символов, строкой `With`. Длина строки `With` может не совпадать с длиной `CharCount`. Если заменяемая строка выходит за пределы исходной строки, то функция неуспешна.

Пример вызова функции:

```
"qAAAAAArty" = tryReplacePart("qwerty", 1, 2, "AAAAAA")
"qAty" = tryReplacePart("qwerty", 1, 3, "A")
```

Пример неуспешного вызова функции:

```
"qAty" = tryReplacePart("qwerty", 2, 10, "A")
```

86. `Char = trySubChar(String, Position)` — функция возвращает символ `Char`, который занимает позицию `Position` в строке `String`. Первый символ строки `String` имеет нулевую позицию. Если позиция `Position` выходит за границы строки `String`, то функция неуспешна.

Пример успешного вызова функции:

```
'r' = trySubchar("Prolog", 1)
```

Пример неуспешного вызова функции:

```
Char = trySubchar("Prolog", 10)
```

87. `SubString = trySubString(String, Position, CharCount)` — функция возвращает подстроку `SubString`, которая начинается с позиции `Position` строки `String` и имеет `CharCount` символов. Первый символ строки `String` имеет нулевую позицию. Если позиция начала или конца подстроки выходит за границы строки `String`, то функция неуспешна.

Пример успешного вызова функции:

```
"rol" = trySubString("Prolog", 1, 3)
```

Пример неуспешного вызова функции:

```
SubString = trySubString("Prolog", 2, 10)
```

88. `String = write(...)` — функция конкатенирует несколько строк, указанных в эллипсисе, в общую строку `String`.

Пример успешного вызова функции:

```
"123ab" = string::write("123", "ab")
```

ПРИЛОЖЕНИЕ 11

Описание электронного архива, сопровождающего книгу

Электронный архив с материалами к этой книге можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке <ftp://ftp.bhv.ru/9785977534871.zip> или со страницы книги на сайте www.bhv.ru.

Электронный архив включает:

- Папку Часть VI с тремя главами и одним приложением:
 - файл Glava40 описывает GUI-проект «Часы и секундомер»;
 - файл Glava41 описывает GUI-проект «Искусственная жизнь»;
 - файл Glava42 описывает GUI-проект «Машина Тьюринга»;
 - файл Pr12 содержит *приложение 12* с описанием графических элементов управления.
- Папку Projects, содержащую 34 консольных проекта нумерованных примеров из книги. В эту папку отобраны те примеры, которые имеют емкий исходный код или могут, по мнению автора, вызвать затруднения при создании. Каждый проект, кроме файлов с исходным кодом, содержит папку Exe с исполнимым файлом, библиотеками и необходимыми файлами исходных данных, если таковые требуются в проекте.
- Папку GUI-Projects, содержащую:
 - папку TC с проектом «Часы и секундомер», описанным в *главе 40*;
 - папку Life с проектом «Искусственная жизнь», описанным в *главе 41*;
 - папку Turing с проектом «Машина Тьюринга», описанным в *главе 42*.
- Папку Дистрибутив Visual Prolog 7.5 PE, содержащую файл дистрибутива *vip7500pe.msi*.

Предметный указатель

A

All 197
Allign 256
Ambiguous 103
Any 41
Anyflow 77
As decorated 259
Assert 88, 89
Asserta 88, 89
Assertz 88, 89
Attach 248

B

BackTracking 55, 65
Binary 42
BinaryNonAtomic 42
BinInclude 50
BitSize 47
Boolean 43
Bound 90
ByVal 258

C

Char 43
ClassName 91
Clauses 75
ClearInput 142
Closure 195
Collection
◊ modifiable 288
◊ persistent 288
Compare 92
CompareLexically 198
CompareResult 43

Compilation_date 51
Compilation_time 51
Compiler_buildDate 51
Compiler_version 51
Constants 50
Constructors 221
Consult 87
Convert 92
CreateSuspended 247
Currying 195
CutBackTrack 112

D

Decompose 198
Determ 76, 82
DifferenceBy 198
DifferenceEq 198
Digits 47
DigitsOf 97
Domains 41

E

EndOfStream 145
Erroneus 76, 84
ErrorExit 99
ESC-последовательность 25, 26
Exception 211
◊ descriptor 214
◊ handler 214
Exists 198

F

FactDB 43
Fail 56, 99

Failure 76
 File 143
 Filter 199
 FilteredMap 199
 Fold 199
 Foldl 199
 Foldr 200
 ForAll 200
 Free 91
 FromEllipsis 96

G

Garbage collector 225
 Generic programming 239
 GetBackTrack 112
 GroupByEq 200
 Guard 253

H

Handle 43
 HasDomain 98, 140, 142

I

If then else
 ◇ ветвление 206
 ◇ выражение 207
 In 99
 Inherits 229
 Inline 257
 InputStream_File 144
 Integer 43
 Integer64 43
 IntersectionBy 200
 IntersectionEq 200
 InvalidHandle 51
 IsErroneus 84, 90
 IsModeLine 140

L

Language 258
 LogicalNot 113
 LowerBound 97

M

Map 200
 MaxDigits 97

MaxFloatDigits 51
 MaximumBy 201
 MinimumBy 201
 Monitor 252
 Multi 76
 Must unify 36
 Mutex 252

N

New 221
 Nondeterm 76, 82
 Not 56, 99, 113
 Null 43, 51
 NullHandle 51

O

Open 103
 Orelse 99
 Out 77
 OutputStream_File 144

P

Pattern matching 31
 Platform_bits 51
 Platform_name 51
 Pointer 43
 Predicate_FullName 91
 Predicate_Name 91
 Predicates 75
 Procedure 76
 ProgramPoint 91
 Properties 103, 223

R

Read 140
 ReadChar 140
 ReadLine 140
 Real 44
 Real32 44
 ReConsult 88
 RemoveAllBy 201
 RemoveAllEq 201
 RemoveBy 201
 RemoveConsecutiveDuplicatesBy 202
 RemoveConsecutiveDuplicatesEq 202
 RemoveDuplicatesBy 202

RemoveDuplicatesEq 202
RemoveEq 201
Repeat 115, 145
Retract 88, 89
RetractAll 88, 90
RetractFactDb 90

ToBoolean 100
ToEllipsis 97
ToString 94
ToTerm 94
Try catch finally 214
TryConvert 95
TryToTerm 95

S

Save 87
SetModeLine 140
Single 82, 83, 88
SizeBitsOf 98
SizeOf 98
SizeOfDomain 98
Sort 202
SortBy 202
SourceFile_LineNo 91
SourceFile_Name 91
SourceFile_TimeStamp 91
Stack Overflow 357
Start 247
Stream 144
String 44
String8 44
Succeed 56, 100
Support 229
Symbol 44

U

UnCheckedConvert 96
Unfold 203
Unicode-символ 25
Unification 34
Union 257
UnionBy 203
UnionEq 203
Unsigned 45
Unsigned64 45
UnZip 204
UpperBound 97

W

Write 141
Writef 141

Z

Zip 204
Zip_nd 204
ZipHead_nd 204
ZipWith 204

T

ToAny 94
ToBinary 94

A

Атрибуты вывода 141

B

Ввод/вывод

- ◊ потоковый 144
 - ◊ списков поэлементно 169
 - ◊ списков целиком 167
 - ◊ файловый 143
- Выравнивание 256

Г

Голова

- ◊ правила 60
- ◊ списка 27, 155

Граф

- ◊ неориентированный 282
- ◊ ориентированный 279

Д

Двоичные данные 26

Дек 157, 158

Декларация

- ◊ класса 218

- ◊ модуля 102

Дерево

- ◊ n-арное 289

- ◊ красно-черное 294

- ◊ поиска решения 69

- ◊ произвольной арности 290

Дескриптор

- ◊ входного потока 144

- ◊ выходного потока 144

- ◊ исключения 214

Дизъюнкт Хорна 59

Должно унифицироваться 36

Домен 40

- ◊ базовый 41

- ◊ базы фактов 175

- ◊ пользовательский 46

- ◊ предикатный 185

- ◊ синоним 45

- ◊ составной 48

- ◊ списочный 48, 157

- ◊ функциональный 185

З

Замыкание 195

Защита 253

И

Идентификатор 23

Именованное значение 212

Имплементация

- ◊ класса 219
- ◊ модуля 102
- ◊ обобщенного класса 241

Импликация 59

Интерфейс

- ◊ мониторный 252
 - ◊ обобщенный 239
 - ◊ поддержка 229
- Интерфейс класса 218
- Исключительная ситуация 211

К

Каррирование 195

Класс 101

- ◊ мониторный 252
- ◊ обобщенный 240

Ключевые слова 22

Комментарии 21

Константы

- ◊ встроенные 51
- ◊ пользовательские 50

Конструктор

- ◊ класса 219
- ◊ обобщенный 240
- ◊ списка 31, 155

М

Массивы 299

- ◊ бинарные 299

- ◊ булевы 304

- ◊ двумерные 303

- ◊ одномерные 301

Мемоизация 138

Механизм логического вывода 65

Мир вывода

- ◊ закрытый 61

- ◊ открытый 61

Множество подстановок 31

Модуль 101

◊ export 267

Монитор 252

H

Наследование 229

O

Объявление

◊ предиката 75

◊ свойства 103

◊ факта 82

◊ функции 80

Операции 23

◊ арифметические 38

◊ булевые 38

◊ приоритет 38

◊ сравнения 38

Определение

◊ предиката 78

◊ факта 83

◊ функции 80

Откат 66

Отсечение 108

◊ динамическое 112

◊ зеленое 111

◊ красное 111

Очередь 157

P

Память

◊ atomic 299

◊ nonAtomic 299

Переменная 29

◊ анонимная 30

◊ изменяемая 122

◊ свободная 29

◊ связанная 29

Подстановка 31

Подтип 45

◊ вещественный 47

◊ целочисленный 46

Поиск с возвратом 65

Полиморфизм

◊ категориальный 45

◊ параметров доменов 179

◊ параметров предикатов 178

Полиморфный

◊ домен 179

◊ списочный домен 182

Поток

◊ ввода строк 149

◊ ввода/вывода стандартный 144

◊ возобновление 248

◊ вывода строк 151

◊ завершение 248

◊ параметра 77

◊ приостановка 248

◊ создание 247

Предикат 55

◊ анонимный 190

◊ внеродственный 74

◊ выполнимый 56

◊ декларативный смысл 56

◊ детерминированная дизъюнкция 99, 119

◊ дизъюнкция 57

◊ конъюнкция 57

◊ неуспешный 55

◊ отрицание 56

◊ процедурный смысл 56

◊ рекурсивный 127

◊ тождественно-истинный 56

◊ тождественно-ложный 56

◊ успешный 55

◊ чистый 74

Префикс @ 25

Присваивание

◊ неразрушающее 29, 37

◊ разрушающее 29, 84, 122, 205

Проверка вхождения 34

P

Разделяемый ресурс 251

Реализация модуля 102

Режим детерминизма

◊ предикатов и функций 75

◊ фактов 82

Рекурсивное правило 127

- Рекурсия 127
 - ◊ неоптимизированная 128
 - ◊ невхостовая 129
 - ◊ оптимизированная 128
 - ◊ с заданным числом повторений 130
 - ◊ с управляемым режимом детерминизмом 358
 - ◊ с условием завершения 127
 - ◊ с условием продолжения 127
 - ◊ хвостовая 128

C

Сборщик мусора 225, 363

Свойства

- ◊ модуля 103

- ◊ объекта 223

Связывание

- ◊ позднее 265

- ◊ раннее 265

Сигнатура предикатов 185

Символ 25

Синтаксический сахар 57

Сопоставление

- ◊ с образцом 31

- ◊ списков 33

- ◊ термов 32

Состояние

- ◊ класса 221, 227

- ◊ объекта 221

Список 27, 30, 153

- ◊ вложенный 31

- ◊ коллектор 172

- ◊ конструктор 31, 155

- ◊ многоуровневый 48

- ◊ пустой 31, 155

- ◊ селектор 31, 156

- ◊ фактов 174

Стек 157

- ◊ адресов возврата 66

- ◊ изменение размера 361

- ◊ переполнение 357

- ◊ размер 275, 357

Стековый фрейм 66, 357

Строки 25

Структура

- ◊ декларации класса 220

- ◊ имплементации класса 220
- ◊ интерфейса класса 219
- Счетчик 84, 117

T

Табулирование 138

Тело правила 60, 61

Терм

- ◊ арность 28

- ◊ простой 28

- ◊ составной 28

- ◊ рекурсивный 28

- ◊ нульярный 29

- ◊ вложенный 29

- ◊ основной 30

- ◊ неосновной 30

- ◊ префиксный 30

- ◊ инфиксный 30

У

Удаление

- ◊ всех фактов 88

- ◊ всех фактов раздела БД 89

- ◊ факта 88

Унификация 34

- ◊ списков 36

- ◊ термов 35

Управляемый детерминизм 357

Ф

Факт 64, 82

- ◊ добавление 88

- ◊ загрузка 87

- ◊ когерентность 174

- ◊ перезагрузка 88

- ◊ переменная 83, 122

- ◊ сохранение 87

- ◊ удаление 88

Функтор 28

Функция 80

- ◊ анонимная 190

Х

Хвост списка 27, 155

Ц

Цель 65

Цикл с откатом

◊ на основе отрицания 125

◊ с заданным числом повторений 117

◊ с условием завершения 116

◊ с условием продолжения 116

Циклы

◊ foreach 207

◊ с заданным числом повторений 210

◊ восьмеричные 24

◊ десятичные 24

◊ целые 24

◊ шестнадцатеричные 24

Ш

Шаблон потоков параметров

◊ предикатов 77

◊ свойства 103

◊ фактов 82

Э

Эллипсис 75, 183

Ч

Числа

◊ вещественные 24