Министерство Образования
Российской Федерации

Кемеровский государственный университет

В. В. Терёхин

TURBO PROLOG

Учебное пособие

Новокузнецк 2005

Содержание

Глава 1. Начала работы на Турбо-Прологе

1.1 Введение

Турбо-Пролог (**Turbo Prolog**) яляется языком искусственного интеллекта (ИИ). Разработан этот язык компанией **Borland International** и характеризуется высокой скоростью работы и низкой стоимостью, что делает Турбо-Пролог весьма популярным в широких кругах обладателей персональных компьютеров.

Для чего нужен Турбо-Пролог ?

Турбо-Пролог является компиляторно-ориентированным языком программирования высокого уровня. Он предназначен для программирования задач из области искусственного интеллекта. Как язык программирования ИИ Турбо-Пролог особенно хорош для создания экспертных систем, динамических баз данных, программ с применением естественно-языковых конструкций. Он также может быть использован и для других задач общего характера. Турбо-Пролог имеет окна, цветную графику и интерактивные средства ввода-вывода, что свидетельствует о его максимальном удобстве для пользователя прикладных программ.

Встроенные предикаты Турбо-Пролога и другие дополнительные возможности делают его пригодным для решения многих стандартных задач из сферы бизнеса, включая бухгалтерский учет, расчет зарплаты и представление графиков.

Популярность Пролога резко возросла после того, как он был выбран в качестве базового языка японской программы ЭВМ пятого поколения. Спрос на программы искусственного интеллекта, применяемые либо взамен, либо совместно с традиционными прикладными программами, постоянно увеличивается. В ходе продолжающейся эволюции применений вычислительной техники наблюдается все возрастающая потребность в создании прикладных программ на Прологе. Турбо-Пролог может прекрасно удовлетворить такую потребность.

Наряду с Турбо-Прологом в США созданы еще несколько реализаций Пролога: **Arity Prolog, Prolog II, Wisdom Prolog** и **Micro Prolog**. В отличие от них Турбо-Пролог имеет великолепный полноэкранный редактор, множество рабочих окон и интерактивный диалоговый отладчик. Он поддерживает цветную графику IBM PC, снабженного цветным графическим адаптером (CGA) и расширенным графическим адаптером (EGA). Предикаты графики и система с графическим экранным пером являются составной частью Турбо-Пролога. Он также снабжен средствами работы с последовательными файлами, файлами прямого доступа и двоичными файлами.

Еще одной причиной предпочтительности выбора Турбо-Пролога является то, что написанные на нем программы компилируются, в отличие от других версий Пролога, где программа интерпретируется. Интерпретатор выбирает предложения программы одно за другим и последовательно ис-

полняет их. Компилятор транслирует в машинные коды сразу весь текст программы. Турбо-Пролог транслирует исходный текст программы особенно быстро (быстрее всех других версий Пролога для компьютеров IBM PC). Также он позволяет легко создавать отдельные выполнимые файлы. Далее, Турбо-Пролог имеет прекрасный пользовательский интерфейс для отладки программ. Другими словами, Турбо-Пролог - это наилучший товар на рынке программного обеспечения на сегодняшний день.

Что такое Пролог ?

Название Пролог произошло от словосочетания "программирование при помощи логики" (Prolog PROgramming in LOGic). Пролог был разработан и впервые реализован в 1973 г. Алэном Колмероэ и другими членами "группы искусственного интеллекта" (Франция) Марсельского университета. Главной задачей группы было создание программы для перевода с естественного языка. С тех пор было сделано несколько расширений и усовершенствований языка, здесь можно отметить работу группы из Эдинбургского университета (Шотландия). Шотландский вариант получил название C&M Prolog в честь авторов классической работы "Программирование на Прологе" Уильяма Клоксина и Кристоффера Меллиша. Хотя официального стандарта Пролога не существует, в этой книге приведен тот его вариант, который считается неофициальным стандартом.

Турбо-Пролог имеет несколько отличий. В нем отсутствуют некоторые элементы C&M, но такие, которые встречаются только в очень сложных программах. Самым существенным отличием от упомянутого стандарта (как и от других версий языка) является наличие в Турбо-Прологе строгой типизации элементов данных. Сделанные отступления от неофициального стандарта позволили значительно увеличить скорость трансляции и счета программ. И хотя некоторые программисты возражают против этих отступлений, говоря что Турбо-Пролог не есть Пролог "в чистом виде", однако именно в таком виде язык имеет массу преимуществ, например наличие графики и возможность работы с окнами. Этих средств нет в других реализациях. Независимо от того, является ли Турбо-Пролог "чистым" Прологом или нет, он тем не менее является очень современным, полноценным и гибким языком программирования.

Турбо-Пролог и другие языки программирования

Такие языки программирования, как Паскаль, Бэйсик и Си относятся к разряду императивных или процедурных. Программа, написанная на императивном языке, состоит из последовательности команд, определяющих шаги, необходимые для достижения назначения программы. Пролог является декларативным языком. Программа на декларативном языке являет собой набор логический описаний, определяющих цель, ради которой она написана. Таким образом в Прологе отсутствуют такие явные управляющие структуры, как **DO WHILE и IF ... THEN**. Вместо расписывания шагов выполнения программы программист должен определить логический метод для достижения

цели программы. Мощные внутренние унификационные процедуры (работа которых будет пояснена в гл. 2) будут затем искать решение поставленной задачи.

Сила Турбо-Пролога заключается в его возможностях поиска и сопоставления. Внутренние унификационные процедуры бесстрастно перебирают все возможные комбинации правил, пытаясь удовлетворить заданную программистом цель. Пролог, как видим, базируется на естественных для человека логических принципах, и поэтому, чем больше вы им занимаетесь, тем он становится все более привлекательным.

Кому предназначена эта документация ?

Это пособие написано для всех, кто интересуется Турбо-Прологом, вне зависимости от того, является ли он новичком или профессиональным программистом. От вас не требуется знания ни процедурных языков (например, Бэйсик, Паскаль или Си), ни декларативных языков (например, Лисп). В книге изложены основные концепции и методы, необходимые для написания программ на Турбо-Прологе. С ее помощью вы научитесь мыслить категориями Пролога.

Если вы не работали с Турбо-Прологом вообще, эта книга именно для вас. "Использование Турбо-Пролога" позволит понять, что представляет собой программирование на декларативном языке, как проектируются и пишутся на нем программы. Логические предикаты и правила - это основные средства программирования для всех версий Пролога.

Если система Турбо-Пролог у вас уже имеется, и вы пробовали с ней работать, то книга пригодится и в этом случае. Она покажет, как проектировать и реализовать прикладные программы, предназначенные для решения различных задач.

Турбо-Пролог - это осуществленная реализация языка программирования высокого уровня Пролог компиляторного типа. Ее отличает большая скорость компиляции и счета. Турбо-Пролог предназначен для выдачи ответов, которые он логически выводит при посредстве своих мощных внутренних процедур. Так, программа на Турбо-Прологе в несколько строк, может заменить несколько страниц текста при программировании на каком-либо другом языке. Благодаря наличию мощных средств сопоставления, Турбо-Пролог пригоден не только для использования в приложениях, относящихся к области искусственного интеллекта и обработке естественно-языковых конструкций, но также применим в таких традиционных областях, как, например, управление базами данных.

Турбо-Пролог предназначен для машин класса IBM PC и совместимых с ними.

1.2 Пакет системы Турбо-Пролог

Пакет компилятора Турбо-Пролог состоит из двух дистрибутивных дисков и Руководства пользователя, насчитывающего свыше 200 страниц.

Один диск имеет имя **PROGRAM DISK**, другой - **LIBRARY&SAMPLE PROGRAMS.** В табл.1 приведены описания файлов, содержащихся на обоих дистрибутивных дисках.

Таблица 1.1. Файлы дистрибутивных дисков версии 1.1

1.Содержимое диска **PROGRAM DISK**:

PROLOG.EXE	основной файл системы Турбо-Пролог.
PROLOG.OVL	оверлейный файл, используемый системой при запуске, при создании .**EXE**-файлов и при выполнении некоторых других функций.
PROLOG.SYS	файл, содержащий информацию о цвете, расположении окон системы, также как и информацию об используемых системой директориях.
PROLOG.ERR	файл с сообщениями об ошибках
PROLOG.HLP	файл с текстом применяемых в системе подсказок (обращение к нему осуществляется при помощи функциональной клавиши **F1)**.
GEOBASE.PRO и **GEOBASE.INC**	демонстрационная программа базы данных по географии США **GeoBASE.**
GEOBASE.DBA	данные для программы **GeoBase**.
GEOBASE.HLP	текст подсказок программы **GeoBase.**
README.COM	программа для выдачи на экран содержимого файла **README**.
README	текстовой файл, содержащий необходимую для работы информацию, не включенную в руководство пользователя Турбо-Пролога.

2.Содержимое диска **LIBRARY&SAMPLE PROGRAMS**:

PROLOG.LIB и **INIT.OBJ**	файлы, используемые системой Турбо-Пролог при создании выполнимых файлов.
EXAMPLES	директория, содержащая программы, используемые в разделе обучения Руководства пользователя.
ANSWERS	директория, содержащая ответы на упражнения из Руководства.
PROGRAMS	директория, содержащая демонстрационные программы.

1.2.1 Главное меню системы Турбо-Пролог

Главное меню Турбо-Пролога высвечивает 7 доступных пользователю опций (команд) в верхней части экрана. Первая буква названия каждой из команд выделена при помощи увеличенной яркости; на репродукциях книги это, к сожалению, не видно. Выделение имеет целью напоминать, что для задания команды достаточно нажать лишь первую букву ее названия.

Команды определяются 7 функциями Турбо-Пролога, каковыми являются:

1. Запуск программы на счет (**Run**).
2. Трансляция программы (**Compile**).
3. Редактирование текста программы (**Edit**).
4. Заданий опций компилятора (**Options**).
5. Работа с файлами (**Files**).
6. Настройка системы в соответствии с индивидуальными потребностями (**Setup**).
7. Выход из системы (**Quit**).

Переход от одной команды к другой прост и удобен. Существует два способа задания команд. Первый требует нажатия клавиши, соответствующей первой букве названия выбранной команды. Так, для выбора команды ***Edit*** необходимо нажать **E**. (Нет никакой разницы, какая буква была введена большая или маленькая, т. е. использование ***Shift*** не обязательно.) Для окончания работы с командой используется клавиша **Esc.** Второй способ состоит в перемещении по меню при помощи стрелок; переход к работе с выбранной командой осуществляется нажатием ***Enter***.

Главное меню содержит четыре окна. В левом верхнем углу располагается окно редактора Турбо-Пролога (***Editor***), в правом верхнем углу - окно диалога **(*Dialog*)**, в левом нижнем окно сообщений (***Message***), в правом нижнем - окно трассировки (***Trace***). Если вы используете цветной монитор, то по умолчанию для окна редактора задается голубой цвет, для окна диалога - красный и черный - для окон сообщений и трассировки.

Верхняя строка окна редактора содержит информацию о высвечиваемом в этом окне файле. ***Line*** 1 и ***Col*** 1 свидетельствуют о том, что курсор в настоящий момент располагается в первой позиции первой строки. Значения этих индикаторов строки и позиции меняются вслед за изменением положения курсора. Надпись ***Indent*** сигнализирует о том, что включен режим автоматического выравнивания строк, а надпись ***Insert*** - о том, что задан режим вставки. **WORK.PRO** является заданным по умолчанию именем рабочего файла; **.PRO** есть заданное по умолчанию расширение для файлов, содержащих программы на Турбо-Прологе. Если вы набьете в редакторе какой-либо текст и запишите его на диск без изменения имени файла, то файл с вашим текстом получит имя **WORK.PRO**.

1.2.2 Запуск на счет программы, написанной на Турбо-Прологе

Программа, работу с которой мы сейчас начинаем, имеет целью дать необходимые навыки в использовании меню системы и основных команд редактора. Программа называется **WELCOME.PRO**. Действия, которые следует предпринять, чтобы создать эту программу, не зависят от того, работаете ли вы на машине с винчестерским диском, или на машине с двумя дисководами для гибких дисков.

Перейдите при помощи стрелки к команде главного меню ***Edit*** и нажмите клавишу ***Enter*** (либо просто введите латинскую букву ***E***).При этом в левом верхнем углу окна ***Editor*** появится мерцающая черточка - курсор редактора. Теперь редактор Турбо-Пролога готов принять вводимый вами с клавиатуры текст.

Набейте текст программы **WELCOME.PRO**:

```
        predicates
        hello
goal
        hello.
clauses
        hello :-
write("Welcome to Turbo Prolog!"), nl.
```

Когда вы доходите до конца очередной строки, нажимайте клавишу ***Enter*** для перехода на следующую. Для удаления неверного символа нужно прибегнуть к помощи клавиши ***BackSpace.*** Выравнивание строк в Турбо-Прологе, так же как и в других языках, преследует цель придания программе несколько большей наглядности и не влияет на ее выполнение.

1.2.3 "Добро пожаловать в систему Турбо-Пролог!"

Вы ввели в компьютер вашу первую программу на Турбо-Прологе. Для того чтобы запустить ее на счет, сначала требуется покинуть редактор системы, а для этого нужно нажать клавишу ***Esc***. Курсор редактора при этом исчезнет, а курсор главного меню станет указывать на команду ***Edit***. Задайте теперь команду ***Run*** и пронаблюдайте за двумя появившимися во время трансляции программы строками в окне сообщений ***Message*** и за результатом работы программы в окне диалога ***Dialog***.

Первая строка в окне сообщений указывает на то, что началась трансляция программы **WELCOME.PRO**. Трансляция задается автоматически при задании команды ***Run***, т. е. нет необходимости прибегать к помощи специальной команды ***Compile***.

Турбо-Пролог позволяет адресовать результат трансляции либо на диск, либо в оперативную память. При задании ***Run*** программа транслируется в оперативную память. Транслировать программу на диск пока вы лишь учитесь работать на Турбо-Прологе, нет необходимости. Но в дальнейшем вы можете создавать и объектные файлы для совместного редактирования с другими объектными модулями, и выполнимые файлы, которые

можно запускать на счет вне среды Турбо-Пролога. Как вы, вероятно, заметили, Турбо-Пролог транслирует столь малую программу очень и очень быстро, за какие-то доли секунды. Вторая строка в окне сообщений сигнализирует о трансляции предиката ***hello***.

* Упражнение
1.1. Запустите на счет программу **WELCOME** пару раз подряд и попрактикуйтесь переходить от одной команды к другой, пока не почувствуете себя в этом деле достаточно уверенно.

Теперь можно окончить сеанс работы с Турбо-Прологом, если вы этого хотите, конечно. Если в рабочий файл были внесены хоть какие-то изменения, система спросит, нужно ли записывать на диск новый исправленный вариант файла. Для обозначения положительного ответа необходимо нажать клавишу ***Y.*** Если команда ***Quit*** была задана случайно, ее можно отменить при помощи ***Esc.***

1.2.4 Сохранение программного файла

Для того, чтобы записать на диск программу и таким образом сохранить ее, необходимо выйти из редактора (если вы находитесь в режиме редактирования), нажав клавишу ***Esc***, а затем выбрать команду ***Files*** и подкоманду ***Save*** во вновь появившемся меню (либо нажав S, либо используя стрелки и клавишу ***Enter***). В результате этих действий на экране возникнет небольшое окно, в котором будет высвечено либо заданное по умолчанию имя файла (как, например, **WORK.PRO**), либо то имя, которое вы присвоили файлу сами. Имя файла можно оставить без изменений, а можно и отредактировать. В нашем случае следует ввести имя **WELCOME.PRO**, а затем нажать ***Enter***.

Если на диске уже есть файл с указанным именем (более ранняя версия редактируемой программы или какая-либо иная программа), то в результате операции записи на диск расширение имени этого файла будет сменено на .**BAK**, чтобы пометить старый вариант файла. Не забывайте сохранять отредактированный файл перед тем, как окончить сеанс работы с Турбо-Прологом. В противном случае модифицированный вариант программы будет утерян.

Достаточно частое сохранение рабочего файла на диск очень полезно. В случае отказа электропитания, или программного сбоя компьютера вы всегда будете иметь достаточно "свежий" вариант программы. Некоторые программисты записывают очередные версии программы под разными именами. Генерация таких многочисленных "поколений" программных файлов может также быть полезной, так как позволяет проследить все этапы развития программы.

1.2.5 Просмотр каталога директории

Для того чтобы просмотреть каталог файлов какой-либо директории, необходимо выбрать в главном меню команду ***Files*** и подкоманду ***Directory*** в появившемся меню команды. На экране возникнет окно, в котором перечисляются все файлы текущей директории .**PRO**. Если вы хотите увидеть каталог другой директории, то следует ввести путь доступа к этой директории, а затем нажать клавишу ***Enter***. В ответ система попросит задать маску интересующих вас файлов (***File mask***). По умолчанию стоит маска ****.PRO.*** Можно оставить эту маску, а можно ввести и свою. После того, как вы нажмете ***Enter***, в окне появятся имена всех файлов заданной директории, удовлетворяющие заданной маске.

1.2.6 Загрузка и редактирование программного файла

Турбо-Пролог обладает очень мощным экранным редактором, оснащенным большим количеством средств, облегчающих работу программиста. В этом состоит важное отличие Турбо-Пролога от других реализаций Пролога: некоторые из них вообще не обладают встроенными редакторами и приходится выходить из них каждый раз, как только возникает необходимость внести в программу хоть малейшее изменение. Большинство команд редактора Турбо-Пролога совпадают с командами редактора **WordStar**. Если вы знакомы с **WordStar,** или с другими похожими редакторами, такими, например, как редактор ТУРБО-ПАСКАЛя, то будет несложно обучиться и командам Турбо-Пролога.

Для того чтобы загрузить в окно редактора уже существующий файл, требуется выбрать команду ***Files*** главного меню и подкоманду ***Load*** в меню ***Files***. Если в ответ на запрос имени файла просто нажать клавишу ***Enter***, то на экране в специальном окне будет высвечен перечень файлов директории .**PRO.** Теперь, используя четыре стрелки можно добраться до имени интересующего вас файла, и после этого нажать клавишу ***Enter***. Если же вы решили набрать имя файла с клавиатуры, нет необходимости указывать его расширение, так как по умолчанию считается, что файл имеет расширение ***.PRO***. Работая с редактором Турбо-Пролога, можно в любой момент получить информацию о любой из его команд; для этого нужно нажать функциональную клавишу ***F1***. На экране появляется небольшое меню подсказки ***Help.*** Если вы выберете первую опцию из предлагаемого списка, то на экране возникнет окно ***Help***. Окно демонстрирует краткий перечень команд редактора и другую полезную информацию о редакторе. Нажав комбинацию ***Shift-F10***, можно расширить это окно до размеров полного экрана; повторное нажатие ***Shift-F10*** вернет окно к его первоначальным габаритам (эта операцию можно проделать с любым из окон системы). Другие опции подсказки позволяют получить несколько более специфическую информацию об определенных группах команд.

Если вам требуется создать новый программный файл, а в окне редактора уже находится какой-либо другой, то возникает необходимость очи-

стить окно от этого файла. Чтобы проделать это, необходимо задать команду главного меню ***Files***, а в нем подкоманду ***Zap file in editor***. Система запросит подтверждения. Если нажать ***Y***, то окно очистится от текста.

Вслед за этим возникнет маленькое окошко, предназначенное для ввода имени нового файла. Введите новое имя, если таковое требуется, и нажмите ***Enter***. Если никакое имя введено не будет, то новый файл будет именоваться старым именем. После нажатия ***Enter*** окошко исчезнет, а курсор главного меню будет указывать на ***Files***. Теперь его можно сместить к ***Edit*** и нажать ***Enter***, после чего приступить к набивке новой программы.

* Упражнение

1.2. Войдите в редактор Турбо-Пролога. Загрузите программу **WELCOME** и добавьте в нее строку

nl,***write***("**Have a nice** day.")

вслед за строкой

write**("Welcome to Turbo Prolog!")**

Запустите на счет измененный вариант программы.

1.2.7 Печать файла

Подкоманда ***Print*** команды ***Files*** может быть использована для печати файла, находящегося в окне редактора. Как обычно, сначала нужно задать в главном меню команду ***Files***, а во вновь появившемся меню - подкоманду Print. Не следует использовать подкоманду ***Print***, если к компьютеру не подключен принтер.

1.4 Заключение

В настоящей главе, весьма небольшой по размеру, были описаны основные характеристики Турбо-Пролога, а также приведены инструкции по установке системы. На примере ввода и запуска на счет короткой и очень простой по структуре программы вы познакомились с основами работы в Турбо-Прологе. В главе было также приведено достаточно подробное описание меню системы, которое постоянно используется на протяжении всей работы над программой.

Информация, представленная в настоящей главе, так же как, впрочем, и в некоторых других частях книги, рассчитана не столько на чтение и запоминание, сколько на использование ее в качестве руководства в процессе работы с системой.

Глава 2. Основные понятия языка Турбо-Пролог

2.1 Ведение

В этой главе язык Турбо-Пролог рассматривается на концептуальном уровне. И, хотя здесь вводятся основные понятия языка, данная глава не предназначена для изучения синтаксиса Турбо-Пролога и правил по-

строения программ. Основная цель состоит в том, чтобы показать, почему Турбо-Пролог называется декларативным языком, и в чем отличие декларативных языков от императивных.

При освоении нового языка важно изучить способ представления данных и процесс выполнения программы. В этой главе показано, как внутренние подпрограммы унификации системы Турбо-Пролог управляют выполнением программы и обработкой данных. "Псевдопрограммы" и примеры фрагментов программ помогут понять основы использования Турбо-Пролога для решения задач.

2.2 Декларативные и императивные языки

По-видимому, наилучший способ понять идеи декларативных языков состоит в рассмотрении их в контексте эволюции языков программирования. Первоначально способ решения задачи на ЭВМ заключался в составлении программ на "естественном языке" ЭВМ, т.е. в машинных кодах. Имея полностью в своем распоряжении память и центральный процессор ЭВМ, программисты, использовавшие машинные языки, должны были задавать машине способ решения задачи в виде последовательности шагов, которые должна выполнить машина, и способы обработки данных. Программы на машинных языках были императивными, то есть они диктовали шаги решения задачи (алгоритм) и способы обработки данных.

Программирование на машинных языках требует много времени, и кроме того, эти языки сложны для чтения и составления программ программистом. Поэтому были разработаны специальные программы для трансляции в машинный код более естественных и легко понимаемых языковых конструкций, описывающих алгоритмы. Транслирующие программы преобразуют команды, заданные в символической форме, в последовательность команд машинного языка. Эти программы называются компиляторами и интерпретаторами. Транслируемые команды, заданные в символической форме, представляют собой языки высокого уровня, такие, например, как Фортран и Кобол. Создание языков высокого уровня существенно увеличило скорость разработки программ и сократило потребность в программистах, хорошо знакомых с архитектурой ЭВМ. Но программы, написанные на языках высокого уровня, все еще оставались императивными по своей природе, так как все еще в них было необходимо указывать шаги обработки и манипулирования данными.

Большинство используемых сейчас языков являются императивными. Но в 60-х гг. были разработаны неимперативные языки, например, Лисп и Исвим, основанные на декларативных принципах. Неимперативные языки, создали новые методы программирования, которые позволили сгладить различия между шагами решения задачи и манипулированием данными.

Пролог (ПРОграммирование на ЛОГическом языке) был разработан в течении этого периода. Программы на Прологе содержат утверждения, сделанные программистом, т.е. "декларации" логических соотношений, необходимых для решения задачи. Языки, позволяющие описывать логиче-

ские соотношения, при помощи которых программа может решить задачу, называются декларативными языками. Программы на таких языках объявляют, какие результаты дадут те или иные действия. Программист предоставляет самой системе разобраться в совокупности способов решения задачи.

Для тех, кто привык пользоваться императивным языком, декларативные языки кажутся почти лишенными какой-либо строгости. Несмотря на это, часто программа решения конкретной задачи, составленная на декларативном языке, требует значительно меньше операторов, чем при использовании императивных языков. Основная часть программы на императивном языке предназначена для управления шагами программы, обрабатывающими данные. На Прологе, наоборот, большая часть управления программой неявно содержится в конструкциях языка или оказывается объявленной в декларативной форме.

Программисты, имеющие опыт работы на императивных языках, иногда теряются, сталкиваясь с полным отсутствием средств управления программой. Люди, никогда раньше не занимавшиеся программированием, могут освоить декларативные языки быстрее, чем программисты, знакомые с императивными языками. Если вы никогда не программировали, то не думайте, что изучение Турбо-Пролога будет сложным из-за неумения программировать на языках типа Си, Бейсик или Паскаль. Если вы программировали на некоторых императивных языках, то не думайте, что вы сразу же сможете освоить Турбо-Пролог, используя ранее полученные навыки. Необходимо научиться формулировать логические соотношения, описывающие задачу, а программа на Турбо-Прологе выполнит все необходимые действия.

2.2.1 Логика предикатов

Как вы теперь знаете, Турбо-Пролог - это декларативный язык, программы на котором содержат объявления логических взаимосвязей, необходимых для решения задачи. Обозначения, используемые в Турбо-Прологе для выражения этих логических взаимосвязей, унаследованы из логики предикатов.

В логике предикатов рассматриваются отношения между утверждениями и объектами. Не пугайтесь термина "логика предикатов". По всей вероятности основные понятия (а может быть и терминология) логики предикатов вам уже знакомы.

Например, рассмотрим предложение

Мэри любит яблоки

Прочитав это предложение, вы узнаете, что имеется факт, утверждающий, что Мэри любит яблоки. Добавим еще одно предложение к тому, что вы уже знаете:

Бет любит то же самое, что и Мэри

Используя эти два предложения, вы можете придти к заключению, что Бет тоже любит яблоки. Получая это заключение, вы пользовались упро-

щенной формой логического вывода, используемого в логике предикатов для получения нового факта из двух имеющихся утверждений.

Теперь опять посмотрим на эти предпосылки, сделав некоторые незначительные изменения в одном из предложений:

Мэри любит яблоки

Бет любит нечто, если Мэри любит (это же) нечто

Порядок слов, т.е. синтаксис второго предложения был изменен, но его значение осталось тем же самым. Другими словами, второе предложение семантически эквивалентно:

Бет любит то же самое, что и Мэри

Если известны факты о том, что любит Мэри, то опять можно заключить, что Бет любит яблоки.

Вспомним, что логика предикатов рассматривает отношения между утверждениями и объектами. В предпосылках "Мэри любит яблоки" и "Бет любит нечто, если Мэри любит (это же)" существует отношение между субъектом и объектом. То же самое справедливо и для полученного путем логического вывода факта. "Субъект" - это Мэри или Бет, а "объект" - это яблоки. Отношение между Мэри или Бет и яблоками называется отношением связывания. Объектами в этом отношении являются Мэри, Бет и яблоки.

Если убрать лишние слова, то отношение, соответствующее первому предложению будет иметь вид:

Объект	**Отношение**	**Объект**
Мэри	любит	яблоки

Заметьте, что слова, обозначающие отношение и объекты имеют порядок, естественный для фраз русского языка. Но можно поместить имя отношения перед объектами:

Отношение	**Объект**	**Объект**
любит	Мэри	яблоки

Несмотря на изменение формы, его смысл остался тем же самым, так как был изменен только синтаксис.

Теперь опять рассмотрим предложение:

Бет любит нечто, если Мэри любит (это же) нечто ,

и заменим слово "нечто" на местоимение "это":

Бет любит это, если Мэри любит это

Заметьте, что это предложение выражает два отношения "любит". Они соединены условием, выраженным словом "если".

Условие "если" требует проверки предпосылки для вывода нового факта. И, таким образом, имеются следующие отношения:

Отношение	**Объект**	**Объект**	**Условие**
любит	Мэри	яблоки	
любит	Бет	это	если
любит	Мэри	это	

Как вы, вероятно знаете, "это" является местоимением, а местоимения используются вместо имени существительного. Местоимение "это" мо-

жет обозначать любое имя существительное, и то, что обозначает местоимение "это" может меняться от предложения к предложению.

Если группа предложений не содержит информации о том, что конкретно и в каком предложении обозначает местоимение "это", то вы не знаете точного смысла предложения. Вы не можете вывести каких-либо новых фактов из предложения

Бет любит это, если Мэри любит это

потому, что вы не знаете, что обозначает местоимение "это".

Но, если вы опять посмотрите на предпосылки, то обнаружите, что

Мэри любит яблоки

Теперь вы имеете вполне определенное значение для местоимения "это" - яблоки. Сейчас вы можете принять на веру, что местоимение "это" означает яблоки, хотя "это" не всегда обозначает яблоки, так как Мэри может также любить апельсины, груши или жареные кукурузные зерна.

Имея в виду значение "яблоки" для местоимения "это", предложения могут быть переформулированы следующим образом:

любит Мэри яблоки

любит Бет это (яблоки), если Мэри любит это (яблоки)

Теперь может быть получен новый факт: Бет любит яблоки. Выше этот факт был получен с использованием только семантики русского языка. В следующем разделе рассматривается, как этот же способ логического вывода используется в логике предикатов.

2.2.2 Пролог и логика предикатов

Вы понимаете смысл предложения "Мэри любит яблоки", так как слова в этом предложении появляются в привычном для вас порядке. Порядок слов, или синтаксис, помогает передать смысл предложения. Те же самые слова, но в другом порядке, имели бы другой смысл. Но если мы согласимся, что предложения "Мэри любит яблоки" и "Любит Мэри яблоки" имеют один и тот же смысл, то мы можем использовать тот и другой порядок слов, и при этом понимать друг друга. К сожалению, до сих пор никто еще не разработал удобный язык программирования, использующий правильный синтаксис английского или русского языка. Но как только вы привыкните к синтаксису Турбо-Пролога, то обнаружите, что он весьма близок к естественному.

Вам уже встречалась следующая конструкция:

Отношение	**Объект**	**Объект**
Любит	Мэри	яблоки

Отношение "любит" связывает объекты "Мэри" и "яблоки" в конструкцию, обладающую определенным смыслом. Отношение "любит" может быть также названо предикатом:

Предикат	**Объект**	**Объект**
любит	Мэри	яблоки

То же самое может быть записано в виде

предикат(объект1,объект2)

или

любит (Мэри, яблоки)

Эта синтаксическая форма очень близка к синтаксису Турбо-Пролога. Но слова, начинающиеся с прописных букв, являются переменными языка Пролог. Объекты с известными или постоянными значениями должны начинаться со строчных букв, если они не являются числами. (Строго говоря, существуют исключения из этого правила; в последующих главах это будет объяснено). Тогда, используя синтаксис Турбо-Пролога, этот факт будет иметь вид:

любит(мэри, яблоки).

Заметьте, что это выражение заканчивается точкой. Точка означает: "Здесь конец части информации".

Вспомним, что второе предложение на русском языке "Бет любит это, если Мэри любит это" является условной предпосылкой со связкой "если", указывающей на условное отношение. Также вспомните, что местоимение "это" может иметь переменное значение, изменяющиеся от предложения к предложению. Вы не должны удивляться, если узнаете, что эквивалентом переменной Турбо-Пролога является слово "это" русского языка. Переменные в Турбо-Прологе имеют некоторые общие свойства с местоимениями. Если вы не знаете, что обозначает слово "это" или какое значение имеет переменная Турбо-Пролога, вы не сможете вывести новых фактов. Если вы все же знаете значение слова "это", то тогда вы сможете вывести новые факты, связанные условным отношением с другими уже известными фактами.

Второе предложение с использованием синтаксиса Турбо-Пролога имеет вид:

отношение(объект1,объект2) условие отношение(объект1,объект2)

любит (бет, ***X***) если любит (мэри, ***X***)

Это предложение является правилом по терминологии Турбо-Пролога. Само название подразумевает, что правило используется Прологом для проверки условия вывода нового факта. Все известные отношения, касающиеся Мэри, Бет и яблок с использованием синтаксиса Турбо-Пролога имеют вид:

любит(мэри, яблоки).

любит(бет, *X*) если любит (мэри, *X*).

Эти знания включают один факт "Мэри любит яблоки" и одно правило "Бет любит это, если Мэри любит это". Вы знаете, по крайней мере, одно значение для переменной ***X*** (или слова "это"), так как известен факт, что Мэри любит яблоки. Местоимение "это" обозначает "яблоки", так что переменная ***X*** имеет значение "яблоки".

Может возникнуть вопрос, как Турбо-Пролог определяет, что надо подставить "яблоки" вместо ***X*** в этом правиле. Вспомните, что в декларативных языках различие между данными программы и шагами, которые должна выполнить программа для получения решения, менее существенно,

чем в императивных языках, например, таких как Си или Паскаль. Турбо-Пролог сопоставляет все доступные ему факты (в данном случае кто что любит) с любым правилом, с помощью которого могут быть выведены новые факты.

Турбо-Пролог не предоставляет возможность программисту выполнить такой приказ, как на "данном этапе сравнить ***X*** с яблоками и проверить, что это сравнение удовлетворяет некоторым условиям". Турбо-Пролог построен так, что он автоматически пытается выполнить сопоставление, используя принципы логики предикатов. Все, что вам необходимо сделать - это, используя синтаксис Турбо-Пролога, описать к каким заключениям приведут какие условия. Другими словами, вы формулируете логические взаимоотношения, а программа использует ваши утверждения о логических связях для получения заключения.

Теперь вы, по-видимому, уже достаточно подготовлены, чтобы, во-первых, принять к сведению некоторые свойства Турбо-Пролога как декларативного языка и, во-вторых, понять, как Турбо-Пролог использует логический синтаксис предикатов для проверки фактов. В следующей главе вы подробнее ознакомитесь с точным синтаксисом и структурой программы на Турбо-Прологе, а сейчас необходимо сосредоточиться только на основных понятиях, не беспокоясь о том, как написать программу.

Написание правил и фактов в Турбо-Прологе кажется более громоздким по сравнению с записью фактов и условий на русском языке. Но простые примеры из этой главы, вероятно, дадут возможность увидеть сходство между, например, русским языком и логикой предикатов или синтаксическими конструкциями Пролога. Благодаря этому сходству вскоре исчезнет впечатление громоздкости и вы будете чувствовать себя уверенно "думая на Прологе".

2.3 Управление программой

Теперь вы знаете, что Турбо-Пролог строит заключения на основе логических соотношений, определенных в программе. В этом разделе объясняется, каким образом входные данные и логические связи программы совместно используются для генерации выходного результата.

Предикатная конструкция, называющаяся целью, используется для запуска процесса выполнения программы. Турбо-Пролог пытается сопоставить цель с фактами и правилами программы. Если цель является фактом, таким как "любит(мэри, яблоки)", то Турбо-Пролог отвечает ***True*** (истина) или ***False (***ложь); если цель содержит переменные, то Турбо-Пролог выдает либо те их значения, которые приводят к решению, если оно существует, либо сообщение ***No solutions*** (решений нет).

Данный раздел не содержит всех элементов программы на Турбо-Прологе. Они рассматриваются в следующей главе. Данный раздел содержит общие понятия и сведения, которые будут полезны позднее, когда вы начнете создавать собственные программы.

2.3.1 Цели программы

Цель - это формулировка задачи, которую программа должна решить. Цель также служит "триггером" для запуска программы. Турбо-Пролог использует как внутренние цели, которые содержатся в программе, так и внешние цели, которые вводятся с клавиатуры после запуска программы. Внутренние и внешние цели имеют больше сходства, чем различий. В этом разделе рассматриваются только внешние цели, но большая часть из того, что вы узнаете о внешних целях применимо также и к внутренним целям.

После того, как Турбо-Пролог запущен, он не выполняет ничего полезного до тех пор, пока ему не встретится оператор цели. Цель состоит из взаимосвязанных предикатов. Ее структура точно такая же как у правила или факта. В примере с Мэри, Бет и яблоками отношением (символом предиката) является "любит". Тот же самый символ "любит" может быть использован как цель. Предикат цели сопоставляется с предикатами в программе, так что цель в этом примере будет иметь объект, как и предикат "любит", рассмотренный ранее. Цель будет иметь вид:

любит(бет,яблоки).

Эта цель фактически означает, что вы хотите знать, действительно ли Бет любит яблоки в соответствии с информацией, имеющейся в программе.

2.3.2 Решение задачи

Пусть программа на Турбо-Прологе содержит факт

***likes*(*mary*, *apples*).** **/* Мэри любит яблоки */**

и правило

***likes*(*beth*, *X*) if *likes*(*mary*,*X*). /* Бет любит это, если Мэри любит это */**

Предположим также, что программа уже запущена и запросила вас ввести цель. Вы ввели

***likes*(*beth*,*apples*).** **/* бет любит яблоки */**

Теперь Турбо-Пролог предпринимает действия, необходимые для решения задачи, содержащейся в цели. Факты и правила программы используются для вывода логических заключений. Турбо-Пролог воспринимает утверждение цели и начинает поиск правил и фактов программы для сопоставления с предикатом ***likes***.

Первое сопоставление выполняется с фактом

likes(mary,apples).

Очевидно, что терм предиката цели, ***likes,*** сопоставим с таким же термом в факте. Так как предикаты ***likes*** сопоставимы, Турбо-Пролог попытается сопоставить первый объект цели ***beth*** с первым объектом факта ***mary***. Турбо-Пролог пытается сопоставлять термы слева направо до тех пор, пока очередное сопоставление окажется неуспешным. Сравним цель и факт:

likes(beth, apples).

likes(mary, apples).

Первые объекты ***beth*** и ***mary*** несопоставимы. Следовательно, вся попытка сопоставить цель и факт будет неуспешной. Турбо-Пролог не бу-

дет пытаться сопоставить второй объект цели с соответствующим объектом в правиле, даже если они и сопоставимы. Затем Турбо-Пролог осуществляет поиск следующей точки входа и находит правило

***likes*(*beth*,*X*) if *likes*(*mary*,*X*).**

Первая часть этого правила ***likes(beth,X)*** называется головой правила. Пролог пытается сопоставить цель с головой правила точно также, как он сопоставляет цель с фактом. Если Турбо-Пролог сможет сопоставить предикат и все его объекты, то он попытается выполнить часть правила, которая следует за условием if (эта часть называется телом правила).

Сейчас Турбо-Пролог пытается сопоставить цель ***likes(beth, apples)*** и голову правила ***likes(beth, X)*** :

***likes*(*beth*, *apples*).**

***likes*(*beth*, *X*).**

Цель и голова правила имеют сопоставимые предикатные термы и первые объекты. Строго говоря, ***apples*** и ***X*** не сопоставимы. Но прежде чем заключить, что сопоставление опять неуспешно, вспомните, что ***X*** работает как местоимение "это". Являясь переменной, ***X***, может быть сопоставлен с чем угодно.

В русском языке значение для "это" определяется контекстом. Всякий раз, когда Турбо-Пролог встречает переменную, пытаясь сопоставить объекты, он присваивает переменной значение, полученное из контекста для этой переменной. Так как позиция объекта ***X*** в голове правила точно такая же, как позиция ***apples*** в цели, то Турбо-Пролог присваивает значение ***apples*** переменной ***X***. Теперь ***X*** и ***apples*** означают для Турбо-Пролога одно и то же. Это связывание используется до тех пор, пока либо цель не будет вычислена, либо нельзя будет выполнить никакого другого сопоставления с учетом этого связывания.

Так как теперь ***X*** имеет значение ***apples***, то предикат ***likes***(***beth***, ***X***) имеет "значение" ***likes(beth, apples).*** Турбо-Пролог успешно сопоставил голову правила и цель, присвоив значение ***apples*** переменной ***X***.

Теперь Турбо-Пролог пытается выполнить условие ***if likes***(***mary***, ***X***). Так как переменной ***X*** присвоено значение ***aplles***, то Турбо-Пролог должен доказать или опровергнуть условие "если Мэри любит яблоки" для того, чтобы доказать действительно ли истинна голова правила "Бет любит яблоки". Таким образом, новая задача состоит в том, что бы проверить, что Мэри любит яблоки. Эта "подзадача", или подцель, была создана самим Турбо-Прологом как шаг, направленный на решение задачи, содержащейся в цели. Подцелью теперь является

likes(***mary***, ***aplles***).

Турбо-Пролог пытается выполнить эту подцель для того, чтобы доказать или опровергнуть условие головы правила ***likes***(***beth***, ***apples***).

Анализируя факты и правила программы, Турбо-Пролог находит факт

likes(***mary***, ***apples***).

и пытается сопоставить его слева направо с целью ***likes***(***mary***,***apples***).

Как вы видите, этот факт сопоставим с подцелью. Таким образом, подцель ***likes***(***mary***,***apples***) оказывается успешно доказанной. Следовательно, голова правила ***likes***(***beth***, ***apples***) имеет место. А так как голова правила истинна, то доказана истинность цели ***likes***(***beth***, ***apples***). Другими словами, цель оказалась успешной. Новый факт был успешно выведен, хотя он нигде в программе явно указан не был.

2.3.3 Результат доказательства цели

Если цель ***likes***(***beth***, ***apples***) успешно доказана, то Турбо-Пролог информирует об этом, выдавая сообщение ***True***. Если бы целью было ***likes***(***beth***, ***oranges***), а программа содержала бы те же факты и правила, то эта цель была бы неуспешной, а сообщение было бы ***False.***

На этом примере видно, что манипулирование данными и управление программой определяется фактами и правилами. В Турбо-Прологе факты и правила называются утверждениями. Факт

finds(john, gold). ***/* Джон нашел золото */***

является утверждением. Правило

is(john, rich) if finds(john, gold). /* Джон богат, если */
/* Джон нашел золото */

так же является утверждением. Так как головы утверждений, являющиеся правилами, имеют форму фактов, то правила могут рассматриваться как факты. Но, тем не менее, различия между фактами и правилами весьма существенны, и это обстоятельство используется в данной книге.

В программе на Турбо-Прологе утверждения для одного и того же предиката группируются вместе. Как это происходит, вы увидите в следующей главе.

2.3.4 Связки и другие символы

Часто условия требуют более одной предпосылки. Например, "Джон и Мэри женаты, если Мэри жена Джона и если Джон муж Мэри". На Турбо-Прологе это правило записывается в виде:

married(john, mary) If wife(john, mary) and husband(mary, john).

Данное правило имеет два условия, соединенные словом ***and*** (и). Используя терминологию Пролога, это есть связка двух условий, объединенных словом ***and.*** В Прологе вместо слова ***and*** часто используется запятая (,). В Турбо-Прологе также используется слово ***and***, однако применение запятой предпочтительнее, т.к. в этом случае чтение программы легче. Кроме того, в Прологе используется так же специальный символ для слова ***if*** (если). Это символ :- (двоеточие с последующим тире). Таким образом, если в программе на Турбо-Прологе вам встретиться символ :-, то читайте его как "если", а запятую - как "и". С использованием символов для представления "и" и "если" отношение "женаты" (***married***) записывается в виде:

married(john, mary) :-
wife(john, mary),

husband(mary, john).

Заметим, что символ :- следует за головой утверждения, а оба условия начинаются с абзацного отступа и разделены запятой. Отступы не обязательны, они используются для удобства чтения программы. Как и все утверждения, это утверждение заканчивается точкой.

Точка с запятой (;) используется для представления связки ***or*** (или). Как и для связки ***and*** (и), Турбо-Пролог допускает использование и слова ***or*** (или) и символа ;. Связка ***or*** не очень часто используется в правилах и фактах. Она имеет и другие применения, о которых вы узнаете в следующих главах.

В большинстве случаев Турбо-Пролог игнорирует избыточные пробелы и переносы на следующую строку (позже вы узнаете об исключениях из этого правила). Хотя кажется, что предыдущее утверждение читается сверху вниз, Турбо-Пролог читает его как упорядоченную слева направо конструкцию, а переносы на следующую строку игнорируются. Голова правила обычно называется левой частью, а условия, следующие за символом :-, обычно называются правой частью правила. Просмотр слева направо является важным принципом, который необходимо твердо усвоить , так как далее в этой главе будут рассматриваться внутренние процессы Турбо-Пролога.

2.3.5 Цели и подцели

Если вспомнить, что цели, подобно фактам и правилам, строятся из предикатов, то не трудно понять, что цели также могут иметь связки. Цели, имеющие связки, называются целями, имеющими подцели. Подцели утверждения цели разделяются запятыми точно так же, как условия в правилах. Когда цель введена в программу на Турбо-Прологе, то Турбо-Пролог обрабатывает ее подцели слева направо, пытаясь сопоставить подцели с фактами и правилами программы. Если одна из подцелей несопоставима, то вся цель является неуспешной. Если все подцели сопоставимы, то вся цель является успешной.

Пусть, например, программа содержит следующие утверждения:

likes*(*mary*, *apples*).** ***/* Мэри любит яблоки */

color(apples*, *red*).** ***/* цвет яблок красный */

Введем цель

likes*(*mary*, *apples*),*color(apples, red).

Утверждения заканчиваются точками и являются независимыми фактами, а подцели разделены запятыми и точка завершает цель как целое.

Самая левая подцель ***likes*(*mary*,*apples*)** сопоставима с утверждением ***likes*(*mary*,*apples*)**, так что эта подцель успешна. Следующая подцель справа сопоставима с утверждением ***color(apples, red),*** и поэтому вся цель является успешной. Информация, выведенная на экран, уведомит вас, что эта цель истинна. По-русски, цель есть "Мэри любит яблоки и яблоки красные". Либо можно сформулировать цель как вопрос "Любит ли Мэри яблоки, и красные ли эти яблоки?". Факты подтверждают эту цель.

В реальной ситуации, вероятно, потребовалось бы подтверждение вопроса, такого как "Любит ли Мэри красные яблоки?". Эта цель отличается от предыдущей. Ранее требовалось, только проверит факты, что Мэри любит яблоки и что эти яблоки красные. Сейчас необходимо знать, любит ли Мэри яблоки, если они красные. Яблоки могут быть и красные, и зеленые, и желтые. Но Мэри может вообще не любить яблоки, либо же она их любит, но они не красные. В таком случае ответ ***False***.

Для того, что бы ввести эти факты в программу, необходимо убрать утверждение

likes(***mary***, ***apples***).

и добавить правило

likes(***mary***, ***apples***) :-

color(apples, red).

По-русски это правило читается как "Мэри любит яблоки, если они красные". Если вы хотите узнать "Любит ли Мэри яблоки?", то введите цель

likes(***mary***,***apples***).

Турбо-Пролог сопоставляет цель с головой правила ***likes***(***mary***,***apples***). Но это правило имеет условие ***color(apples, red).*** Поэтому это условие должно быть доказано для того, чтобы цель могла быть успешной.

Даже если цель сама по себе не имеет подцелей, то Турбо-Пролог все равно должен сопоставить правило с условием с соответствующим утверждением. Фактически Турбо-Пролог создает подцель. Для доказательства головы правила, должны быть доказаны условия тела правила точно так же, как если бы целевое утверждение состояло из связки этих подцелей. Турбо-Пролог успешно доказывает эту подцель, обнаружив среди утверждений факт ***color(apples, red).*** Доказав эту подцель, Турбо-Пролог доказал, что голова правила истинна. Так как голова правила сопоставима с целью, то цель является успешной.

Для того, чтобы понять как Турбо-Пролог генерирует внутренние подцели, можно считать, что условия справа от ***if*** (или символа :-) становятся подцелями исходной цели, когда, пытаясь доказать эту цель, Турбо-Пролог сопоставляет ее с головой правила.

Для простоты в этом разделе были приведены ясные несложные примеры процесса сопоставления. Эти примеры были упрощены так, чтобы можно было легко увидеть логику управления программой в Турбо-Прологе. Теперь вы должны понимать, что сопоставление является очень важным элементом Турбо-Пролога.

2.4 Внутренние подпрограммы унификации Турбо-Пролога

Турбо-Пролог (как и другие реализации Пролога) имеет внутренние подпрограммы для выполнения сопоставления и связанных с ним процессов. Эти неотъемлемые части языка называются внутренними подпрограммами унификации. Эти подпрограммы выполняют сопоставление целей и подцелей

с фактами и головами правил для того, что чтобы доказать (или вычислить) эти цели или подцели. Эти же подпрограммы также определяют, сгенерированы ли новые подцели правой часть правила. Программист в соответствии с логическим синтаксисом Турбо-Пролога объявляет, какие факты и правила дают тот или иной результат при различных подцелях, а внутренние подпрограммы унификации выполняют оставшуюся часть работы.

2.4.1 Представление данных при помощи фактов и правил

В программах на Турбо-Прологе данные представляются при помощи фактов и правил. Хотя факт может быть предикатом, не содержащим объектов, большинство фактов записывается как предикат с одним или более объектами.

Некоторые примеры:

likes(mary,apples). ***/* Мэри любит яблоки */ em-***
ployee(smith,john,1984). ***/* служащий Джон Смит */***
gender(female). ***/* пол - женский */***

Если имя объекта начинается со строчной буквы, то этот объект имеет известное значение. Другими словами, объект есть в этом случае константа. Когда Турбо-Пролог пытается сопоставить две константы, то это сопоставление успешно только в том случае, если константы равны. Следовательно, ***mary*** никогда не сопоставима с ***beth***.

Пусть программа содержит следующие факты:

likes(***beth***, ***apples***).
likes(***mary***, ***pears)***.
likes(***mary***, ***oranges***).
likes(***mary***, ***apples***).

Теперь предположим, что введена внешняя цель

likes(***mary***,***apples***).

Как вы понимаете, цель по структуре подобна факту и правилу. Оба объекта в этой цели ***mary*** и ***apples*** являются константами. Пытаясь выполнить эту цель, Турбо-Пролог просматривает программу слева направо, выполняя поиск предиката с тем же предикатным термом, что и целевой предикат. (Вспомним, что то, что выглядит как "сверху вниз", на самом деле есть "слева направо"). Это первый шаг, выполненный внутренними унификационными подпрограммами.

В этом примере предикатный терм цели есть ***likes***. Первый факт для ***likes***, встречающийся в программе, это ***likes***(***beth***, ***apples***). Сопоставив предикатный терм в цели и факте, внутренние унификационные подпрограммы теперь пытаются сопоставить первый объект для ***likes*** в цели и факте. Так как константа ***mary*** не сопоставима с константой ***beth***, то попытка неуспешна. Теперь внутренние унификационные подпрограммы пытаются сопоставить цель со следующим предикатом ***likes***. На этот раз сопоставление успешно, так как константа ***mary*** сопоставима с точно такой же константой в правиле. Но для завершения сопоставления внутренние унификационные подпрограммы должны сопоставить следующий объект справа. Константы

pears и ***apples*** не сопоставимы. Так как и эта попытка неуспешна, то внутренние унификационные подпрограммы опять повторяют попытку для следующего предиката ***likes***.

Каждое возможное сопоставление проверяется слева направо. Очевидно, что ***likes***(***mary***, ***apples***) не сопоставимо с ***likes***(***mary***, ***oranges***).

Внутренние унификационные подпрограммы просматривают всю программу слева направо, пробуя сопоставить каждый факт или правило с целью. Только предикаты с соответствующими объектами проверяются на сопоставимость с целью. Последний кандидат для сопоставления - это факт ***likes***(***mary***,***apples***). Оба объекта в этом факте сопоставимы с соответствующими объектами в предикате цели. Цель успешно доказана. Рассматривая этот пример, следует понять, что механизм внутренней унификации обработал сверху до низу (или слева направо) все подходящие факты и правила. После того как сопоставимый предикат был найден, объекты этого предиката сопоставлялись слева направо, пока это сопоставление не становилось либо успешным, либо неуспешным. Если сопоставление заканчивалось неуспешно, то проверялись следующие факты или правила до тех пор, пока либо одно из сопоставлений заканчивалось успешно, либо все релевантные факты и правила оказывались опробованными и неуспешными.

В случае, когда объектами сопоставления являются две константы, то успешный результат будет лишь тогда, когда они совпадают. Другими словами, константа всегда сопоставима сама с собой.

2.4.2 Обработка правил и фактов во время унификации

Переменные в Турбо-Прологе являются аналогами местоимений в естественном (в частности, русском) языке. В определенный момент переменная может получить некоторое значение. Иногда этого может не произойти. Возможно вы имеете некоторое представление о поведении переменных, основанное на других языках программирования. Если это так, то обратите особое внимание на то, как ведут себя переменные Турбо-Пролога. Вы можете обнаружить некоторые неожиданные отличия.

Следующий пример включает кое-что из того, что вы узнали в предыдущем разделе о внутренних подпрограммах унификации. Данными является факт

likes(***mary***, ***apples***). /* Мэри любит яблоки */

Вы хотите сформулировать цель, чтобы узнать, что любит Мэри (если таковые объекты имеются). Вид этой цели следующий:

likes(***mary***, ***What).*** ***/* M***эри любит Что */

Вспомним, что в Турбо-Прологе объект, начинающийся с прописной буквы, есть переменная. Переменная в этой цели есть ***What*** (что). Когда внутренние унификационные подпрограммы пытаются сопоставить эту цель с фактами или правилами программы, переменная ***What*** не имеет значения. ***What*** не равна нулю и не равна пробелу. Эта переменная не имеет даже значения, являющегося "мусором" (какой бы мусор не оказался в памяти, отведенной для переменной).

Переменная ***What*** не имеет значения, так как она **неозначена**, или **неинициализирована**. Эти два термина используются как взаимозаменяемые. Неозначенные переменные еще называются свободными переменными. Когда свободная переменная цели **What** сопоставляется с соответствующим объектом ***apples***, значением ***What*** становится ***apples***. Теперь переменная ***What*** означена (или инициализирована) объектом ***apples***. Другими словами, эта переменная более не свободна (связана).

Пока ***What*** имеет значение ***apples***, эта переменная для Турбо-Пролога "означает" ***apples***. Фактически всякий раз, когда неозначенная переменная сопоставляется с константой, она получает значение этой константы. Турбо-Пролог выдает на экран, что ***What=apples*** , и, что внутренняя подпрограмма унификации нашла все (единственный) объекты, сопоставимые с целью.

Теперь предположим, что программа содержит следующие факты:

likes(***mary***, ***apples***). /* Мэри любит яблоки */
likes(***mary***, ***pears).*** /* Мэри любит персики */
likes(***mary***, ***popcorn***). /* Мэри любит кукурузные зерна */

Такая же цель, что и выше ***likes***(***mary***, ***What)*** сопоставима со всеми этими фактами. Поэтому Турбо-Пролог выдает все значения для ***What,*** которые удовлетворяют этой цели. Внешние цели заставляют внутренние унификационные подпрограммы Турбо-Пролога найти все решения, удовлетворяющие цели. Но внутренние цели заставляют внутренние унификационные подпрограммы останавливаться после первого успешного сопоставления цели.

Может показаться, что переменная ***What*** получает значения ***apples***, ***pears*** и ***popcorn*** в одно и тоже время, но это не так. Подпрограммы внутренней унификации присваивают переменной конкретное значение только во время сопоставления. Переменная становится вновь свободной, когда сопоставление оказывается неуспешным или цель оказывается успешно вычисленной.

Программисты, имеющие опыт работы с императивными языками, иногда нелегко понимают "ликвидацию значения" или освобождение переменной. Обычно в императивных языках, если переменная получила значение, то сохраняет его в области своего определения до тех пор, пока ей не будет присвоено новое значение. Такие присваивания значений находятся под строгим контролем операторов программы. Но в Турбо-Прологе присваивания значений переменным выполняется внутренними подпрограммами унификации. Переменные становятся свободными, как только для внутренних подпрограмм унификации отпадает необходимость связывать некоторое значение с переменной для выполнения доказательства подцели.

Может показаться, что в Прологе нельзя управлять значениями, которые присваиваются переменным. Действительно, Пролог не имеет "грубых средств" для управления значениями, точно также в Прологе нет "грубых средств" для управления ходом выполнения программы. Но, понимая принцип работы внутренних унификационных подпрограмм, программист

может использовать логический синтаксис Турбо-Пролога для указания того, что необходимо выполнить и какие данные должны быть найдены. Если это не совсем ясно, то не беспокойтесь. Примеры помогут вам ближе познакомиться с этим процессом. В остальных главах книги содержатся специально подготовленные примеры, которые должны вам помочь.

Пример использования оператора = поможет лучше понять процесс означивания переменой. В языке программирования, таком как Бэйсик, оператор = может означать "Сделать два терма равными", например в выражении ***X***=6. Здесь оператор = является оператором присвоения значения. Его действие состоит в том, чтобы сделать ***X*** равным 6, т. е. присвоить значение 6 переменной ***X***. Но в Бейсике оператор = имеет еще и другой смысл - проверить равенство двух термов. Такое использование символа = встречается в условных выражениях, таких как ***IF X=6 GOSUB 3010***. Оператор = , использованный таким образом, выражает следующее: "Два терма, между которыми я нахожусь, должны быть равными для того, чтобы весь оператор был истинным". Оператор = интерпретируется как оператор присваивания или как оператор проверки равенства в зависимости от контекста. (Но некоторые языки для присваивания и равенства используют различные символы).

Турбо-Пролог также использует оператор = , но используется ли он как оператор сравнения или как оператор присваивания определяется в зависимости от того, являются ли значения термов свободными или означенными.

Рассмотрим, например, выражение

apples = ***apples*** .

Это выражение может быть подцелью, сгенерированной внутри Турбо-Пролога. Так как оба значения известны, то оператор = является оператором сравнения. В этом случае равенство истинно, и подцель является успешной.

Рассмотрим выражение

apples = ***oranges*** .

Так как эти два терма имеют различные значения, то равенство ложно. Рассмотрим пример, в котором используется переменная:

X = ***apples*** .

Это выражение так же могло бы быть подцелью, сгенерированной внутри Турбо-Пролога во время попытки вычислить цель. Но здесь знак равенства не обязательно является оператором присваивания. Эта подцель присваивает значение ***apples*** переменной ***X*** (означивает эту переменную) только в том случае, если ***X*** еще не была означена. Но если ***X*** уже имеет значение, то оно известно внутренним подпрограммам унификации и сравнивается с константой ***apples***. Если эти два значения одинаковы, то подцель успешна; если они различны, то подцель неуспешна.

Интерпретация Турбо-Прологом оператора = зависит от того, известны ли оба значения или нет. Если оба значения известны, то оператор интерпретируется как оператор сравнения, даже если оба терма переменные. Если известно только одно из значений, то это известное значение будет присвоено другому. Для Турбо-Пролога несущественно, известно ли значение справа или слева от = ; в любом случае неизвестное получит значение из-

вестного. Например, предположим, что переменная ***Fruit*** (фрукт) неозначена. Подцель ***apples=Fruit*** вызовет присвоение значения ***apples*** переменной ***Fruit***. Теперь предположим, что подцель ***Fruit=X*** непосредственно следует за предыдущей, и что ***X*** свободная переменная. Результат попытки выполнить эту подцель состоит в присваивании переменной ***X*** значения ***apples***. Вы видите, что даже если оба терма ***Fruit*** и ***X*** переменные, Турбо-Пролог присваивает (означивает) неозначенной переменной известное значение. Во многих языках программирования присваивание встречается только в форме присваивания правого терма левому. Но в Турбо-Прологе присваивание может иметь место в обоих направлениях. В качестве последнего примера рассмотрим подцель ***X=oranges.***

Переменная ***X*** получила значение ***apples*** в результате предыдущей инициализации. Константа ***oranges*** (апельсины) также имеет известное значение. Так как обо значения известны, то Турбо-Пролог проверяет их равенство. Результирующее выражение ***apples=oranges*** ложно. Следовательно, эта подцель неуспешна и цель также неуспешна. После этого неуспешного доказательства цели переменные ***Fruit*** и ***X*** становятся неозначенными.

Изучение поведения оператора = при выполнении сравнения и присваивания весьма полезно по двум причинам. Во-первых, оператор = широко используется при программировании на Турбо-Прологе. Во-вторых, что более важно, оператор = ведет себя точно так же, как ведут себя внутренние унификационные подпрограммы при сопоставлении целей и подцелей с фактами или правилами программы. Переменным могут быть присвоены значения во время попыток выполнить цели, и они же могут сравниваться для проверки равенства. Результаты означивания передаются дальше другим подцелям, которые используют соответствующие объекты во время присваивания значений или сравнения значений. Если некоторая подцель оказывается неуспешной или последующие подцели не используют значений означенных переменных, то эти переменные становятся неозначенными, т.е. они становятся снова свободными.

Следующая глава дает более подробное описание означивания и освобождения переменных, а сейчас достаточно помнить только, как Турбо-Пролог вычисляет и присваивает значения.

2.4.3 Откат

Откат - это механизм, который Турбо-Пролог использует для нахождения дополнительных фактов и правил, необходимых при вычислении цели, если текущая попытка вычислить цель оказалась неудачной. Некоторая аналогия прояснит понятие отката. Предположим, что ваша цель - попасть домой к другу, но вы не имеете точного представления, как туда добраться. Однако вы знаете, что после поворота с автострады на перекрестке необходимо свернуть направо. Друг живет в доме через дорогу от входа в парк. После того, как вы свернете с автострады и повернете направо на первом перекрестке, вы не найдете входа в парк. Поэтому вы вернетесь (выполните откат) к перекрестку и продолжите первоначальное движение до следующего

перекрестка. Здесь вы опять повернете направо и поищите вход в парк. Если вам вновь неудастся найти вход в парк, вы еще раз вернетесь назад (выполните откат) и сделаете попытку на следующем перекрестке. Вы будете повторять этот процесс до тех пор, пока вам не удастся достичь вашей цели найти дом, или же вы потерпите полную неудачу и откажитесь от этой затеи.

Для вас этот вид поиска, вероятно, будет утомительным, но Турбо-Пролог неутомим при поиске всех возможных путей решения задачи. Так же как и вы, Турбо-Пролог использует откат для того, чтобы попробовать новые пути к решению. И, так же как вы, Турбо-Пролог использует всю доступную информацию для выбора подходящего направления. Типичная программа на Турбо-Прологе содержит факты и правила, основанные на самых различных взаимосвязях предикатов. Правила могут иметь несколько правосторонних частей, которые соединены связками. Цель может состоять из нескольких подцелей, а переменные могут быть объектами предиката как в утверждениях, так и в подцелях. Другими словами, типичные программы на Турбо-Прологе представляют собой комбинации всех элементов, о которых вы узнали до сих пор в этой главе.

Турбо-Пролог пытается вычислить цель при помощи сопоставления терма предиката и объектов цели с соответствующими элементами в фактах и головах правил. Сопоставление выполняется слева направо. Некоторые подцели, вероятно, будут неуспешными при сопоставлении с некоторыми фактами или правилами, поэтому Турбо-Прологу требуется способ "запоминания точек", в которых он может продолжить альтернативные попытки найти решение. Прежде чем попробовать один из возможных путей решения подцели, Турбо-Пролог фактически помещает в программу "указатель". Указатель определяет точку, в которую может быть выполнен откат, если текущая попытка будет неудачной.

По мере того, как Турбо-Пролог успешно заканчивает свои попытки вычисления подцелей слева направо, указатели отката расставляются во всех точках, которые могут привести к решению. Если некоторая подцель оказывается неуспешной, то Турбо-Пролог откатывается влево и останавливается у ближайшего указателя отката. С этой точки Турбо-Пролог начинает попытку найти другое решение для неуспешной цели.

До тех пор, пока следующая цель на данном уровне не будет успешной, Турбо-Пролог будет повторять откат к ближайшему указателю отката. Эти попытки выполняются внутренними подпрограммами унификации и механизмом отката. Окончательным результатом будет либо успешное, либо неуспешное вычисление цели.

С некоторыми небольшими дополнениями факты и правила о Мэри, Бет и яблоках могут быть использованы для иллюстрации отката и внутренней унификации. Данный пример содержит переменные, а также факты и правила, которые обрабатываются подпрограммой внутренней унификации при выполнении операций сопоставления.

Факты для отношения ***likes***:

likes(***mary***, ***pears***). ***/* Мэри любит персики */***

```
likes(mary, popcorn).     /* Мэри любит кукурузные зерна */
likes(mary, apples).      /* Мэри любит яблоки  */
```

Ниже следуют правила, из которых можно сделать вывод о том, что любит Бет:

```
likes(beth,X) :-          /* Бет любит то,          */
        likes(mary,X),    /* что любит Мэри,        */
        fruit(X),         /* если это фрукт,        */
        color(X,red).     /* и если он красный      */
likes(beth,X) :-          /* Бет любит то,    */
        likes(mary,X),    /* что любит Мэри,        */
        X=popcorn.        /* если это кукурузные зерна */
```

А эти утверждения дают конкретные факты:

```
fruit(pears).          /* персики - фрукт */
fruit(apples).         /* яблоки - фрукт */
color(pears, yellow).  /* цвет персиков желтый    */
color(oranges, orange). /* цвет апельсинов оранжевый */
color(apples, red)./* цвет яблок   красный       */
color(apples, yellow).  /* цвет яблок   желтый       */
```

Заметьте, что эти предикаты сами по себе не образуют законченной программы на Турбо-Прологе. В следующей главе вы узнаете об остальных необходимых частях программы.

Ниже дано целевое утверждение, которое используется для выборки информации из приведенных утверждений:

```
likes(beth, X).
```

Эта цель означает "Что любит ***Beth***?". Для того, чтобы ответить на данный вопрос, внутренние унификационные подпрограммы Турбо-Пролога ищут факты или голову правила, сопоставимую с этим целевым утверждением. Поиск начинается с первого утверждения для отношения ***likes***, которое содержит три факта о том, что любит Мэри. Турбо-Пролог опробует все утверждения слева направо (сверху вниз). Сопоставление для всех них будет неуспешным, так как константа ***beth*** несопоставима с константой ***mary***.

Внутренние унификационные подпрограммы Турбо-Пролога переходят к правилу:

```
likes(beth, X) :-
        likes(mary, X),
        fruit(X),
        color(X, red).
```

Переменные, как в голове правила, так и в цели неозначены, так что цель и голова правила сопоставимы. Так как голова (левая часть) первого правила сопоставима с целью, то факты правой части становятся подцелями, которые Турбо-Пролог должен вычислить, обрабатывая их слева направо. Вспомним, что Турбо-Пролог рассматривает термы, разделенные запятыми, как следующие друг за другом, даже если они находятся на разных строках.

Так как другие утверждения для ***likes*** следуют за этим правилом, то Турбо-Пролог помещает указатель отката на начало следующего правила для ***likes***.

Первой подцелью является ***likes***(***mary***, ***X***). Это новая подцель, поэтому Турбо-Пролог снова начинает просмотр с вершины списка предикатов для ***likes*** и находит ***likes***(***mary***, ***X***). Этот факт сопоставим с подцелью ***ikes***(***mary***, ***X***), так как все ее термы сопоставимы, поскольку ***X*** получила значение ***pears*** во время унификации. Теперь подцель ***likes***(***mary***, ***X***) успешно вычислена, но правило, которое сгенерировало эту подцель сгенерировало также и другие подцели, которые еще должны быть доказаны. Поэтому внутренние унификационные подпрограммы ставят указатель на следующий факт для ***likes***. Этот указатель показывает, что существует по крайней мере еще одно утверждение для ***likes***, которое может быть использовано для вычисления текущей подцели. В случае, если следующая подцель окажется неуспешной, механизм отката будет иметь точку для поиска другого кандидата для вычисления цели.

К этому моменту цель ***likes***(***beth***, ***X***) была сопоставлена с головой правила ***likes***(***beth***, ***X***). Внутренние унификационные подпрограммы установили указатель на голову следующего правила ***likes***(***beth***, ***X***) и начали попытки вычисления утверждений в правой части правила. В результате была сгенерирована подцель ***likes***(***mary***, ***X***). Пытаясь вычислить эту подцель, унификационные подпрограммы обнаружили сопоставимое утверждение ***likes***(***mary***, ***pears)***. Теперь ***X*** получил значение ***pears,*** а значение подцели стало ***likes***(***beth***, ***pears)***. Существуют и другие утверждения, которые могут быть использованы для вычисления подцели, поэтому указатель отката был установлен на ***likes***(***mary***, ***popcorn***).

Следующая подцель справа есть ***fruit(X***). Так как сейчас ***X*** имеет значение ***pears***, то подцель означает ***fruit(pears).*** Внутренние унификационные подпрограммы находят сопоставление с первым утверждением для ***fruit.*** Так как существуют другие утверждения, которые могут быть использованы для вычисления подцели, то создается еще один указатель отката на это утверждение.

Теперь два указателя отката отмечают альтернативные пути к решению правила:

```
likes(beth,X) :-
        likes(mary, X),
        fruit(X),
        color(X, red).
```

Это точки 2 и 3. (Точка 1 это альтернативный путь к решению основной цели). Последняя отмеченная точка всегда является точкой, с которой будет начат поиск альтернативного решения. Последняя подцель правила есть ***color***(***X***, ***red***). Внутренние подпрограммы унификации, всегда просматривая утверждения слева направо, пытаются сопоставить подцель с утверждением ***color(pears,yellow).*** Так как ***X*** имеет значение ***pears,*** то текущая подцель есть ***color(pears,red***).

Все попытки вычислить эту подцель неуспешны, так как программа не содержит утверждения ***color*(*pears,red***). Подцель вычислена неуспешно.

Внутренние унификационные подпрограммы выполняют откат к последнему указателю, который установлен на ***fruit(pears).*** Это сопоставление неуспешно, так что механизм отката повторяет откат к ближайшему предыдущему указателю, который установлен на ***likes*(*mary*,*popcorn).***

Переменная ***X*** опять становится свободной из-за неуспешного вычисления подцели во время попытки сопоставления с ***pears***. В точке, определяемой указателем отката, Турбо-Пролог находит факт ***likes*(*mary*,*popcorn)***. Указатель отката, отмеченный цифрой 4, устанавливается на следующее утверждение для ***likes***. Переменная ***X*** имеет значение ***popcorn,*** так что теперь подцели эквивалентны "Мэри любит кукурузные зерна и кукурузные зерна фрукт красного цвета".

Подцель ***fruit(popcorn)*** не может быть доказана при помощи фактов и правил, имеющихся в программе, так что подцель ***likes*(*mary*,*X***) неуспешна. Переменная ***X*** освобождается, и подцель ***likes*(*mary*,*X***) в правиле ***likes*(*beth*,*X***) имеет еще один шанс на успех, так как был отмечен для отката еще один факт, о том что любит Мэри. Внутренние унификационные подпрограммы выполняют откат в точку 4.

Теперь подцель сопоставляется с ***likes*(*mary*, *apples***), и ***X*** получает значение ***apples***. Выполняется попытка для следующей подцели ***ruit(apples***). Первое утверждение для ***fruit*** имеет объект ***pears***. Объекты не сопоставимы, так что внутренние унификационные подпрограммы переходят к следующему факту ***fruit*(*apples***), который сопоставим с подцелью.

Наконец, последняя подцель первого правила проверена. Снова делается попытка сопоставить ее с фактом для ***color***. В этот момент подцель есть ***color*(*apples*, *red***). Начав с вершины списка фактов для ***color***, внутренние унификационные подпрограммы пытаются сопоставить эту подцель с фактами ***color(pears, yellow), color*(*oranges*, *orange***) и ***color*(*apples*, *red***). Во время этой последней попытки объект ***apples*** (присвоенный переменной ***X***) сопоставляется с объектом ***apples*** в факте, но последние объекты ***red*** и ***yellow*** не сопоставимы, так что попытка неудачна. Последний факт, связанный с цветом это ***color*(*apples*, *red***), который сопоставим с подцелью ***color*(*apples*, *red***).

С успешным сопоставлением последней подцели правило доказано. Переменная ***X***, получив значение ***apples***, тем самым доказывает правую часть. Все правило с переменной ***X***, означенной объектом ***apples***, с точки зрения внутренних унификационных подпрограмм выглядит как

```
likes(beth, apples) :-
        likes(mary, apples),
        fruit(apples),
        color(apples, red).
```

Выдав сообщение ***X*=*apples***, Турбо-Пролог показывает, что для цели найдено по крайней мере одно решение.

Снова используется последняя подцель. Теперь со значением

color(***apples***,***red***).

Еще раз все утверждения для ***color*** проверяются по очереди для сопоставления с новой подцелью. Сопоставление найдено в последнем утверждении ***color***(***apples***, ***red***). Теперь все три подцели правила доказаны. Переменная ***X*** имеет значение ***apples***. Следовательно, сейчас голова правила имеет вид: ***likes***(***beth***, ***apples***). Эта голова правила сопоставима с утверждением цели

likes(***beth***, ***X***).

так что цель успешна, и Турбо-Пролог выдает сообщение ***X=apples***. Поскольку внешняя цель была использована с программой, являющейся "черным ящиком" для Турбо-Пролога, то он продолжает поиск других решений для цели. Цель была успешной, так что переменная ***X*** освобождена и может быть снова означена подпрограммами внутренней унификации.

Поиск решений снова начинается с указателя отката, который теперь является последним. Это указатель точки 1. Указатель точки 2 был удален, так как в конце пути было найдено решение.

Теперь внутренние унификационные подпрограммы начинают поиск с правила

likes(***beth***, ***X***) :-
likes(***mary***, ***X***),
X=popcorn.

Снова первая подцель есть ***likes***(***mary***, ***X***), и внутренние унификационные подпрограммы осуществляют поиск утверждений ***likes*** для сопоставления. Утверждение

likes(***mary***, ***pears)***.

сопоставимо с подцелью, так что ***X*** получает значение ***pears***. Указатель для отката устанавливается на следующее утверждение ***likes***(***mary***, ***popcorn***).

Вычислив текущую подцель и означив ***X*** объектом pears, Турбо-Пролог пытается вычислить оставшуюся подцель, которая есть

X=popcorn.

Как было объяснено ранее в этой главе, оператор = работает как оператор сравнения, поскольку значения обоих термов известны: переменная ***X*** имеет значение pears, а ***popcorn*** есть константа. Операция сравнения будет неуспешной:

pears=popcorn.

Термы не сопоставимы, поэтому подцель неуспешна, и ***X*** снова освобождена.

Теперь внутренние унификационные подпрограммы выполняют откат к последнему указателю, с тем, чтобы попробовать альтернативный путь решения. Последний указатель отката установлен на утверждение

likes(***mary***, ***popcorn***).

Ранее ***X*** была освобождена, поэтому сейчас она получает значение ***popcorn***. Установив указатель отката на следующее утверждение ***likes***(***mary***,***apples***), внутренние унификационные подпрограммы снова пытаются вычислить подцель ***X=popcorn***. Внутренние значения подтермов есть

popcorn*=*popcorn.

Сопоставление успешно, и последняя подцель правила успешно вычислена. Переменная ***X*** в голове правила имеет значение ***popcorn***. Таким образом, выведенный факт есть ***likes***(***beth***, ***popcorn***). Турбо-Пролог сообщает об этом, выдавая ***X*=*popcorn***. Теперь найдены два решения, а указатель отката остался на утверждении

likes(***beth***, ***popcorn***).

Турбо-Пролог возвращается в указанную точку и пытается найти еще одно решение.

Вспомним, что данный путь является альтернативным для второй подцели второго правила для ***likes***. Следовательно, имеем подцель

likes(***mary***, ***X***).

Так как ***X*** снова освобождена, то она получает значение ***apples*** и подцель снова успешна. Хотя среди утверждений для ***likes*** и не осталось фактов, но еще остались два правила, так что указатель устанавливается на факт ***likes***(***mary***,***apples***). Возвратившись к следующей подцели, Турбо-Пролог сравнивает ***X*=*popcorn*** или ***apples*=*popcorn***. Иными словами, подцель неуспешна.

Снова внутренние унификационные подпрограммы выполняют откат к голове правила ***likes***(***beth***, ***X***). Эта голова правила не сопоставима с подцелью ***likes***(***mary***, ***X***), управляющей данной попыткой использовать данный альтернативный путь, поэтому унификационный механизм проверяет сопоставимость головы следующего правила. Эта голова правила также есть ***likes***(***beth***, ***X***), поэтому снова из-за несопоставимости ***beth*** и ***mary***, сопоставление оказывается неудачным.

На этот раз правило не смогло дать решение, и цель оказалась неуспешной. Ранее найдены два решения. Чтобы показать, что на этом процесс завершен, Турбо-Пролог выдает сообщение ***Two solutions*** (Два решения).

2.5 Заключение

В данной главе Турбо-Пролог рассматривался на концептуальном уровне. Как вы узнали, Турбо-Пролог является декларативным языком, основанным на логике предикатов. Предикаты в общем случае определяют отношения между объектами.

Факты и правила являются утверждениями, которые образуют данные программы на Турбо-Прологе. Правила имеют левую часть (голову) и правую часть(тело). Левая часть правила истинна, если истинна правая часть правила. Правила генерируют новые факты, когда все утверждения в теле оказываются вычисленными.

Цели - это конструкции на основе предикатов, которые объявляют, что должна доказать программа на Турбо-Прологе. Связки в целях и правилах выполняют генерацию подцелей в качестве шага процесса доказательства цели.

Внутренние унификационные подпрограммы означивают переменные. Означенные переменные и константы имеют значения, "известные" Турбо-Прологу. Свободные переменные значений не имеют. Турбо-Пролог использует откаты для определения альтернативных путей вычисления цели или подцели. Если подцель оказалась неуспешной, а указатели отката были установлены, то для предыдущей подцели будет сделана попытка добиться успеха, начиная с точки отката.

Понимание поведения переменных, унификации и отката может оказаться весьма сложным для начинающих знакомиться с Турбо-Прологом. Эта глава дала возможность познакомиться с несколькими примерами, иллюстрирующими связь между переменными, унификацией и откатом во время работы системы. После того, как вы продолжите знакомство со следующими главами, вы почувствуете себя более уверенными в скрытых операциях, выполняемых программами на Турбо-Прологе.

Глава 3. Основы программирования на Турбо-Прологе

3.1 Введение

Синтаксис и структура программ Турбо-Пролога, описанию которых посвящена данная глава, отражают концепции логики предикатов, преставленье в гл. 2.

В целях упрощения организации фактов и правил Турбо-Пролог поддерживает составные доменные структуры; кирпичиками для их создания служат базисные типы доменов Турбо-Пролога. В настоящей главе рассматривается вопрос создания составных объектов и доменных структур на основе этих базисных типов.

Примеры программ главы ставят целью продемонстрировать новые концепции и методы программирования, а упражнения дают возможность поэкспериментировать с программами. Прочитав главу, Вы уже будете иметь достаточно знаний об использовании некоторых полезных приемов программирования на Турбо-Прологе.

3.2 Структура программ Турбо-Пролога

Любая программа, написанная на Турбо-Прологе, состоит из пяти разделов. Таковыми являются раздел описания доменов, раздел базы данных, раздел описания предикатов, раздел описания цели и раздел описания утверждений. Ключевые слова ***domains, database, predicates, goal*** и ***clauses*** отмечают начала соответствующих разделов.

Назначение этих разделов таково:

- раздел ***domains*** содержит определения доменов, которые описывают различные классы объектов, используемых в программе;
- раздел ***database*** содержит утверждения базы данных, которые являются предикатами динамической базы данных. Если программа такой базы дан-

ных не требует, то этот раздел может быть опущен. Возможности динамической базы данных описываются в гл. 9 данной книги;
- раздел ***predicates*** служит для описания используемых программой предикатов;
- в разделе ***goal*** на языке Турбо-Пролога формулируется назначение создаваемой программы. Составными частями при этом могут являться некие подцели, из которых формируется единая цель программы;
- в раздел ***clauses*** заносятся факты и правила, известные априорно. О содержимом этого раздела можно говорить как о данных, необходимых для работы программы.

Большинство программ, однако, не содержит всех пяти названных разделов в силу причин, о которых будет сказано несколько позднее.

Турбо-Пролог обеспечивает возможность включения в программу комментариев, которые обрамляются символами /* и */. Комментарии можно помещать в любом месте программы, причем на их длину нет практически никаких ограничений. Для того, чтобы служить своему назначению, комментарии должны содержать информацию о самой программе, имени программного файла, компиляторе, базе данных, а также о назначении каждого из предикатов и правил, которые не являются в достаточной степени очевидными.

3.2.1 Описание доменов и предикатов

В гл. 2 приводилось несколько примеров использования предиката ***likes***, как, например,

likes(***mary***, ***apples***).

Вспомним, что ***likes*** здесь является предикатом (термом предиката), а ***mary*** и ***apples*** - объектами предиката. Турбо-Пролог требует указания типов объектов для каждого предиката программы. Некоторые из этих объектов могут быть, к примеру, числовыми данными, другие же - символьными строками. В разделе ***predicates***, поэтому, Вы должны задать тип объектов каждого из предикатов.

Для того чтобы предикат ***likes*** можно было использовать в программе, необходимо сделать следующее описание:

```
predicates
        likes(symbol, symbol)
```

Это описание означает, что оба объекта предиката ***likes*** относятся к типу ***symbol***. Этот тип является одним из базисных типов Турбо-Пролога; базисные типы будут описаны в следующем разделе главы.

В некоторых случаях, однако, представляется необходимым иметь возможность несколько больше конкретизировать тип используемого предикатом объекта. Например, в предикате ***likes*** объекты имеют смысл "тот, кто любит" и "вещь, которую любят". Турбо-Пролог позволяет конструировать свои собственные типы объектов из базисных типов доменов. Предположим, для примера, что объектам предиката ***likes*** Вы хотите присвоить соот-

ветственно имена ***person*** и ***thing***. Тогда в разделе программы ***domains*** должны появиться такие описания:

```
domains
        person, thing = symbol
predicates
        likes(person, thing)
```

Имена ***person*** и ***thing*** при этом будут обозначать некие совокупности (домены) значений. В примере из гл. 2 термы ***mary*** и ***beth*** принадлежат домену ***person***, а ***apples*** - домену ***thing.***

Любое значение домена ***person*** может в утверждениях занимать место объекта person из соответствующего предиката. То же самое можно сказать и про домен ***thing.*** Рассмотрим, например, такие три утверждения:

likes(john, camera).
likes(tom, computer).
likes(kathy, computer).

Термы ***john, tom*** и ***kathy*** принадлежат здесь к домену ***person***,а термы ***camera*** и ***computer*** - к домену ***thing***. Все три утверждения восходят к одному и тому же предикату - ***likes***; отличие состоит лишь в значениях, которые принимают объекты. Другими словами, все три утверждения являются вариациями одного и того же предиката.

* Описание доменов

Турбо-Пролог имеет 6 встроенных типов доменов: символы, целые числа, действительные числа, строки, символические имена и файлы. Тип каждого из доменов должен быть объявлен в разделе программы ***domains***.

В таблице 3.1 приведены все 6 стандартных типов доменов Турбо-Пролога.

Таблица 3.1. Стандартные типы доменов Турбо-Пролога

Тип данных	**Ключевое слово**	**Диапазон значений**	**Примеры использования в Турбо-Прологе**
Символы	**char**	Все возможные символы	'a', 'b', '#', 'B', '\13', '%'
Целые числа	**integer**	от -32768 до 32767	-63, 84, 2349, 32763
Действительные числа	**real**	от +1E-307 до +1E308	-42769, 8324, 360, 093, 1.25E23,5.15E-9

Строки	**string**	Последовательность символов (не более 250)	"today", "123", "just_a_reminder"
Символические имена	***symbol***	1.Последовательность букв, цифр и знака подчеркивания; первый символ - строчная буква	pay_check, school_day, flower
		2.Последовательность любых символов, заключенная в кавычки	"Stars ***and*** Stripes", "singing in the rain"
Файлы	**file**	Допустимое в DOS имя файла	mail.txt, BIRDS.DBA

Следующий предикат иллюстрирует использование доменов различных типов:

payroll(employee_*name*, pay_rate, weekly_hours)

Этот предикат требует такого описания доменов:

employee_*name* = *symbol*
pay_rate = integer
weekly_hours = real

Описание домена ***employee_name*** показывает, что объекты этого домена являются символическими именами. Аналогично, объекты домена ***pay_rate*** - это целые числа, а домена ***weekly_hours*** - действительные. Примером корректных утверждений, использующих предикат ***payroll***, могут служить

payroll("*John* Walker",16,45.25).
payroll("Arthur Berman",28,32.50).
payroll("Sandy Taylor",23,40.00).

* Описание предикатов

В программах, написанных на Турбо-Прологе, предикаты используются для представления, как данных, так и правил для оперирования данными. Описываются предикаты в разделе ***predicates***. Терм предиката представляет собой строку символов, первым из которых является строчная буква. Предикаты могут иметь очень простой вид, как, например,

go
do_menu
repeat
look_for_fruits
search_for_items

Такие имена пригодны для именования и правил, и целей программы. Например, имя предиката ***go*** ассоциируется с началом какого-либо процесса, соответственно оно хорошо подходит в качестве имени цели программы.

Имя ***do_menu*** более специфично, оно может служить в качестве имени правила, создающего меню. Предикаты подобные только что приведенным обычно называют "голыми", поскольку у них отсутствуют объекты.

Однако, в большинстве случаев, особенно, когда правило используется в качестве подцели другого правила, значения переменных одного из правил используются при попытке удовлетворить другое. Так, значение ***X*** из левой части правила

likes(***beth***, ***X***) ***if***

likes(***mary***, ***X***).

используются при попытке удовлетворить правую часть.

В подобных случаях объекты предикатов определяются при описании этих правил в разделе программы ***predicates***:

```
predicates
        likes(symbol,symbol)
        book(symbol,symbol,integer)
                        /* книга (название, автор, количество страниц) */
        person(symbol,char,integer)
                        /* некто (имя, пол (м или ж), возраст) */
        do_search(integer)
        sum(real, real, real, real)
```

Заметим, что перечень объектов предиката заключается в круглые скобки, а в качестве разделителя используется запятая. Так предикат ***likes*** имеет два объекта, каждый из которых относится к базисному типу ***symbol***. Этот предикат можно использовать для создания утверждений типа

likes(***mary***,***peaches***).

likes(***john***, ***plums***).

likes(***jack***,***oranges***).

Поскольку все термы : ***mary***, ***peaches***, ***john***, ***plums***, ***jack*** и ***oranges*** удовлетворяют требованиям, предъявляемым к символическим именам, то эти утверждения не противоречат описанию предиката в разделе ***predicates***.

Ввиду того, что Вы используете стандартные базисные типы доменов, нет необходимости отдельно описывать домены объектов этих утверждений. Если, однако, задаться целью явно описать все домены объектов, используемые предикатами, то в разделах ***domains*** и ***predicates*** в этом случае должны были бы появиться следующие предложения:

```
domains
        name, fruit = symbol
predicates
        likes(name,fruit)
```

С описаниями подобного рода работать достаточно легко, так как имя домена ***name*** сразу же наводит на мысль об именах людей, а ***fruit*** - о названиях фруктов. Правда, эти описания не возбраняют написать и такое утверждение, как ***likes***(***mary***,rain), так как предикат допускает использование любого объекта, коль скоро он представляет собой символическое имя, хотя бы утверждение и выглядело странно***. rain*** (дождь) трудно зачислить в раз-

ряд фруктов. Лучше поэтому дать соответствующему домену имя ***thing*** или item. Тогда ***peaches*** и ***rain*** не противоречили бы друг другу.

Предположим теперь, что Вы хотите создать картотеку своих книг. Тогда предикат ***book*** логично было бы использовать в виде

book("***Tom Sawyer","Mark Twain",1855).***

book("***Man and Superman","Bernard Shaw",1905).***

Отметим, что первые два объекта предиката принадлежат к типу ***symbol***, тогда как третий - к типу ***integer***. Сам предикат можно было бы описать как

book(***symbol***,***symbol***,***integer)***

или, что то же самое,

```
domains
        title, author = symbol
        year          = integer
predicates
        book(title, author , year)
```

Чуть позднее в настоящей главе мы рассмотрим более подробно преимущества записи, использующей явное определение доменов.

3.2.2 Правила образования имен в Турбо-Прологе

Любое имя, используемое в Турбо-Прологе, должно состоять не более чем из 250 символов, первый из которых при этом должен обязательно быть строчной буквой латинского алфавита (от **a** до z). Пробелы в записи имени недопустимы, однако можно использовать подчерк (_) в качестве разделителя компонент так, как это сделано ниже:

```
employee_name
color_of_box
wild_animal_kingdom
        beginning_of_year_to_date_activities_report
```

Большинство программистов, однако, предпочитают пользоваться более краткими именами. Имена, приведенные в качестве примера очень удобны для понимания программы, однако у них есть весьма существенный недостаток - они длинны. В Турбо-Прологе предусмотрена возможность задавать имена, состоящие все-го из одной буквы:

```
domains
        a, b = symbol
predicates
        s(a,b)
clauses
        s(brave,daring)
```

3.3 Предикаты и утверждения

В этой части главы Вы начнете знакомиться с основами программирования на Турбо-Прологе. Представленные здесь программы иллюстрируют основные особенности языка.

Программа "Конструктор слов" (листинг 3.1) является примером законченной программы с использованием предикатов и утверждений. Целью программы является поиск и печать синонима некоторого слова. Синонимом слова ***brave*** является ***daring.*** Следующее утверждение указывает на синонимичность этих слов:

***synonym*(*brave*, *daring*). /* *daring* - синоним *brave* */**

Термом предиката здесь является ***synonym***, а объекты это ***brave*** и ***daring***. Описание предиката для этого утверждения будет выглядеть так:

synonym(word,syn)

где ***word*** и ***syn*** - объекты описываемого предиката.

Листинг 3.1

```
/* Программа:  Конструктор слов  Файл: PROG0301.PRO */
/* Назначение: Демонстрация ассоциаций слов при     */
      /*           помощи небольшого словаря.          */
      domains
            word, syn ,ant = symbol

      predicates
            synonym(word,syn)
            antonym(word,ant)

      goal
            synonym(brave,X),
            write("A synonym for 'brave' is "),
            nl,
            write("'",X,"'."),
            nl.

      clauses
            synonym(brave,daring).
            synonym(honest,truthful).
            synonym(modern,new).
            synonym(rare,uncommon).
            antonym(brave,cowardly).
            antonym(honest,dishonest).
            antonym(modern,ancient).
            antonym(rare,common).
/*****          конец программы          *****/
```

Описание их доменов - это

word, syn = symbol

Данное описание показывает, что оба объекта предиката ***synonym*** представляют собой символические имена. Предикат ***synonyn*** используется в четырех утверждениях:

synonym*(*brave,daring).
synonym(honest,truthful).
synonym(modern,new).
synonym*(*rare,uncommon).

В любом из этих утверждений на первой позиции стоит некоторое слово, а на второй - его синоним.

3.3.1 Использование внешних и внутренних целей

Не каждая из программ Турбо-Пролога содержит внутри себя описание своей цели, часто цель задается в процессе работы программы, т.е. является внешней. Программы Турбо-Пролога внешней целью называются интерактивными. Смысл применения внешних целей - дать пользователю полную свободу использования имеющихся данных; программа в этом случае играет роль "нейтральной" базы данных.

*** Внутренние цели**

Целью программы "Конструктор слов" является, как уже было сказано, поиск и печать синонима к выбранному слову. Цель поиска здесь задана в самой программе, следовательно она является внутренней. Само предложение, определяющее цель, состоит из пяти подцелей, разделенных запятыми. Первая из них - это

***synonym*(*brave*, *X*)**

X здесь является свободной переменной, ее значение не конкретизировано. (Запомните, что в Турбо-Прологе имя переменной обязательно начинается с большой буквы.) Говоря обычным языком, в данном предложении сформулирована такая цель: "Найти утверждение, использующее предикат ***synonym***, такое, что первым объектом в нем является ***brave***, и связать переменную ***X*** с его вторым объектом".

После запуска программы, Турбо-Пролог будет просматривать утверждения, содержащие ***synonym***. Если таковое с объектом ***brave*** будет обнаружено, то ***X*** примет значение второго объекта. В нашем случае им будет ***daring***.

Второй подцелью является печать следующей строки символов на экране:

A ***synonym for 'brave' is***

Эта подцель образована при помощи предиката ***write***, одного из многих "встроенных" предикатов Турбо-Пролога. Подобные предикаты не требуют специального описания в программе, их можно использовать сразу.

Встроенный предикат ***write*** в данной программе встречается в виде

write("A ***synonym for 'brave' is*** ")

Двойные кавычки при этом применяются для ограничения символьной строки ***A synonym for 'brave' is*** ; подобным образом должны выделяться все символьные строки. Предикат ***write*** может также содержать имена переменных, в этом случае кавычки не требуются.

Простейшим примером может служить

***write*(*X*),**

где ***X*** - это имя переменной. Если ***X*** принял значение ***daring,*** то ***write*** это ***daring*** и напечатает.

В обоих случаях как символьная строка, так и переменная являются аргументами предиката ***write***. Так же как и у других предикатов аргументы ***write*** разделяются запятыми. Аргументы можно произвольно смешивать при условии соблюдения описываемых соглашений. Компилятор Турбо-Пролога поправит Вас, если Вы пропустите кавычки, или сделаете какую-либо другую ошибку того же порядка.

Пример смешанной записи аргументов:

write("Today is the ",N,"th day of ",M,", a ",D,".").

Этот предикат напечатает предложение

Today is the 19th day of August, a Tuesday.

(Сегодня 19 августа, вторник)

если значениями переменных ***N, M*** и ***D*** будут соответственно 19, ***August*** и ***Tuesday.***

Третья подцель задается еще одним встроенным предикатом: ***nl***. Предикат ***nl*** сдвигает курсор в начало следующей строки. Четвертая подцель - печать всех трех объектов, заключенных в круглые скобки. Первый из них представляет собой обычную кавычку; второй, обозначенный как ***X,*** - это ***daring;*** третий состоит из кавычки и точки. Пятая подцель, еще один предикат ***nl***, сдвигает курсор к началу строки, следующей за строкой с ***daring***. Отметим, что предложение, описывающее цель, должно оканчиваться точкой; если точка будет опущена, компилятор выдаст сообщение об ошибке, и трансляция программы будет прекращена.

В результате выполнения всей совокупности целевых утверждений на экране возникнет картинка.

*** Внешние цели**

Если Вы запустите на счет программу, в которой будет отсутствовать описание цели, то Турбо-Пролог попросит Вас ввести цель с экрана. Если, например, из программы "Конструктор слов" удалить раздел goal и запустить ее на счет, то на экране в окне ***Dialog*** возникнет приглашение ***Goal***: . Предположим теперь, что Вы хотите задать вопрос: "Какое слово является синонимом слова ***modern*** ?" Так же, как и при записи внутренней цели, переменная ***Q*** будет использоваться для представления значения, которое Турбо-Пролог поставит ей в соответствие, используя утверждения базы данных. С ее помощью вопрос формулируется в виде

***synonym*(*modern,Q*)**.

Результатом ввода этого предложения явится картинка экрана. После удовлетворения внешней цели выполнение программы, однако, не завершается. Турбо-Пролог просит ввести следующую внешнюю цель. Таким образом, можно задать столько целей, сколько это представляется необходимым; чтобы остановить этот процесс нужно нажать клавишу ***Esc*** при выдаче очередного приглашения.

В формулировке запроса к программе можно использовать не только одну переменную, но и несколько. Если, к примеру, Вы введете две переменные, то программа выдаст все возможные комбинации значений, удовлетворяющие данные переменные.

Применение внешних целей бывает полезно при записи коротких целевых формулировок, а также для получения всего набора допустимых значений. Другое преимущество этого средства программирования заключается в том, что оно позволяет адресовать базе данных совершенно произвольные вопросы.

* Упражнение

3.1. Придумайте и напишите программу, использующую предикат

capital(state,city) /* столица (государство, город) */

Введите в базу данных утверждения, касающиеся Вашей страны, а также трех ее соседей. Ваша программа будет интерактивной, поэтому в нее не следует включать раздел ***goal.*** Еще раз позволим себе напомнить, что все имена предикатов должны начинаться с маленьких букв, также как и имена объектов, если Вы, конечно, не заключаете их в кавычки.

Запустите Вашу программу и введите цели, реализующие следующие вопросы: Какой город является столицей Вашей страны ?
Столицей какой страны является Москва ?

3.3.2 Предикаты и утверждения разных арностей

Как уже говорилось в предыдущей главе, при трансляции утверждений Турбо-Пролог контролирует правильность написания имени предиката, количество объектов и типы их доменов. Термин арность обозначает число объектов утверждения. Так утверждение ***likes***(***mary***, ***apples***) имеет арность 2.

Большинство программ, написанных на Турбо-Прологе, содержит несколько различных утверждений. Примером этих "несколько" может служить наличие в программе утверждений ***likes***(***mary***,***apples***), ***runs***(***john***,6.3) и ***drinks***(***beth***,***tea,coke).*** Если в вопросе содержится утверждение ***watches***(***john***, ***cats***), то Турбо-Пролог пытается найти соответствие с предикатом ***watches***, все остальные предикаты при этом игнорируются. Процесс сопоставления начинается с поиска утверждений с предикатом ***watches***. Затем ищутся утверждения, которые имеют такое же общее число объектов. В утверждении с ***watches*** объектов два: ***john*** и ***cats***. Количество объектов в предикате вопроса и утверждениях программы должно совпасть, иначе процесс сопоставления прекращается.

Если утверждения с таким же, как в вопросе, предикатом и таким же числом объектов найдены, то Турбо-Пролог проверяет соответствие типов объектов. В случае с ***watches***(***john***,***cats***) как ***john***, так и ***cats*** принадлежат типу ***symbol***. Если в типах объектов обнаруживается несоответствие, то считается, что попытка сопоставления окончилась неуспешно.

Рассмотрим теперь такие предикаты разных арностей:

go_home

female(person)
father(person,person)
owns(person,book,book) europe(country,country,coutry,country)

Объектами предикатов являются ***person, book*** и ***country***. Следующие утверждения используют эти предикаты; арность каждого из них указана в крайней правой колонке.

Предикат	**Утверждения**	**Арность**
go_home	go_home	0
female(person)	female(betty) female(katty)	1
father(person,person)	father(***john***,kathy) father(***john***,tom)	2
owns(person,***book***,***book***)	owns(sam,"Hobbit","Lord of the Rings")	3
europe(country,country, country,country)	europe("France","Germany", "Spain","Italy")	4

Первый из предикатов, ***go_home***, не имеет объектов, его арность равна нулю. Предикаты этого типа часто используются для построения правил, как, например,

go_home if condition(sickness)
go_home if (condition(sickness) and transportation(bus))

Предикаты арности 1 полезны при сортировке объектов программы по доменам. В приведенном выше примере предикат ***female*** указывает, что имя ***betty*** относится к домену женских имен.

Предикаты арности 2 используются для установления отношения между двумя объектами. Так, предикат

father(person,person)

и соответствующее ему утверждение

father(john,kathy)

будут означать, что ***john*** является отцом ***kathy***. Заметьте, что это же утверждение можно было бы записать и так:

father(kathy,john)

Тогда бы оно "переводилось" как "Отец kathy есть ***john***". Таким образом в данном примере порядок следования объектов не важен.

Предикаты арностей выше 2 пригодны для установления связи нескольких объектов по какому-либо признаку. В утверждении

europe("France","Germany","Spain","Italy")

все используемые значения***: France, Germany, Spain*** и ***Italy*** принадлежат домену ***country***. Общим для них является то, что все они обозначают европейские страны.

Программа "Словарь" (листинг 3.2) представляет собой некоторое усложнение программы "Конструктор слов". В ней предикат ***synonym*** содержит уже четыре объекта: само слово и три его синонима. Дополнительно введен предикат ***antonym*** с таким же количеством объектов. Эта программа являет собой несколько более реалистичный по сравнению с "Кон-

структором" пример электронного словаря.

Листинг 3.2

```
/* Программа: Словарь    Файл: PROG0302.PRO        */
/* Назначение: Демонстрация ассоциаций слов при     */
/*           помощи небольшого словаря.          */
        domains
                word,
                syn1, syn2, syn3,
                ant1, ant2, ant3 = symbol

        predicates
                synonym(word,syn1,syn2,syn3)
                antonym(word,ant1,ant2,ant3)

        goal
                synonym(brave,S1,S2,S3) and
                write("The synonyms for 'brave' are ")
                and nl and write(S1,",",S2,",",S3,".") and nl and
                antonym(rare,A1,A2,A3) and
                write("The antonyms for 'rare' are") and nl and
                write(A1,",",A2,",",A3,".") and nl.

        clauses
synonym(brave,daring,defiant,courages).
synnym(honest,truthful,open,sincere).
synonym(modern,new,novel,recent).
synnym(rare,uncommon,scarce,infrequent).

antonym(brave,cowardly,fearful,timid).
antoym(honest,dishonest,crooked,deceithful).
antoym(modern,ancient,old,obsolete). anto-
nym(rare,common,ordinary,ubiqutious).
/*****          конец программы          *****/
```

Внутренняя цель программы "Словарь" скомпонована из двух подцелей. Задачей первой из них является выдача трех синонимов слова ***brave***, задачей второй - выдача антонимов слова ***rare***.

Отметим, что в записи целевого утверждения присутствует ключевое слово ***and***, используемое в качестве разделителя. Здесь оно играет ту же роль, что и запятая в конструкции цели предыдущей программы.

* Упражнение

3.2. Модифицируйте программу из упражнения 3.1 так, чтобы она использовала предикат

cities_of_state(state,city1,city2,city3,city4)

/* города страны(страна,город1,город2,город3,город4) */

Напишите утверждения для пяти стран и их городов. Запустите программу и задайте такую внешнюю цель, чтобы программа напечатала четыре города одного из государств.

Программа "Президенты" (листинг 3.3) демонстрирует использование различных типов объектов. Утверждения данной программы содержат сведения о шести президентах США. Предикат

president(*name*,party,state,birth_year,year_in,year_out)

/* президент(имя, партия, штат, год рождения, начальный год пребывания у власти, конечный год пребывания у власти) */

имеет объекты типа символьной строки и типа целого числа, как это видно из раздела программы ***domains***.

Листинг 3.3

```
/* Программы:  Президенты                                    */
/* Назначение: Демонстрация отношений (предикатов)           */
      /*       и получение информации из БД.                 */
      domains
            name,party,state = symbol
            birth_year,year_in,year_out = integer

      predicates
president(name,party,state, birth_year,year_in,year_out)

      goal
president(X,democrat,S,Yb,Yi,Yo),nl, write(X," - democrat"),nl, write("State
",S),nl,
write("Birth year - ",Yi),nl, write("Year-in - ",Yi),nl, write("Year-out -
",Yo),nl,nl.

      clauses
president(eisenhower, republican, texas, 1890,1953,1961).
president(kennedy, democrat, massachusetts, 1917,1961,1963).
president(johnson,democrat,texas, 1908,1963,1969).
president(nixon,republican,california, 1913,1969,1974).
president(ford,republican,nebraska, 1913,1974,1977).
president(carter,democrat,georgia, 1924,1977,1981).

/*****          конец программы          *****/
```

Последние три объекта предиката ***president*** - целые числа, их доменами являются соответственно ***birth_year, year_in*** и ***year_out***. Объявление доменов в разделе программы ***domains*** выглядит достаточно просто:

birth_year,year_in,year_out = integer

Запятая в этом объявлении служит в качестве разделителя при перечислении имен доменов одного типа.

Цель программы на естественном языке можно сформулировать следующим образом: "Выдать место, год рождения, начальный и конечный год пребывания у власти всех президентов-демократов." Первая подцель содержит свободные переменные ***X, S, Yb, Yi*** и ***Yo***, и значение объекта ***democrat***. (Заметим, что в данном примере имена переменных являются ключом для понимания смысла соответствующих объектов, так, например, ***Yb*** - это "***year of birth***" - год рождения.). Когда программа запускается на счет, свободным переменным присваиваются соответствующие значения из второго утверждения. Следующие пять подцелей включают в себя предикаты ***write*** для печати этих значений в разных строках экрана. Данную программу можно также использовать и при отсутствии внутренней цели, то есть когда опущен раздел ***goal.***

* Упражнение

3.3. Введите программу "Президенты". Модифицируйте программу таким образом, чтобы цель стала внешней, запустите программу на счет и задайте вопрос:

president(X,republican,S,Yb,Yi,Yo).

На выходе Вы должны получить информацию о всех оставшихся президентах (республиканцах).

3.3. Введите программу "Президенты. Модифицируйте программу таким образом, чтобы цель стала внешней, запустите программу на счет и задайте вопрос:

president(X,republican,S,Yb,Yi,Yo).

На выходе Вы должны получить информацию о всех оставшихся президентах (республиканцах).

3.3.3 Использование правил в запросах

В предыдущих примерах программ вы использовали целевые утверждения, содержащие утверждения, подобные тем, что встречались в программе. Такие целевые утверждения обеспечивают одну из возможностей осуществления запросов к программе.

Запросы строятся из предикатов, содержащих условия, которые ограничивают пути поиска желаемых результатов, причем, в случае, когда какой-либо запрос нужно повторить несколько раз, разумно предусмотреть возможность не задавать всякий раз одни и те же условия, что утомительно. Полезно также для получения ответов из базы данных, не использовать фактов из базы данных.

В Турбо-Прологе эта задача решается конструированием правил, не содержащих в себе данных, т. е. правил нулевой арности. Задача, таким образом, сводится к написанию сокращенного варианта запроса. Пояснить только что сказанное можно на следующем примере. Представим себе некую гипотетическую семью: Фрэнк и Мэри являются мужем и женой. Их сына зовут Сэмом, а дочку - Дебби. Ниже приведен небольшой диалог, касающийся этой семьи.

Вопрос: Кем приходятся друг другу Дебби и Сэм ?

Ответ : Дебби - сестра Сэма.

Вопрос: Из чего Вы это заключили ?

Ответ : У Дебби и Сэма одни и те же родители, Дебби - девочка. Таким образом, Дебби - сестра Сэма.

Второй из вопросов является разговорной формулировкой правила, которое будет использоваться для ответа на запрос. Это правило можно перефразировать таким образом:

Дебби - сестра Сэма, если
Дебби - существо женского пола,
и родители Дебби есть родители Сэма.

Фраза включает в себя условие "если" (***if)***, который логически связывает оба утверждения. Утверждение, предшествующее "если" называется заключением или логическим следствием, а утверждение, следующее за "если" - допущением или предпосылкой.

Предикатные выражения являются теми блоками, из которых в Прологе строятся правила. Факты, описывающие отношения между Фрэнком, Мэри, Сэмом и Дебби можно сформулировать при помощи таких утверждений Турбо-Пролога:

```
male("Frank").    /* Фрэнк - мужского пола */
male("Sam").      /* Сэм - мужского пола   */
female("Mary").   /* Мэри - женского пола */
female("Debbie"). /* Дебби - женского пола */
parents("Sam","Frank","Mary"). /* Родители Сэма есть
                                  Фрэнк и Мери */
parents("Debbie","Frank","Mary").
    /* Родители Дебби есть Фрэнк и Мери */
```

Имея в наличии эти утверждения, необходимо лишь ввести правило, задающее отношение брат-сестра:

```
sister(Sister,Brother) if female(Sister),
                          parents(Sister,Father,Mother),
                          parents(Brother,Father,Mother).
```

Отметим, что в формулировке правила нет никаких данных о конкретной семье; объекты ***Sister, Brother, Father*** и ***Mother*** являются переменными.

Двойное использование предиката ***parents*** устанавливает логическую связь между переменными **Sister** и **Brother**. Наличие предиката ***female*** позволяет выбрать ребенка женского пола. Три предиката предпосылки правила вполне достаточны для получения нужного заключения. Программа

"Родственники" (листинг 3.4) является конечной программой на Турбо-Прологе, демонстрирующей использование перечисленных фактов и правила ***sister***.

Листинг 3.4

```
/* Программа:   Родственники      Файл: PROG0304.PRO */
/* Назначение:  Демонстрация конструкции правила.    */
            domains
                    person = symbol
            predicates
                    male(person)
                    female(person)
                    parents(person,person,person)
                    sister(person,person)
                    who_is_the_sister

goal who_is_the_sister
clauses
                     /* факты */
        male("Frank"). male("Sam").
        female("Mary"). female("Debbie").
        parents("Sam","Frank","Mary").
        parents("Debbie","Frank","Mary").
                    /* правила */
        who_is_the_sister if
                sister(Sister,Brother),
                write(Sister,
                " is the sister of ",
                Brother, "."),
                nl.
        sister(Sister,Brother) if
                female(Sister),
                parents(Sister,Father,Mother),
                parents(Brother,Father,Mother).
                /*****          конец программы        *****/
```

Программа "Родственники" содержит еще одно правило: предикат ***who_is_the_sister***. ***who_is_the_sister*** является целью программы, ее единственным целевым утверждением. Это правило определяется в разделе утверждений программы ***clauses***. Ввиду того, что цель задается в виде правила, точка входа раздела ***goal*** являет собой единственное целевое утверждение без подцелей. Тело правила состоит из двух частей. Первая часть это правило ***sister***. В качестве второй части - предпосылки - используется предикат ***write,*** который выводит полученные правилом ***sister*** результаты. При разра-

ботке программы такой способ записи цели более предпочтителен, так как он упрощает эту самую запись. Особенно полезным он бывает тогда, когда программа имеет много разных подцелей, включающих в себя достаточно сложные операции. С точки зрения человека, лишь использующего эту программу, такой способ также более предпочтителен, так как упрощает процедуру запросов.

Правила в Турбо-Прологе записываются в разделе утверждений. Программа может содержать достаточно много правил, вырабатывающих различные заключения. Эффект введения правил точно такой же, как если бы программа содержала большое число утверждений-фактов. В программе "Родственники", например, можно в раздел ***clauses*** ввести еще ряд правил в дополнение к уже определенным. Предположим, что Вы хотите узнать имя сына. Этот запрос оформляется при помощи правила:

```
who_is_the_son if
        son(parents,_,_),
        male(Son),
        write("The son is ",Son,"."),
        nl.
```

Это правило можно поместить в раздел **clauses** и заменить этим правилом целевое утверждение. Когда при запуске программы новое правило будет испытано, на экране появится надпись "***The son is Sam".***

Подобным образом можно ввести еще достаточное количество правил. Правила можно будет активизировать выборочно путем использования их в целевой конструкции. Возможность Турбо-Пролога выбирать нужные правила делает программу более гибким и мощным инструментом. Важным непосредственным приложением данного средства программирования является возможность задания запросов в форме правил, а также возможность "запасать" эти правила для использования при дальнейшей работе с базой данных. Примеры этого приложения будут приведены в главах 9 и 10.

* Упражнение

3.4. Введите программу "Родственники" и запустите ее на счет, чтобы убедиться в правильности набивки. Модифицируйте программу, введя правило определения имени брата. Цель изменений состоит в получении сообщения о том, что Сэм является братом Дебби.

3.3.4 Простые базы данных

Программа "Подбор партнера" (листинг 3.5) демонстрирует использование правил для получения информации в результате поиска соответствия по модели среди совокупности фактов. Эта программа является упрощенным прототипом программы для службы знакомств. Она содержит сведения о 7 мужчинах по параметрам: рост, цвет волос, класс машины. Единственным доменом программы является домен ***man***.

Листинг 3.5

```
/* Программа:   Подбор партнера                                    */
/* Назначение:  Демонстрация конструкции правила.    */
        domains
                man = symbol

        predicates
                choice(man)
                short_height(man)       /* мужчина низкого роста */
                medium_height(man)  /* мужчина среднего роста */
                tall_height(man)  /* мужчина высокого роста */
                black_hair(man)  /* мужчина - брюнет */
                brown_hair(man)         /* мужчина - шатен */
                blond_hair(man) /* мужчина - блондин */
                old_car(man)      /* владелец старого автомобиля */
                new_car(man)      /* владелец нового автомобиля */
                sports_car(man)    /* владелец спортивного автомобиля */
                kathy_choice(man)
                who_is_the_choice

        goal
                who_is_the_choice.

        clauses
                /* факты */
                choice(bill).
                choice(jim).
                choice(mark).
                choice(robert).
                choice(willy).
                choice(tom).
                choice(frank).
                short_height(bill).
                short_height(tom).
                medium_height(bill).
                medium_height(tom).
                tall_height(jim).
                tall_height(robert).
                tall_height(frank).
                black_hair(bill).
                black_hair(willy).
                brown_hair(jim).
                brown_hair(tom).
                blond_hair(mark).
                blond_hair(robert).
```

```
blond_hair(frank).
new_car(bill).
new_car(willy).
new_car(frank).
old_car(mark).
old_car(tom).
sports_car(jim).
sports_car(robert).
/* правила */

who_is_the_choice :-
                    kathy_choice(Choice),
                    write("Kathy's choice is ",Choice,"."),
                    nl.
kathy_choice(Choice) :-
                    choice(Choice),
                    tall_height(Choice),
                    blond_hair(Choice),
                    sports_car(Choice).
/*****       конец программы          *****/
```

В разделе ***clauses*** 7 утверждений предиката ***choice*** содержат имена ***bill, jim, mark, robert, willy, tom и frank***. Утверждения с предикатами ***short_hair, medium_height, tall_height, black_hair, brown_hair, blond_hair, new_car, old_car*** и ***sports_car*** также содержат эти мужские имена. Целью программы является отыскание мужчины, соответствующего вкусам некой ***kathy***: высокого роста, блондина, обладателя спортивного автомобиля.

В случае обнаружения мужчины, обладающего перечисленными свойствами, программа должна напечатать его имя. Правилом для поиска служит

```
kathy_choice(Choice) :-
                    choice(Choice),
                    tall_height(Choice),
                    blond_hair(Choice),
                    sports_car(Choice).
```

Назначение правила заключается в том, чтобы найти среди утверждений ***choice*** хотя один объект, который одновременно является объектом еще трех атрибутных утверждений: ***tall_height, blond_hair, sports_car***.

Вначале данное правило присваивает переменной ***Choice*** значение одного из объектов утверждения ***choice***. Как Вы можете вспомнить из обсуждения утверждений ***likes***, Турбо-Пролог просматривает утверждения с нужным предикатом, начиная с первого; следовательно, при первой попытке переменная ***Choice*** примет значение ***bill***. Затем правило попытается найти соответствие этого значения одному из объектов утверждений первого из

атрибутных предикатов, ***tall_height***. Так как такого соответствия нет, то Турбо-Пролог откатывается к следующему утверждению ***choice*** и делает еще одну попытку. Теперь значением **Choice** становится ***jim. jim*** удовлетворяет запросу ***kathy*** т. к. обладает высоким ростом, поэтому Турбо-Пролог пытается найти соответствие этого значения утверждениям ***blond_hair***. Ввиду того, что эта попытка оканчивается неуспехом, весь процесс повторяется с третьим утверждением ***choice***. Повторы будут происходить до тех пор, пока не будет найдено значение ***Choice***, удовлетворяющее всем трем подцелям

tall_height(Choice),
blond_hair(Choice),
sports_car(Choice).

Имя избранника Вы сможете назвать, если просмотрите листинг программы.

* Упражнение

3.5. Модифицируйте программу "Подбор партнера", написав правило для отыскания имени высокого мужчины, блондина, владельца нового автомобиля.

3.3.5 Отрицание

Помимо принадлежности одному и тому же домену, некоторые объекты могут иметь еще некоторое число общих атрибутов. Например, определенные страны Европы имеют общие между собой границы, в то время как другие их не имеют. Предикатом для представления этого отношения служит

border(country,country)

Тот факт, что "Германия и Франция имеют общую границу", можно представить в виде утверждения

border("France", "Germany").

Франция с Германией имеют общую границу, так же как и Франция с Испанией, и Франция с Италией.

Шесть утверждений задают все возможные пары четырех выбранных европейских стран:

euro_pair("France","Germany"). euro_pair("France","Spain").
euro_pair("France","Italy"). euro_pair("Germany","Spain").
euro_pair("Germany","Italy"). euro_pair("Spain","Italy").

Утверждения для стран с общей границей выглядят так:

border("France","Germany").
border("France","Spain").
border("France","Italy").

Предположим теперь, что вы хотите определить, какие из стран не имеют общей границы. Вместо того чтобы выдавать на экран все пары стран с общей границей, а потом визуально искать все пары, не попавшие в этот список, лучше воспользоваться более простым и эффективным средством -

отрицанием. Отрицание предиката ***border*** задается при помощи предиката ***not:***

not(border(Country1,Country2)).

Этот предикат выдает все пары не граничащих друг с другом стран. Программа "Пары стран Европы" (листинг 3.6) - это законченная программа на Турбо-Прологе для решения поставленной задачи.

Листинг 3.6

```
/* Программа:  Пары стран Европы   Файл: PROG0306.PRO   */ /* Назначение: Демонстрация работы предиката отрицания. */ /* Указание:   Цель - внутренняя.                */
        domains
                country = symbol

        predicates
                euro_pair(country,country)
                border(country,country)
                find_non_border_pair

        goal
                find_non_border_pair.

        clauses
                /* факты */
euro_pair("France","Germany"). euro_pair("France","Spain").
euro_pair("France","Italy"). euro_pair("Germany","Spain").
euro_pair("Germany","Italy"). euro_pair("Spain","Italy").
                border("France","Germany").
                border("France","Spain").
                border("France","Italy").

                /* правила */
                find_non_border_pair :-
                                euro_pair(X,Y),
                                not(border(X,Y)),
                                write(X," - ",Y),nl.
        /*****        конец программы        *****/
```

При программировании на Турбо-Прологе отрицания иногда бывают удобны для логического вывода неких фактов из уже имеющихся в базе данных. И вообще, использование предиката ***not*** в конструкции правила зачастую дает возможность ввести в программу элементы логики. Например, вопрос с отрицанием можно использовать для определения того, являются ли

соседями две выбранные страны. Предположим, вы хотите знать, граничат ли между собой Германия и Испания. Запрос формулируется так:

not (border("Germany", "Spain")).

Ответом на запрос явится ***True,*** так как согласно имеющимся в базе данных фактам, эти страны общей границы не имеют. Рассмотрим теперь другой запрос:

not (border("France", "Italy")).

Ответом на него будет ***False***, так как Франция и Италия соседи.

* Упражнение

3.6. Переделайте программу для работы с внешними целями. В ответ на приглашение ***Goal:*** введите ***not(border("Spain","Italy")).***
Каким будет ответ ?

3.3.6 Использование составных объектов

Объекты утверждений представляют собой данные, тип же простых объектов ограничен 6 типами доменов. Рассмотрим такой пример утверждения:

owner("Mary","Garfield"). /* У Мери есть Гарфильд */

Первый объект ***Mary*** имеет простую структуру; он представляет сам себя. То же можно сказать и про объект **Garfield.** Любой объект, представляющий сам себя, называется простым объектом. Аналогично, структура, состоящая из простых объектов, называется простой структурой.

Утверждение ***owner*** отражает тот факт, что ***Mary*** обладает ***Garfield***, который может быть либо именем домашнего животного ***Mary***, либо названием книги. Для разделения этих случаев утверждение можно записать в форме, более определенно описывающей объект:

owner("Mary",pet("Garfield")). /* У Мери есть любимец - Гарфильд */

owner("Mary",book("Garfield")). /* У Мери есть книга - "Гарфильд" */

Объект, представляющий другой объект или совокупность объектов, называется составным объектом. Записанные же таким образом предикаты owner называются составными структурами, поскольку они скомпонованы из составных объектов. В данном примере ***pet*** представляет ***Garfield*** в первом утверждении, в то время как ***book*** - во втором, здесь ***pet*** и ***book*** являются составными объектами. Отметим, что объекты, которые они представляют, заключены в скобки.

Утверждение

likes("Tom",apples,orange,banana).

констатирует, что Том любит фрукты: яблоки, апельсины и бананы. Все эти три вида фруктов можно объединить в отдельной структуре:

fruits(apples,orange,banana).

В результате появляется составной объект, который поясняет отношение:

likes("Tom",fruits(apples,orange,banana)).

(Обратите внимание на двойные скобки в конце утверждения.)

Терм ***fruits*** в этом утверждении называется ***функтором***. Функтор является первым термом составного объекта. Функтор составного объекта есть на самом деле предикат, хотя он и вставлен внутрь другого предиката. Главным функтором здесь является предикат ***likes.***

Для облегчения написания утверждений и предикатов в данной форме Турбо-Пролог позволяет объявлять составные объекты в разделе ***domains.*** Для настоящего примера описаниями будут служить

domains
personal_liking = fruits(type1,type2,type3)
type1,type2,type3 = symbol

Имя домена ***personal_liking*** является именем составного объекта, образованного при помощи функтора ***fruits***. Имя ***fruits*** представляет одновременно составной объект и функтор. Если объекты структуры принадлежат к одному и тому же типу доменов, то этот объект называется однодоменной структурой. Структура с объектами ***apples, peaches*** и ***oranges*** (все типа ***symbol***), является однодоменной структурой. Если структура содержит объекты разных типов, она называется многодоменной структурой. Примером тому совокупность объектов ***apples, r*** и 16. Здесь ***apples*** имеет тип ***symbol, r*** - тип ***char***, 16 - тип ***integer***. Таким образом составной объект является определенной структурой доменов. Каждая структура предполагает особое представление фактов в базе данных. Структура обеспечивает средство сортировки объектов по категориям. Ссылки на доменную структуру осуществляются по имени функтора.

*** Функторы и доменные структуры**

Программа "Библиотека" (листинг 3.7) демонстрирует использование доменной структуры с именем ***personal_library***. Эта структура содержит сведения о книгах из личных собраний.

Листинг 3.7

```
/* Программа:  Библиотека          */
/* Назначение:  Демонстрация одноуровневого составного    */
/*                  объекта.                    */
domains
     personal_library = book (title, author, publisher, year)  /* персональная
                                        библиотека  =  книга(название,автор,
                                                издательство,год издания) */
      collector, title, author, publisher = symbol
      year = integer

predicates
      collection(collector, personal_library)
/* коллекция (имя коллекционера, библиотека) */

clauses
```

```
collection(kahn, book("The Computer and the Brain",
        "von Neumann", "Yale University Press",1958)).
collection(kahn, book("Symbolic Logic", "Lewis Carroll",
                "Dower Publications",1958)).
collection(johnson, book("Database: A Primer", "C.J.Date",
                                        "Addison-Wesley",1983)).
collection(johnson, book("Problem-Solving Methods in AI",
        "Nils Nilsson", "McGraw Hill",1971)).
collection(smith,  book("Alice in Wonderland",
        "Lewis Carroll", "The New American Library",1960)).
collection(smith, book("Fables of Aesop", "Aesop-Calder",
                                    "Dover Publications",1967)).
/*****          конец программы          *****/
```

Функтор структуры ***personal_library*** имеет имя ***book***. Описание таково:

personal_library = book(title, author ,publisher, year)
collector, title, author, publisher = symbol
year = integer

Предикат, использующий эту структуру, определяется так:

collection (collector, personal_library)

Описание содержит два имени объектов. Первое имя относится к обычному объекту, второе - к структуре из нескольких объектов.

Использование доменной структуры упрощает структуру предиката. Если не использовать конструкцию доменной структуры, то программы требовала бы такого описания предиката ***collection:***

collection(collector, title, author, publisher, year)

В этом описании 4 последних объекта обозначают атрибуты книги. Правило, которое оперирует с персональными библиотеками рассматривало бы эти 4 последних объекта как независимые сущности, что сделало бы код программы более сложным.

Данная программа использует внешнюю цель. Для того, чтобы узнать, какие книги принадлежат Смиту, необходимо ввести такое целевое утверждение:

Collection (smith, Books).

Объект ***smith*** является частным значением из домена ***collector,*** а ***Books*** - свободной переменной. Цель заключается в отыскании всех книг, принадлежащих Смиту.

Предположим теперь, что Вы хотите знать имена владельцев и названия книг, напечатанных в 1967 году. Цель для поиска этой информации выглядит следующим образом:

collection(Collector,book(Title,_,_,1967)).

Здесь свободными переменными являются уже ***Collector*** и ***Title***. Подчеркивание (_) указывает на то, что Вас не интересуют объекты с родовыми

именами ***author*** и ***publisher***. (Напомним, что подчерк замещает собой анонимную переменную.)

Следующие два упражнения познакомят Вас с использованием целевых утверждений различных типов.

* Упражнения

3.7. Вы хотите задать вопрос: Как зовут коллекционера, которому принадлежит книга под названием "Database.A Primer." ? Этот запрос в Турбо-Прологе формулируется в виде

collection(Collector,book("Database:A Primer",_,_,_)).

Что получится ?

3.8. Вы хотите задать вопрос: Каковы названия книг, опубликованных после 1980 года ? Целевая конструкция для этого вопроса выглядит так:

collection(_,book(Title,_,_,Year)),Year > 1980.

Что получится ?

Программа "Библиотека - 2" (листинг 3.8) демонстрирует использование трехуровневой доменной структуры и четырехуровневой предикатной.

Листинг 3.8

```
/* Программа:   Библиотека -2      */
/* Назначение:  Демонстрация двухуровневого составного    */
/*              объекта.                                  */
domains
        personal_library = book(title,author,publication)
        publication = publication(publisher,year)
        collector,title,author,publisher = symbol
        year = integer

predicates
        collection(collector,personal_library)

clauses
        collection("Kahn",
                   book("The Computer and the Brain",
                   "von Neumann",
                   publication("Yale University Press", 1958))).
        collection("Kahn",
                book("Symbolic Logic",
                "Lewis Carroll",
                publication("Dower Publications", 1958))).
        collection("Johnson",
                book("Database: A Primer",
                "C.J.Date",
                publication("Addison-Wesley", 1983))).
```

```
        collection("Johnson",
                book("Problem-Solving Methods in AI", "Nils Nilsson",
                 publication("McGraw Hill", 1971))).
        collection(smith,
                book("Alice in Wonderland", "Lewis Carroll",
                publication("The New American Library", 1960))).
        collection(smith,
                book("Fables of Aesop",
                "Aesop-Calder",
                publication("Dover Publications", 1967))).
/*****                    конец программы              *****/
```

Программа "Библиотека - 2" использует те же данные, что и программа "Библиотека". Однако, здесь ***publisher*** и ***year*** являются объектами ***publication*** - функтора третьего уровня. Описание доменов выглядит так:

```
personal_library = book(title,author,publication)
publication = publication(publisher,year)
collector,title,author,publisher = symbol
year = integer
```

Описание предикатов и утверждений в точности совпадает с описаниями из предыдущей программы, т. е. все отличие заключается только в описании доменов. Функторы здесь - ***book*** и ***publication***, причем ***publication*** является компонентом функтора ***book***. Усилия и время, затраченные при разработке доменных и предикатных структур с лихвой вознаградят Вас удобством пользования созданной Вами базы данных. Чем больше функторов, тем более определенные запросы можно адресовать базе данных; а именно такие запросы, вне всякого сомнения, представляют наибольший интерес.

* Упражнения

3.9. Запустите на счет программу "Библиотека - 2" и введите внешнюю цель в виде

collection(smith,Books).

или в виде

collection(Collector,book(Title,_,publication(_,1967))).

Окажется ли среди перечисленных на экране книг хотя бы одна из собрания ***Kahn ?***

3.10. Измените описание структур программы "Библиотека -2". Поместите объекты ***title*** и ***author*** в подструктуру с именем ***volume,*** которая определяется как

volume = volume(author,title)

Не забудьте привести в соответствие синтаксис утверждений. Запустите эту модифицированную программу. Теперь Вы сможете обращаться к базе данных несколько иначе. Попробуйте ввести такую цель:

collection(_,Book(Volume,_)).

Будет ли выдан список всех авторов и книг ? Если нет, еще раз проверьте правильность написания утверждений.

3.3.7 Использование альтернативных доменов

Представление данных часто требует наличия большого числа структур. В Турбо-Прологе эти структуры должны быть описаны. Более того, возникают трудности с предикатами, работающими с объектами этих доменов. Для устранения данного недостатка Турбо-Пролог предлагает пользователю альтернативные описания доменов. Программа "Предметы" (листинг 3.9) как раз и использует эти альтернативные описания:

Листинг 3.9

```
/* Программа:   Предметы    */
/* Назначение:  Демонстрация использования конструкций */
/*          альтернативных доменов.          */
domains
        thing = misc_thing(whatever)      ;
        book(author, title)       ;
        record(artist, album, type)
        person, whatever,
        author, title,
        artist, album, type = symbol

predicates
        owns(person, thing)

clauses
    /* факты */
        /* Разнообразные вещи */
    owns("Bill", misc_thing("sail boat")).
    owns("Bill", misc_thing("sports car")).
    owns("Jack", misc_thing("Motor cycle")).
    owns("Jack",
        misc_thing("house trailer")). owns("Beth",
        misc_thing("Chevy wagon")).
    owns("Beth", misc_thing("Piano")).
    owns("Linda", misc_thing("motor boat")).
        /* книги */
    owns("Bill",
        book("J.R.R. Tolkein",
        "Return of the Ring")).
    owns("Bill",
        book("James A. Mishener", "Space")).
    owns("Jack", book("Manuel Puig",
```

```
        "Kiss of the Spider Woman")). owns("Beth",
        book("Frank Herbert", "Dune")).
    owns("Beth",
        book("Tom Clancy",
        "The Hunt for Red October")).
    owns("Linda", book("Garrison Keillor",
        "Lake Wobegon Days")).
        /* грампластинки */
    owns("Bill", record("Elton John",
        "Ice Fair", "popular")).
    owns("Bill",
        record("Michael Jackson - Lionel Richie",
        "We are the World",
        "popular")).
    owns("Jack",
        record("Bruce Springsteen",
        "Born to Run", "popular")).
    owns("Jack",
        record("Benny Goodman",
        "The King of Swing", "jazz")).
    owns("Beth",
        record("Madonna", "Madonna", "popular")).

/*****          конец программы          *****/
```

Приведенный ниже фрагмент показывает, как выглядела бы программа, если бы Турбо-Пролог не поддерживал конструкций альтернативных описаний доменов.

```
    domains
person,whatever,author,title = symbol
artist,album,type            = symbol
misc_thing      = misc_thing(whatever)
book_library    = book(author,title)
record_library = record(artist,album,type)

    predicates
personal_thing(person,misc_thing)
personal_books(person,book_library)
personal_records(person,record_library)

    clauses
personal_thing("Bill",misc_thing("sail boat")).
personal_books("Bill",book("J.R.R. Tolkein",
    "Return of the Ring")).
```

personal_records("Bill",record("Elton John",
"Ice of Fire","popular")).

Программа "Предметы" использует 3 доменные структуры. Первой из них является ***misc_thing***, единственный объект которой назван ***whatever***. Вторая структура имеет имя ***book***; ее объекты - это ***author*** и ***title***. Третьей структурой является ***record;*** она скомпонована из трех объектов: ***artist, album*** и ***type.***

Объекты всех трех структур относятся к типу ***symbol***. Все три объединены под именем ***thing***. Описанием этого домена является

thing = misc_thing(whatever) ;
book(author,title) ;
record(artist,album,type)

Для разделения альтернативных доменов здесь применена точка с запятой (;).

Для связи персоны и того, чем эта персона обладает, введен очень простой предикат:

person(person, things)

Использование альтернативных доменов позволяет писать в утверждениях предикат ***owns*** применительно к различным классам вещей. В отсутствие этой конструкции требовалось бы ввести уже три предиката, как это видно из приведенного фрагмента.

Попробуйте обратиться к программе "Предметы" с таким целевым предложением:

owns(P,misc_thing(T)).

Перевод этого запроса на естественный язык выглядит так: "Перечислите все возможные вещи, которыми кто-либо обладает." Интересно попробовать и запросы ***owns(_,book(A,T))*** и ***owns(P,record(_,A,_)).***

Заметим, что термы ***misc_thing***, ***book*** и ***record*** являются именами структур. Однако когда они появляются в предикатных выражениях, то одновременно играют и роль имен функторов.

Турбо-Пролог не делает различия между функтором и доменной структурой. Это средство введено в Турбо-Пролог преднамеренно, так как оно очень удобно именно в декларативных языках.

* Упражнения

3.11. Запустите программу "Предметы" и введите внешнюю цель

owns(_,book(_,T)).

Какой текст появится на экране ? Сформулируйте запрос на естественном языке.

3.12. Повторите запуск программы со следующей внешней целью:

owns(P,record(_,A,_)).

Что можно будет увидеть на экране ? Сформулируйте данный запрос на естественном языке.

Программа "Предметы - 2" (листинг 3.10) представляет собой видоизмененный вариант программы "Предметы".

Листинг 3.10

```
/* Программа:  Предметы - 2   Файл: PROG0310.PRO      */
/* Назначение: Демонстрация использования конструкций  */
/*                  альтернативных доменов.              */
domains
        thing = misc_thing(whatever)      ;
        book(author,title)        ;
        record(artist,album,type)
                person,
                whatever,
                author, title,
                artist,album,type = symbol

predicates
        owns(person,thing)
        show_misc_things
        show_books
        show_records

goal
        write("Here are the books:"), nl, nl, show_books.

clauses
                /* факты */
                /* Различные вещи */
        owns("Bill",
                misc_thing("sail boat")).
                owns("Bill",
                misc_thing("sports car")).
        owns("Jack",
                misc_thing("мotor cycle")).
        owns("Jack",
                misc_thing("house trailer")).
        owns("Beth",
                misc_thing("Chevy wagon")).
        owns("Beth",
                misc_thing("Piano")).
                owns("Linda",
                misc_thing("motor boat")).
                /* книги */
        owns("Bill",
                book("J.R.R. Tolkein",
```

```
        "Return of the Ring")).
owns("Bill",
        book("James A. Mishener",
        "Space")).
owns("Jack",
        book("Manuel Puig",
        "Kiss of the Spider Woman")).
owns("Beth",
        book("Frank Herbert",
        "Dune")).
owns("Beth",
        book("Tom Clancy",
        "The Hunt for Red October")).
owns("Linda",
        book("Garrison Keillor",
        "Lake Wobegon Days")).
        /* грампластинки */
owns("Bill",
        record("Elton John",
        "Ice Fair", "popular")).
owns("Bill",
        record("Michael Jackson - Lionel Richie",
        "We are the World",
        "popular")).
owns("Jack",
        record("Bruce Springsteen",
        "Born to Run", "popular")).
owns("Jack",
        record("Benny Goodman",
        "The King of Swing", "jazz")).
owns("Beth",
        record("Madonna", "Madonna", "popular")).
        /* правила */
show_misc_things :-
         owns(Owner, misc_thing(Whatever)), write(Owner,"
        ",Whatever), nl, fail.
show_misc_things.
show_books :-
        owns(_,book(_,Title)),
        write("  ",Title), nl,
        fail.
show_books.
show_records :-
        owns(Owner,record(_,Album,_)),
        write("  ",Owner," ",Album), nl,
```

```
        fail.
    show_records.
/*****        конец программы        *****/
```

В этой программе дополнительно присутствуют три правила. Каждое из них можно использовать в качестве компоненты внутренней цели. (В самой программе в целевом утверждении задействовано правило ***show_books***. По желанию это правило можно заменить другим.)

Первое из правил есть

```
show_misc_things :-
        owns(Owner, misc_thing(Whatever)),
        write(Owner,"   ",Whatever), nl,
        fail.
```

Это правило осуществляет запрос: "Выдать все возможные предметы и их владельцев".

Второе правило есть

```
show_books :-
        owns(_,book(_,Title)),
        write("  ",Title), nl,
        fail.
```

На естественный язык это утверждение можно перевести приблизительно так: "Выдать названия книг, содержащиеся в базе данных."

Третье правило - это

```
show_records :-
        owns(Owner,record(_,Album,_)),
        write("  ",Owner," ",Album), nl,
        fail.
```

Перевод этого правила: "Выдать имена всех коллекционеров пластинок и названия альбомов из их коллекций."

Применение альтернативных доменов делает программу более "управляемой", а программирование - более эффективным.

* Упражнение

3.13. Рассмотрим запрос: Перечислить названия всех популярных (***popular***) музыкальных записей и имена их исполнителей. Постройте правило Пролога для реализации этого запроса, включите его в программу и запустите ее на счет. Что выдаст программа ?

3.4 Арифметика в Турбо-Прологе

Турбо-Пролог располагает двумя числовыми типами доменов: целыми и действительными числами. Четыре основные арифметические операции - это сложение, вычитание, умножение и деление. Для их реализации в Турбо-Прологе используются предикаты. Программа "Числа" (листинг 3.11)

показывает, как можно при помощи предикатов реализовать эти операции.

Листинг 3.11

```
/* Программа:  Числа          */
/* Назначение: Демонстрация реализации арифметики. */
        predicates
                add(integer,integer).
                substruct(integer,integer).
                multiply(integer,integer).
                divide(integer,integer).
                fadd(real,real).
                fsubstruct(real,real).
                fmultiply(real,real).
                fdivide(real,real).

        goal
                write("       Results"), nl, nl,
                add(44, 23),
                substruct(44, 23),
                multiply(44, 23),
                divide(44, 23),
                fadd(12.65, 7.3),
                fsubstruct(12.65, 7.3),
                fmultiply(12.65, 7.3),
                fdivide(12.65,7.3), nl,
                write("          All done, bye!").

        clauses
                add(X,Y):-
                        Z = X + Y, write("Sum = ",Z), nl.
                substruct(X,Y):-
                        Z = X - Y, write("Diff = ", Z),  nl.
                multiply(X,Y):-
                        Z = X * Y, write("Pro = ", Z), nl.
                divide(X,Y):-
                        Z = X / Y, write("Quo = ", Z),  nl.
                fadd(P,Q):-
                        R = P + Q, write("Fsum = ",R),  nl.
                fsubstruct(P,Q):-
                        R = P - Q, write("Fdiff = ",R),  nl.
                fmultiply(P,Q):-
                        R = P * Q, write("Fpro = ",R), nl.
                fdivide(P,Q):-
                        R = P / Q, write("Fquo = ",R), nl.
/*****            конец программы          *****/
```

Правилами для реализации сложения, вычитания, умножения и деления целых чисел являются

```
add(X,Y):-
        Z = X + Y, write("Sum = ", Z), nl.
substruct(X,Y):-
        Z = X - Y, write("Diff = ", Z),  nl.
multiply(X,Y):-
        Z = X * Y, write("Pro = ", Z),  nl.
divide(X,Y):-
        Z = X / Y, write("Quo = ", Z), nl.
```

а четырьмя правилами для реализации сложения, вычитания, умножения и деления действительных чисел -

```
fadd(P,Q):-
        R = P + Q, write("Fsum = ",R), nl.
fsubstruct(P,Q):-
        R = P - Q, write("Fdiff = ",R), nl.
fmultiply(P,Q):-
        R = P * Q, write("Fpro = ",R), nl.
fdivide(P,Q):-
        R = P / Q, write("Fquo = ",R), nl.
```

Внутренняя цель составлена из последовательности утверждений, использующих эти правила. В ее формулировке присутствуют числовые значения, которые передаются в тела правил. Очень важно соблюсти соответствие типов данных и типов объектов предикатов. В результате счета программы на экране возникнет картинка, представленная на рис. 3.17.

Отметим, что деление целого числа на целое может дать десятичную дробь. В этом случае все знаки вплоть до десятого являются верными.

* Упражнения

3.14. Предположим, что Вы хотите сложить четыре десятичных числа. Предикатом для выполнения этой операции служит

sum(real,real,real,real,real)

Напишите правило для сложения четырех чисел. Включите правило и предикат в программу "Числа".

3.15. Запустите эту модифицированную программу и задайте такую внешнюю цель:

sum(3.9, 4.6, 2.6, 9.7, Z).

Каков будет результат ?

3.5 Заключение

В главе были рассмотрены основные принципы программирования на Турбо-Прологе. Вы познакомились с такими фундаментальными понятиями, как предикаты, утверждения и домены. Вы также узнали о структуре и организации программ на Турбо-Прологе. Попутно Вы научились применять

внутренние и внешние цели. Обсуждались также техника построения Турбо-Прологовских программ и написание простых запросов к базе данных. Материал пояснялся с помощью специально написанных программ, рассматривались способы модификации этих программ в соответствии с потребностями пользователя.

Были представлены способы формирования составных объектов с целью получения иерархических доменных структур. В довершение всего были рассмотрены правила работы с числовой информацией.

Глава 4. Повторение и рекурсия

4.1 Введение

Очень часто в программах необходимо выполнить одну и ту же задачу несколько раз. В программах на Турбо-Прологе повторяющиеся операции обычно выполняются при помощи правил, которые используют откат и рекурсию. В этой главе рассматриваются итеративные и рекурсивные правила, а так же общие способы их построения. Вводятся четыре важных метода: **метод отката после неудачи, метод отсечения и отката, правило повтора, определяемое пользователем, и обобщенное рекурсивное правило**. Простые программы, использующие названные методы построения правил, помогут лучше понять технику программирования, и вы легко сможете использовать ее в собственных программах.

Правила повтора и рекурсии должны содержать средства управления их выполнением с тем, чтобы их использование было удобным. Встроенные предикаты Турбо-Пролога ***fail*** и ***cut*** используются для управления откатами, а условия завершения используются для управления рекурсией. Далее рассматриваются все эти вопросы и специальные примеры, позволяющие лучше понять эти методы.

4.2 Программирование повторяющихся операций

Цели управляют программой на Турбо-Прологе, обеспечивая выполнение последовательности определенных задач. Вы уже знаете что, во-первых, цели могут содержать подцели и, во-вторых, цели (и подцели) могут содержать правила. Правила часто требуют, чтобы такие задачи, как поиск элементов в базе данных или вывод данных на экран выполнялись несколько раз.

Существуют два способа реализации правил, выполняющих одну и туже задачу многократно. Первый их них будем называть повторением, а второй - рекурсией. Правила Турбо-Пролога, выполняющие повторения, используют откат, а правила, выполняющие рекурсию, используют самовызов.

Вид правила, выполняющего повторение, следующий:

```
repetitive_rule :-                    /* правило повторения */
        <предикаты и правила>,
```

```
fail.                    /* неудача        */
```

Конструкция <предикаты и правила> в теле правила обозначает предикаты, содержащие несколько утверждений, а так же правила, определенные в программе. Встроенный предикат ***fail*** (неудача) вызывает откат, так что предикаты и правила выполняются еще раз.

Вид правила, выполняющего рекурсию, следующий:

```
recursive_rule :-          /* правило рекурсии */
        <предикаты и правила>,
        recursive_rule.
```

Заметьте, что последним правилом в теле данного правила является само правило ***recursive_rule***. Правила рекурсии содержат в теле правила сами себя.

Правила повтора и рекурсии могут обеспечивать одинаковый результат, хотя алгоритмы их выполнения не одинаковы. Каждый из них в конкретной ситуации имеет свои преимущества.

Рекурсия, например, может потребовать больше системных ресурсов. Так всякий раз при рекурсивном вызове новые копии используемых значений помещаются в стек. Стек представляет собой область памяти, используемую в основном для передачи значений между правилами. Значения сохраняются пока правило не завершиться либо успешно, либо неуспешно. В некоторых ситуациях такое использование стека может быть оправдано, если промежуточные значения должны храниться в определенном порядке для дальнейшего использования. В следующей главе вы увидите, какую особую роль играет стек при обработке списков.

Турбо-Пролог имеет средства для сокращения "потребления" стека, однако правила повтора, использующие откат, не увеличивают занятую часть стека. В данной главе не ставилась задача обоснования выбора того или иного метода, это будет сделано в следующих главах. Здесь же приведены программы, демонстрирующие как пользоваться этими двумя методами.

4.3 Повторение и откат

Обычно цель программы на Турбо-Прологе содержит одну или несколько подцелей, которые могут быть либо фактами, либо правилами. Факт вычисляется немедленно. Результат будет либо успешным, либо неуспешным в зависимости от того, сопоставим ли факт в программе с фактом в правиле. Правило, являясь связкой подправил, вычисляется путем вычисления подправил. Если подправило не может быть успешно вычислено, то Турбо-Пролог выполняет откат, что бы найти другие возможные пути вычисления этого подправила.

Проанализируем понятие отката на примере простой базы данных о спортивных увлечениях, содержащей следующие факты:

```
plays(tom,football)      /* Том играет в американский  */
        /* футбол   */
plays(john,soccer)       /* Джон играет в европейский  */
```

/* футбол */

plays(john,volleyball) **/* Джон играет в волейбол */**

plays(tom,basketball) /* Том играет в баскетбол */

plays(tom,*volleyball*) /* Том играет в волейбол */

plays(john,baseball) /* Джон играет в бейсбол */

Задача программы определить, в какую игру одновременно играют Джон и Том.

Утверждение цели следующее:

plays(john,G),plays(tom,G).

Заметьте, что эта цель есть связка двух подцелей. Каждая подцель содержит переменную ***G***. Задача заключается в нахождении значения для ***G,*** удовлетворяющего обеим подцелям.

Чтобы вычислить первую подцель ***plays(john,G),*** Турбо-Пролог ищет в базе данных сопоставимое утверждение. Первый объект утверждения ***plays(john, soccer)*** сопоставим с первым объектом первой подцели, так что подцель успешно вычисляется и переменная ***G*** получает значение ***soccer***. Существуют другие утверждения для ***plays***, которые могут быть использованы для вычисления этой же подцели. Поэтому Турбо-Пролог должен сохранить след этих утверждений на случай неуспешного вычисления второй подцели со значением ***G*** равным ***soccer***. Для этого внутренние унификационные подпрограммы устанавливают указатель отката на точку, с которой могут быть продолжены усилия по вычислению первой подцели.

Теперь Турбо-Пролог пытается вычислить вторую подцель. Так как ***G*** имеет значение ***soccer***, то эта подцель есть ***plays(tom, soccer).*** Турбо-Пролог просматривает базу данных в поиске утверждения, сопоставимого с этой подцелью. Сопоставимых утверждений нет, поэтому подцель неуспешна.

Так как вторая подцель вычислена неуспешно, то Турбо-Пролог вновь должен начать просмотр с первой подцели. После того, как попытка вычислить вторую подцель оказалась неуспешной, переменная ***G*** освобождается и Турбо-Пролог снова начинает поиск утверждения для вычисления подцели ***plays(john,G).***

Следующее сопоставляемое утверждение - это ***plays(john,volleyball)***. Переменная ***G*** получает значение ***volleyball***. Снова указатель отката помещается на следующее утверждение. Теперь Турбо-Пролог пытается вычислить вторую подцель ***plays(tom,volleyball)***. Во время этой попытки удается найти сопоставимое утверждение. Присвоение переменной ***G*** значения ***volleyball*** приводит к успеху. Обе подцели удовлетворены. Больше подцелей нет, поэтому вся исходная цель оказывается успешно вычисленной.

Если бы цель в этом примере была внутренней, то процесс вычисления остановился бы после первого ее успешного вычисления. Однако цель здесь внешняя, поэтому процесс повторяется до тех пор, пока не будут найдены все успешные способы вычисления цели. Но информация, содержащаяся в данных утверждениях, дает только одно допустимое значение для ***G***,

которое удовлетворяет обеим подцелям, т.е. результат вычисления подцели есть ***G=volleyball***.

Откат является автоматическим инициируемым системой процессом, если не используются средства управления им. Для управления процессом отката в Турбо-Прологе предусмотрены два встроенных предиката ***fail*** (неудача) и ***cut*** (отсечение). Использование этих предикатов рассматривается ниже в разделах, посвященных методу **Отката После Неудачи (ОПН)** и методу **Отсечения и Отката (ОО).**

Во время выполнения программы Турбо-Пролог создает необходимые внутренние структуры данных (такие как списки и деревья) для выполнения обработки программы. Эта обработка включает такие процессы как поиск, сопоставление с образцом, создание экземпляра, означивание и освобождение переменных, откаты. Вы должны вспомнить сказанное в главе 2 о том, что эти процессы выполняются внутренними подсистемами языка, которые называются внутренними унификационными подпрограммами. Во время выполнения программы эти подпрограммы всегда находятся в рабочем состоянии во время выполнения программы. Вы встретите много ссылок на них при объяснении работы элементов программ.

4.4 Методы повторения

Вспомните, что оператор внешней цели побуждает переменные получать все возможные значения одно вслед за другим. (Вы можете прочитать об этом в гл. 2 и 3). Если полученных значений не много (скажем меньше 10), то выдача их на экран компьютера дело несложное. Но если их число велико, то текст на экране будет быстро меняться. Поэтому прочитать его очень трудно или даже невозможно.

Однако если цель является внутренней целью программы, то внутренние унификационные подпрограммы Турбо-Пролога останавливают поиск решений после первого же успешного вычисления цели. Таким образом, выявляется только первое решение. Другие значения, которые могут быть присвоены переменным цели, не активизируются до тех пор пока программа не заставит внутренние унификационные подпрограммы повторно искать еще имеющиеся решения. В следующих разделах вы узнаете два способа реализации повторов в Турбо-Прологе. При программировании весьма удобны метод отката после неудачи и метод отсечения и отката.

4.4.1 Метод отката после неудачи

В данном разделе вы увидите, как метод отката после неудачи (ОПН) может быть использован для управления вычислением внутренней цели при поиске всех возможных ее решений. Метод ОПН использует предикат ***fail***. Программа о городах (листинг 4.1) демонстрирует использование этого предиката.

Назначение программы о городах состоит в перечислении названий десяти городов США. Утверждение первой подцели выдает заголовок ***Here are the cities*** (Это названия городов). Вторая подцель является правилом

для перечисления названий городов. Подправило ***cities(City)*** означивает переменную ***City (город)*** названием города. Затем это значение выдается на экран. Предикат ***fail*** вызывает откат к следующему утверждению, которое может обеспечить вычисление цели.

Листинг 4.1

```
/* Программа:  Города */
/* Назначение: Демонстрация использования          */
/*                  предиката fail и метода ОПН    */
        domains
                    name=symbol
        predicates
                    cities(name)
                    show_cities

        goal
              write("Here are the cities:"),nl
              show_cities

        clauses
cities("ANN ARBOR ").
cities("ATLANTA").
cities("NEW HAVEN").
cities("INDIANAPOLIS").
cities("BOSTON").
cities("MESA").
cities("MINEAPOLIS").
cities("SAN ANTONIO").
cities("SAN DIEGO").
cities("TAMPA").

        show_cities :-
                    cities(City),
                    write("      ", City), nl,
                    fail.
```

Из листинга 4.1 видим, что имеется 10 предикатов, каждый из которых является альтернативным утверждением для предиката ***cities(name).*** Во время попытки вычислить цель, внутренние унификационные подпрограммы означивают переменную ***City*** объектом первого утверждения, который есть ***ANN ARBOR*** (название города в США). Так как существует следующее утверждение, которое может обеспечить вычисление подцели ***cities(city***), то

указатель отката помещается в следующую точку. Значение ***ANN ARBOR*** выводится на экран.

Предикат ***fail*** вызывает неуспешное завершение правила, внутренние унификационные подпрограммы выполняют откат в точку 1, и процесс повторяется до тех пор, пока последнее утверждение не будет обработано.

* Упражнения

4.1. Измените программу о городах таким образом, что бы результатом была выдача на экран целых чисел 66, 46, 32, 93, 44, 98, 37, 16, 12. Выдавайте только одно число на строке.

4.2. Какие необходимы модификации для выдачи целых чисел на одну строку так, что бы они были разделены двумя пробелами?

Программа о городах показывает, что использование метода ОПН позволяет извлекать данные из каждого утверждения базы данных. Если в программе содержатся утверждения для 10 вариантов, то результат так же содержит 10 строк. Данные извлекаются из каждого утверждения, так как каждый вариант удовлетворяет подцели ***cities(city)***. Но добавив дополнительные условия на значения объектов для одной или более переменных предиката, можно извлекать данные только из определенных утверждений. Программа о служащих (листинг 4.2) демонстрирует этот метод.

Листинг 4.2

```
/* Программа:  Служащие          */
/* Назначение: Демонстрация использования          */
/*        селектирующих правил на основе ОПН-метода  */
domains
    name, sex, department = symbol
    pay_rate      = real

predicates
    employee(name,sex,department,pay_rate)
    show_male_part_time
    show_data_proc_dept

goal
    write("Служащие мужского пола с почасовой оплатой"), nl, nl,
                                    show_male_part_time.
clauses
    employee("John Walker      ","M","ACCT",3.50).
    employee("Tom Sellack      ","M","OPER",4.50).
    employee("Betty Lue        ","F","DATA",5.00).
    employee("Jack Hunter      ","M","ADVE",4.50).
    employee("Sam Ray          ","M","DATA",6.00).
```

```
        employee("Sheila Burton ","F","ADVE",5.00).
        employee("Kelly Smith   ","F","ACCT",5.00).
        employee("Diana Prince  ","F","DATA",5.00).
/* Правило для генерации списка служащих мужского пола  */
        show_male_part_time :-
                                employee(Name, "M", Dept, Pay_rate),
                                write(Name, Dept,  "$", Pay_rate),
                                nl,
                                fail.
/* Правило для генерации списка служащих отдела обработки данных */
        show_data_proc_dept :-
                                employee(Name, _, "DATA", Pay_rate),
                                write(Name, "$", Pay_rate),
                                nl,
                                fail.
```

Утверждения программы о служащих содержат данные о служащих компании, работающих неполный рабочий день. Предикат базы данных имеет вид:

```
employee(name, sex, department, pay_rate)
                /* служащий(имя, пол, отдел, почасовая_оплата) */
```

Следующее правило выдает содержимое всей базы данных:

```
show_all_part_time_employees :-  /* выдать_служаших */
                                employee(Name,Sex,Dept,Pay_rate),
                                write(Name, " ", Sex, " ", Dept,  " ", Pay_rate),
                                nl,
                                fail.
```

Переменные ***Name*** (имя), ***Sex*** (пол), ***Dept*** (отдел), ***Pay_rate*** (почасовая оплата) не означены, и следовательно, все они могут получить значения. Если бы это правило являлось целью программы, то результатом выполнения этой программы был бы список всех служащих с неполным рабочим днем.

Предположим, что необходимо получить список, содержащий данные только о служащих мужского пола. Для этого требуется, чтобы процесс сопоставления значений для ***Sex*** был успешным только для утверждений, содержащих ***M*** в позиции второго объекта.

Вспомните гл. 2, где говорилось, что константа сопоставима только сама с собой. Во время внутренней унификации постоянное значение ***M*** сопоставимо только с ***M***. Правило, накладывающее это условие на выборку данных, имеет вид:

```
        show_male_part_time :-
                        employee(Name,"M", Dept, Pay_rate),
                        write(Name,Dept, "$",Pay_rate),
                        nl,
                        fail.
```

Альтернативной формой условия выборки по половому признаку является предикат равенства ***Sex=M***. Используя этот предикат равенства, можно построить другое правило, имеющее тот же самый результат:

```
show_male_part_time :-
                    employee(Name, Sex, Dept, Pay_rate),
                    Sex="M",
                    write(Name,Dept, "$", Pay_rate),
                    nl,
                    fail.
```

Заметьте, что для этих правил некоторые подцели могут оказаться неуспешными из-за невозможности удовлетворить условию полового признака. Откат возникнет до момента, когда информация, содержащиеся в факте, будет выдана на экран. Предикат ***fail*** не потребуется, если условия правил невозможно выполнить, т.е. подцель будет неуспешной сама по себе. Предикат ***fail*** включен в правило для того, чтобы вызвать откат, если условия правила будут выполнены и все правило окажется успешным.

Теперь предположим, что необходимо получить список служащих с неполным рабочим днем, работающих в отделе обработки данных (***Data Processing Department).*** В этом случае условие, которое должно быть наложено на значение объекта для переменной ***Dept*** есть ***Data***. Два следующих правила позволяют достичь желаемого результата:

```
show_list_data_proc :-
                        employee(Name,Sex,"DATA",Pay_rate),
                        write(Name,Sex,"$",Pay_rate),
                        nl,
                        fail.
show_list_data_proc :-
                        employee(Name,Sex,Dept,Pay_rate),
                        Dept="DATA",
                        write(Name,Sex,"$",Pay_rate),
                        nl,
                        fail.
```

Метод ОПН удобен для программирования на Турбо-Прологе прикладных задач по обработке служебных данных. Типичными примерами обработки служебных данных являются следующие: создание платежной ведомости, вычисление выплаты, использующей данные из базы данных, генерация отчета о выплатах. Программа, формирующая платежную ведомость (листинг 4.3), является расширением программы о служащих.

Листинг 4.3

```
/* Программа:  Платежная ведомость                    */
/* Назначение: Демонстрация использования             */
/*             построения правил и генерации отчета   */
/*                  с использование ОПН-метода        */
```

```
domains
        name, sex, department     = symbol
        pay_rate,hours, gross_pay = real

predicates
        employee(name,sex,department,pay_rate,hours)
        make_pay_roll_report
        compute_gross_pay(pay_rate,hours,gross_pay)

goal
        write("Отчет о выплатах служащим"),
                                        nl, nl,
                                        make_pay_roll_report.
clauses
        employee("John Walker       ","M","ACCT",3.50,40.00).
        employee("Tom Sellack       ","M","OPER",4.50,36.00).
        employee("Betty Lue         ","F","DATA",5.00,40.00).
        employee("Jack Hunter       ","M","ADVE",4.50,25.50).
        employee("Sam Ray           ","M","DATA",6.00,30.00).
        employee("Sheila Burton ","F","ADVE",5.00,32.50).
        employee("Kelly Smith   ","F","ACCT",5.00,25.50).
        employee("Diana Prince  ","F","DATA",5.00,20.50).

        make_pay_roll_report :-
                        employee(Name,_,Dept,Pay_rate,Hours),
                        compute_gross_pay(Pay_rate,Hours,Gross_pay),
                        write(Name,Dept,"  $", Gross_pay),nl,
                        fail.

        compute_gross_pay(Pay_rate,Hours,Gross_pay) :-
                        Gross_pay = Pay_rate * Hours.
```

В программе формирования платежной ведомости предикат ***employee*** имеет пять объектов:

```
employee(name,sex,department,pay_rate,hours)
    /* служащий(имя, пол, отдел, почасовая_оплата, часы) */
```

Так как объекты ***pay_rate*** (почасовая оплата), ***hours*** (часы) и ***gross_pay*** (выплата) принадлежат домену типа ***real***, то над ними можно выполнять операции десятичной арифметики. Правило для вычисления выплаты несложно:

```
compute_gross_pay(Pay_rate, Hours, Gross_pay) :-
                                        Gross_pay=Pay_rate*Hours.
```

Задача правила **make_pay_roll_report** (выдать отчет о выплатах) заключается в формировании отчета. Оно вызывает правило ***comute_gross_pay*** для вычисления выплат.

* Упражнения

4.3. Напишите правило для выдачи на экран списка всех служащих женского пола, используя программу о служащих.

4.4. Для той же программы напишите правило для генерации списка всех служащих, у которых почасовая оплата 5 долларов.

4.5. Обувной магазин фиксирует количество обуви, проданной в течение дня. Данные о продаже включают индекс товара, его цену, проданное количество экземпляров данного типа, размер. Придумайте и спроектируйте программу на Турбо-Прологе, которая формирует отчет о продаже. Включите в суммарный объем дохода от продажи налог, предположив, что он равен 6.5% .

4.4.2 Метод отсечения и отката (ОО)

В некоторых ситуациях необходимо иметь доступ только к определенной части данных. Это бывает, например, когда запасные части, поступившие на склад, вносятся в опись сразу же после их поступления, или когда заявки на них фиксируются в произвольном порядке. Метод отсечения и отката (ОО) может быть использован для фильтрации данных, выбираемых из утверждений базы данных. Удобство этого метода становится более явным при большем числе условий для выборки. Например, метод ОО дает возможность выбрать все заявки, зафиксированные с 18 по 19 июня, имеющие литеру В в номере накладной, и полученные до того, как клерк Элайн Ларк заступил на дежурство. Задавая инициалы клерка как условие окончание просмотра базы данных, можно получить только требуемую часть информации.

Вы уже видели, каким образом предикат ***fail*** может вызвать откат к другим возможным вариантам решения задачи или подцели. Однако, для того, чтобы из базы данных выбирать данные, удовлетворяющие некоторым условиям, необходимо иметь средства управления откатом. Для этих целей Турбо-Пролог имеет встроенный предикат ***cut*** (отсечение). Предикат ***cut*** обозначается символом восклицательного знака (!). **Этот предикат, вычисление которого всегда завершается успешно, заставляет внутренние унификационные подпрограммы "забыть" все указатели отката, установленные во время попыток вычислить текущую подцель.**

Другими словами, предикат ***cut*** "устанавливает барьер", запрещающий выполнить откат ко всем альтернативным решениям текущей подцели. Однако последующие подцели могут создать новые указатели отката, и тем самым создать условия для поиска новых решений. Область действия предикат ***cut*** на них уже не распространяется. Но если все более поздние цели окажутся неуспешными, то барьер, установленный предикатом ***cut***, заставит механизм отката отсечь все решения в области действия ***cut*** путем немедленного отката к другим возможным решениям вне области действия ***cut***.

Метод отсечения и отката использует предикат ***fail*** для того, чтобы имитировать неуспешное вычисление и выполнять последующий откат, до

тех пор, пока не будет обнаружено определенное условие. Предикат ***cut*** служит для устранения всех последующих откатов.

Метод отката и отсечения демонстрирует простая ситуация, в которой предикаты базы данных содержат несколько имен, как это имеет место для предикатов ***child*** (ребенок) в программе, формирующей список имен детей (листинг 4.4).

Листинг 4.4

```
/*Программа: Имена детей                                    */
/* Назначение: Демонстрация использования предиката         */
/*          cut (!) и ОО-метода                             */
domains
        person = symbol

predicates
        child(person)
        show_some_of_them
        make_cut(person)

goal
        write("Мальчики и девочки"),
                                nl, nl,
                                show_some_of_them

clauses
        child("Tom").
        child("Beth").
        child("Jeff  ").
        child("Sarah       ").
        child("Larry       ").
        child("Peter       ").
        child("Diana       ").
        child("Judy        ").
        child("Sandy       ").

        show_some_of_them :-
                        child(Name),
                        write("   ", Name),
                        nl, make_cut(Name),!.

        make_cut(Name) :-Name="Diana".
```

Предположим, что необходимо выдать список имен до имени ***Diana*** включительно. Предикат ***cut*** выполнит отсечение в указанном месте. Предикат ***fail*** используется для продолжения откатов и доступа к последовательности имен базы данных до элемента с именем ***Diana***. Таким образом, поставленную задачу выполняет соответствующая комбинация предикатов ***cut*** и ***fail***. Эта комбинация называется методом отката и отсечения (ОО). Программа, формирующая список имен детей демонстрирует этот метод.

В программе об именах предикатом базы данных является ***child(person).*** Для этого предиката имеется 9 альтернативных утверждений. Правило, обеспечивающие генерацию всех имен (а не только некоторых) имеет вид:

```
show_them_all :-
                child(Name),
                write("   ", Name), nl,
                fail.
```

Оно основано на методе ОПН. При этом для того, что бы использовать предикат ***cut***, необходимо определить некоторое условие, которое может быть и простым, как в нашем примере, и очень сложным. В данном случае достаточным является условие ***Name=Diana***. Правило, определяющее, это условие имеет вид:

```
make_cut(Name) :-
                Name="Diana".
```

Это правило с последующим предикатом ***cut*** (!) образует правило ***make_cut*** (сделать отсечение). Теперь выполнение программы будет неуспешным, а откаты будут повторяться до тех пор, пока ***Name*** не окажется равным ***Diana.*** В этот момент результат выполнения правила ***make_cut*** будет успешным, что в свою очередь, вызовет выполнение предиката ***cut***. Таким образом, комбинируя правила отсечения с методом ОПН, можно получить ОО-правило:

```
show_some_of_them :-
                child(Name),
                write("   ", Name), nl,
                make_cut(Name),!.
make_cut(Name) :-
                Name="Diana".
```

Программа, формирующая список имен детей включает ОО-правило. Вы можете изменить выдачу программы, изменив имя объекта в условии отсечения. Например, ***Name=Beth***, выдает список имен до имени ***Beth*** включительно. Этот способ использования ОО-метода называется методом ранжированного отсечения.

* Упражнение

4.6. Измените правило ***make_cut*** в программе, формирующей список имен детей так, чтобы выдать список имен до имени ***Peter*** включительно.

Метод ОО можно использовать и иначе, например, для обработки выбираемых элементов. Программа, изображенная на листинге 4.5, демонстрирует использование ОО-метода для создания списка выбранных из базы данных элементов.

Листинг 4.5

```
/* Программа:  Новые детские имена   */
/* Назначение: Демонстрация использования предиката       */
/*                cut (!) и ОО-метода         */
domains
        name = symbol

predicates
        child(name)
        go_and_get_them

goal
        go_and_get_them

clauses
        child("Tom").
        child("Alice       ").
        child("Diana       ").
        child("Alice       ").
        child("Beth").
        child("Lee  ").
        child("Alice       ").

        go_and_get_them :-
                write(" Список имен"),
                nl,nl,
                child(Name),
                Name="Alice",
                write("   ",Name),nl,
                fail.
```

Как и в программе, формирующей список имен детей, в этой программе ***child(name)*** является предикатом базы данных, содержащим информацию, необходимую для генерации списка имен. Имеется 7 альтернативных утверждений. Три из них содержат имя ***Alice***. Правило, которое выбирает и выдает имя ***Alice,*** имеет вид:

```
get_alice :-
                child(Name),
```

```
                    Name="Alice",
                    write("  ", Name),nl,
                    fail.
```

В базе данных имя ***Alice*** встречается трижды, поэтому результатом работы приведенного правила будет список трех одинаковых имен.

Правило ***get_alice*** находит все (три) возможные означивания переменной Name, что является результатом применения метода ОПН. Однако при желании можно выдать только первый экземпляр значения переменной Name. Это достигается введением в правило выборки отсечения:

```
get_first_alice :-
                    child(Name),
                    Name="Alice",
                    write("  ", Name), nl,
                    !,
                    fail.
```

В результате будет получен список, состоящий из единственного имени ***Alice***. Утверждение цели (правила) в предыдущем примере программы содержит элементы правила ***get_first_alice***. Оно может быть использовано и как внутренняя и как внешняя цель. Тщательно проанализировав это правило, можно заметить, что предикат ***fail*** используется только один раз. К моменту, когда он получает управление, предикат ***cut*** уже устранил всякую возможность отката, в результате чего предикат ***fail*** оказывается бесполезным. Отметим, что методы отката и отсечения здесь представлены в самой общей форме, модифицировать которую для конкретного применения не составит труда. Таким образом, диапазон применения отката и отсечения достаточно широк.

* Упражнение

4.7. Модифицируйте программу, формирующую список новых имен. Для этого:

a) добавьте предикат, который содержит как первое, так и второе имя некоторых детей. Используйте следующий формат предиката:

child(First_name, last_name).

b) расширьте набор утверждений так, что бы включить первое и второе имя для всех детей;

c) напишите правило для выдачи на печать имен детей, второе имя которых **Smith (Смит);**

d) напишите правило для выдачи на печать полного имени, если первое имя есть **Alice;**

e) выполните модифицированные программы.

4.4.3 Метод повтора (МП), определяемый пользователем

МП-метод, как и ОО-метод, использует откат. Но в МП-методе выполнить откат возможно всегда в отличии от ОО-метода, где откат выполняются

только после искусственного созданного неуспешного результата. Правило рекурсии общего вида имеет более сложную структуру и является обобщением этих методов.

Вид правила повтора, определяемого пользователем, следующий:

repeat. **/* повторить */**

repeat :- repeat.

Первый ***repeat*** является утверждением, объявляющим предикат ***repeat*** истинным. Первый ***repeat*** не создает подцелей, поэтому данное правило всегда успешно. Однако, поскольку имеется еще один вариант для этого правила, то указатель отката устанавливается на первый ***repeat***. Второй ***repeat*** - это правило, которое использует само себя как компоненту (третий ***repeat***).Второй ***repeat*** вызывает третий ***repeat***, и этот вызов вычисляется успешно, так как первый ***repeat*** удовлетворяет подцели ***repeat***. Следовательно, правило ***repeat*** так же всегда успешно. Предикат ***repeat*** будет вычисляться успешно при каждой новой попытке его вызвать после отката. Факт в правиле будет использоваться для выполнения всех подцелей программы. Таким образом, ***repeat*** это рекурсивное правило, которое никогда не бывает неудачным.

Правило ***repeat*** широко используется в качестве компоненты других правил. Примером этого может служить программа Эхо (листинг 4.6), которая считывает строку, введенную с клавиатуры, и дублирует ее на экран. Если пользователь введет ***stop,*** то программа завершается.

Листинг 4.6

```
/* Программа:  Эхо                                          */
/* Назначение: Демонстрация использования МП-метода,        */
/*                  определенного пользователем             */

predicates
        write_message
        repeat
        do_echo
        check(name)

goal
        write_message,
                    do_echo.

clauses
        repeat.
        repeat :- repeat.

        write_message :-
              nl, write("Введите, пожалуйста, имена"), nl,
```

```
        write("Я повторю их"), nl,
        write("Чтобы остановить меня, введите stop"),nl,nl.

 do_echo :-
                repeat,
                readln(Name),
                write(Name),nl,
                check(Name),!.

check(stop) :-
                nl, write(" - OK, bye!").
check(_) :- fail.
```

Правило ***repeat*** является первым в разделе утверждений программы Эхо. Второе правило выводит информацию для пользователя. Третье правило ***do_echo*** (выполнить_эхо) является правилом повтора, определенным пользователем. Его первая компонента есть ***repeat***:

```
do_echo :-
        repeat,
        readln(Name),
        write(Name),nl,
        check(Name),!.
```

Утверждение ***repeat*** вызывает повторное выполнение всех следующих за ним компонент. Предикат ***readnl(Name)*** считывает строку с клавиатуры, ***write(Name)*** выдает (или моделирует эхо) ее на экран.

Последнее подправило ***check(Name)*** имеет два возможных значения. Одно определяется подправилом:

```
check(stop) :-
                nl, write(" - OK, bye!").
```

Если вводимая пользователем строка имеет значение ***stop***, то правило будет успешным. При этом курсор сдвигается на начало следующей строки, на экране появляется сообщение "***OK, bye!"*** (до свидания), и процесс повторения завершается. Обратите внимание на символ отсечения (!). Он служит для прекращения откатов, если условие ***check*** выполнено. Другое значение ***check(Name)*** определяется подправилом:

```
        check(Name) :- fail.
```

Если значение строки отлично от ***stop,*** то результат выполнения этого правила будет ***fail***. В этом случае произойдет откат к правилу ***repeat***. Таким образом, ***do_echo*** является конечным правилом в цепи повторений, условие выхода из которой определяется предикатом ***check***. Благодаря тому, что правило ***repeat*** является компонентой, правило ***do_echo*** становится конечным правилом повтора.

В программе Эхо правило повтора является первой компонентой правила ***do_echo***. Это очень гибкое средство программирования. Ниже будут

приведены сведения о других способах его применения. Правило повтора, являясь одной из компонент правила, обеспечивает циклическое выполнение основных функций данного правила.

Подобным образом в МП-правиле можно использовать более одного правила повтора. Ниже приведено МП-правило, включающие два правила повтора:

```
do_two_things :-
                    repeat1,
                    <повторяемое тело>,
                    <условие выхода>,!,
                    repeat2,
                    <повторяемое тело>,
                    <условие выхода>,!.

repeat1.
repeat1 :- repeat1.

repeat2.
repeat2 :- repeat2.

<правило для условия выхода 1>.
<правило для условия выхода 2>.
```

Во время определения правил повтора можно вместо имени ***repeat*** использовать какое-нибудь другое. Ниже в качестве примеров приведены несколько альтернатив слову ***repeat***:

```
loop.                          /* цикл  */
loop :- loop.

loop1.
loop1 :- loop1.

loop2.
loop2 :- loop2.

iterate.                       /* итерация */
iterate :- iterate.

recurse.                       /* рекурсия */
recurse :- recurse.
```

МП-метод наиболее эффективен при реализации доступа к данным в базе данных и файлах на диске, а также для организации выдачи на экран и формирования меню. Все эти вопросы будут рассмотрены в следующих главах.

* Упражнения

4.8. Измените программу Эхо так, чтобы она воспринимала целые числа с клавиатуры и дублировала их на экран. Напишите правило так, чтобы программа завершалась при вводе числа 0 (нуль). (Встроенный предикат Турбо-Пролога для считывания целых чисел с клавиатуры - это

readin(Number)

Здесь ***Number*** - имя переменной для целых чисел).

4.9. Модифицируйте программу Эхо так, что бы она воспринимала два десятичных числа с клавиатуры и дублировала их на экран. Затем заставьте программу вычислить сумму введенных десятичных чисел и выдать эту сумму на экран. Программа должна завершаться, если одно из двух вводимых чисел 0 (ноль). (Встроенный предикат Турбо-Пролога для считывания десятичных чисел с клавиатуры - это

readreal(Number)

Здесь ***Number*** - имя переменной для десятичных чисел). Правило для сложения двух десятичных чисел может быть записано в виде:

```
sum(X,Y,Z) :-
        Z = X+Y.
```

Здесь ***X, Y, Z*** - имена переменных для десятичных чисел.

4.5 Методы организации рекурсии

Правила, не использующие правил повтора в качестве компонент, являются наиболее общими способом организации рекурсивной обработки данных при программировании на Турбо-Прологе. Этот метод подходит для целого ряда применений, начиная с обработки файлов и кончая математическими вычислениями.

4.5.1 Простая рекурсия

Правило, содержащее само себя в качестве компоненты, называется правилом рекурсии. Правила рекурсии так же как правила повтора реализуют повторное выполнение задач. Они весьма эффективны, например, при формировании запросов к базе данных, а также при обработке таких доменных структур, как списки. Списки и рекурсия в Турбо-Прологе рассматриваются в гл. 5.

Пример правила рекурсии:

```
write_srting :- /* выдать строку */
        write("МЫ - ЭТО ВЕСЬ МИР"),
        nl,
        write_string.
```

Это правило состоит из трех компонент. Первые две выдают строку "МЫ - ЭТО ВЕСЬ МИР" и переводят курсор на начало следующей строки экрана. Третья - это само правило. Так как оно содержит само себя, то чтобы быть успешным, правило ***write_string*** должно удовлетворять само себе. Это приводит снова к вызову операции выдачи на экран строки и смещение курсора на начало новой строки экрана. Процесс продолжается бесконечно и в результате строки выдается на экран бесконечное число раз.

Однако в случае возникновения бесконечной рекурсии число элементов данных, используемых рекурсивным процессом, непрерывно растет и в некоторый момент стек переполнится. На экране появится сообщение об ошибке. Возникновение переполнения во время выполнения программы для пользователя нежелательно, так как в результате могут оказаться утерянными существенные данные. Избегать подобные ситуации можно увеличением размеров стека, для чего служит опция ***Miscellaneous settings*** (прочие установки параметров) в меню ***Setup*** (установка).

Если рекурсивное правило не генерирует указателей отката и последняя подцель правила является рекурсивным вызовом самого правила, то Турбо-Пролог устранит дополнительные расходы, вызываемые рекурсией. Этот процесс называется устранением хвостовой рекурсии.

Избежать возникновения бесконечной рекурсии можно. Для этого следует ввести предикат завершения, содержащий условие выхода. Формулировка условия выхода на русском языке для правила ***write_string*** может иметь вид: "Продолжать печать строки до тех пор, пока счетчик печати не превысит число 7. После чего остановить процесс". Определение условий выхода и включение их в правило рекурсии является очень важным элементом программирования на Турбо-Прологе.

Программа "Вернись" (листинг 4.7) демонстрирует простое правило рекурсии, в которое включено условие выхода.

Листинг 4.7

```
/* Программа:  Вернись                                          */
/* Назначение: Демонстрация построения рекурсивных              */
 /*                  правил для вводы и вывода символов          */
domains
        Char_data = char

predicates
        write_prompt
        read_a_character

goal
        write_prompt,
                read_a_character.

clauses
        write_prompt :-
                write("Пожалуйста, введите символы."), nl, nl,
                write("Для завершения введите #    "), nl, nl.
        read_a_character :-
                        readchar(Char_data),
                        Char_data <> '#',
```

write(Char_data),
read_a_character.

Программа циклически считывает символ, введенный пользователем: если этот символ не #, то он выдается на экран, если этот символ - #, то программа завершается. Правило рекурсии имеет вид:

read_a_character :-
readchar(Char_data),
Char_data <> '#',
write(Char_data),
read_a_character.

Первая компонента правила есть встроенный предикат Турбо-Пролога, обеспечивающий считывание символа. Значение этого символа присваивается переменной ***Char_data***. Следующее подправило, проверяет, является ли символ символом #. Если нет, то подправило успешно, символ выдается на экран и рекурсивно вызывается ***read_a_character***. Этот процесс продолжается до тех пор, пока внутренняя проверка не обнаружит недопустимый символ #. В этот момент обработка останавливается, и программа завершается.

* Упражнение

4.10. Запустите программу Вернись. После приглашения ***Goal***: введите последовательность символов:

The early bird gets the worm.#

4.5.2 Метод обобщенного правила рекурсии (ОПР)

Обобщенное правило рекурсии содержит в теле правила само себя. Рекурсия будет конечной, если в правило включено условие выхода, гарантирующее окончание его работы. Тело правила состоит из утверждений и правил, определяющих задачи, которые должны быть выполнены.

Ниже в символическом виде дан общий вид правила рекурсии:

<имя правила рекурсии> :-
<список предикатов>, (1)
<предикат условия выхода>, (2)
<список предикатов>, (3)
<имя правила рекурсии>, (4)
<список предикатов>. (5)

Хотя структура этого правила сложнее чем структура простого правила рекурсии, рассмотренного в предыдущем разделе, однако принципы, применяемые к первому из них применимы и ко второму.

Данное правило рекурсии имеет пять компонент. Первая - это группа предикатов. Успех или неудача любого из них на рекурсию не влияет. Следующая компонента - предикат условия выхода. Успех или неудача этого предиката либо позволяет продолжить рекурсию, либо вызывает ее оста-

новку. Третья компонента - список других предикатов. Аналогично, успех или неудача этих предикатов на рекурсию не оказывает влияния. Четвертая группа - само рекурсивное правило. Успех этого правила вызывает рекурсию. Пятая группа - список предикатов, успех или неудача которых не влияет на рекурсию. Пятая группа также получает значения (если они имеются), помещенные в стек во время выполнения рекурсии.

Вспомним, что правило рекурсии должно содержать условие выхода. В противном случае рекурсия бесконечна и правило бесполезно. Ответственность за обеспечение завершаемости правила рекурсии лежит на программисте. Правила, построенные указанным образом, являются обобщенными правилами рекурсии (ОПР), а метод называется ОПР-методом.

Например, вы хотите написать правило генерации всех целых чисел начиная с 1 и кончая 7. Пусть имя правила будет

write_number(Number).

Для этого примера первая компонента структуры общего правила рекурсии не используется. Второй компонентой, т.е. предикатом выхода, является ***Number < 8.*** Когда значение ***Number*** равно 8, правило будет успешным и программа завершится.

Третья компонента правила оперирует с числами. В этой части правила число выдается на экран и затем увеличивается на 1. Для увеличенного числа будет использоваться новая переменная ***Next_Number***. Четвертая компонента - вызов самого правила рекурсии ***write_number(Next_number).*** Пятая компонента, представленная в общем случае, здесь не используется.

Листинг 4.8

```
/* Программа:  Генерация ряда                                  */
/* Назначение: Демонстрация использования рекурсии для  */
/*             генерации ряда чисел в порядке                  */
/*                  возрастания                                */
domains
        number = integer

predicates
        write_number(number)

goal
        write("Here are the numbers:"),
                                        nl,nl,
                                        write_number(1),
                                        nl,nl,
                                        write("          All done, bye!").
clauses
        write_number(8).
        write_number(Number) :-
```

```
        Number < 8,
        write("          ", Number), nl,
        Next_number = Number + 1,
        write_number(Next_number).
```

Программа генерации ряда чисел (листинг 4.8) использует следующее правило рекурсии:

```
write_number(8).
write_number(Number) :-
                Number < 8,
                write(Number), nl,
                Next_Number = Number + 1,
                write_number(Next_number).
```

Программа начинается с попытки вычислить подцель ***write_number(1).*** Сначала программа сопоставляет подцель с первым правилом ***write_number(8).*** Так как 1 не равно 8, то сопоставление неуспешно. Программа вновь пытается сопоставить подцель, но уже с головой правила ***write_number(Number)***. На этот раз сопоставление успешно вследствие того, что переменной ***Number*** присвоено значение 1. Программа сравнивает это значение с 8; это условие выхода. Так как 1 меньше 8, то подправило успешно. Следующий предикат выдает значение, присвоенное ***Number***. Переменная ***Next_Number*** получает значение 2, а значение ***Number*** увеличивается 1. В этот момент правило ***write_number*** вызывает само себя с новым значением параметра, равным 2 и присвоенным ***Next_Number***.

Заметим, что необязательно вызывать правило, используя то же имя переменной, что используется в голове правила. Это всего лишь позиция в списке параметров, имеющая значение при передаче значений. Фактически если не передавать значение ***Next_Number***, то приращение основного числа программы невозможно. При рекурсивном вызове головы правила, программа снова пытается выполнить подцель ***write_number***(8). Программа продолжает выполнять цикл сопоставления, присвоения и выдачи значений ***Number*** до тех пор, пока значение ***Number*** не станет равным 8. В этот момент цель выполнена, правило успешно и программа завершается после выдачи сообщения ***All done, bye!*** (Все сделано, привет!). Результат работы этой программы есть список целых чисел, выданных на экран.

* Упражнения

4.11. Измените программу генерации ряда чисел так, чтобы она выдавала все целые числа от 53 до 62.

4.12. Измените подцель и правило рекурсии так, чтобы результатом программы была генерация целых чисел от 1 до 7 в порядке убывания.

Важное свойство правила рекурсии состоит в его расширяемости. Например, оно может быть расширено для подсчета суммы ряда целых чисел.

Листинг 4.9

```
/* Программа:  Сумма ряда 1                                    */
/* Назначение: Демонстрация использования рекурсивного  */
 /*                  предиката для нахождения суммы S(N) ряда */
/*                  S, где N положительное целое число          */
/* Пример:          S(7) = 7+6+5+4+3+2+1 = 28                   */
/* Указание:        Запустите программу. Оператор цели          */
/*                        включен в программу                   */
domains
        number, sum = integer
predicates
        sum_series(number, sum)
goal
        sum_series(7,Sum),
                        write("Сумма ряда:"),nl,nl,
                        write("   S(7) =  ", Sum), nl.
clauses
        sum_series(1,1).            /* сумма ряда */
        sum_series(Number,Sum) :-
                                Number > 0,
                                Next_number = Number - 1,
                                sum_series(Next_number, Partial_Sum),
                                Sum = Number + Partial_Sum.
```

Программа Сумма ряда 1 (листинг 4.9) использует правило рекурсии для вычисления суммы ряда целых чисел от 1 до 7:

S(7) = 1 + 2 + 3 + 4 + 5 + 6 + 7 = 28

или

S(7) = 7 + 6 + 5 + 4 + 3 + 2 + 1 = 28

Правило рекурсии программы выполняет вычисления по обычной схеме сложения:

```
   1      Начальное значение
 + 2      Следующее значение
 ___
   3      Частичная сумма
 + 3      Следующее значение
 ___
   6      Частичная сумма
  ...
```

Правило рекурсии имеет вид:

```
sum_series(1,1).                    /* сумма ряда */
sum_series(Number,Sum) :-
```

Number > 0,
Next_number = Number - 1,
sum_series(Next_number, Partial_Sum),
Sum = Number + Partial_Sum.

Данное правило имеет четыре компоненты и одно дополнительное нерекурсивное правило. Заметим, что последняя компонента правила рекурсии - это правило ***Sum*** с ***Partial_Sum*** (частичная сумма) в качестве переменной. Это правило не может быть выполнено до тех пор, пока ***Partial_Sum*** не получит некоторого значения.

Программа Сумма ряда 1 начинается с попытки выполнить подцель ***sum_series(7,Sum).*** Сначала программа пытается сопоставить подцель с подправилом ***sum_series(1,1).*** Сопоставление неудачно. Затем она пытается сопоставить подцель с ***sum_series(Number,Sum).*** На этот раз сопоставление завершается успешно с присвоением переменной ***Number*** значения 7. Затем программа сравнивает значение ***Number***, которое равно 7, с 0, т.е. проверяется условие выхода. Так как 7 больше 0, то сопоставление успешно, программа переходит к следующему подправилу.

Для этого подправила переменной ***Next_number*** присвоено значение 6, т.е. значение ***Number*** - 1. Затем правило вызывает само себя в виде ***sum_series(6,Partial_Sum).*** Следующим подправилом является правило ***Sum,*** содержащее свободную переменную ***Partial_Sum***. Так как только что был вызван рекурсивный процесс, то правило ***Sum*** не может быть вызвано.

Теперь программа пытается сопоставить неизменяемое правило ***sum_series(1,1)*** с ***sum_series(6,Partial_Sum).*** Процесс сопоставления неуспешен, поскольку несопоставим ни один из параметров. В результате программа переходит к следующему правилу с головой ***sum_series(Number,Sum),*** присваивая переменной ***Number*** значение 6.

Этот циклический процесс сопоставления продолжается до тех пор, пока не будет получено ***sum_series(1,Partial_Sum).*** Теперь это правило сопоставляется с ***sum_series(1,1), а Partial_Sum*** приписывается значение 1. При сопоставлении правила с головой правила переменная ***Sum*** получает значение 1. Так как сопоставление продолжается дальше, то ***Next_number*** получает значение 0 (1 - 1). При следующем цикле сопоставления переменная ***Number*** получает значение 0. Во время сопоставления с условием выхода правило оказывается неуспешным, и сопоставление "прыгает" к правилу ***Sum.***

Во время процесса сопоставления переменная ***Partial_Sum*** была свободна, а программа запоминала значения ***Number*** для последующего использования. Но это правило продолжает означивать переменную ***Sum***, присваивая ей последовательно значения 1, 3, 6, 10, 15, 21 и 28. Конечное значение ***Sum*** есть 28.

* Упражнения

4.13. Модифицируйте программу Сумма ряда 1 так, чтобы ее результатом была сумма следующего ряда нечетных чисел:

S(15) = 1 + 3 + 5 + . . . + 15

4.14. Постройте таблицу обрабатываемых значений для программы Сумма ряда 1, показывающую работу правила рекурсии.

Листинг 4.10

```
/* Программа:  Сумма ряда 2                                    */
/* Назначение: Демонстрация использования рекурсивного         */
/*             предиката для нахождения суммы ряда S,          */
/*             где S(N), где N положительное целое число       */
/* Пример:     S(7) = 7+6+5+4+3+2+1 = 28                       */

domains
        number, sum = integer

predicates
        sum_series(number, sum)

goal
        sum_series(7,Sum),
                write("Сумма ряда:"),nl,nl,
                write("   S(7) =  ", Sum), nl.
clauses
        sum_series(1,1) :- !.           /* сумма ряда */
        sum_series(Number,Sum) :-
                        Next_number = Number - 1, s
                        um_series(Next_number,
                        Partial_Sum),
                        Sum = Number + Partial_sum.
```

Программа Сумма ряда 2 (листинг 4.10) является модификацией программы Сумма ряда 1. Модификация выполнена посредством удаления условия выхода ***Number*** > 0 и введения правила ***sum_series(1,1) :- !.*** вместо ***sum_series(1,1).***

Сравним вид правила рекурсии в предыдущей программе с модифицированным правилом рекурсии в программе Сумма ряда 2:

Правило рекурсии Сумма ряда 1

```
sum_series(1,1).            /* сумма ряда */
sum_series(Number,Sum) :-
                        Number > 0,
                        Next_number = Number - 1,
                        sum_series(Next_number, Partial_Sum),
```

```
                    Sum = Number + Partial_Sum.
```

Правило рекурсии Сумма ряда 2

```
sum_series(1,1) :- !.          /* сумма ряда */
sum_series(Number,Sum) :-
                    Next_number = Number - 1,
                    sum_series(Next_number, Partial_Sum),
                    Sum = Number + Partial_sum.
```

Результаты работы этих правил идентичны. Использование отсечения (!) в Сумме ряда 2 не улучшает работы правила рекурсии. Эти два правила следует рассматривать как альтернативные варианты.

* Упражнения

4.15. Модифицируйте программу Сумма ряда 2 так, чтобы она вычисляла сумму следующего ряда целых четных чисел:

S(16) = 2 + 4 + 6 + 8 + 10 + 12 + 14 + 16.

4.16. Составьте таблицу обрабатываемых значений для предыдущей программы, показывающую работу правила рекурсии.

Листинг 4.11

```
/* Программа:  Факториал */
 /* Назначение: Демонстрация использования рекурсии для  */
/*                  процедуры вычисления факториала N!        */
/*                  положительного числа N.   Процедура          */
/*                  использует  предикат cut для запрещения отката */
/* Пример:     7! = 7*6*5*4*3*2*1 = 5040                          */

domains
        number, product = integer

predicates
        factorial(number, product)

goal
        factorial(7,Result),
                        write("  7! =     ",Result),nl.
clauses
        factorial(1,1) :- !.
        factorial(Number,Result) :-
                        Next_number = Number -1,
                        facto-rial(Next_number,
                        Partial_factorial),
                        Result = Number * Partial_factorial.
```

Программа Факториал (листинг 4.11) использует правило рекурсии для вычисления и печати факториала целого числа. (Факториал числа N записывается как N!. Восклицательный знак это распространенное обозначение факториала и его не следует путать с символом для отсечения). N! есть произведение всех целых чисел от 1 до N:

N! = N * (N-1) * (N-2) * ... 2 * 1

Примеры:

1! = 1
2! = 2 * 1 = 2
3! = 3 * 2 * 1 = 6
4! = 4 * 3 * 2 * 1 = 24
5! = 5 * 4 * 3 * 2 * 1 = 120
6! = 6 * 5 * 4 * 3 * 2 * 1 = 720
7! = 7 * 6 * 5 * 4 * 3 * 2 * 1 = 5040

Основополагающая структура правила рекурсии для вычисления факториала точно такая же как и для правила рекурсии предыдущей программы. Для суммирования ряда использовалось последовательное суммирование. Это суммирование выполнялось посредством рекурсии. Для вычисления факториала используется последовательное произведение. Его значение получается после извлечения значений из стека в качестве списка параметров для последнего подправила после того, как рекурсия была остановлена. Правило рекурсии для вычисления факториала следующее:

```
factorial(1,1) :- !.
factorial(Number,Result) :-
                        Next_number = Number -1,
                        facto-rial(Next_number,
                        Partial_factorial),
                        Result = Num-ber*Partial_factorial.
```

В результате работы программы получим 7! = 5040.

* Упражнение

4.17. Измените программу Факториал так, чтобы она вычисляла и выдавала на экран факториал 10. Факториал 10 равен 3 628 800. Предупреждение: Для вычисления используйте домен действительных чисел. Результат слишком велик для того, чтобы его хранить в переменной целого типа. Это объясняется тем, что в Турбо-Прологе верхний предел для значения целого числа равен 32 767.

4.6 Обзор содержания главы

В данной главе были рассмотрены четыре метода построения правил: метод отката после неудачи (ОПН), метод отсечения и отката (ОО), метод повтора (МП), определяемый пользователем и обощенное правило рекурсии (ОПР).

При помощи ОПН-метода вы научились использовать встроенный предикат ***fail*** для управлением механизмом отката Турбо-Пролога. Работа этого правила была продемонстрирована на примерах программ.

Введение ОО-метода продемонстрировало использование отсечения (!), которое является встроенным средством Турбо-Пролога, а при обсуждении МП-метода вы узнали, как присущая Турбо-Прологу возможность выполнять рекурсию работает в определяемом пользователем правиле рекурсии. Вы также узнали, как вызывать это правило из других правил.

Наконец, был представлен ОПР-метод построения рекурсивных правил. Его обсуждение включало применение этих правил в типичных ситуациях. В качестве примеров рассматривалось печать последовательности целых чисел, суммирование рядов и нахождение факториала целого числа. Надеемся, что с этого момента применение этих методов для построения правил не составит для вас труда. Рассмотренные методы являются мощным средством, которое очень часто используется при создании программ.

Глава 5. Использование списков

5.1 Введение

В гл. 3 вы познакомились с основами представления данных в Турбо-Прологе. Это прежде всего утверждения, объектами которых являются конкретные значения (данные). Турбо-Пролог также поддерживает связанные объекты, называемые списками. **Список - это упорядоченный набор объектов, следующих друг за другом.** Составляющие списка внутренне связаны между собой , поэтому с ними можно работать и как с группой (списком в целом), так и как с индивидуальными объектами (элементами списка).

Турбо-Пролог позволяет выполнять со списком целый ряд операций. Их перечень включает:

* доступ к объектам списка;
* проверка на принадлежность к списку;
* разделение списка на два;
* слияние двух списков;
* сортировку элементов списка в порядке возрастания или убывания.

Списки бывают, полезны при создании баз знаний (баз данных), экспертных систем, словарей; перечень областей применения можно продолжать еще долго. В настоящей главе рассматриваются структура, организация и представление списков, демонстрируются некоторые из методов, применяемых при программировании на Турбо-Прологе.

5.2 Списки и Турбо-Пролог

Список является набором объектов одного и того же доменного типа. **Объектами списка могут быть целые числа, действительные числа, символы, символьные строки и структуры**. Порядок расположения

элементов является отличительной чертой списка; те же самые элементы, упорядоченные иным способом, представляют уже совсем другой список. Порядок играет важную роль в процессе сопоставления.

Турбо-Пролог допускает списки, элементами которых являются структуры. Если структуры принадлежат к альтернативному домену, элементы списка могут иметь разный тип. Такие списки используются для специальных целей, и их рассмотрение не входит в задачу настоящей работы.

Совокупность элементов списка заключается в квадратные скобки ([]), а друг от друга элементы отделяются запятыми. Примерами списков могут служить:

[1,2,3,6,9,3,4]
[3.2,4.6,1.1,2.64,100.2]
["YESTERDAY","TODAY","TOMORROW"]

Элементами первого списка являются целые числа. Элементами второго - действительные числа, третьего - символьные строки, т. е. конкретные значения символов. Любой печатный символ кода ASCII пригоден для списков этого типа.

5.2.1 Атрибуты списка

Объекты списка называются элементами списка. Список может содержать произвольное число элементов, единственным ограничением является лишь объем оперативной памяти. Турбо-Пролог требует, чтобы **все элементы списка принадлежали к одному и тому же типу доменов.** Другими словами, либо все элементы списка - целые числа, либо все - действительные, либо все - символы, либо - символьные строки. В Турбо-Прологе список

["JOHN WALKER",3.50,45.50]

некорректен ввиду того, что составлен из элементов разных типов. **Списки структур являются исключением из этого правила**.

Количество элементов в списке называется его длиной. Длина списка **["MADONNA","AND","CHILD"]** равна 3.Длина списка **[4.50,3.50,6.25,2.9,100.15]** равна 5. **Список может содержать всего один элемент и даже не содержать элементов вовсе:**

["summer"]
[]

Список, не содержащий элементов, называется пустым или нулевым списком.

Непустой список можно рассматривать как состоящий из двух частей: (1) первый элемент списка - его **голова**, и (2) остальная часть списка - **хвост**. Голова является элементом списка, хвост есть список сам по себе. Голова - это отдельное неделимое значение. Наоборот, хвост представляет из себя список, составленный из того, что осталось от исходного списка в результате "усекновения головы". Этот новый список зачастую можно делить и дальше. Если список состоит из одного элемента, то его можно разделить на голову, которой будет этот самый единственный элемент, и хвост, являющийся пустым списком.

В списке

[4.50,3.50,6.25,2.9,100.15]

например, головой является значение 4.50, а хвостом - список

[3.50,6.25,2.9,100.15]

Этот список в свою очередь имеет и голову, и хвост. Голова - это значение 3.50, хвост - список

[6.25,2.9,100.15]

В табл. 5.1 показаны головы и хвосты нескольких списков.

Таблица 5.1. Головы и хвосты различных списков

Список	**Голова**	**Хвост**
[1,2,3,4,5]	1	[2,3,4,5]
[6.9,4.3,8.4,1.2]	6.9	[4.3,8.4,1.2]
[cat, dog, horse]	cat	[dog, horse]
['S', 'K', 'Y']	'S'	['K','Y']
["PIG"]	"PIG"	[]
[]	не определена	не определен

5.2.2 Графическое представление списков

Графическое представление списков является полезным наглядным вспомогательным средством при проектировании доменных структур и задании данных для Ваших программ на Турбо-Прологе. Его также используют при документировании прикладных программ и системного матобеспечения. В настоящем разделе обсуждаются два способа графического представления списков.

Первый из них - это изображение списка при помощи линейного графа. Рассмотрим следующее утверждение:

number([66, 84, 12, 32]).

Объектом предиката ***number*** является четырехэлементный список. Голова этого списка есть число 66, хвост - список [84,12,32]. Нумерация списка начинается с головы и заканчивается на его последнем элементе, числе 32.

Список, составленный из 4 целых чисел, можно представить в виде направленного линейного графа, элементы списка связаны между собой ребрами этого графа. Направление показывает очередность, в которой можно добраться до соответствующего элемента. Данный способ изображения списка весьма уместен для наглядного представления порядка элементов в списке.

Этот же список можно представить в виде бинарного дерева-графа. Функтор списка, ***number***, является корнем этого дерева. От корня отходят две ветви. Левая заканчивается листом со значением 66. Правая ветвь кончается узлом, из которого расходятся еще две ветви. Левая кончается значением 84, правая опять разветвляется на две ветви. На левой из них располага-

ется лист со значением 12, правая ветвится еще раз. Левая из этих ветвей ведет к листу со значением 32, правая заканчивается пустым списком.

Очередность, в которой элементы связаны друг с другом, начинается с корня дерева. Лист самой левой ветви дает первый элемент списка. Затем очередность при посредстве узла правой ветви переходит на следующую левую ветвь, на ее листе находится второй элемент списка, 84. Аналогично нумеруются и все остальные элементы вплоть до последнего, 32. Так как больше неупорядоченных элементов не остается, то дерево оканчивается ветвью с пустым списком.

Внутренние процедуры унификации в Турбо-Прологе соблюдают это заданное графами направление упорядочивания. Турбо-Пролог пытается сопоставить первый элемент с головой списка, а затем продолжает в заданном графом направлении. Изображение в виде бинарного дерева бывает особенно полезным для наглядной интерпретации процесса отката в виде направленного перебора ветвей дерева.

* Упражнения

5.1. Для списка

games([football,soccer,tennis,baseball])

а) Нарисовать линейный граф. Для упорядочивания элементов используйте стрелки;

б) Нарисовать бинарный граф.

Сколько узлов будет иметь это дерево ?

5.2. Составьте список, элементами которого являются имена гномов из сказки "Белоснежка и семь гномов" (сколько вспомните). Какова длина получившегося списка ?

5.3. Составьте список, состоящий из названий десяти наиболее популярных видов спорта.

5.4. Различаются ли с точки зрения Турбо-Пролога два таких списка:

[63,29,24,27,86]

[63,24,29,27,86]

Аргументируйте свой ответ.

5.5. Можно ли назвать корректным следующий список ?

score(["Kennedy High School",6,3,8,9,6.2])

5.6. Корректен ли список

symbols(["*","+","-","?","#","&"])

Если да, то почему ?

5.3 Применение списков в программе

Для того чтобы использовать в программе список, необходимо описать предикат списка. Ниже приведены примеры таких предикатов:

num([1,2,3,6,9,3,4])
realnum([3.2,4.6,1.1,2.64,100.2])
time(["YESTERDAY","TODAY","TOMORROW"])

В этих выражениях **num, realnum** и **time** все представляют предикаты списков. Предикатам списков обычно присваиваются имена, которые характеризуют либо тип элемента (***num)***, либо сами данные (***time***).

Введение списков в программу с необходимостью отражается на трех ее разделах. Домен списка должен быть описан в разделе ***domains***, а работающий со списком предикат - в разделе ***predicates***. Наконец, нужно ввести сам список; то есть, другими словами, его нужно задать где-то в программе: либо в разделе ***clauses,*** либо в разделе ***goal.*** Покажем, как можно создавать в программе списки и как их использовать.

Каждый элемент приведенного ниже списка обозначает одну из птиц.

birds(["sparrow", "robin", "mockingbird", "thunderbird", "bald eagle"]).

Если этот список необходимо использовать в программе, то следует описать домен элементов списка; ему логично присвоить имя подобное ***name_birds*** (названия птиц). Как уже известно, список может содержать много элементов, может содержать один элемент, а может не содержать ни одного.

Отличительной особенностью описания списка является наличие звездочки (*) после имени домена элементов. Так запись

bird_name *

указывает на то, что это домен списка, элементами которого являются ***bird_name***, т. е. запись ***bird_name**** следует понимать как список, состоящий из элементов домена ***bird_name***. Описание в разделе ***domains,*** следовательно, может выглядеть либо как

bird_list = bird_name *
bird_name = symbol ,

либо как

bird_list = symbol *

Домен ***bird_list*** является доменом списка элементов типа ***symbol*** (списка птиц).

В разделе описания предикатов ***predicates*** требуется присутствия имени предиката, а за ним заключенного в круглые скобки имени домена.

birds(bird_list)

Как видим, описание предиката списка ни в чем не отличается от описания обычного предиката.

Сам список присутствует в разделе утверждений ***clauses***:

birds(["sparrow","robin","mockingbird","thunderbird", "bald eagle"]).

Чуть позднее будет описано, как использовать списки в разделе программы ***goal***.

Законченный пример использования списков приведен в программе "Списки" (листинг 5.1), в которую дополнительно включены список из 7 целых чисел (домен ***number_list***) и предикат ***score.***

Листинг 5.1

```
/* Программа:   Списки                    */
/* Назначение:  Работа со списками.       */
```

```
domains
        bird_list = bird_name *
        bird_name = symbol
        number_list = number *

number = integer
        predicates
        birds(bird_list)
        score(number_list)

clauses
        birds(["sparrow",
                        "robin",
                        "mockingbird",
                        "thunderbird",
                        "bald eagle"]).

        score([56,87,63,89,91,62,85]).
        /*****           конец программы          *****/
```

Эта программа создавалась в расчете на следующие внешние запросы:

```
birds(All).
birds([_,_,_,B,_]).
birds([B1,B2,_,_,_]).
score(All).
score([F,S,T,_,_,_,_]).
```

Поясним теперь работу программы при задании каждой из этих целей, это прояснит способ работы со списками в программах на Турбо-Прологе. Заметим, что свободная переменная ***All*** представляет весь список в целом. Он рассматривается при этом как некое целое, элементы играют роль частей этого целого.

Что касается второй цели, **birds([_,_,_,B,_]),** то процесс сопоставления начинается с первого элемента. Первые три переменные в целевом утверждении являются анонимными, при сопоставлении это обстоятельство, однако, роли не играет. Переменной В присваивается значение ***thunderbird.*** В этом процессе используется внутренняя связь элементов. В результате удовлетворения цели появляется строка ***B=thunderbird***.

Третья цель, **birds([B1,B2,_,_,_]),** запрашивает первые два элемента списка. На выходе можно будет увидеть ***B1=sparrow, B2=robin***, т. е. значения первого и второго элементов списка.

Обратимся теперь ко второй части программы, имеющей дело со списком целых чисел. В случае со ***score(All)*** переменной ***All*** присваивается весь список из 7 элементов, а выдача будет выглядеть так:

```
All=[56,87,63,89,91,62,85].
```

Пятая цель - это **score([S1,_,_,S4,_,S6,_]).** На выходе будем иметь ***S1=56, S4=89, S6=62***. Также как и в случае с третьей целью, внутренне связанные элементы будут выбираться селективно в соответствии с порядком в списке.

* Упражнения

5.7. Нарисуйте линейный граф для списка

birds(bird_list).

5.8. Нарисуйте дерево для списка

score(number_list).

5.9. Запустите программу "Списки" и введите такое целевое утверждение:

birds([S,R,M,T,B]).

Какова будет выдача программы, и в чем она будет отличаться от выдачи при целевом утверждении ***birds(All)?***

5.4 Использование метода с разделением списка на голову и хвост

В программе "Списки" для получения доступа к элементам списков были использованы внешние целевые утверждения. Задание цели в виде ***birds(All)*** обеспечивало присваивание переменной ***All*** всего списка в целом. Напротив, цель ***birds([_,_,_,B,_])*** позволила извлечь из списка лишь один элемент. В этом случае, однако, требовалось точное знание числа элементов списка, являющегося объектом предиката ***birds***. Если зададать цель в виде ***birds([B1,B2,B3]),*** то цель не будет удовлетворена ввиду несоответствия между количеством элементов в списке и количеством элементов в целевом утверждении.

Турбо-Пролог, однако, позволяет отделять от списка первый элемент и обрабатывать его отдельно. Данный метод работает вне зависимости от длины списка, до тех пор, пока список не будет исчерпан. Этот метод доступа к голове списка называется методом разделения списка на голову и хвост.

Рассмотрим список [4, 9, 5, 3]. В этом исходном списке головой является элемент 4, а хвостом - список [9,5,3]. Головой нового списка будет уже число 9, хвостом - список [5,3]. Этот список также имеет голову 5 и хвост [3]. Наконец, список [3] состоит из головы - числа 3 и хвоста, являющегося пустым списком. Как Вы скоро увидите, неоднократное разделение списка на голову и хвост играет важную роль в программировании на Турбо-Прологе.

Операция деления списка на голову и хвост обозначается при помощи вертикальной черты (|):

[Head|Tail].

Head здесь является переменной для обозначения головы списка, переменная ***Tail*** обозначает хвост списка. (Для имен головы и хвоста списка пригодны любые допустимые Турбо-Прологом имена.)

Программа "Голова-хвост" (листинг 5.2) демонстрирует использования метода разделения списка. Два списка описаны в ней: список целых чисел (имя домена - number_list) и список символических имен (домен ***animal_list***). Правило ***print_list*** применяется для доступа к элементам обоих списков.

Листинг 5.2

```
/* Программа:   Голова-хвост     Файл: PROG0502.PRO */
/* Назначение:  Работа со списками  путем          */
/*              деления на голову и хвост.         */
        domains
                number_list = integer *

                animal_list = symbol *

        predicates
print_list(number_list) print_list(animal_list)

        clauses
                print_list([]).
                print_list([Head|Tail]) :-
                        write(Head),nl,
                        print_list(Tail).
        /*****           конец программы          *****/
```

Программа "Голова-хвост" использует правило

```
print_list([]).
print_list([Head|Tail]) :-
        write(Head),nl,
        print_list(Tail).
```

для доступа к элементам списков. Так как предикат ***print_list*** определен для объектов обоих доменов, то это правило используется для работы как со списком ***number_list***, так и списком ***animal_list***.

Когда правило пытается удовлетворить цель

```
print_list([4,9,5,3])
```

то первый вариант правила, ***print_list[],*** дает неуспех, так как его объект является пустым списком. Напротив, введенный список соответствует объекту второго варианта предиката, ***print_list([Head|Tail]).*** Переменной ***Head,*** следовательно, присваивается значение первого элемента в списке, 4, в то время как переменной ***Tail*** ставится в соответствие оставшаяся часть списка, [9,5,3].

Теперь, когда выделен первый элемент списка, с ним можно обращаться так же, как и с любым простым объектом:

```
write(Head),nl,
```

Так как хвост списка есть список сам по себе, значение переменной Tail может быть использовано в качестве объекта рекурсивного вызова print_list:

print_list(Tail)

Когда испытывается данное подправило, ***Tail*** имеет значение [9,5,3]. Снова не проходит первый вариант, и соответствие устанавливается при помощи второго. Переменной ***Head*** присваивается значение 9,котрое затем печатается на экране. Процесс повторяется со списком [5,3].

В конце концов, когда переменная ***Head*** принимает значение 3, переменной ***Tail*** присваивается пустой список. Теперь при рекурсивном вызове ***print_list(Tail)*** значение ***Tail*** соответствует объекту правила

print_list([])

Ввиду того, что этот вариант не имеет рекурсивных вызовов, цель считается удовлетворенной, и таким образом вырабатывается условие окончания рекурсии ***print_list***. Первый вариант позоляет ***print_list*** завершиться успехом, когда рекурсивные вызовы опустошат весь список.

Похожий процесс имеет место и при задании цели

print_list(["cat","dog","horse","cow"]).

Сначала переменной ***Head*** присваивается значение ***cat***, ***cat*** печатается на экране, а ***Tail*** принимает значение **["dog","horse","cow"].** В дальнейшем при последовательном выполнении рекурсий все эти значения также отображаются на экран. Наконец, когда ***Head*** приняло значение последнего элемента исходного списка, ***cow***, значением ***Tail*** становится пустой список. Вариант

print_list[]

дает успех, тем самым завершая рекурсию.

Рекурсивные правила для работы со списками просты, но вместе с тем и очень важны, ввиду их применимости в большинстве программ.

* Упражнение

5.10. Для списка **["Boston","Philadelphia","Seattle","Chicago"]**

а) нарисуйте диаграмму работы со списком при помощи метода деления списка на голову и хвост;

б) запишите значения голов списков в процессе работы метода. в. Напишите рекурсивное правило, которое печатает все элементы списка через два пробела.

5.5. Различные операции над списками

Различные операции, которые можно проделать над списками, включают в себя поиск элемента в списке, деление списка на два, присоединение одного списка к другому, сортировку списка и создание списка из содержимого базы данных. Настоящий раздел посвящен технике программирования этих операций.

5.5.1 Поиск элемента в списке

Поиск элемента в списке является очень распространенной операцией. Поиск представляет собой просмотр списка на предмет выявления соответствия между элементом данных (объектом поиска) и элементом просматриваемого списка. Если такое соответствие найдено, то поиск заканчивается успехом. В противном случае поиск заканчивается неуспехом. Результат поиска, так же как и результат любой другой операции Турбо-Пролога, базирующейся на унификации термов, всегда бывает либо успехом, либо неуспехом. Для сопоставления объекта поиска с элементами просматриваемого списка необходим предикат, объектами которого и являются эти объект поиска и список:

find_it(3 ,[1,2,3,4,5]).

Первый из объектов утверждения, 3, есть объект поиска. Второй - это список [1,2,3,4,5].

Для выделения элемента из списка и сравнения его с объектом поиска можно применить метод разделения списка на голову и хвост. Стратегия поиска при этом будет состоять в рекурсивном выделении головы списка и сравнении ее с элементом поиска.

Так же как в программе "Голова-хвост", при рекурсии хвостом каждый раз становится новый список, голова которого присваивается переменной, сравниваемой с объектом поиска. Правило поиска может сравнить объект поиска и голову текущего списка. Саму операцию сравнения можно записать в виде

find_it(Head,[Head|_]).

Этот вариант правила предполагает наличие соответствия между объектом поиска и головой списка. Отметим, что хвост списка при этом присваивается анонимной переменной. В данном случае, поскольку осуществляется попытка найти соответствие между объектом поиска и головой списка, то нет необходимости заботиться о том, что происходит с хвостом. Если объект поиска и голова списка действительно соответствуют друг другу, то результатом сравнения явится успех. Если нет, то неуспех. Другими словами, утверждение удовлетворяется в случае совпадения объекта поиска и головы списка. Если же эти два элемента данных различны, то попытка сопоставления считается неуспешной, происходит откат и поиск другого правила или факта, с которыми можно снова попытаться найти соответствие. Для случая несовпадения объекта поиска и головы списка, следовательно, необходимо предусмотреть правило, которое выделило бы из списка следующий по порядку элемент и сделало бы его доступным для сравнения. Поскольку следующий за головой текущего списка элемент является головой текущего хвоста, мы можем представить этот текущий хвост как новый список, голову которого можно сравнить с объектом поиска:

```
find_it(Head, [Head|_].
find_it(Head, [_,Tail]) :-
        find_it(Head, Tail).
```

Если правило ***find_it(Head,[Head|_])*** неуспешно, то происходит откат, и делается попытка со вторым вариантом ***find_it.***

На этом втором вхождении предиката ***find_it*** Турбо-Пролог унифицирует имеющиеся термы с заголовком правила ***find_it([Head,[_,Rest]).*** Заметим, что при этом первый элемент списка ставится в соответствие анонимной переменной. Так делается вследствие того, что значение первого элемента не представляет для нас интереса; данное правило не было бы задействовано, если бы этот элемент совпадал с объектом поиска при попытке с ***find_it(Head,[Head,_]).*** Теперь мы хотим присвоить переменной хвост списка (не голову !), чтобы Турбо-Пролог попытался установить соответствие между объектом поиска и головой списка хвоста. Попытка удовлетворить рекурсивное правило ***find_it(Head,Rest)*** заставляет Турбо-Пролог представить хвост текущего как новый самостоятельный список. Опять присвоенный переменной ***Rest*** список разделяется на голову и хвост при посредстве утверждения ***find_it(Head, [Head|_]).*** Процесс повторяется до тех пор, пока это утверждение дает либо успех в случае установления соответствия на очередной рекурсии, либо неуспех в случае исчерпания списка.

Программа "Элементы" (листинг 5.3) демонстрирует реализацию операции поиска элемента в списке. Поскольку предикат ***find_it*** определен как для списков целых чисел, так и для списков символических имен, то в данной программе он и работает со списками обоих типов.

Листинг 5.3

```
/* Программа:  Элементы       Файл: PROG0503.PRO  */
/* Назначение:  Поиск нужного элемента в списке.    */
        domains
                number_list = number *
                        number = integer
                member_list = member *
                        member = symbol

        predicates
                find_it(number, number_list)
                find_it(member, member_list)

        clauses
                find_it(Head, [Head|_]).
                find_it(Head, [_|Tail]) :-
                        find_it(Head, Tail).
        /*****                  конец программы          *****/
```

Если задать цель

```
find_it(3,[1,2,3,4,5])
```

то первый вариант правила пытается установить соответствие между головой списка, 1, и объектом поиска, 3. Вследствие неравенства 1 и 3 результа-

том применения этого правила является неуспех. Процесс установления соответствия продолжается со следующей головой списка (уже усеченного), 2, и снова неуспешно. При следующей попытке голова списка, а вместе с ней и объект поиска, равны 3 - успешное завершение процесса. На экране появляется True, что как раз указывает на успешное завершение процесса установления соответствия, то есть на присутствие числа 3 в списке.

Цель

find_it(1,[2,3,4,5]).

дает неуспех, так как элемент 1 в списке отсутствует. Цель

find_it("Alice",["Diana","Peter","Paul","Mary","Alice"]).

дает успех, так как список содержит элемент ***Alice***. Цель

find_it("Toledo",["Cleveland","Dayton","Chardon",
"Youngstown","Cincinnati"]).

очевидно, также неуспешна.

* Упражнение

5.11. Нарисуйте диаграмму поиска для следующей внешней цели:
find_it(44,[11,22,33,44,11,22,33,44,11,22,33,44,55]).
В скольких случаях будет достигнут успех ?

5.5.2 Деление списков

При работе со списками достаточно часто требуется разделить список на несколько частей. Это бывает необходимо, когда для целей текущей обработки нужна лишь определенная часть исходного списка, а оставшуюся часть нужно на время оставить в покое. Сейчас Вы увидите, что деление списков на части является достаточно простой операцией.

Для пояснения сказанного рассмотрим предикат ***split,*** аргументами которого являются элемент данных и три списка:

split(Middle,L,L1,L2).

Элемент ***Middle*** здесь является компаратором, ***L*** - это исходный список, а ***L1*** и ***L2*** - подсписки, получающиеся в результате деления списка ***L***. Если элемент исходного списка меньше или равен ***Middle***, то он помещается в список ***L1***; если больше, то в список ***L2.***

Предположим, что вначале значением переменной ***Middle*** является число 40, переменной ***L*** присвоен список [30,50,20, 25,65,95], а переменные ***L1*** и ***L2*** не инициализированы.

split(40,[30,50,20,25,65,95],L1,L2).

Правило для разделения списка должно быть написано таким образом, чтобы элементы исходного списка, меньшие либо равные 40, помещались в список ***L1,*** а большие 40 - в список ***L2***.

Правило устроено следующим образом: очередной элемент извлекается из списка при помощи метода разделения списка на голову и хвост, а потом сравнивается с компаратором ***Middle.*** Если значение этого элемента меньше или равно значению компаратора, то элемент помещается в список ***L1***, в противном случае - в список ***L2***.

В результате применения правила к списку [30,50,20,25, 65,95] значениями списков ***L1*** и ***L2*** станут соответственно [30, 20,25] и [50,65,95].

Само правило для разделения списка записывается в Турбо-Прологе следующим образом:

```
split(Middle,[Head|Tail],[Head|L1],L2) :-
                                        Head <= Middle,
                                        split(Middle,Tail,L1,L2).
split(Middle,[Head|Tail],L1,[Head|L2]) :-
                                        split(Middle,Tail,L1,L2),
                                        Head > Middle.
split(_,[],[],[]).
```

Отметим, что метод деления списка на голову и хвост используется в данном правиле как для разделения исходного списка, так и для формирования выходных списков.

Приведенное правило годится для любых допустимых в Турбо-Прологе типов данных. Если список состоит из целых чисел, то тогда нужно элементы списка и компаратор описать как целые. Если же Вы имеете дело со списком символических имен, то элементы списка и компаратор должны относиться к типу ***symbol.***

Программа "Деление списка" (листинг 5.4) включает в себя только что приведенное правило. Попробуйте ввести такое целевое утверждение:

```
split(40,[30,50,20,25,65,95],L1,L2).
```

Листинг 5.4

```
/* Программа:  Деление списка                          */
/* Назначение: Разделение списка на два.          */

domains
        middle = integer
        list = integer *

predicates
        split(middle,list,list,list)

clauses
        split(Middle,[Head|Tail],[Head|L1],L2) :-
                        Head <= Middle,
                        split(Middle,Tail,L1,L2).
        split(Middle,[Head|Tail],L1,[Head|L2]) :-
                        split(Middle,Tail,L1,L2),
                        Head > Middle.
        split(_,[],[],[]).
/*****       конец программы          *****/
```

---

* Упражнение

5.12. Для программы "Деление списка"

а) задайте внешнюю цель

split(12,[96,32,8,16,55,12],L1,L2).

Как будут выглядеть списки L1 и L2 ?

б) нарисуйте диаграмму изменения значений списков в процессе работы программы.

5.5.3 Присоединение списка

Слияние двух списков и получение таким образом третьего принадлежит к числу наиболее полезных при работе со списками операций. Этот процесс обычно называют присоединением одного списка к другому. Метод, представленный в данном разделе, особенно часто используется в таких приложениях, каковыми являются системы управления базами данных и разработка интерфейсов пользователя. В сущности, Вы, вероятно, найдете его необходимым для большинства программ Турбо-Пролога, требующих преобразования списков.

В качестве примера рассмотрим две переменные, ***L1*** и ***L2***, представляющие списки. Переменные имеют значения [1,2,3] и [4,5]. Назовем их входными списками. Предикат, присоединяющий ***L2*** к ***L1*** и создающий выходной список ***L3***, в который он должен переслать все элементы ***L1*** и ***L2***. Весь процесс можно представить себе в виде такой совокупности действий:

1. Список ***L3*** вначале пуст.

2. Элементы списка ***L1*** пересылаются в ***L3***, теперь значением ***L3*** будет [1,2,3].

3. Элементы списка ***L2*** пересылаются в ***L3***, в результате чего тот принимает значение [1,2,3,4,5].

Структура правила для выполнения этих действий достаточна проста:

```
append([],L,L).
append([N|L1],L2,[N|L3]) :-
                        append(L1,L2,L3).
```

Поясним теперь, как будет функционировать это правило, если на вход подать списки ***L1=[1,2,3]*** и ***L2=[4,5].***

Сначала Турбо-Пролог пытается удовлетворить первый вариант правила:

append([],L,L).

Для того чтобы удовлетворить это правило, первый объект предиката ***append*** нужно сделать пустым списком. Вначале же предикат ***append*** имеет форму

append([1,2,3],[4,5],_).

Отметим, что третий, выходной список в этой форме пока еще не определен. Внутренний процесс унификации Турбо-Пролога, пытаясь удовлетворить второе правило ***append,*** раскручивает цепочку рекурсий до тех пор, пока не обнуляет первый список. Элементы списка при этом последовательно

пересылаются в стек. Стек, логическая структура данных в памяти компьютера, обеспечивает временное хранение этих элементов.

Когда первый объект предиката ***append*** окажется пустым списком, становится возможным применение первого варианта правила. Третий список при этом инициализируется вторым. Такой процесс можно пояснить при помощи двух состояний ***append,*** до и после применения первого варианта правила:

append([],[4,5],_).
append([],[4,5],[4,5]).

В данный момент процедуры унификации Турбо-Пролога полностью удовлетворили это правило, и Турбо-Пролог начинает сворачивать рекурсивные вызовы второго правила:

append([N|L1], L2, [N|L3]) :-
append(L1,L2,L3).

Извлекаемые при этом из стека элементы помещаются один за другим в качестве головы к первому и третьему спискам. Следует особо отметить, что элементы извлекаются в обратном порядке (это стек !), и что значение извлеченного из стека элемента присваивается переменной ***N*** одновременно в **[N|L1]** и **[N|L3].**

Шаги данного процесса можно представить так:

append([],[4,5],[4,5])
append([3],[4,5],[3,4,5])
append([2,3],[4,5],[2,3,4,5])
append([1,2,3],[4,5],[1,2,3,4,5])

Присвоение элементов стека происходит рекурсивно до тех пор, пока стек не будет исчерпан. В результате список L3 будет содержать элементы обоих входных списков - [1,2,3,4,5].

Программа "Присоединение списка" (листинг 5.5) демонстрирует применение данного метода. В этой программе ***n_list*** является доменом списков целых чисел. Описание предиката для присоединения одного списка к другому записано в виде

append(n_list,n_list,n_list)

Раздел ***clauses*** содержит уже приведенные описания правил ***append.*** Внешней целью для программы может служить, скажем, ранее разобранный пример:

append([1,2,3],[4,5],L).

Листинг 5.5

```
/* Программа:  Присоединение списка                */
/* Назначение: Слияние двух списков.               */
    domains
        n_list = integer *

    predicates
```

```
        append(n_list,n_list,n_list)

    clauses
        append([],L,L).
        append([N|L1], L2, [N|L3]) :-
                                append(L1,L2,L3).
    /*****        конец программы        *****/
```

Программу "Присоединение списка" можно смело отнести к числу весьма эффективных. Написаны только две строки, однако Турбо-Пролог в своих недрах создает временные списки, перемещает элементы из одного в другой. Таким образом вся кухня определенно остается за сценой. Программисту нет необходимости специфицировать все эти действия.

Правила присоединения списка являются важным инструментом программирования. Способ работы этих правил на первых порах может несколько смутить Вас, поэтому стоит попрактиковаться при помощи приведенных ниже упражнениями.

* Упражнения

5.13. Запустите программу "Присоединение списка" и введите целевое утверждение

append([9,15,3,60,55],[15,2,21],L).

Что получится ?

5.5.4 Сортировка списков

Сортировка представляет собой переупорядочивание элементов списка определенным образом. Назначением сортировки является упрощение доступа к нужным элементам. Сортировка важна как в реальной жизни, так и в применениях вычислительной техники. Сортируются фамилии в телефонном справочнике, сортируется по номерам информация в отчетах по соцобеспечению, почта сортируется по индексу, маршруту доставки, номеру дома. Сортировка данных при помощи компьютера является рутинной, но важной работой. Гораздо проще и гораздо эффективнее искать что-либо в отсортированном списке, нежели в неотсортированном.

Существует много способов сортировки списков. Рассмотрим, например, список целых чисел:

[51,23,84,17,11]

Элементы этого списка не расположены в каком-то определенном порядке в обычном понимании этого слова. Тот же самый список, но уже отсортированный в порядке возрастания элементов, выглядит так:

[11,17,23,51,84]

Сортирующее правило Турбо-Пролога принимает на вход неотсортированный список, и выдает отсортированный на выходе. Входной список называется исходным, выходной - целевым.

Три метода обычно применяются для сортировки списков: метод перестановки, метод вставки и метод выборки. Сортировку можно произвести любым из трех названных методов или их комбинацией. Первый из перечисленных методов заключается в перестановке элементов списка до тех пор, пока они не выстроятся в нужном порядке. Второй осуществляется при помощи неоднократной вставки элементов в список до тех пор, пока он не будет упорядочен. Третий включает в себя многократную выборку и перемещение элементов списка.

Второй из методов, метод вставки, особенно удобен для реализации на Турбо-Прологе. Сейчас мы зададимся целью написать правило, реализующее этот метод.

Рассмотрим список [4,7,3,9], элементы которого расположены случайным образом. Мы хотим получить список [3,4,7,9], упорядоченный по порядку возрастания элементов.

Опишем предикат, производящий нужную сортировку списка методом вставки.

insert_sort(source_list,target_list)

Внешней целью в задаче сортировки списка [4,7,3,9] будет утверждение

insert_sort([4,7,3,9],S).

В этом утверждении отсортированный список обозначен переменной ***S***. Для того, чтобы воспользоваться преимуществами мощного средства Турбо-Пролога - внутренней унификацией, мы проделаем то, что обычно называется сортировкой хвоста. Напомним, что список всегда можно разбить на голову и хвост. Перед тем, как первый элемент списка будет присвоен голове списка, его хвост в действительности содержит весь список целиком. Следовательно, сортировка хвоста есть не что иное, как сортировка самого списка.

Правила, реализующие этот способ сортировки, имеют следующую структуру:

```
insert_sort([],[]).
insert_sort([X|Tail],Sorted_list) :-
                        insert_sort(Tail,Sorted_Tail),
                        in-sert(X,Sorted_Tail,Sorted_list).
insert(X,[Y|Sorted_list],[Y|Sorted_list1]) :-
                        asc_order(X,Y), !,
                        insert(X,Sorted_list,Sorted_list1).
                        insert(X,Sorted_list,[X|Sorted_list]).
asc_order(X,Y) :- X>Y.
```

Дальнейшее обсуждение продемонстрирует работу правил на примере списка [4,7,3,9].

Вначале Турбо-Пролог применяет приведенные правила к исходному списку, выходной список в этот момент еще не определен.

insert_sort([4,7,3,9],_).

Первая попытка удовлетворить правило ***insert_sort*** осуществляется с первым вариантом правила

insert_sort([],[]).

Для удовлетворения правила оба списка должны быть пустыми. Второй вариант правила ***insert_sort*** трактует список как комбинацию головы списка и его хвоста. Внутренние унификационные процедуры Турбо-Пролога пытаются сделать пустым исходный список. Устранение элементов списка начинается с головы списка и осуществляется рекурсивно.

Та часть правила, которая производит удаление элементов, выглядит так:

insert_sort([X|Tail],Sorted_list) : - insert_sort(Tail, Sorted_Tail).

По мере того как Турбо-Пролог пытается удовлетворить первое из правил, происходят рекурсивные вызовы ***insrt_sort***, при этом значениями ***X*** последовательно становятся все элементы исходного списка, которые затем помещаются в стек. В результате применения этой процедуры список становится нулевым. Теперь первый вариант правила ***insert_sort*** производит обнуление выходного списка, и таким образом правилу придается форма

insert_sort([],[]).

Далее, после того как удовлетворен первый вариант правила ***insert_sort***, Турбо-Пролог пытается удовлетворить второе правило из тела ***insert_sort*** - правило ***insert***. Переменной ***X*** сначала присваивается первое взятое из стека значение, 9, а правило ***insert*** принимает форму

insert(9,[],[9]).

Теперь удовлетворено и второе правило из тела ***insert_sort*** - правило ***insert*** (его второй вариант), поэтому происходит возврат на один круг рекурсии ***insert_sort***. Из стека извлекается следующий элемент, 3, и испытывается первый вариант ***insert,*** а значит и правило упорядочивания

asc_order(X,Y) :- X>Y.

Переменная **X** здесь имеет значение 3, ***Y*** - значение 9, само правило имеет вид

asc_order(3,9) :- 3>9.

Так как оно неуспешно (3 на самом деле не больше, а меньше 9), то вместе с ним неуспешен и первый вариант ***insert.*** При помощи второго варианта ***insert*** 3 вставляется в выходной список слева от 9:

insert(3,[9],[3,9]).

Происходит возврат к ***insert_sort***, теперь оно имеет форму

insert_sort([3,9],[3,9]).

На следующем круге рекурсии происходит вставка очередного элемента из стека, а именно 7. В начале работы на этом круге правило ***insert*** имеет вид

insert(7,[3,9],_).

Сначала происходит сравнение 7 и 3,

asc_order(7,3):- 7>3.

Так как данное правило успешно, то элемент 3 убирается в стек, и insert вызывается рекурсивно еще раз, но уже с хвостом списка - [9] :

insert(7,[9],_).

Так как правило

asc_order(7,9):- 7>9.

неуспешно, то испытывается второй вариант ***insert*** (успешно), происходит возврат на предыдущие круги рекурсии сначала ***insert***, а потом ***insert_sort.*** В результате 7 помещается в выходной список между элементами 3 и 9.

insert(7,[3,9],[3,7,9]).

При возврате еще на один круг рекурсии ***insert_sor***t из стека извлекается элемент 4. Правило ***insert*** выглядит как

insert(4,[3,7,9],_).

Далее имеем: правило

asc_order(4,3) :- 4>3.

успешно, следовательно, следует попытка

insert(4,[7,9],_).

Правило же

asc_order(4,7) :- 4>7.

неуспешно. Нетрудно сообразить теперь, что 4 окажется в выходном списке между 3 и 7 :

insert(4,[3,7,9],[3,4,7,9]).

insert_sort([4,7,3,9],[3,4,7,9]).

Теперь в стеке больше нет элементов, и все рекурсии ***insert_sort*** уже свернуты. Выходной список получил при этом значение [3,4,7,9], удовлетворена цель программы.

Программа "Сортировка списка" (листинг 5.6) реализует правило сортировки при помощи вставок для целочисленных списков. Программа работает в интерактивном режиме.

Листинг 5.6

```
/* Программа:   Сортировка списка                    */
/* Назначение:  Сортировка списка целых чисел.       */
/*              в порядке возрастания при помощи     */
domains
        number = integer
        list   = number *

predicates
        insert_sort(list,list)
        insert(number,list,list)
        asc_order(number,number)

clauses
        insert_sort([],[]).
        insert_sort([X|Tail],Sorted_list) :-
                                        insert_sort(Tail,Sorted_Tail),
                                        insert(X,Sorted_Tail,Sorted_list).
```

```
        insert(X,[Y|Sorted_list],[Y|Sorted_list1]) :-
        asc_order(X,Y), !, insert(X,Sorted_list,Sorted_list1).
        insert(X,Sorted_list,[X|Sorted_list]).
        asc_order(X,Y) :- X>Y.
/*****           конец программы           *****/
```

Запустите программу на счет с целевыми утверждениями

insert_sort([4,7,3,9],S).

и

insert_sort([7,6,5,4,3,2,1],S).

* Упражнения

5.15. Запустите программу "Сортировка списка". Введите цель

insert_sort([53,11,93,77,11],S).

Что получится ?

5.16. Внесите в программу такие изменения, чтобы она могла сортировать целые числа в порядке убывания. Запустите на счет новый вариант программы. Задайте следующее целевое утверждение:

insert_sort([1,2,3,4,5,6,7],S).

Что получится ?

5.6 Компоновка данных в список

Иногда, при программировании определенных задач, возникает необходимость собрать данные из базы данных в список для последующей их обработки. Турбо-Пролог содержит встроенный предикат, позволяющий справиться с этой задачей без каких бы то ни было хлопот. Таким предикатом является предикат ***findall.*** Требуемый список представляется означенной переменной, являющейся одним из объектов предиката.

Предописание встроенного предиката ***findall*** выглядит следующим образом:

findall(Variable_name,Predicate_expression,List_name).

Variable_name обозначает здесь объект входного предиката ***Predicate_expression***, а ***List_name*** является именем переменной выходного списка. Переменная должна относиться к домену списков, объявленному в разделе ***domains.***

Для пояснения только что сказанного рассмотрим предикат базы данных

football(name,points)

Этот предикат порождает 5 утверждений :

ootball("Ohio State",116.0). football("Michigan",121.0).
football("Michigan State",114.0). football("Purdue",99.0).
football("UCLA",122.0).

Эти утверждения содержат сведения об очках, набранных командами. Предположим, что необходимо сложить все очки и усреднить их. Сбор очков в список осуществляется при помощи встроенного предиката ***findall :***

findall(Points,football(_,Points),Point_list)

Здесь ***Poits*** является свободной переменной для значений набранных командами очков, а ***Point_list*** - списочной переменной, элементы которой принадлежат к тому же домену, что и ***Points,*** в данном случае, к домену ***real.***

Сама работа предиката скрыта от глаз пользователя. В нашем примере ***findall*** просматривает все утверждения с предикатом ***football,*** начиная с первого. Значение переменной ***Points*** (116), взятое из этого утверждения, присваивается голове списка ***Point_list***. Остальные значения ***Points*** помещаются в список на последующие позиции. По завершению работы ***findall*** переменная ***Point_list*** принимает значение

[116.0,121.0,114.0,99.0,122.0]

Для подсчета среднего значения набранных очков применяется рекурсивное правило

```
sum_list([],0,0).
sum_list([H|T], Sum, Number) :-
                        sum_list(T,Sum1,Number1),
                        Sum = H + Sum1,
                        Number = Number1 + 1.
```

Оно напоминает правило ***sum*** из гл. 4.

Для получения суммы всех элементов списка ***Point_list*** это правило необходимо задать в качестве подцели

sum_list(Point_list,Sum,Number).

Сначала Турбо-Пролог сопоставляет эту подцель с вариантом правила ***sum_list([H|T],Sum,Number). Point_list*** при этом сопоставляется с **[H|T],** а переменные ***Sum*** и ***Number*** оказываются неопределенными. Рекурсии с первым вариантом правила продолжаются до тех пор, пока ***Point_list*** не превратиться в нулевой список; на каждой рекурсии очередной элемент списка помещается в стек. Теперь Турбо-Пролог пытается удовлетворить правило

sum_list([],0,0).

Переменным ***Sum*** и ***Number*** присваиваются нули, и таким образом правило полностью удовлетворено. Сворачивая рекурсию ***sum_list***, Турбо-Пролог последовательно, один за другим, извлекает из стека засланные туда элементы и складывает их с уже имеющимся значением суммы; при этом переменная ***Number*** каждый раз увеличивается на единицу. В итоге имеем следующие формы ***sum_list***:

```
sum_list([122],122,1)
sum_list([99,122],221,2)
sum_list([114,99,122],335,3)
sum_list([121,114,99,122],456,4)
sum_list([116,121,114,99,122],572,5)
```

По окончанию рекурсий значениями переменных ***Sum*** и ***Number*** являются соответственно 572 и 5.

Совсем просто выглядит правило для нахождения среднего значения набранных очков:

Average = Sum / Number

Правило ***Average*** использует подсчитанное правилом ***Sum_list*** значения переменных ***Sum*** и ***Number***. Результатом применения этого правила является присвоение переменной ***Average*** частного от деления ***Sum*** и ***Number,*** т. е. 114.4.

Программа "Очки" (листинг 5.7) демонстрирует использование предиката ***findall*** для сбора данных из базы данных в список.

Листинг 5.7

```
/* Программа:   Очки                                          */
/* Назначение:  Показ использования предиката                 */
/*              findall для вычисления среднего значения.     */
domains
        name = string
        points = real
        list = points *

predicates
        football(name,points)
        sum_list(list,points,integer)
        report_average_football_score

goal
        report_average_football_score.

clauses
        /* факты (футбольная база данных) */
        football("Ohio State",116.0).
        foot-ball("Michigan",121.0).
        football("Michigan State",114.0).
        football("Purdue",99.0).
        football("UCLA",122.0).
        report_average_football_score:-
                        findall(Points,football(_,Points),Point_list),
                        sum_list(Point_list,Sum,Number),
                        Average = Sum / Number,
                        write("College Football Power Rating:"),
                        nl,
                        write("     Average Points = ",Average).
        sum_list([],0,0).
        sum_list([H|T], Sum, Number) :-
                        sum_list(T,Sum1,Number1),
                        Sum = H + Sum1,
```

```
                    Number = Number1 + 1.
/*****                конец программы          *****/
```

Программа подсчитывает сумму очков команд и их среднее значение. Внутренняя цель программы есть

report_average_football_score

Цель представляет собой правило, содержащее подцели ***findall, sum_list, Average***, а также предикаты, осуществляющие вывод полученных результатов в нужной форме.

Начиная свою работу программа, пытается удовлетворить подцель ***findall*** в том виде, в котором она была описана. Когда подцель удовлетворена, делается попытка удовлетворить подцель ***sum_list***, а затем ***Average***. Переменной ***Average*** при этом присваивается значение 114.4, которое используется затем предикатом ***write***. Теперь все подцели удовлетворены, следовательно, удовлетворена и цель программы.

* Упражнение

5.17. Возьмите текущею таблицу чемпионата СССР по футболу, введите в базу данных "Очки" результаты лучших десяти команд. Запустите программу на счет. Каким будет средний результат десятки ?

5.7. Заключение

В настоящей главе были представлены структура и функции списков в Турбо-Прологе, а также различные операции для работы с ними. Рассмотрение работы со списками началось с операций создания и печати списков при посредстве метода деления списка на голову и хвост. Завершилось это операциями поиска нужного элемента списка, деления, присоединения и сортировки списка. Затрагивался вопрос об использовании предиката ***findall*** для компоновки списка из данных базы данных.

Читая главу, вы познакомились с приемами создания правил, позволяющих описать цель программы. Вы научились превращать сформулированные на естественном языке запросы к программе в правила Турбо-Пролога.

Рекомендуется читателю не пропускать предлагаемые упражнения, которые призваны углубить понимание основных структур и техники программирования. Упражнения помогают также научиться модифицировать демонстрационные программы с целью приспособить их к своим нуждам. Семь законченных программ данной главы показывают основные методы работы со списками и отличительные черты Турбо-Пролога. Их можно использовать в качестве заготовок для ваших собственных программ.

Чтение главы является обязательным, если вы собираетесь использовать списки в ваших программах, так как представленные здесь методы находят самое широкое применение при разработке программ на Турбо-

Прологе. Эти методы можно применять даже в том случае, если вы не понимаете до конца механизм их работы, хотя, конечно, понимание сделает Вам честь как программисту.