РОЖДЕСТВЕНСКИЙ О.Н.

TURBO PROLOG 2.0

(пособие для учащихся)

РОЖДЕСТВЕНСКИЙ О.Н.

TURBO PROLOG 2.0

(Пособие для учащихся)

Сумы – 2010

ОГЛАВЛЕНИЕ

1. ЗНАКОМСТВО С СИСТЕМОЙ ПРОГРАММИРОВАНИЯ TURBO PROLOG 2.0

1.1. ИСКУССТВЕННЫЙ ИНТЕЛЛЕКТ И ТЕХНОЛОГИЯ ЗНАНИЙ

В связи с развитием вычислительной техники и информатики возникла необходимость и появились возможности при помощи компьютеров имитировать интеллектуальную деятельность.

Искусственный интеллект (artificial intelligence, machine intelligence) - это область научных исследований разумного поведения и искусственного моделирования его.Исследователи ставят задачу с помощью теорий и моделей научиться понимать принципы и механизмы интеллектуальной деятельности.

Практической целью является создание методов и техники, необходимой для программирования “разумности” и ее передачи вычислительным машинам, а через них всевозможным системам и средствам.

Технология знаний (knowledge engineering)-это инженерные методы и навыки в области искусственного интеллекта. Технология знаний - это работающий искусственный интеллект.

Технология знаний непосредственно применяется к решению сложных проблем, направленных на увеличение производительности и качества умственного труда, на поднятие профессиональных навыков работника. Продуктом технологии знаний являются активные знания и навыки работы со знаниями, которые отличаются от пассивных знаний тем, что их использование (в оптимальном варианте) не предполагает чтения и освоения всех необходимых для решения проблемы знаний.

1.2. ПОДХОДЫ К СОЗДАНИЮ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Идеи создания и использования искусственного интеллекта были выдвинуты еще в середине 50х годов в различных научных лабораториях.В этом направлении было определено несколько подходов.

Первый подход(восходящий метод)-моделирование технических средств, имитирующих структуру и функционирование биологических клеток живого интеллекта.Создано и создается большое количество интеллектуальных роботов, в настоящее время появляются новые технологии для реализации этих идей, но успехов покрывающих затраты на реализацию этих идей пока не достигнуто.При настоящем уровне развития техники данное направление еще неперспективно.

Второй подход(низходящий метод)-предусматривает создание программ, наделенных искусственным интеллектом, воплощающих в себе, умения, аналогичные человеческим.Важнее имитировать не саму живую клетку естественного интеллекта, а ее способности.

На начальном этапе развития этого направления, на него возлагались, пожалуй, слишком большие надежды.Исследования были направлены на разработку механизмов решения общих проблем произвольного характера.Такой подход в данном направлении не привел к успеху. Из-за слишком самонадеянных целей возникли разочарование и предубеждения по отношению к самой области и к занятым в ней исследователям.Например, в 60е годы в США официально были признаны бесперспективными исследования в области машинного перевода.

С созданием в конце 60х и в 70х годах первых экспертных систем (DENDRAL, MACSYMA, MYCIN и других) было замечено, что сосредотачиваясь на решении проблемы в некоторой узкой области и пытаясь ее решить с помощью знаний из этой специальной области, а не с помощью общих механизмов принятия решений, можно достичь успеха.

В 80е годы были широко осознаны большие потенциальные возможности искусственного интеллекта как в исследованиях, так и в развитии производства.В рамках новой технологии появились первые коммерческие программные продукты и в различных странах были начаты крупнейшие в истории об-

работки данных национальные и международные исследовательские проекты, нацеленные на интеллектуальные вычислительные машины “пятого поколения”.

Искусственный интеллект рожден в области вычислительной техники, но как наука он находится на пересечении информатики, математической логики, языкознания, психологии, философии. В сферах применения искусственного интеллекта используются также и конкретные специальные знания из соответствующей области, например из области естественных наук, медицины, юриспруденции, экономики и других наук.

Искусственный интеллект и технологию знаний следует рассматривать как расширение и обобщение традиционной информатики и вычислительной техники, а не как альтернативу им.

1.3. ОСОБЕННОСТИ ПРОГРАММ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Программы искусственного интеллекта отличаются от традиционных программ тем, что в них обрабатываются данные, представленные в символьном, а не в числовом виде.Исследователи в этой области заметили уже в 50е годы, что присущие человеческому мышлению и языку или реальному миру вещи, объекты, ситуации, случаи и другие понятия нельзя естественным образом отобразить в виде чисел и массивов.В программах искусственного интеллекта нужна возможность работать с данными и знаниями, представленными в символьном виде или в виде символьных структур.

В обработке символьной информации важна не только форма рассматриваемых знаний, но и их содержание и значение.Если в численных вычислениях внимание уделяется количественной стороне и численным значениям данных, то в символьной обработке важна качественная сторона данных, т.е. особенности их строения и их функциональные особенности. Для численных вычислений характерно большое количество простых действий (вычисление, присваивание, вывод) над большим числом простых элементов (числа, строки и т.д.).А в программах искусственного интеллекта обрабатываются большие и сложные структуры данных и описываются сложные действия, но количество вводимых и выводимых данных в большинстве случаев невелико, например предложение на естественном языке, описание деятельности и т.д.

Другое отличие программирования искусственного интеллекта от традиционного состоит в том, что в программах искусственного интеллекта проблема решается часто эвристически(heuristic), в то же время традиционное программирование основывается главным образом на алгоритмическом (algorithmic) способе.

При алгоритмическом решении задачи в различных ситуациях, требующих принятия решения, как правило, достаточно информации, чтобы сделать верный выбор.Решение будет найдено всегда, когда оно вообще возможно.

Но когда имеющиеся в наличии данные не достаточны или не внушают доверия или когда другие способы не пригодны для принятия решения, приходится прибегать к эвристическим методам.Они обычно основываются на связанных с задачей специальных знаниях, простейших правилах, интуитивных критериях, базирующихся на предыдущем опыте и на других ненадежных методах вплоть до угадывания. Если выводы, сделанные на основе имеющихся знаний, признаются ошибочными, то они аннулируются, полученные результаты анализируются и решение задачи продолжается каким-нибудь другим способом. Эвристические методы не всегда приводят к достижению цели, даже когда решение существует, или они могут привести к неверному решению.В этом программы искусственного интеллекта подобны человеку.

Для многих задач искусственного интеллекта алгоритмические решения вообще неизвестны.К эвристикам обращаются в том случае, когда специалисты не в состоянии представить точные данные или когда их трактовки отличаются или даже противоречат друг другу. К тому же сущесвуют задачи, для которых можно показать, что они не имеют алгоритмического решения.

Эвристики используют также и в связи с ограничениями налагаемыми вычислительной техникой.Их используют, когда средства и методы решения

задачи могут быть известны, но их использование потребовало бы большого объема работы.Когда время или какой-нибудь другой параметр алгоритмической процедуры неприемлимо велик.

В программировании задач искусственного интеллекта используются все алгоритмические методы, которые только возможны, и таким образом пытаются найти алгоритмическое решение проблемы.Однако такая попытка не должна стать препятствием на пути решения. Во многих случаях на основе эвристических решений создаются алгоритмы.Естественно, и эвристические программы в своей реализации базируются на алгоритмах.

Эвристическое решение задач и символьные вычисления характеризуют почти все исследования и методы программирования в области искусственного интеллекта.

1.4. ПРИМЕНЕНИЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Потенциальные возможности применения искусственного интеллекта и технологии знаний кажутся почти безграничными.Эта технология используется, когда необходимо передать компьютеру какие-нибудь способности к рассуждению, разумность, мудрость, творческие или другие свойственные человеку способности и когда нужно создать для них действующую теорию или модель.

Исследования по искусственному интеллекту часто классифицируются, исходя из области их применения.В каждой из этих областей на протяжении десятков лет разрабатывались свои методы программирования, формализмы;каждой из них присущи свои традиции, которые могут заметно отличаться от традиций соседней области исследования.

ОБРАБОТКА ЕСТЕСТВЕННОГО ЯЗЫКА (natural language processing)-позволяет вести диалог между машиной и человеком на обычном или близком ему языке. Это необходимо, поскольку с развитием автоматизации все большее число людей должны будут иметь дело со все более часто встречающимися системами обработки данных и их применениями.

Исследования в этой области относятся к общему языкознанию (linguistics) и к математической лингвистике(computational linguistics).

Областью применения обработки естественного языка являются:

- системы, синтезирующие (speech synthesis) и распознающие речь (speech recognition);
- распознаватели ошибок написания;
- интерфейсы пользователя (human interface) информационных, экспертных и других систем обработки данных;
- машинный перевод (machin translation);
- автоматическая интерпретация или генерация документов;
- обучение языку (language learning).

ЭКСПЕРТНЫЕ СИСТЕМЫ И ИХ ОБОЛОЧКИ (expert systems)-это программы выполняющие роль эксперта в некоторой области деятельности человека.

Экспертные системы обладают следющими свойствами:

- Степень разумности и объем знаний. Системы могут решить особенно сложные задачи в некоторых случаях, лучше, чем эксперты-люди.
- Развитый интерфейс пользователя. С такими системами можно, например, общаться на ограниченном естественном языке(в письменной форме).
- Способность обосновывать действия. У многих систем можно спросить, почему они задали некоторый вопрос или на основании чего они пришли к некоторому заключению.Таким образом пользователь может наблюдать за процессом рассуждений и разумностью решений.
- Эвристики и работа с нечеткими данными. Системы часто в состоянии обрабатывать неполные, нечеткие, недостаточные или даже ошибочные данные и знания.
- Постепенное накопление знаний. Работу системы можно улучшить, добавив порцию знаний.
- Работа с информацией представленной в символьном виде.

Диапазон областей применения экспертных систем в настоящее время довольно широк.

Примеры областей применения экспертных систем:

- оценка займов, рисков страхования и капитальных вложений для финансовых организаций;
- помощь химикам в нахождении верной последовательности реакций для создания новых молекул;
- отладка программного и аппаратного обеспечения компьютеров в соответствии с индивидуальными требованиями заказчика;
- диагностика и обнаружение неполадок в электрических цепях на основании тестов;
- помощь геологам в расшифровке данных о месторождениях;
- помощь медикам в постановке диагноза и лечения некоторых групп заболеваний, таких, как заболевания крови, различные виды рака, некоторых инфекционных заболеваний;
- помощь навигаторам в расшифровке данных от подводных звукоулавливателей;
- получение молекулярной структуры химического вещества на основании опытов;
- изучение телеметрических данных с искусственных спутников.

Примеры наиболее популярных экспертных систем:

- XCON-система конфигурирования компьютеров;
- PROSPECTOR-определение минералов, которые могут находиться в данном месторождении.
- MYCIN-система диагностики заболеваний крови и минингит ных инфекций;
- PUFF-система диагностики нарушений дыхания;
- EMYCIN-оболочка для построения медицинских экспертных систем;
- VM-система управления аппаратом искусственного дыхания;
- MOLGEN-система планирования экспериментов по исследованию ДНК в области молекулярной генетики;
- DENDRAL-определение химической структуры вещества на основании аналитических методов исследования (спектроскопии и магнитного резонанса);
- EL-анализ электрических цепей.

СИМВОЛЬНЫЕ И АЛГЕБРАИЧЕСКИЕ ВЫЧИСЛЕНИЯ (symbolic and algebraic computing, SAC) - представляют собой одну из основных областей применения символьной обработки. Первыми ее достижениями были системы дифференцирования и интегрирования выражений.

Такие математические системы, как MACSIMA, REDUCE, DERIVE, АНАЛИТИК и другие, могут обрабатывать сложные математические выражения и решать задачи, которые требуют большой затраты труда вычислителей. В этих системах легко осуществляется сокращение выражений, разложение многочлена на множители, решение уравнений, решение систем линейных уравнений в символьном виде, работа с рядами, матрицами, вычисления с рациональными или десятичными числами неограниченной точности.

ДОКАЗАТЕЛЬСТВО ТЕОРЕМ (theorem proving)-разработка общей, не зависящей от задачи системы доказательства и решения проблем.

МОДЕЛИРОВАНИЕ (modelling). Методы искусственного интеллекта применяются в моделировании различных систем и в изучении их деятельности путем моделирования работы. При помощи методов искусственного интеллекта особенно удобно моделирование дискретных систем.

ОБРАБОТКА СИГНАЛОВ И РАСПОЗНАВАНИЕ ОБРАЗОВ (pattern recognition) - автоматическое наблюдение и идентификация(или классификация) объектов. Распознавание осуществляется на основе характеристик и закономерностей, присущих принимаемым сигналам или с учетом строения образа.

РОБОТОТЕХНИКА И АВТОМАТИЗАЦИЯ ПРОИЗВОДСТВА. Создание роботов на методах самостоятельного принятия решений для реализации гибкого производства.

МАШИННОЕ ПРОЕКТИРОВАНИЕ. Создание интеллектуальных систем для разработки проектов в различных направлениях науки и техники.

АВТОМАТИЧЕСКОЕ ПРОГРАММИРОВАНИЕ. Создание компьютером программы по результатам интерактивных действий с ним или по описаниям высокого уровня.

1.5. ЯЗЫКИ ПРОГРАММИРОВАНИЯ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА

Решать задачи искусственного интеллекта, в принципе возможно на любых развитых в настоящее время языках программирования высокого уровня.

PASCAL,C -процедурно- и объектно- ориентированные языки с сильно развитой структурой данных, с большим количеством прикладных библиотек. Они в основном реализуют алгоритмические методы решения задач.

LISP - язык функционального программирования. Основная идея- устанавливание функциональной зависимости между данными. Сильно развиты возможности символьной обработки информации и работа со списками.

PROLOG - язык логического, декларативного программирования. Методы программирования основываются на описании структуры и условий задачи. Сильно развита рекурсия, способы символьной обработки данных, работа со списками и базами данных.

Для решения задач искусственного интеллекта в основном приме няются языки логического и функционального программирования так как в них предусмотрены возможности символьной обработки, методы алгоритмического и эвристического решения задач. Эти языки и создавались для решения проблем искусственного интеллекта.

LISP начал применяться в решении задач искусственного интеллекта еще в 50е годы и по этому в настоящее время является лидером в этой области.Большая часть имеющихся на рынке систем искусственного интеллекта разработана на языке LISP.

1.6. ЯЗЫК ПРОГРАММИРОВАНИЯ PROLOG

Слово PROLOG произошло от слов logic programming и означает программирование в терминах логики. Идея использовать логику в качестве языка программирования возникла впервые в начале 70х. Первыми исследователями, разрабатывавшими эту идею, были Роберт Ковальский из Эдинбурга (теоретические аспекты), Маартен ван Эмден из Эдинбурга (экспериментальная демонстрационная система) и Ален Колмероэ из Марселя - реализация системы доказательства теорем. Первая эффективная реализация языка программирования PROLOG была осуществлена в Эдинбурге Дэвидом Уорреном в середине 70х годов.

В 80е годы реализацией языка PROLOG для компьютеров IBM PC и совместимых с ними занималась фирма BORLAND. Этой фирмой были разработаны диалекты: TURBO PROLOG 1.0 и TURBO PROLOG 2.0.

TURBO PROLOG 2.0 реализует больше возможностей, чем многие другие реализации языка PROLOG для больших компьютеров. В данной разновидности реализаций языка PROLOG предусмотренны все стандартные возможности и созданы:

- настраиваемый на конкретные приложения пользовательский интерфейс;
- стандартный графический интерфейс(BGI); TURBO PROLOG 2.0 поддерживает такие же графические возможности, что и языки TURBO PASCAL, TURBO C;
- дополнительные средства управления окнами, позволяющие создавать гибкие многооконные среды;
- компилятор с оптимизацией кода;
- определение констант и условная компиляция;
- средства диагностики и отладки программ;
- обработка исключительных ситуаций, перехват ошибок, управление пользовательскими прерываниями;
- гарантирован меж языковый интерфейс с TURBO C;

- введены предикаты редактирования, позволяющие реализовать из пользовательских программ доступ к текстовому редактору с полным набором функциональных возможностей;
- возможность создания и управления несколькими внутренними динамическими базами данных, которые могут быть локальными и глобальными;
- система ведения внешних баз данных, которая предусматривает работу с V+ деревьями и использование памяти типа EMS;
- стандартный набор математических предикатов и емкие числовые типы;
- доступ к портам ввода-вывода;
- данная версия поставляется с хорошей документацией и очень большим количеством прикладных программ, поясняющих и демонстрационных.

PROLOG и все его диалекты относятся к языкам логического программирования, потому что они основаны на логике исчисления предикатов первого порядка. В таких языках используют точное логическое описание структуры и условий задачи, т.е. на первый план выдвигается “что нужно сделать”, а “как сделать” остается на втором плане и часто может быть реализовано внутренними механизмами.

В таких языках не очень строго фиксируется последовательность программных строк и способ выполнения программы и поэтому бывают программы, которые в принципе могут работать в обоих направлениях.

Например, чисто логическая программа, написанная на языке PROLOG, может на основании исходных данных вычислить результат, но и без дополнительного программирования на основе результата - исходные данные. В связи с таким описательным подходом к решению задач языки логического программирования еще называют декларативными языками, а в языке PROLOG говорят о декларативной семантике.

Но при написании сложных программ на языке PROLOG необходимо, учитывать и “как сделать”, в этом случае говорят о процедурной семантике.Поэтому программы в языке PROLOG имеют два смысла, а именно: декларативный смысл и процедурный смысл. Наличие таких двух подходов в языке PROLOG, делают его применимым к решению практически любых задач, но с ним как и языком LISP связывают решение задач искусственного интеллекта.

1.7. СТРУКТУРА ИНТЕРАКТИВНОЙ СРЕДЫ TURBO PROLOG 2.0

Для работы с системой программирования TURBO PROLOG 2.0 необходимо на системную дискету или в созданный каталог на жестком диске скопировать файлы:

PROLOG.EXE - содержит интерактивную среду, компилятор, и систему отладки программ;
PROLOG.LIB - библиотека предикатов, нужна при создании файлов EXE;
BGI.LIB - библиотека графических предикатов, нужна при создании файлов EXE;
PROLOG.ERR - содержит сообщения об ошибках;
PROLOG.HLP - содержит помощь по интерактивной среде;
PROLOG.OVL - необходим для работы среды и создания файлов EXE;
INIT.OBJ - содержит необходимые для редактирования связей коды, нужен при создании файлов EXE;
TLINK.EXE - редактор связей, нужен для создания файлов EXE;
TLIB.EXE - библиотека TURBO систем, нужна при создании файлов LIB;
.BGI - драйверы графических устройств, нужны при работе с графикой;
.CHR - графические шрифты, нужны при работе с графикой.

Если при работе не будет осуществляться компиляция на диск, то достаточно скопировать файлы: PROLOG.EXE, PROLOG.ERR, PROLOG.HLP, PROLOG.OVL, соответствующий вашему дисплею файл с расширением BGI и файлы с расширением CHR.

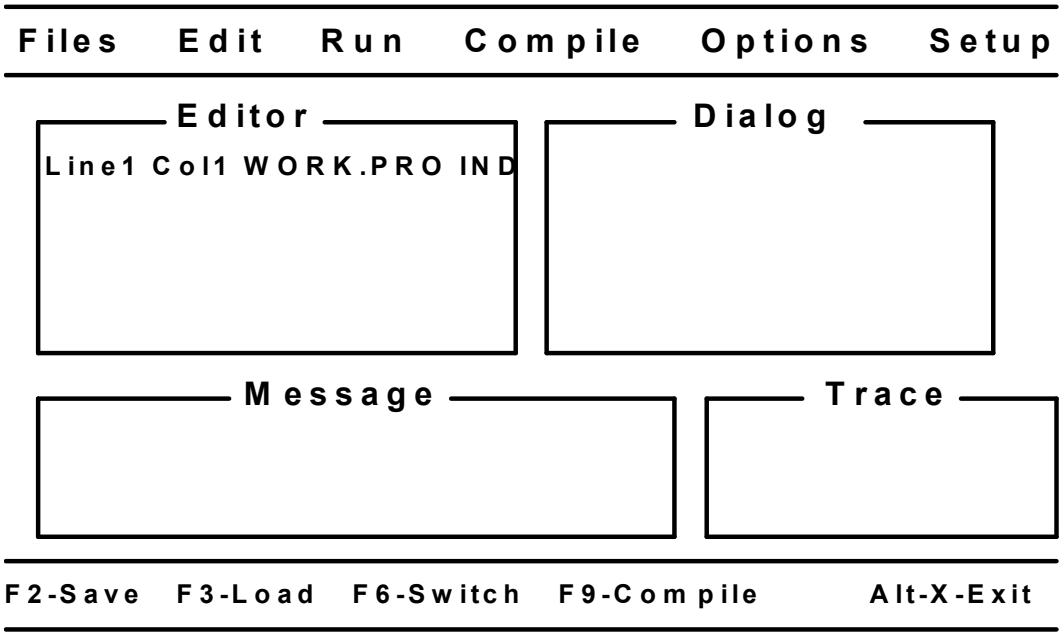
Для запуска интерактивной среды необходимо:сделать текущим каталог содержащий указанные выше файлы, в командной строке ввести PROLOG и нажать клавишу RETURN или ENTER.

TURBO PROLOG 2.0
Copyright © 1986, 88 by
Borland International, Inc

В результате таких действий на экране отобразится интерактивная среда и начальное сообщение о версии среды.

Для начала работы нажмите любую клавишу.

На экране появятся: четыре окна системы, главное меню и статусная строка с описанием основных клавиш.



1.7.1. ОКНА ИНТЕРАКТИВНОЙ СРЕДЫ TURBO PROLOG 2.0

Окно редактирования EDIT представляет собой текстовый редактор совместимый с редакторами сред: BASIC, PASCAL, C фирмы BORLAND.

Для запуска программы на выполнение или компиляцию, ее исходный текст должен быть помещен в это окно; текст можно набрать непосредственно с клавиатуры или загрузить из файла, воспользовавшись элементом меню Files/Load. Для входа в редактор с целью редактирования или ввода программы следует нажать Alt-E или выбрать в главном меню пункт Edit.

В верхней строке окна Edit указаны: текущее положение курсора (строка, позиция), имя редактируемого файла (по умолчанию WORK.PRO), режим редактора (вставка/замена).

Краткое описание редактора:

Перемещение курсора вправо на один символ.

Правая стрелка ..... или Ctrl-D.

Перемещение курсора влево на один символ.

Левая стрелка ..... или Ctrl-S.

Перемещение курсора на одну строку вверх.

Стрелка вверх ..... или Ctrl-E.

Перемещение курсора на одну строку вниз.

Стрелка вниз ..... или Ctrl-X.

Прокрутка вверх на одну строку ..... Ctrl-W.

Прокрутка вниз на одну строку ..... Ctrl-Z.

Перемещение курсора вправо на одно слово.

Ctrl-Правая стрелка ..... или Ctrl-F.

Перемещение курсора влево на одно слово.

Ctrl-Левая стрелка ..... или Ctrl-A.

Перемещение курсора в правый конец строки End ..... или Ctrl-Q D.

Перемещение курсора в левый конец строки Home ..... или Ctrl-Q S.

Перемещение курсора к первой строке окна .....Ctrl-Home.
Перемещение курсора к последней строке окна ..... Ctrl-End.
Перемещение курсора на одну страницу вверх PgUp ..... или Ctrl-R.
Перемещение курсора на одну страницу вниз PgDn..... или Ctrl-C.
Перемещение курсора в начало файла Ctrl-PgUp ..... или Ctrl-Q R.
Перемещение курсора в конец файла Ctrl-PgDn .....или Ctrl-Q C.
Удаление символа в позиции курсора Del..... или Ctrl-G.
Удаление символа слева от курсора Backspace ..... или Ctrl-H.
Удаление слова, на которое указывает курсор .....Ctrl-T.
Удаление строки, на которую указывает курсор ..... Ctrl-Y.
Удаление части строки слева от курсора..... Ctrl-Q T.
Удаление части строки справа от курсора ..... Ctrl-Q Y.
Отметить начало блока .....Ctrl-K B.
Отметить конец блока .....Ctrl-K K.
Скопировать(вставить) отмеченный блок Ctrl-F5..... или Ctrl-K C.
Переместить(вставить) отмеченный блок .....Ctrl-K V.
Удалить отмеченный блок .....Ctrl-K Y.
Отменить удаление блока Ctrl-F7..... или Ctrl-K U.
Записать отмеченный блок в файл на диске.....Ctrl-K W.
Вывести отмеченный блок на принтер .....Ctrl-K P.

Окно диалога Dialog - это окно стандартного ввода-вывода программ. Если программа запускается на выполнение из среды разработки (“внутри” TURBO PROLOG) и не создает своего собственного окна вывода, то любой записываемый или читаемый текст будет появляться в окне диалога.

Окно сообщений Message представляет собой окно вывода системной информации.TURBO PROLOG выводит в этом окне такие сообщения, как

Load c:\tprolog2\my_file.pro
Compile c:\tprolog2\my_file.pro

Окно трассировки Trace используется исключительно для трассировки программ (пошагового отслеживания выполнения программы, с тем чтобы можно было увидеть логику работы TURBO PROLOG). Для использования окна трассировки нужно выбрать в меню Options элемент Trace (или ShortTrace) или поместить в начало исходного текста программы директиву компилятора trace (или shorttrace). Окно трассировки используется всякий раз, когда программа запускается на выполнение внутри системы, в которой активизирован режим трассировки.

1.7.2. МЕНЮ ИНТЕРАКТИВНОЙ СРЕДЫ TURBO PROLOG 2.0

ГЛАВНОЕ МЕНЮ.

В составе главного меню высвечиваются команды и ряд других доступных ниспадающих меню. Элемент главного меню можно выбрать одним из трех способов:

1. Когда главное меню активно, нажатием выделенной засветкой заглавной буквы (F Files, S Setup и т.п.).
2. Когда главное меню активно, переводом в нужное место с помощью клавиш управления курсором полосы засветки и нажатием Enter или Return.
3. Из любого состояния системы (за исключением внешнего редактора) нажатием сочетания Alt-<буква>, где <буква> - соответствующий первый символ выбираемого элемента главного меню (Alt-E - Edit, Alt-R - Run и т.п.).

Примечание: Сочетание Alt-<буква> касается только элементов-главного меню.

EDIT - осуществляет переход из главного меню в окно редактирования, для написания или редактирования программ. Возврат из окна редактирования в главное меню, можно осуществить припомощи клавиши F10.

RUN - запускает, находящуюся в окне EDIT программу. Если программа не компилировалась после редактирования, то перед запуском произойдет ее компиляция.

НИСПАДАЮЩИЕ МЕНЮ.

Четыре элемента главного меню активизируют в случае их выбора ниспадающие меню; это элементы Files, Compile, Options, Setup.

Files Элементы этого меню позволяют работать с файлами (загружать, сохранять, указывать, создавать и записывать на диск), каталогами (отображать, изменять), обращаться к ДОС, а также выйти из системы.

Элементы пункта меню Files:

DIRECTORY - позволяет просмотреть каталоги на текущем диске.

CHANGE DIR - осуществляет смену текущего каталога.Для этого, после активизации данного элемента меню, нужно в окошке написать путь по правилам DOS к нужному каталогу.

WRITE TO - переименовывает, находящийся в редакторе файл и записывает его на диск в текущий каталог или по указанному пути.

SAVE - записывает, находящийся в окне Edit файл, в текущий каталог.

NEW FILE - делает запрос на сохранение файла, находящегося в окне Edit, удаляет все находящееся в этом окне.

LOAD - показывает все файлы с указанной маской, находящиеся в текущем каталоге или в выбранном и позволяет загрузить указанный файл в среду.Если в среде уже был файл, то последует запрос сохранять его или нет. После ответа, произойдет загрузка файла.

PICK - Позволяет загрузить в среду , ранее загружавшиеся файлы.

OS SHELL - Делает временный выход в DOS.Для возврата в среду в командной строке DOS, нужно ввести команду EXIT.

QUIT - Осуществляет полный выход из среды.

Compile Эти элементы позволяют управлять режимами компиляции.

Элементы пункта меню Compile:

MEMORY - Скомпилированная программа помещается в память.

OBJ FILE - Скомпилированная программа помещается в файл с расширением OBJ, содержащий объектный код программы.

EXE FILE - Программа компилируется в исполняемый файл с расши рением EXE.

PROJECT - Компиляция программных проектов.

LINK ONLY - Устанавливает связи в OBJ файле.

Options В этом меню предусмотрено несколько вложенных подменю, с помощью которых можно задавать отдельные опции компиляции (проверка на переполнение, отладочная информация, размеры памяти), опции ре дактирования связей, а также библиотеки, которые должны использоваться при редактировании связей; в этом меню также предусмотрен режим редактирования файла определения проекта (.PRJ).

Элементы пункта меню Options:

LINK Options - Установка параметров редактора связей.

EDIT PGJ FILE - Установка режима редактирования файла опреде ления проекта (.PRJ).

COMPILER DERECTIVES - Установка из среды директив компилятора.

Setup Ниспадающие меню этого элемента главного меню позволяют формировать нужную рабочую среду системы. Можно устанавливать цвета и размеры окон, устанавливать текущие каталоги, модифицировать конфигурацию клавиатуры и текст помощи, выдаваемый в нижней строке, сохранять опции компилятора и загружать ранее сохраненные опции из файла конфигурации (PROLOG.SYS).

Элементы пункта меню Setup:

COLORS - Установка цветов фона и символов в среде.

WINDOW SIZE - Изменение расположения и размеров текущего окна.

DIRECTORIES - Установка каталогов.

MISCELLANEUS - Подключение адаптера CGA, загрузка системных сообщений, установка режимов экрана, просмотр сообщений, помещаемых в статусную строку.

LOAD SYS FILE - Загрузка файла PROLOG.SYS со значениями параметров среды.

SAVE SYS FILE - Сохранение текущих значений параметров Среды в файл PROLOG.SYS.

Командные клавиши.

В общем, независимо от выполняемых в системе в текущий момент действий (трассировка, редактирование, выполнение прикладной программы), нажатие некоторых клавиш (или комбинаций клавиш) будет всегда давать определенный результат. Такие клавиши называются командными. Статусная строка в нижней части экрана содержит сообщения, описывающие командные клавиши текущего режима.

F2-Save	F3-Load	F6-Switch	F9-Compile	Alt-X-Exit

Вызов меню Files	Alt-F
Вызов редактора (Editor)	Alt-E
Запуск программы из окна Edit	Alt-R
Компиляция программы из окна Edit	Alt-C
Вызов меню Options	Alt-O
Вызов меню Setup	Alt-S
Сохранение файла из окна Edit	F2
Загрузка файла	F3
Развертка-свертка текущего окна	F5
Переход между окнами	F6
Изменение размеров окон	Shift-F10
Компиляция программы в памяти	F9
Компиляция программы в файл .OBJ	Shift-F9
Компиляция программы в файл .EXE	Ctrl-F9
Компиляция проекта	Alt-F9
Вызов ДОС	Alt-D
Выход из Турбо-Пролога	Alt-X
Вызов онлайновой помощи	F1
Копирование из внешнего редактора(Xcopy)	F7
Вызов внешнего редактора (Xedit)	F8
Вызов главного меню	F10
Трассировать следующий шаг	F10
Вызов меню Trace	Alt-T
Вызов меню Printer-Log	Alt-P
Возврат из режима задания цели в окно Edit	Ctrl-E
Изменение размера окон	Shift-F10

2. ОСНОВЫ ПРОГРАММИРОВАНИЯ В СРЕДЕ TURBO PROLOG 2.0

2.1. АЛФАВИТ ЯЗЫКА TURBO PROLOG 2.0

Алфавит языка - это фиксированный для данного языка набор основных символов, из которых может состоять текст программы.

Алфавит языка TURBO PROLOG 2. 0 включает в себя:

- прописные латинские буквы (A-Z);
- строчные латинские буквы (a-z);
- цифры 0 1 2 3 4 5 6 7 8 9;
- специальные символы + - * / “ , . _ ! ; \ ‘ [ ] | < > = ( ) :
- пары символов <= >= <> :- /* */
- зарезервированные слова - это совокупности символов, имеющие для языка TURBO PROLOG 2. 0 определенный смысл.

ЗАРЕЗЕРВИРОВАННЫЕ СЛОВА:

global domains	db_selector	include
global	bt_selector	string
clauses	char	nobreak
domains	checkdeterm	nowarnings
predicates	code	symbol
clauses	integer	project
database	nondeterm	trace
constants	config	file
goal	real	trail
dbasedom	diagnostics	shorttrace

2.2. ОБЪЕКТЫ И ТИПЫ ДАННЫХ

Язык программирования PROLOG имеет единообразную форму представления данных и методов их обработки. Любая PROLOG - система распознает вид объекта по его синтаксической форме в тексте программы.

Объекты данных в языке PROLOG называются термами. Термы могут быть: простыми и составными (структурами). Простые термы - это константы и переменные. Константы характеризуются типом и значением.

В языке программирования TURBO PROLOG 2. 0 константы можно определять с символьными именами и в дальнейшем использовать их имена.

В языке программирования TURBO PROLOG 2.0 определены стандартные типы (домены от слова domains).

Стандартные домены:

Домен	Описание
char	Символ, заключенный в одинарные кавычки (например ‘a’)
integer	Целые числа от -32768 до 32767.

real	Вещественные числа с необязательным знаком (+ -), после которого следует несколько цифр DDDDDDD, затем необязательная десятичная точка (.), затем еще несколько цифр DDDDDDD и, наконец, необязательная экспоненциальная часть(e(+или-)DDD):<+|->DDDDD<. >DDDDDDD<e<+|->DD. Примеры вещественных чисел: 42705 9999 86. 72 9111. 929437 521e238 79. 83e+21 Допустимый диапазон значений (1e-307 - 1e+308). Целые при необходимости автоматически преобразуются в вещественные.
string	Любая последовательность символов, заключенная в двой ные кавычки, например, “книга Анатолия Рыбакова”. Строки в программах могут быть длиной до 255 символов, но строки, которые TURBO PROLOG 2. 0 считывает из файла или строит внутри себя, могут быть длиной до 64 К символов.
symbol	Для символьной информации предусмотрены два формата: 1) последовательность букв, цифр и знаков подчеркивания, первой в которой является строчная буква; 2) последовательность символов в двойных кавычках (такой формат нужен тогда, когда в число символов входят пробелы или началом является не строчная буква). Примеры символьных данных: номер_телефона “автобусный билет” “Dorib Inc”.
file	Этим доменом определяется символьное имя файла.

В контексте программы символы и строки взаимозаменяемы, но TURBO PROLOG хранит их по - разному. Символы хранятся в таблице просмотра, а для их представления используются не сами эти символы, а их адреса. Это значит, что установить соответствие для символов можно очень быстро, и, если символ встречается в программе неоднократно, он может храниться очень компактно.

Строки в таблице просмотра не хранятся; TURBO PROLOG при установлении соответствия с ними осуществляет проверку посимвольно.

То, какое представление символьной информации лучше, определяется для каждой конкретной программы отдельно.

Простые объекты:

abc, “любитель животных”, b_t_r_m, dog..... (symbol)

1, 3, 5, 0 ..... (integer)

3. 45, 0. 01, -30. 5, 123. 4e+5 ..... (real)

‘a’, ‘b’, ‘c’, ‘/’, ‘&’ ..... (char)

“Один два”, “номер 5”, “&&”, abc, turbo ..... (string)

myfile, myprog..... (file)

Переменные характеризуются идентификатором, типом и значением. Идентификаторы переменных могут состоять из латинских букв, цифр и символа подчеркивания. Первым символом в имени переменной должна быть прописная буква или символ “_”. Тип переменной определяется ее положением в фактах

и правилах. Он может соответствовать любому стандартному домену или быть комбинированным (структурой).

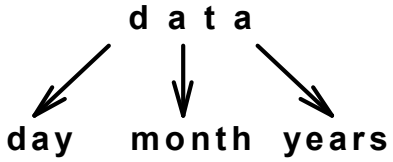
В языке программирования PROLOG нет оператора присваивания, это важное его отличие от других языков. Переменную можно считать локальным именем некоторого объекта данных. Значения переменные в языке PROLOG получают в результате сопоставления с объектами данных в фактах и правилах. Пока переменная не имеет какого - либо значения, она называется СВОБОДНОЙ; когда же такое значение появляется, она становится СВЯЗАННОЙ. Но она остается связанной лишь на то время, которое необходимо для достижения цели; затем PROLOG может освободить ее, вернуться назад и начать поиск альтернативных решений. Это очень важный момент: ХРАНИТЬ ИНФОРМАЦИЮ, ПОЛУЧЕННУЮ ПУТЕМ ПРИСВОЕНИЯ ПЕРЕМЕННОЙ ЗНАЧЕНИЯ, НЕЛЬЗЯ.

Переменные используются как часть процесса сопоставления с образцом, а не как способ хранения информации.

Диапазон действия переменной-одно предложение. Внутри одного предложения каждое появление идентификатора обозначает одну и ту же переменную. Если один и тот же идентификатор встречается в разных предложениях, то он принадлежит различным переменным.

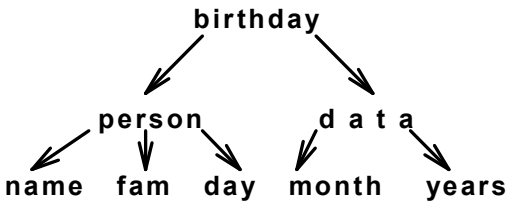
В языке PROLOG предусмотрено использование анонимных переменных. Анонимные переменные обозначаются одиночным символом “_”. Они позволяют убрать из программы ненужную, загромождающую ее информацию. Если из запроса нужна только некоторая определенная информация, то для игнорирования ненужных значений на их место следует поставить анонимные переменные. Анонимная переменная может использоваться на месте любой другой переменной. Разница будет лишь в том, что анонимной переменной никогда не будет присвоено значение. Каждая анонимная переменная в предложении это отдельная сущность, которая отличается от всех анонимных переменных этого предложения.

В языке PROLOG предусмотрено создание и использование комбинированных типов данных, которые называются структурами. Структуры-это способ организации информации, при котором несколько объектов информации объединяются в один объект, причем таким образом, что, при необходимости, они могут быть снова легко отделены. Например, дату можно рассматривать как структуру, состоящую из трех компонентов: день, месяц, год. Для создания структур нужно: задать им символьное имя (функтор), идентифицирующее данную структуру, в круглых скобках через запятую указать компоненты. Количество компонентов называют арностью структуры. Любая структура характеризуется: функтором и арностью. В нашем примере можно создать такую структуру data(day, month, years). Функтор этой структуры-data, компоненты-day, month, years, арность 3. Компонентами структур могут быть: константы, переменные, другие структуры. В языке PROLOG допустимы рекурсивные структуры. Для более наглядного представления структур их изображают в виде деревьев. Корнем дерева служит функтор, ветвями, выходящими из него-компоненты. Если некоторая компонента тоже является структурой, тогда ей соответствует поддерево в дереве, изображающем весь структурный объект.



Построим более сложную структуру, например чей - либо день рождения-birthday (person(name, fam), data(day, month, years)).

Данная структура имеет: функтор-birthday, компоненты - person(name, fam), data(day, month, years), арность 2. Эти компоненты тоже структуры, имеющие: функтор-person, компоненты-name, fam, арность 2; функтор-data, компоненты-day, month, years, арность 3.



Структуры в языке TURBO PROLOG должны быть предварительно описаны в разделе программы domains. При их описании можно под одним доменом описать несколько различных структур.

2.3. ОСНОВНЫЕ ПОНЯТИЯ

Язык программирования PROLOG рожден математической логикой, поэтому его синтаксис и семантику можно наиболее точно описать при помощи логики. Синтаксис языка PROLOG-это синтаксис предложений Хорна, являющийся разновидностью логики предикатов первого порядка.

Эти предложения описывают отношения между объектами данных.

Определим понятия: прямое произведение множеств, отношение, предикат.

Прямым произведением множества A на множество B называют множество P, состоящее из всех упорядоченных пар, первые компоненты которых - элементы множества A, а вторые - элементы множества B.

Например, пусть A={a, b, c}, B={a, d}, тогда P={(a; a), (a; d), (b; a), (b; d), (c; a), (c; d)}.

Отношение - это подмножество прямого произведения множеств.

Например, сложение двух целых положительных чисел можно представить в виде отношения между множествами целых чисел, т. е. как множество P, являющееся прямым произведением трех множеств.

P={(1; 1; 2), (1; 2; 3), (1; 3; 4), . . . }

Предикат - это отношение между объектами данных, которое приводит к преобразованию данных и дает логический результат (успех, неуспех или истина, ложь).

Синтаксис предикатов совпадает с синтаксисом структур. Каждый предикат имеет символьное имя, за которым в круглых скобках через запятую следуют объекты данных(аргументы), связанные данным отношением. Количество аргументов предиката определяют его размерность (арность). Имена предикатов должны начинаться со строчной буквы, содержать в себе только латинские буквы, цифры, символ “_”. Количество символов в имени предиката не может быть больше 255. Аргументами предиката могут быть любые допустимые в языке TURBO PROLOG термы.

Примеры записи предикатов:

likes(“Иван”,”футбол”), likes - имя предиката, устанавливающего отношение likes, ”Иван”, ”футбол” - аргументы, они являются константами домена string, арность предиката 2. Данный предикат можно обозначить likes/2.

Можно использовать и запись likes(X, Y), где X, Y-переменные. В этом случае отношение likes обобщается для всех значений переменных X, Y. read_persons(Name, person(Name, Adr, Tlf)), read_persons-имя предиката, устанавливающего отношение read_persons, Name, person(Name, Adr, Tlf)-аргументы данного предиката. Name-переменная, person(Name, Adr, Tlf)-структура арности 3 с функтором person. Арность данного предиката 2. Предикаты могут быть и без аргументов, например repeat. Арность такого предиката 0. Область действия таких предикатов сильно ограничена.

Действия, производимые предикатами над информацией зависят от направления потока информации. При вводе информации в предикат он может производить одни действия, а при выводе другие.

Предикаты могут быть детерминированными и недетерминированными.

Предикаты, использование которых приводит к множеству решений, называют недетерминированными.

Предикаты, использование которых приводит к единственному решению, называют детерминированными.

С точки зрения декларативной семантики языка PROLOG предикаты - это отношения между объектами данных.

С точки зрения процедурной семантики предикаты - это процедуры.

2.4. СИНТАКСИС ЯЗЫКА TURBO PROLOG 2.0

Синтаксис языка TURBO PROLOG 2. 0 рассчитан на выражение знаний о свойствах и отношениях. Программы состоят из целого ряда разделов, содержащих в себе: описание объектов данных, определение и описание отношений, цели, которые должна достичь система.

Программа на языке TURBO PROLOG 2. 0 может состоять из описания и предложений или только из предложений. Предложения состоят из головы и тела, связанных словом if (можно вместо этого слова использовать символы ":-"). Голова представляется одним предикатом, тело одним или несколькими предикатами, связанными между собой логическими операциями: конъюнкция (можно писать and или ", "), дизъюнкция (можно писать or или "; "). Чаще для связи предикатов в предложении используется конъюнкция. Каждое предложение заканчивается точкой.

Предложения могут быть трех видов: факты, правила, цели.

С процедурной точки зрения:

факты - это предложения, имеющие голову и пустое тело. Факты содержат утверждения, которые безусловно истинны. Системе TURBO PROLOG нет необходимости никуда обращаться за подтверждением истинности фактов, поэтому они являются основой для логического вывода;

правила - это предложения, имеющие голову и непустое тело. Правила содержат утверждения, которые могут быть истинными только при выполнении некоторых условий. Для доказательства истинности правила системе TURBO PROLOG необходимо обращаться к фактам и другим правилам.

Например, пусть мы имеем некоторое абстрактное правило:

P:-Q, R. С процедурной точки зрения его можно интерпретировать так:

Чтобы получить истинность P, нужно сначала получить истинность Q, а затем истинность R;

цели (вопросы или запросы)-это предложения, имеющие только непустое тело. Цели могут быть достижимы (успешны) или недостижимы (имеют неуспех). При работе система TURBO PROLOG на основе своих механизмов логического вывода, фактов и правил программы стремится достичь цели. Достичь цели-это значит показать, что она логически следует из фактов и правил программы.

Цели могут быть внешними и внутренними. Внешние цели ставятся перед системой уже в процессе ее работы над программой, если в тексте программы нет цели. Они вводятся интерактивно по запросу системы. При постановке внешней цели и ее достижении системой, мы получаем все возможное множество решений. Внутренние цели ставятся перед системой в тексте программы и если не применять специальных мер, то при их достижении мы получим единственное решение.

С декларативной точки зрения:

факты-это предложения, описывающие: единственные экземпляры объектов, свойства объектов или отношения между объектами; правило - это предложение описывающее, что может быть получено на основе другой информации. С данной точки зрения правило P:-Q, R можно интерпретировать так: P-истинно, если Q и R истинны; или из Q и R следует P; цель-это предложение, определяющее то, что мы можем получить на основании описанных нами свойств и отношений.

Примеры:

Факты:
```
likes(“Елена”, reading).
likes(“Иван”, computers).
likes(“Иван”, badminton).
likes(“Леонид”, badminton).
likes(“Артем”, swimming).
likes(“Артем”, reading).
```

Данные факты определяются предикатом likes(string, symbol), имеющим два аргумента типа string, symbol. Данный предикат устанавливает между аргументами отношение likes. Это отношение на естественном языке можно интерпретировать как : ”Елене нравится читать. ”, “Ивану нравятся: компьютеры и бадминтон”, ”Артему нравится плавать и читать” и т. д.

Факты:
```
car(“жигули”).
car(“запорожец”).
for_sale(“москвич”).
for_sale(“жигули”).
for_sale(“запорожец”).
```

Данные факты определяются предикатами: car и for_sale. У них по одному аргументу типа string. Они определяют экземпляры объектов. На естественном языке данные факты можно интерпретировать как: ”Машины-жигули, запорожец”, ”Продаются-москвич, жигули, запорожец”.

Факты :
```
colors(“снег”, ”белый”).
colors(“уголь”, ”черный”).
colors(“дерево”, ”зеленое”).
```

Данные факты определяются предикатом: colors(string, string)

Эти факты выражают свойство объектов. На естественном языке их можно интерпретировать как : ”Снег-белый”, ”Уголь-черный”, ”Дерево-зеленое”.

Правила:
```
likes(“Максим”, A):-likes(“Иван”, A).
```
Голова данного правила - likes(“Максим”, A).
Тело правила - likes(“Иван”, A).

Голова данного правила будет истинна, если системе на основании фактов удастся доказать истинность тела этого правила. Если в программе содержащей такое правило не будет фактов относительно “Максима”, а будут факты относительно “Ивана”, то система на основании данного правила получит новый факт относительно “Максима”. На естественном языке это правило можно интерпретировать как : “Максиму нравится то, что нравится Ивану. ”

```
likes(“Максим”, A):-likes(“Иван”, A), likes(“Леонид”, A).
```
Голова данного правила - likes(“Максим”, A).
Тело правила - likes(“Иван”, A), likes(“Леонид”, A).

В теле правила стоит два предиката, связанных при помощи конъюнкции. В данном случае системе удастся доказать истинность правила, если будет получена истинность likes(“Иван”,A) и likes(“Леонид”,A). При этом система получит новый факт относительно “Максима”.

На естественном языке это правило можно интерпретировать как: “Максиму нравится то, что нравится Ивану и Леониду ”. Если в теле правила предикаты будут связаны при помощи дизъюнкции, то правило будет истинно и позволит получить новый факт, если хотя бы один из предикатов даст на основании фактов истину.

Цели :
```
likes(X, reading).
```

В этом случае мы ставим перед системой цель определить, кто любит читать. Цель будет достигнута, если в программе будут соответствующие факты и правила. В случае внешней цели мы получим все возможные решения. При наличии в программе фактов:

```
likes("Елена", reading).
likes("Иван", computers).
likes("Иван", badminton).
likes("Леонид", badminton).
likes("Артем", swimming).
likes("Артем", reading).
```

мы получим решения. X="Елена", X="Артем". Если цель будет внутренней мы можем получить единственное решение X="Елена". Перед системой можно поставить такую цель:

```
likes("Артем", У).
```

В нашем примере при внешней цели мы получим: У=swimming, У=reading.

В случае внутренней цели мы можем получить только У=swimming.

Перед системой можно поставить и обобщенную цель, касающуюся всех фактов:

```
likes(X, У).
```

В случае внешней цели мы получим:

```
X= "Елена"  У="reading"
X= "Иван"   У="computers"
X= "Иван"   У="badminton"
X= "Леонид" У="badminton"
X= "Артем"  У="swimming"
X= "Артем"  У="reading"
```

В случае внутренней цели мы можем получить:

```
X= "Елена"  У="reading"
```

2.5. РАЗДЕЛЫ ПРОГРАММ

2.5.1. РАЗДЕЛ CONSTANTS

В программе на языке TURBO PROLOG можно объявлять и использовать символьные константы. Раздел объявления констант озаглавливается ключевым словом constants , после которого следуют сами объявления с соблюдением следующего синтаксиса:

<Идентификатор> = <Макроопределение>

<Идентификатор> - это имя константы, а <Макроопределение> - это то, что этому имени соответствует. Каждое <Макроопределение> заканчивается символом <новая строка>, так что в одной строке может размещаться только одна константа. На объявленные таким образом константы можно затем ссылаться в программе.

Рассмотрим следующий пример:

```
constants
        zero = 0
        one = 1
        two = 2
        hundred = (10*(10-1)+10)
        pi = 3. 141592653
        slash_fill = 4
        red = 4
```

Перед компиляцией программы TURBO PROLOG заменит каждую константу действительной строкой, которую она представляет. Например:

```
  . . . ,
A = hundred * 34,  delay(A),
setfillstyle(slash_fill, red),
Lengths_Circle = pi * Diameter,
  . . .
```

будет обрабатываться компилятором как

```
    . . . ,
A = (10*(10-1)+10)*34, delay(A),
setfillstyle(4, 4),
Lengths_Circle = 3. 141592653 * Diameter,
    . . .
```

На использование символьных констант накладывается несколько ограничений.

- Определение константы не может ссылаться на себя. Например, задание my_number = 2 * my_number/2 приведет к выдаче сообщения об ошибке Recursion in constant definition (рекурсия в определении константы).
- Система не различает в определении константы прописные и строчные буквы. Соответственно, когда определенный в разделе constants идентификатор используется в фактах и правилах, его первая буква должна быть строчной во избежание толкования константы как переменной. Вот пример правильной конструкции:

```
constants    Two = 2
goal         A = two, write(A).
```

В программе может быть несколько разделов constants, но константы должны объявляться до их использования.

Идентификаторы констант являются глобальными и могут объявляться только один раз. Несколько объявлений одного и того же идентификатора приведут к выдаче сообщения Constant identifier can only be declared once (Идентификатор константы может быть объявлен только один раз).

2.5.2. РАЗДЕЛ DOMAINS

Домены в языке TURBO PROLOG выполняют роль типов данных. Они дают возможность присваивать различным видам информации (которая в противном случае выглядела бы одинаково) отличные имена. В программе на языке TURBO PROLOG объекты в отношении (аргументы предиката) принадлежат доменам; это могут быть стандартные домены или специальные, определяемые программистами.

Раздел domains служит двум очень важным целям. Вопервых, можно определить для доменов осмысленные имена, причем даже в том случае, если внутренне они совпадают с именами уже существующих доменов.Во - вторых, объявления специальных доменов используются для объявления структур данных, которые стандартными доменами не определяются. Через один домен можно описать сразу несколько структур, разделяя их символом “; ”. Возможно описание рекурсивных структур данных (деревья, списки).

Иногда целесообразно объявить домен тогда, когда возникает потребность более четкого выделения каких-либо аргументов предиката.

Объявление программистом своих собственных доменов помогает документировать предикаты, которые определяются путем задания в качестве типа аргумента удобного и понятного имени.

Примеры:

```
domains
        day, years=integer
        month=symbol
        dates=data(day, month, years)
        name, fam=string
        persons=person(name, fam)
         birthdays=birthday(persons, dates)
```

В данном примере через стандартные домены integer, symbol, string,описаны понятные и удобные для использования структуры данных: dates, persons, birthdays.

```
 domains
        func_name=sin(x);cos(x);tan(x);arctan(x);abs(x);exp(x);
                    ln(x);log(x);sqrt(x)
                x=real
domains
        treetype = tree(string,  treetype,  treetype) ; empty()
```

В этом примере приведено рекурсивное описание домена, оно применяется для создания двоичных деревьев.

```
domains
        intlst=integer*
        reallst=real*
        charlst=char*
        strlst=string*
```

В этом примере приведено описание доменов для списков стандартных типов.

Для более простого описания домена для списков любого типа, его можно описать как список структур, компоненты которых могут быть любого типа, в том числе и списками.

Это показано в следующем примере:

```
domains
        llist = l(list); s(symbol); i(integer); c(char); t(string)
        list = llist*
```

Во всех предыдущих примерах описание доменов было локальным.

Домены, описанные в разделе domains, применимы в пределах только одного программного модуля.

2.5.3. РАЗДЕЛ GLOBAL DOMAINS

Для описания доменов, применимых во всех программных модулях или в программных проектах, используется раздел global domains.

Пример:

```
global domains
        row, col, len, attr  = integer
        stringlist = string*
        sp = string*
        file = tlf
        integerlist = integer*
        key    = cr; esc; break; tab; btab; del; bdel; ctrlbdel;ins;
          end; home; fkey(integer); up; down; left; right ;
                ctrlleft; ctrlright; ctrlend; ctrlhome; pgup; pgdn;
               ctrlpgup; ctrlpgdn; char(char) ; otherspec
```

2.5.4. РАЗДЕЛ DATABASE

Программа на языке TURBO PROLOG представляет собой совокупность фактов и правил. Иногда в процессе выполнения программы может возникнуть потребность видоизменения (модификации, удаления или добавления) некоторых фактов, с которыми работает программа.

В таком случае факты можно представить в виде динамической или внутренней базы данных; она может изменяться в процессе выполнения программы. В языке TURBO PROLOG для объявления в программе фактов,которые должны стать частью динамической (или изменяющейся) базы данных, предусмотрен специальный раздел database. Такой раздел базы данных объявляется с помощью ключевого слова database. Здесь объявляются факты, предназначенные для включения в динамическую базу данных. В языке TURBO PROLOG имеются встроенные предикаты, облегчающие использование динамической базы данных.

Пример:

```
database
        person(string, string, string, string, string)
```

Внутренним базам данных можно присваивать символические имена, для этого после ключевого слова database=mydba. Если имя для динамической базы данных не указывается, то, по умолчанию, принимается стандартное имя dbasedom. Посредством символьных имен мы можем объявлять внутренние базы данных аргументами предикатов. Важно помнить о том, что имена предикатов базы данных в пределах программного модуля являются неповторяющимися. В двух разных разделах database одно и то же имя предиката использовать нельзя.

Пример:

```
database = mydba
                myfirstpred(integer)
                mysecondpred(real, string)
                mythirdpred(char, string)
```

2.5.5. РАЗДЕЛ PREDICATES

Если программист хочет в программе использовать свой собственный предикат, то он должен объявить его в разделе predicates. В противном случае TURBO PROLOG не будет знать, о чем идет речь.

Когда объявляется предикат, среде сообщается о том, к каким доменам относятся аргументы этого предиката. TURBO PROLOG располагает множеством встроенных предикатов. Объявлять встроенные предикаты при использовании их в программе не нужно. Полностью и подробно встроенные предикаты языка TURBO PROLOG описываются в Справочном руководстве по стандартным предикатам. Предикаты определяются фактами и правилами. В разделе predicates программы просто перечисляется каждый предикат с указанием типов (доменов) его аргументов. Объявление предиката начинается с имени предиката, за которым следует открывающая (левая) скобка. После имени предиката и открывающей скобке следует нуль или более аргументов предиката.

имяПредиката(тип_аргумента1, тип_аргумента2, . . . , тип_аргументаN)

После каждого типа аргумента, кроме последнего, идет запятая, а после последнего - закрывающая (правая) скобка. Следует заметить, что объявление предиката не заканчивается точкой. Типы аргументов представляют собой либо стандартные домены, либо домены, объявленные в разделе domains.

Примеры:

Если в разделе predicates объявить предикат,

```
predicates
        my_predicates(symbol, integer)
```

то его аргументы не нужно объявлять в разделе domains, поскольку приведенные аргументы являются стандартными.

Но если объявить предикат,

```
predicates
        my_predicates(name, number)
```

то понадобится объявить name(тип symbol) и number(тип integer), как принадлежащие соответствующим стандартным доменам:

```
domains
        name = symbol
        number = integer
predicates
        my_predicates(name, number)
```

Предикаты объявленные в разделе predicates являются локальными и применимы только в пределах одного программного модуля.

2.5.6. РАЗДЕЛ GLOBAL PREDICATES

Для объявления предикатов, применимых во всех программных модулях или в программных проектах , используется раздел global predicates.

Пример:

```
GLOBAL PREDICATES
                maxlen(STRINGLIST, COL, COL)
                listlen(STRINGLIST, ROW)
                writelist(ROW, COL, STRINGLIST)
```

2.5.7. РАЗДЕЛ CLAUSES

Раздел clauses - это ядро программы на языке TURBO PROLOG; в нем описываются факты и правила, которыми оперирует система, пытаясь достичь цели программы, используя механизмы логического вывода.

Примеры:

```
clauses
```

```
likes("Елена", reading).
likes("Иван", computers).
likes("Иван", badminton).
likes("Леонид", badminton).
likes("Артем", swimming).
likes("Артем", reading).
```

Возможности данного примера можно расширить вводом правила:

```
clauses
    likes("Елена", reading).
    likes("Иван", computers).
    likes("Иван", badminton).
    likes("Леонид", badminton).
    likes("Артем", swimming).
    likes("Артем", reading).
    likes("Максим", A):-likes("Иван", A).
```

В следующем примере описывается уже несколько предикатов при помощи различных фактов и правила:

```
clauses
    can_buy(X, Y) :-person(X), car(Y),
                    likes(X, Y), for_sale(Y).
    person("Иван").
    person("Артем").
    car("запорожец").
    car("москвич").
    likes("Иван", "москвич").
    likes("Артем", "жигули" ).
    for_sale("жигули").
    for_sale("запорожец").
    for_sale("москвич").
```

2.5.8. РАЗДЕЛ GOAL

Если данный раздел отсутствует в программе, то при ее запуске в окне Dialog появится подсказка Goal: и компьютер будет ждать ввода цели с клавиатуры. После ввода цели система будет пытаться ее достичь. Если на основании фактов и правил программы цель будет достигнута, то система в окно Dialog выведет все возможные решения, укажет количество решений перед словом Solutions и будет ждать введения новых целей.

Если на основании фактов и правил программы цель не будет достигнута, то система в окно Dialog выведет No Solutions (нет решений) и будет ждать введения новых целей.

Цели, которые ставятся перед системой интерактивно, в процессе выполнения программы называют внешними. Они обычно при достижении приводят к множеству решений. Внешние цели используют в тех случаях, когда программы будут исполняться только в среде TURBO PROLOG. Чтобы можно было использовать внешние цели, в программе раздел goal должен отсутствовать.

Если программа будет компилироваться в автономный загрузочный модуль(в EXE файл), то раздел goal должен быть в программе обязательно.

Задаваемые в этом разделе цели называются внутренними целями, поскольку они являются частью исходного текста программы и компилируются наряду со всеми другими ее частями. При запуске программы система вызывает цель, обращаясь к разделу goal, программа, выполняясь, пытается достичь целевое предложение. Если будут истинны все предикаты раздела goal, то программа успешно завершается. Если же в процессе выполнения программы какой-либо предидикат целевого предложения дает ложь, то программа заканчивает работу неудачно. (Хотя, если смотреть на программу извне, разница между этими двумя случаями не обязательно должна быть видна, программа просто завершается). Внутренние цели, если не применять специальных мер, обычно приводят к единственному решению.

Примеры:

```
goal
 makewindow(1, 7, 7, "Window 1",1,1,10, 40,1,0,"\176\176\176\176\176\179"),
 makewindow(2, 7, 7, "Window 2",6,20,10,40,0,30,"******"),
 readchar(_),window_str(X),removewindow,
 readchar(_),write(X), readchar(_).
```

Данная программа может состоять только из раздела goal, так как в ней используются только стандартные предикаты.

```
goal
   makewindow(1,1,7, "one",5,0,10,20),write("ONE"),
   makewindow(2,2,7,"two",1,10,10,20), write("TWO"),
   makewindow(3,3,7,"three",2,20,10,20),write("THREE"),
   makewindow(4,4,7,"four",8,30,10,20),write("FOUR"),
   makewindow(5,5,7,"five",4,40,10,20),write("FIVE"),
   makewindow(6, 6,7,"six",5,50,10,20),write("SIX"),
   makewindow(7,7,7,"seven",9,5,10,20),write("SEVEN"),
   makewindow(8,8,7,"eight",1,10,10,20),write("EIGHT"),
   makewindow(9,9,7,"nine",15,20,10,20),write("NINE"),
   repeat, random(9, X), N=X+1, shiftwindow(N),
   framewindow(112),
   delay(1000), framewindow(7), keypressed.
```

Эта программа будет иметь и другие разделы потому, что в ней используются нестандартные предикаты.

Текст программы в языке TURBO PROLOG можно сопровождать комментариями. Комментарии делают более понятным текст программы. Они могут располагаться в любом месте программы. Комментарий начинается с последовательности символов /* и заканчивается символами */ или может начинаться символом % и все, что будет написано после этого символа до конца строки, является комментарием.

2.6. ДИРЕКТИВЫ КОМПИЛЯТОРА

В языке TURBO PROLOG предусмотрено использование директив компилятора, они могут добавляться к программе для того, чтобы при компиляции текст программы трактовался определенным образом. Большинство директив могут быть установлены и интерактивно с использованием элемента Compiler Directives меню Options.

(Подробно директивы компилятора рассматриваются в справочном руководстве.) Познакомимся с основными директивами.

2.6.1. ДИРЕКТИВА INCLUDE

По мере накопления опыта программирования на языке TURBO PROLOG, как правило,обнаруживается, что некоторые предикаты регулярно в программах повторяются. Чтобы сэкономить на вводе часто используемых предикатов, можно воспользоваться директивой включения include. Пример ее возможного использования.

- 1. Создается файл (скажем, MYFILE. PRO), в котором объявляются часто используемые предикаты (с использованием разделов domains и predicates) и приводятся описания этих предикатов в разделе clauses.
- 2. Создается текст программы, в которой предполагается использование этих предикатов.
- 3. На естественной границе исходного текста помещается строка include "myfile. pro" (Естественной границей является любое место в программе, в котором может быть помещен раздел domains или goal).
- 4. При компиляции исходного текста программы TURBO PROLOG будет компилировать и содержимое myfile. pro.

Директива include может использоваться для включения в исходный текст практически любого фрагмента, а один включаемый файл может включать в свою очередь другой (но любой файл может включаться в программу только один раз).

Директива include может указываться на любой естественной границе исходного текста. Тем не менее, при включении в исходный текст файла следует соблюдать накладываемые на структуру программы ограничения.

2.6.2. ДИРЕКТИВЫ TRACE И SHORTTRACE

В среде TURBO PROLOG имеются очень мощные средства трассировки, позволяющие:

- отслеживать пошагово выполнение программы;
- расставить точки трассировки для отслеживания выполнения определенных фрагментов программы;
- трассировать только некоторые указанные предикаты;
- выполнять трассировку в режиме оптимизации;
- интерактивно отключать и включать трассировку;
- интерактивно направлять трассу и входную информацию на печатающее устройство или в файл.

Трассировка представляет собой диагностическое средство, позволяющее отслеживать выполнение программы пошагово, наблюдая каждое вызываемое выражение и возвращаемое в результате вызова значение. При помощи трассировки можно увидеть то, что TURBO PROLOG делает в попытке удовлетворить поставленную цель.

Хотя есть несколько способов активизации режима трассировки, простейший состоит во включении в программу соответствующей директивы компилятора в самое начало исходного текста; для этого нужно поместить туда trace или shorttrace. Директива trace будет обеспечивать выдачу информации о каждом вызове и возврате, являющихся логической частью выполнения программы, а shorttrace учитывает ряд специализаций, осуществляемых системой, и обеспечивает выдачу информации только о подмножестве вызовов и возвратов.

При запуске программы в режиме трассировки в окне Trace отображается каждый вызываемый предикат с текущими значениями его аргументов. Редактор же помещает курсор на вызванный в текущий момент предикат, указывая место выполнения программы и значения любых связанных переменных. Выполнение текущего шага трассировки и переход к следующему осуществляется при нажатии клавиши F10. По мере того, как TURBO PROLOG пытается удовлетворить очередной вызов, курсор передвигается по программе к тому месту, в котором система пытается найти соответствие с текущим вызовом.

Сообщения, выдаваемые в окне Trace

CALL	Каждый раз, когда вызывается предикат, в окне Trace отображаются имя предиката и значения его параметров.
RETURN	Когда выражение удовлетворяется, в окне Trace отображается RETURN и осуществляется возврат в любой вызывающий предикат. Если имеются другие удовлетворяющие входным параметрам выражения, то на экран выводится звездочка (*), указывающая, что данное выражение соответствует точке возврата (недетерминированной).
FAIL	Когда для предиката имеет место неудачный исход, выводится слово FAIL, за которым следует имя соответствующего предиката.
REDO	Данное слово указывает на то, что имеет место возврат. В окне Trace при этом отображаются имя предиката, на который делается переход при поиске с возвратом, а также значения его параметров.

Пример:

```
trace
goal
  makewindow(1, 7, 7, "Моя первая программа", 4, 56, 14, 22), nl,
  write("Введите свое имя, \n а затем нажмите\n Enter. "),cursor(5, 4),
```

```
    readln(Name),  nl,
    write("Добро пожаловать\n в TURBO PROLOG\n ", Name),nl.
```

Вставив в программу trace, можно попытаться запустить ее на выполнение с трассировкой (Alt-R). Нужно следить за тем, что происходит на экране. В окне Edit курсор мигает в начале цели, а в окне Trace запуск на отработку этой цели отражается сообщением:

```
CALL:  goal()
```

Чтобы отработка этой цели осуществлялась в пошаговом режиме, необходимо далее нажать F10. После первого нажатия F10 в окне Trace появится сообщение:

```
CALL:  makewindow(1, 7, 7, "Моя первая программа", 4, 56, 14, 22)
```

отражающее тот момент, что для достижения цели должен быть удовлетворен предикат makewindow. Снова нужно нажать F10; TURBO PROLOG отработает вызов makewindow, построив окно в верхнем правом углу экрана. Это найдет свое отражение в окне трассировки:

```
RETURN:  makewindow(1, 7, 7, "Моя первая программа", 4, 56, 14, 22)
```

И снова нужно нажать F10 и наблюдать за тем, что происходит в каждом из окон, вплоть до появления в окне трассировки сообщения

```
CALL:  readln(_)
```

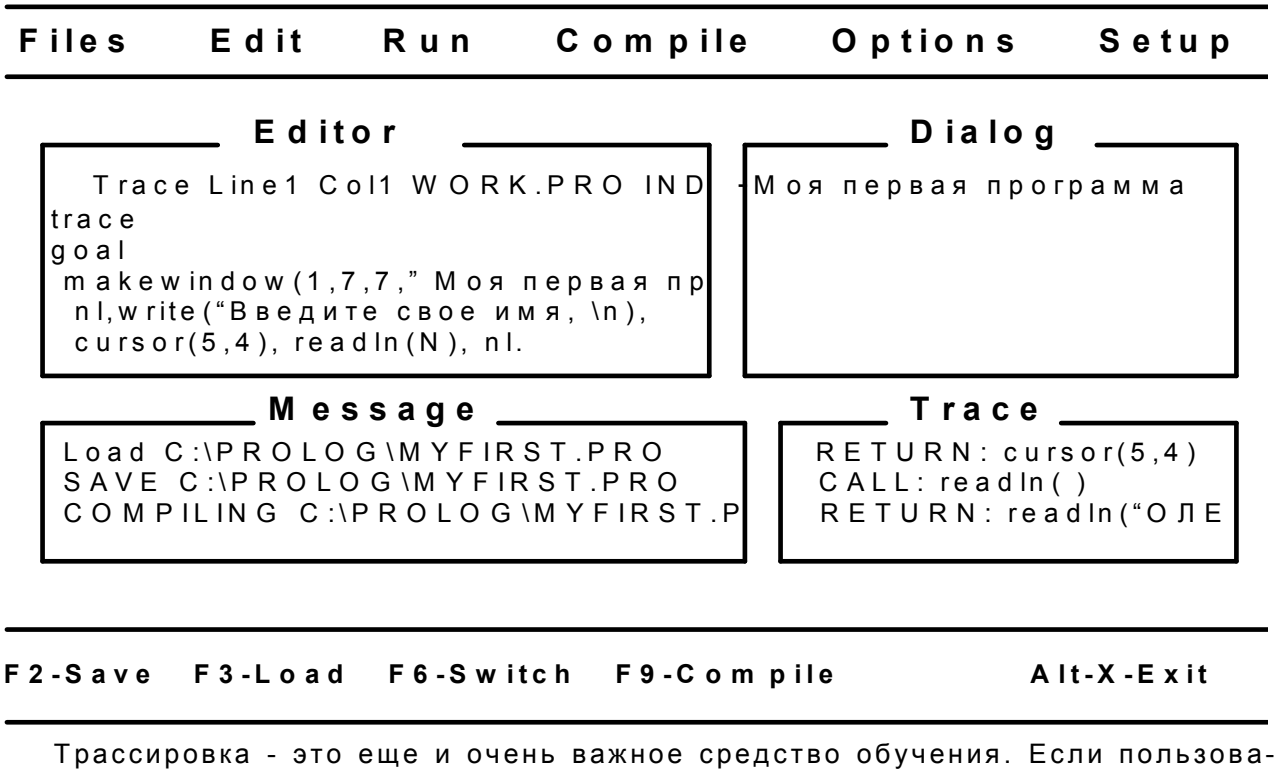
Снова нужно нажать F10. Курсор должен будет остановиться и мигать в том месте окна программы, где должно быть введено имя. Нужно набрать Олег и нажать Enter. В окне Trace появится сообщение

```
RETURN:  readln("Олег")
```

После этого нужно нажимать F10 до тех пор, пока в окне Trace не появится сообщение

```
RETURN:  goal(),
```

означающее завершение работы программы. После нажатия F10 еще один (последний) раз система выдаст запрос на нажатие пробела для возврата в главное меню.



Трассировка - это еще и очень важное средство обучения. Если пользователь не понимает, почему некоторый предикат работает так, как он работает, трассировка обращений к этому предикату должна пролить на эту проблему определенный свет.

В языке TURBO PROLOG предусмотрена оптимизация , увеличивающая скорость выполнения программ. Одним из важных методов оптимизации является устранение хвостовой рекурсии.

Обычно при использовании директивы trace в состав трассы входят все вызовы (CALL) и возвраты (RETURN), являющиеся логической частью выполнения программы. Сюда входят и все вызовы и возвраты, которые обычно бывают частью цикла хвостовой рекурсии, даже если, как это обычно бывает, TURBO PROLOG оптимизирует эти вызовы. Но можно запустить программу в таком режиме трассировки, который учитывает упомянутую внутреннюю оптимизацию. При работе в оптимизированном режиме трассировки вызовы, относящиеся к хвостовой рекурсии, отображаются менее избыточно, а информация RETURN, относящаяся к этим вызовам, выдается не всегда. Трассировка с использованием встроенной оптимизации определяется (в отличие от trace) директивой shorttrace. (Можно также выбрать элемент Shorttrace из меню Options/Compiler Directives/Trace).

В среде TURBO PROLOG можно осуществлять трассировку и только некоторых предикатов. Для этого после директивы trace или shorttrace указывается перечень трассируемых предикатов:

trace n1, n2, n3, . . . или shorttrace n1, n2, n3, . . .

При такой записи директив компилятора в трассу будут входить только сообщения CALL и RETURN по указанным предикатам.

Помимо оптимизации скорости трассировки, директива shorttrace обеспечивает вывод в окне Trace и более короткой трассы. При трассировке большого программного фрагмента это также может быть преимуществом. Тем не менее, при использовании как trace, так и shorttrace вполне может выводиться избыточная трасса; в таком случае для существенного сокращения трассы или сведения к нулю информации по конкретному предикату можно использовать предикат trace. Этот предикат всегда дает удачный исход, а работает вообще при активизированном режиме трассировки.

Включить же режим трассировки можно либо задав в самом начале программы директиву trace или shorttrace, либо выбрав этот режим в меню Options/Compiler Directives/Trace. Следовательно, если программа содержит trace(off), система отключит трассировку до тех пор, пока не встретится trace(on), после чего трассировка будет возобновлена.

В распоряжении пользователя всегда имеется сочетание клавиш Alt-P, служащее для перехода в режим сохранения трассы. Alt-P перенаправляет выводимую трассу на печатающее устройство или в файл PROLOG. LOG.

После запуска программы в режиме трассировки можно нажать сочетание Alt-T для модификации этого режима. При этом появляется меню из трех опций: Status, Trace Window и Edit Window. Для выбора опции используются клавиши управления курсором <ВВЕРХ> и <ВНИЗ>, а также <ВВОД> для включения-отключения опции. Для возобновления работы программы нужно нажать клавишу F10. Рассмотрим описание приведенных опций меню Trace.

Опция Status:

Данная опция определяет режим директивы trace. Для изменения этого режима нужно нажать клавишу S или <ВВОД>. Действует же эта опция аналогично предикатам trace(on) и trace(off) внутри программы.

Опция Trace Window:

Данная опция определяет, выводится или нет обычная трасса в окне Trace. Когда эта опция установлена в состояние “выключено”(а опция Edit Window в состояние “включено”), TURBO PROLOG проходит от вызова к вызову без вывода каких-либо сообщений в окне Trace. Для прохождения от вызова к вызову нужно лишь нажимать клавишу F10.

Опция Edit Window:

Данная опция определяет, будет ли TURBO PROLOG показывать в окне Edit каждый шаг выполнения программы. Когда эта опция установлена в состояние “выключено”, (а опция Trace Window - в состояние “включено”), трасса отображается в окне Trace, а программа выполняется быстрее, и нет необходимости нажимать F10.

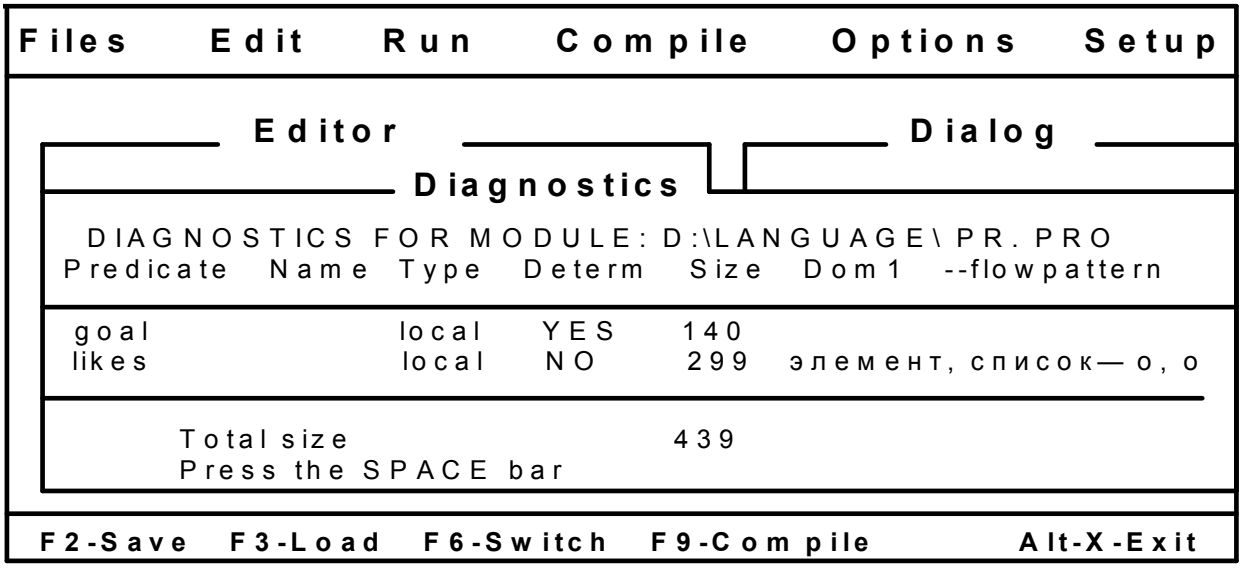
Примечание: Установка двух последних опций в состояние “выключено” равносильна установке в “выключено” опции Status.

2.6.3. ДИРЕКТИВА DIAGNOSTICS

Данная директива компилятора генерирует список всех предикатов и определяет, являются ли они локальными, глобальными (во внешнем модуле) или базой данных. В этом списке также указывается, являются ли предикаты детерминированными, а также даются размеры программных кодов, домены аргументов и структуры потока. Кроме того, указываются общий размер программы и список справочных доменов в модуле.

Это средство имеет большое значение для сопровождения и документирования больших программ. Рассмотренная директива помогает также идентифицировать предикаты, необходимости в которых нет, а также ненужные потоки. Для управления выводом результатов диагностики (на печатающее устройство или в файл PROLOG. LOG) можно воспользоваться меню Printer-Log (Alt-P).

Пример:



2.6.4. ДИРЕКТИВЫ CHECK_DETERM И NONDETERM

Директива компилятора check_determ указывает на необходимость проверки наличия недетерминированных выражений. Если такая директива задается, TURBO PROLOG в процессе компиляции при обнаружении недетерминированного выражения выдает предупреждение. А директиву nondeterm необходимо указывать перед объявлением заведомо недетерминированных предикатов.

Примечания: Если используется директива check_determ, то задание nondeterm перед заведомо недетерминированными предикатами является обязательным. Тем самым будут идентифицированы недетерминированные части программы и сэкономлено время на отладку.

2.6.5. ДИРЕКТИВА NOWARNINGS

Одним из характерных симптомов ошибки программирования является появление некоторой переменной только один раз (чаще всего из-за орфографической ошибки). Обычно, если нам не нужно значение переменной, следует использовать анонимную переменную (подчеркивание).

Компилятор обычно выдает предупреждение при обнаружении того, что какая-то переменная упомянута только один раз; отключить такого рода предупреждения можно путем задания в самом начале программы директивы nowarnings.

Примечание: Рекомендуется все же избегать, где только возможно, использования директивы nowarnings.

В языке TURBO PROLOG предусмотрены директивы компилятора, которые можно использовать для управления выдачей сообщений об ошибках при выполнении программ (файлов . EXE). Эти директивы позволяют выбирать:

- должны ли генерироваться программные коды, контролирующие переполнение целых;
- уровень детализации сообщений об ошибках;
- должны ли генерироваться программные коды автоматического контроля стека.

Соответствующие директивы можно поместить в начало программы или выбрать в меню Options/Compiler Directives (элементы Run-Time Check и Error Level).

2.6.6. ДИРЕКТИВА ERRORLEVEL

В языке TURBO PROLOG предусмотрена директива errorlevel, позволяющая управлять степенью детализации сообщений об ошибках. Она имеет следующий синтаксис:

errorlevel=d, где d может иметь значение 0,1 или 2, представляя один из следующих уровней:

- 0 - На этом уровне генерируются наиболее эффективные коды; он соответствует стратегии выдачи сообщений об ошибках в версии 1.0.
- 1 - Этот уровень принимается по умолчанию. Когда возникает ошибка, TURBO PROLOG сообщает о ее источнике(имя модуля и включаемый файл) в окне Message. Показывается также соответствующее ошибке место в исходном тексте программы, выраженное в виде числа байтов от начала файла программы. Если после загрузки текста программы в редактор нажать Shift-F2, ввести это значение, а затем снова нажать Shift-F2, то курсор переместится к месту возникновения ошибки.
- 2 - На этом уровне выдаются сообщения о некоторых ошибках, не затрагиваемых уровнем 1, таких, как переполнение стека, переполнение динамической области памяти(“кучи”), переполнение “хвоста” и т. п.

Уровень детализации сообщений об ошибках можно выбрать и с помощью элемента меню Error level; None соответствует уровню 0,Default - уровню 1, а Maximum - уровню 2.

В проекте директива errorlevel контролирует выдачу сообщений об ошибках в каждом модуле. Однако, если она задается в главном модуле со значением, превышающим значения в других модулях, то это может приводить к выдаче путанных сообщений.

Если, к примеру, ошибка возникает в модуле, который компилировался с errorlevel=0, а главный модуль при этом компилировался со значением 1 или 2, то система не сможет правильно указать место возникновения ошибки; она укажет на место, соответствующее уже отработавшим кодам.

2.6.7. ОПЦИИ КОМПИЛЯТОРА ИЗ МЕНЮ

В среде TURBO PROLOG предусмотрены две опции компилятора для динамической диагностики программ, доступные только через меню; для них нет эквивалентных директив компилятора. Это опции Integer Overflow Check (контроль переполнения целых) и Stack Check (контроль стека), являющиеся элементами меню Options /Compiler Directives/Runtime Check.

Опция Integer Overflow Check.

Данная опция указывает компилятору на необходимость генерации кодов, контролирующих переполнение целых значений (значения, превышающие 32767, дают переполнение).

Если программист знает, что целые в его программе не могут переполниться, то он вполне может эту опцию не использовать (это принимается по

умолчанию). Если же вероятность переполнения целых есть, можно установить Integer Overflow Check в состояние “включено” (On), и тогда программа при возникновении переполнения выдает соответствующее предупреждение.

Опция Stack Check.

Данная опция указывает компилятору на необходимость генерации кодов, контролирующих переполнение стека. При наличии в программе этих кодов в результате переполнения стека будет выдаваться сообщение об ошибке, а аварийного завершения программы не будет.

Для перехвата этого сообщения можно использовать предикат trap.Чтобы соответствующие программные коды генерировались, необходимо “включить”(On) элемент меню Stack Check.

2.7. СРАВНЕНИЕ И МАТЕМАТИКА

В языке программирования TURBO PROLOG имеется широкий набор арифметических операций, математических функций и предикатов, логических операторов.

2.7.1. СРАВНЕНИЕ

В языке TURBO PROLOG могут сравниваться математические выражения, а также значения типов integer, real, char, string и symbol.

Следующее утверждение “X плюс 4 меньше, чем 9 минус Y” эквивалентно выражению X + 4 < 9 - Y.

Оператор “меньше” (<) устанавливает отношение между двумя выражениями, X+4 и 9-Y. В языке TURBO PROLOG используется инфиксная запись, означающая, что операторы размещаются между операндами (скажем,X+4), а не перед ними (скажем,+(X, 4)).

Операторы сравнения

Обозначение	Отношение
<	меньше
<=	меньше или равно
=	равно
>	больше
>=	больше или равно
<> или ><	отличается(или не равно)

В языке TURBO PROLOG равенство (=) может выступать в роли операратора и в роли предиката. Если слева от знака (=) стоит несвязанная значением переменная, то это будет оператор.

Пример:

N=N1-2.

Пусть N - свободная переменная, оператор может быть удовлетворен путем связывания N значением N1-2. Это в какой-то степени соответствует тому, что в других языках называется оператором присваивания. Заметим, что у N1 всегда должно быть значение, поскольку эта переменная - часть вычисляемого выражения.

Если слева от знака (=) стоит связанная значением переменная или значение константы, то это будет предикат, который дает истину при полном совпадении того, что стоит слева и права.

При использовании предиката = для сравнения вещественных чисел следует учитывать, что должна быть соблюдена точность их представления; в противном случае могут возникнуть непредсказуемые результаты.

Например, 7/3*3=7 дает неуспех. При сравнении двух вещественных чисел на равенство лучше проверять их попадание в один определенный диапазон.

Помимо числовых выражений, сравнивать можно и отдельные элементы типа char, string и symbol. Рассмотрим следующие сравнения:

‘a’<’b’ ..... char

“антонина”>”антоний” ..... string

P1=петр, P2=софья, P1>P2 ..... symbol

TURBO PROLOG преобразует ‘a’<’b’ в эквивалентное арифметическое выражение 97<98, используя коды ASCII сравниваемых литер. При сравнении двух символьных констант или строк результат зависит от сравнения символов в соответствующих позициях. Результат сравнения будет таким же, как результат сравнения первых символов, если только они не равны. Если равны, то сравниваются следующие, и так далее. Результатом сравнения “антонина”>”антоний” будет “истина”, поскольку в соответствующей позиции код буквы “н” больше кода буквы “й”. Истинным будет и сравнение “аа”>”а”. А вот выражение “петр”>”софья” будет ложным, поскольку в действительности код “п” меньше кода “с”. Именно сравнение этих кодов даст “ложь”. Значения типа symbol непосредственно сравнивать из-за их синтаксиса нельзя. Чтобы можно было их сравнить, они должны быть связаны с переменными или записаны в виде строк.

2.7.2. АРИФМЕТИЧЕСКИЕ ОПЕРАТОРЫ И ВЫРАЖЕНИЯ

Арифметические выражения состоят из операндов(чисел и переменных), операторов (+, -, *, /, div и mod) и скобок.Символы справа от знака равенства (представляющего собой предикат =) являются арифметическим выражением:

A=1+6/(11+3)*Z

Шестнадцатиричные числа могут представляться с указанием перед значением знака доллара. Например,

$FFF=4095
86=$4A+12

Значение выражения может быть получено только в том случае, если во время его вычисления все переменные оказываются связанными. Тогда вычисление производится в определенном порядке, определяемом приоритетом арифметических операторов; операторы с наивысшим приоритетом вычисляются первыми.

TURBO PROLOG может выполнять все четыре основные арифметические операции (сложение, вычитание, умножение и деление) применительно к целым и вещественным значениям; тип результата определяется в соответствии с таблицей.

Операнд1	Оператор	Операнд2	Результат
целое	+, -, *	целое	целое
вещественное	+, -, *	целое	вещественное
целое	+, -, *	вещественное	вещественное
вещественное	+, -, *	вещественное	вещественное
целое или вещественное	/	целое или вещественное	вещественное
целое	div	целое	целое
целое	mod	целое	целое

Арифметические выражения вычисляются в следующем порядке:

1. Если выражение содержит подвыражения в скобках, то сначала вычисляются подвыражения.
2. Если в выражении есть умножение (*) или деление (/, div или mod), то следующими выполняются эти операторы слева направо.
3. Наконец, последними в порядке слева направо выполняются операторы сложения (+) и вычитания (-).

В выражении A=1+6/(11+3)*Z предположим, что Z=4, поскольку переменные должны быть связаны перед вычислением выражения.

1. Первым вычисляется подвыражение в скобках (11+3); получаем значение 14.
2. Затем вычисляется 6/14, поскольку / и * вычисляются слева направо; получаем 0. 428571.
3. Затем 0. 428571*4 дает 1. 714285.
4. Наконец, вычисляя 1+1. 714285, получаем окончательное значение 2.714285.

Если A принадлежит домену real, то значение его будет 2. 714285, но если A принадлежит домену integer, то значением будет 3 (поскольку результат округляется до ближайшего целого).

Предшествование операторов

Оператор	Приоритет
+ -	1
* / mod div	2
- + (унарные)	3

Оператор mod находит остаток от целочисленного деления (X и Y - целые).

```
X mod Y   /* (i, i) */
```

Выражение Z=X mod Y связывает Z значением результата. Например,

```
Z = 7 mod 4          /* Z станет равной 3  */
Y = 4 mod 7          /* Y станет равной 0  */
```

Оператор div выполняет целочисленное деление X/Y (X и Y-целые).

```
X div Y              /* (i, i) */
```

Выражение Z=X div Y связывает Z значением целой части результа та. Например,

```
Z = 7 div 4      /* Z станет равной 1 */
Z = 4 div 7      /* Z станет равной 0 */
```

- **Математические предикаты и функции.**

В языке TURBO PROLOG предусмотрены два стандартных предиката для генерации случайных чисел. Один дает случайное вещественное число в диапазоне от 0 до 1, а второй - случайное число в диапазоне от 0 до заданного целого.

- **Предикат random/1.**

Эта версия предиката random предназначается для получения случайного вещественного числа Rndreal, удовлетворяющего ограничениям 0 <=Rndreal < 1. random/1 имеет следующий формат:

```
random(Rndreal)      /* (o) */
```

- **Предикат random/2.**

У этой версии предиката random два аргумента; записывается он в следующем формате:

```
random(Max, Rndint)  /* (i, o) */
```

Переменная Rndint связывается случайным целым значением, удовлетворяющим ограничениям 0 <= Rndint < Max random/2 значительно быстрее random/1, поскольку использует только целую арифметику.

- **Функция abs/1.**

Функция abs дает абсолютное значение аргумента.

```
abs(X)               /* (i) */
```

Выражение Z=abs(X) связывает Z (если она свободна) значением результата или возвращает “Истина/Ложь”, если Z уже связана значением. Например:

```
Z = abs(-7)          /* Z станет равной 7 */
```

- **Функция cos/1.**

Функция cos вычисляет косинус аргумента.

```
cos(X)               /*  (i)  */
```

Выражение Z=cos(X) связывает Z (если она свободна) значением результата или возвращает “Истина/Ложь”, если Z уже связана значением. Например: Pi = 3.141592653,

```
Z = cos(Pi)          /* Z станет равной -1 */
```

- **Функция sin/1.**

Функция sin вычисляет синус аргумента.

- **sin(X) /* (i) */**

Выражение Z=sin(X) связывает Z (если она свободна) значением результата или возвращает “Истина/Ложь”, если Z уже связана значением. Например: Pi = 3.141592653,

```
Z = sin(Pi)          /* Z станет равной 0 */
```

- **Функция tan/1.**

- **Функция tan вычисляет тангенс аргумента.**

```
tan(X)                  /*  (i)  */
```

Выражение Z=tan(X) связывает Z (если она свободна) значением результата или возвращает “Истина/Ложь”, если Z уже связана значением. Например: Pi = 3. 141592653,

```
Z=tan(Pi)               /* Z станет равной 0 */
```

- **Функция arctan/1.**
- **Функция arctan вычисляет арктангенс вещественного числа, являющегося значением X.**

```
arctan(X)               /*  (i)  */
```

Выражение Z=arctan(X) связывает Z (если она свободна) значением результата или возвращает “Истина/Ложь”, если Z уже связана значением. Например: Pi = 3.141592653,

```
Z = arctan(Pi)          /* Z станет равной 1. 2626272556 */
```

- **Функция exp/1.**

Функция exp вычисляет e в степени X.

```
exp(X)                  /* (i) */
```

Выражение Z=exp(X) связывает Z (если она свободна) значением результата или возвращает “Истина/Ложь”, если Z уже связана значением. Например:

```
Z = exp(2. 5)           /* Z станет равной 12. 182493961 */
```

- **Функция ln/1.**

Функция ln вычисляет натуральный логарифм X (основание e).

```
ln(X)                   /* (i) */
```

Выражение Z=ln(X) связывает Z (если она свободна) значением результата или возвращает “Истина/Ложь”, если Z уже связана значением. Например:

```
Z = ln(12. 182493961)           /* Z станет равной 2. 5 */
```

- **Функция log/1.**

Функция log вычисляет десятичный логарифм X.

```
log(X)                  /* (i) */
```

Выражение Z=log(X) связывает Z (если она свободна)значением результата или возвращает “Истина/Ложь”, если Z уже связана значением. Например:

```
Z = log(2. 5)           /* Z станет равной 0. 39794000867 */
```

- **Функция sqrt/1.**

Функция sqrt вычисляет квадратный корень из X.

```
sqrt(X)                 /* (i) */
```

Выражение Z=sqrt(X) связывает Z (если она свободна) значением результата или возвращает “Истина/Ложь”, если Z уже связана значением. Например:

```
Z = sqrt(25)            /* Z станет равной 5 */
```

- **Функция round/1.**

Функция round округляет значение X.

```
round(X)                /* (i) */
```

Округление X осуществляется до ближайшего целого. Например:

```
Z1 = round(4. 51)       /* Z1 станет равной 5 */
Z2 = round(3. 50)       /* Z2 станет равной 3 */
```

- **Функция trunc/1.**

Функция trunc усекает X справа до десятичной точки.

```
trunc(X)                /* (i) */
```

- **Например:**

```
Z = trunc(4. 7)         /* Z станет равной 4 */
```

2.8. ПРЕДИКАТЫ ВВОДА/ВЫВОДА

2.8.1. ПРЕДИКАТЫ ВВОДА

В языке TURBO PROLOG для ввода предусмотрено несколько стандартных предикатов; пять базовых - это readln (чтение строк символов), readint, readreal, readchar (чтение целых, вещественных и символов соответственно) и

readterm (чтение составных объектов). Все эти предикаты могут быть переориентированы на чтение из файлов.

Ряд других, более специальных предикатов, принадлежащих к категории предикатов ввода-inkey (чтение символа, вводимого с клавиатуры), keypressed (проверка того, нажата ли клавиша), unreadchar(выталкивание символов назад в буфер клавиатуры). При перенаправлении ввода эти специальные предикаты не затрагиваются.

- **Предикат readln/1.**

Предикат readln обеспечивает чтение текстовой строки; записывается он в следующем формате:

```
readln(Str)            /* (o) */
```

Домен переменной Str должен быть либо string, либо symbol. Перед вызовом readln переменная Str должна быть свободной. Предикат readln обеспечивает чтение до 127 символов (плюс возврат каретки) с клавиатуры или до 64Кбайт с других устройств. Если при вводе с клавиатуры нажать Esc, readln даст неудачный исход.

- **Предикаты readint/1.**

Предикат readint читает целое значение; записывается он в следующем формате:

```
readint(X)             /* (o) */
```

Домен переменной X должен быть integer, а перед вызовом эта переменная должна быть свободной. Предикат readint будет читать целое значение с текущего устройства ввода (возможно, с клавиатуры) до нажатия клавиши Enter. Если вводимая строка не соответствует синтаксису целых, то readint дает неудачный исход, а TURBO PROLOG обращается к своему механизму возврата. Если в процессе ввода с клавиатуры нажимается Esc, то readint также дает неудачный исход.

- **Предикат readreal/1.**

Предикат readreal работает соответственно своему имени: он читает число типа real(в отличие от readint, читающего число типа integer); записывается он в следующем формате:

```
readreal(X)            /* (o) */
```

Домен переменной X должен быть real, а перед вызовом эта переменная должна быть свободной. Предикат readreal будет читать вещественное значение с текущего устройства ввода (возможно, с клавиатуры) до нажатия клавиши Enter. Если вводимая строка не соответствует синтаксису вещественных, то readreal дает неудачный исход. Если в процессе ввода с клавиатуры нажимается Esc, то readreal также дает неудачный исход.

- **Предикат readchar/1.**

Предикат readchar читает символ с текущего устройства ввода; записывается он в следующем формате:

```
readchar(Sb)           /* (o) */
```

Переменная Sb должна быть перед вызовом readchar свободной переменной типа char. Если текущий ввод осуществляется с клавиатуры, то readchar будет ожидать ввода символа. Если же в это время нажимается Esc, readchar дает неудачный исход.

- **Предикат readterm/2.**

Предикат readterm читает составной терм и преобразует его в объект; записывается он в следующем формате:

```
readterm(ИмяДомена, Терм)      /* (i, i) */
```

Этот предикат вызывается с двумя аргументами: именем домена и термом. Предикат readterm читает строку и преобразует ее в объект данного типа. Если эта строка не представляется в формате, который мог бы быть сформирован с помощью предиката write, то readterm дает ошибку. Предикат readterm может использоваться для обработки термов в текстовых файлах.

- **Предикат inkey/1.**

Предикат inkey читает вводимый с клавиатуры символ; записывается он в следующем формате:

inkey /* (o) */

Переменная С должна принадлежать домену char.

Если при вызове inkey нажимается какая-либо клавиша (или если уже не пуст буфер клавиатуры), то inkey отработает, связав свой аргумент этим символом. Если же никакая клавиша не нажимается, то сразу же следует неудачный исход.

- **Предикат keypressed/0.**

Предикат keypressed проверяет, не нажата ли клавиша на клавиатуре; записывается он в следующем формате:

- **keypressed**

Успешно keypressed отрабатывает в том случае, когда нажата клавиша или не пуст буфер клавиатуры. Из буфера код клавиши не удаляется.

- **Предикат unreadchar/1.**

Предикат unreadchar выталкивает символы во внутренний буфер клавиатуры; записывается он в следующем формате:

unreadchar(C) /* (i) */

Размер буфера - 128 байт.Если происходит его переполнение, то unreadchar выдает сообщение об ошибке.

2.8.2. ПРЕДИКАТЫ ВЫВОДА

В языке TURBO PROLOG для вывода предусмотрены три стандартных предиката. Это предикаты write, nl и writef.

- **Предикаты write и nl.**

Предикат write можно вызывать с произвольным числом аргументов:

write(Пар1, Пар2, Пар3, . . . , ПарN) /* (i, i, i, . . . , i) */

Этими аргументами могут быть либо константы из стандартных доменов, либо переменные. Если это переменные, то они должны быть входными параметрами (т. е. связанными). В предикатах вывода предусмотрено использование специальных символьных констант:

\n выдается пробел;

\t переход к следующей позиции табуляции;

\число-код ASCII выводимого символа выводит символ по коду.

Стандартный предикат nl (new line - новая строка) часто используется вместе с write, генерируя переход на новую строку в изображении на экране.

- **Предикат writef.**

Предикат writef позволяет генерировать форматированный вывод; он записывается в следующем формате:

writef(СтрокаФормата, Арг1, Арг2, . . . , АргN) /* (i, i, i, . . . , i) */

Арг1...АргN должны быть константами или связанными переменными из стандартных доменов; составные домены форматироваться не могут. Строка формата состоит из обычных символов и указателей формата; обычные символы выводятся без модификаций, а указатели формата имеют следующий вид:

%-m. pf

Символы, идущие в указателях формата за знаком %, не являются обязательными и имеют следующий смысл:

• **дефис (-)**	• **Указывает, что поле должно выравниваться слева (выравнивание справа принимается по умолчанию).**
• **полем**	• **Десятичное число - минимальная длина поля.**

• поле.р	• Десятичное число - точность значения с плавающей точкой, либо максимальное число символов строки, которые должны выводиться.
• поле f	• Определяет форматы, отличные от принимаемого по умолчанию для данного объекта. Например, поле f может указывать, что целые должны выводиться как числа без знака или шестнадцатеричные числа, или вещественные должны выводиться в шестнадцатеричном виде.

- **TURBO PROLOG распознает в поле f следующие указатели формата:**

	• вещественные как десятичные с фиксированной точкой (скажем, 123. 4 или 0. 004321);
	• вещественные в экспоненциальном виде (скажем, 1. 234e2 или 4. 321e-3);
	• вещественные в коротком формате f или e (для вещественных значений принимается по умолчанию);
	• символы или целые как десятичные числа;
	• символы или целые как целые без знака;
	• символы или целые как шестнадцатиричные числа;
	• символы или целые как символы;
	• как число обращения к базе данных (только для домена ref);
	• как длинное шестнадцатиричное число (строки, числа обращения к базе данных);
	• как строка (типы symbol и string).

•

Пример:

```
goal
  A=один,
  B=330. 12,
  C=4. 3333375,
  D="один два три",
  writef("A= '%-7'\nB= '%8. 1e'\n", A, B),
  writef("A= '%'\nB= '%8. 4e'\n", A, B), nl,
  writef("C= '%-7. 7g'\nD= '%8. 8'\n", C, D),
  writef("C= '%-7. 0f'\nD= '%0'\n", C, D),
  writef("симв:%c,дес:%d,шестн:%x,беззн:%u",97,'a',33,
         -1).
```

- **После запуска программа выведет:**

```
A= 'один   '
B= ' 3. 3E+02'
A= 'один'
B= '3. 3012E+02'
C= '4. 3333375'
D= 'один два'
C= '4'
D= 'один два три'
симв: a, дес: 97, шестн: 21, беззн: 65535
```

3. ОСНОВНЫЕ МЕХАНИЗМЫ СРЕДЫ TURBO PROLOG 2.0

Любая PROLOG - система включает в себя машину логического вывода, которая представляет собой процесс логического рассуждения об информации. Для сознательного программирования на языке PROLOG необходимо познакомиться с основными механизмами логического вывода.

3.1. СОПОСТАВЛЕНИЕ (УНИФИКАЦИЯ)

Машина логического вывода включает механизм сопоставления с образцом, с помощью которого осуществляется поиск хранимой (известной) информации путем приведения к соответствию вопросов и ответов. PROLOG пытается вывести, что некоторая гипотеза истинна (другими словами, ответить на поставленный вопрос), "опрашивая" определенную совокупность информации, о которой уже известно, что она истинна. Знания PROLOG - системы - это конечная совокупность фактов(и правил), которые задаются применительно к программе.

Сопоставление (унификация) подразумевает выполнение следующих действий: передача параметров, выбор из нескольких вариантов, построение структур, доступ к структурам и присваивание.

Сопоставление (унификация) - это процесс поиска решения, на вход которого подаются два терма или предиката, а он проверяет, оответствуют ли они друг другу. Если они не сопоставимы, говорят, что этот процесс терпит неуспех. Если же они сопоставимы, тогда процесс находит конкретизацию переменных (присваивание), делающую их тождественными, и завершается успешно.

Общие правила сопоставления термов:

1) Если S и T - константы, то S и T сопоставимы, только если они являются одним и тем же объектом.

2) Если S - переменная, а T - произвольный объект, то они сопоставимы, и S присваивается значение T. Наоборот, если T - переменная, а S - произвольный объект, то T присваивается значение S.

3) Если S и T - структуры, то они сопоставимы только если: S и T имеют одинаковый функтор и все их соответствующие компоненты сопоставимы. Результирующая конкретизация определяется сопоставлением компонент.

Сопоставления при достижении цели:

Цель может быть достигнута, если все предикаты (кроме стандартных) целевого предложения могут сопоставиться с головой какого-нибудь предложения программы. Если сопоставление происходит с фактом, то несвязанным переменным присваиваются значения, и процесс успешно завершается. Если же сопоставление происходит с головой правила, то цель будет достигнута только тогда, когда каждый предикат в теле этого правила сопоставится с каким - нибудь предложением, описывающим его. В целевом предложении для сопоставления предикаты вызываются в том порядке, в котором они записаны.

Просмотр предложений программы для сопоставления осуществляется в порядке их следования при описании предикатов в разделе clauses.

Рассмотрим пример программы, которая позволяет установить, кому что нравится. В ней определен предикат likes(string, symbol), арности-2.

Программа состоит из фактов, предусматривает постановку внешней цели и будет исполняться в режиме трассировки.

```
trace
predicates
  likes(string, symbol)
clauses
        likes("Елена", reading).
        likes("Иван", computers).
        likes("Иван", badminton).
        likes("Леонид", badminton).
        likes("Артем", swimming).
```

likes(“Артем”, reading).

При запуске этой программы в окне Dialog появится Goal() и система будет ждать постановки цели. Введем цель likes(X,Y) , где X, Y-переменные. Решения будут отображаться в окне Dialog, шаги вывыполнения в окне Trace. Для выполнения каждого последующего шага нужно будет нажимать клавишу F10. Так как перед системой ставится внешняя цель, будут получены все возможные решения. Поиск решения начнется в порядке следования предложений.

Согласно правилам сопоставления поставленная цель сопоставится с первым фактом и переменным присвоятся значения:

Goal: likes(X,Y)
RETURN: *likes(“Елена”, ”reading”) X=Елена, Y=reading.

На следующем шаге система освободит переменные от значений и попытается достичь цель опять. Теперь цель сопоставится со следующим фактом и переменные приобретут новые значения.

REDO: likes(_, _)
RETURN: *likes(“Иван”, ”computers”) X=Иван,Y=computers.

Это в нашей программе повторится с каждым фактом.

REDO: likes(_, _)
RETURN: *likes(“Иван”, ”badminton”) X=Иван,Y=badminton
REDO: likes(_, _)
RETURN: *likes(“Леонид”, ”badminton”) X=Леонид,Y=badminton
REDO: likes(_, _)
RETURN: *likes(“Артем”, ”swimming”) X=Артем, Y=swimming
REDO: likes(_, _)
RETURN: likes(“Артем”, ”reading”) X=Артем, Y=reading
6 Solutions

Поставим перед системой внешнюю цель likes(X,badminton). В этом случае первый аргумент - переменная X, а второй - символьная константа badminton.

Goal: likes(X, badminton)

Попытка сопоставления с первым и вторым фактами даст неуспех, не сопоставятся вторые аргументы.

REDO: likes(_, ”badminton”)
REDO: likes(_, ”badminton”)

Поставленная цель сопоставится с третьим и четвертым фактами, и при этом переменной X будут присвоены значения:

RETURN: *likes(“Иван”, ”badminton”) X=Иван
REDO: likes(_, ”badminton”)
RETURN: likes(“Леонид”,”badminton”) X=Леонид
2 Solutions.

Попытки сопоставления с остальными фактами дадут неуспех. В результате работы программы мы получим два решения:

X=Иван, X=Леонид.

Расширим данную программу правилом:

likes(“Максим”, A):- likes(“Иван”, A).

Это правило можно интерпретировать:”Максиму” нравится то, что нравится “Ивану”. Запустим программу и поставим внешнюю цель likes(“Максим”, X). Попытки сопоставления с фактами дадут неуспех. Не сопоставится первый аргумент предиката.

Goal: likes(“Максим”, Y)
REDO: likes(“Максим”, _)
REDO: likes(“Максим”, _)
REDO: likes(“Максим”, _)
REDO: likes(“Максим”, _)
REDO: likes(“Максим”, _)

Цель сопоставится с головой правила.

REDO: likes(“Максим”, _)

Но для достижения цели, нужно получить истинность всех предикатов в теле правила. Поэтому система обратится к телу правила.

CALL: likes(“Иван”, _)

Процесс поиска решений опять начнется с первого предложения программы, но теперь в качестве цели будет выступать предикат likes(“Иван”, A). Попытка сопоставления с первым фактом даст неуспех.

REDO: likes(“Иван”, _)

Система перейдет к сопоставлению со следующим фактом, которое закончится успешно и переменной A будет присвоено значение “computers”.

RETURN: *likes(“Иван”, ”computers”)

Система вернется к голове правила, которая станет новым фактом программы, сопоставимым с внешней целью. Переменной У будет присвоено значение переменной A и система получит первое решение.

RETURN: likes(“Максим”, ”computers”) Y=computers

После этого система освободит переменные от значений, вернется к телу правила и будет сопоставлять его со следующим фактом.

REDO: likes(“Иван”, _)

В этом случае сопоставление будет успешным и переменной A будет присвоено значение “badminton”.

RETURN: likes(“Иван”, ”badminton”)

Система вернется к голове правила, которая станет новым фактом программы, сопоставимым с внешней целью. Переменной У будет присвоено значение переменной A, и система получит второе решение.

RETURN: likes(“Максим”, ”badminton”) Y=badminton

После этого система освободит переменные от значений, вернется к телу правила и будет сопоставлять его со следующим фактом. Попытки сопоставления тела правила с оставшимися фактами дадут неуспех и система закончит поиск решений. При постановке внешней цели Goal: likes(“Максим”, Y) на основании правила и механизма сопоставления система выведет два новых факта и получит два решения:

likes(“Максим”, ”computers”) Y=computers
likes(“Максим”, ”badminton”) Y=badminton.

При написании программ на языке PROLOG надо учитывать механизм сопоставления, ибо от него зависят результаты работы программ. В описанных примерах наряду с механизмом сопоставления действовал и механизм возврата, благодаря ему система в нашей программе находила все возможное множество решений. В режиме трассировки на возврат указывает слово REDO: .

3.2. ВОЗВРАТ (BACKTRACKING)

Механизм возврата (backtracking или поиск с возвратом) начинает действовать при постановке внешних целей или в теле правила при неуспешном выполнении какого-нибудь предиката. При постановке внешних целей поиск решения начинается с первого предложения, определяющего данный предикат. Если внешняя цель сопоставляется с каким - нибудь предложением, система устанавливает точку возврата с указателем на данное предложение, переменным присваиваются значения, и система получает решение. (При помощи точек возврата фиксируется то место, с которого система начнет поиск следующего решения.) После этого система освобождает переменные от текущих и продолжает поиск других решений.

При постановке внутренних целей (в разделе goal) после получения первого решения дальнейший поиск решений прекращается, т.е. механизм возврата не действует, если не применять специальные меры.

При поиске решения в теле правила на предикаты, которые в своем определении содержат несколько предложений, система устанавливает точки возврата (такие предикаты могут приводить к альтернативным решениям). Процесс сопоставления предиката в теле правила начнется с первого предложения, описывающего этот предикат, и будет проходить в порядке их

следования. Если сопоставление предиката терпит успех, система ставит точку возврата, несвязанные переменные - аргументы предиката приобретают значения, и система переходит к следующему предикату в теле правила. Если процесс сопоставления предиката в теле правила терпит неуспех, система делает возврат на предшествующую точку возврата, т.е. на предшествующий предикат, освобождает переменные - аргументы этого предиката от их значений и продолжает сопоставление этого предиката с предложением, следующим за предложением, на которое установлена точка возврата. Если повторно данный предикат не сопоставится, все предложения уже будут пройдены, то система совершит возврат в правиле на предшествующую точку возврата. Когда дело доходит до последней точки возврата, все ранее связанные переменные освобождаются от значений, и система переходит к следующему предложению программы, если оно есть, а если все предложения исчерпаны, и цель ранее не достигалась, процесс поиска решений прекращается, и цель терпит неуспех. Если цель раннее достигалась, то прекращается поиск новых решений .

Пример :

```
trace
predicates
        likes(string, string)
        tastes(string, string)
        food(string)
clauses
        likes("Олег",X) :- food(X),
                           tastes(X,"хороший").

        tastes("пицца","хороший" ).
        tastes("капуста", "плохой").

        food("капуста").
        food("пицца").
```

В данной программе поставим директиву компилятора trace для исполнения в режиме трассировки.

Эта простая программа состоит из двух наборов фактов и одного правила. Правило, представляющее отношение likes ("нравится"), просто констатирует, что Олегу нравится пища, имеющая хороший вкус.

Чтобы увидеть, как работает механизм поиска с возвратом, зададим внешнюю цель. Загрузим программу в память, скомпилируем ее в режиме главного меню, а затем в ответ на подсказку GOAL: введем: likes("Олег",X).

PROLOG предпринимает попытку удовлетворить цель, начиная поиск соответствия с верхней части программы (с первого предложения).

```
Goal: likes("Олег", X)
CALL: likes("Олег",  _)
```

Поставленная цель сопоставляется с головой правила. Сопоставление с головой правила "заставляет" TURBO PROLOG пытаться удовлетворить все правило. При этом сначала вызывается предикат этого правила

```
CALL:  food(_) .
```

Система начнет поиск с первого предложения, описывающего предикат food. Это предложение будет фактом (food("капуста").), сопоставимым с предикатом food(X). Сопоставление произойдет успешно, и переменная X свяжется значением "капуста".

```
RETURN:  *food("капуста")
```

Но ответов на вызов food(X) больше одного, и TURBO PROLOG устанавливает точку возврата на факт food("капуста"). При помощи этой точки возврата фиксируется то место, с которого система начнет поиск другого решения для food(X).

После сопоставления с фактом, система осуществляет возврат в правило и переходит к следующему предикату правила.

С учетом того, что значением X теперь является “капуста”, следующий вызов принимает вид

CALL: tastes(“капуста”, ”хороший”).

Поиск в попытке удовлетворить этот вызов TURBO PROLOG начинает с первого предложения, описывающего предикат tastes.

REDO: tastes(“капуста”, ”хороший”)

Поскольку при таких значениях аргументов предикат не сопоставится ни с одним фактом, вызов завершается неудачно, а TURBO PROLOG обращается к своему механизму автоматического возврата.

FAIL: tastes(“капуста”, ”хороший”)

Когда начинается отработка поиска с возвратом, система отходит к последней установленной в предложении точке возврата. В рассматриваемом случае это food(“капуста”). (Поскольку переменная в выражении связана, единственный способ освободить ее - это возврат.) Когда система отступает в точку возврата, все связанные после этой точки переменные освобождаются от значений и начинается поиск другого решения первоначальной цели.

REDO: food(_)

Теперь система пытается снова решить этот вызов, начиная с точки возврата, вернувшись она переходит к следующему факту.

RETURN: food(“пицца”)

Сопоставление с фактом происходит успешно, переменная X связывается значением “пицца”. Система приходит к следующему предикату с новым значением переменной. Делается новый вызов,

CALL: tastes(“пицца”, ”хороший”)

и поиск снова начинается с первого предложения, описывающего предикат tastes. Теперь сопоставление произойдет успешно, потому что в программе есть факт, сопоставимый с предикатом tastes при таких значениях аргументов.

RETURN: tastes(“пицца”, ”хороший”)

Так как сопоставление всех предикатов правила произошло успешно, система возвращается к голове правила, передает значение переменной и получает факт, сопоставимый с внешней целью. Сопоставление цели с фактом, полученным на основании правила и других фактов, произойдет успешно, следовательно, цель достигается. Система выводит на экран решение и ждет постановки новой цели.

```
RETURN: likes(“Олег”, ”пицца”)
X=пицца
1 Solution
Goal:
```

Как уже отмечалось ранее, используя поиск с возвратом, TURBO PROLOG будет искать не только первое решение задачи, а все возможные решения.

Рассмотрим программу, которая содержит факты об именах и возрасте некоторых игроков теннисного клуба.

Пример:

```
trace
 domains
    child= symbol
    age = integer
 predicates
    player(child, age)
 clauses
                player(“Петр”, 9).
                player(“Павел”,10).
                player(“Роман”,9).
                player(“Ольга”,9).
```

Используем TURBO PROLOG для подготовки турнира по настольному теннису между девятилетними членами клуба. Для каждой пары игроков должно быть предусмотренно по две игры. Цель будет заключаться в нахождении всех возможных пар игроков, которым по девять лет. Для этого логично задать составную внешнюю цель:

```
player(P1,9),
player(P2,9),
P1<>P2.
```

(На естественном языке: Найти P1 (9лет) и P2 (9 лет) таких,чтобы P1 отличался от P2).

Будем исполнять программу в режиме трассировки. На запрос системы введем внешнюю цель.

Goal: player(P1,9), player(P2,9), P1<>P2.

Для более понятного объяснения процесса исполнения программы условимся предикаты целевого предложения называть подцелями.

1. TURBO PROLOG попытается найти решение первой подцели (предикат player(P1,9)), вызвав ее для сопоставления,

CALL: player(_, 9)

а ко второй подцели (предикат - player(P2,9)) перейдет только после достижения первой.

На первых предикатах целевого предложения система поставит точки возврата, так как эти предикаты описываются несколькими предложениями.

Первая подцель сопоставляется с первым фактом, переменная P1 связывается значением “Петр”, система поставит точку возврата на этом факте, вернется в целевое предложение и перейдет к следующей подцели (предикат-player(P2,9)).

RETURN: *player(“Петр”,9)

CALL: player(_,9)

Для второй подцели поиск решения тоже начнется с первого предложения программы. Она сопоставится тоже с первым фактом, переменная P2 свяжется значением “Петр”, система поставит точку возврата на этот факт, вернется в целевое предложение и перейдет к следующей подцели P1<>P2.

RETURN: *player(“Петр”,9)

2. Поскольку значения у обеих этих переменных “Петр”, подцель не достигается. Ввиду этого TURBO PROLOG возвращается к предыдущей подцели, освобождает от значения переменную P2 и ищет другое ее решение.

REDO: player(_,9)

Используя точку возврата, система пытается сопоставить подцель со следующим фактом. Сопоставление терпит неуспех, потому что не сопоставляются вторые аргументы.

Система переходит к сопоставлению со следующим фактом. В этом случае сопоставление завершается успешно. Переменной P2 присваивается значение “Роман”, система поставит точку возврата на этот факт, вернется в целевое предложение и перейдет к следующей подцели P1<>P2.

RETURN: *player(“Роман”, 9)

3. Эта подцель удовлетворяется текущими значениями переменных. Таким образом, достигается конечная цель, и система получает решение P1=Петр, P2=Роман (турнирная пара - “Петр” и “Роман”).

4. Однако, поскольку TURBO PROLOG должен найти все возможные решения, он возвращается к предыдущей подцели и требует удовлетворить ее снова.

REDO: player(_, 9)

Поскольку для нее находится новое решение “Ольга”, TURBO PROLOG пытается выполнить снова и третью цель. Она достигается и находится новое решение всей задачи.

RETURN: player(“Ольга”,9)

P1=Петр, P2=Ольга

5. Пытаясь найти еще решения, TURBO PROLOG снова возвращается ко второй подцели, но все факты для нее уже исчерпаны, ввиду чего TURBO PROLOG продолжает возврат и переходит к первой подцели. В следствии возврата переменные P1 и P2 освобождаются от значений.

REDO: player(_,9)

Сопоставление первой подцели начнется с точки возврата. Она установлена на первом факте. Попытка сопоставления со вторым фактом окончится неуспехом. Система перейдет к сопоставлению со следующим фактом, которое успешно завершится, и теперь ее решением будет (в качестве значения переменной P1) “Роман”.

RETURN: *player(“Роман”,9)

Установив точки возврата, система перейдет к второй подцели, поиск решения для нее начнется с первого факта. Решением становится “Петр”.

RETURN: *player(“Петр”,9)

Третья подцель также достигается, и мы получаем новое решение задачи: P1=Роман, P2=Петр (создается другая турнирная пара - “Роман” и “Петр”).

6. В поиске еще одного решения TURBO PROLOG возвращается ко второй подцели правила. Здесь P2 связывается значением “Роман”.

REDO: player(_,9)
RETURN: *player(“Роман”,9)

После этого происходит переход к третьей подцели.Она не достигается, поскольку значения переменных P1 и P2 равны. Делается возврат на шаг и ищется другое решение второй подцели.

REDO: player(_,9)

Таковым становится значение “Ольга”. В этом случае достигается и третья подцель, давая новую турнирную пару - “Роман” и “Ольга”.

RETURN: player(“Ольга”,9)
P1=Роман, P2=Ольга

7. И снова продолжается поиск всех решений. TURBO PROLOG возвращается ко второй подцели, но не достигает ее (больше нет фактов).

REDO: player(_,9)

Происходит возврат к первой подцели. Новым значением переменной P1 становится “Ольга”.

RETURN: player(“Ольга”,9)
CALL: player(_,9)

При попытке удовлетворить вторую подцель система опять начинает сопоставление с первого факта. Переменная P2 связывается значением “Петр”.

RETURN: *player(“Петр”,9)

Третья подцель после этого достигается, и в результате получается пятая турнирная пара игроков.

P1=Ольга, P2=Роман

8. И опять после возврата ко второй подцели находится решение “Роман”. Третья цель достигается, и получается шестая турнирная пара игроков, завершающая все возможные решения.

REDO: player(_,9)
RETURN: *player(“Роман”,9)
P1=Ольга, P2=Роман

9. В самом конце предпринимается попытка связать обе переменные значением “Ольга”, но при этом третья подцель не достигается.

REDO: player(_,9)
RETURN: player(“Ольга”,9)

TURBO PROLOG должен теперь вернуться ко второй подцели, но вариантов ее достижения уже не остается. Аналогичная ситуация возникает и с первой

подцелью. Больше решений найти нельзя, так что программа завершается. Итак, если ввести для программы цель

```
player(P1,9),
player(P2,9),
P1<>P2.
```

TURBO PROLOG должен дать следующий результат :

```
P1=Петр,    P2=Роман
P1=Петр,    P2=Ольга
P1=Роман,   P2=Петр
P1=Роман,   P2=Ольга
P1=Ольга,   P2=Петр
P1=Ольга,   P2=Роман
6 Solutions
```

Заметим, что при возвратах TURBO PROLOG вполне может находить избыточные решения.

ВОПРОСЫ К ПРОГРАММЕ:

Какие решения можно получить, если в этой программе поставить цели:

```
1) player(P1,9),
   player(P2,9).
2) player(P1,9),
   player(P2,10).
```

В предыдущих примерах перед системой ставились внешние цели, они инициировали механизм возврата для нахождения всех возможных решений.

Теперь рассмотрим пример программы с использованием внутренних целей.

```
predicates
        type(symbol,symbol)
        is_a(symbol,symbol)
        lives(symbol,symbol)
        can_swim(symbol)
goal
        can_swim(Who),
        write( Who,“-умеет плавать.”).
clauses
        type(“непарнокопытное”,”животное”).
        type(“рыба”,    “животное”).

        is_a(“зебра”,”непарнокопытное”).
        is_a(“сельдь”,”рыба”).
        is_a(“акула”, “рыба”).

        lives(“зебра”,”на_земле” ).
        lives(“лягушка”,“на_земле”).
        lives(“лягушка”,”в_воде” ).
        lives(“акула”,“в_воде”).

        can_swim(Y) :-type(X,”животное”), is_a(Y,X),
                      lives(Y,“в_воде”).
```

В этом примере для иллюстрации поиска с возвратом используется внутренняя цель. Когда программа компилируется и запускается на выполнение, TURBO PROLOG автоматически начинает поиск цели, пытаясь удовлетворить все подцели раздела GOAL.

1. TURBO PROLOG вызывает предикат can_swim(Who) со свободной переменной Who.

```
CALL: goal()
CALL: can_swim(_)
```

Пытаясь решить этот вызов, TURBO PROLOG начинает поиск для сопоставления данного предиката, с первого предложения описывающего его. Со-

поставление происходит с головой правила, в результате переменная Who сопоставляется с переменной Y.

2. Затем TURBO PROLOG предпринимает попытку удовлетворить тело правила; он вызывает первую подцель тела

CALL: type(_, ”животное”) ,

и пытается сопоставить ее с фактами. Поиск для сопоставления начинается с первого факта, описывающего этот предикат. Сопоставление проходит успешно с первым фактом.

RETURN: *type(“непарнокопытное”, ”животное”)

3. Переменная X связывается значением “непарнокопытное”. Поскольку имеется более одного решения, TURBO PROLOG ставит точку возврата на этот факт и возвращается в правило .

4. Со значением X, равным “непарнокопытное”, TURBO PROLOG вызывает вторую подцель правила

CALL: is_a(_, ”непарнокопытное”)

и снова начинается поиск для сопоставления. Эта подцель успешно сопоставляется первым же фактом.

RETURN: *is_a(“зебра”, ”непарнокопытное”)

Y связывается значением “зебра”, и TURBO PROLOG формирует новую точку возврата is_a(“зебра”, ”непарнокопытное”).

5. Теперь TURBO PROLOG пытается удовлетворить последнюю подцель.

CALL: lives(“зебра”, ”в_воде”)

Обращаясь к каждому предложению, описывающему предикат, TURBO PROLOG не может успешно произвести сопоставление так, что происходит возврат к поиску другого решения.

REDO: lives(“зебра”, ”в_воде”)
REDO: lives(“зебра”, ”в_воде”)
REDO: lives(“зебра”, ”в_воде”)
FAIL: lives(“зебра”, ”в_воде”)

6. Возврат происходит в последнюю установленную точку возврата; в рассматриваемом случае это is_a(“зебра”, ”непарнокопытное”).

7. После возврата TURBO PROLOG освобождает переменные, связанные в этой точке, и пытается найти новое решение для вызова; в данном случае это вызов is_a(Y, ”непарнокопытное”).

8. TURBO PROLOG опускается ниже, пытаясь найти другой подходящий факт, начиная с факта, следующего за точкой возврата. Поскольку подходящих предложений для сопоставления в программе не оказывается, вызов дает неуспех, и TURBO PROLOG возвращается снова, пытаясь все-таки удовлетворить исходную цель.

9. В этом случае последней точкой возврата оказывается type(“непарнокопытное”, ”животное”).

10. TURBO PROLOG освобождает связанные здесь переменные и пытается найти новое решение для вызова

REDO: type(_, ”животное”).

Поиск начинается с предложения следующего за точкой возврата. Сопоставление со следующим фактом происходит успешно; X связывается значением “рыба”, после чего относительно этого факта снова формируется точка возврата.

RETURN: type(“рыба”, ”животное”)

11. TURBO PROLOG переходит к следующей подцели правила; поскольку это новый вызов, поиск начинается с самого первого предложения программы, описывающего этот предикат.

CALL: is_a(_, ”рыба”)
REDO: is_a(_, ”рыба”)

12. Сопоставление происходит успешно со вторым для этого предиката фактом. Y связывается значением “сельдь”, и на этом факте устанавливается точка возврата.

RETURN: *is_a(“сельдь”, ”рыба”)

13. Поскольку значением Y теперь является “сельдь”, следующая вызываемая подцель-

CALL: lives(“сельдь”, ”в_воде”).

Это новый вызов, так что поиск для сопоставления начинается с самого первого предложения программы для этого предиката.

14. TURBO PROLOG проверяет каждый факт, но соответствия не находит, т.е. подцель не достигается.

```
REDO: lives(“сельдь”, ”в_воде”)
REDO: lives(“сельдь”, ”в_воде”)
REDO: lives(“сельдь”, ”в_воде”)
FAIL: lives(“сельдь”, ”в_воде”)
```

15. TURBO PROLOG возвращается в точку is_a(“сельдь”, ”рыба”).

16. Связанные в этой точке переменные освобождаются, и начинается поиск нового решения для вызова is_a(Y, ”рыба”).

REDO: is_a(_, ”рыба”)

17. Сопоставление происходит со следующим фактом, и Y становится связанной значением “акула”.

```
RETURN: is_a(“акула”, ”рыба”)
CALL: lives(“акула”, ”в_воде”)
REDO: lives(“акула”, ”в_воде”)
REDO: lives(“акула”, ”в_воде”)
REDO: lives(“акула”, ”в_воде”)
RETURN: lives(“акула”, ”в_воде”)
RETURN: can_swim(“акула”)
write(“акула”)
write(“-умеет плавать.”)
акула-умеет плавать.
RETURN: goal()
```

Программа успешно завершается. В случае внутренних целей возврата для получения новых решений не происходит.

Управление поиском решений.

Встроенный механизм поиска с возвратом может давать в результате выполнения при поиске ненужные шаги; ввиду этого может падать эффективность. Например, могут быть случаи, когда требуется нахождение единственных решений.

В других случаях может возникать необходимость в поиске дополнительных решений даже после достижения поставленной цели, и тогда процессом поиска с возвратом нужно управлять.

В языке TURBO PROLOG предусмотрены два вспомогательных средства, дающие возможность управлять механизмом поиска с возвратом: предикат fail, который используется для ФОРМИРОВАНИЯ возврата, и отсечение (обозначаемое !), используемое для ПРЕДОТВРАЩЕНИЯ возврата.

3.2.1. ИСПОЛЬЗОВАНИЕ ПРЕДИКАТА fail.

TURBO PROLOG начинает возврат, когда вызов завершается неудачно.

В определенных ситуациях необходимо форсировать возврат с целью поиска альтернативных решений. В языке TURBO PROLOG предусмотрен специальный предикат fail (неудача) для форсирования неудачного исхода и способствования таким образом возврату.

Предикат fail может стоять в телах правил, в целевых предложениях, но где бы он не стоял, при выполнении всегда дает неуспех.

Действие предиката fail аналогично действию в результате сравнения 2=3 или достижения любой невозможной подцели.

Заметим, что после fail в теле правила бесполезно помещать какой-либо предикат связанный конъюнктивно, поскольку предикат fail всегда вызывает неудачный исход стоящего за ним предиката.

Рассмотрим пример программы:

```
domains
 name = symbol
predicates
 father(name,    name)
 everybody
clauses
                    father(“Леонид”, ”Екатерины”).
                    father(“Константин”, ”Якова”).
                    father(“Константин”, ”Марии”).

                    everybody:- father(X,Y),
                                write(X,“-отец “,Y,“.\n”), fail.
```

В данной программе определены отношения: father (отец), everybody (каждого). Программа позволяет определить, кто чей отец. В этой программе цель-father(X,Y) можно поставить двумя способами:

1) как запрос к системе в интерактивном режиме (внешняя цель);

2) через раздел goal программы (внутренняя цель).

Внешняя цель и процесс выполнения программы с ней может выглядеть так:

```
CALL: father(_, _)
Goal: father(X, Y)
RETURN: *father(“Леонид”, ”Екатерины”)
X=Леонид,    Y=Екатерины
REDO: father(_, _)
RETURN: *father(“Константин”, ”Якова”)
X=Константин,    Y=Якова
REDO: father(_, _)
RETURN: father(“Константин”, ”Марии”)
X=Константин,    Y=Марии
3 Solutions
```

Для внешней цели Goal: father(X,Y) TURBO PROLOG выведет все возможные решения в привычном уже виде:

```
X=Леонид,    Y=Екатерины
X=Константин,    Y=Якова
X=Константин,    Y=Марии
3 Solutions
```

Когда же цель father(X,Y) является внутренней:

```
goal
   father(X,Y),
   write(X,“-отец “,Y, “.\n”).
```

исполнение программы даст только одно решение:

```
CALL: goal()
CALL: father(_,  _)
RETURN: *father(“Леонид”,  ”Екатерины”) write(“Леонид”)
Леонид
write(“-отец “)
-отец
write(“Екатерины”)
Екатерины.
write(“.\n”)
(“Леонид-отец Екатерины.”).
```

После полного достижения внутренней цели ничего не указывает системе на необходимость возврата. Поэтому внутреннее обращение с целью father(X,Y) даст только одно решение.

А вот, если в качестве внутренней цели поставить предикат everybody, являющийся головой правила, в котором для форсирования возврата используется предикат fail, то будут найдены все возможные решения. Предикат everybody служит для того, чтобы обеспечивать выдачу информации в удобочитаемом виде.

Данный предикат использует возврат для генерации дополнительных решений для предиката father(X,Y) путем стимулирования системы к возврату через тело правила. Это правило никогда не может быть удовлетворено, что приводит к возврату. При возврате TURBO PROLOG идет к предикату father(X,Y), который может дать несколько решений.

Предикаты, которые могут дать несколько решений, называются НЕДЕТЕРМИНИРОВАННЫМИ.

Предикаты, которые могут дать только одно решение, называются ДЕТЕРМИНИРОВАННЫМИ.

Предикат write не может быть повторно удовлетворен (он не может давать новые решения, он является детерминированным), поэтому TURBO PROLOG должен возвращаться к первому предикату правила. (father(X,Y)- недетерминированный предикат). В описанном случае внутренняя цель будет иметь вид:

```
goal
everybody.
```

Трасса программы:

```
CALL: goal()
CALL: everybody()
CALL: father(_, _)
RETURN: *father("Леонид", "Екатерины") write("Леонид")
Леонид
write("-отец ")
отец
write("Екатерины")
Екатерины
write(".\n")
REDO: father(_, _)
RETURN: *father("Константин", "Якова") write("Константин")
Константин
write("-отец ")
отец write("Якова")
Якова.
write(".\n")
REDO: father(_, _)
RETURN: father("Константин", "Марии") write("Константин")
Константин
write("-отец ")
отец
write("Марии")
Марии
write(".\n").
```

Решения:

```
Леонид-отец Екатерины.
Константин-отец Якова.
Константин-отец Марии.
```

3.2.2. ПРЕДОТВРАЩЕНИЕ ВОЗВРАТА: ОТСЕЧЕНИЕ(!)

В языке TURBO PROLOG предусмотрен стандартный предикат отсечение (cut), используемый для предотвращения возврата;обозначается отсечение восклицательным знаком (!).

Действует отсечение просто: через отсечение выполнить возврат невозможно. В программе отсечение размещают так же, как и другие предикаты - в теле

правила. Когда обработка проходит через отсечение, вызов отсечения сразу же признается успешным, после чего вызывается следующий предикат (если такой есть). После прохождения отсечения в прямом направлении вернуться к предшествующим ему предикатам текущего предложения становиться невозможно, как и невозможным становится возврат к другим предложениям, описывающим данную цель.

Отсечение имеет два основных назначения.

1. Когда вы знаете заранее, что определенные комбинации никогда не дадут приемлемых решений, то поиск альтернативных решений будет простой тратой времени и пространства памяти. Если в такой ситуации воспользоваться отсечением, то получающаяся в результате программа будет работать быстрее и займет меньше памяти. Это так называемое “ЗЕЛЕНОЕ” ОТСЕЧЕНИЕ. (Предотвращение возврата к предыдущему предикату.)

2. Когда этого требует логика программы, отсечение предотвращает рассмотрение альтернативных подцелей. В таких ситуациях говорят о “КРАСНОМ” ОТСЕЧЕНИИ.

   (Предотвращение возврата к следующему предложению.)

Ниже будут приведены примеры, показывающие, как можно пользоваться отсечением в программах. В этих примерах используются схематические правила языка TURBO PROLOG (r1, r2 и r3), которые все описывают один и тот же предикат r плюс несколько подцелей (a, b, c и т.д.).

Предотвращение возврата к предыдущему предикату в теле правила.

R1:-A,B,!,C.

Это способ указания системе на то, что нас удовлетворяет первое найденное решение подцелей A и B. Хотя TURBO PROLOG может найти несколько решений для C с использованием возврата, такой возврат через отсечение для поиска альтернативных решений A или B не разрешается. Не разрешается также возврат к другому предложению, определяющему предикат R1.

В качестве конкретного примера рассмотрим программу.

```
predicates
   buy_car(symbol,symbol)
   car(symbol,symbol,integer)
   colors(symbol,symbol)
clauses
                    buy_car(Model,Color) :-car(Model,Color,Price),
                                           colors(Color,”яркий”),!,
                                           Price < 25000.

                    car(bmw,”зеленый”,25000).
                    car(ford,”черный”,24000).
                    car(ford,“красный”,26000).
                    car(porsche,”красный”,24000).

                    colors(“красный”,”яркий” ).
                    colors(“черный”,”бледный”).
                    colors(“зеленый”, ”умеренный”).
```

В этом примере цель состоит в отыскании автомобиля FORD, имеющего яркий цвет и цену меньшую 25000$.

Отсечение в правиле “buy_car” означает, что, поскольку в базе данных есть только один автомобиль FORD яркого цвета, в том случае, если его цена окажется слишком высокой, необходимости продолжать поиск другого автомобиля уже не будет.

В случае задания цели

```
CALL:  buy_car(“ford”, _)
Goal:  buy_car(ford,Y)
```

1) PROLOG вызывает “car”, первую подцель предиката “buy_car”.

CALL: car(“ford”, _, _)
REDO: car(“ford”, _, _)

2) Проверяется первый автомобиль “bmw”; результат отрицательный.

3) Проверяются следующие автомобили; находится соответствие, связывающее переменную Color со значением “черный”, переменную Price со значением 24000.

RETURN: *car(“ford”,”черный”,24000)

4) После перехода к следующему вызову осуществляется проверка на то, является ли цвет выбранного автомобиля ярким. Черный цвет таковым не является, поэтому проверка заканчивается неудачей.

CALL: colors(“черный”,”яркий”)
REDO: colors(“черный”,”яркий”)
REDO: colors(“черный”,”яркий”)
FAIL: colors(“черный”,”яркий”)

5) PROLOG возвращается к вызову “car” и снова ищет удовлетворяющий поставленным условиям автомобиль FORD.

REDO: car(“ford”, _, _)
RETURN: car(“ford”,”красный”,26000)

6) Подходящий автомобиль находится, после чего снова проверяется цвет.

CALL: colors(“красный”,”яркий”)
RETURN: colors(“красный”,”яркий”)

На этот раз и цвет оказывается подходящим, поэтому PROLOG переходит к следующей подцели правила-отсечению. Отсечение сразу же принимается как успешный исход и “замораживает” ранее выполненные связывания переменных в выражении.

7) PROLOG переходит к следующей (и последней) подцели правила- сравнению: Price < 25000.

26000<25000

8) Проверка в данном случае завершается неудачно и PROLOG пытается вернуться, чтобы найти другой автомобиль для проверки. Но поскольку отсечение предотвращает возврат, другого пути достижения последней подцели не остается, так что и общая цель не достигается.

FAIL: buy_car(“ford”, _)
No Solution

Предотвращение возврата к следующему предложению.

Отсечение может использоваться как способ указания системе на то, что для конкретного предиката найдено правильное предложение.

Рассмотрим, к примеру, следующий фрагмент:

```
r(1):-!, a, b, c.
r(2):-!, d.
r(3):-!, c.
r(_):-write(“Это выражение - ловушка всех ошибок”).
```

Использование отсечения делает предикат r детерминированным. В рассматриваемом случае PROLOG вызывает r с одним целым аргументом. Предположим, что имеет место вызов r(1). PROLOG ищет в программе соответствие с этим вызовом; подходящим является первое определяющее r предложение. Поскольку для вызова есть более одного возможного решения, PROLOG помещает на это предложение точку возврата. Теперь начинает обрабатываться тело правила. Первое, что происходит - это прохождение отсечения; в результате устраняется возможность возврата и перехода к другому предложению r. Тем самым убирается точка возврата, за счет чего повышается эффективность выполнения программы. Обеспечивается также выход на предложение, предназначенное для перехвата ошибок, в случае неудачного вызова r.

Следует отметить, что структура такого типа во многом похожа на структуру других языков программирования, в основе которой лежит оператор типа

CASE. Также следует отметить, что проверяемое условие кодируется в заголовке правил. Ведь можно было бы записать те же выражения несколько иначе:

```
r(X):-X=1,!,a,b,c.
r(X):-X=2,!,d.
r(X):-X=3,!,c.
r(_):-write("Это выражение - ловушка всех ошибок").
```

Однако, проверяемое условие всякий раз, когда это возможно, следует помещать в заголовке правила, поскольку тогда повышается эффективность программы и программа становится более удобочитаемой. В качестве примера рассмотрим следующую программу.

```
predicates
        classify(integer,symbol)
clauses
                        classify(0, zero):-!.
                        classify(X,negative) :- X < 0,!.
                        classify(X,positive).
```

К данной программе поставим внешнюю цель:

```
CALL:  classify(12, _)
Goal:  classify(12,Y).
```

В данном случае происходит следующее:

- Между целью classify(12,Y) и заголовком первого предложения соответствие не устанавливается, поскольку 12 не равно нулю. Так что первое предложение использоваться не может.
- Теперь цель сопоставляется с заголовком второго предложения:
- X связывается значением 12, а Y - свяжется значением после выполнения тела правила. Но проверка X < 0 дает неудачный исход, поскольку X=12, а 12 не меньше нуля. Так что PROLOG возвращается, освобождая связанные переменные.

```
REDO:  classify(12,_)
12<0
FAIL:  classify(12,  _)
```

Наконец, цель сопоставляется с заголовком третьего предложения: X связывается значением 12, а Y - свяжется значением после выполнения тела правила. Проверка X>0 дает положительный исход. Поскольку это положительное решение и последнее предложение, возврат для дальнейшего поиска больше не делается, Y-свяжется значением "positive" и осуществляется переход к введенной цели. И поскольку переменные X и Y относятся к цели, последняя может воспользоваться их значениями в данном случае для автоматического вывода значения.

```
REDO:  classify(12, _)
RETURN:  classify(12, "positive")
Y=positive
1 Solution
```

При постановке внешней цели classify(12,Y) ничего необычного в в процессе выполнения программы не произошло.

Теперь к данной программе поставим внешнюю цель:

```
CALL:  classify(-3, _)
Goal:  classify(-3,Y)
```

В данном случае происходит следующее:

- Между целью classify(-3,Y) и заголовком первого предложения соответствие не устанавливается, поскольку -3 не равно нулю. Так что первое предложение использоваться не может.
- Теперь цель сопоставляется с заголовком второго предложения:
- X связывается значением -3, а Y - свяжется значением после выполнения тела правила. Проверка X < 0 дает удачный исход, поскольку X=-3, а -3 меньше нуля. Система переходит к выполнению следующего предиката предложения (!), выполнение отсечения всегда происходит успешно. Следо-

вательно, все предложение выполняется успешно. Переменная Y свяжется значением “negative”, и осуществляется переход к введенной цели. Но так как в предложении система прошла через отсечение, возврат к поиску новых решений не произойдет, и система прекратит выполнение программы.

```
REDO: classify(-3,_)
3<0
RETURN: classify(-3,”negative”)
Y=negative
1 Solution
```

Отсечение позволило исключить из поиска последнее предложение программы, выполнение которого привело бы к неправильному результату. Аналогичная ситуация произойдет и если в качестве внешней цели будем использовать:

```
CALL: classify(0, _)
Goal: classify(0,Y)
RETURN: classify(0,”zero”)
Y=zero
1 Solution.
```

Рассмотренный пример показывает, что отсечение не оказывает никакого влияния (если только оно не выполняется непосредственно).

То есть для того, чтобы выполнить отсечение, PROLOG должен “войти” в правило, содержащее отсечение, и достичь точки, в которой оно расположено. Отсечению могут предшествовать другие проверки, например:

```
classify(X, negative) :- X < 0, !.
```

В этом правиле отсечение не вступит в силу до тех пор, пока сначала не будет достигнута подцель X<0.

Заметим, что теперь стал важным и порядок правил. Если в данной программе последнее предложение (classify(X, positive).) поставить первым или вторым, мы будем получать неправильные решения.

Отсечения, которые мы использовали в данной программе, некоторые называют КРАСНЫМИ ОТСЕЧЕНИЯМИ (изменяющими логику программы).

* * *

Во многих реализациях языка PROLOG программисты должны уделять недетерминированным предикатам особое внимание из-за накладываемых, как правило, ограничений на ресурсы памяти во время выполнения программы.

TURBO PROLOG же делает внутренние проверки на предмет выявления недетерминированных предикатов, облегчая тем самым работу программиста.

Тем не менее, для отладки(и других целей) всетаки может понадобиться защита; для этого служит директива компилятора check_determ.

Если она указывается в самом начале программы, то TURBO PROLOG в случае обнаружения при компиляции каких-либо недетерминированных выражений будет выдавать предупреждение. Если после выдачи такого предупреждения нажать клавишу F10, предупреждение будет проигнорировано, тогда как нажатие любой другой клавиши приведет к прекращению компиляции.

Вставляя в тела правил, определяющих предикат, отсечения, можно недетерминированные предикаты делать детерминированными.

В предыдущей программе, отсечения в определяющих предикат “classify” предложениях, сделали его детерминированным, поскольку данный предикат обеспечивает поиск одного и только одного решения.

* * *

Применяя отсечение и дизъюнкцию, можно описать взаимоисключающие правила, исполняющиеся как ветвление:

```
если Условие , то Решение1 , иначе Решение2.
```

Пример:

```
max2(X,Y,Max):-X>Y, !,
                Max=X; Max=Y.
```

При помощи этого правила можно находить большее из двух чисел X и Y.

При сопоставлении цели с головой такого правила система начнет выполнять по порядку операции, стоящие в теле правила. Если логический оператор X>Y выполняется успешно (например при X=5, Y=3), система проходит через отсечение, предотвращающее возврат, выполняет оператор сравнения. Если переменная Max несвязана, оператор выполняется успешно, переменной Max присваивается значение переменной X, и система успешно заканчивает работу с этим предложением. В данном случае часть предложения, стоящая после дизъюнкции, системой не выполняется.

Если логический оператор X>Y выполняется неуспешно (например при X=3, Y=5 или при равных значениях X и Y), система вернется к голове правила и попытается пройти его снова, но теперь система будет выполнять действия, описанные в данном предложении после дизъюнкции. Если переменная Max несвязана, оператор сравнения выполнится успешно, переменной Max присвоится значение переменной Y, и система успешно завершит работу с этим предложением.

* * *

Отсечение часто используется в комбинации с предикатом fail.

Комбинация предикатов (!, fail) выполняет роль отрицания. Эта комбинация предикатов позволяет выразить на языке PROLOG, что что-то не есть истина. Выполнение отсечения предотвращает возврат и поиск решения в других предложениях, а предикат fail вызывает неуспех всего предложения и цели, сопоставившейся с головой этого предложения. Поэтому выполнение такой комбинации завершает поиск решения неуспехом. В языке TURBO PROLOG от неуспешного выполнения предложения, содержащего (!, fail), не спасет и использование дизъюнкции, потому что отсечение предотвратит возврат.

Аналогичное действие производит предикат not/1.

Аргументом его могут быть предикаты. Если предикаты, являющиеся аргументами, содержат переменные, то к моменту выполнения not они должны быть связаны, иначе система выдаст сообщение об ошибке Free variables not allowed in NOT (свободные переменные в not не разрешены). Для того, чтобы корректно обрабатывать переменные в пределах предиката not, можно использовать анонимные переменные. Записывается он так: not(predicat). Действие предиката not можно описать следующим образом:

Если predicat выполняется успешно, то not(predicat) выполняется неуспешно, иначе not(predicat) выполняется успешно. На языке PROLOG данный предикат можно было описать следующим правилом:

```
not(predicat):-predicat, !, fail; true.
```

Но в языке TURBO PROLOG этого делать не надо, так как not является стандартным предикатом.

Стандартный предикат true всегда выполняется успешно. В предыдущем примере нахождения большего из двух чисел рассмотренное предложение неправильно даст результат при равенстве чисел. Чтобы исключить из решений равенство чисел, можно использовать такое предложение:

```
max2(X,X, _):-!, fail.
```

В программе данное предложение должно предшествовать ранее рассмотренному нами предложению.

Если в поставленной цели числа будут равны, то она сопоставится с головой первого предложения max2(X,X,_), и система начнет выполнять тело правила. Выполнение комбинации !, fail вызовет неуспех данного предложения, поставленной цели и этим исключит из поиска решений равные числа.

Рассмотрим пример программы, позволяющей находить большее из двух и из трех чисел.

```
predicates
    max(integer)
    max2(integer,integer, integer,integer)
    max3(integer,  integer,  integer,  integer)

clauses
                    max(2):-clearwindow,
```

```
                    cursor(5,2), write("ВВЕДИТЕ ПЕРВОЕ ЧИСЛО:"),
                    readint(First),
                    cursor(5,3),write("ВВЕДИТЕ ВТОРОЕ ЧИСЛО:"),
                    readint(Second),
                    max2(2,First,Second, Max),
                    clearwindow, cursor(5,4),
                    write("БОЛЬШЕЕ ИЗ НИХ РАВНО:",Max),nl;
                    clearwindow, cursor(5,2),write("ЧИСЛА РАВНЫ.").
max(3):-clearwindow,cursor(5,2),
                    write("ВВЕДИТЕ ПЕРВОЕ ЧИСЛО:"),
                    readint(First),
                    cursor(6,2),write("ВВЕДИТЕ ВТОРОЕ ЧИСЛО:"),
                    readint(Second),
                    cursor(7,2),write("ВВЕДИТЕ ТРЕТЬЕ ЧИСЛО:"),
                    readint(Third),
                    max3(First,Second,Third,Max),
                    clearwindow, cursor(5,4),
                    write("БОЛЬШЕЕ ИЗ НИХ РАВНО:",Max), nl;
                    clearwindow, cursor(5,2), write("ЧИСЛА РАВНЫ.").

max(_):-clearwindow, cursor(5,5),
                    write("ВЫ ОШИБЛИСЬ !!!"),!.

max2(2,X,X, _):-!,fail.
max2(3, X, X, X).
max2(_,X,Y,Max):-X>Y,!,
                        Max=X;Max=Y.

max3(X,X,X,_):-!,fail.
max3(First,Second,Third,Max):-max2(3,First,Second,Max_f_s),
                              max2(3,Max_f_s,Third,Max).
goal
                    clearwindow,cursor(5,5), write("ЕСЛИ ХОТИТЕ ИСКАТЬ :"),
                    cursor(6,2), write("БОЛЬШЕЕ ИЗ ДВУХ,  ВВЕДИТЕ 2,  "),
                    cursor(7,2), write("БОЛЬШЕЕ ИЗ ТРЕХ,  ВВЕДИТЕ 3."),
                    cursor(8,4), readint(M), max(M), nl.
```

В данной программе используются стандартные предикаты, неописанные в этом пособии, рассмотрим в начале их.

Предикат clearwindow/0.

Описание: clearwindow очищает текущее текстовое окно от информации путем заполнения его фоновым цветом. Курсор выставляется в позицию (0,0).

Неудачного завершение никогда не дает. Ошибки при выполнении этого предиката отсутствуют.

Предикат cursor/2.

Назначение: Устанавливает или считывает положение курсора в окне.
Обращение: cursor(Стр, Кол)
Домены: (integer, integer)

Шаблоны описания аргументов: (i, i)- когда переменные в предикате связаны, (o,o) - когда переменные в предикате несвязаны.

Описание: (i, i)

Перемещает курсор в указанное место (Стр номер строки, Кол номер позиции в строке) относительно положения (0,0) в активном окне.

(o,o)

Связывает переменные Стр и Кол с текущим положением курсора.

Неудачное завершение: Никогда не дает.
Ошибки: 1001 Недопустимые значения позиции курсора.

В разделе predicates программы описаны нестандартные предикаты:

max(integer) - осуществляет выбор действий, производимых в программе (Если его аргумент будет связан значением: 2-будет производиться поиск большего из двух чисел, 3-будет производиться поиск больше го из трех чисел, любое другое число вызовет сообщение об ошибке ввода),

max2(integer, integer, integer, integer) - осуществляет поиск большего из двух чисел; его аргументы: первый аргумент дает возможность применять этот предикат для нахождения большего из двух чисел и большего из трех чисел (если значением его будет 2, то предикат можно использовать для нахождения большего из двух чисел; если 3, то предикат можно использовать для нахождения большего из трех чисел), второй и третий аргументы-это числа, большее из которых нужно найти, с четвертым аргументом будет связан результат поиска,

max3(integer, integer, integer, integer) - осуществляет поиск большего из трех чисел;его аргументы : первый, второй, третий - это числа, большее из которых нужно найти, с четвертым аргументом будет связан результат поиска.

При поиске большего из двух чисел в этой программе равенство чисел мы можем исключить при помощи правила:

max2(2,X,X, _):-!, fail.

Для этого в предложении max(2) предикат max2(2,First,Second,Max) вызывается с первым аргументом 2. При выполнении данного правила система получит неуспех и вернется в правило max(2). Это правило тоже терпит неуспех в случае выполнения правила max2(2,X,X, _):-!,fail. , а так как в правиле max(2) перед предикатом max2(2,First,Second,Max) стоят только детерминированные предикаты, возврат будет осуществлен к его голове. После этого система попытается выполнить правило max(2) опять, но теперь будут выполнятся предикаты , стоящие после дизъюнкции

(clearwindow , cursor(5,2), write("ЧИСЛА РАВНЫ.")).

Эти предикаты выполнятся успешно(произойдет очистка окна Dialog, и в 4-позиции 5-строки, этого окна будет выведена строка-ЧИСЛА РАВНЫ.). Система успешно завершает выполнение правила max(2), осуществляет возврат к цели max(M), которая в этом случае достигается, и переходит к достижению следующей цели.

При вызове предиката max2(2,First,Second,Max) в правиле max(2), с первым аргументом 2 факт max2(3,X,X,X) исключится из поиска решения, потому что он будет несопоставим с вызовом из за несопоставимости первых аргументов. Этот факт ненужен при поиске большего из двух чисел.

Данный факт нужен для нахождения большего из трех чисел, где допускается равенство двух чисел. По этому в правиле

max3(First,Second,Third,Max):-max2(3,First,Second, Max_f_s),
max2(3,Max_f_s,Third,Max).),

первым аргументом предиката max2 является 3. Этим исключается из поиска решений правило max2(2,X,X, _):-!,fail. Оно ненужно при поиске большего из трех чисел.

Правило (max2(_,X,Y,Max):-X>Y,!,Max=X;Max=Y.), непосредственно находящее большее из двух чисел, нужно при поиске большего из двух и большего из трех чисел. Оно должно работать при любых значениях первого аргумента. Для этого на месте первого аргумента стоит анонимная переменная, она будет сопоставима с любым значением первого аргумента при вызове данного предиката.

Сопоставление при вызове предиката max2 с головой этого правила в нашей программе произойдет, если значения второго и третьего аргументов будут неравны, при этом система найдет большее из двух чисел. Проделав это дважды, используя результат первого вызова в качестве одного из чисел, во втором вызове мы найдем большее из трех чисел.

Подобное использование анонимной переменой в данной программе применено и при описании предиката max.

max(_):-clearwindow, cursor(5,5), write("ВЫ ОШИБЛИСЬ !!!"),!.

Данное предложение защищает от неправильного ввода при выборе действия.

Отсечение - это мощная, но трудно контролируемая операция языка PROLOG. Она похожа на операторы безусловных переходов в других языках программирования. С помощью отсечения можно добиться многого, но оно может сделать программу очень трудной для понимания.

Существуют ограничения в применении отсечения: его появление может нарушить соответствие между декларативным и процедурным смыслами программы. Поэтому хороший стиль программирования предполагает осторожное применение отсечений и отказ от их применения без достаточных оснований.

3.3. РЕКУРСИЯ

Одно из главных достоинств компьютеров заключается в том, что с их помощью можно многократно выполнять одни и те же действия с большой скоростью. В языке PROLOG повторение можно задавать в правилах и в структурах данных. Повторяющаяся структура данных - это звучит довольно необычно, но PROLOG позволяет создавать структуры данных, определенный размер которых во время их создания остается неизвестным. К таким структурам данных можно отнести списки, деревья, базы данных.

В языке программирования PROLOG нет стандартных предикатов, которые бы непосредственно реализовывали повторения. В языке PROLOG предусмотрено только два вида повторения: возвраты, когда обеспечивается поиск нескольких решений по одному запросу, и рекурсия.

Рекурсия в языке PROLOG может использоваться в определении структур данных и в правилах.

Рекурсивные структуры-это структуры, компонентами которых являются сами эти структуры.

Рекурсивные правила-это правила, в теле которых встречаются предикаты, совпадающие с их головой. При организации рекурсии в правилах необходимо учесть:

1) Чтобы для организации повторного выполнения правила предикат, стоящий в теле правила и совпадающий с головой, был сопоставим с головой только этого предложения, а не с другими предложениями, описывающими данный предикат, потому что при вызове предиката поиск для сопоставления начинается с первого предложения, описывающего данный предикат.

2) Что, кроме рекурсивного предложения, описывающего предикат, в программе должны быть и нерекурсивные предложения, описывающие его, с которыми он может сопоставиться при определенных значениях аргументов. Или в теле рекурсивного правила использовать логические операции, которые дадут неуспех при определенных значениях переменных. Такие предложения или логические операции называют граничными условиями. Граничные условия позволяют управлять процессом повторения и прекратить его в нужный момент.

Рассмотрим в качестве примера программу, вычисляющую факториал натурального числа.

```
trace
predicates
  factorial(integer,    real)
clauses
                factorial(1,1) :- !.
                factorial(X,FactX) :-Y = X-1,
                                     factorial(Y,FactY),
                                     FactX = X*FactY.
```

Алгоритм, используемый в этой программе, можно представить в виде таких правил:

1)Если число, факториал которого мы ищем, равно 1, то его факториал равен 1. (Это предложение является граничным условием. Оно определяет, когда

процесс рекурсивных обращений должен закончиться, и дает данные для вычисления факториала числа 2.)

В программе это делает предложение:

factorial(1,1) :- !.

2)Во всех других случаях найти факториал числа на единицу меньшего и умножить на текущее число.

В программе это делает предложение:

factorial(X,FactX) :-Y = X-1, factorial(Y,FactY), FactX = X*FactY.

Чтобы понять как система находит факториал, нужно эту программу исполнить в режиме трассировки. При запуске программы в режиме трассировки с внешней целью :

Goal: factorial(5,Y)

система начнет поиск с первого предложения. Сопоставление с первым предложением произойдет неуспешно, система перейдет ко второму предложению и сопоставится с его головой. При этом первый аргумент - переменная X - приобретет значение, второй аргумент будет несвязанной переменной.

REDO: factorial(5, _)

(При каждом вызове предикатов и вычислении выражений для их аргументов(переменных) выделяется необходимое количество памяти в области памяти, которая называется стеком, и туда будут заноситься значения связанных переменных, а если переменные несвязаны, эти места остаются зарезервированы для будущих значений. Причем каждый раз в новом месте стека, известном системе для каждого вызова. За счет этого система сохраняет значения, промежуточных вычислений и может их в дальнейшем использовать. Но доступ к значениям переменных и результатам вычислений в стеках осуществляется в порядке, обратном их поступлению. (т.е. первыми будут использованы данные, которые в стек занесены последними)).

После сопоставления с головой правила система начнет выполнять его тело. После выполнения оператора сравнения переменная Y приобретет значение 4, и последует рекурсивный вызов предиката factorial с новым значением первого аргумента, вторым аргументом будет несвязанная переменная FactY. (Следует отметить, что при сопоставлении переменных с различными именами они отождествляются, т.е. между областями памяти, в которых должны находится их значения устанавливается связь, при которой связывание одной переменной приводит к связыванию другой тем же значением от одного рекурсивного вызова к другому. В случае нашей программы, такие связи устанавливаются между переменными X и Y, FactX и FactY в теле правила и при сопоставлении цели с головой правила между переменными Y и FactX. Не следует путать переменную Y в цели и переменную Y в теле правила, система их рассматривает как совершенно различные переменные, никак не связанные друг с другом.)

CALL: factorial(4, _)

При повторном вызове предиката factorial из тела правила, для его аргументов выделится новое место в стеке. Процесс поиска для сопоставления опять начнется с первого предложения, описывающего предикат factorial. Сопоставление произойдет успешно опять с головой этого же предложения, и установится в стеке связь между аргументами предшествующего вызова. Система опять начнет выполнять тело этого правила, еще уменьшит на единицу значение первого аргумента, опять последует рекурсивный вызов предиката factorial и т.д. Посмотрим по трассе программы:

REDO: factorial(4, _)
3=3
CALL: factorial(3, _)
REDO: factorial(3, _)
2=2
CALL: factorial(2, _)
REDO: factorial(2, _)
1=1
CALL: factorial(1, _)

Последний рекурсивный вызов из тела правила последует при значении первого аргумента, равном 1. При этом вызове сопоставление произойдет успешно с головой первого предложения , переменная FactY приобретет значение 1, отсечение предотвратит поиск других сопоставлений для вызова предиката factorial из тела второго правила. Так как рекурсивный вызов предиката factorial следовал из тела правила и завершился успешно, система должна вернуться в это предложение и выполнить его оставшуюся часть (FactX = X*FactY) столько раз, сколько было рекурсивных вызовов. В стеке уже будут находится данные для вычислений, причем установлена связь для передачи данных между соседними вызовами. Данные из стека будут браться в порядке, обратном их поступлению и передаваться от Y к X, от FactX к FactY.

Посмотрим по трассе программы:

```
RETURN: factorial(1,  1)
2=2
RETURN: factorial(2,  2)
6=6
RETURN: factorial(3,  6)
24=24
RETURN: factorial(4,  24)
120=120
```

После этого система совершит возврат к цели, успешно с ней сопоставится, так как значение первого аргумента головы предложения станет 5 и цели 5, второй аргумент в цели-несвязанная переменнаясвяжется значением второго аргумента головы предложения, который и будет равен факториалу числа 5.

```
RETURN: factorial(5,  120)
Y=120
1 Solution
```

На этом поиск решений закончится.

3.3.1. ПРЕИМУЩЕСТВА РЕКУРСИИ

У рекурсии есть три основных преимущества:

- с ее помощью могут быть выражены алгоритмы, которые никаким традиционным способом выражены быть не могут;
- она логически проще циклов;
- она широко используется в обработке рекурсивных структур данных.

Рекурсия-это естественный способ описания любой задачи, которая содержит внутри себя другую задачу того же вида. В качестве примеров можно назвать поиск по дереву (дерево состоит из деревьев меньших размеров) и рекурсивную сортировку (для сортировки списка он разбивается на части, которые затем сортируются и соединяются), вычисление факториала.

Логически рекурсивные алгоритмы имеют структуру индуктивного математического доказательства. Представленный выше рекурсивный алгоритм вычисления факториала описывает бесконечное число разных вычислений средствами только лишь двух выражений. Это дает возможность легко проанализировать правильность этих выражений. Более того, правильность каждого выражения может быть установлена независимо от других.

3.3.2. ОПТИМИЗАЦИЯ РЕКУРСИИ ХВОСТА ПРЕДЛОЖЕНИЯ

У рекурсии один большой недостаток: она забирает много памяти. Всякий раз, когда один предикат вызывает другой, рабочее состояние вызывающего предиката должно быть сохранено, чтобы его(вызывающий предикат) можно было (после завершения вызванного предиката) восстановить в том рабочем состоянии, в котором он был. А это значит, что если предикат вызывает себя сто раз, то должны в одно и то же время храниться сто различных его рабочих состояний (сохраняемое рабочее состояние - это состояние стека).

В памяти компьютера IBM PC поместится три-четыре тысячи таких состояний. А что же делать, если какие-либо операции необходимо повторить более четырех тысяч раз?

Оказывается, что есть особый случай, когда предикат может вызывать самого себя без сохранения рабочего состояния. Что, если вызывающий предикат после завершения вызываемого не должен возобновлять свою работу?

Предположим, что вызывающий предикат вызывает предикат на самом своем последнем шаге. Когда вызванный предикат завершается, у вызывающего предиката никакой работы уже не остается. Это и означает, что необходимости сохранять рабочее состояние вызывающего предиката нет, поскольку никакая информация больше не нужна.

Сразу же после завершения вызванного предиката управление может передаваться непосредственно туда, куда бы оно передавалось в случае завершения вызывающего предиката.

Например, предположим, что предикат A вызывает предикат B, а предикат B на последнем своем шаге вызывает предикат C. Когда B вызывает C, у него больше не остается работы. Так что вместо сохранения текущего рабочего состояния B (которое больше не нужно) мы можем сохранить текущее состояние C путем внесения в сохраняемую информацию соответствующих изменений. И когда C завершается, получается так, как будто он вызывался непосредственно предикатом A.

А теперь предположим, что B вызывает не C, а самого себя на своем последнем шаге. Наш рецепт говорит, что, когда B вызывает B, область стека вызывающего предиката B должна быть заменена областью стека вызванного предиката B. Это совершенно простая операция;необходимо только обновить значения аргументов, после чего выполнить переход назад к началу предложения. Итак, с процедурной точки зрения то, что происходит, очень похоже на обновление управляющих переменных цикла. Описанные здесь действия носят название оптимизации рекурсии хвоста выражения (tail-recursion optimization) или оптимизации последнего вызова (last-call optimization).

3.3.2.1. РЕАЛИЗАЦИЯ ХВОСТОВОЙ РЕКУРСИИ

Что означает, когда говорят, что один предикат вызывает другой на самом своем последнем шаге?

В языке PROLOG это означает, что:

1. Данный вызов - это самый последний предикат предложения.

2. Перед этим вызовом в предложении нет точек возврата.

Рассмотрим пример программы, удовлетворяющий этим требованиям.

```
predicates
  count(real)
clauses
        count(N) :-write(N), nl, NewN = N+1, count(NewN).
goal
        count(1).
```

Данная программа при своей работе будет генерировать числа и выводить их на экран, каждый раз с начала новой строки. В описании предиката count имеет место хвостовая рекурсия. Предикат count вызывает сам себя, не создавая новой области в стеке для сохранения текущего состояния. Стека для такой программы всегда будет хватать. При запуске программы мы попадем в бесконечный цикл, потому что нет граничного условия рекурсии.

3.3.2.2. КАК ИЗБАВИТЬСЯ ОТ ХВОСТОВОЙ РЕКУРСИИ

Рассмотрим примеры программ, в которых используются различные способы избавления от хвостовой рекурсии.

1. Если рекурсивный вызов не является самым последним шагом, то хвостовой рекурсии в процедуре нет. Например:

```
predicates
    badcount1(real)
clauses
        badcount1(X) :-write(X), nl,NewX = X+1,
                        badcount1(NewX),  nl.
```

Каждый раз, когда предикат badcount1 вызывает сам себя, рабочее состояние стека приходится сохранять, чтобы можно было вернуть управление вызывающему предикату, который не выполнил последнего предиката nl, осуществляющего перевод курсора в начало новой строки текущего окна. Так что до исчерпания памяти такая программа сможет выполнить только несколько тысяч рекурсивных вызовов.

2. Другой способ избавиться от хвостовой рекурсии - сделать так, чтобы во время рекурсивного вызова предпринимались попытки нахождения альтернативных решений. В таком случае рабочее состояние стека должно быть сохранено, чтобы, если рекурсивный вызов дает неудачный исход, вызывающий предикат мог вернуться и попытаться найти альтернативное решение. Например:

```
predicates
    badcount2(real)
clauses
                    badcount2(X) :- write(X),nl, NewX = X+1,
                                    badcount2(NewX).
                    badcount2(X) :- X < 0,
                                    write("X-отрицательное значение.").
```

Здесь первое предложение badcount2 вызывает само себя до того, как будет вызвано второе выражение. И здесь имеет место исчерпание памяти после некоторого числа вызовов.

3. Но рассматриваемым альтернативным вариантом вовсе не должно быть предложение, соответствующее самому рекурсивному предикату. Это вполне может быть альтернативный вариант другого вызываемого предиката. Например:

```
predicates
    badcount3(real)
    check(real)
clauses
                    badcount3(X) :- write(X), nl,
                                    NewX = X+1,
                                    check(NewX),
                                    badcount3(NewX).

                    check(Z) :- Z >= 0.
                    check(Z) :- Z < 0.
```

Предположим, что X - положительное, как это в действительности и есть. Тогда при вызове badcount3 самого себя исход первого check успешен, но попытка удовлетворить второе предложение, описывающее предикат check еще не предпринята. Поэтому предикат badcount3 вынужден сохранять копию своего рабочего состояния в стеке, чтобы можно было вернуться и попытаться удовлетворить второе предложение, описывающее предикат check в случае неудачного исхода рекурсивного вызова.

3.3.2.3. ИСПОЛЬЗОВАНИЕ ОТСЕЧЕНИЯ ДЛЯ РЕАЛИЗАЦИЯ ХВОСТОВОЙ РЕКУРСИИ

Из приведенных примеров можно сделать вывод: что сделать рекурсивный вызов последним в предложении достаточно легко, но как гарантировать, что не будет альтернатив в каких-либо других вызываемых этим предложением предикатах?

Отменить любые возможные альтернативы может встроенный предикат отсечение(!). Правильность же расстановки отсечения можно проверить, воспользовавшись директивой компилятора check_determ.

Реализовать хвостовую рекурсию в предикате badcount3 можно следующим образом :

```
predicates
    cutcount3(real)
```

```
    check(real)
clauses
                    cutcount3(X) :-write(X),nl,NewX = X+1,
                                  check(NewX), !,cutcount3(NewX).
                    check(Z) :- Z >= 0.
                    check(Z) :- Z < 0.
```

Отсечение после его выполнения указывает системе не обращать внимание на другие предложения, описывающие этот предикат (cutcount3), и не искать альтернативные решения для предшествующих предикатов, в данном предложении для предиката check. А это как раз то, что нужно. Поскольку альтернативы выводятся из рассмотрения, необходимость в повторном копировании текущего рабочего состояния стека отпадает, и мы получаем хвостовую рекурсию.

Аналогичным образом отсечение может помочь и в предикате badcount2.

```
predicates
    cutcount2(real)
clauses
                    cutcount2(X) :-X>=0, !, write(X), nl,
                                  NewX = X + 1,
                                  cutcount2(NewX).
                    cutcount2(_) :-write("X - отрицательное.").
```

Если выполняется отсечение, компьютер считает, что непроверенных альтернативных предложений для предиката cutcount2 нет и копий рабочих состояний стека не создает.

При исполнении хвостовая рекурсия подобна операторам циклов в процедурных языках программирования. Поэтому при использовании в программах хвостовой рекурсии мы должны определить условие цикла, параметр цикла и задать ему начальное значение.

Рассмотрим пример программы - вычисления факториала с использованием хвостовой рекурсии.

```
predicates
    factorial(integer,real)
    factorial(integer, real, integer, real)
clauses
                    factorial(N,FactN):-factorial(N,FactN,1,1).
                    factorial(N,FactN,N,FactN):- !.
                    factorial(N,FactN,I,P):-NewI=I+1,NewP = P*NewI,
                                          factorial(N,FactN,NewI,NewP).
```

В программе определены два одноименных предиката factorial/2 и factorial/4, система их будет воспринимать как различные предикаты, потому что у них различное количество аргументов. Предикат factorial/2 используется для: ввода числа факториал, которого надо найти, для присваивания начального значения параметру цикла и переменной, при помощи которой вычисляется произведение, вывода значения факториала в поставленную цель. Данный предикат описывается первым предложением программы, в теле которого вызывается предикат factorial/4. Предикат factorial/4 непосредственно обеспечивает вычисление факториала. Первый аргумент-это число, факториал которого нужно найти, второй аргумент-результат вычисления факториала, третий-это числа от 1 до N, которые нужно умножать и одновременно с этим является параметром цикла, четвертый аргумент-принимает последовательно значения факториалов чисел от 1 до N. Первое правило, описывающее предикат factorial/4-это условие цикла, оно завершает цикл, когда значение параметра цикла станет равным числу, факториал которого вычисляется, а также передает результат вычислений аргументу, предназначенному для результата.

Второе правило, описывающее предикат factorial/4 -это цикл, реализованный посредством хвостовой рекурсии, в нем последовательно вычисляются факториалы чисел по правилу: N!=(N-1)!*N, а так же в конце вычислений передается результат в первое предложение программы.

***

При рассмотрении механизмов логического вывода в данной теме, мы раскрывали процедурный смысл программ. Он определяет, как система выполняет программу.

Но программы на языке PROLOG имеют еще и декларативный смысл. Декларативный смысл программ определяет, какие факты и правила мы должны использовать, чтобы написать программу, и что мы можем получить в результате.

В следствии декларативности языка PROLOG описанные предикаты в зависимости от параметров могут производить прямые и обратные действия. В программе вычисления факториала прямым действием будем считать нахождения факториала заданного числа, а обратным-нахождение числа по значению его факториала.

Если описанную нами программу нахождения факториала запустить и поставить перед системой цель: factorial(5,F), система после выполнения программы, найдет значение факториала и свяжет им переменную F(120). Если эту программу запустить и поставить перед системой цель: factorial(F,120), система после выполнения программы, найдет число по значению его факториала и свяжет им переменную F(5).

Для написания хороших программ на языке PROLOG необходимо учитывать их декларативный и процедурный смыслы.

ЗАДАНИЯ К ТЕМЕ :

1) Вычислить значение функции:

$$sign(x)=\begin{cases}0, & \text{если } x=0,\\ 1, & \text{если } x>0,\\ -1, & \text{если } x<0\end{cases}$$

2) Вычислить значение функции:

$$y=\begin{cases}2, & \text{если } x<-1,\\ 1-x, & \text{если } -1 \le x \le 1,\\ 0, & \text{если } x>1.\end{cases}$$

3) Вычислить значение функции:

$$y(A,B)=\begin{cases}A*B, & \text{если A и B} < 0,\\ \text{большему из A и B}, & \text{если A и B} >0,\\ A-B, & \text{если A или B} < 0.\end{cases}$$

4) Возвести число в степень с целым показателем.

5) Вычислить число Фиббоначчи по его введенному номеру.

6) Вычислить наибольший общий делитель двух натуральных чисел, введенных с клавиатуры, по алгоритму Евклида.

7) Составить программу, которая бы позволяла выбирать и табулировать, все стандартные математические функции языка TURBO PROLOG в введенном интервале с любым шагом, с учетом области определения.

8) Составить программу, которая бы определяла все родственные отношения в вашей династии. Вопросы системе должны вводиться интерактивно.

4. СПИСКИ

Обработка списков-манипулирование объектами,которые содержат произвольное число элементов-это мощный метод использования языка PROLOG.

4.1. ПРЕДСТАВЛЕНИЕ СПИСКОВ В ЯЗЫКЕ PROLOG

Списки-это организованная совокупность данных любого типа, состоящая из произвольного числа элементов одного и того же типа.

В языке TURBO PROLOG списки-это дискретные, динамические структуры данных.Они состоят из элементов одного и того же типа, но тип может быть любым допустимым в языке TURBO PROLOG. Элементами списков могут быть другие списки. Степень вложенности списков неограничена. Размеры списков предварительно не определяются и могут изменяться в процессе работы с ними.

Но в отличии от других структур данных, списки имеют свои способы описания, внешнего и внутреннего представления.

Внешне списки можно представить как последовательность элементов, отделяющихся друг от друга запятой,заключенную в прямоугольные скобки.

Примеры:

```
                    [1 , 2 , 3]   - список типа integer,
[1.256,-0.789,1.5678e+12]   - список типа real,
    ["TURBO"," PROLOG"]   - список типа string,
              [turbo,prolog]   - список типа symbol,
          ['t','u','r','b','o']   - список типа char,
           [[1,2],[3,[4,5]]]   - составной список,
                          []   - пустой список.
```

Пустой список - это список , в котором нет элементов. Он служит признаком конца любого списка.

Так как списки - это структуры данных, их надо предварительно описывать в разделе domains.

Примеры:

Чтобы объявить домен для списка целых чисел,используется объявление,подобное следующему:

```
domains
list_int = integer*
```

Звездочка обозначает,что описываемая структура - список.

Элементами списка могут быть любые объекты данных,включая другие списки.Однако,все элементы списка должны принадлежать одному и тому же домену,а объявление domains должно иметь вид:

```
domains
                list_object = object*
                object = ...
```

где элементы приравниваются к одному типу данных (например, integer, real, string, char или symbol) или набору альтернативных типов,помеченных различными функторами. TURBO PROLOG не допускает смешивания в списке стандартных типов. Например,следующее объявление не будет правильно отражать список,составленный из элементов типа integer, real, symbol:

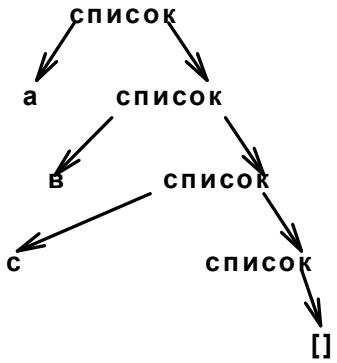
```
domains
list_object = object*
object = integer;real;symbol /*неверно*/
```

Чтобы объявить подобный список,следует определить один домен, включающий все три типа, при помощи функторов. Например:

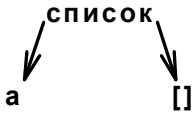
```
domains
                list_object = object*
                object = i(integer);r(real);c(symbol)
```

Список в языке PROLOG можно рассматривать как специальный частный случай двоичного дерева, как и другие составные объекты.

Древовидная структура списка [a,в,c] может быть представлена так:



Более того, одноэлементный список [a] - это не то же самое, что содержащийся в нем элемент, поскольку в действительности он соответствует структуре:



Список во внутреннем представлении (т.е. при обработке системой) является рекурсивным составным объектом. Он либо пуст,либо состоит из двух частей: головы,представляющей первый элемент, и хвоста, представляющего собой список, состоящий из всех последующих элементов. Хвост списка-это список. Голова списка-это всегда элемент.
Например:

голова [a,b,c] - это a,
хвост [a,b,c] - это [b,c].

А что же в случае одноэлементного списка? В этом случае:

голова [a] - это a,
хвост [a] - это [].

Если взять из списка первый элемент несколько раз, то в конце концов будет получен пустой список. Пустой список на голову и хвост разбить невозможно.

Сказанное означает, что разбивая список на голову и хвост мы имеем последовательный доступ к его элементам, пока не дойдем до пустого списка.

В языке PROLOG предусмотрена возможность явного задания головы и хвоста списка. Вместо разделения элементов запятыми, голову и хвост можно разделить с помощью символа “|”.

Например:

[a,b,c] эквивалентно [a|[b,c]] и, продолжая этот процесс, [a|[b|[c|[]]]].

Можно даже в одном списке использовать оба вида разделителей, но при этом вертикальная черта должна быть последним разделителем. Значит, если это действительно нужно, можно записать [a,b,c,d] как [a,b|[c,d]].

Головы и хвосты списков

список	голова	хвост
[a,b,c]	a	[b,c]
[a]	a	[]
[]	не определена	не определен
[[1,2,3],[2,3,4],[]	[1,2,3]	[[2,3,4],[]]

В списки можно записывать связанные и несвязанные переменные, при этом если переменные будут связаны,то список будет заполняться их значениями.

Списки могу сопоставляться с другими списками по правилам сопоставления структур.

Рассмотрим на примерах результаты сопоставления списков.

Список1	Список2	Связывание переменных
[X,Y,Z]	[анна,ест,торт]	X=анна Y=ест Z=торт
[7]	[X|Y]	X=7 Y=[]
[1,2,3,4]	[X,Y|Z]	X=1 Y=2 Z=[3,4]
[1,2]	[3|X]	неудачный исход

4.2. ТИПИЧНЫЕ ЗАДАЧИ ОБРАБОТКИ СПИСКОВ

Ввиду того, что списки являются рекурсивными составными структурами данных, для их обработки нужны рекурсивные алгоритмы.

Наиболее общим способом обработки списка является прохождение по списку до конца с выполнением над каждым элементом каких-либо действий.

Для такого алгоритма необходимы два предложения. Одно, говорящее о том,что делать с обычным списком (тем,который может быть разбит на голову и хвост),это обычно-рекурсивное правило,и второе,указывающее действие для пустого списка.

При решении задач по обработке списков необходимо учитывать декларативность языка PROLOG, потому что решение одних задач при изменении направления потока информации и несущественных переделках,может являться решением других задач.

Так же при решении задач необходимо помнить, что исходный список изменить нельзя,из него можно брать элементы ,преобразовывать и записывать в новый список,в результате получится новый преобразованный список.

Задачи ввода элементов в список и вывода списков на экран.

Рассмотрим пример программы:

```
domains
   splist=integer*
predicates
   prsp(integer,integer,splist,symbol)
   readkey(string)
goal
                    clearwindow,
                    prsp(M,0,Splist,begin),clearwindow,cursor(5,5),
                    write("Элементы списка  ",Splist),cursor(7,5),
                    write("Количество элементов в списке: ",M).
clauses
                    prsp(Y,Y,[],end):-!.
                    prsp(N,Y,[H|T],Str):-cursor(15,0),
                                         write("\t\t\t\t\t\t\t\t\t\t"),
                                         Z=Y+1,cursor(12,15),
                                write("Введите элемент списка ",Z,"  "),
                                       readint(H),!,readkey(Str1),
                                prsp(N,Z,T,Str1) ; prsp(N,Y,[H|T],begin).

                    readkey(Str):- cursor(12,0),
                                   write("\t\t\t\t\t\t\t\t\t\t"),cursor(15,10),
                                   write("Для завершения нажмите клавишу
                                   End,","для продолжения-пробел."),
                                   readchar(Chr),char_int(Chr,0),
                                   Str=end,!;Str=begin,!.
```

Данная программа реализует: ввод списка с клавиатуры,подсчет количества элементов и вывод списка на экран. В разделе domains этой программы описываются целочисленные списки. В программе определены и описаны два предиката prsp/4 и readkey/1.

Предикат prsp/4 осуществляет ввод чисел с клавиатуры в список и подсчет количества элементов.

prsp(N,Y,[H|T],Str) (o,i,o,i)

- Первый аргумент-выходной параметр,с ним в конце ввода будет связано количество элементов в списке.
- Второй аргумент-входной параметр,с ним в процессе ввода будет связан номер текущего элемента списка.
- Третий аргумент-выходной параметр,с ним в конце ввода будет связан вводимый список.
- Четвертый аргумент-входной параметр,является признаком продолжения или конца ввода чисел в список.Если он связан значением begin,то будет произ-

водиться ввод,а если значением end,то ввод элементов в список заканчивается.

- Предикат readkey/1 производит анализ нажатой клавиши.

readkey(Str) (o)

Он имеет один выходной параметр,с которым в зависимости от нажатой клавиши будет связываться признак продолжения или конца ввода (begin или end).

В программе используется ранее не описанный стандартный предикат char_int/2.

Предикат char_int/2.

Назначение: Преобразует символ в значение ASCII кода и обратно.
Обращение: char_int(СимвАрг,ЦелАрг).
Домены: (char,integer).
Шаблоны описания аргументов:(i,o),(o,i),(i,i)
Описание: (i,o)

Связывает ЦелАрг кодом ASCII символа-СимвАрг.

(o,i)

Связывает символьную переменную СимвАрг символом, ASCII код которого является значением-ЦелАрг.

(i,i)

Выполняется успешно,если ЦелАрг соответствует ASCII коду СимвАрг.В противном случае терпит неуспех.

Ошибки: Отсутствуют.

Пример:

```
Goal: char_int('a',X)
X=97
1 Solution
Goal: char_int(X,97)
X=a
1 Solution
Goal: char_int('a',97)
Yes
Goal: char_int('a',197)
No
```

Ввод целых чисел в список реализуется двумя предложениями, описывающими предикат prsp/4.

Первое предложение-граничное условие рекурсии.Во втором предложении реализована хвостовая рекурсия. При запуске программы начнется выполнение предикатов целевого предложения.После выполнения стандартного предиката clearwindow, производящего очистку текущего окна, в качестве цели будет вызван предикат prsp(M,0,Splist,begin) и начнется поиск решения для достижения этой цели. Сопоставление начнется с первого предложения, описывающего этот предикат.При начальных значениях аргументов попытка сопоставления с голой первого предложения окончится неуспехом (не сопоставятся четвертые аргументы). Система будет сопоставлять цель с головой второго предложения. Сопоставление произойдет успешно (так, как все переменные головы предложения будут несвязанными). В результате сопоставления переменные Y и Str приобретут начальные значения, а третий аргумент-список будет разделен на голову и хвост [H|T], переменная H-голова, переменная T-хвост, но они не будут еще связаны значениями. Система начнет по порядку выполнять предикаты тела этого правила. При помощи предикатов cursor(15,0), write("\t\t\t\t\t\t\t\t\t\t") производится очистка 15 строки текущего окна. Совокупность символов \t в предикате write выводит на экран пробелы в количестве, равном количеству позиций табуляции. Операция сравнения позволит получить номер вводимого в список элемента. При помощи стандартного предиката readint(H) переменной H, являющейся головой списка присваивается целочисленное значение, вводимое с клавиатуры, т.е. осуществляется ввод элемента в список. Выполнение отсечения в правиле предотвращает возвраты, чем способствует реализации

хвостовой рекурсии. При выполнении предиката readkey(Str1) система переходит в правило, описывающее этот предикат,и начинает выполнять его тело. При выполнении стандартного предиката readchar(Chr) система приостановит выполнение программы и будет ждать нажатия клавиши, при нажатии клавиши система свяжет символ, соответствующий нажатой клавише, с переменной Chr. После этого система начнет выполнять стандартный предикат char_int(Chr,0) с двумя связанными аргументами. Если будет нажата клавиша, дающая код ASCII, предикат char_int(Chr,0) даст неудачный исход, система вернется к голове этого предложения и выполнит то, что стоит после дизъюнкции, т.е. переменной Str будет присвоенно значение begin. Наличие отсечения в данном предложении делает предикат readkey(Str1) детерминированным, этим способствует организации хвостовой рекурсии в предложении, из которого этот предикат вызывается. Если будет нажата клавиша, дающая расширенный код (любая управляющая клавиша, например End), предикат char_int(Chr,0) даст удачный исход, переменной Str будет присвоенно значение end.

После выполнения предложения,описывающего предикат readkey(Str1), система возвращается в предложение,из которого он был вызван и в результате сопоставления переменная Str1 приобретает значение begin или end.После этого следует рекурсивный вызов предиката prsp(N,Z,T,Str1) с новыми значениями аргументов.В качестве списка берется хвост списка,переменная T не связана значением.

Сопоставление опять начнется с первого предложения, описывающего этот предикат. Если переменная Str1 будет иметь значение end, то сопоставление произойдет успешно, переменная T приобретет значение [], а вводимый список будет [H|[]], что эквивалентно списку [H]. Значение приобретет первый аргумент предиката, оно будет соответствовать количеству элементов в списке и передано через переменную Y. Далее произойдет сопоставление с предикатом prsp(M,0,Splist,begin) - целевого предложения, несвязанным переменным M и Splist будут присвоены значения (количества элементов в списке и сам список). После выполнения оставшихся предикатов целевого предложения, обеспечивающих вывод списка на экран, система успешно завершит выполнение программы.

Если переменная Str1 будет иметь значение begin, то сопоставление рекурсивного вызова произойдет успешно опять с головой второго предложения prsp(N,Y,[H|T],Str), и хвост списка будет разбит на новую голову и новый хвост, и все описанные действия повторятся. В данной программе элементы будут вводится в голову списка после очередного разбиения его хвоста на новую голову и новый хвост.

Дизъюнкция во втором предложении , описывающем предикат prsp/4 (; prsp(N,Y,[H|T],begin).), защищает программу от неправильного ввода элементов списка. Эта часть предложения будет выполняться, если предикат readint(H) даст неудачный исход, при этом будет осуществляться рекурсивный вызов с старыми значениями аргументов.

Для формирования списков в языке TURBO PROLOG есть стандартный предикат findall/3.

Предикат findall/3.

Назначение: Собирает значения указанных переменных в указанных предикатах, получаемые в процессе поиска с возвратом в список.

Обращение: findall(Перем,<ВызывПред>,Список)

Описание: findall формирует при помощи поиска с возвратом в <ВызывПред> список-Список из значений указанной переменной-Перем.
Формирование идет до получения окончательного неудачного завершения. Переменная-Список содержит все значения переменной-Перем,которые являются выходными аргументами предиката <ВызывПред>.
Для работы findall необходимо объявить домен спискового типа для переменной-Список в разделе domains.

Неудачное завершение: Никогда не дает.

Ошибки: 1020 Свободная переменная недопустима здесь.

(Если свободная переменная возвращается в “Перем”.)

Пример:

```
domains
  name, address = string
  age = integer
  list = age*
predicates
  person(name, address, age)
  sumlist(list, age, integer)
goal
                    findall(Age, person(_, _, Age), L),
                    sumlist(L, Sum, N),
                    Ave = Sum/N,
                    write(“Average =”, Ave), nl.

clauses
                    sumlist([], 0, 0).
                    sumlist([H|T], Sum, N) :- sumlist(T, S1, N1),
                                              Sum=H+S1,
                                              N=1+N1.
                    person(“Sherlock Holmes”, “22B Baker Street”, 42).
                    person(“Pete Spiers”, “Apt. 22, 21st Street”, 36).
                    person(“Mary Darrow”, “Suite 2, Omega Home”, 51).
```

В данной программе формируется целочисленный список-L при помощи предиката findall,находится сумма элементов списка и их количество при помощи предиката sumlist,находится их среднее арифметическое и выводится на экран.

При запуске программы начнется выполнение предикатов целевого предложения.Первым в качестве цели будет вызван стандартный предикат findall(Age, person(_, _, Age), L).Его выполнение начнется с сопоставления предиката person(_, _, Age) с фактами,описывающими его.Так как аргументами этого предиката являются несвязанные переменные,он будет сопоставим с каждым фактом.В следствии механизма возврата этот предикат по очереди сопоставится с каждым фактом, переменная Age при каждом новом сопоставлении будет приобретать новое целочисленное значение,которое каждый раз будет заносится в список L предикатом findall.Значение первого и второго аргументов предиката person(_, _, Age) в программе ненужны,поэтому на их местах стоят анонимные переменные.

Затем в качестве цели будет вызван предикат sumlist(L, Sum, N), у которого в качестве первого аргумента будет введенный список, второй и третий аргументы будут несвязанными переменными. Данный предикат описывается двумя предложениями. Первое предложение будет граничным условием рекурсии. Второе-рекурсивным правилом, в котором реализована обычная (нехвостовая) рекурсия.

При вызове предиката sumlist(L, Sum, N) в качестве цели сопоставление с первым предложением даст неуспех, список L-непустой.

Сопоставление с головой второго предложения произойдет успешно, так как в голове предложения будут стоять несвязанные переменные. При этом исходный список будет разбит на голову и хвост. Переменная H-голова, будет связана первым элементом списка, переменная T-хвост, оставшейся частью списка. Переменные Sum и N останутся несвязанными. После этого система начнет выполнять тело правила. Последует рекурсивный вызов предиката sumlist(T, S1, N1) с новыми значениями аргументов. Теперь в качестве первого аргумента берется хвост списка, переменные S1 и N1 останутся несвязанными. Система начнет сопоставление с первым предложением, если T не будет пустым списком, попытка сопоставления с первым предложением терпит неуспех. Система этот вызов успешно сопоставит с головой второго предложения, так как непустой список можно разбить на голову и хвост. При этом хвост исходного списка будет разбит на новую голову и новый хвост. Этими значениями будут связаны переменные H-новая го-

лова, T-новый хвост. Процесс разбиения хвоста списка будет рекурсивно повторяться до тех пор, пока хвост-непустой список. Когда в результате последовательных разбиений списка, его хвост станет пустым списком , сопоставление предиката sumlist(T, S1, N1) произойдет успешно с первым предложением - sumlist([], 0, 0). При этом переменные S1, N1 приобретут начальные значения. Так как в программе реализована нехвостовая рекурсия, система при каждом вызове предиката sumlist отводила новое место в стеке для его аргументов (запоминала рабочее состояние этого предиката), а при сопоставлениях устанавливалась связь между этими областями памяти для каждого аргумента. За счет этого будут последовательно передаваться значения аргументов предиката sumlist от области стека, соответствующей последнему вызову, к области стека, соответствующей первому вызову предиката sumlist.

После успешного сопоставления предиката sumlist с первым предложением система вернется во второе предложение и выполнит его оставшуюся часть (Sum=H+S1,N=1+N1.) столько раз, сколько было рекурсивных вызовов предиката sumlist. При этом нахождение суммы элементов списка начнется с последнего элемента, и после каждого сложения значение переменной Sum будет передаваться S1, а значение переменной N переменной N1.В результате таких действий система вычислит сумму элементов и их количество в списке L и, при возврате в целевое предложение, свяжет этими значениями переменные Sum и N целевого предложения. Далее система найдет среднее арифметическое элементов списка L (Ave = Sum/N,),выведет его на экран и успешно закончит выполнение программы.

Задачи модификации списков.

В этих задачах исходный список последовательно разбивается на голову и хвост, при каждом разбиении голову преобразуют и после преобразования делают головой нового списка.

В результате работы таких программ создаются новые модифицированные списки.

Пример:

```
domains
   list = integer*
predicates
   add1(list, list)
clauses
                    add1([], []).
                    add1([H|T], [H1|T1]):-H1=H+1,
                                          add1(T,T1).
```

В данной программе определяется и описывается предикат add1/2.

Первый аргумент-входной параметр-исходный целочисленный список. Второй аргумент-выходной параметр-модифицированный список.

Второй список получается вследствии добавления 1 к каждому элементу первого списка.

Предикат add1/2 описывается двумя предложениями.Первое предложение-факт,является граничным условием рекурсии.Второе предложение-рекурсивное правило,в котором реализована хвостовая рекурсия.

На естественном языке данную программу можно описать такими правилами:

Чтобы добавить единицу ко всем элементам пустого списка, нужно сгенерировать другой пустой список.

Чтобы добавить 1 ко всем элементам любого другого списка, нужно добавить 1 к голове и сделать ее головой результата, а затем добавить 1 к хвосту и сделать его хвостом результата.

При запуске данной программы с внешней целью add1([1,2,3,4],N) мы получим в результате:

```
N=[2,3,4,5]
1 Solution.
```

При помощи такого предиката:

```
two([],[]).
```

```
two([H|T],[H,H,|T1]):-two(T,T1).
```

Можно получить новый список, в котором каждый элемент будет фигурировать дважды.

Задачи поиска заданного элемента в списке.

К задачам поиска можно отнести:

- поиск первого или последнего вхождения заданного элемента в список;
- поиск количества вхождений заданного элемента в список.

Решение данных задач на естественном языке можно описать правилом:

Искомый элемент принадлежит списку, если он является его головой, либо принадлежит его хвосту.

Поиск первого вхождения элемента в список можно реализовать на языке TURBO PROLOG такими предложениями:

```
poisk([],X,0):-cursor(14,15),
                write("Элемента ",X," в списке нет"),!,fail.
poisk([X|T],X,1).
poisk([H|T],X,C):-poisk(T,X,Col),C=Col+1.
```

В данной программе определяется порядковый номер первого вхождения заданного элемента в список.

Первые два предложения являются граничными условиями рекурсии. Сопоставление с первым предложением произойдет успешно, если искомого элемента в списке нет. В этом случае рекурсивно разбивая исходный список на голову и хвост, мы обязательно прийдем к пустому списку и получим неудачный исход для целевого предложения. Сопоставление со вторым предложением произойдет успешно, если после разбиения, искомый элемент является головой списка. Третье предложение является рекурсивным правилом.Оно обеспечивает поиск заданного элемента в хвосте списка и вычисление его порядкового номера в списке.

Задачи удаления элементов из списка.

К задачам удаления элементов из списка можно отнести:

- удаление первого или последнего элемента,совпадающего с заданным;
- удаление всех элементов,совпадающих с заданным.

Решение данных задач на естественном языке можно описать правилами:

Если заданный элемент является головой списка,тогда результатом удаления будет хвост этого списка.

Если заданный элемент находится в хвосте списка,тогда его нужно удалить оттуда.

Задачи конкатенации,разбиения,вставки списков.

Задачи конкатенации связаны с образованием нового списка, в котором сначала идут элементы первого списка,а за ними элементы второго списка или наоборот. В предикате, реализующем конкатенацию списков, на вход подается два списка, на выходе мы получаем третий, новый список.

На естественном языке такой предикат можно описать в виде правил:

Если первый список-[],то второй список является результирующим. Это правило будет являться граничным условием рекурсии.

Если первый список не пуст, то последовательно разбивать его на голову и хвост, и каждый раз голову первого списка делать головой результирующего списка. Это правило должно быть рекурсивным.

Первое правило позволит прекратить повторение в программе и второй список сделает хвостом результирующего списка.

В следствии декларативности языка PROLOG предикат, реализующий конкатенацию, можно использовать для разбиения списка на два новых списка. Для этого исходный список в этом предикате поставить на место выходного параметра, а на место входных параметров-несвязанные переменные. Если мы этот предикат с такими значениями аргументов вызовем в качестве

внешней цели, то получим множество возможных разбиений исходного списка на два.

Для получения единственного решения задачи разбиения можно модифицировать предикат, реализующий конкатенацию. Для этого в нем добавить еще один аргумент, указывающий номер элемента, за которым произойдет разбиение. В граничном условии на месте этого аргумента поставить 0,а в рекурсивном правиле уменьшать его значение на 1.

К задачам вставки списков относятся:

- вставка элемента в любое место списка;
- вставка одного списка в любое место другого списка.

Решение этих задач легко можно получить,используя разбиение и конкатенацию списков.

Например,вставить один список S1 в любое место другого S2 можно таким способом:

разбить список S2,в который должна производиться вставка на два (L1-состоит из элементов списка S2, идущих от начала до места вставки, L2-от места вставки до конца списка S2); произвести конкатенацию списков L1 с вставляемым списком S1, получить новый список L3; произвести конкатенацию списков L3 и L2, получить результирующий список S3.

Задачи перестановки элементов списка.

К задачам перестановки элементов списка можно отнести:

- обращение списков (имея список, получить новый список, в котором элементы расположены в обратном порядке);
- циклическая перестановка элементов списка на заданное число шагов ([1,2,3,4]-исходный список, [4,1,2,3] или [2,3,4,1]-результрующий список - перестановка на 1 шаг; [1,2,3,4]-исходный список, [3,4,1,2]-результирующий список - перестановка на 2 шага; и т.д.).

В общем решение этих задач на естественном языке можно описать при помощи правил:

Если исходный список пуст, то и результирующий список должен быть пустым.

Если исходный список не пуст, то разбить его на голову и хвост [H|T],получить список T1-перенося в него, элементы списка T, вставить H в нужное место списка T1 и повторить такие действия заданное количество раз.

Задачи сортировки списков.

Задачи сортировки на практике применяются очень часто, например: в справочно-информационных системах, в экспертных системах и других программах, работающих с базами данных.

Сортировать можно различные структуры данных: массивы, списки, записи в базах данных, строки и т.д.

Список можно отсортировать (упорядочить),если между его элементами определено отношение порядка. Отношение порядка для списков с информацией различного типа может быть определено по-разному. Для числовых списков отношениями порядка могут быть: больше-его можно определить как (max(X,Y):-X>Y.) , или меньше - (min(X,Y):-X<Y.).

Сортировка списков сводится к перестановке их элементов, пока во всем списке не установится отношение порядка.

Для числовых списков сортировка сводится к размещению их элементов в порядке возрастания или убывания.

Рассмотрим несколько алгоритмов сортировки:

Метод пузырька:

Переносить элементы исходного списка S в список T, если соседние элементы списка H и H1 не удовлетворяют отношению порядка, если соседние элементы удовлетворяют отношению порядка, поменять их местами и перенести в список T ([H,H1|S1], [H1,H|T1]). Проделать то же с полученным списком.

Если в списке S нет ни одной пары соседних элементов, удовлетворяющих отношению порядка, то считать, что полученный список уже отсортирован.

Метод вставок:

Для того, чтобы упорядочить непустой список S=[H|T], необходимо: Упорядочить T-хвост списка, получить список Sort_t. Вставлять голову H списка S в список Sort_t, в такое место, чтобы новый получающийся список Sort_lst был упорядоченным.

Метод быстрой сортировки:

Удалить из исходного списка какой-нибудь элемент H при разбиении [H|T] и разбить оставшуюся часть T на два списка Min, Max, следующим образом: все элементы исходного списка больше чем H переносить в список Max, а остальные в список Min. Проделать такие же действия со списком Min, т.е. отсортировать его и получить список Sort_min.

Проделать такие же действия со списком Max и получить список Sort_max.

Получить результирующий список Sort_lst, как конкатенацию списков Sort_min и [H|Sort_max].

Метод быстрой сортировки более эффективен по сравнению с методами пузырька и вставок.

4.3. СТАНДАРТНЫЕ ПРЕДИКАТЫ РАБОТЫ С ОКНАМИ

В языке TURBO PROLOG предусмотрен обширный набор предикатов работы с экраном и окнами.

Благодаря предикатам оконного интерфейса TURBO PROLOG обеспечивает возможность управления такими характеристиками экранного изображения, как инверсное изображение, мигание и цвета. Такая информация передается стандартными предикатами через значения атрибутов. Значение атрибута определяет цвет символов (основной цвет) и цвет за символами (фоновый цвет).В языке TURBO PROLOG предусмотренны два набора атрибутов для монохромного и цветного адаптеров.

Значения атрибутов монохромного адаптера дисплея

Атрибут	Значение
Черные символы на черном фоне(т.е. пустой экран)	0
Белые символы на черном фоне(нормальное изображение)	7
Черные символы на белом фоне(инверсное изображение)	112

Если компьютер снабжен монохромным адаптером дисплея, значения атрибутов экранного изображения вычисляются следующим образом:

- 1. Выбирается целое число, представляющее искомый атрибут, из таблицы атрибутов.
- 2. Если нужна более высокая интенсивность белой части изображения, к значению атрибута из таблицы прибавляется 8.
- 3. Если требуется мигание символа, к значению атрибута из таблицы прибавляется 128.

Для вычисления значений атрибутов цветного экранного изображения необходимо выполнить следующие действия:

- 1. Выбирать цвет символа и фоновый цвет из таблицы атрибутов цветного графического адаптера.
- 2. Сложить их, полученный результат использовать в предикатах.
- 3. Если необходимо мигание, к полученному результату прибавить 128.

Значения атрибутов цветного графического адаптера

Основные цвета	Значение	Фоновые цвета	Значение
Черный	0	Черный	0
Серый	8	Синий	16

Синий	1	Зеленый	32
Светло-синий	9	Бирюзовый	48
Зеленый	2	Красный	64
Светло-зеленый	10	Малиновый	80
Бирюзовый	3	Коричневый	96
Светло-бирюзовый	11	Белый	112
Красный	4		
Светло-красный	12		
Малиновый	5		
Светло-малиновый	13		
Коричневый	6		
Желтый	14		
Белый	7		
Белый(высокой интенсив-ности)	15		

Например,чтобы получить желтое изображение на красном фоне, значение атрибута должно быть 64+14, т.е 78.

Предикат attribute/1.

Назначение: Устанавливает или выдает отсутствующий атрибут вывода.
Обращение: attribute(Атр).
Домены: (integer).
Шаблоны описания аргументов: (i),(o).
Описание: attribute устанавливает или считывает текущий атрибут вывода. Вывод при помощи write или writef дает отображение на экране с этим атрибутом. Каждое окно имеет свой атрибут вывода. Когда вы создаете окно, атрибут вывода автоматически принимает значение атрибута окна, но вы можете изменить его предикатом attribute.

(i)

Устанавливает текущий атрибут вывода со значением Атр.

(o)

Связывает Атр со значением текущего атрибута.
Неудачное завершение: Никогда не дает.
Ошибки: Отсутствуют.

Пример:

```
goal
        attribute(1), write("\nЭта строка голубая"),
        attribute(2), write("\nЭта строка зеленая"),
        attribute(A), A1=A+2,
        attribute(A1), write("\nЭта строка красная").
```

Предикат makewindow/8.

Назначение: Создает новое окно на экране с указанными границами.
Обращение: makewindow(НомОкна,АтрЭкр,АтрРамки,СтрРамки, Строка,Столбец,Высота,Ширина)
Домены: (integer,integer,integer,string,integer,integer, integer,integer)
Шаблоны описания аргументов: (i,i,i,i,i,i,i,i) и (o,o,o,o,o,o,o,o)
Описание : (i,i,i,i,i,i,i,i)
makewindow определяет область экрана как окно.
Аргументы в makewindow следующие:
НомОкна: Каждое окно определяется номером НомОкна, который вы потом используете, выбирая активное окно.
АтрЭкр: Значение цветового атрибута внутренности окна.
АтрРамки: Значение цветового атрибута рамки окна. Если АтрРамки

не является нулем, makewindow рисует границу вокруг описываемой области (обрамляя окно).

СтрРамки: Строка, которая будет помещаться в центре линии верхней границы контура рамки. Если СтрРамки = “” (пустая строка), текста не будет в верхней линии рамки окна. Если текст строки шире, чем граница, он будет обрезаться.

Строка: Определяет вертикальные координаты верхнего левого угла окна относительно всего экрана.

Столбец: Определяет горизонтальные координаты верхнего левого угла окна относительно всего экрана.

Высота: Количество строк в окне, включая рамку.

Ширина: Количество символов в строке экрана, включая рамку.

Как только окно определено, экран очищается и курсор перемещается в верхний левый угол окна. Комбинация параметров Строка и Высота,Столбец и Ширина определяет окно, которое располагается полностью внутри экрана. Если какая-либо часть окна выходит за пределы экрана, то активизируется программа обработки ошибок. Размер экрана может быть изменен с помощью стандартного предиката textmode.

(o,o,o,o,o,o,o,o)

Этот шаблон аргументов позволяет определить текущие значения параметров текущего окна.

Неудачное завершение: Никогда не дает.

Ошибки: 1000 Аргументы в ‘makewindow’неверные.
1016 Максимальный номер окна превосходит границы.

Пример:

```
goal
        makewindow(1,20,7," Синее окно ",2,5,10,50),
        write("Символы красные"),nl,
        makewindow(2,176,7,"Светло-голубое
                    окно",14,25,10,40),
        write("Это окно светло-голубое"),
        write("буквы черные и мерцают"),nl,
        write("Для выхода введите целое"),nl, readint(_),
        removewindow, write("для выхода введите целое"),nl,
        readint(_),removewindow.
```

Окна могут накладываться. Если текст так велик, что не может поместиться в окне, будет выполняться его прокручивание, как и в случае полного экрана.

Предикат makewindow/11.

Назначение: Создает окно с заданной пользователем рамкой.

Обращение: makewindow(НомОкна,АтрЭкр,АтрРамки,СтрРамки, Строка,Столбец,Высота,Ширина, ЧиститьОкно,ПозСтрРамки,ЗнакиГраницы)

Домены: (integer,integer,integer,string,integer,integer, integer,integer,integer,integer,string)

Шаблоны описания аргументов: (i,i,i,i,i,i,i,i,i,i,i)
(o,o,o,o,o,o,o,o,o,o,o)

Описание: Используя эту версию makewindow,вы можете задавать символы для рамки окна, расположение заголовка окна, необходимость чистить окно после его создания. Первые восемь аргументов аналогичны аргументам в makewindow/8.

Дополнительные три аргумента в makewindow следующие:

ЧиститьОкно: Определяет будет ли чиститься окно после его создания:

0 = Не чистить окно.

1 = Чистить окно.

ПозСтрРамки: Определяет,где будет размещаться заголовок окна (внутри верхней линии рамки окна):

1 = Заголовок в центре.

N = Размещает заголовок с указанной позиции.

ЗнакиГраницы: Описывают,как рисовать рамку окна;этот аргумент состоит строго из шести символов, которые означают следующее:

1 символ Верхний левый угол
2 символ Верхний правый угол
3 символ Нижний левый угол
4 символ Нижний правый угол
5 символ Горизонтальная линия
6 символ Вертикальная линия

Например:
“\218\191\192\217\196\179” -граница из одной линии.
“\201\187\200\188\205\186” -граница из двух линий.
“++++|-“ или “++++-¦ “ -другие варианты описания границы.
Неудачное завершение: Никогда не дает.
Ошибки: 1000 Аргументы в makewindow неверные.
1016 Максимальный номер окна превосходит границы.

Пример:

```
goal
        makewindow(1,7,7,”Окно 1”,
                   1,1,10,40,1,0,”\176\176\176\176\176\179”),
        readchar(_),
        makewindow(2,7,7,”Окно 2”,6,20,10,40,0,30,”******”),
        readchar(_).
```

Когда окно создается, оно становится текущим окном, и в него автоматически будет направляться весь вывод информации. Все предикаты оконного интерфейса работают относительно текущего окна. Например, когда вызывается предикат cursor, чтобы поместить курсор в позицию 15 строки 4, отсчет строки 4 и позиции 15 производится в пределах текущего окна. Каждое окно сопровождается позицией курсора, которую программа “вспоминает” при переходе от окна к окну. При удалении окна содержимое экрана под окном автоматически восстанавливается.

Предикат colorsetup/1.

Назначение: Изменяет цвета в текущем окне.
Обращение: colorsetup(ОкнРамк)
Домены: (integer)
Шаблоны описания аргументов: (i)
Описание: colorsetup позволяет использовать интерактивное изменение цветов текущего окна и на экране можно видеть значения атрибутов. Используется такой же способ выбора
цветов, как и в среде TURBO PROLOG.
colorsetup(0) Изменение цвета окна (фона).
colorsetup(1) Изменение цвета рамки.
Неудачное завершение: Никогда не дает.
Ошибки: Отсутствуют.

Пример:

```
goal
        makewindow(1,7,7,”контрольное окно”,5,30,10,40),
        cursor(3,5),write(“Изменение цвета окна”),
        colorsetup(0),
        cursor(3,5),write(“Изменение цвета рамки”),
        colorsetup(1),
        clearwindow,
        makewindow(_,WindAttr,FrameAttr,_,_,_,_,_),
        write(“\n\nАтрибутОкна=”,WindAttr),
        write(“\n\nАтрибутРамки=”,FrameAttr), readchar(_).
```

Предикат framewindow/1.

Назначение: Изменяет атрибут для рамки окна.
Обращение: framewindow(АтрРамк).
Домены: (integer).

Шаблоны описания аргументов: (i).
Описание: Список используемых атрибутов нужно брать из таблицы атрибутов. При вызове с АтрРамк=0 рамка текущего окна вообще удаляется.
Неудачное завершение: Никогда не дает.
Ошибки: Отсутствуют.

Пример:

```
goal
                makewindow(1,23,7,"",5,5,10,50),
                write("привет"),nl,readchar(_),
                framewindow(0),
                makewindow(_,_,Fattr,_,A,B,C,D),
                write("Fattr=",Fattr,",",A,",",B,",",C,",",D),nl,
                write("опять привет"),nl,readchar(_),
                framewindow(112), write("еще привет"),nl,readchar(_).
```

Предикат framewindow/4.

Назначение: Изменяет атрибут и символы в рамке окна.
Обращение: framewindow(АтрРамк,ТекстРамк,Поз,Тип).
Домены: (integr,string,integer,string).
Шаблоны описания аргументов: (i,i,i,i).
Описание: У предиката framewindow следующие аргументы:
АтрРамк: (Цвет) Атрибут для рамки окна.
ТекстРамк: Верхняя линия рамки включает текст ТекстРамки.
Поз: Определяет, где будет размещена метка окна (в пределах верхней линии рамки).
-1 = метка помещается в центре
>0 = размещение метки определяется позицией (колонкой)
Тип: Определяет, как вычерчивается рамка окна;
этот аргумент содержит ровно шесть следующих знаков:
1 знак Верхний левый угол.
2 знак Верхний правый угол.
3 знак Нижний левый угол.
4 знак Нижний правый угол.
5 знак Горизонтальная линия.
6 знак Вертикальная линия.
Например:
"\218\191\192\217\196\179" определяют одинарную рамку.
"\201\187\200\188\205\186" определяют двойную рамку.
"++++|-" или "++++-¦ " -другие варианты описания рамки.
Неудачное завершение: Никогда не дает.
Ошибки: Отсутствуют.

Пример:

```
predicates
  delay(integer)
  changeframe(integer)
clauses
                delay(0):-!.
                delay(N):-N1=N-1,delay(N1).
                changeframe(-1):-!,changeframe(50).
                changeframe(N):- not(keypressed),
                framewindow(7,"НАЖМИТЕ ПРОБЕЛ",N,
                                "\176\176\176\176\176\179"), delay(2000),
                N1=N-1,changeframe(N1).  changeframe(_).
goal
                makewindow(1,23,7,"КОНТРОЛЬ",5,5,10,50),
                changeframe(0).
```

Предикат existwindow/1.

Назначение: Проверяет,существует ли окно с указанным номером.
Обращение: existwindow(НомОк).
Домены: (integer)
Шаблоны описания аргументов: (i).

Описание: existwindow выполняется успешно,если окно,определенное НомОк, существует.
Неудачное завершение: Если окно с номером НомОк не существует.
Ошибки: 1005 Означенное окно неизвестно.

Предикат shiftwindow/1.

Назначение: Переключает активное окно или возвращает номер текущего окна.
Обращение: shiftwindow(НомОкна)
Домены: (integer)
Шаблоны описания аргументов: (i),(o)
Описание: (i)

Делает текущим окно с номером , определенным в НомОкна. Окно, определяемое переменной Nокна, будет выведенно на экран. Курсор при этом будет помещен в ту позицию, в которой он находился в предыдущем сеансе работы с этим окном. Новое текущее окно будет создано заново в том случае, когда оно было стерто с момента его последней активизации.

(o)

Связывает переменную НомОкна с номером текущего окна.
Неудачное завершение: Никогда не дает.
Ошибки: 1005 Ссылка на неизвестное окно.
Пример:

```
goal
                makewindow(1, 7, 7, " Окно 1 ", 0, 0, 25, 40),
                makewindow(2, 7, 7, " Окно 2 ", 0, 40, 25, 40),
                makewindow(3, 7, 7, " Окно 3 ", 0, 0, 12, 80),
                makewindow(4, 7, 7, " Окно 4 ", 12, 0, 13, 80),
                shiftwindow(1),
                write("Это #1!\n\nНажмите любую клавишу"),
                readchar(_),shiftwindow(2),
                write("Сейчас вы находитесь в окне #2.
                \nЧто затем?\n\nНажмите любую клавишу."),
                readchar(_),shiftwindow(3),
                write("Вот что затем; окно #3.
                      \nНажмите любую клавишу."),
                readchar(_),shiftwindow(4),
write("Сейчас мы можем записать в окно #4.\n \nНажмите любую клавишу для
продолжения."),
                readchar(_),shiftwindow(1), write("\n\nЭто все !").
```

Предикат resizewindow/0.

Назначение: Изменяет размер текущего окна.
Обращение: resizewindow.
Описание: Вызов предиката resizewindow/0 позволяет изменить размер текущего окна при помощи клавиш управления курсором,точно так же, как это делается в интерактивной среде TURBO PROLOG.
Неудачное завершение: Никогда не дает.
Ошибки: 1017 Неверные аргументы в изменяемом окне.
Пример:

```
goal
                makewindow(1,7,7,"тест окна",5,5,15,70),
                write("\nизмените размер окна\n"),
                resizewindow,
                makewindow(Wno,Wattr,Fattr,Text,Srow,Scol,Rows,Cols),
                write("\nНовое состояние окна:"),
                writef("\nWno=%,Wattr=%,Fattr=%,Text=%,Srow=%,
                Scol=%,Rows=%,Cols=%",Wno,Wattr,Fattr,Text,Srow,Scol,
                Rows,Cols),
                readchar(_).
```

Предикат resizewindow/4.

Назначение: Изменяет местоположение и размер текущего окна.
Обращение: resizewindow(НачСтр,КолСтр,НачКол,КолКол).
Домены: (integer,integer,integer,integer).

Шаблоны описания аргументов: (i,i,i,i)
Описание: resizewindow/4 изменяет размер или местоположение (или то и другое) текущего окна; его аргументы описывают новое местоположение (начальная строка и колонка) и размеры (количество строк и колонок) для окна.

Пример:

```
                /* Создание увеличивающихся окон */
predicates
  exp_window(integer)
clauses
                exp_window(0):- !.
                exp_window(N):- makewindow(_,_,_,_,A,B,C,D),
                A1=A-1,B1=B-2,C1=C+2,D1=D+4,
                resizewindow(A1,B1,C1,D1), N1=N-1,exp_window(N1).
goal
                makewindow(1,7,7,"Тест 1",10,20,3,10), exp_window(9),
                makewindow(2,7,7,"Тест 2",12,50,3,10), exp_window(9),
                makewindow(3,7,7,"Тест 3",14,16,3,10), exp_window(8).
```

Предикат removewindow/0.

Назначение: Удаляет текущее окно.
Обращение: removewindow.
Описание: removewindow/0 удаляет текущее окно с экрана и переходит к предшествующему окну. Если удаляемое окно перекрывало предшествующее ему, то закрываемое изображение восстанавливается.
Неудачное завершение: Никогда не дает.
Ошибки: 5101 Системное окно (номокн>127) не может быть удалено.
Пример:

```
goal
                makewindow(1,7,7,"Проверка окна",5,5,15,70),
                write("\nНажмите любую клавишу - удалить окно"),
                readchar(_), removewindow.
```

Предикат removewindow/2.

Назначение: Удаляет определенное окно.
Обращение: removewindow(НомОкна,Регенер).
Домены: (integer,integer).
Шаблоны описания аргументов: (i,i).
Описание: removewindow/2 удаляет текущее окно,определенное параметром НомОкна, которое может быть не только текущим. Значение параметра Регенер определяет, удалять ли указанное окно с экрана или не удалять.
Регенер = 0 Не удалять окно с экрана.
= 1 Удалять окно с экрана.
Неудачное завершение: Никогда не дает.
Ошибки: 5101 Системное окно (НомОкн>127) не может быть удалено.
Пример:

```
goal
                makewindow(1,7,7,"Проверка окна",5,5,15,70),
                write("\nНажмите любую клавишу - удалить окно"),
                readchar(_), removewindow(1,0).
```

Предикат scroll/2.

Назначение: Перемещает содержимое текущего окна.
Обращение: scroll(НомСтр,НомКол).
Домены: (integer,integer).
Шаблоны описания аргументов: (i,i).
Описание: scroll перемещает содержимое текущего окна вверх (или вниз) и влево (или вправо). НомСтр показывает число строк, на которое текст перемещается вверх или вниз. Положительное число - перемещение вверх, отрицательное число - перемещение вниз. НомКол показывает число перемещений влево или вправо. Положительное число -

перемещение влево, отрицательное число - перемещение вправо.

Неудачное завершение: Никогда не дает.

Ошибки: Нет ошибок.

Пример:

```
goal
                makewindow(2,112,0,"",21,5,1,70),
                makewindow(1, 12, 7, "Окно перемещения", 5, 5, 15, 70),
                write("Перемещение текста вверх"),scroll(3,0),
                write("Перемещение текста вниз"),scroll(-6,0),
                write("Перемещение текста вверх"),scroll(3,0),
                write("Перемещение текста влево"),scroll(0,15),
                write("Перемещение текста вправо"),scroll(0,-30),
                write("Перемещение текста влево"),scroll(0,15).
```

Рассмотрим пример программы, в которой оконный интерфейс языка TURBO PROLOG используется для создания игры в угадывание слов.

Для краткости, действия программы достаточно примитивны, но окна, несмотря ни на что, делают ее экранное представление весьма выразительным. Игрок должен угадать всего шесть слов. Сначала программа просит игрока ввести букву. Если эта буква есть в слове, она помещается в окно "YES". Если же ее там нет, то буква помещается в окно "NO". После каждого угадывания буквы у игрока запрашивается все слово, которое должно быть введено по буквам с нажатием после каждой буквы Enter. Программа также фиксирует суммарное число угаданных слов.

```
domains
   list   = symbol*
   scores = integer
predicates
   member(symbol, list)
   run
   continue(list, scores)
   yes_no_count(symbol, list)
   guessword(scores, list)
   word(list, integer)
   read_as_list(list, integer)
goal
                makewindow(1, 7, 0, "", 0, 0, 25, 80),
                makewindow(2, 7, 135, "Counting", 1, 20, 4, 34),
                makewindow(3, 112, 112, "YES", 5, 5, 7, 30),
                makewindow(4, 112, 112, "NO", 5, 40, 7, 30),
                makewindow(5, 7, 7, "", 14, 20, 10, 34),
                run.
clauses
                run :- word(W, L),shiftwindow(1), clearwindow,
                       write("В слове ",L," букв"),shiftwindow(2),
                       clearwindow,shiftwindow(3), clearwindow,
                       shiftwindow(4), continue(W, 0), fail.

                continue(L, R) :-shiftwindow(5), clearwindow,
                                 write("Угадайте букву:"),Total=R+1,
                                 readln(T), yes_no_count(T, L),
                                 shiftwindow(5), clearwindow,
                                 guessword(Total, L), continue(L, Total).

                yes_no_count(X, List) :-member(X, List), shiftwindow(3),
                                        write(X), shiftwindow(2), write(X), !.

                yes_no_count(X, _) :-shiftwindow(4), write(X),
                                     shiftwindow(2), write(X).
```

```
guessword(Count, Word) :-write("Уже знаете слово? Нажми
                                те д или н"),
      readchar(A), A='y', cursor(0, 0),
      write("Вводите его по одной букве в строке \n"),
      word(Word, L), read_as_list(G, L), G=Word,
      clearwindow, window_attr(112),
write("Правильно! Вы использовали ",Count," попыток"),
readchar(_), window_attr(7), !, fail.

guessword(_, _).
word([b, i, r, d], 4).
word([p, r, o, l, o, g], 6).
word([f, u, t, u, r, e], 6).
word([a, r, t, i, f, i, c, i, a, l], 10).
word([p, o, w, e, r, f, u, l], 8).
word([e, l, e, g, a, n, t], 7).

member(X, [X|_]) :- !.
member(X, [_|T]) :- member(X, T).

read_as_list([], 0) :- !.
read_as_list([Ch|Rest], L) :- readln(Ch), L1=L-1,
                              read_as_list(Rest, L1).
```

ЗАДАНИЕ К ТЕМЕ :

Создать интерактивную среду, реализующую основные задачи обработки списков.

5. СТРОКИ

Для работы с символьной информацией в языке TURBO PROLOG существует три стандартных типа:char,string,symbol.

CHAR - любой символ заключенный в апострофы. ‘Ф’,’r’,’+’,’_’, и т.д.

STRING - совокупность символов заключенная в кавычки. “”-пустая строка. “TURBO PROLOG”,”школа”,”+++++----- @@&&{” и т.д. В строках символ (\) представляет собой символ переключения.Для печати же этого символа, как такового, должны быть указаны подряд два знака \.Например,для задания пути поиска ДОС A:\TPROLOG2\MYPROG\MYFILE.PRO в программе на языке TURBO PROLOG должно быть записано “a:\\tprolog2\\myprog\\myfile.pro”. Строковые константы, строковые переменные вводимые с клавиатуры могут содержать 255 символов, но строки которые TURBO PROLOG считывает из файлов или строит внутри себя могут быть длиной 64кБайт.

SYMBOL - для этого типа предусмотренно несколько форматов:

- последовательность букв, цифр и знаков подчеркивания, начинающаяся со строчной буквы и без пробелов. turbo, _1234 и т.д.
- или любая совокупность символов заключенная в кавычки.
- “Myfile.pro”,”TURBO PROLOG” и т.д.

Типы STRING и SYMBOL совместимы.Их различие проявляется только во внутреннем представлении.Величины типа SYMBOL хранятся в таблице просмотра и для их представления используются адреса символов,а в величинах типа STRING хранятся сами символы.

5.1. СТАНДАРТНЫЕ ПРЕДИКАТЫ ОБРАБОТКИ СТРОК

Эти предикаты служат для:

- разбиения строк на строчные компоненты или лексемы(tokens);
- построения строк из заданных строк или лексем;
- проверки того, что строка состоит из заданных строк или лексем;
- получения строки, лексемы или списка этих элементов из заданной строки;
- проверки или получения длины строки;
- создания пустой строки заданной длины;
- проверки того,что строка-это допустимое в языке TURBO PROLOG имя;
- форматирования переменного числа аргументов в пределах строчной переменной.

Предикат readln/1.

Назначение: Читает строку символов.
Обращение: readln(ПеременнаяСтр)
Домены: (string)
Шаблоны описания аргументов: (o)
Описание: readln читает символы с текущего устройства ввода, пока не прочитает символ возврат каретки (ASCII 13). Текущим устройством ввода является по умолчанию клавиатура, пока оно не будет изменено с помощью readdevice. Если нажат Esc (ключ), readln немедленно дает неудачный исход.
Неудачное завершение: Если был нажат Esc (ключ) во время ввода с клавиатуры.
Если во время ввода файла встретился знак конца файла.
Ошибки: Нет ошибок.

Пример:

```
Goal: readln(L)
Это строка
L=Это строка
1 Solution
Goal: readln(L)
No Solution     Нажат Esc (Ключ)
Goal: readln(L)
Пустая строка(нажат только Enter (Ввод))
это не тоже самое как Esc
```

L=
1 Solution.

Выводить строки на текущее устройство вывода можно при помощи предикатов write, writef.

Для ввода/вывода строк на экран/с экрана, с изменением атрибутов используются предикаты:

Предикат field_str/4.

Назначение: Пишет или читает строку в заданном поле текущего окна с текущими атрибутами.
Обращение: field_str(Стр,Кол,Длн,Строка).
Домены: (integer,integer,integer,string).
Шаблоны описания аргументов: (i,i,i,i),(i,i,i,o)
Описание: (i,i,i,i)

Выводит значение Строка с позиции,определяемой координатами Стр, Кол.Эта позиция должна находиться в пределах текущего окна (см. makewindow и shiftwindow), и все поле длины Длн, начинающееся в этой позиции, должно размещаться в пределах текущего окна. Вывод значения зависит от следующих условий:

Если Строка имеет длину более чем значение Длн,то будут выведены только первые символы в количестве Длн.

Если Строка короче,чем значение Длн,то остальные позиции в поле будут заполнены пробелами.

(i,i,i,o)

Считывает текст, размещенный в поле в переменную Строка. Поле размером Длн символов начинается в позиции с координатами Стр и Кол в текущем окне. Как и в случае (i,i,i,i), указанное поле должно размещаться в окне.

Неудачное завершение: Никогда не дает.
Ошибки: 1001 Не правильное значение курсора.

Пример:

```
goal
field_str(2,5,10,"привет").
```

Предикат field_attr/4.

Назначение: Устанавливает или возвращает атрибуты поля.
Обращение: field_attr(Стр,Кол,Длн,Атр).
Домены: (integer,integer,integer,integer).
Шаблоны описания аргументов: (i,i,i,i),(i,i,i,o).
Описание: (i,i,i,i)

Если строка Стр и столбец Кол соответствуют позиции в пределах текущего окна, и поле данной длины Длн, начинающееся в этой позиции, может быть создано внутри этого окна, для всех позиций в поле устанавливается атрибут Атр.

(i,i,i,o)

field_attr связывает Атр со значением атрибута первого символа в указанном поле. Количество символов, занимаемых полем в текущем окне, определяется аргументом Длн. Как и в случае (i,i,i,i), указанное поле должно размещаться в окне.

Неудачное завершение: Никогда не дает.
Ошибки: 1001 Недопустимые значения позиции курсора.

Пример:

```
domains
   fname = string
   row, col, len, atr = integer
predicates
  field(fname,row,col,len,atr)
  writescr
clauses
                    writescr:- field(Str,Row,Col,Len,Atr),
                               field_attr(Row,Col,Len,Atr),
                               field_str(Row,Col,Len,Str),fail.
                    writescr.
```

```
field(prolog,2,30,6,30).
field(pascal,4,30,6,28).
field(basic,6,30,5,63).
field(lisp,8,30,4,94).
```

Предикат window_attr/1.

Назначение: Устанавливает атрибут активного окна.
Обращение: window_attr(Атрибут).
Домены: (integer).
Шаблоны описания аргументов:(i).
Описание: window_attr устанавливает для текущего окна значение атрибута, которое связано с переменной Атрибут.
Неудачное завершение: Никогда не дает.
Ошибки: Нет ошибок.
Пример:

```
GOAL
        makewindow(1,7,0,"",0,0,25,80),window_attr(7),
        write("\nСейчас атрибут - белые символы на черном
              фоне"),readchar(_),window_attr(112),
        write("\nСейчас атрибут - черные символы на белом
              фоне"), readchar(_).
```

Предикат window_str/1.

Назначение: Считывает или записывает строку из активного окна или в активное окно.
Обращение: window_str(СтрокаЭкрана).
Домены: (string).
Шаблоны описания аргументов: (i),(o).
Описание: (o)

Связывает переменную СтрокаЭкрана со строкой, размещенной в настоящий момент в активном окне; поэтому переменная СтрокаЭкрана содержит те же строки, какие имеются в активном окне. Длина каждой строки определяется последним непустым символом в строке.

(i)

СтрокаЭкрана выводится в окно согласно следующим критериям:

- Если строк в переменной больше,чем строк в окне, то window_str выводит их, пока пространство окна не будет заполнено.
- Если строк в переменной меньше,чем строк в окне, то window_str заполняет остальные строки в окне пробелами.
- Если в переменной больше символов,чем имеется в распоряжении на строке окна,window_str обрезает строку в соответствующую по размерам.
- Если символов в переменной меньше,чем колонок в окне,window_str заполняет остаток строки пробелами.

Неудачное завершение: Никогда не дает.
Ошибки: Нет ошибок.
Пример:

```
goal
        makewindow(1,7,0,"",0,0,25,80),
        makewindow(2,7,7,"",5,10,10,50),
        file_str("prolog.err",STR1),window_str(STR1),
        readchar(_),window_str(STR2),
        removewindow,window_str(STR2).
```

Предикат format/*.

Назначение: Форматирует несколько аргументов в строке.
Обращение: format(ВыхСтр,ФормСтр,Арг1,Арг2,...,АргN).
Шаблоны описания аргументов: (o,i,i...).
Описание: format выполняет некоторое форматирование подобно writef, но присваивает результат переменной.Арг1,АргN должны быть переменными или константами, принадлежащими к стандартным доменам.Форматируемая строка содержит обычные символы, которые печатаются без преобразования, и указатели формата в виде %-m.pf; это означает, что формат определен таким образом:

дефис (-)	Указывает, что поле должно выравниваться слева (выравнивание справа принимается по умолчанию).
поле m	Десятичное число - минимальная длина поля.
поле . p	Десятичное число - точность значения с плавающей точкой, либо максимальное число символов строки, которые должны выводиться.
поле f	Определяет форматы, отличные от принимаемого по умолчанию для данного объекта. Например, поле f может указывать, что целые должны выводиться как числа без знака или шестнадцатеричные числа, или вещественные должны выводиться в шестнадцатеричном виде.

Например, в поле f можно определить указатель, который отформатирует целое число так, что оно будет представлено без знака, или char будет представлено в шестнадцатиричном виде. format (формат) воспринимает следующие указатели f поля:

f	вещественные как десятичные с фиксированной точкой (скажем, 123. 4 или 0. 004321);
e	вещественные в экспоненциальном виде (скажем, 1. 234e2 или 4. 321e-3);
g	вещественные в коротком формате f или e (для вещественных значений принимается по умолчанию);
d	символы или целые как десятичные числа;
u	символы или целые как целые без знака;
x	символы или целые как шестнадцатиричные числа;
c	символы или целые как символы;
R	как число обращения к базе данных (только для домена ref);
X	как длинное шестнадцатиричное число (строки, числа обращения к базе данных);
s	как строка (типы symbol и string).

Неудачное завершение: Никогда не дает.

Ошибки: 1025 Недостаточно аргументов в строке,задающей формат.

Пример:

```
Goal: format(X,"this % the %'st % test",is,1,"small")
X=this is the 1'st small test
1 Solution.
Goal: format(X,"realS: Default=%, Exp=%e,
Shortest=%g",99E9,99E9,99E9)
X=realS: Default=98999999999.99999989, Exp=9.9E+10, Shortest=9.9E+10
1 Solution.
Goal: format(X,"char=%c, int=%, unsigned=%u,hex=%x",97,-1,-1,-1)
X=char=a, int=-1, unsigned=65535, hex=FFFF
1 Solution.
```

Рассмотрим предикаты позволяющие преобразовывать строки. Следует помнить-эти предикаты не всегда корректно работают с символами кириллицы.

Предикат frontchar/3.

Назначение: Возвращает первый символ в строке.

Обращение: frontchar(Строка,ПервСимв,ОстСтр).

Домены: (string,char,string).

Шаблоны описания аргументов : (i,o,o),(i,i,o),(i,o,i),(i,i,i),(o,i,i).

Описание: frontchar работает, как определено равенством:

Строка = (конкатенация ПервСимв и ОстСтр)

frontchar имеет три аргумента; первый из них строка, второй - символ (char) (первый символ первой строки), и третий - остальная часть первой строки.

(i,o,o)

В этом случае frontchar используется для разделения содержимого переменной Строка на: первый символ- связывается с переменной ПервСимв, оставшаяся часть- связывается с переменной ОстСтр.

(i,i,o)

Если значение переменной ПервСимв является первым символом переменной Строка, то переменная ОстСтр связывается со значением переменной Строка без первого символа. В противном случае следует неудачное завершение.

(i,o,i)

Если значение переменной ОстСтр совпадает со значением переменной Строка без первого символа, то переменная ПервСимв будет связана первым символом переменной Строка. В противном случае следует неудачное завершение.

(i,i,i)

Выполняется успешно, если значение переменной Строка может быть получено конкатенацией значений переменных ПервСимв и ОстСтр. В противном случае следует неудачное завершение.

(o,i,i)

Связывает значение переменной Строка результатом конкатенации переменных ПервСимв и ОстСтр.

В рассмотренных случаях, неудачный исход возможен и когда аргумент Строка будет связан значениемпустая строка.

Ошибки: 1034 Выходная строка в frontchar не может быть больше 64Кбайт.

Пример:

```
Goal: frontchar("TPROLOG",Ch,Rest)
CH=T, Rest=PROLOG
1 Solution.
Goal: frontchar("TPROLOG",'T',Rest)
Rest=PROLOG
1 Solution.
Goal: frontchar("TPROLOG",'P',"ROLOG")
No.
Goal: frontchar(X,'T',"PROLOG")
X=TPROLOG
1 Solution.
```

Предикат frontstr/4.

Назначение: Разделяет строку на две части в заданной позиции.
Обращение: frontstr(КолСимв,Строка,НачСтр,ОстСтр).
Домены: (integer,string,string,string).

Шаблоны описания аргументов: (i,i,o,o).

Описание: frontstr разделяет строку на две части.НачСтр содержит первые символы строки Строка в количестве КолСим, а ОстСтр содержит остальные символы строки Строка.

Неудачное завершение: Смотри описание.

Ошибки: Отсутствуют.

Пример:

```
Goal: frontstr(3,"All boys do fine",Str1,Str2)
Str1=All, Str2= boys do fine
1 Solution.
Goal: frontstr(9,"All boys do fine",Str1,Str2)
Str1=All boys , Str2=do fine
1 Solution.
Goal: frontstr(0,"All boys do fine",Str1,Str2)
Str1=, Str2=All boys do fine
1 Solution.
Goal: frontstr(-1,"All boys do fine",Str1,Str2)
0 Solution.
Goal: frontstr(20,"All boys do fine",Str1,Str2)
0 Solution.
```

Предикат fronttoken/3.

Назначение: Возвращает первое обозначение в строке.
Обращение: fronttoken(Строка,Знак,ОстСтр).
Домены: (string,string,string).

Шаблоны описания аргументов: (i,o,o),(i,i,o),(i,o,i),(i,i,i),(o,i,i).

Описание: действие fronttoken определяется равенством:
Строка = (конкатенация Знак и ОстСтр)

Данный предикат работает со строками,которые в языке TURBO PROLOG являются обозначениями.

Последовательность из одного или более символов составляет обозначение в следующих случаях:

- Последовательность символов составляет <имя>, соответствующее синтаксису языка TURBO PROLOG.
- Последовательность символов составляет целое или вещественное число, соответствующее синтаксису языка TURBO PROLOG (знак числа представляет собой также отдельное обозначение).
- Последовательность состоит из одного символа, но это не символ пробела ASCII (десятичный код 32).
- Слова русского языка не являются обозначениями, соответствующими синтаксису языка TURBO PROLOG.

(i,i,i)

fronttoken дает удачный исход, если переменная Знак связана с первым обозначением в переменной Строка, а переменная ОстСтр связана с остальной частью переменной Строка.

(i,o,o)

fronttoken связывает второй аргумент с первым обозначением в строке и третий аргумент с остальной частью строки.

(i,i,o),(i,o,i),(o,i,i)

fronttoken дает удачный исход при связывании подходящего (o) аргумента с соответствующей частью строки.

Неудачное завершение: Смотри описание.

Ошибки: 2008 Результирующая строка concat или fronttoken не может быть больше 64Кбайт.

Пример:

```
Goal: fronttoken("all boys do fine",Tok,Rest)
Tok=all, Rest= boys do fine
1 Solution.
Goal: fronttoken("all+boys do fine",Tok,Rest),
fronttoken(Rest,Tok1,_)
Tok=all, Rest=+boys do fine, Tok1=+ 1 Solution.
Goal: fronttoken("22all boys do fine",Tok,Rest)
Tok=22, Rest=all boys do fine
1 Solution.
Goal: fronttoken("22.66all boys do fine",Tok,Rest)
Tok=22.66, Rest=all boys do fine
1 Solution.
Goal: fronttoken("-22.66all boys do fine",Tok,Rest)
Tok=-, Rest=22.66all boys do fine
1 Solution.
Goal: fronttoken(".66all boys do fine",Tok,Rest)
Tok=., Rest=66all boys do fine
1 Solution.
```

Предикат isname/1.

Назначение: Проверяет,является ли строка идентификатором.

Обращение: isname(ПерСтроки).

Домены: (string).

Шаблоны описания аргументов: (i)

Описание: isname дает удачный исход, если ПерСтроки является <идентификатором>, соответствующим синтаксису языка TURBO PROLOG.В противном случае следует неудачное завершение.

Идентификатором является любая последовательность букв (латинских), цифр и символов подчеркивания, начинающаяся с буквы или символа подчеркивания.

Ошибки: Нет ошибок.

Пример:

```
Goal: isname("Dan")
Yes
Goal: isname("2Лео")
```

No
Goal: isname("_Джон")
No
Goal: isname("one_str")
Yes

Предикат concat/3.

Назначение: Соединяет две строки (конкатенация).
Обращение: concat(Стр1,Стр2,СуммСтр).
Домены: (string,string,string).
Шаблоны описания аргументов: (i,i,o),(o,i,i)(i,o,i)(i,i,i).
Описание: concat выполняется, как определено равенством

СуммСтр = Стр1 + Стр2

(i,i,o)

Связывает переменную СуммСтр значением,которое является результатом конкатенации значений переменных Стр1,Стр2.

(o,i,i)

Дает удачный исход и связывает переменную Стр1 той частью строки СуммСтр, которая бы при конкатенации со значением переменной Стр2 давала значение переменной СуммСтр. В противном случае дает неудачное завершение.

(i,o,i)

Дает удачный исход и связывает переменную Стр2 той частью строки СуммСтр,которая бы при конкатенации со значением переменной Стр1 давала значение переменной СуммСтр.В противном случае дает неудачное завершение.

(i,i,i)

Дает удачный исход, если значение переменной СуммСтр является результатом конкатенации значений переменных Стр1,Стр2.В противном случае дает неудачное завершение.

Неудачное завершение: Смотри описание.
Ошибки: 2008 Результирующая строка в 'concat' или 'fronttoken' не может быть длиннее 64 Кбайт.
Пример:
Goal: concat("aaa","bbb",X)
X=aaabbb
1 Solution.
Goal: concat("aaa","bbb","aaabbb")
Yes
Goal: concat("aaa","bbb","aaa----bbb")
No
Goal: concat(X,"bbb","aaa----bbb")
X=aaa----
1 Solution.
Goal: concat("aaa",X,"aaa----bbb")
X=----bbb
1 Solution.

Предикат str_len/2.

Назначение: Возвращает длину строки.
Обращение: str_len(Стр,Длн).
Домены: (string,integer).
Шаблоны описания аргументов: (i,i),(i,o),(o,i).
Описание: (i,i)

Дает удачный исход,если строка Стр имеет длину Длн символов.В противном случае дает неудачное завершение.

(i,o)

Переменную Длн связывает количеством символов в строке Стр.

(o,i)

Переменную Стр связывает значением строки, состоящей из Длн пробелов.
Неудачное завершение: Смотри описание.
Ошибки: Нет ошибок.
Пример:
Goal: str_len("abc",Len)

```
Len=3
1 Solution
Goal: str_len("abc",3)
Yes
Goal: str_len(STR,15),writef("|%|\n",STR)
        |           |
STR=
1 Solution
```

Рассмотрим предикаты осуществляющие преобразование типов.

Предикат str_char/2.

Назначение: Выполняет преобразование (обращение) между строкой с единственным символом и символом.
Обращение: str_char(ПарамСтр,ПарамСимв).
Домены: (string,char).

Шаблоны описания аргументов:(i,o),(o,i),(i,i).

Описание: (o,i)

Связывает ПарамСтр со строкой, содержащей символ, являющийся значением переменной ПарамСимв.

(i,o)

Связывает переменную ПарамСимв с единственным символом, содержащимся в переменной ПарамСтр.

(i,i)

Дает удачный исход, если ПарамСимв и ПарамСтр связаны одним и тем же символом.В противном случае дает неудачное завершение.

Неудачное завершение: Варианты потоков (i,o) и (i,i) всегда будут давать неудачный исход, если ПарамСтр не содержит точно один символ.

Ошибки: Нет ошибок.

Пример:

```
Goal: str_char("A",CH)
CH=A
1 Solution
Goal: str_char(STR,'A')
STR=A
1 Solution
Goal: str_char("A",'A')
Yes
Goal: str_char("AB",CH)
No Solution
Goal: str_char("B",'A')
No
```

Предикат str_int/2.

Назначение: Выполняет преобразование (обращение) между строкой и целым числом.
Обращение: str_int(ПарамСтр,ПарамЦел).
Домены: (string,integer).

Шаблоны описания аргументов: (i,o),(o,i),(i,i).

Описание: (o,i)

Связывает ПарамСтр со строкой десятичных цифр, представляющих значение, с которым связан ПарамЦел.

(i,o)

Связывает ПарамЦел с целым числом,символы которого содержатся в переменной ПарамСтр.

(i,i)

Дает удачное завершение,если ПарамЦел является целым числом, состоящим из таких же цифр, что и значение переменной ПарамСтр.

Неудачное завершение: Варианты потоков (i,o) и (i,i) будут давать неудачный исход, если ПарамСтр не образует правильное целое.В начале строки знак необязателен, начальные пробелы игнорируются.

Ошибки: Нет ошибок.

Пример:

```
Goal: str_int("123",INT)
INT=123
```

```
1 Solution.
Goal: str_int(STR,123)
STR=123
1 Solution.
Goal: str_int("123",123)
Yes
Goal: str_int("  -123  ",INT)
INT=-123
1 Solution.
Goal: str_int("  -1x23  ",INT)
No Solution.
```

Предикат str_real/2.

Назначение: Выполняет преобразование (обращение) между строкой и вещественным числом.

Обращение: str_real(ПарамСтр,ПарамВещ).

Домены: (string,real).

Шаблоны описания аргументов: (i,o),(o,i),(i,i)

Описание: (i,o)

Связывает ПарамВещ с вещественным числом,которое выражают символы переменной ПарамСтр.

(o,i)

Связывает ПарамСтр со строкой десятичных цифр, представляющих значение вещественного числа, с которым связана переменная ПарамВещ.

(i,i)

Дает удачный исход, если переменная ПарамВещ связана таким вещественным числом, которое выражают символы переменной ПарамСтр.

Неудачное завершение:Варианты потоков (i,o) и (i,i) будут давать неудачный исход, если ПарамСтр не образует правильное десятичное число с плавающей точкой.

Ошибки: Нет ошибок.

Пример:

```
Goal: str_real("23456789079856",REAL)
REAL=2.345678908E+13
1 Solution
Goal: str_real(STR,-123234.453E-99)
STR=-1.23234453E-94
1 Solution
```

Рассмотрим примеры программ обработки строк.

Пример 1:

```
domains
   charlist = char*
predicates
   string_chlist(string, charlist)
clauses
                    string_chlist("", []).
                    string_chlist(S, [H|T]) :- frontchar(S, H, S1),
                                               string_chlist(S1, T).
```

В этой программе предикат frontchar используется для описания предиката string_chlist, который разбивает строку на символы и из них образует список.

Пример 2:

```
domains
  namelist = name*
  name  = symbol
predicates
  string_namelist(string, namelist)
clauses
                    string_namelist(S, [H|T]) :- fronttoken(S, H, S1), !,
                                                 string_namelist(S1, T).
                    string_namelist(_, []).
```

В этой программе предикат fronttoken используется для описания предиката string_namelist, который разбивает предложения на слова (лексемы) из них образует список.

Пример 3:

```
domains
   tok  = numb(integer); name(string); char(char)
   toklist = tok*
predicates
   scanner(string, toklist)
   maketok(string, tok)
clauses
                    scanner("", []).
                    scanner(Str, [Tok|Rest]) :- fronttoken(Str, Sym, Str1),
                                                maketok(Sym, Tok),
                                                scanner(Str1, Rest).
                    maketok(S, name(S)) :- isname(S).
                    maketok(S, numb(N)) :- str_int(S, N).
                    maketok(S, char(C)) :- str_char(S, C).
goal
                    write("ВВЕДИТЕ ПРЕДЛОЖЕНИЕ:"),nl,readln(Text),nl,
                    scanner(Text,T_List),write(T_List).
```

В этой программе предложение разбивается на слова, слова классифицируются в соответствующие их типу структуры и образуется список структур.

ЗАДАНИЕ К ТЕМЕ :

Создать интерактивную среду, которая бы позволяла различать по родам и числам имена существительные, и изменяла их по падежам.

6. БАЗЫ ДАННЫХ И РАБОТА С ФАЙЛАМИ

6.1. ВНУТРЕННИЕ БАЗЫ ДАННЫХ

Базы данных-это способ организации больших объемов структурированной информации (т.е. связанная совокупность структур данных).

Язык программирования PROLOG с учетом его механизмов:сопоставления и возврата, является естественным языком запросов к базе данных. В языке TURBO PROLOG 2.0 предусмотрены большие стандартные возможности ведения разнообразных баз данных. В нем предусмотрена работа с двумя видами баз данных:

Внутренними-динамическими базами данных, являющимися частью программы. Они обычно небольшие по объему. Внешними базами данных, не являющимися частью программы. Их размеры ограничены только возможностями компьютера. В связи с этим TURBO PROLOG 2.0 можно рассматривать как универсальную систему управления базами данных.

Здесь мы познакомимся с внутренними базами данных.

Внутренние-динамические базы данных состоят из фактов, которые могут непосредственно добавляться в программу или удаляться из нее в процессе выполнения.Они вместе с программой представляют единое целое.

Предикаты внутренних баз данных могут играть роль структур данных (так как у них одинаковый синтаксис) и являться аргументами предикатов, описанных в разделе predicates. В этом случае на месте такого аргумента записывается символьное имя базы данных, а если используется неименованная база данных, то записывается зарезервированное слово dbasedom.

Механизмы: сопоставления (поиска), возврата (поиска с возвратом) работают с внутренними базами данных как с фактами, находящимися в тексте программы. Механизм сопоставления ищет факты с данными аргументами и присваивает значения свободным переменным, механизм возврата позволяет найти все решения для поставленной цели.

Для создания и работы с внутренними базами данных их предварительно необходимо описать в разделе программы database.

6.1.1. ОПИСАНИЕ ВНУТРЕННИХ БАЗ ДАННЫХ

Описание внутренних баз данных осуществляется в разделе database. Этот раздел обычно следует после раздела domains.

```
database
  имя_предиката1(Арг1,Арг2,...,АргN)
  имя_предиката2(Арг1,Арг2,...,АргN)
  ..................................................
```

В программе допускается наличие нескольких разделов database, но в этом случае каждый раздел должен быть поименован т.е. получить свое собственное символьное имя.

```
database - mydb1
 ..............................

database - mydb2
 ..............................
```

В предикатах, работающих с такими базами данных, нужно использовать эти имена. При создании и использовании нескольких разделов database нужно помнить, что имена предикатов базы данных в пределах одного программного модуля являются неповторяющимися. В двух разных разделах database одно и то же имя предиката использовать нельзя.

6.1.2. ВВОД ФАКТОВ В БАЗУ ДАННЫХ

При создании баз данных есть два ограничения:

- 1)Выражения могут добавляться в базу данных только как факты,но не как правила.
- 2)Факты базы данных не могут иметь несвязанных переменных.

СТАНДАРТНЫЕ ПРЕДИКАТЫ:

Предикат assertz/1.

Назначение: Включает факт в конец динамической базы данных.
Обращение: assertz(<факт>).
Домены: (<соответствующие домены динамической базы данных>).
Шаблоны описания аргументов: (i).
Описание: assertz включает <факт> в неименованную динамическую базу данных после всех фактов для соответствующего предиката (т.е. в конец). Факт должен быть термом, принадлежащим к домену секции доменов динамической базы данных.
Неудачное завершение: Никогда не дает.
Ошибки: Отсутствуют.
Пример:

```
domains
  name, address, cityname = string
  age, zipcode = integer
database
  person(name, age, address, zipcode)
  city(zipcode, cityname)
goal
assertz(person("Миша", 26, "", 8600)), assertz(person("Вова", 27, "", 8600)),
assertz(person("Саша", 23, "", 9800)), assertz(city(8600, "Мелитополь")),
assertz(city(6800, "Киев")).
```

Предикат assertz/2.

Назначение: Включает факт в конец указанной динамической базы данных.
Обращение: assertz(<факт>,ИмяДБД).
Домены: (<соответствующие домены динамической базы данных>, ИмяБазыданных).
Шаблоны описания аргументов: (i,i).
Описание: assertz включает <факт> в динамическую базу данных после всех фактов для соответствующего предиката (т.е. в конец). Факт должен быть термом, принадлежащим к домену секции доменов динамической базы данных с именем ИмяДБД.
Неудачное завершение: Смотри assertz/1.
Ошибки: Смотри assertz/1.
Пример:

```
domains
   name, address, cityname = string
   age, zipcode = integer
database - persons          /* Имя = persons */
  person(name, age, address, zipcode)
database - cities           /* Имя = cities  */
  city(zipcode, cityname)
goal
                    assertz(person("Миша", 26, "", 8600), persons),
                    assertz(person("Вова", 27, "", 8600), persons),
                    assertz(person("Саша", 23, "", 9800), persons),
                    assertz(city(8600, "Мелитополь"), cities),
                    assertz(city(6800, "Киев"), cities).
```

Стандартные предикаты: assert/1, assert/2 работают также, как предикаты assertz/1, assertz/2.

Предикат asserta/1.

Назначение: Включает факт в начало динамической базы данных.
Обращение: asserta(<факт>).
Домены: (<соответствующие домены динамической базы данных>).
Шаблоны описания аргументов: (i).
Описание: asserta включает <факт> в динамическую базу данных перед всеми фактами для соответствующего предиката (т.е. в начало). Факт должен быть термом, принадлежащим к домену секции доменов динамической базы данных.
Неудачное завершение: Никогда не дает.

Ошибки: Отсутствуют.

Пример:

```
domains
   name, address, cityname = string
   age, zipcode = integer
database
   person(name, age, address, zipcode)
   city(zipcode, cityname)
goal
asserta(person("Миша", 26, "", 8600)), asserta(person("Вова", 27, "", 8600)),
asserta(person("Саша", 23, "", 9800)), asserta(city(8600, "Мелитополь")),
asserta(city(6800, "Киев")).
```

Предикат asserta/2.

Назначение: Включает факт в начало указанной динамической базы данных.

Обращение: asserta(<факт>,ИмяДБД).

Домены: (<соответствующие домены динамической базы данных>, ИмяБазыданных).

Шаблоны описания аргументов: (i,i).

Описание: asserta включает <факт> в динамическую базу данных перед всеми фактами для соответствующего предиката (т.е. в начало). Факт должен быть термом, принадлежащим к домену секции доменов динамической базы данных с именем ИмяДБД.

Неудачное завершение: Никогда не дает.

Ошибки: Отсутствуют.

Пример:

```
domains
   name, address, cityname = string
   age, zipcode = integer
database - persons          /* Имя = persons */
   person(name, age, address, zipcode)
database - cities           /* Имя = cities  */
   city(zipcode, cityname)
goal
                    asserta(person("Миша", 26, "", 8600), persons),
                    asserta(person("Вова", 27, "", 8600), persons),
                    asserta(person("Саша", 23, "", 9800), persons),
                    asserta(city(8600, "Мелитополь"), cities),
                    asserta(city(6800, "Киев"), cities).
```

Предикат consult/1.

Назначение: Считывает факты из текстового файла в неименованную динамическую базу данных.

Обращение: consult (ИмяФайлDOS).

Домены: (string).

Шаблоны описания аргументов: (i).

Описание: consult/1 считывает факты из текстового файла ИмяФайлDOS (созданного предикатом save) и включает их в конец неименованной динамической базы данных. Если файл содержит хотя бы одну синтаксическую ошибку, consult/ 1 выдает сообщение об ошибке. При написании пути по правилам DOS в местах,где пишется символ (\) нужно писать (\\). (a:\\tprolog\\prog\\mydb.dbf)

Неудачное завершение: Никогда не дает.

Ошибки: 1101 Предполагается целое
(во время считывания терма).
1102 Предполагается вещественное
(во время считывания терма).
1103 Предполагаются двойные кавычки
(во время считывания терма).
1104 Предполагаются кавычки
(во время считывания терма).
1105 Предполагается начало списка
(во время считывания терма).
1106 Предполагается конец списка

```
                    (во время считывания терма).
                    1107 Функтор отсутствует в домене
                    (во время считывания терма).
                    1108 Предполагается (
                    (во время считывания терма).
                    1109 Предполагается , или )
                    (во время считывания терма).
                    1010 Попытка открыть уже открытый файл.
                    1027 Невозможно открыть файл.
 Пример:
 domains
    ilist=integer*
 database
    p1(integer,char,real,string,symbol,ilist)
    p2(integer)
 goal
                        consult("dd.dba").
 /* -------- Содержание dd.dba -----------------
 p1(1,'a',44.44,"Turbo","Prolog",[1,2,3,4])
 p1(2,'b',-4.444E-98,"---","++++",[])
 p2(88) p2(99)
 ------------------------------------------------------------*/
```

Предикат consult/2.

```
  Назначение:   Считывает факты из текстового файла в указанную
                динамическую базу данных.
  Обращение:    consult(ИмяФайлDOS,ИмяДБД).
      Домены:   (string,<ИмяБазыДанных>).
Шаблоны описания аргументов: (i,i).
     Описание:  consult/2 считывает факты из текстового файла ИмяФайлDOS
(созданного предикатом save) и включает их в конец динамической базы дан-
ных с именем ИмяДБД.  Если файл содержит хотя бы одну синтаксическую
ошибку,consult/2 выдает сообщение об ошибке.
Неудачное завершение: Никогда не дает.
      Ошибки:   Те же,что и в consult/1.
 Пример:
 domains
    ilist=integer*
 database - facts1
    p1(integer,char,real,string,symbol,ilist)
 database - facts2
    p2(integer)
 goal
                        consult("facts1.dba",facts1),
                        consult("facts2.dba",facts2).
 /* -------- Содержание facts1.dba -----------------
 p1(1,'a',44.44,"Turbo","Prolog",[1,2,3,4])
 p1(2,'b',-4.444E-98,"---","++++",[])
  ------------------------------------------------------------*/

 /* -------- Содержание facts2.dba -----------------
 p2(88)
 p2(99)
 ------------------------------------------------------------*/
```

Предикат consulterror/3.

```
  Назначение:   Возвращает информацию об ошибке при выполнении
                предиката consult.
  Обращение:    consulterror(Стр,ПозСтр,ФайлПоз).
      Домены:   (string,integer,integer).
Шаблоны описания аргументов: (o,o,o).
     Описание:  consulterror возвращает строку Стр, в которой имеется синтак-
сическая ошибка; позицию в строке, в которой ошибка;позицию ошибки Файл-
Поз как номер байта от начала файла.
```

6.1.3. СОХРАНЕНИЕ ВНУТРЕННИХ БАЗ ДАННЫХ

Предикат save/1.

Назначение: Записывает содержимое неименованной базы данных в текстовый файл.
Обращение: save(ИмяФайлаДОС).
Домены: (string).
Шаблоны описания аргументов: (i).
Описание: Записывает все факты для предикатов базы данных в текстовый файл, который имеет имя ИмяФайлаДОС.Save записывает факт на отдельной строке в файле.Файл может быть после этого считанным в память с помощью предиката consult. Текст файла может также просматриваться и обрабатываться с использованием текстового редактора, однако недопустимы дополнительные пробелы, и все функторы должны состоять из строчных букв. save/1 записывает факты из секции database, которая была описана без имени.
Неудачное завершение: Никогда не дает.
Ошибки: 1027 Невозможно открыть файл.
1028 Невозможно записать файл.

Пример:

```
domains
   list = integer*
database
   fact1(integer,string,list)
   fact2(INTEGER,STRING)
clauses
                    fact1(1,"факт1",[1,2,3]).
                    fact1(2,"факт2",[1,3]).
                    fact1(3,"факт2",[3,2,1]).
                    fact2(1,"один").
                    fact2(1,"более одного").
                    fact2(2,"два").

  Goal: save("con")
  fact1(1,"факт1",[1,2,3])
  fact1(2,"факт2",[1,3])
  fact1(3,"факт2",[3,2,1])
  fact2(1,"один")
  fact2(1,"более одного")
  fact2(2,"два")
  Yes
```

Предикат save/2.

Назначение: Записывает содержимое указанной внутренней базы данных в текстовый файл.
Обращение: save(ИмяФайлаДОС,ИмяБД).
Домены: (string,<symbol>).
Шаблоны описания аргументов: (i,i).
Описание: save записывает содержимое секции базы данных с именем, определенным ИмяБД в файл ИмяФайлаДОС.
Неудачное завершение: Никогда не дает.
Ошибки: 1027 Невозможно открыть файл.
1028 Невозможно записать файл.

Пример:

```
domains
   list = integer*
database - dba1
   fact1(integer,string,list)
database - dba2
   fact2(integer,string)
clauses
                    fact1(1,"факт1",[1,2,3]).
                    fact1(2,"факт2",[1,3]).
                    fact1(3,"факт2",[3,2,1]).
```

```
                    fact2(1,"один").
                    fact2(1,"более одного").
                    fact2(2,"два").
/* Посмотрите что произойдет, если введете следующую цель:
save(con.dba).
Сохраненная база будет изображена в окне диалога */
  Goal: save("con",dba1)
  fact1(1,"факт1",[1,2,3])
  fact1(2,"факт2",[1,3])
  fact1(3,"факт2",[3,2,1])
  Yes
  Goal: save("dba1.dba",dba1)
  Yes
  Goal: save("dba2.dba",dba2)
  Yes
```

6.1.4. УДАЛЕНИЕ ФАКТОВ ИЗ ВНУТРЕННИХ БАЗ ДАННЫХ

Предикат retract/1.

Назначение: Удаляет факты из внутренней базы данных.
Обращение: retract(<Факт>).(nondeterm)
Домены: (dbasedom).
Шаблоны описания аргументов: (i).

Описание: retract удаляет первый <Факт> в базе данных, который сопоставится с данным <Факт>. Данный предикат недетерминирован, поэтому инициируя механизм возврата можно удалить все факты сопоставимые с заданным. <Факт> сопоставляется с фактами в базе данных, это значит, что любая свободная переменная будет связываться при обращении в retract.

Пример:

```
domains
   list = integer*
database
   fact1(integer,string,list)
   fact2(integer,string)
clauses
                    fact1(1,"факт1",[1,2,3]).
                    fact1(2,"факт2",[1,3]).
                    fact1(3,"факт2",[3,2,1]).
                    fact2(1,"один").
                    fact2(1,"более одного").
                    fact2(2,"два").

  Goal: fact1(X,Y,Z)
  X=1, Y=факт1, Z=[1,2,3]
  X=2, Y=факт2, Z=[1,3]
  X=3, Y=факт2, Z=[3,2,1]
  3 Solutions
  Goal: retract(fact1(X,Y,[_,2|Z]))
  X=1, Y=факт1, Z=[3]
  X=3, Y=факт2, Z=[1]
  2 Solutions
  Goal: retract(fact1(X,Y,Z))
  X=2, Y=факт2, Z=[1,3]
  1 Solution
  Goal: fact1(X,Y,Z)
  No Solution
  Goal: retract(fact2(1,X))
  X=один
  X=более одного
  2 Solutions
```

Предикат retract/2.

Назначение: Удаляет факт из указанной внутренней базы данных.
Обращение: retract(<Факт>,ИмяБД).

Домены: (dbasedom,symbol).
Шаблоны описания аргументов: (*,i)
Описание: С помощью retract/2 вы можете указать имя базы данных, из которой будут удалены факты. Однако основное преимущество retract/2 следующее: если первый аргумент является свободной переменной, все факты в указанной базе данных будут удалены.
Пример:

```
domains
   list = integer*
database - dba1
   fact1(integer,string,list)
database - dba2
   fact2(integer,string)
clauses
                    fact1(1,"факт1",[1,2,3]).
                    fact1(2,"факт2",[1,3]).
                    fact1(3,"факт2",[3,2,1]).
                    fact2(1,"один").
                    fact2(1,"более одного").
                    fact2(2,"два").

  Goal: retract(X,dba1)
  X=fact1(1,"факт1",[1,2,3])
  X=fact1(2,"факт2",[1,3])
  X=fact1(3,"факт2",[3,2,1])
  3 Solutions

  Goal: retract(fact2(1,X),dba2)
  X=один
  X=более одного
  2 Solutions
```

Предикат retractall/1.

Назначение: Удаляет все сопоставимые факты из внутренней базы данных.
Обращение: retractall(<Факт>).
Домены: (<dbasedom>).
Шаблоны описания аргументов: (*).
Описание: retractall удаляет все сопоставимые факты для предиката базы данных. Предикат всегда выполняется успешно, даже если в базе данных не было фактов для уничтожения.
Неудачное завершение: Никогда не дает.
Ошибки: Нет ошибок.
Пример:

```
  domains
    LIST = integer*
  database
    fact1(integer,string,list) fact2(integer,string)
  clauses
                    fact1(1,"факт1",[1,2,3]).
                    fact1(2,"факт2",[1,3]).
                    fact1(3,"факт2",[3,2,1]).
                    fact2(1,"один").
                    fact2(1,"более одного").
                    fact2(2,"два").
  Goal: fact1(X,Y,Z)
  X=1, Y=факт1, Z=[1,2,3]
  X=2, Y=факт2, Z=[1,3]
  X=3, Y=факт2, Z=[3,2,1]
  3 Solutions
  Goal: retractall(fact1(_,_,_))
  Yes
  Goal: fact1(X,Y,Z)
```

```
No Solution
Goal: retractall(fact1(_,_,_)) % retractall дает всегда успех
Yes
Goal: retractall(fact2(1,_))   % retract сравнивает все
Yes
```

Предикат retractall/2.

Назначение: Удаляет все сопоставимые факты из указанной внутренней базы данных.

Обращение: retractall(<Факт>,ИмяБД).

Домены: (<fact>,<symbol>).

Шаблоны описания аргументов: (*,i).

Описание: retractall удаляет все сопоставимые факты из указанной базы данных. В retractall не могут находиться свободные переменные. Необходимо, чтобы переменные при обращении были связанными или анонимными (знак подчеркивания). retractall является определенным и всегда будет согласован.

Неудачное завершение: Никогда не дает.

Ошибки: Нет ошибок.

Пример:

```
domains
   LIST = integer*
database - dba1
   fact1(integer,string,list)
database - dba2
   fact2(integer,string)
clauses
                    fact1(1,"факт1",[1,2,3]).
                    fact1(2,"факт2",[1,3]).
                    fact1(3,"факт2",[3,2,1]).
                    fact2(1,"один").
                    fact2(1,"более одного").
                    fact2(2,"два").

Goal: fact1(X,Y,A)
X=1, Y=факт1, A=[1,2,3]
X=2, Y=факт2, A=[1,3]
X=3, Y=факт2, A=[3,2,1]
3 Solutions
Goal: retractall(_,dba1)
Yes
Goal: fact1(X,Y,Z)
No Solution
```

6.2. РАБОТА С ФАЙЛАМИ

Предикаты: save-записи базы данных в файл, consult-чтения базы данных из файла, не обеспечивают гибкой работы с файлами. Поэтому в языке TURBO PROLOG предусмотрены и другие методы работы с файловыми устройствами ввода/вывода.

Все устройства ввода/вывода информации система считает файловыми и устанавливает с ними связь через их символьные имена.

Система TURBO PROLOG распознает 7 стандартных файловых устройств ввода/вывода информации:

com1	вывод через последовательный порт; keyboard ввод с клавиатуры (по умолчанию);
printer	вывод через параллельный порт принтера;
screen	вывод на экран монитора (по умолчанию);
stdin	ввод через стандартное устройство ввода DOS;
stdout	вывод на стандартное устройство вывода DOS;
stderr	вывод на стандартное устройство вывода сообщений об ошибках DOS.

Имена: stdin,stdout,stderr - используют для перенаправления ввода/вывода в командных строках DOS. Эти стандартные альтернативные имена не

должны встречаться в объявлении file; их нет необходимости открывать, и их не следует закрывать.

При работе с файлами на диске надо описать их символьные имена в разделе domains.

```
domains
   file=f1;f2;...;fn
```

Символьные имена должны начинаться со строчной буквы.В любой программе допускается только один домен file.

6.2.1. ОТКРЫТИЕ И ЗАКРЫТИЕ ФАЙЛОВ

Далее файл должен быть открыт. Открыть файл можно в одном из четырех режимов:

- для чтения;
- для записи;
- для добавления(запись);
- для модификации(чтение/запись).

Файл, открытый для каких-либо действий, отличных от чтения, должен быть после завершения этих действий закрыт. В противном случае изменения его могут быть потеряны. Можно открыть несколько файлов одновременно, и между открытыми файлами можно быстро перераспределять любые операции ввода/вывода. Переадресация данных между файлами осуществляется значительно быстрее, чем открытие и закрытие файла.

Предикат openread/2.

Назначение: Открывает файл для чтения.
Обращение: openread(СимволичИмяФайла,ИмяФайлаДос).
Домены: (file,string).
Шаблоны описания аргументов: (i,i).
Описание: openread открывает дисковый файл с именем ИмяФайлаДос для чтения. Он также связывает СимволичИмяФайла с открываемым файлом для будущих ссылок внутри программы, которая содержит обращение в openread.
Неудачное завершение: Никогда не дает.
Ошибки: 1010 Попытка открыть предварительно открытый файл.
1027 Невозможно открыть файл.

Предикат openwrite/2.

Назначение: Открывает файл для записи.
Обращение: openwrite(СимволичИмяФайла,ИмяФайлаДос).
Домены: (file,string).
Шаблоны описания аргументов: (i,i).
Описание: openwrite открывает дисковый файл с именем ИмяФайлаДос для записи.Он также связывает СимволичИмяФайла с открываемым файлом для будущих ссылок внутри программы,которая содержит обращение в openwrite. Если файл, названный СимволичИмяФайла, уже имеется на диске, openwrite уничтожит его содержимое.
Неудачное завершение: Никогда не дает.
Ошибки: 1010 Попытка открыть предварительно открытый файл.
1027 Невозможно открыть файл.

Предикат openappend/2.

Назначение: Открывает файл для добавления(записи в конец файла).
Обращение: openappend(СимволичИмяФайла,ИмяФайлаДос).
Домены: (file,string).
Шаблоны описания аргументов: (i,i).
Описание: openappend открывает дисковый файл с именем ИмяФайлаДос для добавления. Он также связывает СимволичИмяФайла с открываемым файлом для будущих ссылок внутри программы, которая содержит обращение к openappend.
Неудачное завершение: Такое же, как в openread.
Ошибки: Такие же,как в openread.

Предикат openmodify/2.

Назначение: Открывает файл для модификации.
Обращение: openmodify(СимволичИмяФайла,ИмяФайлаДос).

Домены: (file,string).
Шаблоны описания аргументов: (i,i).
Описание: openmodify открывает дисковый файл с именем ИмяФайлаДос для двух операций чтения и записи. Он также связывает СимволичИмяФайла с открываемым файлом для будущих ссылок внутри программы, которая содержит обращение в openmodify. Этот предикат может быть использован в сочетании с filepos для установления последовательного и произвольного доступа к файлу.
Неудачное завершение: Такое же, как в openread.
Ошибки: Такие же, как в openread.

Файлы на диске в языке TURBO PROLOG могут быть двух видов: двоичные и текстовые.В двоичном файле информация представлена непрерывно, в текстовом файле информация представлена в виде текстовых строк, каждый элемент это новая строка. В конце строки ставится признак конца строки и перехода на новую строку (коды ASCII 13,ASCII 10). В конце файла ставится признак конца файла.

Предикат filemode/2.

Назначение: Устанавливает или определяет вид файла.
Обращение: filemode(ИмяФайлСимв,РежФайл).
Домены: (file,integer).
Шаблоны описания аргументов: (i,i) (i,o).
Описание:

- РежФайл = 0 (Текстовый режим).
- РежФайл = 1 (Двоичный режим).

(i,i)
Устанавливает файлу ИмяФайлСимв режим, указанный в РежФайл.
(i,o)
Возвращает текущий режим файла ИмяФайлСимв в РежФайл.
Неудачное завершение: Никогда не дает.
Ошибки: 1018 Файл не открыт.
1019 Недопустимый режим файла, должен быть 0 или 1.

Предикат filepos/3.

Назначение: Перемещает указатель файла.
Обращение: filepos(ИмяФайлСимв,ПозФайл,Режим).
Домены: (file,real,integer).
Шаблоны описания аргументов: (i,i,i),(i,o,i).
Описание: (i,i,i)
filepos выбирает позицию в указанном файле, где будет иметь место следующая операция обработки файла. Позиция рассчитывается согласно значению Режим.

Режим	Позиция
0	Относительно начала файла
1	Относительно текущей позиции
2	Относительно конца файла

Например, если ПозФайл связан со значением 11 и Режим связан со значением 0, следующая операция будет в позиции 11 (относительно начала файла).
(i,o,i)
Возвращает текущую относительную позицию файла от начала файла. Режим игнорируется при этом Шаблоне описания аргументов.

При использовании filepos имейте ввиду, что операционная система при каждом нажатии клавиши ВВОД в текстовых файлах, генерирует два символа - ‘новая линия’ (ASCII 13) и ‘перевод строки’ (ASCII 10). При чтении строк из таких файлов эти символы автоматически пропускаются, но их необходимо учитывать при использовании filepos.
Неудачное завершение: Никогда не дает.
Ошибки: 1018 Файл не открыт.
1070 Невозможно выполнить предикат.
filepos вышел за пределы файла.
Итог: файл расширен за следующую запись.

Предикат flush/1.

Назначение: Записывает на диск файловый буфер.
Обращение: flush(ИмяФайлСимв).
Домены: (file).
Шаблоны описания аргументов: (i).
Описание: flush инициирует запись содержимого внутреннего файлового буфера системы TURBO PROLOG на текущее записывающее устройство. flush полезен, когда выход ориентирован на последовательный порт и может иметься необходимость посылать данные в порт до заполнения буфера.
Примечание: Из-за ограничений операционной системы flush не обеспечивает защиту файла в операционной системе. Обеспечить целостность файла можно путем закрытия его и последующего открытия вновь.
Неудачное завершение: Никогда не дает.
Ошибки: 2001 Не могу выполнить операцию записи.

Предикат eof/1.

Назначение: Контролирует признак конца файла.
Обращение: eof(ИмяФайлСимв).
Домены: (file).
Шаблоны описания аргументов: (i).
Описание: eof проверяет, не указывает ли указатель текущей позиции на конец файла. Если да, то eof выполняется успешно, иначе предикат дает неудачное завершение.
Неудачное завершение: Не конец файла.
Ошибки: Отсутствуют.
Пример:

```
domains
   file = input
predicates
   repfile(FILE)
clauses
                repfile(_).
                repfile(F):-not(eof(F)),repfile(F).
goal
                openread(input,"dd.txt"),
                readdevice(input),
                repfile(input),
                readln(L),write(L),nl,
                fail.
```

Предикат closefile/1.

Назначение: Закрывает файл.
Обращение: closefile(ИмяФайлСимв).
Домены: (file).
Шаблоны описания аргументов: (i).
Описание: closefile закрывает файл, открытый с именем ИмяФайлСимв. closefile выполняется успешно, даже если файл не был открыт.
Неудачное завершение: Никогда не дает.
Ошибки: Отсутствуют.

6.2.2. ЧТЕНИЕ И ЗАПИСЬ

Для чтения из файлов или записи в файлы, предварительно надо переназначить текущее устройство ввода/вывода (по умолчанию текущее устройство ввода-keyboard, вывода-screen) на файл.

Предикат readdevice/1.

Назначение: Устанавливает или определяет текущее устройство ввода.
Обращение: readdevice(СимволичИмяФайла).
Домены: (symbol).
Шаблоны описания аргументов: (i),(o).
Описание: (i).
Переназначает текущее устройство ввода на открытый файл с данным СимволичИмяФайла. Открываемый файл может быть один из стандартных файлов или любой файл пользователя с символическим именем, открытый для чтения или модификации.
(o).

Связывает СимволичИмяФайла с именем текущего устройства ввода.
Стандартные файлы, которые могут быть открыты для ввода:

com1	чтение из последовательного порта связи
keyboard	чтение с клавиатуры (по умолчанию)
stdin	чтение из стандартного ввода DOS

Неудачное завершение: Никогда не дает.

Ошибки: 1011 Попытка назначить устройство ввода в файл, находится ни в режиме ввода ни в режиме модификации.

Пример:

```
domains
   file = input
goal
                    openread(input,"dd.txt"),
                    readdevice(input),
                    readln(L1),write(L1),nl,
                    readln(L2),write(L2),nl,
                    readdevice(keyboard),
                    write("Введите текст: "),
                    readln(L3),write(L3),nl.
```

Предикат writedevice/1.

Назначение: Устанавливает или определяет текущее устройство вывода.
Обращение: writedevice(СимвИмяФайла).
Домены: (symbol).
Шаблоны описания аргументов: (i),(o).
Описание: (i).

Переназначает текущее устройство вывода в открытый файл с данным СимвИмяФайла. Открытый файл может быть одним из стандартных символических файлов или любым пользовательским файлом, открытым для записи или для модификации.

(o)

Связывает СимвИмяФайла с текущим устройством вывода. Встроенными файлами, которые могут быть использованы для вывода, являются:

com1	записывающий в последовательный порт связи
printer	записывающий в параллельный порт принтера
screen	записывающий в монитор экрана
stdout	записывающий в стандартный выход ДОС
stderr	записывающий в файл стандартных ошибок

Неудачное завершение: Никогда не дает.

Ошибки: 1012 Попытка переназначить устройство ввода в файл, который находится или в режиме записи или в режиме модификации.

Пример:

```
domains
   FILE = myfile
goal
                    openwrite(myfile,"dd.txt"),
                    writedevice(myfile),
                    write("Файл с одной линией\n"),
                    writedevice(screen),
                    write("\nСейчас пишем в экран"),
                    writedevice(myfile),
                    closefile(myfile),
                    writedevice(DEV),
write("\nЗаметим устройство записи автоматически установлено в : ",DEV),
write("\nкогда файл закрыт").
```

В зависимости от информации, чтение из файла можно осуществлять стандартными предикатами read...,из двоичного файла только предикатом readchar.

Для записи информации в файл можно использовать стандартные предикаты write...,nl,file_str.

Предикат file_str/2.

Назначение: Записывает из строки или считывает в строку дисковый

файл полностью.
Обращение: file_str(ИмяФайлDOS,ТекстСтр).
Домены: (string,string).
Шаблоны описания аргументов: (i,o),(i,i).
Описание: (i,o).
Весь файл ИмяФайлDOS (максимальный размер 64K) будет считан в строку ТекстСтр.
(i,i).
Будет создан новый текстовый файл ИмяФайлDOS, содержащий текст ТекстСтр.
Неудачное завершение: Никогда не дает.
Ошибки: 1026 Размер файла,загружаемого предикатом file_str, превышает 64K.
1027 Невозможно открыть файл.
1028 Запись в файл невозможна.

6.2.3. РАБОТА С ФАЙЛОВОЙ СИСТЕМОЙ

Здесь мы познакомимся с некоторыми предикатами, выполняющими действия аналогичные командам DOS для работы с файловой системой.

Предикат existfile/1.

Назначение: Проверяет, существует ли файл в текущем каталоге.
Обращение: existfile(ИмяФайлDOS).
Домены: (string).
Шаблоны описания аргументов: (i).
Описание: existfile дает удачный исход,если файл ИмяФайлDOS присутствует в текущем каталоге.
Неудачное завершение: Если файл не существует.
Ошибки: Отсутствуют.
Пример:

```
Goal: existfile(“dd.txt”)
Yes
Goal: deletefile(“dd.txt”)
Yes
Goal: existfile(“dd.txt”)
No
```

Предикат deletefile/1.

Назначение: Уничтожает файл.
Обращение: deletefile(ИмяФайлDOS).
Домены: (string).
Шаблоны описания аргументов: (i).
Описание: deletefile уничтожает файл ИмяФайлDOS на активном диске в текущем каталоге.
Неудачное завершение: Никогда не дает.
Ошибки: 1072 Невозможно уничтожить файл.
Пример:

```
Goal: file_str(“dd.txt”,X)
X=Файл из одной строки
1 Solution.
Goal: deletefile(“dd.txt”)
Yes
Goal: file_str(“dd.txt”,X)
0 Solution.
```

Предикат disk/1.

Назначение: Устанавливает или возвращает текущий дисковод и путь по правилам DOS.
Обращение: disk(МаршрDOS).
Домены: (string).
Шаблоны описания аргументов:(i),(o).
Описание: (i)
Устанавливает текущий дисковод и путь по правилам DOS МаршрDOS.
(o)
Возвращает текущий дисковод и путь по правилам DOS МаршрDOS.

Неудачное завершение: Никогда не дает.
Ошибки: 2009 Недопустимый путь по правилам DOS.
Пример:

```
goal
                    disk(Dosp),
                    write("Enter directory: "),
                    readln(PathChange),
                    DosP<>PathChange,
                    disk(PathChange),
                    disk(DosPath).
```

Предикат renamefile/2.

Назначение: Переименовывает файл.
Обращение: renamefile(СтароеИмФДос,НовоеИмФДос).
Домены: (string,string).
Шаблоны описания аргументов: (i,i).
Описание: renamefile изменяет СтароеИмФДос (на текущем диске в текущем каталоге) на НовоеИмФДос.
Неудачное завершение: Никогда не дает.
Ошибки: 1073 Невозможно переименовать файл.
Пример:

```
goal
                    file_str("dd.txt", "Где Владимир Петров ?"),
                    renamefile("dd.txt", "dd1.txt"),
                    file_str("dd1.txt", X),
                    deletefile("dd1.txt"),
                    write(X),nl.
```

Рассмотрим пример программы демонстрирующий, действие стандартных предикатов: работы с внутренними базами данных и файлами.

```
/* РАБОТА С ВНУТРЕННИМИ БАЗАМИ ДАННЫХ ФАЙЛ WORK_DB.PRO */
nowarnings
domains
   file=fil_d2
   list=integer*
/* ОПИСАНИЕ ДОМЕНОВ ВНУТРЕННИХ БАЗ ДАННЫХ */
database
   person(string,string,string,string,string)
database - d1
   number(integer)
database - d2
   number1(integer)
database - d3
   col(integer)
predicates
   write_screen_dba(symbol) /* - ВЫВОД БАЗ ДАННЫХ НА ЭКРАН*/
   run(integer)          /* - УПРАВЛЯЕТ РАБОТОЙ ПРОГРАММЫ*/
   select_demo(integer)    /* - ВЫБОР ДЕЙСТВИЙ НАД БАЗАМИ ДАННЫХ*/
   read_dba_assert(integer,integer,integer) /* -ЧТЕНИЕ И ЗАПИСЬ ДАННЫХ
                                              ИЗ,И  В БАЗЫ ДАННЫХ*/
   del_window(list)       /* УДАЛЕНИЕ ОКОН ВВОДА/ВЫВОДА */
goal
clearwindow, run(0),
removewindow,removewindow.

clauses
/* УПРАВЛЯЕТ РАБОТОЙ ПРОГРАММЫ */ run(N):- N<8,Num=N+1,
                                          select_demo(Num), run(Num);
                    makewindow(2,94,94,"",12,19,3,44,1,-1,"++++-¦ "),
write(" РАБОТА ОКОНЧЕНА . НАЖМИТЕ ЛЮБУЮ КЛАВИШУ"), readchar(_).
                /*  ВЫБОР ДЕЙСТВИЙ НАД БАЗАМИ ДАННЫХ*/
               /* ВВОД ДАННЫХ В d1 И ВЫВОД ЕЕ НА ЭКРАН*/
select_demo(1):-makewindow(1,63,63,"ЧТЕНИЕ ФАКТОВ В БАЗУ ДАННЫХ-d1
```

```
                С КЛАВИАТУРЫ ASSERT...",0,0,24,80,1,-1,"++++-¦ "),
            cursor(3,1), write("Действие assert/2,readterm/2"),
            cursor(5,1),write("ВВЕДИТЕ КОЛИЧЕСТВО ЭЛЕМЕНТОВ В Б.Д.-
d1:"),
            readint(End),assert(col(End),d3), cursor(5,1), write("\t\t\t\t\t\t\t\t"),
            cursor(5,1),write("Образец ввода :number(integer) "),nl,
            read_dba_assert(1,0,End),
            cursor(5,1),write("Содержимое базы данных d1 :     "),nl,
            write_screen_dba(d1),readchar(_),!.
/*ПЕРЕЗАПИСЬ ДАННЫХ ИЗ d1 В d2,УДАЛЕНИЕ d1 И ВЫВОД НА ЭКРАН d2*/
select_demo(2):-makewindow(2,62,62,"СОЗДАНИЕ Б.Д.-d2 , УНИЧТОЖЕНИЕ-d1 "
                ,1,40,24,40,1,-1,"++++-¦ "),
                cursor(3,5),write("Действие asserta/2,retract/2"),
                col(End),read_dba_assert(2,0,End),
                cursor(5,1),write("Содержимое базы данных d2 :     "),nl,
write_screen_dba(d2),readchar(_),!.
              /*ЗАПИСЬ d2 В ТЕКСТОВЫЙ ФАЙЛ ЦЕЛИКОМ */
                    select_demo(3):-save("number1.dba",d2),!.
/*ЧТЕНИЕ ДАННЫХ ИЗ ФАЙЛА ПО ОДНОМУ,ПЕРЕНОС ИХ В d1 ВЫВОД
НА ЭКРАН*/
select_demo(4):-makewindow(3,59,59,"ЗАПИСЬ И ЧТЕНИЕ ИЗ ФАЙЛА В Б.Д.- d1
"
                ,11,2,14,50,1,-1,"++++-¦ "),cursor(3,1),
                    write("Действие assertz/2,openread/2,readdevice/1"),
                    openread(fil_d2,"number1.dba"),readdevice(fil_d2),
                    col(End),read_dba_assert(3,0,End),
                    closefile(fil_d2),readdevice(keyboard),cursor(5,1),
                    write("Содержимое базы данных d1 :     "),nl,
                    write_screen_dba(d1),readchar(_),!.
/*ДОБАВЛЕНИЕ ДАННЫХ ИЗ d1 В ФАЙЛ ДЛЯ d2 ПО ОДНОМУ,УДАЛЕНИЕ d1*/
select_demo(5):-makewindow(4,58,58,"ЗАПИСЬ В ФАЙЛ Б.Д.- d1 "
                ,2,12,18,53,1,-1,"++++-¦ "),cursor(1,1),
                    write("Действие openappend/2,writedevice/1,retractall/1"),
                    openappend(fil_d2,"number1.dba"),writedevice(fil_d2),
                    col(End),read_dba_assert(4,0,End),closefile(fil_d2),
                    openread(fil_d2,"number1.dba"),readdevice(fil_d2),
                    read_dba_assert(3,0,End),closefile(fil_d2),
                    readdevice(keyboard),writedevice(screen), cursor(2,1),
                    write("Содержимое базы данных d1 :     "),nl,
                    write_screen_dba(d1), retractall(number(_),d1),
                    retractall(number1(_),d2), deletefile("number1.dba"),
                    readchar(_),!.
/*ПЕРЕНОС ДАННЫХ ИЗ ТЕКСТОВОГО ФАЙЛА В КОНЕЦ БД dbasedom*/
select_demo(6):-makewindow(5,30,30,"ЧТЕНИЕ ИЗ ФАЙЛА Б.Д.- dbasedom "
                ,3,5,20,65,1,-1,"++++-¦ "),
                    cursor(1,14),write("Действие existfile/1, consult/1"),
                    cursor(2,16),col(End), read_dba_assert(5,0,End),
                    cursor(4,12), write("Содержимое базы данных dbasedom :
                    "), cursor(6,0),
                    write_screen_dba(dbasedom),retractall(person(_,_,_,_,_)),
                    retractall(col(_),d3),readchar(_),!.
/* УДАЛЕНИЕ ОКОН ВВОДА/ВЫВОДА */
                    select_demo(7):-del_window([5,4,3,2]),!.
                    del_window([]).
                    del_window([H|T]):-removewindow,readchar(_),
                                       del_window(T).
              /* ВВОД ДАННЫХ В Б.Д.-d1  С КЛАВИАТУРЫ */
read_dba_assert(1,N,End):- N<End, cursor(6,3), readterm(d1,Term),

                    cursor(6,3),write("\t\t\t\t\t"),assert(Term,d1),Num=N+1,
                          read_dba_assert(1,Num,End);true.
```

```
                /*ПЕРЕНОС ДАННЫХ ИЗ d1 в d2 И УДАЛЕНИЕ d1*/
read_dba_assert(2,N,End):-
N<End,number(Number),asserta(number1(Number),d2),
                              retract(number(Number),d1),Num=N+1,
                              read_dba_assert(2,Num,End);!.
/* ЧТЕНИЕ ДАННЫХ ИЗ ФАЙЛА ПО ОДНОМУ И ПЕРЕНОС ИХ В d1 */
read_dba_assert(3,0,End):-not(eof(fil_d2)), readterm(d2,number1(Num)),
              assertz(number(Num),d1),read_dba_assert(3,0,End);!,true.
/* ДОБАВЛЕНИЕ ДАННЫХ ИЗ d1 В ФАЙЛ ДЛЯ d2 ПО ОДНОМУ,УДАЛЕНИЕ d1*/
read_dba_assert(4,N,End):- N<End,number(Number),
                              write("number1(",Number,")"),nl,
                              retract(number(Number),d1),
                              Num=N+1, read_dba_assert(4,Num,End);!.
/*ПЕРЕНОС ДАННЫХ ИЗ ТЕКСТОВОГО ФАЙЛА В КОНЕЦ БД dbasedom*/
read_dba_assert(5,0,_):-existfile("mycons.tlf"), consult("mycons.tlf"),
                        write("ФАЙЛ УСПЕШНО ПРОЧИТАН."),nl;
                        write("ФАЙЛ НЕ ПРОЧИТАН."),nl.
/*ВЫВОД НА ЭКРАН БД - dbasedom*/
write_screen_dba(dbasedom):-person(Name,Io,Adr,Tlfh,Tlfw),
          write("   ",Name," ",Io," ",Adr," ",Tlfh," ",Tlfw),nl,fail;true.
/*ВЫВОД НА ЭКРАН БД - d1*/
write_screen_dba(d1):-number(Number),write("   ",Number), nl,fail;true.
/*ВЫВОД НА ЭКРАН БД - d2*/
write_screen_dba(d2):-number1(Number),write("   ",Number), nl,fail;true.
```

ЗАДАНИЕ К ТЕМЕ :

Создать интерактивную среду-телефонный справочник, которая бы позволяла:

- 1) Создать базу данных-абонентов (на диске).
- 2) Редактировать базу данных (просматривать,дополнять,удалять).
- 3) Осуществить поиск абонента:
  - а) по фамилии,
  - б) по фамилии, имени, отчеству.

В данном пособии рассмотрены основы языка программирования TURBO PROLOG, для более широкого его изучения можно использовать:

"Руководство пользователя", "Справочник стандартных предикатов", фирменные примеры программ поставляемые вместе со средой.

7. СПИСОК ЛИТЕРАТУРЫ:

"TURBO PROLOG 2.0 - руководство пользователя."
BORLAND INTERNATIONAL, Scott Valley, 1988.

"TURBO PROLOG 2.0 - стандартные предикаты."
BORLAND INTERNATIONAL,Scott Valley,1988.

"Программирование на языке Пролог для искусственного интеллекта."
И.Братко,Москва "Мир",1990.

"Язык программирования Пролог."
Дж.Стобо,Москва "Радио и связь",1993.

"Пролог - язык программирования будущего."
Дж.Доорс,А.Р.Рейблейн,С.Вадера, Москва "Финансы и статистика",1990.

"Программирование экспертных систем на Турбо Прологе."
Д.Марселлус,Москва "Финансы и статистика",1994.

"Турбо Пролог в сжатом изложении."
А.Янсон,Москва "Мир",1991.

Для записей

Для записей