Министерство сельского хозяйства Российской Федерации

Федеральное государственное бюджетное
образовательное учреждение
высшего образования
«Пермская государственная сельскохозяйственная академия
имени академика Д.Н. Прянишникова»

А.Ю. БЕЛЯКОВ

Практика логического программирования на языке Пролог

Учебное пособие

Пермь
ИПЦ «Прокрость»
2017

УДК 004.43
ББК 32.973-018.1

Рецензенты:
Харитонов В.А. – заведующий кафедрой Строительного инжиниринга и материаловедения ПНИПУ, Заслуженный работник высшей школы РФ, академик РАЕН и МАИ, профессор, доктор технических наук.
Суворов А.В. –

__-____ Беляков А.Ю.
Практика логического программирования на языке Пролог: Учебно пособие / А.Ю. Беляков М-во с.-х. РФ, федеральное гос. бюджетное образов. учреждение высшего образов. «Пермская гос. с.-х. акад. им. акад. Д.Н. Прянишникова». – Пермь : ИПЦ «Прокростъ», 2017. – 129 с.
ISBN _____

В пособии в доступной форме изложены основы логического программирования, объяснены базовые механизмы языка и проанализированы некоторые приемы программирования на практических примерах. Пособие ориентировано на самостоятельное освоение материала с исследованием программ в среде программирования PDC Prolog. Подробно рассмотрены основные теоретические понятия логического программирования: декларативные предложения, механизмы унификации и бэктрекинга. Отдельное внимание уделено исследованию возможностей Пролога по организации итеративной обработки данных, созданию и практическому использованию динамических баз данных и экспертных систем.

Пособие предназначено для студентов, обучающихся по специальности «Информационные системы и технологии». Отдельные главы пособия могут быть полезны для аспирантов, магистров и преподавателей вузов, занимающихся исследованиями проблем создания искусственного интеллекта и разработки экспертных систем.

УДК 004.43
ББК 32.973-018.1

Утверждено в качестве учебного пособия на заседании Методического совета Пермской государственной сельскохозяйственной академии имени академика Д.Н. Прянишникова (протокол № __ от __.__.2017 г.).

ISBN _____

Содержание

Введение

Когда я начинал свое знакомство с Прологом, то самое шокирующее впечатление состояло в осознании одного, казалось бы, незначительного факта: в системе Пролог заметно меньше конструкций самого языка, операторов, или как принято говорить в логическом программировании – стандартных предикатов, чем в традиционных языках программирования. В частности, полностью отсутствуют такие привычные конструкции как оператор выбора и операторы цикла. Приступая к написанию программ на процедурных (императивных) языках (к примеру, BASIC или Pascal) всегда опираешься на какой-то базис из огромного количества команд, заготовленных разработчиками «на все случаи жизни». Откройте, к примеру, какую-нибудь книгу, типа «Delphi в подлиннике» или «VBA для профессионалов»… Их не то что открывать страшно, но и в закрытом состоянии для непосвященного они производят тяжелое впечатление, как в прямом, так и в переносном смысле. Проблема первоначального освоения процедурных языков заключается скорее в изучении значительного по объему синтаксиса языка и правил применения конкретных команд при переводе алгоритма решения задачи в программный код. Совсем иная проблема возникает при изучении Пролога...

Здесь пришло время напомнить несколько принципиальных моментов касающихся истории происхождения парадигмы логического программирования и различий между декларативными и императивными языками программирования.

Итак, основная концепция традиционных и повсеместно используемых императивных языков программирования на-

прямую связана с «фон-неймановской» моделью архитектуры компьютера. Согласно идее американского математика Джона фон Неймана все данные и программы компьютера хранятся в одной памяти. Процессор получает из памяти очередную команду, декодирует её, выбирает из памяти указанные в качестве операндов данные, выполняет команду и размещает результат снова в памяти. Программа на императивном языке представляет собой последовательность команд (алгоритм действий). Команды программы выполняются последовательно: сверху-вниз и слева-направо. Встречаются команды, которые могут изменять последовательность выполнения команд. Вообще, императив (происходит от лат. imperativus – повелительный, лат. imperator – повелитель) – приказ, требование. К наиболее известным и распространенным императивным языкам программирования относятся: ALGOL, FORTRAN, PL/1, BASIC, Pascal, C, C++, Java.

Декларативные (declaration – объявление, заявление) языки – это языки программирования, в которых операторы представляют собой высказывания об отношениях между объектами некоторого предметного поля. Высказывания могут быть двух видов: 1) факт – задает отношение между конкретными объектами предметного поля; 2) правило – опосредованно задает отношение между классами объектов, заменяя их переменными и требуя выполнения некоторых условий. Таким образом, программа на декларативном языке, по сути, представляет собой совокупность фактов и правил, а работа с программой заключается в формулировке вопроса, ответ на который ищется самой системой программирования в процессе перебора с подстановками значений на множестве фактов и правил. Программисту нет необходимости разрабатывать алгоритм перебора этим занимается сама система про-

граммирования. Последовательность выполнения операторов программы детерминирована совокупностью фактов, правил и вопросов, сформулированных программистом.

Наиболее известным языком декларативного программирования является Пролог. Язык программирования Пролог базируется на ограниченном наборе механизмов, включающих в себя сопоставление образцов (унификация), автоматический перебор (поиск) с возвратами (бэктрекинг), а также рекурсию (как способ итеративной обработки данных). Этот небольшой набор в руках опытного программиста образует удивительно мощный и гибкий инструмент *логического программирования*. Пролог находит свое воплощение при решении задач искусственного интеллекта, а также любых задач, требующих нечислового программирования.

Само название *Пролог* есть сокращение, означающее *программирование в терминах логики (Prolog – programming in logic)*. Идея использовать логику предикатов в качестве языка программирования возникла впервые в начале 70-х годов. 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). Сегодняшней своей популярности Пролог во многом обязан эффективной реализации этого языка, полученной в Эдинбурге Дэвидом Уорреном в середине 70-х годов.

Мы остановимся на наиболее известной у нас в стране и довольно эффективной версии Пролога – Турбо Прологе. Его начинала разрабатывать фирма Borland International в содружестве с датской компанией Prolog Development Center (PDC). Первая версия вышла в 1986 году. Последняя совме-

стная версия имела номер 2.0 и была выпущена в 1988 году. В 1990 году PDC получила монопольное право на Турбо Пролог и дальше продвигала его под названием PDC Prolog. С 1996 года компания Prolog Development Center стала выпускать Visual Prolog версии 4.0 и выше. Основы логического программирования мы будем изучать с использованием PDC Prolog 3.21.

Так вот, возвращаемся к размышлениям о сути логического программирования и трудностях, которые вам предстоит преодолеть на пути к самостоятельному написанию своей первой логической программы, решающей какую-нибудь нетривиальную задачу.

Приступая к созданию программы на императивном языке, вы обдумываете алгоритм решения задачи, как последовательность отдельных операций. Далее пишите код, базируясь на значительном количестве команд языка. Если что-то забылось, то смотрите в справочник. Потом тестируете, тестируете, тестируете… и, наконец, хвалите себя и растете в своих глазах.

Сравнивая этот процесс с написанием логической программы можно отметить сходство только на заключительном этапе, что, вообще-то, получается только через несколько месяцев, реже недель, от начала освоения программирования в логике. Дело в том, что в логической программе напрочь отсутствует алгоритм, в традиционном понимании этого термина. И тут случается сшибка. «Традиционный» программист, привыкший за несколько лет мыслить шаблонно (используя чужие заготовки или свои наработки алгоритмов), сталкивается с проблемой отсутствия опыта решения задач в логике предикатов при значительном опыте программирования вообще. Косность мышления не позволяет оторваться от

привычного порядка перевода задачи с человеческого языка на машинный, уйти от алгоритмического описания порядка решения задачи к декларативному описанию предметного поля.

Пора обсудить пример. Если вас попросят написать программу поиска минимума среди некоторого списка (одномерного массива) значений, то ваши размышления могут привести к следующему результату:

```
Min = Cells(1, 1)
For i = 2 To 10
    If Cells(i, 1) < Min Then Min = Cells(i, 1)
Next
Cells(11, 1) = Min
```

В данном случае вы видите VBA код, выполненный в Excel`e. Все значения располагаются в первом столбце в диапазоне строк от 1 до 10, ответ заносится в ячейку Cells(11, 1). Суть размышлений такова: возьмем первое значение из списка и назначим его текущим минимальным; далее будем сравнивать текущее минимальное значение последовательно со всеми значениями из списка и если какое-то из них будет меньше текущего минимального, то назначим уже его текущим минимальным; так будем продолжать до тех пор, пока список не кончится; когда список кончится, то текущее минимальное назовем просто минимальным значением списка и зафиксируем его.

Мы последовательно и в деталях объяснили машине как решать задачу.

Давайте сравним этот подход с реализацией на Прологе:

```
min([H|T],MT):-min(T,MT),MT<=H,!.
min([H|_],H).
```

Очевидно, что тут представлены два предложения. В переводе на русско-алгоритмический это будет звучать примерно так:

1) если минимум, найденный в списке без первого элемента, меньше первого элемента исходного списка, то результатом будет минимум из конца списка,
2) иначе, первый элемент списка и есть минимальный элемент списка.

По существу мы не рассказали машине как решать задачу, мы лишь передали как мы понимаем что есть минимум списка. Иначе говоря, мы описали отношение между объектами предметного поля. В этом описании (декларации) указаны признаки минимума, отличающие его от других значений (максимум, среднее арифметическое и т.п.). Кроме того, обратите внимание на то, что в лексике Пролога нет команды min. Вместо min можно написать всё, что угодно. К примеру, nim или mother и программа будет работать с таким же результатом. Программист волен давать предикатам любые имена, ограничиваясь лишь удобством последующего чтения и понимания существа программы. Читатель, имеющий некоторый опыт программирования на других языках сразу заметил также, что в основе поиска ответа лежит рекурсия. Дело в том что в программе на прологе не только отсутствует алгоритм, как я выше уже отметил, но и вообще отсутствуют привычные команды ветвления и цикла. В данном примере рекурсивный подход обеспечивает итерации, а два предложения – ветвление. Вот таким необычным и непривычным способом реализуется решение задачи на языке логического программирования. Саму же программу в Прологе корректнее называть экспертной системой, базой знаний или базой данных, в зависимости от наполненности структурами.

Почему стоит изучить логическое программирование?

Тезис первый (совсем не обязательный) – просто для расширения кругозора. Если вы считаете себя специалистом

в информационных технологиях или собираетесь таковым стать, то такой серьезный пробел в области практического приложения математической логики как непонимание логических программ может сильно сказаться на вашей самооценке в дальнейшем.

Тезис второй – эстетическое наслаждение. Действительно, если вы не просто существуете в мире информационных технологий, а отчасти и разрабатываете их, то понимание основных механизмов порождения решений логическими программами может доставить вам истинное наслаждение, а конкретные реализации некоторых программ будут бросать вас от недоумения до восторженного восприятия.

Тезис третий (непременное условие) – интеллектуальное развитие. Мое субъективное мнение заключается в том, что человек, сумевший понять методологию написания логических программ, а, тем более, осиливший её настолько, что может на практике применять свои познания, имеет право считать себя интеллектуально продвинутым. Тут дело не в пустом бахвальстве, а в том, что «декларативный» программист имеет в своем багаже дополнительную модель представления мира и явно обладает факультативным инструментом, расширяющим возможности своего мыслительного аппарата.

Тезис четвертый (вполне вероятный) – повышение качества процедурных программ. Любой опыт развивает, а опыт деятельности в альтернативной парадигме позволяет взглянуть на обыденные вещи в несколько ином ракурсе. Возвращаясь из мира логических программ к традиционным языкам программирования, проявляется способность по-новому, четче и более структурировано конструировать код. Иногда по-

рождаются решения, до которых ранее не удавалось дойти, докопаться.

Пролог освоить можно. Сложно, но «дорогу осилит идущий». Поэтому приступим, наконец. Как всегда начнем с «бытовухи» и постараемся её поскорее проскочить.

Глава 1. ПРОГРАММИРОВАНИЕ БЕЗ АЛГОРИТМА.

1.1. Механизм вывода решений.

Решение поставленной задачи на языке логического программирования сводится к формализованному описанию предметной области и контекстной формулировке запросов (вопросов). В ответ на запрос механизм вывода Пролога перебирает возможные варианты и формулирует ответ. Если в запросе присутствовали переменные, то Пролог ищет возможные значения обозначенных переменных, при которых истинность запроса подтверждается, и выдает их значения в качестве ответа. Если же в запросе отсутствуют переменные, не имеющие конкретного значения, то Пролог также делает перебор вариантов решения для предложения, обозначенного в запросе, но в качестве ответа только подтверждает или опровергает предложение запроса (выдает значение ИСТИНА/ЛОЖЬ).

Описание предметной области представляет собой логическую модель, состоящую из фактов и правил. Факт описывает наличие объектов в предметном поле, их свойства, а также отношения между конкретными объектами предметного поля. Правило задает отношения между классами объектов, фактами и самими правилами.

Любая программа на языке Пролог представляет собой совокупность предложений. Каждое предложение заканчивается точкой. Можно выделить три вида предложений Пролога: факты, правила и вопросы.

Предложение вида факт в программе представлено утверждением с использованием конкретных значений (констант).

Например, факт

```
mother("Наташа","Саша").
```

может быть интерпретирован так – Наташа является мамой Саши, то есть этот факт задает отношение между объектами предметного поля. Обратите внимание, что аргументы факта не могут быть переменными, они могут быть только константами.

Формально факт может также задавать принадлежность объекта предметного поля определенному классу, что также является отношением на множестве объектов предметного поля. Например, факт

```
women("Саша").
```

указывает на принадлежность Саши классу женщины.

Факт может задавать отношение между атрибутами, например факт:

```
age("Саша",20).
```

содержит два аргумента “Саша” (строковый) и 20 (целочисленный), что задает возраст Саши.

Предложение вида правило является более сложной конструкцией, чем факт. Правило – это предложение, истинность которого зависит от истинности одного или нескольких предложений.

В простом варианте предложение вида «правило» в программе представлено следующей конструкцией:

```
A :- B.
```

Здесь высказывание A истинно, если (логическая связка «если» обозначается знаком «:–») истинное высказывание B. Высказывание A является «головой» предложения, а B – телом (хвостовой частью).

Если в программе есть необходимость задать более сложную зависимость высказывания A от нескольких других

высказываний, то можно использовать следующую конструкцию:

$$A :- B_1, \ldots , B_n.$$

Здесь высказывание A истинно, если истинны все высказывания от B1 до Bn (логическая связка «и» обозначается запятой). Предполагается, что цель A достижима, если достижимы все входящие в тело предложения подцели B.

Обычно правило содержит несколько хвостовых целей B, которые должны быть истинными для того, чтобы правило было истинным.

Когда необходимо задать логическую связь типа «или», тогда можно в программе сконструировать процедуру (состоит из двух и более предложений с одинаковым высказыванием A, для которого проверяется истинность). Например, когда необходимо обозначить, что высказывание A истинно, если истинно B1 или B2, то можно использовать следующую конструкцию:

$$A :- B_1.$$
$$A :- B_2.$$

Высказывания в предложении в качестве аргументов могут содержать как константы, так и переменные. Областью действия переменной в Прологе является только одно предложение. Поэтому в разных предложениях может использоваться одно имя переменной для обозначения разных объектов. Глобальных переменных в Прологе не бывает.

Например, правило:

```
full_age_daughter ( M , D ):-
          mother( M , D ),
          women( D ),
          age( D , Age ).
          Age >= 18.
```

позволяет найти всех совершеннолетних дочерей у определенной матери M. В данном предложении идентификаторы M,

D и Age обозначают переменные, которые могут принимать разные значения во время выполнения программы. Синтаксис языка Пролог требует, чтобы переменные начинались с заглавной буквы, а константы, описанные в интерфейсной части программы (в разделе constants) как в традиционных языках программирования, задаются прописными символами.

И, наконец, вопрос как вид предложения в Прологе представляет собой тело предложения без головы, то есть может состоять из одной или нескольких перечислимых через запятую целей. В вопросе могут использоваться как константы так и переменные. Например, вопрос может выглядеть следующим образом:

```
mother( "Наташа" , D ) , women( D ) .
```

Такой вопрос предполагает поиск такого значения D, при котором истинна цель, состоящая из двух подцелей mother и women. В данном случае вопрос может быть интерпретирован как поиск таких детей у Наташи, которые имеют женский пол, то есть попросту поиск всех дочерей Наташи. Результатом поиска будет перечисление значений D.

Таким образом, можно сделать вывод, что если предложение содержит только голову с константами, то это факт, если предложение содержит и голову и тело, то это правило, если же предложение состоит только из тела, то это вопрос.

Теперь уже имеет смысл определить понятие ***предикат*** (лат. praedicatum — заявленное, упомянутое, сказанное). Оно в каком-то смысле эквивалентно понятию «команда» для традиционных языков программирования. Ранее мы уже использовали такие предикаты как mother, women, age и другие. Каждый из этих предикатов не является встроенным (стандартным) предикатом Пролога, а определен программистом для описания объектов предметного поля и отношений между ними. Например, предикат women является одноаргумент-

ным и задает следующее отношение – упомянутый аргумент является женщиной; предикат mother является двухаргументным и задает следующее отношение – объект предметного поля, упомянутый в качестве первого аргумента, является по отношении ко второму мамой. Предикат может и не иметь аргументов. Идентификатор mother сам по себе не является предикатом. Дело в том, что Пролог идентифицирует предикаты не только по имени (mother, women и т.п.), но и по количеству аргументов. В базе данных могут встречаться предикаты с одинаковым идентификатором, но с разным количеством аргументов. Такие предикаты воспринимаются как совершенно разные, как если бы у них были разные идентификаторы. Например, в одной программе может встречаться двухаргументный предикат mother(string,string) (в скобках указаны количество аргументов и их тип), задающий отношение «мать-дочь» и, наряду с ним, одноаргументный mother(string), относящий некий объект к классу «мать». Таким образом, ***предикат*** – конструкция логической программы, задающая отношение между объектами предметного поля, описываемого программой.

Подведем некоторые итоги. Программа на языке логического программирования не является алгоритмом, переведенным на язык программирования, ведь под алгоритмом, обычно, подразумевается некоторая последовательность действий. ***Программа на языке Пролог*** состоит из предложений (деклараций), которые описывают модель предметной области знаний в виде совокупности отношений между объектами предметного поля. Если в программе присутствуют только факты, то её можно назвать ***базой данных***. Если в программе присутствуют факты и правила (в редких случаях только правила), то её можно назвать ***базой знаний***. Если к про-

грамме добавлен интерфейс пользователя с совокупностью вопросов (запросов) к базе знаний, то такую программу можно назвать ***экспертной системой***.

Встает очевидный вопрос: если программа на языке Пролог не основана на алгоритме и не содержит последовательности инструкций, то, как же она работает и выдает результаты? Основная особенность Пролога состоит в том, что алгоритм анализа базы знаний, продекларированной программистом, встроен в компилятор языка и программисту нет нужды заниматься разработкой собственного алгоритма поиска решений. После декларации фактов и правил работы с ними остается только корректно формулировать запросы к базе знаний, а интегрированный в компилятор Пролога алгоритм сам обеспечит перебор возможных вариантов решения. Этот алгоритм обычно называют ***механизмом вывода решений*** и состоит он из двух базовых механизмов: ***унификации*** и ***бэктрекинга***. Следует также отметить, что ввиду отсутствия в логическом программировании таких структур как циклы возникает сложность с итеративной обработкой данных. Для преодоления данной проблемы наиболее приемлемым подходом следует считать ***рекурсивный способ декларации правил*** описания предметного поля. В настоящей главе рассмотрим указанные особенности логического программирования на несложных примерах.

1.2. Унификация.

Унификация – это процесс сопоставления вопроса с фактами и правилами базы знаний.

Разберем простейший пример №1. Пусть задан предикат t с двумя аргументами. К примеру, первый аргумент есть порядковый номер какого-либо объекта, а второй – его величина. Для примера возьмем случай, когда в базе есть только один факт t(1,2):

```
DOMAINS
    i = integer
PREDICATES
    t(i,i)
CLAUSES
    t(1,2).
```

Тут следует сделать некоторое отступление к описанию синтаксиса Пролога, а именно, к описанию разделов программы. Пока обратим внимание только на три раздела – DOMAINS, PREDICATES и CLAUSES. Раздел DOMAINS (области) используется для переопределения типов данных, чтобы, к примеру, не писать при описании предиката ключевое слово integer, его можно заменить на более короткое i. Раздел PREDICATES предназначен для описания предикатов – задаются идентификаторы предикатов, количество аргументов и их тип. Раздел CLAUSES (пункты, предложения) предназначен для декларации предложений – фактов и правил. Разделы не являются обязательными и включаются в программу по мере необходимости. В частности раздел DOMAINS может быть упразднен, в таком случае программу можно будет переписать следующим образом:

```
PREDICATES
    t(integer,integer)
CLAUSES
```

```
t(1,2).
```

Кроме того, в раздел PREDICATES попадают только предикаты не являющиеся стандартными, то есть те, которые отсутствуют в самом языке программирования. На данном этапе можно ограничиться рассмотрением программ без использования среды Пролог, а только лишь исследуя логику работы на описываемых примерах. В следующей главе более детально рассмотрим процесс запуска программ в среде Пролог.

Один из вариантов обращения к программе, которым мы пока и ограничимся, это формулирование вопроса, внешнего по отношению к программе, то есть такого, которого нет непосредственно в теле программы. В дальнейшем будет показано, что такой вопрос может быть задан только из среды программирования в окне диалога Пролога. В данной главе достаточно ограничиться теоретическим рассмотрением исследуемых процессов без запуска среды программирования. Напомню, что вопрос это предложение без головы. К примеру, зададим в окне диалога следующий вопрос к этой программе:

```
t(X,Y).
```

В ответ получим результат в том же окне: X=1, Y=2.

Такое решение было получено в результате унификации. Процесс унификации проходил следующим образом.

Для Пролога вопрос есть цель, которую необходимо достичь. Пролог берет вопрос t(X,Y) и начинает последовательно сверху-вниз сравнивать его с фактами и правилами базы знаний. Там где обнаруживается предикат с таким же идентификатором как и у вопроса и с таким же количеством аргументов происходит сопоставление. В данном случае в базе знаний есть только один факт и нет никаких правил. Если в вопросе аргументы ничем не означены (то есть X и Y

свободные переменные), то значения из найденного факта присваиваиваются соответствующим переменным.

Зададим другой вопрос: t(1,Y). Получим ответ: Y=2.

Процесс унификации проходил следующим образом. Пролог взял вопрос t(1,Y) и последовательно сверху-вниз сравнивая его с фактами из базы знаний обнаружил, что существует предикат с таким же идентификатором как и у вопроса и с таким же количеством аргументов. Так как в вопросе первый аргумент имеет конкретное значение, то Пролог проводит сопоставление сравнивая значение из вопроса и из факта. В данном случае они совпали, поэтому процесс сопоставления переходит на вторые аргументы. В вопросе второй аргумент ничем не означен, поэтому ему передается значение из факта. О чем, собственно, Пролог и сообщает.

Разберем так же простой пример №2. Пусть задан предикат t с двумя аргументами. Он выполняется, если выполнимы две цели, указанные в его теле. Первая цель X=1, а вторая цель 2=Y. Запятая между ними означает, что предикат выполнится только при условии выполнения обеих целей в его теле, то есть запятая имеет смысл "И".

```
DOMAINS
     i = integer
PREDICATES
     t(i,i)
CLAUSES
     t(X,Y):-
          X=1,
          2=Y.
```

Зададим вопрос этой программе: t(X,Y).

Получим ответ: X=1, Y=2.

Процесс унификации проходил следующим образом.

По аналогии с описанным выше, Пролог берет вопрос t(X,Y) и начинает последовательно сверху-вниз сравнивать

его с фактами и правилами базы знаний. В данной программе есть только одно правило. Наткнувшись на него Пролог уясняет, что идентификатор правила и количество его аргументов соответствуют вопросу. После чего Пролог пытается выяснить выполнимо ли это правило при условиях, указанных в вопросе. В вопросе все переменные свободны, а правило выполнимо при выполнимости двух целей из его тела: X=1 и 2=Y.

В Прологе знак "=" не есть знак присвоения. Его действие зависит от контекста. Только если с одной стороны от знака равенства стоит свободная переменная (то есть не имеющая никакого значения), а с другой стороны расположено какое-либо значение или означенная переменная (имеющая конкретное значение), тогда действие знака "=" можно считать эквивалентным присвоению. По итогам его выполнения свободная переменная получит значение с противоположной стороны от знака равенства (сторона не имеет значение) и цель признается выполненной.

В данном случае обе переменные свободны, поэтому каждая из целей в теле правила оказывается достижимой с выполнением присвоения соответствующих значений. Обратите внимание, что здесь не имеет значения ни последовательность целей в предложении ни место расположения относительно знака равенства переменной и ее будущего значения.

Зададим программе другой вопрос: t(X,2).

Получим ответ: X=1.

Процесс унификации проходил следующим образом.

Первая переменная, как и ранее, приобретает значение "1" – цель №1 достижима. Переходим ко второй цели: Y=2. Здесь перед выполнением цели, переменная уже не была сво-

бодной, а имела значение "2". Когда с обеих сторон от знака равенства находятся конкретные значения, тогда знак "=" имеет смысл сравнения. При равенстве обеих сторон цель признается достижимой. Именно так и произошло в данном случае.

Зададим другой вопрос: t(X,3).

Получим ответ: No Solution.

Процесс унификации проходил следующим образом.

Первая переменная, как и ранее, приобретает значение "1" – цель №1 достижима. Переходим ко второй цели: Y=2. Здесь перед выполнением цели, переменная уже не была свободной, а имела значение "3". Когда с обеих сторон от знака равенства находятся конкретные значения, тогда знак "=" имеет смысл сравнения. Так как значения сторон отличаются, то цель признается не достижимой.

Но в базе знаний может оказаться несколько фактов или правил. Разберем процесс унификации для таких случаев.

```
DOMAINS
    i = integer
PREDICATES
    t(i,i)
CLAUSES
    t(1,3).
    t(X,Y):-
        X=1,
        2=Y.
```

Зададим вопрос этой программе: t(X,3).

Получим ответ: X=1.

1 Solution.

Процесс унификации проходил следующим образом.

Пролог последовательно сопоставляет вопрос со всеми фактами и правилами из базы знаний. Очевидно, что один факт и одно правило совместно предоставляют две возмож-

ности к унификации исходной цели. Обратите внимание, что в данном случае последовательность предложений в программе не имеет никакого значения. Можно поменять факт и правило местами без изменения смысла программы. Однако в данном случае процесс унификации проходит только на факте. О чем и заявляет Пролог. Но возможна такая постановка вопроса, которая приведет к унификации цели как на факте, так и на правиле.

Зададим такой вопрос: t(X,Y).

Получим ответ:

X=1, Y=3

X=1, Y=2

2 Solutions.

Итак, ***унификация*** – это встроенный базовый механизм Пролога, представляющий собой процесс последовательного сопоставления вопроса с фактами и правилами базы знаний с целью доказательства его реализуемости. В ходе сопоставления проверяется совпадение следующих сущностей: идентификаторы предикатов, количество аргументов, их значения. Если в вопросе есть свободные переменные, то Пролог ищет значения переменных, при которых цель достижима. Если в вопросе нет свободных переменных, то Пролог ищет возможность унификации цели и в качестве ответа выдает результативность такой унификации.

1.3. Бэктрекинг.

Бэктрекинг – это механизм возврата, в случае неудачи унификации, к ближайшей точке развилки процесса унификации, где может быть рассмотрен альтернативный вариант унификации цели. Бэктрекинг возможен только при наличии

альтернативных путей унификации цели. Альтернативный путь может быть обнаружен только при наличии в программе для унифицируемой цели двух или более предложений с головой такой же как и в вопросе цели.

Рассмотрим суть механизма бэктрекинга. В процессе унификации цели Пролог в специальном стеке точек возврата запоминает те точки в программе, в которых при последовательном просмотре предложений базы знаний возникали ситуации, когда для унификации есть белее одного пути (более одного подходящего предложения). Пролог сначала рассматривает первый подходящий путь до конца, то есть осуществляет «поиск в глубину». Если возникает ситуация невозможности унификации цели по унифицируемому пути (нет в базе подходящих к унификации предложений – не совпадают имена предикатов, значения аргументов или их количество) то, в этом случае и срабатывает механизм бэктрекинга. Пролог обращается к стеку точек возврата и, при наличии там точек возврата, возвращается к *ближайшей* точке развилки. При таком возврате переменные восстанавливают те значения, которые они имели в данной точке до начала унификации по текущему неуспешному пути. В дальнейшем процесс унификации уже протекает по следующему (альтернативному) пути. В случае исчерпания всех возможных путей в некоторой точке развилки механизм бэктрекинга обращается уже к *дальнейшей* (предыдущей) точке развилки.

Таким образом, бэктрекинг – это перебор вариантов унификации цели с возвратами, который организуется компилятором Пролога без участия программиста. Каждый раз, когда Пролог встречается с ситуацией наличия альтернативного варианта унификации, то он оставляет для себя информацию для последующего возврата. Просматривает основной

вариант и затем уже возвращается к организованной ранее точке возврата. Таких точек может быть несколько и в каждой из них может быть целый ряд альтернативных вариантов. Автоматический перебор с возвратами – существенное преимущество Пролога, так как освобождает программиста от необходимости явной организации такого перебора, например, с помощью циклов.

Рассмотрим бэктрекинг на конкретном примере.

Пусть необходимо организовать вывод на экран таблицы истинности логической функции "И". Задачу решим опираясь на встроенный предикат Пролога `bitand`.

В базе знаний будут размещаться два факта f(0) и f(1), задающие возможные значения аргументов логической функции. Пролог должен будет совершить полный перебор всех входных наборов для логической двухаргументной функции. Очевидно, что всего таких варианта четыре, а именно: 00, 01, 10, 11.

Надо только грамотно сформулировать цель поиска. Цель должна содержать такой вопрос, который будет иметь альтернативные варианты как для первого так и для второго аргумента.

Программа выглядит следующим образом.

```
PREDICATES
     f(integer)
CLAUSES
     f(0).
     f(1).
```

Зададим в окне диалога вопрос этой программе:

```
     f(A),f(B),bitand(A,B,X).
```

Получим ответ:

```
     A=0, B=0, X=0
     A=0, B=1, X=0
     A=1, B=0, X=0
```

```
A=1, B=1, X=1
4 Solutions
```

Процесс унификации проходил следующим образом.

Цель сложная, имеет три подцели. Первая f(A) запускает процесс унификации и Пролог, просматривая базу знаний сверху-вниз, останавливается уже на первом же факте f(0). Так как в вопросе переменная ничем не означена, то сопоставление заканчивается присвоением переменной A значения "0". Однако этот факт не единственно возможный для реализации унификации. Есть ещё и f(1), поэтому Пролог оставляет себе информацию об альтернативном варианте унификации (точку возврата №1).

Вторая подцель вопроса f(B) принуждает Пролог начать процесс унификации опять с самого начала, то есть просматривать все наличествующие факты сверху-вниз, начиная с первого. И первый же факт имеет тот же идентификатор f и необходимое количество аргументов (один). Таким образом сопоставление возможно. Так как переменная B ничем не означена, то сопоставление заканчивается присвоением ей значения "0". Однако этот факт не единственно возможный для реализации унификации. Есть ещё и f(1), поэтому Пролог оставляет себе информацию об альтернативном варианте унификации (точку возврата №2).

Третья подцель - стандартный предикат Пролога. Используя значения A и B как входные переменные, а X как выходную переменную - bitand выдает соответствующий ответ. А достигнутая исходная цель, состоящая из трех подцелей, позволяет Прологу вывести общий ответ.

И тут начинается самое интересное. Без дополнительных усилий со стороны программиста Пролог самостоятельно принимает решение о необходимости проверки альтернативных вариантов. Пролог возвращается к ближайшей точке

возврата - к точке №2. И берет за основу следующий, не исследованный ранее альтернативный вариант, а именно - унификация подцели f(B) возможна не только на первом факте f(0), но и на втором факте f(1). То есть в данном случае, после сопоставления со вторым фактом, переменная B означивается "1".

После этого комбинация переменных A=0 и B=1 передается на вход bitand и производится вывод соответствующего ответа.

Ну а что же далее, ведь альтернативы для второй подцели исчерпаны. А далее Пролог припоминает, что есть ещё и точка возврата №1, где были ещё нерассмотренные альтернативы и делает отступ в своих размышлениях до этой точки. То есть как бы забывает, что уже рассматривал когда-либо вторую подцель. Можно рассматривать это так, как если бы Пролог, приступив к унификации исходной цели и начав с первой подцели, сразу выбрал для неё второй из двух альтернативных вариантов - факт f(1). Сопоставление завершается присвоением переменной A значения "1".

После этого, то есть после унификации первой подцели Пролог переключается на вторую, так как если бы ранее этого еще не делал.

Вторая подцель вопроса f(B) принуждает Пролог начать процесс унификации опять с самого начала, то есть просматривать все наличествующие факты сверху-вниз, начиная с первого. И первый же факт имеет тот же идентификатор f и необходимое количество аргументов (один). Таким образом сопоставление возможно. Так как переменная B ничем не означена, то сопоставление заканчивается присвоением ей значения "0". Однако этот факт не единственно возможный для реализации унификации. Есть ещё и f(1), поэтому Пролог ос-

тавляет себе информацию об альтернативном варианте унификации (точку возврата №3).

После этого комбинация переменных A=1 и B=0 передается на вход bitand и производится вывод соответствующего ответа.

После этого Пролог возвращается к ближайшей точке возврата - к точке №3. И берет за основу следующий, не исследованный ранее альтернативный вариант, а именно - унификация подцели f(B) возможна не только на первом факте f(0), но и на втором факте f(1). То есть в данном случае, после сопоставления со вторым фактом, переменная B означивается "1".

Далее комбинация переменных A=1 и B=1 передается на вход bitand и производится вывод соответствующего ответа.

1.4. Рекурсия.

В Прологе отсутствуют операторы цикла, поэтому следует разобраться каким же образом организовать однотипную обработку данных (итерации). Наиболее очевидным способом организации итераций является создание рекурсивных процедур.

Решим задачу нахождения степени числа с помощью рекурсии.

Рекурсивное определение функции, вычисляющей степень числа таково:

```
ЧИСЛО x В СТЕПЕНИ y –
          есть ЧИСЛО x В СТЕПЕНИ y-1,
          УМНОЖЕННОЕ НА САМО ЧИСЛО x.
```

Например, два в пятой это два в четвертой, умноженное на два, где два в четвертой это два в третьей, умноженное на два и т.д.

Переведем эту мысль с русского на прологовский:

```
st(X,Y,Z):-
          YY=Y-1,
          st(X,YY,ZZ),
          Z=ZZ*X.
```

В этой программе используется трехаргументный предикат st. Первый аргумент (входной) предиката – это число, возводимое в степень; второй аргумент (входной) – это степень, в которую возводится число; третий аргумент (выходной) – ответ, куда размещается результат вычислений.

Зададим Прологу цель – st(2,5,Z).

Когда Пролог будет унифицировать эту цель, то произойдет рекурсивный вызов цели st(2,4,ZZ). Что в свою очередь приведет к рекурсивному цели st(2,3,ZZ) и так далее. Если рекурсивный вызов ничем не ограничивать, то он уйдет в область отрицательных чисел.

Ограничить ход рекурсии можно добавлением такого предложения, где бы цель могла быть унифицирована без дополнительных вызовов. Например, два в первой это два или

```
ЛЮБОЕ ЧИСЛО x В СТЕПЕНИ 1 - есть ЧИСЛО x.
```

Переведем эту мысль с русского на прологовский:

```
st(X,1,X).
```

Итак, рекурсия в Прологе это процедура, обычно составленная из двух предложений ***шаг рекурсии*** и ***базис рекурсии***. Шаг рекурсии обеспечивает последовательный рекурсивный вызов, а базис необходим для останова рекурсии. В связи с тем, что процесс унификации всегда проходит последовательно сверху-вниз и слева-направо, то, как правило, базис рекурсии размещают раньше, чем шаг. В обратном случае может произойти зацикливание встроенного механиз-

ма вывода решений, так как шаг рекурсии обязательно имеет в своем составе рекурсивный вызов.

Для нашего случая корректная очередность предложений в рекурсивной процедуре выглядит следующим образом:

```
st(X,1,X).
st(X,Y,Z):-
     YY=Y-1,
     st(X,YY,ZZ),
     Z=ZZ*X.
```

В ходе унификации исходной цели st(2,5,Z) на такой программе Пролог сначала попробует произвести унификацию первого предложения. Такая попытка закончится неудачей, так как второй аргумент в предложении указан "1", а в цели второй аргумент заявлен как "5". Тогда Пролог перейдет к унификации второго предложения, где и будет осуществлен рекурсивный вызов с измененным значением второго аргумента. Процесс унификации новой цели st(2,4,ZZ) снова будет начат с первого предложения, где опять закончится неудачей. Что приведет Пролог к необходимости рассмотрения второго предложения. Путем таких рекурсивных вызовов Пролог постепенно (начиная от 5 через 4,3,2) доберется до значения второго аргумента равного 1. После чего уже будет унифицировано первое предложение. Это приведет к присвоению третьему аргументу значения первого – именно так обозначено в первом предложении: st(X,1,X).

Достигнутая цель запустит обратный ход рекурсии, который будет представлен последовательным выполнением третьей подцели второго предложения Z=ZZ*X всех шагов рекурсии в обратном порядке от 1 до 5. То есть после срабатывания базиса рекурсии первое такое выполнение будет таким: Z=2*2. И Пролог уже узнает не только чему равно два в первой, но и чему равно два во второй степени.

Обратный ход рекурсии будет продолжаться до унификации основной цели st(2,5,Z), после чего ответ будет выдан на экран:

```
Z=32
```

Однако, выполнение программы на этом не исчерпано и вскоре мы узнаем, что стек переполнен. О чем нам пытается сообщить Пролог, ведь ответ то уже был получен. Дело в том, что запрос st(2,1,ZZ) может быть унифицирован не только в первом предложении, но во втором. Ведь там на это не наложено никаких ограничений. Это приводит к тому, что Пролог все таки уходит в область отрицательных значений. И продолжает это до тех пор пока не происходит переполнение стека.

Ну что же исправим это недоразумение, добавив соответствующее ограничение:

```
st(X,1,X).
st(X,Y,Z):-
        Y>1,
        YY=Y-1,
        st(X,YY,ZZ),
        Z=ZZ*X.
```

Дополнительное условие не позволит проводить унификацию цели на втором предложении, если второй аргумент меньше или равен единице.

Возможен и другой вариант:

```
st(X,1,X):-!.
st(X,Y,Z):-
        YY=Y-1,
        st(X,YY,ZZ),
        Z=ZZ*X.
```

В данном случае при успешной унификации (и только при успешной) первого предложения срабатывает безусловный стандартный предикат отсечение "!", который означает удаление из памяти альтернативных путей унификации цели.

В данном случае альтернативным путем унификации цели является второе предложение в процедуре. Предикат «отсечение» всегда унифицируется успешно и, в данном случае, его использование приводит к тому, что после получения первого возможного решения Пролог не будет пытаться найти другие и не уйдет в область отрицательных значений. Обычно для останова рекурсии используют именно предикат отсечение.

Глава №2. Практика разработки элементарных баз знаний.

2.1. Организация простейшего диалога с программой.

Данная глава посвящена практическому исследованию особенностей работы механизмов унификации и бэктрекинга, поэтому для полноценного изучения данного материала следует установить на своем компьютере систему PDC Prolog (например, с ресурса proprolog.narod.ru) и создавать программы вслед за рассуждениями автора.

Каждая программа на Прологе представляет собой текстовый файл, который для запуска из системы Пролог должен иметь расширение «PRO». Пусть система PDC Prolog расположена в папке C:\PROLOG. Создайте проводником в папке C:\PROLOG дополнительную папку с именем MY_PROG и установите текущую директорию Пролога C:\PROLOG\MY_PROG (меню: Files/Change dir).

Создайте новый файл. На начальном этапе будем опираться в основном на встроенные предикаты (write, readln, nl). Наберите следующий текст программы, содержащий безаргументный предикат **ok**:

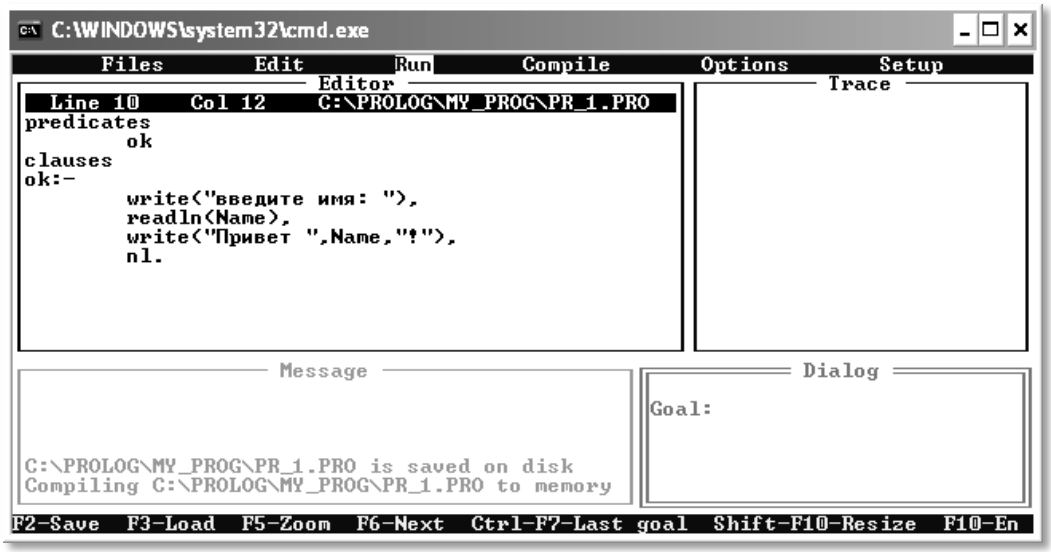


Рис.1. Организация простейшего диалога.

Сохраните программу (клавиша F2) под именем pr_1.pro и проверьте, в какую папку была сохранена программа.

Нажмите сочетание клавиш Alt-R для запуска программы на выполнение – курсор переместится в окно Dialog и система Пролог будет ожидать от вас вопроса **Goal: _** (цели, которую необходимо достичь). Введите **ok** и нажмите Enter. После диалога с Прологом вы можете получить такой результат:

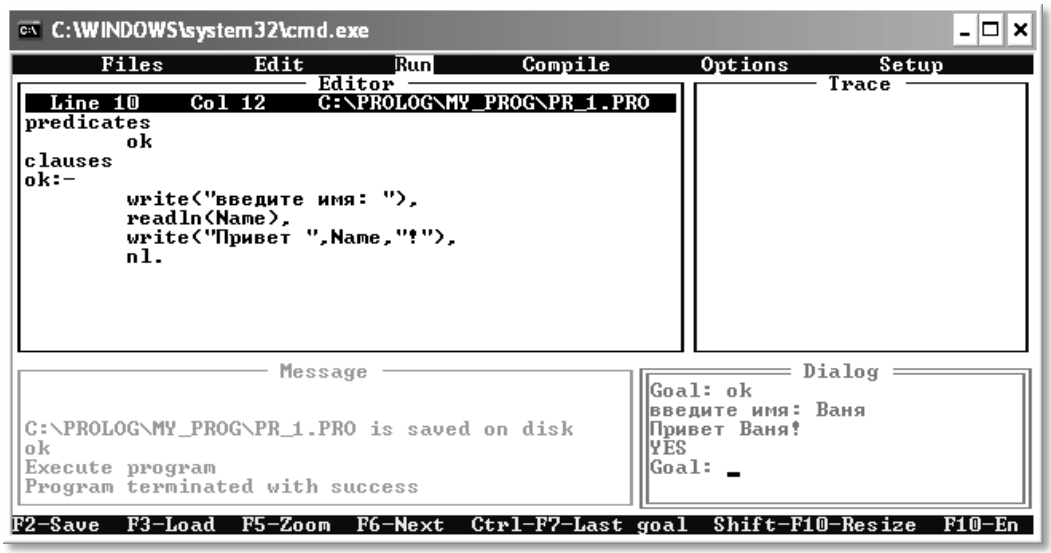


Рис.2. Реализация простейшего диалога в окне Dialog.

Что произошло? После получения задачи на достижение цели **ok** Пролог находит её голову в теле программы и пытается проверить её реализуемость. Оказывается, что чтобы предикат **ok** выполнился необходимо достичь подцель write(“введите имя: ”) и подцель readln(Name) и подцель write(“Привет ”,Name,”!”,) и подцель nl. Пролог пытается последовательно их достичь и когда все они выполнятся, то он с удовлетворением отмечает, что цель достижима – **Yes** (см. окно Dialog). После чего объявляет пользователю, что он готов к поиску следующей цели – **Goal: _** .

В этой программе используется один нестандартный предикат, то есть тот который отсутствует в лексике Пролога, а именно предикат **ok**. Все нестандартные предикаты должны быть объявлены (см. тест программы). Вы можете поменять имя предиката на такое, которое вам кажется более приемлемым для обозначения решаемой им задачи и попробовать запустить программу вновь.

2.2. Организация диалога в оконном режиме.

Внесите некоторые изменения в текст программы и сохраните её под именем pr_2.pro (только не используйте F2, иначе вы сотрете программу pr_1.pro; надо использовать пункт меню Files/Write to).

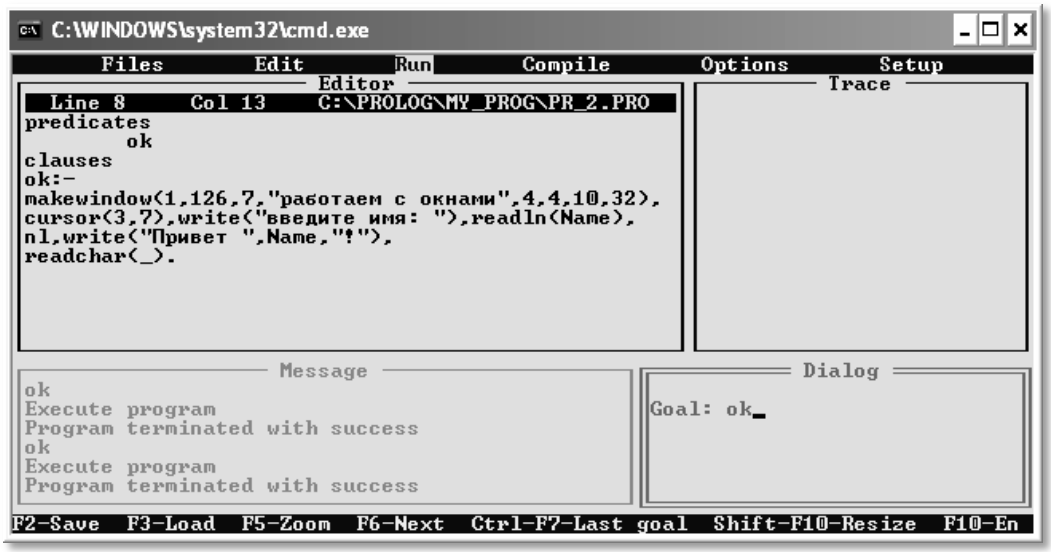


Рис.3. Организация диалога в оконном режиме.

Поработайте с программой. В частности запустите её с предикатом readchar и без него. И вам станет понятно, зачем я добавил его...

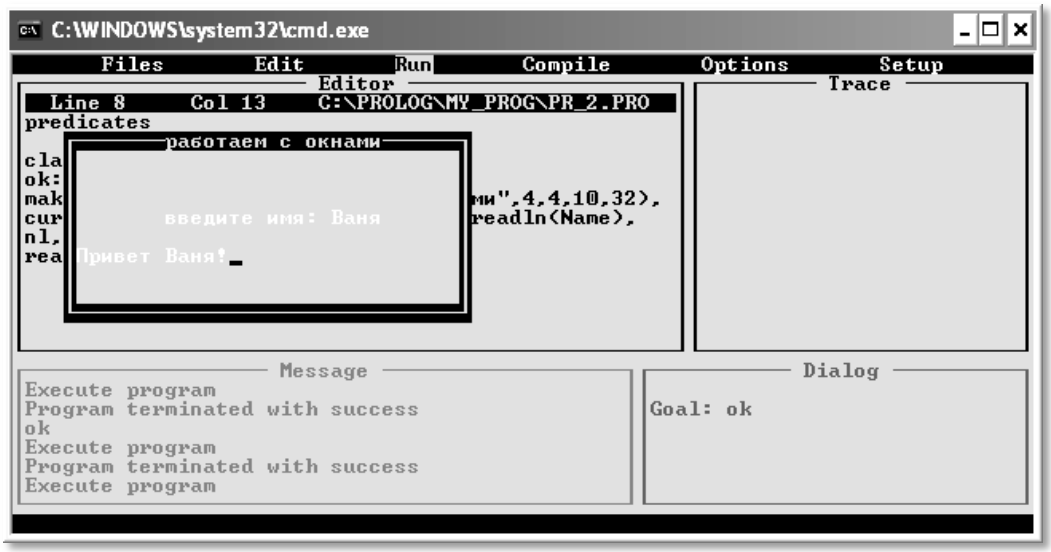


Рис.4. Реализация диалога в оконном режиме.

Естественно, что этот стандартный предикат удерживает окно до того момента пока не получит какой-нибудь символ (т.е. нажатие любой клавиши).

Знак подчеркивания в качестве аргумента предполагает, что это анонимная переменная и после нажатия той самой «какой-нибудь» клавиши её код нигде не будет сохранен. Сравните с readln(Name), здесь как раз нам важно сохранить набранную пользователем строку, поэтому у переменной есть имя.

Дальше я предлагаю вам поработать самим. Разберитесь с предикатом cursor, с оформлением окна. Напомню, что в Прологе есть справка (F1).

Вам необходимо понять, как назначать размеры и положение окна, как назначать цвета для фона и шрифта в окне и в рамке окна. К примеру, сделайте так, чтобы непосредственно в окне был зеленый шрифт на синем фоне, а в рамке зеленый шрифт на красном фоне.

2.3. Описание мира и процесс унификации.

Возвращаемся собственно к трудностям, возникающим на начальном этапе освоения Пролога. Чтобы решить задачу хочется машине рассказать, как её решать, то есть составить алгоритм. Однако, программа в логическом программировании не есть алгоритм, а есть база знаний – совокупность фактов и правил, описывающих отношения между объектами предметного поля. Мы формулируем вопрос к базе знаний, а Пролог сам в ходе последовательной унификации пытается найти ответ.

Унификация есть процесс сопоставления вопроса с фактами и правилами базы знаний с целью доказательства его реализуемости. Здесь важно отметить, что процесс сопоставления первичной основной цели происходит последовательно по всем предложениям в базе знаний – сверху вниз. Причем,

если цель оказывается головой какого-то предложения, тело которого состоит из нескольких предикатов, то основная цель разбивается на подцели, доказательством которых Пролог занимается по отдельности и последовательно. То есть процесс сопоставления подцелей в теле правила всегда происходит слева направо.

Попробуем разобраться, как именно происходит процесс унификации, для чего обсудим два вопроса:

1) определение отношений на основе фактов;

2) определение отношений на основе правил.

2.3.1. Определение отношений на основе фактов.

Пролог не приспособлен для решения вычислительных задач. Суть языка позволяет решать задачи, которые связаны с описанием объектов и отношений между ними.

Пусть мир представлен фактами:

1) Наташа есть мама Даши;

2) Даша – мама Маши;

3) Маша – мама Глаши и Параши.

Пусть двухаргументный предикат `mama` задает отношение между первым и вторым аргументами, предписывая, что первый аргумент является мамой по отношению ко второму.

Создайте новый файл pr_3.pro и наберите следующую программу:

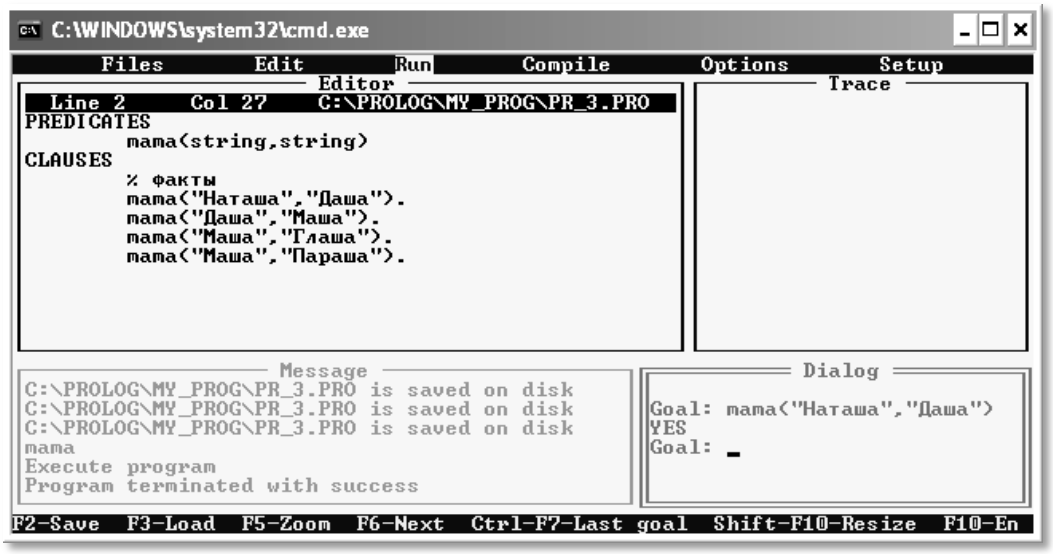


Рис.5. Описание предметной области только фактами.

Спросим, является ли Наташа мамой Даши:

```
mama (“Наташа”, ”Даша”)
```

Приступив к последовательному (сверху - вниз) сравнению поставленной цели с фактами, содержащимися в базе знаний, и, обнаружив уже на втором шаге, что mama (“Наташа”,”Даша”) – это факт, о существовании которого явно утверждается в тексте программы, Пролог отвечает Yes. Иными словами, если вопрос по сути есть предикат с означенными аргументами, то поиск решения есть простое сравнение со всеми фактами из базы знаний, а возможные исходы поиска Yes или No.

Спросим иначе:

```
mama (X, "Даша")
```

Этот вопрос содержит только один означенный аргумент (Даша), а в качестве первого аргумента – неозначенная переменная X. Пролог последовательно просмотрит все факты и, наткнувшись на такой, который содержит в качестве второго аргумента значение “Даша” конкретизирует переменную X значением “Наташа”.

Сравните, чем будет отличаться поиск ответа на вопрос: `mama ("Маша", X).`

Проверьте также вариант: `mama (Y, X).`

Если есть необходимость определить только часть информации, например, перечислить только дочерей, тогда можно использовать анонимную переменную в вопросе:

```
mama (_, X).
```

Получим ответ:

```
X=Даша
X=Маша
X=Глаша
X=Параша
4 Solutions
```

И, наконец, если надо получить ответ на вопрос: есть ли информация о людях, находящихся в отношении "мама - дочка", то его можно сформулировать в виде:

```
mama (_,_),
```

В данном случае нам не важны конкретные имена, а интересует, есть ли в нашей базе знаний хотя бы один соответствующий факт. Ответом в данном случае будет просто "Yes". Система сообщит о том, что у нее есть информация об объектах, связанных отношением "mama".

Однако, нашей программе можно задать вопрос и посложнее, например, спросить кто является мамой мамы Глаши:

```
mama(X,Y),mama(Y,”Глаша”).
```

Проверьте, какой будет ответ. Поменяйте в вопросе предикаты местами и проверьте, будет ли получено верное решение и имеет ли значение для Пролога порядок следования предикатов?

Можно также узнать, найдется ли такая женщина, которая является одновременно мамой как для Глаши так и для Параши.

Можно задать вопрос и более общий – есть ли такая мама, у которой две дочери:

```
mama(Q,W),mama(Q,E),W<>E.
```

Проверьте, как сработает Пролог при поиске ответа на такой вопрос, и прокомментируйте результаты.

2.3.2. Определение отношений на основе правил.

Правила значительно расширяют возможности базы знаний, так как они универсальны и могут вместо конкретных значений содержать переменные.

К примеру, введем в программу правило, которое определяет отношение бабушка, как мама мамы:

```
baba(X,Y):-
          mama(X,Z),
          mama(Z,Y).
```

Обычно для легкости чтения правила записывают именно в несколько строчек, но это не предопределено специально синтаксисом Пролога и при необходимости вы можете то же самое правило записать и в одну строчку:

```
baba(X,Y):-mama(X,Z),mama(Z,Y).
```

Сделайте программу такой как показано ниже:

```
DOMAINS
     s = string
PREDICATES
     mama(s,s)
     baba(s,s)
CLAUSES
     mama("Наташа","Даша").
     mama("Даша","Маша").
     mama("Маша","Глаша").
     mama("Маша","Параша").
baba(X,Y):-
     mama(X,Z),
```

```
mama(Z,Y).
```

Сохраните эту программу под именем pr_4.pro.

Несколько комментариев по поводу оформления программы.

Комментарий №1.

Программа на Прологе состоит из разделов. В данном случае вы видите следующие разделы: DOMAINS – раздел описания типов (доменов), PREDICATES – раздел описания предикатов и CLAUSES – раздел описания предложений. Возможно использование и других разделов: GOAL – раздел описания внутренней цели, CONSTANTS – раздел описания констант и DATABASE – раздел описания предикатов внутренней базы данных. Мы постепенно познакомимся со всеми этими разделами.

Комментарий №2.

Заметьте, что раньше мы писали mama(string,string), а теперь mama(s,s).

Дело в том, что можно использовать описание доменов для сокращения имен стандартных доменов. В частности, чтобы не писать каждый раз string, можно задать описание s = string и далее использовать вместо ключевого слова string односимвольное обозначение s.

К стандартным доменам относятся:

`integer` - целое число (из промежутка -32768...32767);

`real` - действительное число (лежащее между ±1e-307...±1e308);

`char` - символ, заключенный в одиночные апострофы;

`string` - последовательность символов, заключенная в двойные кавычки.

Комментарий №3.

Советы общего плана по написанию программ:

1) не забывайте наглядно и однообразно оформлять все предложения в программе;

2) оставляйте комментарии, так:

```
% однострочный комментарий
```

или так:

```
/* многострочный
   комментарий */;
```

3) давайте идентификатор (имя) предикатам, так чтобы интуитивно был понятен их смысл;

4) учитывайте порядок сопоставления предложений в процедуре.

Итак, запустите программу pr_4.pro и задайте машине вывода вопрос baba(“Наташа”,W).

Процесс унификации будет проходить следующим образом:

Последовательно сверху – вниз будет происходить поиск предиката baba. Первые четыре факта не подходят. Наткнувшись на правило baba, Пролог сопоставляет вопрос baba(“Наташа”,W) с головой правила baba(X,Y), после чего переменная X конкретизируется значением “Наташа”, а W связывается с переменной Y из правила baba(X,Y). Далее Пролог пытается последовательно достигнуть подцели mama(“Наташа”,Z) и mama(Z,Y).

Просматривая факты Пролог натыкается на факт mama("Наташа","Даша"), что приводит к конкретизации переменной Z значением “Даша”. Подцель считается достигнутой и Пролог переходит к доказательству подцели mama(Z,Y), а если точнее, то – mama(“Даша”,Y). Поиск снова начинается с первого факта, который в данном случае не может быть сопоставлен с подцелью ввиду несопоставимости

первых аргументов (оба они конкретизированы, но разными значениями, соответственно, “Наташа” и “Даша”). Но уже на втором шаге сопоставление становится возможным и переменная Y конкретизируется значением “Маша”.

Среда программирования позволяет анализировать программисту описанный процесс унификации по шагам. Для этого добавьте первой строчкой программы слово trace (см. рис.6).

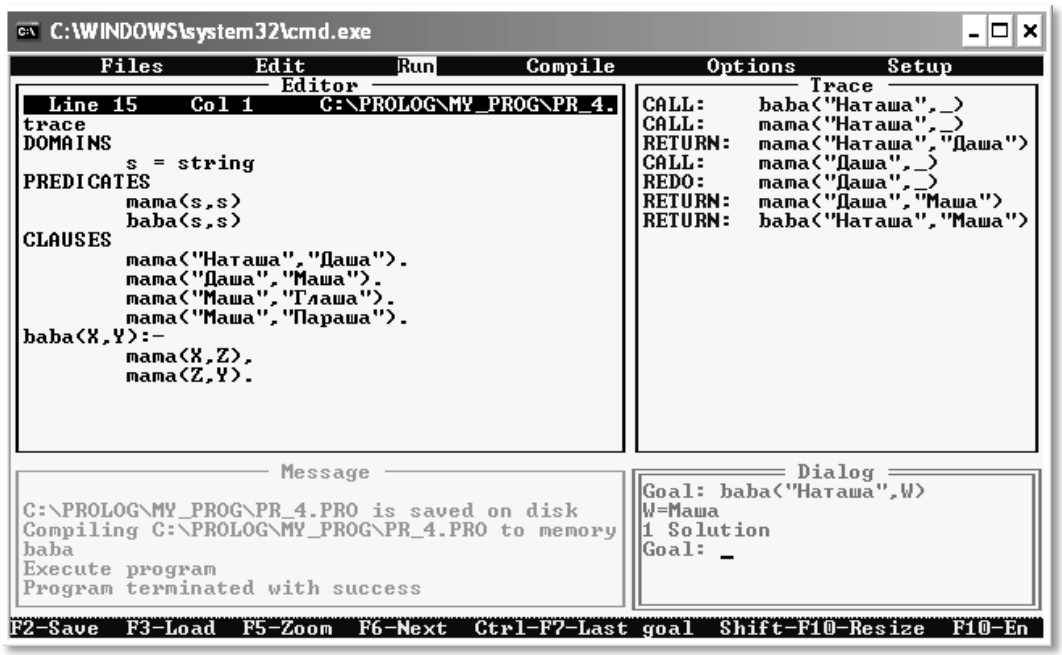


Рис.6. Трассировка процесса унификации.

Запустите программу, она будет выполняться пошагово с отображением текущего выполняемого действия (курсором в тексте программы и результатом этого действия в окне Trace). Чтобы выполнить следующий шаг необходимо нажимать клавишу **`F10`**. Исследуйте процесс унификации всех предыдущих программ. Стандартный предикат trace обычно используют в процессе отладки программы.

Если описываемый мир несколько сложнее, чем это может описать одно правило, то необходимо внести корректи-

вы. Например, бабушка это не только мама мамы, но и мама папы:

```
baba(X,Y):-mama(X,Z),mama(Z,Y).
baba(X,Y):-mama(X,Z),papa(Z,Y).
```

Совокупность предложений с одинаковым предикатом в заголовке принято называть процедурой. Считается, что между правилами процедуры неявно существует «ИЛИ».

То есть, к примеру, в семье, где у Глаши есть папа Степа и мама Маша, причем, у Степы мама Оля, а у Маши мама Даша – выходит, что у Глаши две бабушки.

Попробуйте самостоятельно написать базу знаний про эту семью и сформулировать так вопрос к ней, чтобы получить исчерпывающий ответ о количестве и именах бабушек Глаши.

Интересным с исследовательской точки зрения может быть разработка базы знаний про некую семью, отношения между членами которой могут быть описаны следующей схемой:

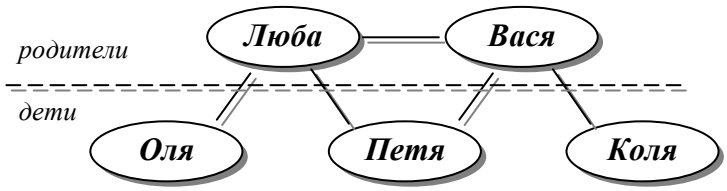


Рис.7. Схема связей для построения базы знаний.

База знаний будет состоять из совокупности фактов и правил. Для начала продекларируйте все объекты предметного поля и связи изображенные на схеме (рис.7) в виде фактов:

– мужчина,

– женщина,

– родитель (4 шт.),

– в браке (1 шт.).

Далее опишите двухаргументные предикаты и разработайте для них правила, определяющие для конкретного человека (указанного в первом аргументе):

1) сестру/брата;
2) мужа/жену;
3) кровную сестру/брата
4) сводную сестру/брата.

Ответ после работы правила должен размещаться во втором аргументе.

2.4. Организация поиска решений.

Приступая к изучению данной главы вы должны уже разбираться в следующих вопросах:

– что такое декларативное программирование и в чем его отличие от процедурного;

– все программы Пролога представляют из себя базы знаний, включающие два вида предложений – факты и правила;

– факты содержат безусловные утверждения, а правила – условные;

– совокупность правил с одинаковым предикатом в заголовке принято называть процедурой;

– суть выполнения программы на языке Пролог сводится к поиску ответа на вопрос, предъявляемый к базе знаний;

– процесс поиска ответа на вопрос – это процесс последовательной унификации, то есть сопоставление вопроса с предложениями программы.

Однако ранее мы лишь обозначили некоторые возможности описания мира в Прологе через определение отношений на основе фактов и правил. Возможности логического программирования на языке Пролог расширяются за счет использования дополнительных механизмов, в частности, задание данных через структуры, списки, определение отношений на основе рекурсивных правил. В данной главе мы продолжим изучать основы логического программирования и сконцентрируемся на следующих вопросах:

– встроенный механизм поиска решений, как совокупность унификации и бэктрекинга;

– рекурсия как способ итеративной обработки информации;

– организация запроса к базе знаний в виде внутренней или внешней цели;

– порядок включения в текст основной программы программного кода из другого файла;

– способы создания своих и использования встроенных логических функций.

2.4.1. Последовательная детализация правил вывода.

И начнем мы с описания потенциальных возможностей совершения ошибки в логической программе. Зачастую такие ошибки выявляются только при расширении базы знаний. В этой части уточним также особенности работы Пролога с внутренней целью, в частности отличие процесса унификации внутренней цели от внешней.

Машины осуществляют любую деятельность в высшей степени буквально. Они будут делать именно то, что вы попросили, а не то что вы подразумевали, но не сумели точно сформулировать. Машина действует только в рамках заложенных в неё правил и все попутные ущербы, сторонние для её миропонимания, её не интересуют. На этом основаны некоторые фантастические фильмы о будущем мире с кибернетическими монстрами.

В логическом программировании существует риск получения неправильных или избыточных решений в том случае, когда программа первоначально тестировалась на ограниченном подмножестве фактов и правил, а в дальнейшем эксплуатируется на ином, более широком подмножестве. Рассмотрим пример.

Пусть есть семья: мама Маша и её дочки Саша и Даша. Предположим, что для описания этой ситуации мы использу-

ем двухаргументный предикат parent (родитель). Первый аргумент этого предиката задает родителя, а второй его ребенка. Запустите систему Пролог, наберите приведенный ниже код (см. рис.8, окно Editor) и сохраните программу под именем parent.pro в папку C:\PROLOG\MY_PROLOG, запустите её на выполнение (Alt-R), в диалоговом окне задайте цель parent(Z,A). Объясните полученный результат (см. рис.8, окно Dialog).

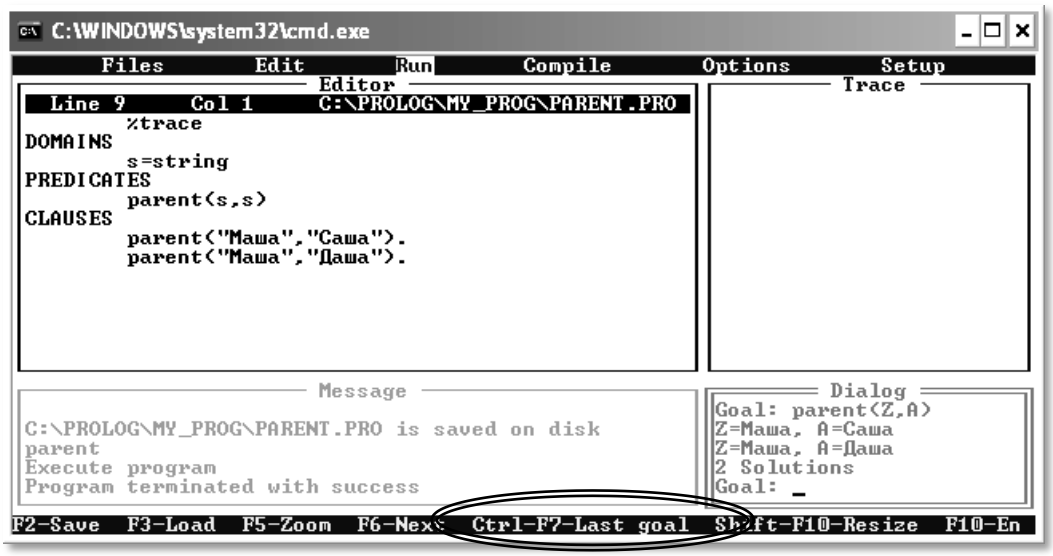


Рис.8. Унификация цели с двумя переменными.

Давайте модифицируем цель. Нажмите **Ctrl-F7** для возвращения последней набранной цели и Z замените на “Маша”. Пусть Пролог выполнит реализацию этой цели. Объясните полученный результат (см. рис.9).

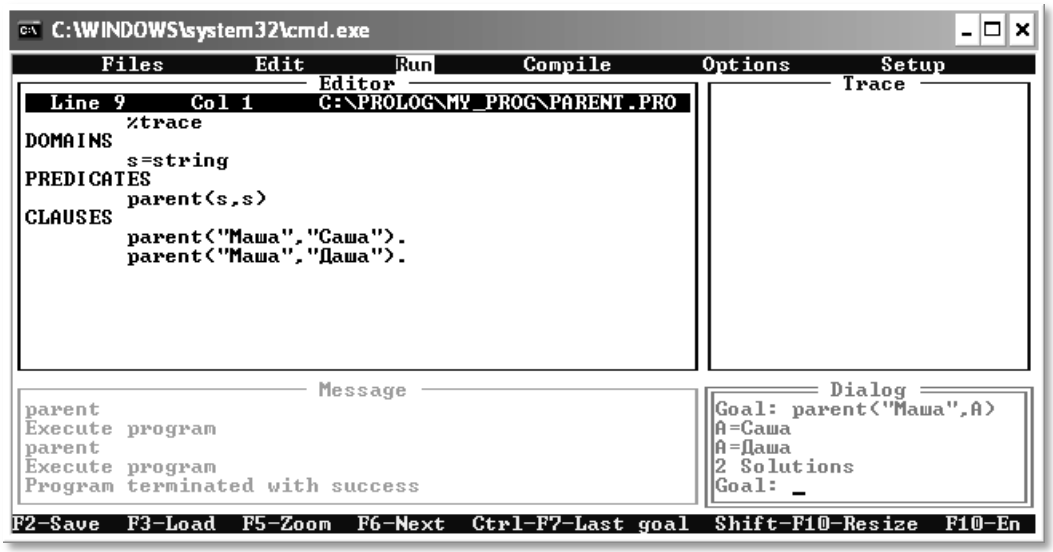


Рис.9. Унификация цели с одной переменной.

Дополним программу предикатом sister, задающим отношение сестра между двумя субъектами. Правило для выполнения этого предиката, предполагает выполнимость двух подцелей: первый и второй аргументы являются сестрами, если у них один родитель. После запуска видоизмененной программы для поиска цели - «Кто есть сестра Саши?» - мы получим как правильное решение («Даша») так и неправильное («Саша») (см. рис. 10, окно Dialog).

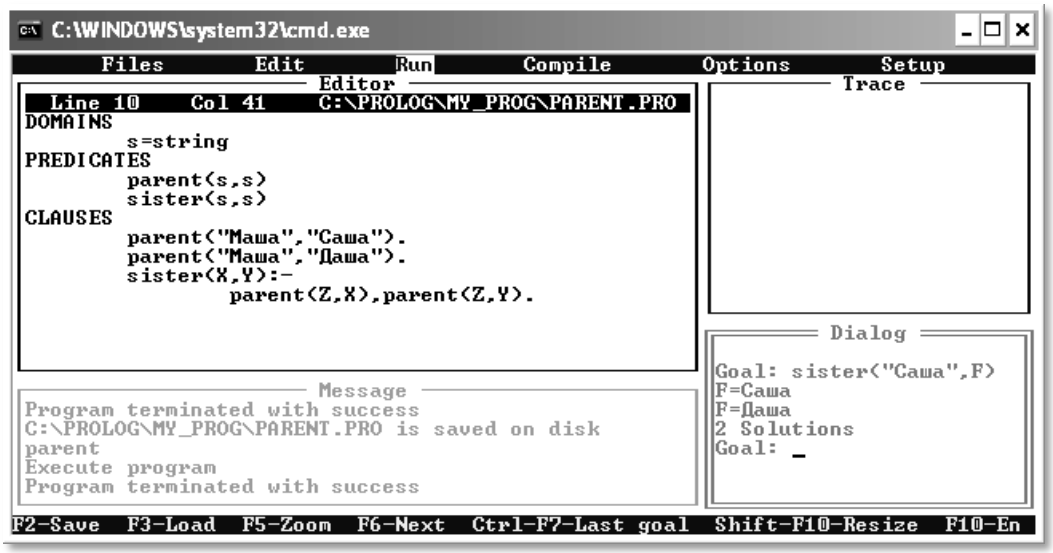


Рис.10. Унификация цели с выводом ошибочного решения.

Но оно неправильное именно с нашей точки зрения. Мы же понимаем, что Саша не является сестрой самой себе. Сделайте трассировку программы, добавив в начало кода предикат trace, и объясните результат.

Пролог в этой ситуации не виноват, так как это мы допустили неточность. Внесем изменения, уточнив, что субъект не может быть сестрой самого себя. После запуска программы мы избавимся от неверного ответа (см. рис.11, окно Dialog).

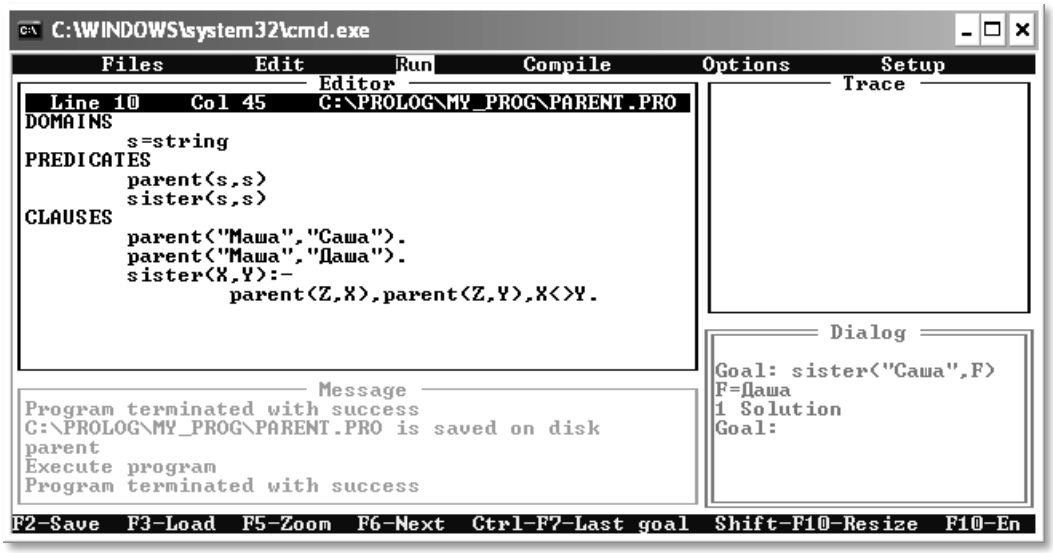


Рис.11. Унификация цели с ограничением.

Однако, если ***в дальнейшем***, кто-то ***добавит*** в текст программы такой факт «у Маши есть сын Коля», то поиск сестры Саши опять даст один неверный результат (см. 12).

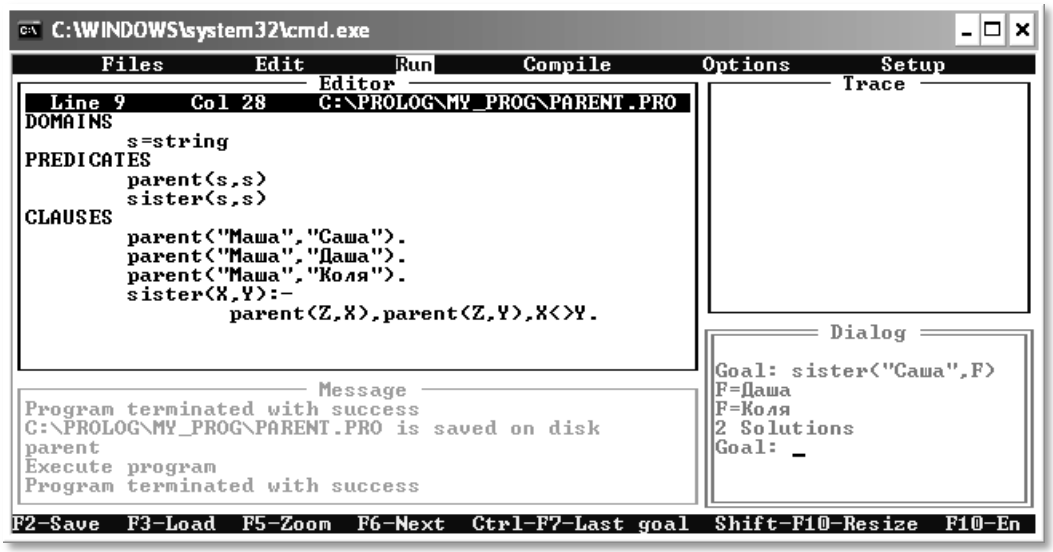


Рис.12. Унификация цели с дополнительным фактом.

Для нас очевидно, что Коля не может быть сестрой не только Саши, но и вообще ничьей сестрой, но машине вывода системы Пролог об этом никто не сказал. Придется добавить в описание правила уточнение о том, что сестра может быть только женского пола и перечислить тех, кто является женщинами (см. рис.13).

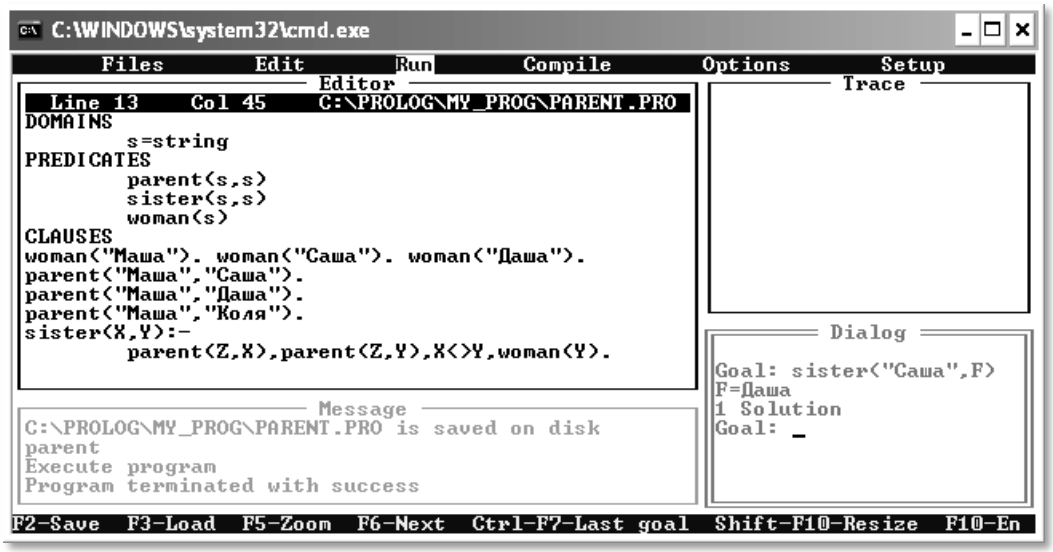


Рис.13. Унификация цели с двумя ограничениями.

Но может возникнуть неприятность при переходе к внутренней цели. Пусть у Маши есть ещё одна дочь – Женя. В этом случае поиск цели sister(“Саша”,F) будет завершен сразу после отыскания первого же подходящего варианта, то есть “Даши”.

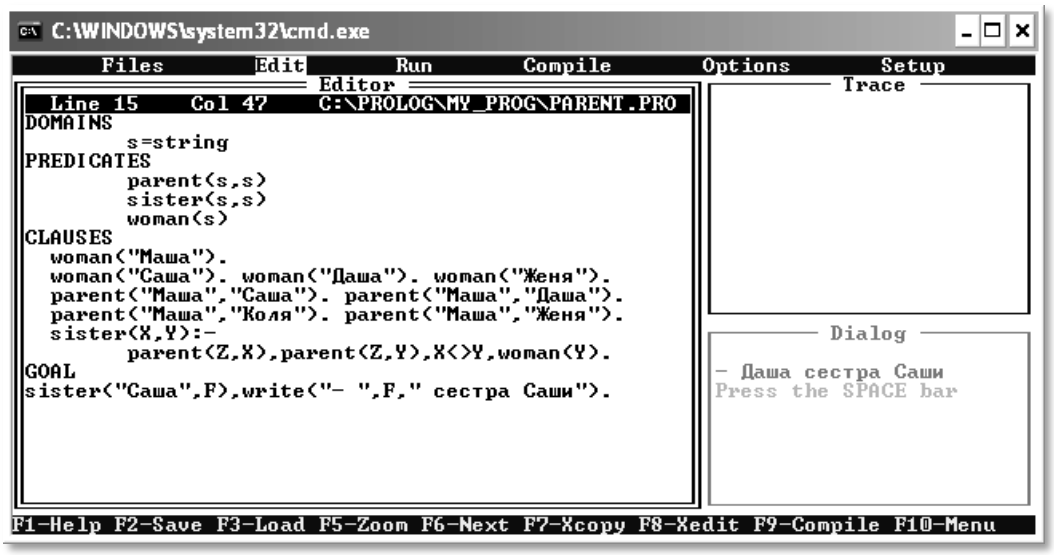


Рис.14. Унификация внутренней цели.

Тем не менее, существует способ корректно принудить Пролог совершить полный перебор. Для того, чтобы заставить Пролог не останавливаться на достигнутом, необходимо каждый раз после отыскания приемлемого варианта унификации цели указывать ему, что эта цель, на самом деле, недостижима. В этом случае Пролог будет пытаться искать альтернативные варианты унификации. Чтобы текущий вариант не терялся, его нужно как-нибудь зафиксировать, например, вывести текущий ответ на экран.

Как именно реализуется подобный механизм? Необходимо в цель добавить стандартный предикат fail, который переводится как “потерпеть неудачу”. Этот предикат безусловный и когда Пролог добирается до него, то происходит то же самое, как если бы повстречалась такая запись “1=2”. Под-

цель fail (или “1=2”) невыполнима, что приводит Пролог в состояние, при котором он делает вывод, что общая цель недостижима и, если ещё есть альтернативные варианты унификации, то можно приступить к их рассмотрению. То есть, в данном случае, обнаружив, что Даша есть сестра Саши, Пролог выводит этот результат на экран, делает переход на следующую строку (предикат nl – “new line”) и сталкивается с непреодолимой проблемой в виде fail, которая не дает ему остановиться в поиске (см. рис. 15).

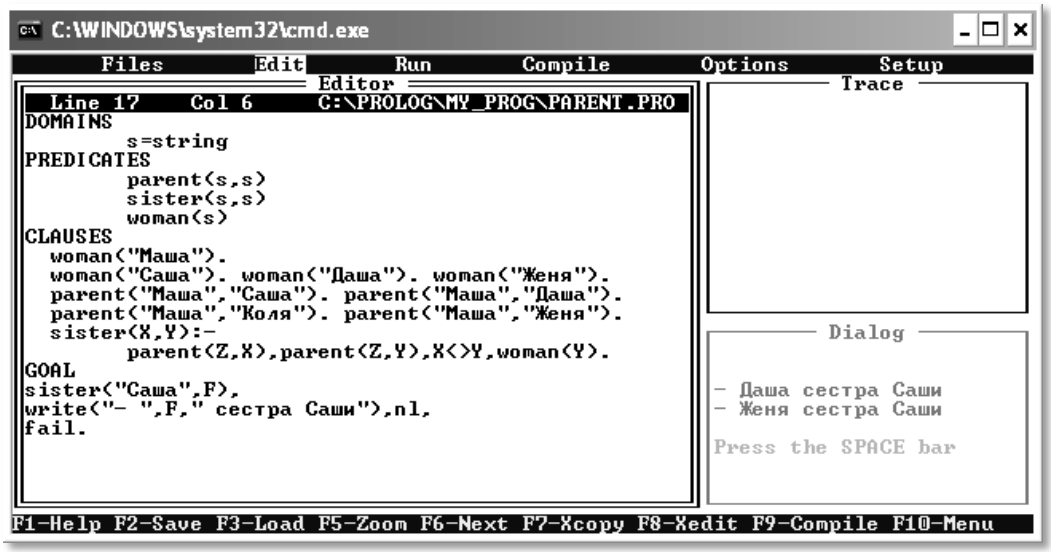


Рис.15. Унификация внутренней цели с полным перебором.

Попробуйте поставить вместо fail какую-нибудь заведомо невыполнимую подцель (например, 1=2) и проверьте работоспособность программы. Оцените результаты с помощью трассировки.

Внутренняя цель программы, конечно, несет больше преимуществ, чем проблем для программиста. Будем с этим постепенно разбираться. В частности давайте вспомним организацию диалога в оконном режиме (рис.16):

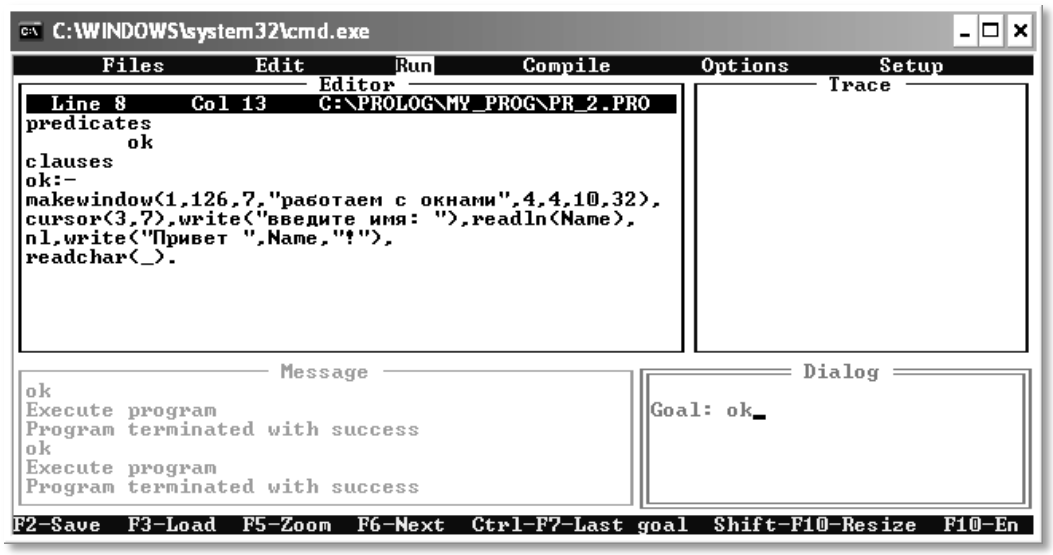


Рис.16. Организация диалога в оконном режиме.

Здесь стандартный предикат readchar с анонимной переменной просто необходим, так как удерживает окно диалога с пользователем до того момента пока не получит какой-нибудь символ (т.е. нажатие любой клавиши).

В этой программе отсутствует внутренняя цель. Каждый раз, когда вы её запускаете, приходится сообщать Прологу, что вам собственно от него нужно – набирать вручную в окне Dialog цель (см. рис. 16, окно Dialog). Это не всегда удобно. Можно поступить иначе. Добавьте к тексту программы две строчки:

```
goal
ok.
```

И можете убрать предикат readchar из тела предиката ok. В нем теперь отпадает необходимость. Запустите программу (Alt-R) в таком варианте и ощутите разницу.

Кроме того, только при наличии внутренней цели в программе её можно откомпилировать в отдельный исполняемый файл.

2.4.2. Организация модулей.

Часть программы, например, базу знаний (совокупность фактов и правил), иногда хочется хранить отдельным файлом, чтобы можно было её использовать в любой другой программе. Для реализации этой цели можно использовать стандартный предикат include. Синтаксис языка подразумевает, что после ключевого слова include будет следовать обозначение файла, необходимого для включения в основную программу. Возможны два варианта подключения внешнего файла.

Вариант №1.

Формат использования:

```
include "диск\\путь к файлу\\имя файла",
```

например:

```
include "C:\\Prolog\\MY_PROG\\000.txt".
```

Обратите внимание, что в Прологе принято в описании пути использовать двойной обратный слеш.

Рассмотрим, как это работает на примере программы про многодетную семью из примера, рассмотренного ранее. Все факты, касающиеся определения пола членов семьи и отношений родитель – дитя вынесем в отдельный файл `000.txt`.

Содержимое файла 000.txt (обратите внимание на то, что кодировки текстового файла с правилами и самой программы должны совпадать):

```
clauses
% факты
woman("Маша").
woman("Саша").
woman("Даша").
woman("Женя").
parent("Маша","Саша").
parent("Маша","Даша").
parent("Маша","Коля").
parent("Маша","Женя").
```

Тогда текст программы с внесенными изменениями будет выглядеть следующим образом:

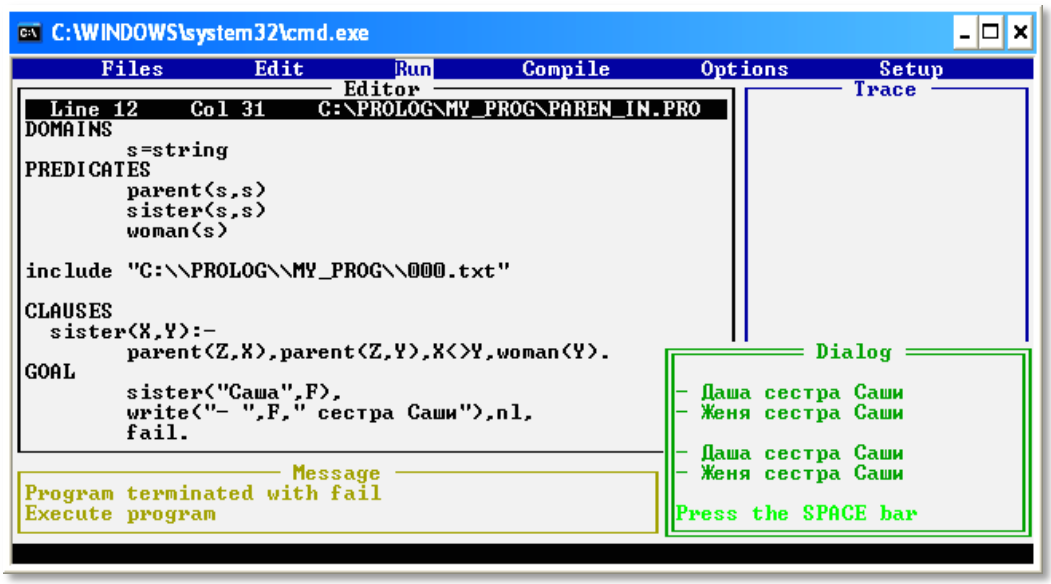


Рис.17. Подключение модуля с кодом с указанием пути.

Вариант №2.

Формат использования: `include "имя файла"`, например, `include "000.txt"`. При этом Прологу следует отдельно указать где именно этот файл `000.txt` находится. Путь к файлу можно определить через меню (см. рис.18).

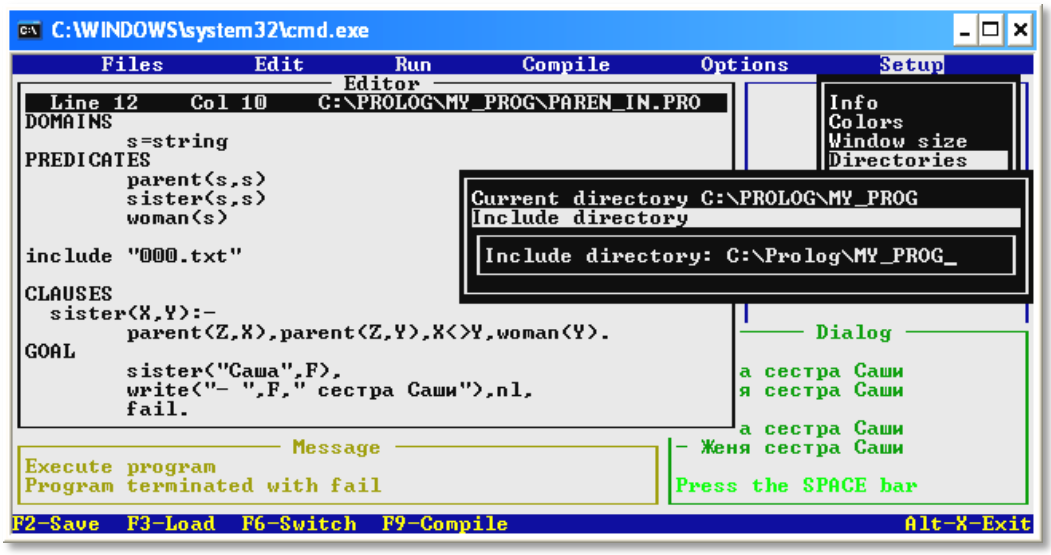


Рис.18. Подключение модуля из предопределенной папки.

Подобным образом можно во внешнем файле хранить любую часть программы и включать её по мере необходимости. Попробуйте, к примеру, следующий вариант.

Текст программы:

```
include "111.txt"
GOAL
     ok.
```

При этом, содержимое файла `111.txt`:

```
DOMAINS
     s=string
PREDICATES
     parent(s,s)
     sister(s,s)
     woman(s)
     ok
CLAUSES
  woman("Маша").
  woman("Саша").
  woman("Даша").
  woman("Женя").
  parent("Маша","Саша").
  parent("Маша","Даша").
  parent("Маша","Коля").
  parent("Маша","Женя").
  sister(X,Y):-
     parent(Z,X),
     parent(Z,Y),
     X<>Y,
     woman(Y).

  ok:-
     sister("Саша",F),
     write("- ",F," сестра Саши"),nl,
     fail.
```

2.4.3. Логические функции.

Проведите кропотливую работу с данной частью пособия, и вы сделаете значительный шаг вперед на пути к самостоятельному логическому программированию.

Давайте составим программу построения таблицы истинности для разных логических функций (не используя

стандартные предикаты логических функций). Начнем с функции «И».

Для компактности написания я не буду оформлять отдельного окна, а вывод организую непосредственно в системное окно Dialog. Вам же я рекомендую посвятить некоторое время конструированию дизайна окна вывода и таблицы в нем.

Итак, таблица будет представлена названием и заголовком:

```
GOAL
 write("таблица И"),nl,
 write("A B Z"),nl,
 i.
```

Здесь A и B есть входные переменные, а Z – значение логической функции. Предикат i следовательно предусмотрен для формирования четырех строк (0 0 0; 0 1 0; 1 0 0; 1 1 1). Обсудим его работу. Он должен делать возрастающий перебор всех вариантов сочетаний значений входных данных и выдавать соответствующее значение искомой функции:

```
i:-
   fact(A), fact(B),
   i(A,B,X),
   write(A," ",B," ",X), nl, fail.
```

Здесь первые две подцели обращаются к фактам определяющим возможные значения переменных A и B. Следовательно необходимо включить в программу описание этих фактов:

```
fact(0). fact(1).
```

Далее, третья подцель обращается к предикату для определения значения логической функции «И» по двум входным переменным A и B. После определения значения оно выводится на экран совместно с входными параметрами - `write(A,"  ",B,"  ",X)`. Далее реализуется перенос

строки и предикатом `fail` принудительно признается цель недостижимой, что приводит к рассмотрению альтернативных вариантов (для фактов `fact(A), fact(B)`).

Теперь организуем процедуру поиска значения логической функции «И» по двум входным переменным.

Самый простой вариант видимо выглядит так:

```
i(0,0,0).
i(0,0,1).
i(1,0,0).
i(1,1,1).
```

По сути это факты, перебирая которые пролог будет выдавать нам ответы. Однако, не всегда поставленная задача будет столь тривиальной. Давайте попробуем другие варианты оформления процедуры поиска значения функции.

Например, так:

```
i(A,B,1):-A>0,B>0,!.
i(A,B,0).
```

Или так:

```
i(A,B,X):-
      A+B=2,X=1,!;
      X=0.
```

Все это конечно интересно, но эти варианты создавались без учета последовательности унификации (сверху-вниз и слева-направо). Если перебор невелик, то можно и не уделять этому внимание. Но некорректно построенная процедура может стать камнем преткновения при большом количестве проверок и затянуть время унификации до неприлично больших величин.

Разберем подробнее. Итак, для исследуемой функции вариант с ответом «1» только один, но именно его мы и проверяем каждый раз в первом предложении процедуры. В то время как вариант с ответом «0» встречается в три раза чаще. Если его поставить на первое место, то отсечение будет сра-

батывать чаще, снижая тем самым общее число выполняемых действий:

```
i(A,B,X):-
        A+B<2,X=0,!;
        X=1.
```

Моя настоятельная рекомендация состоит в том, чтобы начиная с самых маленьких процедур вы учитывали тот факт, что в последующем они могут войти в другую более объемную программу и сильно испортить общую производительность.

А теперь задание для самостоятельного исполнения.

Разработайте и испытайте процедуры поиска и вывода на экран таких логических функций как: «НЕ», «ИЛИ». Их необходимо выполнить без использования стандартных (встроенных) предикатов.

В качестве помощи предлагаю использовать следующий скриншот (рис.19):

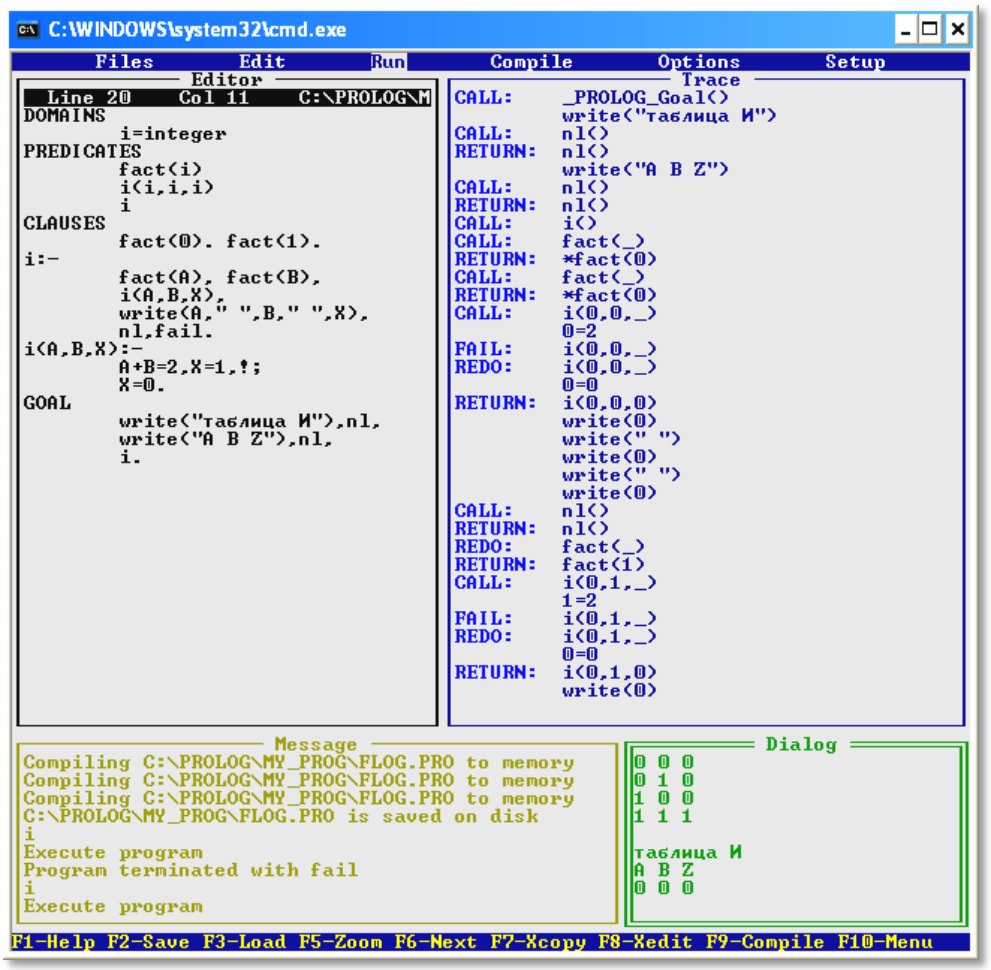


Рис.19. Трассировка процедуры, реализующей логическую функцию И.

Проверьте, как проходит пошаговое исполнение процесса унификации цели. Очевидно, что на данном скриншоте присутствует только часть трассировки.

Далее найдите в помощи Пролога описание логических функций в исполнении стандартных предикатов. Разработайте программу и исследуйте предикаты: `bitand`, `bitor`, `bitxor`, `bitnot`, `bitleft` и `bitright`.

На основе изученного материала самостоятельно разработайте следующую базу знаний.

Факты:

– пусть в виртуальном мире есть несколько объектов (например, фрукты, книги – разные, в количестве 5-8 шт.) – их наличие задать **фактами**;

– пусть в мире есть несколько субъектов (заданы именами, в количестве 4-5 субъектов) – **фактами** перечислены их предпочтения по отношению к объектам.

Правила:

1) найти по имени субъекта все объекты, входящие во множество его предпочтений;

2) найти по имени субъекта все объекты, не входящие во множество его предпочтений.

Запрос к базе знаний оформить в виде **внутренней цели**. При запуске программы на экран выводится сообщение с просьбой ввести имя субъекта и номер запроса (1 или 2, смотри описание правил выше), в ответ программа выводит на экран список объектов.

2.4.4. Имитация операторов выбора.

Оператор выбора в традиционном языке программирования позволяет обеспечить ветвление алгоритма как последовательности действий. В логическом программировании нет алгоритма, но ветвление можно организовать, управляя тем самым механизмом поиска решений (унификация и бэктрекинг). К операторам выбора относят оператор «if … then … else», позволяющий выбрать одну из двух альтернатив и «case», позволяющий организовать и более двух альтернатив. В Прологе возможна организация подобных логических структур путем применения стандартного предиката отсечения, который обозначается символом «!». Действие предиката «отсечение» заключается в стирании из стека точек воз-

врата всех альтернативных путей для унифицируемой в текущий момент цели. Если в процессе унификации цели Пролог встречается с предикатом «отсечение», то после унификации цели по текущему пути все остальные пути не будут рассматриваться при любом исходе унификации текущего пути.

Схематичное построение структуры «if … then … else» выглядит следующим образом:

```
P:-
   <условие>,!,V1. % если условие истинно, то V1
P:-
   V2.             % иначе - V2
```

Здесь P – некий предикат, а V1 и V2 – альтернативные варианты.

Рассмотрим использование такой структуры для организации предиката выбора максимального числа из двух возможных. Пусть идентификатор предиката будет max. Предикат будет описывать отношение между тремя аргументами – max(X,Y,Z), причем первый и второй аргументы входные числа, а третий – выходное значение, максимальное из двух входных.

Тогда можно создать процедуру, состоящую из двух предложений:

```
max(X,Y,Z):-
              X>Y,Z=X.
max(X,Y,Z):-
              X<=Y,Z=Y.
```

Первое предложение может быть успешно унифицировано только при истинности условия X>Y и при этом в Z присваивается X, а второе предложение, соответственно, при истинности условия X<=Y и при этом в Z присваивается Y. В данном случае наличие условий в каждом из предложений оправдано и обусловлено следующим обстоятельством. Ме-

ханизм унификации в случае, если цель внешняя, обеспечивает последовательный перебор всех возможных путей достижения цели. То есть при отсутствии условий в теле предложения оно будет унифицировано успешно безусловно. Например, для такой процедуры:

```
max(X,Y,Z):-Z=X.
max(X,Y,Z):-Z=Y.
```

при любом запросе будет два возможных решения – максимум будет равен X и Y. Если мы поставим условие только в одно из предложений, например в первое:

```
max(X,Y,Z):-X>Y,Z=X.
max(X,Y,Z):-    Z=Y.
```

то возможных исходов будет два:

1) максимум, как и ранее равен и X и Y – этот вариант возможен для случаев, когда на входе X>Y, тогда первое предложение процедуры будет унифицировано успешно, а второе будет успешным безусловно;

2) максимум равен Y – этот вариант возможен для случаев, когда действительно Y>X, так как в этом случае условие в первом предложении не будет истинным и первое предложение не будет унифицировано успешно.

Таким образом, только первый вариант организации процедуры, когда в обоих предложениях есть проверки условий будет корректно работать для любых допустимых значений аргументов. Однако это не есть имитация оператора «if … then … else», такую структуру можно повторить, если условие одно и проверяется оно один раз. В данном случае буквально нужно выполнить следующее: «если первое число больше второго, то первое число есть максимум, иначе – второе число есть максимум».

Избавиться от необходимости во втором предложении вновь проверять условие можно с использованием предиката «отсечение»:

```
max(X,Y,X):-X>Y,!.
max(_,Y,Y).
```

При унификации первого предложения проверяется условие X>Y. Если оно истинно, то срабатывает предикат «отсечение», который и убирает в стеке точек возврата информацию о наличии второго пути унификации (второго предложения). Унификация первого предложения завершается тем, что в третий аргумент помещается значение X.

Если условие X>Y ложно, то первое предложение не унифицируемо и Пролог переходит к альтернативному пути. Во втором предложении первый аргумент уже не имеет смысла, поэтому его можно обозначить анонимной переменной (символ «_»). Унификация второго предложения завершается тем, что в третий аргумент помещается значение Y.

Аналогичным образом можно организовать структуру многоальтернативного выбора (аналог оператора case в Паскале):

```
P:-
   <условие 1>,!,V1;
   <условие 2>,!,V2;
   . . .
   Vn.
```

Здесь используется комбинация предикатов «отсечение» и «;» (повторитель головы предложения). Повторитель головы используется просто для сокращения записи кода. В каждом пути стоит своё условие, при его истинности данный путь выполняется до конца, то есть реализуется соответствующий вариант многоальтернативного выбора. Если в завершающем альтернативном пути не поставить условие, то этот путь будет исполнять роль пункта «else» (иначе).

Изучите пример предиката, который определяет, что за день недели по его номеру:

```
day(D,Name):-
            D<=5,Name=”будний день”,!;
            D<=7, Name=”выходной день”,!;
            Name=”неверно задан номер дня недели”.
```

На основе изученного материала самостоятельно разработайте процедуру, которая позволит определить, как называть мужчину в зависимости от его возраста:

возраст, лет	0–8	9–14	15–20	21–60	61–…
наименование	дитя	отрок	юноша	мужчина	старик

2.5. Определение отношений на основе рекурсивных правил.

Описать мир можно задав отношения на основе фактов и/или правил. Например, фактами можно описать всех членов одной семьи или перечислить множество четных чисел в диапазоне от 0 до 99. Правилами можно пользоваться более универсально, отрешившись от конкретных значений аргументов и определив общую для всех зависимость. Например, любой человек в возрасте до 10 лет дитя или все числа, которые нацело делятся на 2, есть четные числа. Но в Прологе возможен ещё один замечательный подход к описанию мира через задание рекурсивных правил, то есть таких которые ссылаются в той или иной форме сами на себя. Например, предком можно считать как непосредственного родителя, так и родителя предка. Такой подход расширяет возможности универсального описания сходных по каким-либо свойствам подмножеств. Например, подмножество предков одного человека, которые являются последовательными предками друг друга или подмножество путей в лабиринте. В общем случае применение рекурсии оправдано при решении таких задач, для которых свойственно то, что решение задачи в целом

сводится к решению подобной же задачи, но меньшей размерности.

Для начала очень простой пример, только для того чтобы обозначить основные черты рекурсии. Пусть задан лабиринт из девяти клеток (см. рис). По светлым клеткам перемещаться можно, по иным нет. Необходимо определить существует ли путь от входа (точка «1») до выхода (точка «9»).

Для решения этой задачи опишем:

– совокупность возможных шагов по лабиринту как совокупность пар соседних свободных клеток (рис.20): `(1,4), (4,5), (5,6), (4,7), (6,9);`

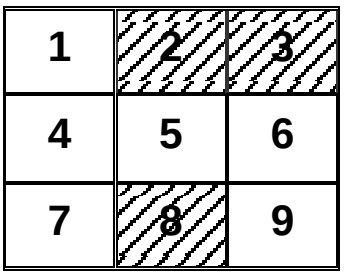


Рис.20. Схема моделируемого лабиринта.

– процедуру **«путь»**, состоящую из двух предложений:

1) путь из точки X в точку Y есть шаг из точки X в точку Y;

2) путь из точки X в точку Y есть шаг из точки X в точку Z и путь из точки Z в точку Y.

Тогда программа будет выглядеть следующим образом:

```
PREDICATES
   step(integer,integer)
   way(integer,integer)
CLAUSES
 % факты
 step(1,4). step(4,5).
 step(5,6). step(4,7).
 step(6,9).
 % правила
 way(X,Y):-step(X,Y).
 way(X,Y):-
      step(X,Z),
      way(Z,Y).
```

Проведем пошаговый разбор процесса унификации цели этой программой и выясним:

– зачем необходимо первое правило в процедуре `way`;

– как пользоваться трассировкой, чтобы понять процесс выполнения программы.

Для лучшего понимания процесса унификации рекурсивной процедуры предлагается изучить схему трассировки программы (см. рис.21) и саму трассировку (см. рис.22), на которой метками обозначены размышления описанные далее.

Разберитесь со схемой трассировки, так как само по себе грамотное составление трассировки подразумевает, что её составитель досконально разбирается в логическом программировании.

Итак, получив цель way(1,9) Пролог пытается её унифицировать (см. метку 1) последовательно просматривая факты и правила базы знаний. Среди фактов такого не обнаруживается, а вот среди правил оказывается есть кое-что и даже более – есть два правила, в заголовке которых стоит именно предикат way.

Подстановка цели в первое правило вынуждает Пролог проверять его выполнимость. Что состоится, если найдется факт step(1,9). Последовательно сопоставляя его со всеми фактами Пролог убеждается, что первое правило с указанными аргументами невыполнимо (метка 2).

Тогда Пролог движется далее по программе с надеждой ещё куда-нибудь пристроить первоначальную цель way(1,9). И сталкивается с правилом №2 (метка 3). Оно гласит: путь (1,9) существует, если есть шаг (1,Y) и путь (Y,9). То есть это правило выполнимо, если выполнимы последовательно обе подцели из тела предложения. Приходится их последовательно унифицировать. Унификация первой подцели step(1,Z) заканчивается при анализе первого же факта

step(1,4) и Z означивается значением «4» (метка 4). После чего происходит вызов второй подцели way(4,9).

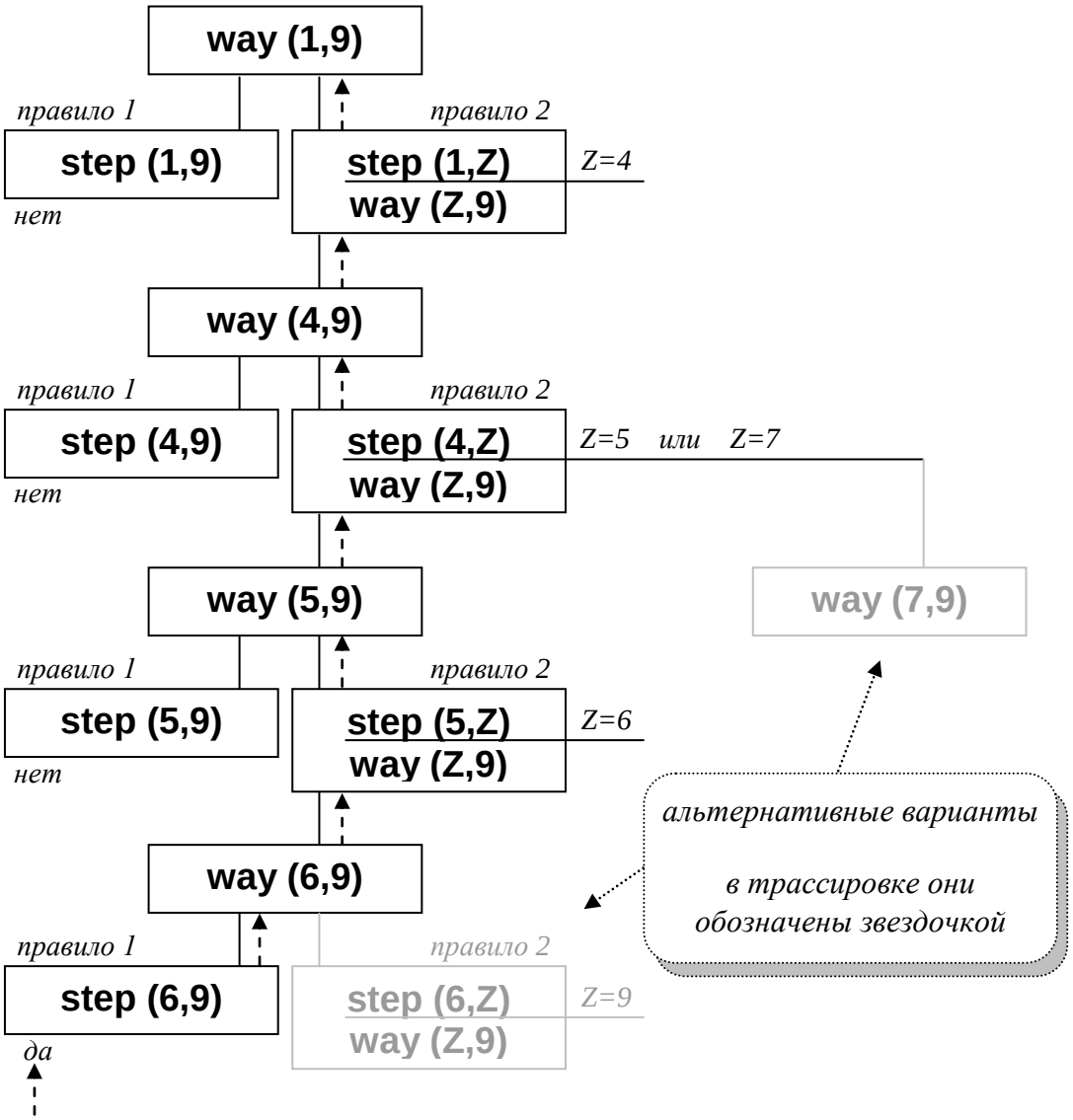


Рис.21. Схема трассировки программы.

Очевидно, что вторая подцель аналогична исходной цели за исключением значения первого аргумента. Поэтому процесс её унификации аналогичен описанному ранее. И именно поэтому последующие четыре метки в трассировке я обозначил теми же номерами с добавлением штриха (1’, 2’, 3’, 4’). Одно изменение вы можете заметить для метки 4’, а именно, присутствие звездочки в трассировке. Звездочка ука-

зывает, что у подцели есть еще альтернативы, к которым возможен возврат. В данном случае у step(4,5) есть альтернативный вариант step(4,7). И если по первому пути унификации основной цели программы успеха не будет достигнуто, тогда Пролог действительно вернется в эту «точку возврата» и продолжит поиски.

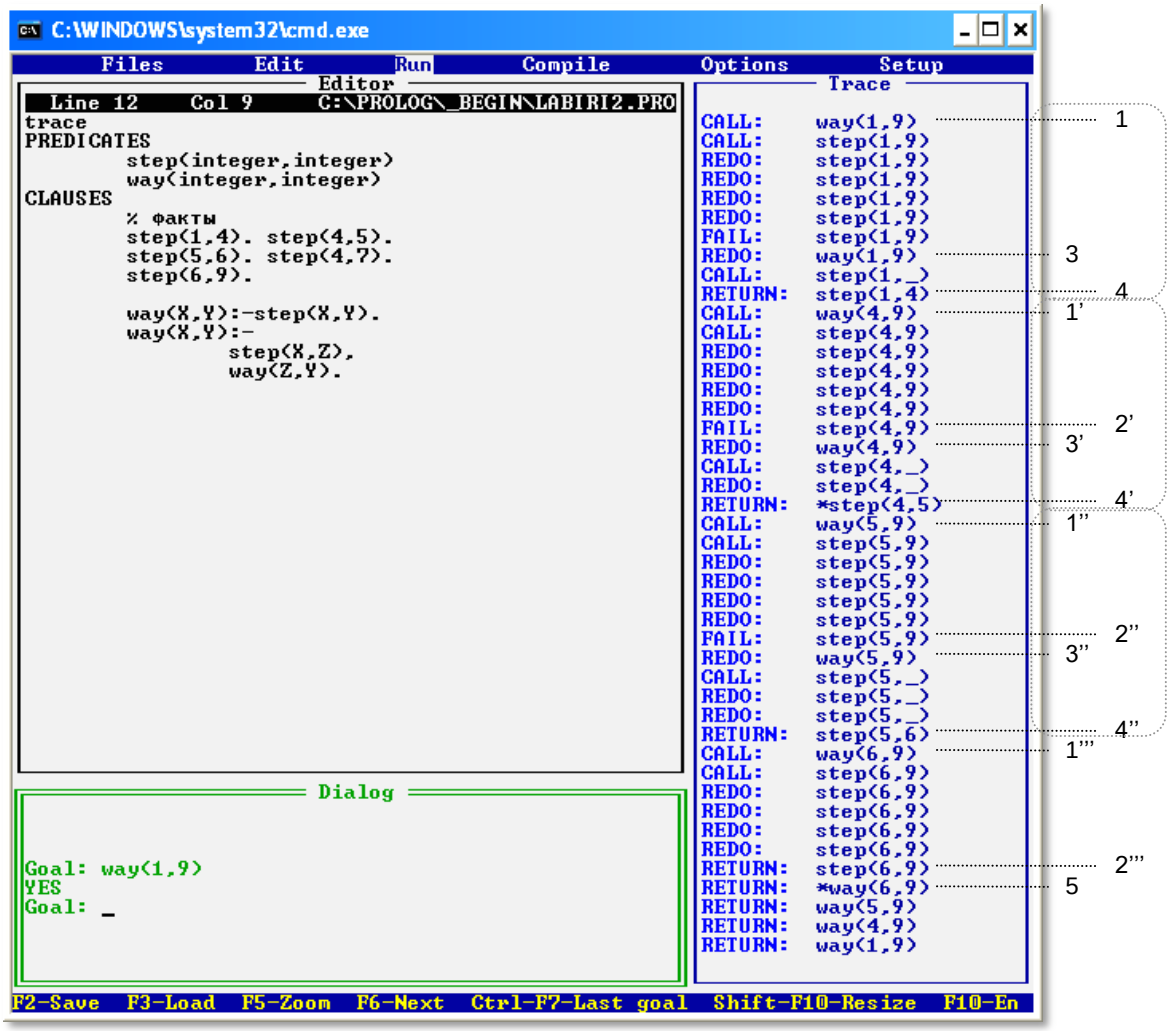


Рис.22. Трассировка программы «Поиск пути в лабиринте».

Далее аналогично происходит обработка подцели way(5,9) (метки 1'' … 4''), что как обычно заканчивается вызовом подцели way(6,9). Но далее уже встречаются отличия, так как правило №1 (*way(X,Y):-step(X,Y).*) с аргументами X=6 и Y=9 в этот раз оказывается выполнимым (метка 2'''). Про-

лог последовательно, но в обратном порядке собирает цепочку доказательств подцелей, отметив лишь, что для way(6,9) существует альтернатива в виде проверки второго правила (метка 5). И, в конце концов, объявляет Yes, что означает существование пути от точки 1 до точки 9.

Попробуйте самостоятельно запустить программу на доказательство цели way(4,S) и объясните результаты трассировки.

2.6. Списки

В различных языках программирования для хранения и обработки данных используют специальные структуры: массивы и множества. В языке Prolog есть подобная структура – список. Список есть упорядоченная последовательность элементов произвольной длины. Элементами списка могут быть переменные, константы или сами списки. Суть списка ближе к массиву, чем к множеству. В списке, как и в массиве, элементы могут повторяться, а также строго упорядочены. Однако в массиве к элементам можно обращаться по индексу, а в списке нет.

Список декларируется перечислением элементов через запятую в квадратных скобках. Ниже приведены примеры списков строк, целочисленных значений и символов:

```
["понедельник", "вторник", "среда"]
[1, 2, 3]
['п', 'в', 'с']
```

Кроме того существует возможность использовать пустой список: [ ]. При вводе обозначения пустого списка в редакторе Prolog между прямоугольными скобками не следует ставить пробел.

Работа со списками основана на расщеплении их на голову и хвост списка. Головой списка является первый его элемент. Хвост списка представляет собой список, состоящий из всех элементов исходного списка, за исключением первого его элемента. Операция разделения списка на голову и хвост в языке Prolog обозначается вертикальной чертой – делителем списка. Например, в процессе унификации такой структуры [Head|Tail] переменная Head будет сопоставлена с первым элементом списка, а Tail – с его хвостом. Процесс сопоставления может проходить и как сравнение и как присвоение. Кроме того в версии PDC Prolog существует возможность продекларировать до делителя списка более чем одно значение, например так: [A,B,C|Z]. При применении операции деления списка к списку, состоящему из одного элемента, в хвосте списка оказывается пустой список.

Например, в списке ['п', 'в', 'с'] компонент 'п' является головой списка, а ['в', 'с'] его хвостом. Последовательно отделяя от списка его голову можно обратиться к любому компоненту списка. В этом есть существенное отличие от массивов. В массиве можно непосредственно обратиться к любому компоненту по его индексу, а в списке только путем последовательного перебора. В языке логического программирования отсутствует такая алгоритмическая структура как «цикл», поэтому для обработки списков обычно организуют рекурсивную процедуру, состоящую как минимум из двух предложений – шаг и базис рекурсии.

Для того чтобы ввести список в разрабатываемую программу и использовать списки в предикатах удобно в разделе переопределения типов (DOMAINS) задать описание типа следующим образом:

```
listI = integer*     % список целых чисел
listR = real*        % список вещественных чисел
```

```
listC = char*        % список символов
lists = string*      % список строк
listL = listI*       % список, состоящий из списков целых чисел
```

Рассмотрим простейшие программы для работы со списками. Начнем с того, что исследуем программу, которая определяет значение головы списка:

```
DOMAINS
     i=integer
     li=integer*
PREDICATES
     pr(li,i)
CLAUSES
     pr([H|L],H).
```

Если данной программе в качестве внешней цели задать: `pr([1,2,3,4],S)`, то в процессе унификации Пролог сопоставит эту цель с единственным предложением в программе – `pr([H|L],H)` и, так как первый аргумент цели имеет значение, то процесс унификации сводится к присвоению. Первый аргумент предложения приобретает значение первого аргумента из цели. Обратите внимание, что первый аргумент предложения оформлен как список с делителем. Делитель в процессе унификации отделяет от списка голову и размещает её в переменной H, а хвост в переменной L. Таким образом, после унификации предложения переменные приобретают следующие значения: H=1 и L=[2,3,4]. Обратите внимание, что переменная H используется в предложении дважды: `pr([H|L],H)`. Это означает, что второй аргумент также имеет значение равное 1 и в переменную S возвращается значение 1. Это обычный способ получить голову списка.

Аналогичным образом устроен предикат для отсечения хвоста. Рассмотрим соответствующую программу:

```
DOMAINS
     li=integer*
PREDICATES
```

```
        pr(li,li)
CLAUSES
        pr([H|L],L).
```

Только теперь следует в разделе объявления предикатов указать, что оба аргумента являются списками: `pr(li,li)`. Если данной программе в качестве внешней цели задать: `pr([1,2,3,4],S)`, то в процессе унификации Пролог сопоставит эту цель с единственным предложением в программе – `pr([H|L],L)` и, так как первый аргумент цели имеет значение, то процесс унификации сводится к присвоению. Итогом унификации будет значение S=[2,3,4], полученное через переменную L.

Теперь ознакомимся с базовым способом обработки списком – рекурсивным способом. Разработаем процедуру проверки принадлежности некоторого значения списку. Для начала обсудим суть процедуры. Необходимо последовательно просмотреть все компоненты списка, сравнивая с заданным значением. Ввиду того что «цикла» нет, организуем итеративный процесс с помощью рекурсии, в каждом шаге которой будет отсекаться голова от списка.

```
member(X,[X|_]).
member(X,[_|T]):-member(X,T).
```

Здесь символ подчеркивание «_» означает анонимную переменную, то есть на данной позиции может быть аргумент с любым значением. Сформулируем вопрос к данной программе: `member(12,[9,12,0,44)`. Так как в вопросе нет переменных и есть только константы, то процесс унификации может закончиться с одним из двух исходов: успешно, что укажет нам на принадлежность первого аргумента списку, или неуспешно. Если бы в вопросе были переменные, то в процессе унификации внешней цели переменные приобретали бы значения и были бы возвращены в качестве результата.

В первом предложении проверяется, является ли искомый компонент головой текущего списка, расположенного на второй позиции. Если является, то унификация проходит успешно – значение принадлежит исходному списку. В противном случае процесс унификации переходит ко второму предложению. Во втором предложении от текущего списка отсекается голова и теряется в анонимной переменной, а хвост остается в переменной T. В теле второго предложения производится рекурсивный вызов предиката `member(X,T)`, что означает необходимость проверки – не принадлежит ли рассматриваемое значение оставшейся части списка. Таким образом второе предложение исполняет роль шага рекурсии, организуя последовательный перебор всего списка, а первое предложение предназначено для останова рекурсии в случае успешного сравнения значения с компонентом списка.

Подавляющее большинство процедур обработки списков основано на описанном выше подходе.

Глава 3. Динамические базы данных.

Внутренние базы данных так называются потому, что они обрабатываются исключительно в оперативной памяти компьютера, в отличие от внешних баз данных, которые могут обрабатываться на диске или в памяти. Так как внутренние базы данных размещаются в оперативной памяти компьютера, конечно, работать с ними существенно быстрее, чем с внешними. С другой стороны, емкость оперативной памяти, как правило, намного меньше, чем емкость внешней памяти. Отсюда следует, что объем внешней базы данных может быть существенно больше объема внутренней базы данных.

Однако использование внутренних баз данных при программировании на Прологе дает ряд дополнительных преимуществ.

Внутренняя база данных состоит из фактов, которые можно динамически, в процессе выполнения программы, добавлять в базу данных и удалять из нее, сохранять в файле, загружать факты из файла в базу данных. Эти факты могут использовать только предикаты, описанные в разделе описания предикатов базы данных.

```
DATABASE [ – <имя базы данных>]
<имя предиката>(<имя домена первого аргумента>,...,
< имя домена n-го аргумента>)
...
```

Пример, показывающий как можно задать внутреннюю базу данных для работы с фактами, представляющими из себя совокупность имени и фамилии студента:

```
DOMAINS
      s = string
DATABASE
```

```
        students(s, s)
```

Если раздел описания предикатов базы данных в программе только один, то он может не иметь имени. В этом случае он автоматически получает стандартное имя dbasedom. В случае наличия в программе нескольких разделов описания предикатов базы данных только один из них может быть безымянным. Все остальные должны иметь уникальное имя, которое указывается после названия раздела DATABASE и тире. Описание предикатов базы данных совпадает с их описанием в разделе описания предикатов PREDICATES.

Обратите внимание на то, что в базе данных могут содержаться только факты, а не правила вывода, причем используемые факты не могут содержать свободных переменных. Однако, есть существенное преимущество в использовании таких баз данных. Дело в том, что факты, использующие предикаты, заданные в разделе DATABASE, могут добавляться и удаляться во время выполнения программы. Прежде чем разбираться, в чем же состоит преимущество такой динамической базы данных, следует ознакомиться с основными способами работы с фактами внутренней базы данных.

3.1. Встроенные предикаты для работы с базами данных.

Давайте познакомимся со встроенными предикатами Турбо Пролога, предназначенными для работы с внутренней базой данных. Все рассматриваемые далее предикаты могут использоваться в варианте с одним или двумя аргументами. Причем одноаргументный вариант используется, если внутренняя база данных не имеет имени. Если же база поимено-

вана, то нужно использовать двухаргументный предикат, в котором второй аргумент – это имя базы.

Начнем с предикатов, с помощью которых во время работы программы можно добавлять или удалять факты базы данных.

Для добавления фактов во внутреннюю базу данных может использоваться один из трех предикатов:

```
assert
asserta
assertz
```

Разница между этими предикатами заключается в том, что предикат asserta добавляет факт перед другими фактами (в начало внутренней базы данных), а предикат assertz добавляет факт после других фактов (в конец базы данных). Предикат assert добавлен для совместимости с другими версиями Пролога и работает точно так же, как и assertz. В качестве первого параметра у этих предикатов указывается добавляемый факт, в качестве второго, необязательного – имя внутренней базы данных, в которую добавляется факт. Можно сказать, что предикаты assert и assertz работают с совокупностью фактов, как с очередью, а предикат asserta – как со стеком.

Для удаления фактов из базы данных служат предикаты:

```
retract
retractall
```

Предикат retract удаляет из внутренней базы данных первый с начала факт, который может быть отождествлен с его первым параметром. Вторым необязательным параметром этого предиката является имя внутренней базы данных.

Так, например, если база данных представлена фактами:

```
students("Сидоров","Павел")
students("Саакашвили","Михаил")
students("Ладен","Усама")
```

то удаление из базы студента Саакашвили может быть представлено предикатом:

```
retract(students("Саакашвили",_)),
```

а удаление Михаила так:

```
retract(students(_,"Михаил")).
```

Для удаления всех предикатов, соответствующих его первому аргументу, служит предикат retractall. Для удаления всех фактов из некоторой внутренней базы данных следует вызвать этот предикат, указав ему в качестве первого параметра анонимную переменную **`retractall(_)`**. Так как анонимная переменная сопоставляется с любым объектом, а предикат retractall удаляет все факты, которые могут быть отождествлены с его первым аргументом, все факты будут удалены из внутренней базы данных. Если вторым аргументом этого предиката указано имя базы данных, то факты удаляются из указанной базы данных. Если второй аргумент не указан, факты удаляются из единственной неименованной базы данных.

Предикат retractall рекурсивно удаляет все факты из базы и, по сути, может быть реализован с использованием встроенного предикат retract. Для этого следует создать рекурсивную процедуру с использованием отката при неудаче (fail) следующим образом:

```
del_all(Fact):-
                retract(Fact),
                fail.
del_all(_).
```

Попробуйте объяснить ***самостоятельно***, как работает данная процедура и в чем смысл второго предложения (**`del_all(_).`**).

Рассмотрим теперь один пример, который позволит осознать – в чем же преимущество именно ***динамической*** базы.

Эффективность программы можно поднять за счет добавления фактов в базу после их вычисления. Суть в том, что рекурсивно построенные программы зачастую производят много повторных вычислений, которых можно избежать, вычислив однажды значение и разместив его в базе как факт. Тогда, при повторной попытке унифицировать некоторую цель с аргументами уже нет необходимости производить все вычисления – Пролог просто проверит, нет ли в базе данных подходящего факта (а он там уже есть после добавления) и искомое значение просто унифицируется из факта (ранее добавленного).

Рассмотрим этот механизм на примере предиката, вычисляющего число Фиббоначи по его номеру. В данный момент будет уместна небольшая историческая справка.

Итальянский купец Леонардо из Пизы (1180-1240), более известный под прозвищем Фибоначчи, был, безусловно, самым значительным математиком средневековья. Роль его книг в развитии математики и распространении в Европе математических знаний трудно переоценить. В одном из научных трактатов Фибоначчи поместил следующую задачу: «некто поместил пару кроликов в некоем месте, огороженном со всех сторон стеной, чтобы узнать, сколько пар кроликов родится при этом в течении года, если природа кроликов такова, что через месяц пара кроликов производит на свет другую пару, а рождают кролики со второго месяца после своего рождения».

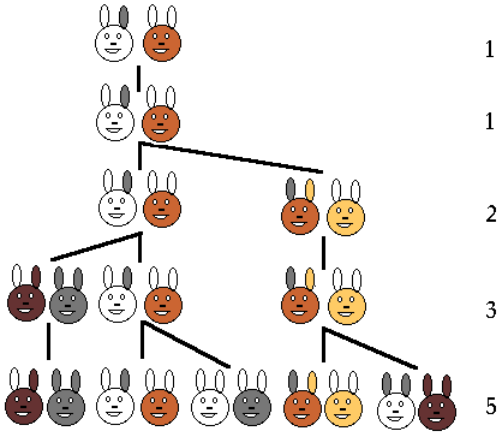


Рис.23. Схема размножения.

Ясно, что если считать первую пару кроликов новорожденными, то на второй месяц мы будем по прежнему иметь одну пару; на 3-й месяц - 1+1=2; на 4-й - 2+1=3 пары (ибо из двух имеющихся пар потомство дает лишь одна пара); на 5-й месяц - 3+2=5 пар (лишь 2 родившиеся на 3-й месяц пары дадут потомство на 5-й месяц); на 6-й месяц - 5+3=8 пар (ибо потомство дадут только те пары, которые родились на 4-м месяце) и т. д.

Таким образом, если обозначить число пар кроликов, имеющихся на n-м месяце через F_k , то F_1=1, F_2=1, F_3=2, F_4=3, F_5=5, F_6=8, F_7=13, F_8=21 и т.д., причем поток этих чисел регулируется общим законом:

$$\mathbf{F_n = F_{n-1} + F_{n-2}}$$

при всех n>2, ведь число пар кроликов на n-м месяце равно числу F_{n-1} пар кроликов на предшествующем месяце плюс число вновь родившихся пар, которое совпадает с числом F_{n-2} пар кроликов, родившихся на (n-2)-ом месяце (ибо лишь эти пары кроликов дают потомство).

Числа F_n, образующие последовательность 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ... называются " числами Фибоначчи", а сама последовательность - последовательностью Фибоначчи.

Итак, в последовательности Фибоначчи начиная с 3-го каждое следующее число получается сложением двух предыдущих. Но почему эта последовательность стала столь известной?

Дело в том, что при увеличении порядковых номеров отношение двух соседних чисел последовательности асимптотически стремится к некоторому постоянному соотношению. Так, если какой-либо член последовательности Фибоначчи разделить на предшествующий ему, результатом будет величина, близкая к 1,61803398875 – это отношение известно как «золотое сечение». В алгебре общепринято его обозначение греческой буквой φ (фи).

Принято считать, что объекты, содержащие в себе «золотое сечение», воспринимаются людьми как наиболее гармоничные. Пропорции пирамиды Хеопса, храмов, барельефов, предметов быта и украшений из гробницы Тутанхамона якобы свидетельствуют, что египетские мастера пользовались соотношениями золотого сечения при их создании. Начиная с Леонардо да Винчи, многие художники сознательно использовали пропорции «золотого сечения». Известно, что Сергей Эйзенштейн искусственно построил фильм Броненосец Потёмкин по правилам «золотого сечения». Он разбил ленту на пять частей. В первых трёх действие разворачивается на корабле. В двух последних – в Одессе, где разворачивается восстание. Этот переход в город происходит точно в точке золотого сечения. Да и в каждой части есть свой перелом, происходящий с учетом пропорции «золотого сечения». Эйзенштейн считал, что такие переходы воспринимаются как наиболее закономерные и естественные.

Тем не менее, существуют исследования показывающие, что значимость золотого сечения в искусстве, архитек-

туре и в природе преувеличена. При обсуждении оптимальных соотношений сторон прямоугольников (размеры листов бумаги А0 и кратные, размеры фотопластинок (6:9, 9:12) или кадров фотоплёнки (часто 2:3), размеры кино- и телевизионных экранов – например, 3:4 или 9:16) были испытаны самые разные варианты. Восприятие гармонии слишком индивидуально и не все принимают «золотое сечение» как оптимальное, удобное или комфортное.

Итак, последовательность чисел Фиббоначи можно задать таблицей:

порядковый номер	1	2	3	4	5	6	7	8	9	10
значение числа	1	1	2	3	5	8	13	21	34	55

или словами: первое и второе число равны единице, а каждое последующее есть сумма двух предшествующих. Формально это можно выразить так:

```
Число 1 = 1
Число 2 = 1
Число N = Число(N-1) + Число(N-2)
```

Напишем соответствующую процедуру:

```
fib(1,1):-!.                % первое число Фиббоначи равно единице
fib(2,1):-!.                % второе число Фиббоначи равно единице
fib(N,F) :-
       N1=N-1, fib(N1,F1),  % F1 это N-1-е число Фиббоначи
       N2=N-2, fib(N2,F2),  % F2 это N-2-е число Фиббоначи
       F=F1+F2.             % N-е число Фиббоначи равно их сумме
```

Недостаток такой организации предиката состоит в том, что при вычислении очередного N-го числа происходит многократное перевычисление предыдущих чисел Фиббоначи.

Изменим нашу программу следующим образом: добавим в нее раздел описания предикатов внутренней базы данных. В этот раздел добавим описание одного-единственного

предиката **db**, который будет иметь два аргумента. Первый аргумент – это номер числа Фиббоначи, а второй аргумент – само число.

```
DATABASE
        db(integer,real)
```

Сам предикат, вычисляющий числа Фиббоначи, будет выглядеть следующим образом. Базис для первых двух чисел Фиббоначи оставим без изменений. Для шага рекурсии добавим еще одно правило. Первым делом будем проверять внутреннюю базу данных на предмет наличия в ней уже вычисленного числа. Если оно там есть, то никаких дополнительных вычислений проводить не нужно. Если же числа в базе данных не окажется, вычислим его по обычной схеме как сумму двух предыдущих чисел, после чего добавим соответствующий факт в базу данных.

Попробуем придать этим рассуждениям некоторое материальное воплощение:

```
fib(1,1):-!.                % первое число Фиббоначи равно единице
fib(2,1):-!.                % второе число Фиббоначи равно единице
fib(N,F):-
       db(N,F),!.           % пытаемся найти N-е число Фиббоначи в
                              базе
fib(N,F):-                  % если в базе не нашли, то
       N1=N-1, fib(N1,F1),  % F1 это N-1-е число Фиббоначи
       N2=N-2, fib(N2,F2),  % F2 это N-2-е число Фиббоначи
       F=F1+F2,             % N-е число Фиббоначи равно их сумме
       assert(db(N,F)).     % добавляем вычисленное N-е число в базу
```

В ходе унификации предиката в оперативной памяти формируется такая база фактов:

```
db(2,2)
db(3,3)
db(4,5)
db(5,8)
db(6,13)
    …
```

Испытайте оба варианта реализации предиката вычисляющего числа Фиббоначи для достаточно больших номеров и почувствуйте разницу во времени работы. Только не ставьте сразу большие значения, иначе попросту не дождётесь ответа. Попробуйте для начала вычислить число Фиббоначи с порядковым номером 30 – разница во времени вычисления будет почти незаметна. Но если вы будете постепенно наращивать значения номера числа, то вскоре вы поймете насколько могут быть полезны динамические базы данных.

Для сохранения динамической базы на диске служит предикат **`save`**. Он сохраняет её в текстовый файл с именем, которое было указано в качестве первого параметра предиката. Если второй необязательный параметр был опущен, происходит сохранение фактов из единственной неименованной внутренней базы данных. Если было указано имя внутренней базы данных, в файл будут сохранены факты именно этой базы данных.

Факты, сохраненные в текстовом файле на диске, могут быть загружены в оперативную память командой **`consult`**. Первым параметром этого предиката указывается имя текстового файла, из которого нужно загрузить факты. Если второй параметр опущен, факты будут загружены в единственную неименованную внутреннюю базу данных. Если второй параметр указан, факты будут загружены в ту внутреннюю базу данных, чье имя было помещено во второй параметр предиката. Предикат будет неуспешен, если для считываемого файла недостаточно свободного места в оперативной памяти или если указанный файл не найден на диске, или если он содержит ошибки.

Заметим, что сохраненная внутренняя база данных представляет собой обычный текстовый файл, который мо-

жет быть просмотрен и/или изменен в любом текстовом редакторе. При редактировании или создании файла, который планируется применить для последующей загрузки фактов с использованием предиката **consult**, нужно учитывать, что каждый факт должен занимать отдельную строку. Количество аргументов и их тип должны соответствовать описанию предиката в разделе **database**. В файле не должно быть пустых строк, внутри фактов не должно быть пробелов, за исключением тех, которые содержатся внутри строк в двойных кавычках, других специальных символов типа конца строки, табуляции и т.д. Давайте на примере разберемся со всеми этими предикатами.

3.2. Разработка базы данных «Журнал учебной группы».

Итак, создадим простой журнал учебной группы, в котором будут содержаться факты вида: **Фамилия Оценка**. Если бы нужно было просто перечислить факты, которые не будут со временем меняться, то можно было бы их задать в разделе предложений. Но сейчас предстоит несколько иная задача – база должна иметь возможность пополняться и изменяться. Для этого следует естественно придумать предикат и описать его в разделе DATABASE. Например, так:

```
DATABASE
        ball(string, integer)
```

Предикат `ball` является двухаргументным. Предполагается, что в качестве первого аргумента будет вводиться ФАМИЛИЯ студента, а в качестве второго его ОЦЕНКА.

Кроме того, следует сразу предусмотреть, что база может быть при необходимости сохранена на жесткий диск и в последствии к ней можно будет обратиться с целью просмот-

ра информации об оценках или их коррекции. Для этого организуем файл для хранения динамической базы данных – `student.ddb`. Чтобы к нему обратиться следует использовать предикат **`consult`**, он загрузит с диска факты из обозначенного файла.

Создайте в текстовом редакторе Пролога файл **`student.ddb`**, который содержит такие данные:

```
ball("Петров",5)
ball("Сидоров",3)
ball("Шванидзе",4)
ball("Арутюнян",2)
```

Напишите программу:

```
domains
      s=string
      i=integer
database
      ball(s,i)
goal
      consult("student.ddb"),
      ball(E,R),
      write(E," ",R).
```

Если запустить её на исполнение, то ответ будет таким:

```
Петров 5
```

И всё, то есть, буквально, один и только один ответ. Так как цель внутренняя, то Пролог найдет только первый подходящий вариант унификации цели, а для перечисления всех вариантов необходимо предпринять дополнительные шаги. К примеру, модифицировать цель таким образом:

```
goal
      consult("student.ddb"),
      ball(E,R),
      write(E," ",R),nl,
      fail.
```

Попытка унификации стандартного предиката fail заканчивается неудачей, что приводит к инициализации механизма бектрекинга и откату к точке, в которой есть альтернативные пути – ball(E,R). В этом варианте исполнения при запуске программы получим исчерпывающий ответ:

```
Петров 5
Сидоров 3
Шванидзе 4
Арутюнян 2
```

Давайте модифицируем цель так, чтобы была возможность добавить факт в базу и сохранить её на жесткий диск:

```
goal
      consult("student.ddb"),
      write("Введите фамилию - "),nl,readln(F),
      write("Введите  оценку - "),nl,readint(O),
      assert(ball(F,O)),
      save("student.ddb").
```

Запустите программу с такой целью и введите новую фамилию и оценку. После исполнения программы просмотрите файл **student.ddb**. Вы обнаружите, что данные, которые вы вводили, были успешно сохранены и могут использоваться в дальнейшем.

Однако, программа должна быть более универсальной и позволять пользователю самому выбирать когда и что делать – необходимо МЕНЮ.

Создадим раздел описания предикатов и добавим туда два предиката **menu** и **m(char)**. Первый будет выводить на экран меню, а второй после выбора пользователем пункта выполнять соответствующее действие. Раздел описания типов данных и описания предикатов базы данных оставим без изменений, исправим внутреннюю цель и добавим раздел предложений следующим образом:

```
CLAUSES
menu:-
    clearwindow,          % очистка текущего окна
    write("1 - Получение оценки по фамилии "),nl,
    write("2 - Добавление новой записи "),nl,
    write("0 - Выйти"),nl,
    readchar(C),          % читаем символ с клавиатуры
    m(C).                 % вызываем выполнение пункта меню
m('1'):-
    clearwindow,
    write("Введите фамилию - "), nl,
    readln(N),            % ждём ввода фамилии
    ball(N, B),           % поиск в базе данных фамилии и
оценки
    write("Оценка: ",B), % если есть, выводим на экран
    readchar(_),          % ждём нажатия любой клавиши
    menu.                 % выводим на экран меню
m('2'):-
    clearwindow,
    write("Введите фамилию"),nl,
    readln(N),            % ждём ввода фамилии
    write("Введите оценку"),nl,
    readint(B),           % ждём ввода оценки
    assert(ball(N,B)),    % добавляем запись в базу данных
    menu.                 % выводим на экран меню
m('0'):-
    save("student.ddb"). % сохраняем базу данных
m(_):-
    menu.                  % если пользователь по ошибке нажал
                           не 0, 1, 2, то ещё раз будет ото-
                           бражено меню
```

Чтобы грамотно написать программу, необходимо предусмотреть две возможные ситуации:

1) если файл **student.ddb** существует, тогда загрузить его в память;

2) если файл ещё не существует, то его надо создать (то есть начать вести журнал группы).

Для описания этих исходов создадим безаргументный предикат **start** с соответствующими вариантами реализации в двух предложениях:

```
start:-
    existfile("student.ddb"),!,
    consult("student.ddb"),
```

```
    menu.
start:-
    openwrite(f,"student.ddb"),
    closefile(f),
    menu.
```

Не забываем, что процесс унификации любой цели происходит всегда последовательно сверху вниз и, если в качестве цели задать предикат **start**, то можно ожидать следующего процесса унификации.

В первом предложении проверяем существование файла **student.ddb**, если файл существует, то выполняется безусловный предикат «отсечение», который устраняет альтернативный путь – второе предложение не будет выполняться. После отсечения выполняется встроенный предикат **consult**, который загружает в оперативную память факты из базы данных. После чего управление передаётся в меню.

Если же файл **student.ddb** не существует, то первая подцель в первом предложении недостижима и первое предложение не может быть унифицировано. Это приводит к тому, что Пролог начинает искать альтернативные пути унификации цели **start** и переходит ко второму предложению. Во втором предложении создаётся файл с базой данных путём открытия файла для записи и последующего его закрытия. Такая процедура приведёт к созданию пустого файла, а в оперативной памяти не окажется никаких фактов относительной фамилий и оценок, так как фактов пока и не было. После чего управление также передаётся в меню.

Модифицируем цель программы:

```
GOAL
    start
```

Однако запустить такую программу не удастся, пока не будет определён тип переменной **f**. Следует в разделе описания типов данных (**DOMAINS**) добавить запись о том, что **f** является файловой переменной. Причем объявление файловой переменной происходит не совсем так, как объявление переменной любого другого типа – сначала назначается тип (файловый), а затем имя переменной (если их несколько, то переменные записываются через знак «**;**»):

```
file=f
```

Теперь программа полностью работоспособна.

Испытайте её в работе – добавьте несколько записей, посмотрите как меняется файл с данными, проверьте как программа ищет ответ на запрос «найти оценку по фамилии».

Для более детального понимания процессов поиска решения в динамической базе знаний настоятельно рекомендую выполнить следующее задание самостоятельно.

Модифицируйте описание предиката **m(char)** так, чтобы программа была способна выполнить такие запросы:

1) выдать на экран всю базу фамилий и оценок,
2) вывести оценку по фамилии,
3) вывести фамилию по оценке,
4) вывести все фамилии по заданной оценке,
5) удалить запись из базы данных по заданной фамилии,
6) вывести фамилии студентов, у которых оценка выше заданной.

Итоговую программу следует оформить с использованием предиката **makewindow** и так, чтобы она могла быть запущена без среды программирования Пролог.

Глава №4. Практика написания нетривиальных логических программ.

Напишем на Прологе логическую игру для двух участников – компьютер и человек.

Правила игры.

Играют двое. Цель игры – угадать четырёхзначное число противника. Перед началом игры каждый из игроков загадывает четырёхзначное число. Ограничения – в четырёхзначном числе не должно быть повторяющихся цифр (например, так правильно – **9413**, так не правильно – **4304**), ноль может стоять на первой позиции (например, **0123**). Выигрывает тот, кто отгадает число противника за меньшее количество ходов, при равном количестве ходов присуждается ничья.

Что такое ход и как делать ходы? Ход это четырехзначное число из неповторяющихся цифр (ноль может стоять на первой позиции). В ответ на ваш ход противник должен обозначить угаданные у него цифры. Делается это так: противник сравнивает свое число с вашим ходом, если есть цифра, которая присутствует в обоих числах, но ее положение в вашем ходе не соответствую загаданному противником, то она называется «коровой», а если цифра есть и стоит именно на позиции загаданной противником то «быком». Подсчитывается общее количество «коров» и «быков» и говорится ответ. К примеру, ответ один «бык» и две «коровы» следует понимать так: вы угадали три цифры из загаданного числа, но две из них стоят не на своем месте. Пусть ваш противник загадал:

1234, а вы сходили

0324

_ККБ *% тут я обозначил где «коровы», а где «бык»*

Тогда противник, поразмыслив, должен дать вам такой ответ: один «бык» и две «коровы». После чего уже он делает свой ход, а вы даете ответ о количестве «быков» и «коров». При грамотной игре неопределенность в отношении цифр и их позиций будет постепенно уменьшаться. Победа, естественно, будет засчитана при четырех «быках» с одной из сторон. Возможен и ничейный вариант, при случае, когда оба соперника угадали число за одинаковое количество попыток

Для большего понимания разберем пример одной игры. Я сыграл с программой, написанной в Excel`е на VBA, специально для сравнения процедурного и логического языков программирования. Результаты заносились программой в ячейки памяти непосредственно во время игры (см. рис.24).

номер хода	это моё число						тут число компьютера					
	1	9	7	2	быков	коров	X	X	X	X	быков	коров
1	2	3	8	0	0	1	1	2	3	4	0	2
2	3	6	9	5	0	1	5	6	7	8	1	0
3	8	9	1	4	1	1	5	3	2	0	1	2
4	8	5	4	7	0	1	5	0	4	3	2	1
5	0	9	7	1	2	1	5	1	0	3	4	0
6	1	9	7	2	4	0						
	эти ходы делал компьютер, пытаясь угадать число у меня				тут я ему отвечал		эти ходы делал я, пытаясь угадать число у компьютера				тут он мне отвечал	

Рис.24. Пример игры с компьютерной программой.

Перед игрой я загадал число (для простоты я взял год моего рождения) – 1972. Первым ходом компьютер предложил 2380 – я ему отвечаю: 0 «быков» и 1 «корова», так как цифра 2 является «коровой», а «быков» нет.

Я, опять же, для простоты собственных рассуждений сходил 1234 и получил ответ – 2 «коровы», «быков» нет. Не

знаю о чем «думал» компьютер, когда делал последующие ходы, но могу лишь прокомментировать свои.

Второй мой ход – опять же исходя из простоты рассуждений – 5678. Ответ компьютера – 1 «бык» – не самый удачный для меня исход, так как появилось неопределенность относительно двух цифр: 0 или 9 должны быть в числе.

В последующих ходах я делал упор на 0 и не прогадал. С третьего хода я основывался в своих рассуждениях, что быком является 5 – тут тоже на моей стороне была удача. В третьем ходе я проверял цифры из первого хода – там было число 1234 с результатом 2 «коровы» – и я решил проверить двойку и тройку, поменяв их местами – 5320. В результате получил ответ от компьютера – 1 «бык» и 2 «коровы» – ура! я на правильном пути, уже три цифры из числа, правда, пока не ясно какие именно :(

Поразмыслив, я оставил в числе 5, 3 и 0, а двойку заменил на четверку – опять угаданы три цифры :|

Ну и пятым ходом, заменив четверку на единицу, я добился успеха!!!

Алгоритм угадал моё число ходом позже…

Прежде чем писать логическую программу, сыграйте в эту игру с человеком, чтобы понять её суть и поразмыслить над будущим алгоритмом. Попробуйте самостоятельно определить, какие основные процедуры нужно будет заложить в логическую программу, чтобы она самостоятельно угадывала число. Проверьте свою готовность к решению нетрадиционных задач. Определите опытным путем какое максимальное число ходов до победы возможно при эффективной стратегии игры.

Я помогу вам написать эту программу на Прологе. Мы вместе создадим простейший дизайн, базовые процедуры и

стратегию решения, но часть предикатов вы сконструируете сами.

Кроме всего, основываясь на своем опыте, вы сможете усовершенствовать, предложенный ниже алгоритм. Замечу сразу, что окончательный объем текста программы займет меньше места, чем мое объяснение правил этой игры и примеры к ним. А если исключить из текста комментарии и убрать описательные и оформительские разделы программы, оставив только процедуры, относящиеся непосредственно к сути игры, то останется не более 500 символов. Как раз как в абзаце, который вы сейчас читаете. Это замечание сделано для того, чтобы подчеркнуть, что Пролог наиболее эффективен для анализа множества вариантов решения одной задачи.

Последующий текст будет идти от лица компьютерной программы, её противник – человек – будет называться ИГРОК.

Прежде всего, определимся с архитектурой программы.

В разделе типов данных (DOMAINS) достаточно определить только два – i=integer и li=integer*, так как мы будем работать только с целыми числами или списками цифр.

Следующий раздел – описание предикатов (PREDICATES). Что нужно иметь в программе, чтобы играть с человеком – иными словами, какие нужны предикаты и процедуры.

Назначение основных предикатов:

1) принимать четырехзначное число от игрока (xod_h(li)) и преобразовывать его в список цифр (num_li(i,li));

2) генерировать своё четырехзначное число из неповторяющихся цифр как список цифр (gen(li));

3) проверять подходит ли это число для того, чтобы сделать ход (usl(li));

4) к предикатам gen(li) и usl(li) нужен ещё обобщающий предикат, который, по сути, будет реализовывать ход компьютера xod_c(li) аналогично предиката для организации хода человека;

5) проверять является ли некоторая цифра элементом списка цифр (mem(i,li));

6) подсчитывать количество быков и коров (bk(li,li,i,i)) – первый список сравнивается со вторым и в третий аргумент заносится количество быков, а в четвертый – коров;

7) проверять условие – если у кого-то уже 4 быка, то останов (ost(i,i)) – первый аргумент – это количество быков после очередного хода у игрока, а второй – у компьютера;

8) нужен также предикат для осуществления принудительного повтора – repeat;

9) ну и, наконец, предикат, осуществляющий ход игры – start(li) – его задача, вести статистику игры, то есть выводить на экран последовательно ходы игрока и компьютера до достижения условия победы одного из участников. У предиката start(li) один входной аргумент – это список цифр числа, загаданного компьютером.

По пункту №3 потребуется небольшое отступление.

После некоторых размышлений о возможной формализации процесса угадывания в этой игре, я пришел к выводу, что компьютер вовсе не должен копировать логику человека. Иными словами, совсем необязательно перекладывать именно человеческий способ поиска решения в алгоритм. Для реализации угадывания по правилам этой игры можно ис-

пользовать как минимум два разных подхода кроме человеческого:

– один из них состоит в последовательном переборе всех подходящих четырехзначных чисел с неповторяющимися цифрами в диапазоне от 0123 до 9876 (этот способ реализован в программе, написанной на VBA);

– другой состоит в генерации случайного четырехзначного числа с неповторяющимися цифрами из того же диапазона (этот способ будет реализован на Прологе).

Оба способа предполагают последующую проверку сгенерированного числа на подходимость по количеству быков и коров к числам из предыдущих ходов. Для лучшего понимания рассмотрим пример для второго способа с генерацией случайного числа.

Пусть первым ходом компьютер сгенерировал 0274 (случайное четырехзначное число с неповторяющимися цифрами). На этот ход игрок дал ответ 0 быков и 2 коровы. Тогда в следующем ходу компьютер будет генерировать число до тех пор пока оно не будет подходить под все предыдущие (то есть пока только под число из первого хода – 0274). Иными словами, если число текущего хода именно и есть загаданное число, тогда и ответы предыдущих ходов к нему подходят. Так, например, если компьютер сгенерирует на втором ходу случайное число 1234, подразумевая, что оно и есть загаданное, то тогда ответ при предыдущем ходе (0274) был бы не 0 быков и 2 коровы, а 2 быка и 0 коров. Несостоявшиеся быки подчеркнуты в таблице:

ход	X	X	X	X	быков	коров
1	0	2	7	4	0	2
2	1	2	3	4		

Значит 1234 не подходит и нужно генерировать другое число. Пусть случайно вышло число 2465, проверка которого дает результат по ходу №1 именно – 0 быков и 2 коровы, поэтому этим числом можно ходить. Коровы подчеркнуты в таблице:

ход	X	X	X	X	быков	коров
1	0	2	7	4	0	2
2	2	4	6	5		

Пусть игрок дал на этот ход такой ответ – 2 быка и 2 коровы. На третьем шаге опять генерируем случайное число, пусть выпало такое – 2761 (см. табл. ниже). Необходимо его проверить на подходимость ко всем предыдущим числам. Проверка по первому ходу дает положительный результат, то есть если бы число 2761 было действительно загаданным числом, то при ходе 0274 ответ и был бы 0 быков и 2 коровы. Коровы подчеркнуты в таблице. Однако проверка второго хода дает отрицательный результат, а именно при загаданном числе 2761 во время второго хода был бы ответ 2 быка и 0 коров, что не соответствует действительности (2 быка и 2 коровы). Ячейки с быками взяты в рамочку – их действительно два, а коров нет совсем.

ход	X	X	X	X	быков	коров
1	0	2	7	4	0	2
2	2	4	6	5	2	2
3	2	7	6	1		

Итак, нужно генерировать другое число и оно вполне может оказаться таким – 2456. Проверка по первому ходу дает положительный результат, так как двумя коровами могли оказаться 2 и 4. Во втором ходе они уже стали быками, а еще

двумя коровами оказались 6 и 5, которые и были поменяны местами на третьем ходу:

ход	X	X	X	X	быков	коров
1	0	2	7	4	0	2
2	2	4	6	5	2	2
3	2	4	5	6		

Указанный порядок размышлений и соответствующих ходов можно продолжать до тех пор, пока не будет получен ответ – 4 быка.

Сам процесс загадывания пройдет ещё в разделе цели (GOAL), там же будет оформлено основное окно, в котором и будет протекать игра (makewindow(1,112,7," БЫКИ - КОРОВЫ ",0,0,25,42)):

```
GOAL
     gen(ZC),            % компьютер загадал свое число
     makewindow(1,112,7," БЫКИ - КОРОВЫ
",0,0,25,42),cursor(1,1),
     write("      ЧЕЛОВЕК              ПРОЛОГ"),nl,
     write("   ваш ход   Б  К     мой ход   Б  К"),nl,
     start(ZC).          % начнем с загаданным числом
```

Кроме основного окна будут использоваться ещё два: makewindow(2,110,7," ВАШ ХОД ",17,3,5,13) – для организации диалога по принятию хода (ход – это четырёхзначное число) от человека, и makewindow(3,111,7," НУ, ЧТО? ",17,19,6,21) – для организации диалога по принятию ответа от игрока на наш (компьютерный) ход.

Четырёхзначные ходы игрока можно не запоминать, а просто выводить на экран вместе с ответами о количестве быков и коров – для удобства в столбик. Они нужны игроку, чтобы он смог поразмышлять над следующим ходом. А вот

наши ходы и ответы игрока о количестве быков и коров надо куда-то записывать. Эти сведения понадобятся для обоснованной генерации последующих ходов.

Посмотрите на три последовательно сделанных копии экрана программы (рис.25), чтобы лучше понять ожидаемый результат.

Запоминать свои ходы (свои, т.е. компьютера) с результатами можно в виде фактов динамической базы данных. Факты о ходе игры можно записывать в базу, например, в таком виде **`x_c(li,i,i)`**, где первый аргумент это список из четырёх цифр (наш ход), второй аргумент – количество быков, третий аргумент – количество коров (не забывайте, что предварительно следует объявить такой предикат в разделе DATABASE).

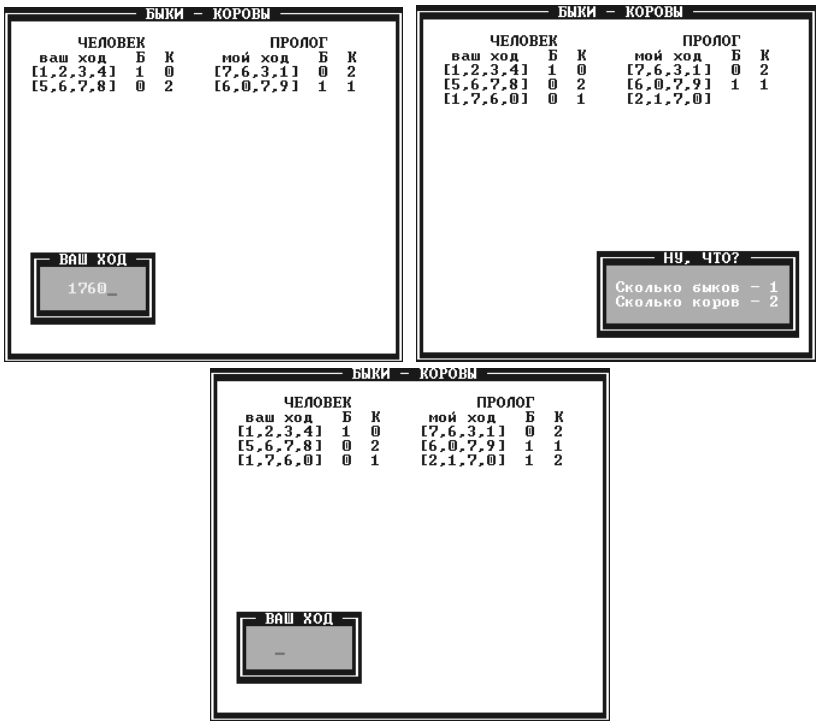


Рис.25. Пример реализации игры с программой на языке Пролог.

Например, таблица из трех ходов:

ход	X	X	X	X	быков	коров
1	0	2	7	4	0	2
2	2	4	6	5	2	2
3	2	4	5	6	4	0

может быть представлена тремя фактами:

```
x_c([0,2,7,4],0,2)
x_c([2,4,6,5],2,2)
x_c([2,4,5,6],4,0)
```

Итак, разделы DOMAINS, PREDICATES, DATABASE и GOAL уже есть. Осталось составить только раздел предложений. Часть из предложений я дам в готовом виде, вам останется просто их переписать в свою программу. Но оставшиеся вам придется составить самостоятельно – в этих местах я оставил соответствующие комментарии. Комментарии, присутствующие в тексте программы помогут вам разобраться с логикой предикатов. Для краткости написания быки обозначены как Б, коровы как К. Ниже приведены два текста раздела предложений с комментариями (листинг 1) и без них (листинг 2).

Текст с комментариями необходим для лучшего понимания логики программы. А для того чтобы не затягивать время вы можете набрать, дополнять и редактировать текст программы без комментариев (листинг №2).

После набора текста из листинга №2 вам останется только создать четыре предиката, причем предикат перевода числа в список цифр и предикат проверки принадлежности цифры списку цифр являются простейшими. Некоторые сложности у вас могут возникнуть только при создании предикатов генерации списка из четырех неповторяющихся цифр и проверки двух списков на количество быков и коров.

```
CLAUSES                                  % вариант программы с комментариями
start(ZC):-
   repeat,                               % повторяем пока кто-нибудь не наберет 4 Б
        xod_h(S2),                       % ждем хода от человека
        bk(ZC,S2,B,K),                   % вычисляем к-во Б и К
        write("  ",S2,"  ",B,"  ",K),    % отвечаем
        xod_c(HC),                       % ход компьютера
        write("    ",HC),                % выводим на экран
                                         % далее - ответ от человека о к-ве Б и К
        makewindow(3,111,7," НУ, ЧТО? ",17,19,6,21),
        cursor(1,1),write("Сколько быков - "),readint(VB),
        cursor(2,1),write("Сколько коров - "),readint(VK),
        assert(x_c(HC,VB,VK)),           % добавим в базу - ход компа, к-во Б и К
        removewindow,                    % удалаем окно НУ, ЧТО?
        write("  ",VB,"  ",VK),nl,       % записыв. к-во Б и К
        ost(B,VB),!,readchar(_).         % кто-то набрал 4 Б ?

ost(4,4):-nl,write("  НИЧЬЯ !"),!.
ost(4,_):-nl,write("  МНЕ ПРОСТО НЕ ПОВЕЗЛО ."),!.
ost(_,4):-nl,write("  МОЯ ВЗЯЛА !").

repeat. repeat:-repeat.                  % организуем принудительное повторение

% ______ процедура ХОД КОМПЬЮТЕРА - начало
xod_c(HC):-
   repeat,                               % пока не найдется подходящее число
        gen(HC),                         % генерируем число
        not(usl(HC)),!.                  % проверяем по всем ходам
usl(HC):-
        x_c(HCR,RB,RK),                  % берем из базы очередной ход
        not(bk(HC,HCR,RB,RK)).           % сравниваем HC и HCR по к-ву Б и К
% ______ процедура ХОД КОМПЬЮТЕРА - конец

% ______ процедура ХОД ЧЕЛОВЕКА - начало
xod_h(S2):-
        makewindow(2,110,7," ВАШ ХОД ",17,3,5,13),
        cursor(1,3),readint(M),          % ждем ввода 4-х значного числа M
        num_li(M,S2),                    % переводим его в список S2
        removewindow.                    % удаляем окно
% ______ процедура ХОД ЧЕЛОВЕКА - конец

% ______ процедура перевода 4-х значн. числа в список цифр - начало
% ______ num_li(+,–)                     num_li(integer,integer*)
% ______ эту процедуру составьте самостоятельно
% ______ процедура перевода 4-х значн. числа в список цифр - конец

% ______ процедура генерации четырех неповторяющихся цифр - начало
% ______ gen(–)                          gen(integer*)
% ______ эту процедуру составьте самостоятельно
```

```
% ______ процедура генерации четырех неповторяющихся цифр - конец

% ______ процедура проверки является ли цифра элементом списка - начало
% ______ mem(+,+)                    mem(integer,integer*)
% ______ эту процедуру составьте самостоятельно
% ______ процедура проверки является ли цифра элементом списка - конец

% ______ процедура для вычисления количества Б и К - начало
% ______ bk(+,+,–,–)                 bk(integer*,integer*,integer,integer)
% ______ эту процедуру составьте самостоятельно
% ______ процедура для вычисления количества Б и К - конец
```

листинг 1 / конец

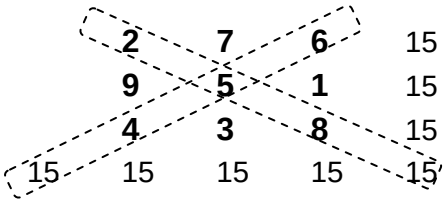
листинг 2 / начало

```
CLAUSES
start(ZC):-
   repeat,
         xod_h(S2),
         bk(ZC,S2,B,K),
         write("  ",S2,"  ",B,"  ",K),
         xod_c(HC),
         write("    ",HC),
         makewindow(3,111,7," НУ, ЧТО? ",17,19,6,21),
         cursor(1,1),write("Сколько быков - "),readint(VB),
         cursor(2,1),write("Сколько коров - "),readint(VK),
         assert(x_c(HC,VB,VK)),
         removewindow,
         write("  ",VB,"  ",VK),nl,
         ost(B,VB),!,readchar(_).

ost(4,4):-nl,write("  НИЧЬЯ !"),!.
ost(4,_):-nl,write("  МНЕ ПРОСТО НЕ ПОВЕЗЛО ."),!.
ost(_,4):-nl,write("  МОЯ ВЗЯЛА !").

repeat. repeat:-repeat.

xod_c(HC):-
   repeat,
         gen(HC),
         not(usl(HC)),!.
usl(HC):-
         x_c(HCR,RB,RK),
         not(bk(HC,HCR,RB,RK)).

xod_h(S2):-
         makewindow(2,110,7," ВАШ ХОД ",17,3,5,13),
         cursor(1,3),readint(M),
         num_li(M,S2),
         removewindow.
```

листинг 2 / конец

По данной главе учебно-методического пособия я рекомендую выполнить ряд нетривиальных заданий.

Задание №1. Написать программу построения магического квадрата размерностью 3x3. Магический квадрат – это квадратная таблица размерностью nxn, заполненная целыми числами от 1 до n таким образом, что сумма чисел в каждой строке, каждом столбце и на обеих диагоналях одинакова. Пример магического квадрата:



Задание №2. Написать программу в помощь кроссвордисту. Суть работы программы. Есть файл со списком русских слов (каждое слово в отдельной строке). После запуска программы пользователь получает запрос: «введите маску». Например, маска **о** означает, что из файла нужно выбрать все пятибуквенные слова, в которых на третьей позиции расположена буква «о» (знак «*» означает любой символ). Программа должна сформировать выходной файл, содержащий все слова, удовлетворяющие маске.

Задание №3. Написать программу, которая могла бы составить максимально длинное слово из списка букв. В исходном списке одна и та же буква может использоваться несколько раз, соответственно и в итоговом слове та же буква может встречаться более одного раза. Но количество таких букв в итоговом слове не должно превышать их количество в исходном списке. Так, если исходный список представлен та-

ким образом – [`о`,`о`,`к`,`т`,`л`,`м`], то слово «молот» соответствует условиям, а слово «молоко» – нет.

В упрощенном варианте эта программа должна составлять просто слово из всех букв списка. Например, апробируйте свою программу на трех таких задачках:

О	Ф	И
С	Я	Ф
О	Л	И

Е	Ц	С
П	Л	Е
Е	И	Н

И	К	А
М	Н	Э
О	О	К

Здесь каждая таблица содержит слово из 9 букв.

Глава 5. Построение продукционной экспертной системы.

Традиционно под экспертной системой понимают компьютерную программу, предназначенную для выдачи ответов по ограниченному ряду вопросов в рамках определенной предметной области знаний. Работа экспертной системы основана на модели знаний, как правило, продукционной или фреймовой.

Структурно экспертная система может быть представлена тремя самостоятельными блоками: базой знаний, машиной вывода и интерфейсом пользователя. Наименее технологичным является этап создания базы знаний ввиду сложности формализации знаний. Знания являются слабо структурированной или вовсе неформализованной информацией. Для того чтобы представить знания в продукционной форме и далее в форме предикатов, которые являются структурными единицами языка Пролог, знания нужно сначала хотя бы частично формализовать. На этом этапе часто приходится идти на компромисс, в чём-то упрощая и снижая достоверность модели предметной области. Преимуществом проектирования экспертной системы в среде Пролог является наличие встроенной в язык машины вывода, основанной на рассмотренных ранее механизмах унификации и бектрекинга. Таким образом, работа программиста в среде Пролог сводится к формированию базы знаний и, при необходимости, разработке интерфейса пользователя.

Рассмотрим подробнее процесс разработки продукционной экспертной системы на языке Пролог. База знаний продукционной экспертной системы содержит факты и правила. Правило с точки зрения продукционной модели знаний

содержит посылку и следствие. Постановка вопроса к продукционной экспертной системе инициирует процесс последовательного перебора фактов и правил из базы знаний и построения из них цепочки вывода. Промежуточные результаты сохраняются в оперативной памяти в фактах динамической базы данных.

В качестве предметной области знаний выберем парк комиссионных автомобилей. Пользователь экспертной системы будет иметь возможность получить информацию о наличии подходящего автомобиля по ряду его характеристик (атрибутов). В качестве опознавательных атрибутов выберем такие: страна производитель, тип привода, тип коробки передач.

Опишем конспективно структуру экспертной системы.

Начнем с инициации процесса поиска. Раздел внутренней цели должен содержать очистку динамической базы данных, чтобы результаты предыдущего анализа не накладывались на текущие, и вызов предиката поиска решения (отбор):

```
цель
    очистить Динамическую БД,
    отбор
```

В свою очередь предикат отбор будет иметь два пути для унификации. Первый путь инициализирует (предикат `авто`) поиск названия автомобиля по его атрибутам и, при успешной унификации, выводит найденное решение на экран. Второй – обеспечивает информирование пользователя при неудачном поиске:

```
отбор:-
  авто(Name),печать(Name);
  печать(нет вариантов).
```

Предикатом авто в базе данных описаны выставленные на продажу автомобили посредством декларации их атрибутов:

```
авто(Нива):-
   атрибут(отечественный,1),
   атрибут(повыш.прох.,1).
авто(Mazda 2):-
   атрибут(отечественный,2),
   атрибут(повыш.прох.,2),
   атрибут(спортивный,1).
```

Предикат для описания атрибутов содержит два аргумента: наименование атрибута и его значение, соответствующее автомобилю. Таким образом, уникальная совокупность атрибутов будет однозначно указывать на конкретный автомобиль. Совокупность предложений с предикатом авто в голове формирует базу данных об автомобилях, основанную на продукционной модели знаний. Успешная унификация предиката `авто(Name)` на любом из предложений такой базы приведет к возврату значения (название автомобиля) в вызывающий предикат, то есть в `авто(Name)` из предложения `отбор` с последующим выводом на экран этого значения.

Успешная унификация предиката `авто(A)` возможна только при успешной унификации всех входящих подцелей, то есть всех предикатов `атрибут`. Данный предикат обеспечивает непосредственный диалог с пользователем, задавая вопросы про атрибуты автомобиля и обрабатывая полученные ответы:

```
атрибут(Title,значение из описания авто):-
  взять значение из Динамической БД и сравнить,!;
  спросить про значение атрибута и сравнить.
```

Обратите внимание, что работа предиката `атрибут` основана на анализе фактов из динамической базы данных, где мы будем хранить ответы пользователя по поводу предпочитаемых им атрибутов автомобиля. Унификация предиката `атрибут` возможна по одному из двух путей. Если ранее пользователю уже задавался вопрос про значение текущего атрибута, то из динамической базы извлекается факт с соот-

ветствующим значением. В обратном случае инициализируется диалог с пользователем через предикат спросить. В любом случае после получения значения атрибута от пользователя оно сравнивается с аналогичным значением из описания автомобиля и, при их совпадении, предикат атрибут унифицируется успешно. Успешная унификация позволяет перейти к рассмотрению следующего атрибута данного автомобиля. При неуспешной унификации осуществляется переход к следующему автомобилю из базы.

Непосредственный диалог по поводу атрибута реализуем через считывание значения с клавиатуры и сохранения его в динамическую базу данных:

```
спросить(Title,значение):-
      печать(Title),
      считать(значение),
      добавить в Динамическую БД.
```

Второй аргумент в предикате спросить предназначен для того, чтобы возвращать значение атрибута, введенное с клавиатуры пользователем в вызывающий предикат атрибут.

После схематичного рассмотрения структуры экспертной системы можно перейти к её реализации на языке Пролог:

```
domains
  i=integer
  s=string
predicates
  otbor
  avto(s)
  at(s,i)
  ask(s,i)
database
  zn_at(s,i)
include "dbd.txt"
clauses
  at(A,B):-
    zn_at(A,K),!,B=K;
```

```
    ask(A,K),B=K.

  ask(A,B):-
    write(A,"? (1-да,2-нет) - "),
    readint(B),
    assert(zn_at(A,B)).

  otbor:-
    avto(Name),
    write(Name),nl,!.
  otbor:-
    write("нет вариантов").

goal
  clearwindow,
  retractall(_),
  otbor
```

Предикат `zn_at(s,i)` объявлен как факт динамической базы данных для того, чтобы во время диалога с пользователем была возможность сохранять сведения о предпочтениях пользователя.

Обратите внимание, что текст программы не содержит информации об автомобилях. Для удобства данная информация вынесена в отдельный файл `dbd.txt` и подключается в программу директивой `include`. Содержимое файла `dbd.txt` формально эквивалентно продукционной модели знаний про парк автомобилей:

```
constants
  v1="отечественный"
  v2="привод полный"
  v3="КПП автомат"
  v4="спортивный"
clauses
  avto("ВАЗ 2114"):-
    at(v1,1),at(v2,2).
  avto("Нива"):-
    at(v1,1),at(v2,1).
  avto("Santa Fe"):-
    at(v1,2),at(v2,1),at(v3,1).
  avto("Audi A6"):-
```

```
    at(v1,2),at(v2,2),at(v4,1).
```

Данную базу можно редактировать и пополнять независимо от текста программы экспертной системы, при этом программный код можно откомпилировать в исполняемый файл и запускать самостоятельно. Основной файл программы и файл с базой данных должны располагаться в одной папке. В меню среды программирования в разделе `Setup / Directories / Include directory` следует указать путь размещения файла `dbd.txt`.

Для удобства пользования экспертной системой интерфейс можно доработать следующим образом. Во-первых, имеет смысл использовать стандартный предикат makewindow (см. п.2.4.1). Во-вторых, необходимо усовершенствовать диалог таким образом, чтобы программа предлагала пользователю после поиска подходящего автомобиля выбор: выходить из программы или повторить поиск. Реализацию такого дополнения можно также вынести в отдельный модуль – файл `escape.txt`:

```
predicates
  repeat
  escape

clauses
  repeat.
  repeat:-repeat.

  escape:-
    write("Esc - выход, Enter - повторить"),
    readchar(D),
    char_int(D,27).
```

При унификации безаргументного предиката `escape` происходит считывание введенного с клавиатуры символа, который сохраняется в переменной D. Если индекс символа в таблице символов равен 27, что соответствует нажатой кла-

више Esc, то унификация предиката `escape` завершается успешно. Обратите внимание на тело предложения `start` в приведенной ниже итоговой программе:

```
domains
  i=integer
  s=string
predicates
  otbor
  avto(s)
  at(s,i)
  ask(s,i)
  start
database
  zn_at(s,i)

include "dbd.txt"
include "escape.txt"

clauses
  at(A,B):-
    zn_at(A,K),!,B=K;
    ask(A,K),B=K.

  ask(A,B):-
    write(A,"? (1-да,2-нет) - "),
    readint(B),
    assert(zn_at(A,B)).

  otbor:-
    avto(Name),
    write(Name),nl,!.
  otbor:-
    write("нет вариантов").

  start:-
    repeat,
    clearwindow(),
    retractall(_),
    otbor,nl,nl,
    escape.
```

```
goal
  makewindow(1,30,30," Диалог ",3,3,16,40),
  start
```

В предложении `start` подцель `escape` расположена на последнем месте. Таким образом, при успешной унификации подцели `escape` (при нажатии клавиши Esc) происходит завершение программы, а иначе срабатывает механизм бектрекинга и происходит откат на ближайшую точку возврата. В данном предложении она сформирована искусственно предикатом `repeat`. Дело в том, что для его унификации возможны два пути:

```
repeat.
repeat:-repeat.
```

Первый путь – безусловная успешная унификация, не содержащая никакого функционала, кроме собственно организации первого пути. Второй путь – рекурсия, необходимая для генерации альтернативного пути, унификация которого опять порождает два пути. И так происходит до тех пор пока пользователь не решит, что больше не жалеет работать с программой.

Глава 6. Хороший стиль написания логических программ.

Вы, наверное, уже заметили, что логические программы существенно отличаются по своей сути от программ, написанных на процедурных языках программирования. Процесс программирования, в свою очередь, представляет собой не интерпретацию алгоритма в синтаксисе языка программирования, а составление предложений, описывающих моделируемые объекты и отношения между ними.

Логические программы, при грамотном подходе, как правило, являются более легкими для написания, понимания и отладки по сравнению с программами, написанными на традиционных языках программирования. Естественно, этому способствуют встроенные механизмы унификации и бэктрекинга, а также ориентация программ на рекурсивную обработку данных.

Более всего язык логического программирования подходит для поиска решений в комбинаторных задачах, решения задач планирования, обработки баз данных, интерпретации языков, разработки экспертных систем, обработки символьной информации. В данных областях применения особенности языка декларативного программирования (`Prolog`) позволяют сокращать время разработки и отладки программ.

Эффективность логической программы, понимаемая как показатель расходования ресурсов памяти и времени выполнения, в б*о*льшей степени зависит от опыта написания программ, от используемого стиля программирования, от грамотного расположения отношений в тексте программы и от продуманной постановки запросов к программе.

Некоторые из способов рационального написания логических программ мы рассмотрели в ходе изучения других приемов программирования. В частности, это:

– отсечение, как повышение эффективности поиска путем предотвращения ненужного перебора с возвратами и останова процесса обработки ненужных альтернатив на самых ранних этапах;

– поиск лучшего упорядочения предложений процедур и целей в телах предложений;

– использование для представления объектов в программе более подходящих структур данных, для того, чтобы можно было реализовать операции над объектами более эффективно;

– использование динамических баз данных для хранения промежуточных результатов, которые могут понадобиться при проведении дальнейших вычислений (тогда вместо повторения вычислений можно просто выполнить выборку результатов из базы фактов).

Некоторых затруднений декларативного программирования удаётся избежать при соблюдении ряда правил оформления текста программы, что можно также отнести к стилю программирования. Наиболее полно особенности декларативного стиля программирования изложены в работе И.Братко [3], посвященной анализу процесса разработки алгоритмов искусственного интеллекта на языке Prolog. Приведём здесь краткий обзор и анализ рекомендаций, ориентация на которые позволит не только писать более эффективные программы, но и облегчит сам процесс программирования.

В большинстве случаев следует стремиться к тому, чтобы предложения программы были короткими. При этом тело предложения, как правило, не должно содержать много под-

целей. При большом количестве подцелей в теле предложения оправданным будет подход, при котором ряд подцелей выделяются в отдельное предложение посредством соответствующего предиката.

Имена предикатов и переменных предпочтительнее выбирать таким образом, чтобы они отражали сущность объектов и смысл отношений между ними в модели предметной области базы знаний.

Для повышений удобства чтения и разбора логики следует придерживаться способа записи предложений в программе в виде определенной структуры. Прежде всего, к этому относятся: определенные правила расстановки пробелов, ввода пустых строк и отступов. Каждая цель в предложении может быть помещена на отдельной строке. Между предложениями должны находиться пустые строки. Исключение составляют случаи, когда предложения представляют собой многочисленные факты, касающиеся одного и того же отношения или предложения относятся к одной и той же процедуре.

Нет особой необходимости придерживаться одного стиля во всех программах. Применяемые стилистические соглашения могут зависеть от конкретной программы, поскольку их выбор диктуется рассматриваемой задачей и личным вкусом. Но важно то, чтобы одни и те же соглашения неизменно использовались во всей программе.

Использование динамических баз данных иногда бывает очень удобным и значительно упрощает программирование, однако, без дополнительных комментариев использование предикатов **`assert`** и **`retract`** способно затруднить понимание поведения программы. В частности, при использовании этих предикатов может оказаться, что одна и та же про-

грамма в разное время отвечает на одни и те же вопросы поразному. Если в подобных случаях необходимо воспроизвести такое же поведение, как и прежде, то следует обеспечить полное восстановление предыдущего состояния программы, которая была модифицирована в результате внесения и извлечения предложений из базы данных с применением этих предикатов.

В результате использования точки с запятой смысл предложения может стать менее очевидным; иногда удобство программ для чтения может быть повышено путем разделения предложения, содержащего точку с запятой, на несколько предложений. Но возможно также, что это улучшение будет достигнуто и за счет увеличения длины программы или даже за счет снижения ее эффективности.

Существенно проще устраняются ошибки при написании программ в таком стиле.

Хороший стиль также подразумевает наличие комментариев в тексте программы.

Комментарии должны, прежде всего, пояснять, для чего предназначена программа и как её использовать, и только после этого представлять подробные сведения об используемом методе решения и других нюансах программирования. Основное назначение комментариев состоит в том, чтобы предоставить пользователю возможность эксплуатировать программу, понять ее и, возможно, модифицировать. Комментарии должны описывать в наиболее краткой из возможных форм всё, что для этого необходимо.

Широко распространенной ошибкой является недостаточное применение комментариев, но в программе может также оказаться и слишком много комментариев. Изложение сведений, очевидных и без комментариев при изучении кода

самой программы, создает лишь ненужную нагрузку для тех, кто изучает эту программу.

Продолжительные комментарии должны предшествовать коду, к которому они относятся, а короткие комментарии следует чередовать с самим кодом.

Обязательным для комментирования являются:

– назначение программы, способы её использования (например, какая цель должна быть вызвана и каковы ожидаемые результаты), а также примеры использования;

– назначение предикатов в программе;

– параметры этих предикатов;

– обозначения входных и выходных параметров (напомню, что входными называются такие параметры, которые при вызове предиката имеют значения);

– основные ограничения программы.

Иногда возникает необходимость делать ссылки на предикаты. При этом имя предиката следует указывать вместе с его арностью. Например, ссылка на предикат **`merge(List1,List2,List3)`** может быть представлена как **`merge/3`**. Арность следует указывать, так как в одной программе можно использовать одно и тоже имя для обозначения разных отношений, если их арность различается.

Типы входных/выходных параметров обозначаются путем указания перед именами параметров префикса "+" (входной) и "–" (выходной). Например, обозначение **`merge(+,+,–)`** указывает, что первые два параметра **`merge`** являются входными, а третий – выходным.

Рассмотрим пример оформления программы поиска наибольшего общего делителя для двух чисел:

```
DOMAINS
    i=integer
```

```
PREDICATES
     nod(i,i,i)
     del(i,i,i,i,i)
CLAUSES
                                % nod – наибольший общий делитель двух чисел
                                % пример: nod(18,12,X), X=6
                                % пример: nod(12,180,X), X=12
                                % nod(+,+,–)
                                % del(+,+,~,~,–) – вспомогательный
                                % Арг3 – счетчик, Арг4 – текущий ответ
                                % если Арг3=N1 или N2, то Арг4 –> Арг5
nod(N1,N2,Z):-
     del(N1,N2,1,1,Z).

del(N1,N2,T,X,X):-
     T=N1,!;T=N2,!.             % если текущий делитель = N1 или N2,
                                % то X – ответ и останов
del(N1,N2,T,X,L):-
     D=T+1,                     % счетчик
     0=N1 mod D,
     0=N2 mod D,!,              % если, D делит нацело и N1 и N2,
     del(N1,N2,D,D,L).          % то это новый общий делитель – Арг4=D
del(N1,N2,T,X,L):-
     D=T+1,                     % счетчик
     del(N1,N2,D,X,L).          % иначе: проверяем следующее число D
```

Есть различные способы организации процедуры поиска наибольшего общего делителя, но в данной программе мы воспользуемся самым примитивным – простым перебором на основе рекурсии. В данном случае нас интересует не столько изучение способов поиска наибольшего общего делителя, сколько порядок оформления программы.

В самом начале раздела предложений (**CLAUSES**) находятся комментарии, содержащие описание программы в такой последовательности:

1) назначение основного предиката;

2) примеры его использования;

3) входные и выходные аргументы (знак «**+**» – это входной аргумент, знак «**–**» – выходной, знак «**~**» – аргумент, меняющий своё значение по ходу рекурсии);

4) вспомогательный предикат;

5) краткий смысл его работы (знак «**->**» означает «передается», иными словами запись «**Арг4 -> Арг5**» означает, что аргументу №5 присвоили значение аргумента №4).

Очевидно, что предложенный выше алгоритм, является в буквальном смысле алгоритмом, что не вписывается в технологию логического программирования, основанную на декларации фактов и правил. Для поиска наибольшего общего делителя есть более эффективный способ:

```
CLAUSES
   nod(X,X,X):-!.
   nod(X,Y,D):-
              X<Y,!,
              Y1=Y-X,
              nod(X,Y1,D);
              nod(Y,X,D).
```

Попробуйте самостоятельно перевести смысл этой процедуры с языка Пролог на русский язык.

Заключение

Материал данного учебного пособия составлен на основе лабораторного практикума по дисциплине «Функциональное и логическое программирование».

Парадигма логического программирования существенно отличается от повсеместно используемого в императивных языках алгоритмического подхода. Логическая программа не содержит последовательности инструкций, предписывающих порядок выполнения вычислений, а, по существу, является базой знаний. Порядок работы с такой базой подразумевает корректное формулирование к ней вопросов и автоматизированный вывод решений. Механизм вывода уже встроен в среду программирования, а от программиста требуется лишь эффективно им управлять.

На начальном этапе освоения логического программирования наиболее сложным для программиста является отказ от алгоритмического мышления в пользу декларативного. Следует отметить, что на современном этапе развития компьютерной техники полный переход к декларативному программированию невозможен ввиду того, что сами компьютеры работают последовательно, а от порядка записи и, соответственно, унификации фактов и правил в логической программе существенно зависит исход поиска решений. Язык логического программирования эффективнее всего может быть применен для решения комбинаторных задач, написания систем поддержки принятия решения, разработки экспертных систем, моделирования интеллектуальной деятельности.

Библиографический список

1. Шрайнер П.А. Основы программирования на языке Пролог: курс лекций: учеб. пособие для студентов вузов, обучающихся по специальностям в обл. инф. технологий. - М.: Интернет-Ун-т Информ. Технологий, 2005. - 176 с.

2. Ездаков А.Л. Функциональное и логическое программирование: учебное пособие. - М.: БИНОМ. Лаборатория знаний, 2009. - 119 с.

3. Братко Иван. Алгоритмы искусственного интеллекта на языке Prolog, 3-е издание.: Пер. с англ. - М.: Издательский дом "Вильямс", 2004. - 640 с.

4. Аляев Ю.А., Тюрин С.Ф. Дискретная математика и математическая логика: учебник. - М.: Финансы и статистика, 2006. - 368с.

5. Хоггер К. Введение в логическое программирование: Пер. с англ. -М.: Мир, 1988. - 348с.

6. Джордж Ф. Люгер Искусственный интеллект: стратегии и методы решения сложных проблем, 4-е издание.: Пер. с англ. - М.: Издательский дом "Вильямс", 2005. - 864с.

7. Стюарт Рассел, Питер Норвиг Искусственный интеллект: современный подход, 2-е изд..: Пер. с англ. - М.: Издательский дом "Вильямс", 2006. - 1408 с.

Приложения

Приложение №1. Горячие клавиши.

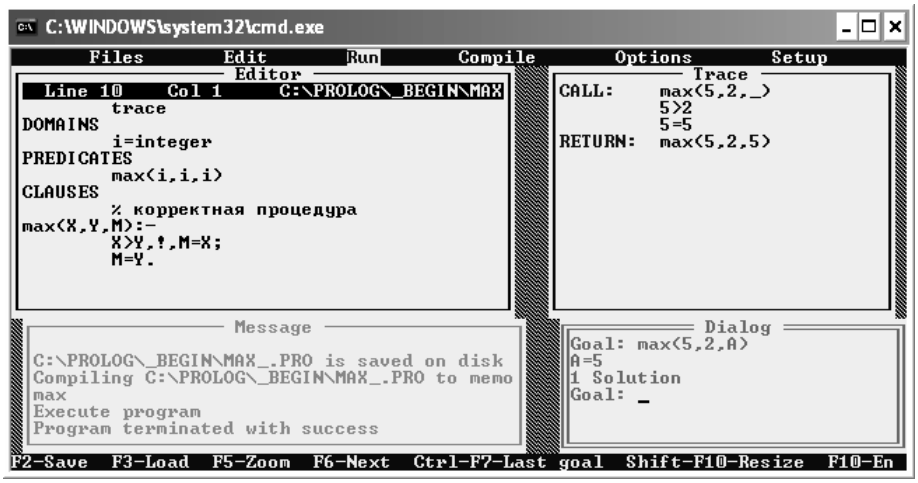
Показать горячие клавиши	`Alt-H`
Помощь по системе команд Prolog	`Shift-F1`
Выход из Prolog	`Alt-X`
Загрузить файл	`F3`
Загрузить файл из списка последних	`Alt-F3`
Сохранить файл	`F2`
Запустить программу на исполнение	`Alt-R`
Активировать меню ФАЙЛ	`Alt-F`
Активировать редактор программ	`Alt-E`
Активировать меню ОПЦИИ	`Alt-O`
Активировать меню УСТАНОВКИ	`Alt-S`
Активировать меню ФАЙЛ	`Alt-F`
Показать информацию о системе Prolog	`Alt-V`
Текущее окно в полный экран/обратно	`F5`
Перейти в следующее окно	`F6`
Открыть вспомогательный файл	`F8`
Вернуться в основной	`Esc`
Редактировать содержимое буфера	`Ctrl-F8`
Режим редактора – “запись поверх”	`Ins`
Авто отступ при нажатии Enter	`Alt-I`
Текущее слово в верхний регистр	`Ctrl-B + Ctrl-U`
Текущее слово в нижний регистр	`Ctrl-B + Ctrl-L`
Сменить регистр текущего слова	`Ctrl-B + Ctrl-R`
Сменить текущую директорию	`Ctrl-K + Ctrl-L`
Копировать блок из файла	
.....сначала выбираем файл:	`F7 …`
.....затем выделяем блок:	`… Ctrl-K + Ctrl-R …`
.....затем вставляем с позиции курсора:	`… Ctrl-K + Ctrl-R …`
Поиск фразы	`Ctrl-F3 … фраза … Ctrl-F3`
Поиск следующей фразы	`Shift-F3`
Заменить	`Ctrl-F4`
Заменить снова	`Shift-F4`
Прервать выполнение программы	`Ctrl-Break`

Cursor movement		
Line up	↑	Ctrl-E
Line down	↓	Ctrl-X
Left	←	Ctrl-S
Right	→	Ctrl-D
Word left	Ctrl-←	Ctrl-A
Word right	Ctrl-→	Ctrl-F
Start of line	Home	Ctrl-Q Ctrl-S
End of line	End	Ctrl-Q Ctrl-D
Start of page	Ctrl-Home	Ctrl-Q Ctrl-E
End of page	Ctrl-End	Ctrl-Q Ctrl-X
Scroll up		Ctrl-W
Scroll down		Ctrl-Z
Page up	PgUp	Ctrl-R
Page down	PgDn	Ctrl-C
Start of text	Ctrl-PgUp	Ctrl-Q Ctrl-R
End of text	Ctrl-PgDn	Ctrl-Q Ctrl-C
Previous position		Ctrl-Q Ctrl-P
Goto line	Ctrl-F2	
Goto position	Shift-F2	
Goto blockstart		Ctrl-Q Ctrl-B
Goto blockend		Ctrl-Q Ctrl-K

Insert & Delete		
Insert new line		Ctrl-N
Backspace	Ctrl-H	Ctrl-H
Delete character	Del	Ctrl-G
Delete word		Ctrl-T
Delete to start of line		Ctrl-Q Ctrl-T
Delete to end of line		Ctrl-Q Ctrl-Y
Delete line	Ctrl-BackSpace	Ctrl-Y

Block functions		
Block select		Ctrl-K Ctrl-M
Copy block to pastebuffer		Ctrl-K Ctrl-I
Move block to pastebuffer		Ctrl-K Ctrl-Y
Paste	Ctrl-F7	Ctrl-U
Mark blockstart		Ctrl-K Ctrl-B
Mark blockend		Ctrl-K Ctrl-K
WordStar show/hide block		Ctrl-K Ctrl-H
WordStar copy block		Ctrl-K Ctrl-C
WordStar move block		Ctrl-K Ctrl-V
MultiMate copy block	Ctrl-F5	
MultiMate move block	Alt-F6	
MultiMate delete block	Alt-F7	

Сразу после запуска среды программирования Пролог (файл PROLOG.EXE) вы можете увидеть примерно такое окно:



Верхнюю строчку экрана занимает Меню (File, Edit, Run, Compile, Options, Setup), названия пунктов которого говорят сами за себя. Вход в соответствующие пункты меню через сочетания клавиш:

Активировать меню ФАЙЛ.....Alt-F

Активировать редактор программ.....Alt-E

Активировать меню ОПЦИИ.....Alt-O

Активировать меню УСТАНОВКИ.....Alt-S

Выполните следующие задачи.

1. Изучите содержимое и назначение опций меню.

2. Установите режим компиляции в память компьютера.

3. Сконфигурируйте положение, размеры и цветовые палитры окон.

4. Определите пути доступа к файлам, установив нужные для вас каталоги.

5. Выполненные установки запишите в файл конфигурации.

Цветовая палитра окон устанавливается в меню Setup / Colors.

Размеры и положение окон устанавливаются посредством выбора меню Setup / Windows size – выбранное окно становится активным, после чего можно «стрелочками» основной на клавиатуре регулировать размеры окна, а положение – «стрелочками» на дополнительной клавиатуре.

Вторую строчку экрана занимает служебная строка, где отображаются:

– текущая позиция курсора в тексте программы,

– путь к файлу программы,

– режим отступа,

– режим ввода с клавиатуры.

Ниже идут окна Пролога (переключение между окнами – клавиша F6):

1) редактор программы (Edit) – вход в окно Alt-E;

2) окно сообщений (Message);

3) окно диалога / выполнения программы (Dialog) – вход в окно Alt-R;

4) окно трассировки программы (Trace).

Активировать редактор программ (перейти в режим редактирования) – сочетание клавиш Alt-E. Другие особенности среды программирования можно изучить вызвав справку Пролога нажатием клавиши F1. Выход из справки – Esc.

Выбрать язык ввода (английский/русский) в среде Windows можно нажатием сочетаний клавиш: левый Ctrl-Shift / правый Ctrl-Shift.

Учебное издание

Беляков Андрей Юрьевич

ПРАКТИКА ЛОГИЧЕСКОГО ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ ПРОЛОГ

Учебное пособие

Подписано в печать 00.00.00.
Формат 60×84 1/16. Усл. печ. л. 0,00.
Тираж 00 экз. Заказ №

ИПЦ «ПрокростЪ»
Пермской государственной сельскохозяйственной академии
имени академика Д.Н. Прянишникова,
614990, Россия, г. Пермь, ул. Петропавловская, 23
тел. (342) 210-35-34