Министерство образования и науки Украины
Севастопольский национальный технический университет

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ **PROLOG**

методические указания

к выполнению лабораторных работ

по дисциплине «ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ»

для студентов специальности «**7.091501-**
«Компьютерные системы и сети»

Севастополь

2007

УДК 681.3

Методические указания к лабораторным занятиям по дисциплине «Логическое программирование» на тему «Основы программирования на языке Prolog» для студентов специальности «7.091501-Компьютерные системы и сети» всех форм обучения. / Сост. А.А. Брюховецкий - Севастополь: Изд-во СевНТУ, 2007. - 53с.

Цель - настоящие методические указания предназначены для изучения основных возможностей среды и типовых приемов логического программирования. Приведены необходимые теоретические сведения об основах логического программирования и типовые примеры решения задач, представлены варианты индивидуальных заданий для лабораторных работ в среде ПРОЛОГ и порядок их выполнения, а также контрольные вопросы для самостоятельной работы студентов. Особое внимание уделено вопросам разработки рекурсивных процедур, обработки списков, динамических баз данных и бинарных деревьев.

Методические указания рассмотрены и утверждены на заседании кафедры кибернетики и вычислительной техники (протокол № 4 от 24.10.2006 г.)

Допущено учебно-методическим центром СевНТУ в качестве методических указаний.

Рецензенты: Бражников С.А., канд. техн. наук, доцент

Фисун С.Н.,канд. техн. наук, доцент

Нормоконтроль: Персидсков Г.М.

СОДЕРЖАНИЕ

Введение

Пролог — язык программирования, в котором основные конструкции заимствованы из логики, начавшей свое развитие еще во времена Аристотеля более 2,2 тыс. лет назад. Логика же предоставляет точный язык для явного выражения целей, знаний и предположений. Она позволяет сделать заключение об истинности или ложности одних утверждений исходя из знаний об истинности или ложности других. Однако положения традиционной логики при создании языков программирования стали использовать сравнительно недавно.

Осознание того, что вычисление – часть случайного логического вывода привело к возникновению логического программирования [4], первая реализация которого была осуществлена в семидесятые годы в виде системы Пролог [1, 2]. Суть идеи – представить в качестве программы формальное описание предметной области, а затем сформулировать необходимую для решения задачу в виде цели или утверждения. Построение же решения этой задачи в виде вывода в этой системе предложить самой машине. Последнее возможно, поскольку нужный алгоритм решения (поиска) осуществляется решателем, строящим вывод по определенной стратегии [3, 8]. При такой постановке основная задача программиста сводится к описанию предметной области в виде системы логических формул и отношений на них, которые с достаточной степенью полноты описывают задачу. Этот подход стал возможен благодаря тому, что были получены достаточно эффективные методы автоматического поиска доказательств [8].

Проиллюстрируем вышесказанное, воспользовавшись исчислением высказываний. Дадим формальное описание предметной области в виде утверждений:

Правило1 : Если жарко и сыро, то будет дождь

Правило2 : Если сыро, то жарко база знаний

Факт3 : Сейчас сыро

Воспользуемся обозначениями:

P – жарко, Q – сыро, R – будет дождь,

тогда

Правило1 =P∩Q –>R

Правило2 =Q–>P формальное описание базы знаний

Факт3 : Q

и зададимся вопросом-целью:

будет ли дождь?

Решим поставленную задачу, воспользовавшись правилом вывода modus ponens: A , A -> B => B (если верно A и A–>B, то верно и B). Тогда Q , Q -> P => P, т. е. верно P.

Но тогда Q -> P, Q /\ P -> R => R, т.е. верно R. То есть, поставленная задача решена, ответ - будет дождь.

Использование правила вывода modus ponens позволяет показать возможность достижения цели - доказательства теорем. На практике обычно используется более эффективные с точки зрения вывода методы доказательства теорем.

Один из таких методов - метод резолюции [3] - позволяет реализовать на практике концепцию логического программирования, согласно которой вычислительная программа может быть записана при помощи логических формул, играющих роль аксиом, а её вычисление представлено в виде доказательства формулы-запроса.

Открытие Робинсоном (1965) правила резолюции позволило разработать эффективные процедуры доказательства в логике первого порядка и представляет собой значительный шаг на пути практического применения автоматического доказательства теорем, поскольку резолюция обладает важными свойствами корректности и полноты.

Декларативный и процедурный смысл программ

В программах на Прологе стоит различать два уровня смысла, а именно:

- декларативный смысл и
- процедурный смысл.

Декларативный смысл касается только отношений, определенных в программе. Таким образом, декларативный смысл определяет, что должно быть результатом работы программы. С другой стороны, процедурный смысл определяет еще и как этот результат был получен, т. е. как отношения реально обрабатываются пролог-системой.

Способность пролог-системы обрабатывать многие процедурные детали самостоятельно считается одним из специфических преимуществ Пролога. Это свойство побуждает программиста рассматривать декларативный смысл программы относительно независимо от ее процедурного смысла. Поскольку результаты работы программы в принципе определяются ее декларативным смыслом, последнего (опять же в принципе) достаточно для написания программ. Этот факт имеет практическое значение, поскольку декларативные

аспекты программы являются, обычно, более легкими для понимания, нежели процедурные детали. Чтобы извлечь из этого обстоятельства наибольшую пользу, программисту следует сосредоточиться главным образом на декларативном смысле и по возможности не отвлекаться на детали процесса вычислений. Последние следует в возможно большей мере предоставить самой пролог-системе.

Такой декларативный подход и в самом деле часто делает программирование на Прологе более легким, чем на таких типичных процедурно-ориентированных языках, как Паскаль. К сожалению, однако, декларативного подхода не всегда оказывается, достаточно. Далее станет ясно, что, особенно в больших программах, программист не может полностью игнорировать процедурные аспекты по соображениям эффективности вычислений. Тем не менее следует поощрять декларативный стиль мышления при написании пролог-программ, а процедурные аспекты игнорировать в тех пределах, которые устанавливаются практическими ограничениями.Таким образом, сформулируем краткие выволы:

– программирование на Прологе состоит в определении отношений и в постановке вопросов, касающихся этих отношений,

– программа состоит из предложений. Предложения бывают трех типов: факты, правила и вопросы,

– отношение может определяться фактами, перечисляющими *n*-ки объектов, для которых это отношение выполняется, или же оно может определяться правилами,

– процедура - это множество предложений об одном и том же отношении,

– вопросы напоминают запросы к некоторой базе данных. Ответ системы на вопрос представляет собой множество объектов, которые удовлетворяют запросу,

– процесс, в результате которого пролог-система устанавливает, удовлетворяет ли объект запросу, часто довольно сложен и включает в себя логический вывод, исследование различных вариантов и поиск с возвратом. Все это делается автоматически самой пролог-системой и по большей части скрыто от пользователя,

– различают два типа смысла пролог-программ: декларативный и процедурный. Декларативный подход предпочтительнее с точки зрения программирования. Тем не менее, программист должен часто учитывать также и процедурные детали.

Воспользуемся примером Пролог-программы, приведенной в работе [3].

вор(питер).

любит(мэри, шоколад).

любит(мэри, сок)

любит(питер, деньги).

любит(питер, X) :- любит(X, сок).

может_похитить(X, Y) :- вор(X), любит(X, Y).

Опишем работу интерпретатора для цели:
может_похитить(питер, Z).

то есть вопрос «что может похитить Питер?».

Интерпретатор отыщет последнее правило программы и унифицирует цель с его левой частью, при этом

X = питер, а Y = Z,

вместо старой цели появятся две новые подцели: ?
вор(питер), любит(питер, Z).

Первая подцель достижима, поскольку она унифицируется с фактом в программе: вор(питер).

Вторая подцель: любит(питер, Z) будет унифицирована с фактом любит(питер, деньги), при этом

Z = деньги.

Поскольку и эта цель достижима, интерпретатор выдаст ответ Y = деньги. То есть Питер может_похитить деньги.

Интерпретатор продолжит поиск других целей и расторгнет унификацию цели любит(питер, X) с фактом любит(питер, деньги). Взамен этого цель любит(питер, X) будет унифицирована с правой частью правила, расположенного в пятой строке программы. Эта унификация означает, что X = Z и вместо старой цели любит(питер, X) будет выставлена цель любит(Z, сок), поскольку X = Z. Эта цель будет достигнута, поскольку она унифицируется с фактом, находящимся в третьей строке – любит(мэри, сок). В результате Z = мэри и интерпретатор выдаст ещё один ответ Z = мэри, то есть питер может_похитить мэри.

Лабораторная работа № 1

Изучение основ программирования на языке Пролог

Цель работы – изучить основы синтаксиса языка Пролог, выработать навыки работы с интерактивной системой Пролог, научиться оформлять отношения между данными на языке Пролог на примере родственных отношений между членами семьи.

Порядок выполнения работы

1. Изучить основные предикаты и возможности системы Пролог.

2. Создать программу, которая описывает родственные отношения между членами семьи.

3. Составить отчет о проделанной работе.

Теоретические положения

Язык программирования Пролог базируется на ограниченном наборе механизмов, включающих в себя сопоставление образцов, древовидное представление структур данных и автоматический возврат. Этот небольшой набор образует удивительно мощный и гибкий программный аппарат.

Программирование на Прологе – программирование в терминах логики. Программы на Прологе записываются в декларативном стиле, это означает, что выполнение программ Пролог-машиной происходит по принципу:

«Ты скажи: " Что сделать", а я разработаю "Как это сделать"». Другими словами, в отличие от императивных языков программирования Пролог– интерпретатор сам выбирает порядок исполнения операторов в программе. Особенно хорошо Пролог приспособлен для решения задач, в которых фигурируют объекты и отношения между ними. Примером такой задачи может служить задача, описывающая родственные отношения в семье. На Прологе легко определить отношения, подобные отношению родитель**(X,Y),** что означает ИСТИНА если X является родителем Y, и ЛОЖЬ в противном случае.

Программа на языке Пролог состоит из предложений. Идею логического программирования можно сформулировать в двух метафорических равенствах:

программа = множество аксиом;

вычисление = конструктивный вывод целевого утверждения

из программы.

Аргументами отношений могут быть атомы (константные объекты) или переменные, списки. В тексте программы названия переменных начинаются с большой буквы, а названия констант – с маленькой.

На Прологе отношения могут быть записаны в виде фактов и правил. В виде фактов записываются простейшие отношения.

human(alexey).

fruit(orange).

Правила нужны для получения новых отношений из уже известных.

father(X,Y):-parent(X,Y),man(X).

Что означает: X является отцом Y, ЕСЛИ X родитель Y и X является мужчиной.

Вопрос к системе формируется в виде цели, которая может содержать переменные. В этом случае Пролог-машина проверит все возможно допустимые состояния переменных и найдет те из них, при которых цель достижима.

Повторяющееся применение фактов к правилам, приводящее к новым фактам, называется прямым рассуждением (*forward chaining*). При обратном рассуждении (*backward chaining*) осуществляется поиск заключений (заголовков правил), соответствующих цели. Если такое заключение найдено, то в свою очередь должны быть доказаны цели, входящие в данное правило. В Прологе используется обратное рассуждение.

Пример программы «родственные отношения».

На рис. 1.1 представлен пример - родственные отношения. Тот факт, что Том является родителем Боба, можно записать на Прологе так:

родитель(том, боб).

Здесь мы выбрали родитель в качестве имени отношения, том и боб - в качестве аргументов этого отношения. По причинам, которые станут понятны позднее, мы записываем такие имена, как том, начиная со строчной буквы. Все дерево родственных отношений рис. 1.1 описывается следующей пролог-программой:

родитель(пам, боб).
родитель(том, боб).
родитель(том, лиз).
родитель(боб, энн).
родитель(боб, пат).

родитель(пам, джим).

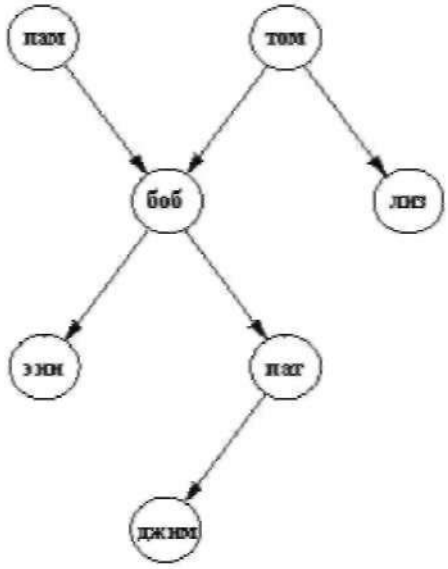

Рис. 1. 1. Дерево родственных отношений.

Эта программа содержит шесть предложений. Каждое предложение объявляет об одном факте наличия отношения родитель.

После ввода такой программы в пролог-систему последней можно будет задавать вопросы, касающиеся отношения родитель. Например, является ли Боб родителем Пат? Этот вопрос можно передать пролог-системе, набрав на клавиатуре терминала:

?- родитель(боб, пат).

Найдя этот факт в программе, система ответит

yes (да)

Другим вопросом мог бы быть такой:

?- родитель(лиз, пат).

Система ответит

no (нет),

поскольку в программе ничего не говорится о том, является ли Лиз родителем Пат. Программа ответит "нет" и на вопрос

?- родитель(том, бен).

потому, что имя Бен в программе даже не упоминается.

Можно задавать и более интересные вопросы. Например:"Кто является родителем Лиз?"

?- родитель**(X,** лиз**).**

На этот раз система ответит не просто "да" или "нет". Она скажет нам, каким должно быть значение **X** (ранее неизвестное), чтобы вышеприведенное утверждение было истинным. Поэтому мы получим ответ:

X = том

Вопрос "Кто дети Боба?" можно передать пролог-системе в такой форме:

?- родитель**(** боб**, X) .**

В этом случае возможно несколько ответов. Сначала система сообщит первое решение:

X = энн

Возможно, мы захотим увидеть и другие решения. О нашем желании мы можем сообщить системе (во многих реализациях для этого надо набрать точку с запятой), и она найдет другой ответ:

X = пат

Если мы потребуем дальнейших решений, система ответит "нет", поскольку все решения исчерпаны.

Нашей программе можно задавать и более общие вопросы: "Кто чей родитель?" Приведем другую формулировку этого вопроса:

Найти X и Y такие, что X - родитель Y.

На Прологе это записывается так:

?- родитель**(X, Y).**

Система будет по очереди находить все пары вида "родитель-ребенок". По мере того, как мы будем требовать от системы новых решений, они будут выводиться на экран одно за другим до тех пор, пока все они не будут найдены. Ответы выводятся следующим образом:

X=пам **Y=** боб**;**

X=том **Y=** боб**;**

X=том **Y=** лиз**;**

. . .

Мы можем остановить поток решений, набрав, например, точку вместо точки с запятой (выбор конкретного символа зависит от реализации).

Нашей программе можно задавать и еще более сложные вопросы, скажем, кто является родителем родителя Джима? Поскольку в нашей программе прямо не сказано, что представляет собой отношение родительродителя, такой вопрос следует задавать в два этапа, как это показано на рис. 1.2.

(1) Кто родитель Джима? Предположим, что это некоторый Y.

(2) Кто родитель Y? Предположим, что это некоторый X.

Такой составной вопрос на Прологе записывается в виде последовательности двух простых вопросов:

?- родитель**(Y,** джим**),** родитель**(X, Y).**

Ответ будет:

X=боб **Y=** пат

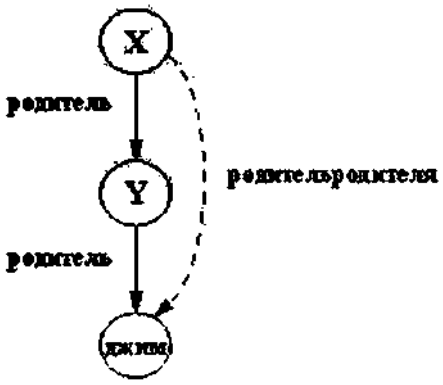

Рис. **1. 2.** Отношение родитель_родителя, выраженное через композицию двух отношений родитель.

Наш составной вопрос можно интерпретировать и так: "Найти X и Y, удовлетворяющие следующим двум требованиям":

родитель **(Y,** джим**)** и родитель**(X,** У**)**

Если мы поменяем порядок этих двух требований, то логический смысл останется прежним:

родитель **(X, Y)** и родитель **(Y,** джим**)**

Этот вопрос можно задать нашей пролог-системе и в такой форме:

?- родитель**(X, Y),** родитель**(Y,** джим**).**

При этом результат будет тем же. Таким же образом можно спросить: "Кто внуки Тома?"

?- родитель**(** том**, X),** родитель**(X, Y).**

Система ответит так:

X= боб **Y =** энн**;**

X= боб **Y =** пат

Следующим вопросом мог бы быть такой: "Есть ли у Энн и Пат общий родитель?" Его тоже можно выразить в два этапа:

(1) Какой X является родителем Энн?

(2) Является ли (тот же) X родителем Пат?

Соответствующий запрос к пролог-системе будет тогда выглядеть так:

?- родитель**(X,** энн**),** родитель**(X,** пат**).**

Ответ:

X = боб

Определения отношений, в которых используется само отношение, называется рекурсивным. Рекурсия – один из фундаментальных механизмов программирования на Прологе. Рекурсию можно применять для достижения такого же эффекта, какой реализуется при употреблении итеративных управляющих конструкций (циклов) в процедурных языках. Примером использования рекурсии может служить определение отношения «предок» [1, 2]. Данное отношение можно выразить с помощью двух правил. Первое правило будет определять непосредственных предков, а второе – отдаленных. Первое правило легко сформулировать через отношение «родитель»:

предок**(X, Z) :-** родитель**(X, Z).**

Аналогичным образом можно попытаться сформулировать второе правило:

предок**(X, Z) :-** родитель**(X, Z).**

предок**(X, Z) :-** родитель**(X, Y),** родитель**(Y, Z).**

предок**(X, Z) :-** родитель**(X, Y1),** родитель**(Y1, Y2),** родитель**(Y2, Z).**

...

Подобное описание отношения будет работать только в определенных пределах, то есть обнаруживать предков лишь до определенной глубины, поскольку длина цепочки людей между предком и потомком ограничена длиной предложений в определении отношения.

Подобные отношения целесообразно описывать с помощью рекурсии. Правило будет выглядеть следующим образом:

Для всех X и Z,

X - предок Z, если
существует Y, такой, что

X - родитель Y и

Y - предок Z.

или на языке Пролог:

предок**(X, Z) :-** родитель**(X, Y),** предок**(Y, Z).**

Таким образом, определение отношения «предок» будет выглядеть следующим образом:

предок**(X, Z) :-** родитель**(X, Z).**

предок**(X, Z) :-** родитель**(X, Y),** предок**(Y, Z).**

Наша программа-пример помогла проиллюстрировать некоторые важные моменты.

- На Прологе легко определить отношение, подобное отношению родитель, указав *п-ку* объектов, для которых это отношение выполняется.

- Пользователь может легко задавать пролог-системе вопросы, касающиеся отношений, определенных в программе.

  Пролог-программа состоит из предложений. Каждое предложение заканчивается точкой.

- Аргументы отношения могут быть (среди прочего): конкретными объектами, или константами (такими, как том и энн), или абстрактными объектами, такими, как **X** и **Y**. Объекты первого типа называются атомами. Объекты второго типа - переменными.

- Вопросы к системе состоят из одного или более целевых утверждений (или кратко целей). Последовательность целей, такая как

  родитель**(X,** энн**),** родитель**(X,** пат**)**

  означает конъюнкцию этих целевых утверждений:

  X - родитель Энн и

  X - родитель Пат.

  Пролог-система рассматривает вопросы как цели, к достижению которых нужно стремиться.

- Ответ на вопрос может оказаться или положительным или отрицательным в зависимости от того, может ли быть соответствующая цель достигнута или нет. В случае положительного ответа мы говорим, что соответствующая цель достижима и успешна. В противном случае цель недостижима, имеет неуспех или терпит неудачу.

- Если на вопрос существует несколько ответов, пролог-система найдет столько из них, сколько пожелает пользователь.

Контрольные вопросы

1.1. Считая, что отношение родитель определено так же, как и раньше в данном разделе (см. рис. 1.1), найдите, какими будут ответы пролог-системы на следующие вопросы:

1. **?** - родитель**(X,** джим**).**

2. **?** - родитель**(** пам**,**X**),** родитель**(** X**,** пат**).**

3. **?** - родитель**(** пам**,** X**),** родитель**(** X**,** Y).

1.2. Сформулируйте на Прологе следующие вопросы

4. Кто родитель Пат?

5. Есть ли у Лиз ребенок?

6. Кто является родителем родителя Пат?

Задание к работе

1.Изобразить граф, иллюстрирующий описываемые родственные отношения.

2. Составить программу, которая описывает родственные отношения.

В качестве фактов описать унарные отношения: мужчина, женщина; и бинарные: состоят_в_браке, родитель_ребенок. Записать правила, которые определяют следующие отношения: муж, жена, родитель, ребенок, сын, дочь, брат, сестра, дядя, тетя, бабушка, дедушка.

3. Подобрать тестовые данные, проверяющие все полученные отношения. Тестовые данные должны содержать операции «И», «ИЛИ», «НЕ».

4. Оформить отчет о проделанной работе.

Содержание отчета

1. Описание родственных отношений, представленных в лабораторной работе, в виде графа.

2. Привести тексты программ на языке Пролог.

3. Выводы по работе.

Лабораторная работа №2

Изучение структуры интерфейса и пролог-программы

Цель работы – изучить основные режимы работы интерфейса, структуру программы и ее разделы, встроенные предикаты: ввода/вывода, для работы с файлами, базами данных, трассировки и другие.

Последовательность выполнения работы

1. Изучить структуру интерфейса ПРОЛОГ.
2. Освоить режимы работы интерфейса: редактирование, запуск программы, отладка и др.
3. Изучить структуру программы ПРОЛОГ на подготовленных примерах.
4. Изучить назначение основных встроенных предикатов ПРОЛОГ

Теоретические положения

Введем основные понятия языка Пролог *(*в дальнейшем просто Пролог*)*.

Факт - это некоторое утверждение, определяющее отношение между объектами или описывающее свойства объекта. Общая форма записи факта имеет следующий вид:

<имя отношения>(имя_объекта_*1*,имя_объекта_*2*,…, имяобъекта_*N*).

Необходимо соблюдать следующие правила :

- имена всех отношений и объектов должны начинаться со строчной буквы;
- сначала записывается имя отношения, затем через запятую записываются имена объектов, а весь список имен объектов заключается в круглые скобки;
- каждый факт должен заканчиваться точкой;
- имена объектов в скобках могут перечисляться произвольно, но по одному произвольному порядку.

Например, факт с двумя объектами может быть описан так:

likes(tom,computer).

На естественном языке вышеприведенный факт означает: "Тому нравится компьютер".

Атом - имя, число без знака или символ.

Аргумент - имя объекта в круглых скобках.

Объект - название отдельного элемента в конструкции Пролога.

Домен - диапазон и тип значений, определенных для базисного типа данных.

Предикат - утверждение о наличии связи между объектами посредством задания имени отношения перед круглыми скобками и доменов его аргументов.

Функтор - имя составного объекта (в Прологе функторы объявляются в разделе программы predicates).

База данных - в Прологе представляет собой совокупность фактов (утверждений).

Унификация - процесс, выполняющий попытки сопоставить цель и утверждение. Он обычно включает поиск, сопоставление и означивание.

Правило - утверждение о связи некоторого факта с другими фактами.

Общая форма записи правила имеет вид:

<заголовок правила>*:-* <тело правила>.

Заголовок представляет собой предикат. Тело состоит из термов, которые могут быть связаны между собой "," или ";". ("- означает И,";"- означает ИЛИ).

Между телом и заголовком стоит символ ":-", который означает ЕСЛИ.

Например, правило, состоящее из двух термов может описано следующим образом:

likes(tom,kathy) :- likes(kathy,computer), likes(kathy,apples).

На естественном языке это означает : "Тому нравится Кэти, если Кэти нравится компьютер и яблоки."

Структура программы и стандартные типы доменов языка Пролог

Программа на Прологе может состоять из следующих разделов:

1. CONSTANS - раздел описания констант. Имена констант должны быть описаны строчными буквами. В качестве констант могут быть целые или вещественные числа, символы, строки. Например, col=17, x=3.62, c='W', st="Выход".

2. DOMAINS - данный раздел содержит определения доменов, которые описывают различные классы объектов используемых в программе (определение типов данных).

Например, имеется факт

likes(mary,apples).

Здесь mary и appless являются объектами предиката likes . Пролог требует указания типов объектов для каждого предиката программы.

Для указания типа объекта Пролог имеет шесть следующих типов данных (типов доменов):

symbol (символические имена) - это последовательность букв, цифр и знаков подчеркивания, которая начинается со строчной буквы или заключена в кавычки, например: flower, pay_check, "Prolog" и т.п..

string (строки) - любая последовательность символов, которая заключена в кавычки (не более 250). Например: "today", "123", "ПРИВЕТ".

char (символы) - отдельный символ, заключенный в апострофы, например: 'A', '3', 'a', '\13'.

integer (целые числа) - можно задавать в диапазоне от -32768 до +32767.

real (действительные числа) - диапазон от +1E-307 до +1E308 .

file (файлы) - допустимое в DOS имя файла.

Вернемся к факту **likes(mary,apples).** С предикатом likes могут быть еще факты , такие ,например, как :

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

Во всех этих фактах с предикатом likes на первом месте стоят имена объектов, имеющие смысл "тот, кто любит", а на втором месте имена объектов - "вещь".

В данном примере эти два вида имен объектов имеют один и тот же тип symbol. Прежде чем показать, как выглядит раздел **DOMAINS** для нашего примера, рассмотрим раздел, который называется **PREDICATES**.

3. PREDICATES - данный раздел служит для описания используемых программой предикатов.

В нашем примере предикат likes не является встроенным предикатом, поэтому его необходимо объявить в разделе **PREDICATES**. Например:

PREDICATES

likes(symbol,symbol)

Это описание означает, что оба объекта относятся к типу symbol. Такое описание предиката **likes** освобождает от необходимости использования раздела **DOMAINS**. Наш пример является очень простым, поэтому мы можем себе такое позволить. Реальные программы могут содержать несколько десятков предикатов, причем с различным количеством объектов. Поэтому, чтобы программа хорошо читалась - видам объектов присваивают имена, но при этом уже необходим раздел **DOMAINS**. Для нашего примера это будет выглядеть так:

DOMAINS

person,thing = symbol
PREDICATES
likes(person,thing)

где person - "тот кто любит" ,thing - "вещь".

4. CLAUSES - в данный раздел заносятся факты и правила. О содержимом этого раздела можно говорить как о данных, необходимых для работы программы. Для нашего примера этот раздел включает три факта :

CLAUSES
likes(mary,apples).
likes(tom,computer).
likes(kathy,computer).

5. GOAL - данный раздел может располагаться перед разделом **CLAUSES** или после него. В этом разделе определяется цель. Цель может состоять из нескольких подцелей. Если программа предназначена для работы в пакетном режиме, раздел **GOAL** не может быть опущен.

В программе могут присутствовать еще два раздела, обеспечивающие определение глобальных доменов и предикатов.

6. GLOBAL DOMAINS - располагается после раздела **DOMAINS**.

7. GLOBAL PREDICATES - следует после раздела **PREDICATES**.

Определение типов данных и предикатов в этих разделах позволяет обеспечить межмодульный интерфейс.

8. DATABASE - следует перед разделом **PREDICATES**. Он содержит определения предикатов динамической базы данных. Если программа такой базы данных не требует, то этот раздел может быть опущен.

Пролог обеспечивает возможность включения в программу комментариев, которые обрамляются символами /* ... */ . Также комментарием считается строка, которая начинается с символа % .

В итоге программа в законченном виде для нашего примера будет выглядеть так :

/* НАЧАЛО */

Domains

person,thing = symbol

Predicates

likes(person,thing)

Goal

likes(mary,apples),nl, write("Мэри любит яблоки**").**

Clauses

likes(mary,apples).

likes(tom,computer).

likes(kathy,computer).

% КОНЕЦ

Предикаты и утверждения разных арностей

Термин «арность» обозначает число объектов утверждения. Количество объектов в одном предикате может быть не более 50. Ниже приведен пример предикатов и утверждений различных арностей.

Domains

s=string

Predicates % раздел предикатов

start % арность 0

woman(s) % арность 1
father(s,s) % арность 2

Clauses % раздел утверждений

start. % арность 0

woman("Маша**")** % арность 1
father("Петр Иванович**",** "Маша**")** % арность 2

Переменные в языке Пролог

Любое имя может быть переменной, если начинается с прописной буквы. Переменная позволяет отвечать на вопросы. Например, если

необходимо узнать кому нравятся яблоки (см. пример 1), то в разделе GOAL должны написать следующее

likes(X,apples),nl, write(X," - любит яблоки**").**

Переменная X конкретизируется первым значением, который имеется в базе фактов с предикатом likes.

Задание к работе

1. Выполнить задачи согласно варианту.

Проверить выполнение программ, хранящихся на диске **C:** в каталоге

\PROLOG\EXAMPLES, исследовать процесс выполнения программ в режиме трассировки.

Выполнить варианты **I=N(MOD 25), J=(I+11)(MOD 25), K=(J+11) (MOD 25),** где **N** – две последние цифры зачетной книжки**.**

2. Описать тестовые примеры и оформить отчет.

Содержание отчета

1. Описать назначение основных пунктов интерфейса.
2. Описать структуру программы и ее разделы.
3. Примеры обработки запросов
4. Описать и обьяснить процесс выполнения программ в режиме трассировки, включая механизм согласования целей, унификации переменных и поиска с возвратом.

Контрольные вопросы

1 Дать определение логического программирования.

2. Что такое фразы Хорна в ПРОЛОГ.

3. Из каких разделов состоит структура ПРОЛОГ-программы.

4. Какие стандартные предикаты Вам известны.

5. Обьяснить процесс сопоставления с образцом в ПРОЛОГ.
6 Дать понятие переменной в ПРОЛОГ.

7. Что представляет собой процесс унификации в ПРОЛОГ

8. Чем отличаются факты от правил в ПРОЛОГ.

9. Как выполняется согласования целей в ПРОЛОГ.

Лабораторная работа № 3

Использование рекурсивных правил

Цель работы – научиться использовать структурированные типы данных и рекурсивные правила при решении задач.

Порядок выполнения работы

1. Выполнить контрольные примеры.

2. Создать в системе программу, которая решает задачи согласно варианту.

3. Составить отчет о проделанной работе.

Теоретические положения

Методы организации рекурсии

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

Пример **1.** Правило рекурсии:

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

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

Однако в случае возникновения бесконечной рекурсии число элементов данных, используемых рекурсивным процессом, непрерывно растет и в некоторый момент стек переполнится. На экране появится сообщение об ошибке. Избегать подобные ситуации можно увеличением размеров стека, для чего служит опция Miscellaneous settings (прочие установки параметров) в меню Setup (установка).

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

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

Программа из примера 2 демонстрирует простое правило рекурсии, в которое включено условие выхода.

Пример **2.**

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

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

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

Метод обобщенного правила рекурсии

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

Ниже в символическом виде дан общий вид правила рекурсии:

```
<имя правила рекурсии> :-
        <список предикатов>,          (1)
```

<предикат условия выхода>, **(2)**

<список предикатов>, **(3)**

<имя правила рекурсии>, **(4)**

<список предикатов>. **(5)**

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

Данное правило рекурсии имеет пять компонент.

Первая – это группа предикатов. Успех или неудача любого из них на рекурсию не влияет.

Следующая компонента - предикат условия выхода. Успех или неудача этого предиката либо позволяет продолжить рекурсию, либо вызывает ее остановку.

Третья компонента - список других предикатов. Аналогично, успех или неудача этих предикатов на рекурсию не оказывает влияния.

Четвертая группа – само рекурсивное правило. Успех этого правила вызывает рекурсию.

Пятая группа - список предикатов, успех или неудача которых не влияет на рекурсию. Пятая группа также получает значения (если они имеются), помещенные в стек во время выполнения рекурсии.

Правила, построенные указанным образом, являются обобщенными правилами рекурсиии (ОПР), а метод называется ОПР-методом.

Например, вы хотите написать правило генерации всех целых

чисел начиная с 1 и кончая 7. Пусть имя правила будет

write_number(Number).

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

Третья компонента правила оперирует с числами. В этой части правила число выдается на экран и затем увеличивается на 1.

Для увеличенного числа будет использоваться новая переменная

Next_Number. Четветая компонента - вызов самого правила рекурсии write_number(Next_number). Пятая компонента, представленная в общем случае, здесь не используется.

Программа генерации ряда чисел (Пример 3) использует следующее правило рекурсии:

Пример **3.**

```
write_number(8).

write_number(Number) :-

        Number < 8,

        write(Number), nl,

        Next_Number = Number + 1,

        write_number(Next_number).
```

Программа начинается с попытки вычислить подцель write_number(1). Сначала программа сопоставляет подцель с первым правилом write_number(8). Так как 1 не равно 8, то сопоставление неуспешно. Программа вновь пытается сопоставить подцель, но уже с головой правила write_number(Number). На этот раз сопоставление успешно вследствие того, что переменной Number присвоено значение 1. Программа сравнивает это значение с 8; это условие выхода. Так как 1 меньше 8, то подправило успешно. Следующий предикат выдает значение, присвоенное Number.

Переменная Next_Number получает значение 2, а значение Number увеличивается 1. Важное свойство правила рекурсии состоит в его расширяемости. Например, оно может быть расширено для подсчета суммы ряда целых чисел.

Программа (Примера 4. – Сумма ряда) использует правило рекурсии для вычисления суммы ряда целых чисел от 1 до 7:

Пример **4. –** Сумма ряда**.** Вычислить

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

или

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

Правило рекурсии программы выполняет вычисления по обычной схеме

сложения:

1 Начальное значение

\+ 2 Следующее значение

———

3 Частичная сумма

\+ 3 Следующее значение

———

6 Частичная сумма

…

Правило рекурсии имеет вид:

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

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

При описании рекурсивных правил следует соблюдать осторожность во избежание зацикливания рекурсии. Для этого любое рекурсивное определение отношения должно включать, по крайней мере, два правила:

(1) тривиальные, или «граничные» случаи;

(2) «общие» случаи, в которых решение получается из решений для (более простых) вариантов самой исходной задачи.

Этот метод мы используем в Прологе постоянно. Рассмотрим еще один пример.

Пример **5.** Вычисление факториала

factorial(X, _) :– X<0, ! ,fail. % если X<0 то стоп, ошибка.

factorial (0 , 1) :– !. % 0!=1

factorial(N, Fact) :– N1=N–1, factorial(N1, Fact1), Fact=N*Fact1. % общий случай

Пример **6**.Определение чисел Фибоначчи:

f(1,1) :– !. % Первое число есть 1

f(2,1) :– !. % Второе число есть 2

f(I,R) :– I>2, I1=I–1, I2=I–2, f(I1,M), f(I2,N), R=N+M. % общий случай

Задание к работе

1. Выполнить задачи согласно варианту:

I=N(MOD 25)+1, J=(I+12)(MOD 25)+1.

2.Описать тестовые примеры и оформить отчет.

Варианты заданий

1. Возведение в степень как повторяющееся умножение.
2. Возведение в степень как повторяющееся сложение.
3. Рекурсивное определение остатка от деления (mod).
4. Рекурсивное определение деления нацело (div).
5. Функция Аккермана

Ak(0,N) = N + 1

Ak(M,0) = Ak(M-1,1)

Ak(M,N) = Ak(M-1,Ak(M,N-1))

7. Умножение через сложение единицы.
8. Деление через вычитание единицы.
9. Вычислить: Y = k*(k+1)*(k+2)*....*(k+n) .
10. Вычислить: W = (3+ *i)*+(6+i)+(9+i)+....+(m+i).

11 . Определить, является ли заданное число числом Фибоначчи.

12. Нахождение чисел Фибоначчи, не превышающих заданное число.

13. Вычисление *n-го* члена арифметической прогрессии, у которой первый член равен 1, а шаг = 2.

14. Вычисление *n-го* члена геометрической прогрессии, у которой первый член равен 2, а знаменатель равен 4

15. Сумма четных натуральных чисел от 1 до *n*

16. Произведение нечетных натуральных чисел от 1 до *n*..

17. Сумма всех двузначных чисел, кратных трем.

18. Сумма всех трехзначных чисел, не делящихся ни на 5, ни на 7.

19. Вычислить сумму *n* первых членов ряда:

1 + 1/2 + 1/3 + ...

20. Вычислить сумму n первых членов ряда:

4 - 4/3 + 4/5 - 4/7 + 4/9 - ... + (-1)^(й-1) × 4 / Г2х$_{й}$ - 1) + ...

21. Построить рекурсивную функцию для вычисления *n-го* члена последовательности, в которой каждый следующий член равен сумме *n-2 -го и n-3* -го. Первые 3 члена равны соответственно 1, 2, 3.

1 2 3 3 5 6 8 11

22. Построить рекурсивную функция для вычисления *n-го* члена последовательности, в которой каждый четный член равен сумме двух предыдущих четных, а нечетный равен сумме двух предыдущих нечетных. Первые четыре члена равны соответственно 1, 2, 3, 4.

1 2 3 4 4 6 7 10 11 16 18 ...

23. Построить рекурсивную функцию для вычисления *n-го* члена последовательности, в которой каждый следующий четный член равен произведению двух предыдущих, а каждый следующий нечетный член равен сумме двух предыдущих, а первые 2 члена равны соответственно 1 и 2.

1 2 3 6 9 54 63 ...

24. Построить рекурсивную функцию для вычисления *n-го* члена последовательности, в которой каждый следующий член равен

произведению двух предыдущих, а первые 2 члена равны соответственно 1 и 2.

1 2 2 4 8 32 …

25. Построить рекурсивную функцию для вычисления *n*-го члена последовательности, в которой первый член равен 0, второй 1, третий 2, а каждый следующий равен сумме трех предыдущих.

1 2 3 6 11 ….

Содержание отчета

1. Исходные тексты и структуры программ на языке Пролог.
2. Примеры обработки запросов
3. Анализ результатов работы программ в режиме трассировки.
4. Выводы по работе.

Контрольные вопросы

1 В чем разница между декларативной и процедурной семантикой в ПРОЛОГ.

2 Что представляют собой правила в ПРОЛОГ.

3 Что Вы понимаете под эффективностью разработки программ в ПРОЛОГ.

4 Дать понятие составных обьектов.

5 Как обьявляются домены составных обьектов в ПРОЛОГ.

6 Как выполняется поиск с возвратом в ПРОЛОГ. Основные правила.

7 Что такое многоуровневые составные обьекты в ПРОЛОГ.

8 Дать определение составных смешанных обьектов в ПРОЛОГ.

Лабораторная работа № **4**

Обработка списков

Цель работы: ознакомиться с реализацией структуры данных типа список в языке Пролог и методами их обработки.

Порядок выполнения работы

1. Выполнить контрольные примеры.
2. Создать программу на языке Пролог, которая решает задачи согласно варианту.
3. Составить отчет о проделанной работе.

Теоретические положения

Пролог также поддерживает связанные объекты, называемые списками.

Список - это упорядоченный набор объектов, следующих друг за другом. Составляющие списка внутренне связаны между собой, поэтому с ними можно работать и как с группой (списком в целом), так и как с индивидуальными объектами (элементами списка).

Пролог позволяет выполнять со списком целый ряд операций, например:

- доступ к объектам списка,
- проверка на принадлежность к списку,
- разделение списка на два,
- слияние двух списков,
- сортировку элементов списка в порядке возрастания или убывания и другие.

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

Список является набором объектов одного и того же доменного типа. Объектами списка могут быть целые числа, действительные числа, символы, символьные строки и структуры. Порядок расположения элементов является отличительной чертой списка; те же самые элементы, упорядоченные иным способом, представляют уже совсем другой список. Порядок играет важную роль в процессе сопоставления.

Пролог допускает списки, элементами которых являются структуры. Если структуры принадлежат к альтернативному домену, элементы списка могут иметь разный тип. Такие списки используются для специальных целей.

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

Примерами списков могут служить:

[1,2,3,6,9,3,4]

[3.2,4.6,1.1,2.64,100.2]

["YESTERDAY","TODAY","TOMORROW"]

Элементами первого списка являются целые числа. Элементами второго - действительные числа, третьего - символьные строки, т. е. конкретные значения символов. Любой печатный символ кода ASCII пригоден для списков этого типа. (Более подробно о коде ASCII будет написано в гл. 6).

Атрибуты списка

Объекты списка называются элементами списка. Список может содержать произвольное число элементов, единственным ограничением является лишь объем оперативной памяти.

Пролог требует, чтобы все элементы списка принадлежали к одному и тому же типу доменов. Другими словами, либо все элементы списка - целые числа, либо - действительные, либо - символы, либо - символьные строки. В Прологе список

["JOHN WALKER",3.50,45.50]

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

Количество элементов в списке называется его длиной.

Длина списка ["MADONNA","AND","CHILD"] равна 3.Длина списка

[4.50,3.50,6.25,2.9,100.15] равна 5. Список может содержать всего один элемент и даже не содержать элементов вовсе:

["summer"]

[ ]

Список, не содержащий элементов, называется пустым списком.

Непустой список можно рассматривать как состоящий из двух частей:

(1) первый элемент списка - его голова, и

(2) остальная часть списка - хвост.

Голова является элементом списка, хвост есть список сам по себе. Голова - это отдельное неделимое значение. Наоборот, хвост представляет из себя список, составленный из того, что осталось от исходного списка в результате "отсеченния головы". Этот новый список зачастую можно делить и дальше. Если список состоит из одного элемента, то его можно разделить на голову, которой будет этот самый единственный элемент, и хвост, являющийся пустым списком.

В списке

[4.50,3.50,6.25,2.9,100.15]

например, головой является значение 4.50, а хвостом – список

[3.50,6.25,2.9,100.15]

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

[6.25,2.9,100.15]

В табл. 1. показаны головы и хвосты нескольких списков.

Таблица 1. Головы и хвосты различных списков

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

Отличительной особенностью описания списка является наличие звездочки (*) после имени домена элементов. Так запись

bird_name *

указывает на то, что это домен списка, элементами которого являются bird_name, т. е. запись bird_name* следует понимать как список, состоящий из элементов домена bird_name.

Описание в разделе domains, следовательно, может выглядеть либо как

```
bird_list = bird_name *

bird_name = symbol
```

либо как

```
bird_list = symbol *
```

Домен bird_list является доменом списка элементов типа symbol (списка птиц).

В разделе описания предикатов predicates требуется присутствия имени предиката, а за ним заключенного в круглые скобки имени домена.

```
birds(bird_list)
```

Как видим, описание предиката списка ни в чем не отличается от описания обычного предиката.

Сам список присутствует в разделе утверждений clauses:

```
birds(["sparrow","robin","mockingbird","thunderbird",
      "bald eagle"]).
```

Чуть позднее будет описано, как использовать списки в разделе программы goal.

Законченный пример использования списков приведен в программе "Списки" (Пример.1), в которую дополнительно включены список из 7 целых чисел (домен number_list) и предикат score.

Пример. **1**

```
/* Работа со списками.                */

domains

  bird_list = bird_name *

        bird_name = symbol

  number_list = number *

        number = integer
```

```
predicates
  birds(bird_list)
  score(number_list)
clauses
  birds(["sparrow",
         "robin",
         "mockingbird",
         "thunderbird",
         "bald  eagle"]).
  score([56,87,63,89,91,62,85]).
/*          конец программы          */
```

Эта программа создавалась в расчете на следующие внешние запросы:

```
birds(All).
birds([_,_,_,B,_]).
birds([B1,B2,_,_,_]).
score(All).
score([F,S,T,_,_,_,_]).
```

Читателю предлагается самостоятельно обработать приведенные запросы.

Пример **2.** Деление списков

При работе со списками достаточно часто требуется разделить список на несколько частей. Это бывает необходимо, когда для целей текущей обработки нужна лишь определенная часть исходного списка, а оставшуюся часть следует обработать позднее. Сейчас Вы увидите, что деление списков на части является достаточно простой операцией.

Для пояснения сказанного рассмотрим предикат split, аргументами которого являются элемент данных и три списка:

split(Middle,L,L1,L2).

Элемент Middle здесь является компаратором, L - это исходный список, а L1 и L2 - подсписки, получающиеся в результате деления списка L. Если элемент исходного списка меньше или равен Middle, то он помещается в список L1; если больше, то в список L2.

Предположим, что вначале значением переменной Middle является число 40, переменной L присвоен список [30,50,20,25,65,95], а переменные L1 и L2 не инициализированы.

split(40,[30,50,20,25,65,95],L1,L2).

Правило для разделения списка должно быть написано таким образом, чтобы элементы исходного списка, меньшие либо равные 40, помещались в список L1, а большие 40 - в список L2. Правило устроено следующим образом: очередной элемент извлекается из списка при помощи метода разделения списка на голову и хвост, а потом сравнивается с компаратором Middle. Если значение этого элемента меньше или равно значению компаратора, то элемент помещается в список L1, в противном случае - в список L2. На рис. 5.7 приведены состояния объектов Middle, L, L1, L2 на различных этапах работы правила.

В результате применения правила к списку [30,50,20,25,65,95] значениями списков L1 и L2 станут соответственно [30,20,25] и [50,65,95].

Само правило для разделения списка записывается в Прологе следующим образом:

```
split(Middle,[Head|Tail],[Head|L1],L2)  :-
    Head <= Middle,
    split(Middle,Tail,L1,L2).
split(Middle,[Head|Tail],L1,[Head|L2])  :-
    split(Middle,Tail,L1,L2),
    Head > Middle.
    split(_,[],[],[]).
```

Отметим, что метод деления списка на голову и хвост используется в данном правиле как для разделения исходного списка, так и для формирования выходных списков.

Попробуйте ввести такое целевое утверждение:

```
split(40,[30,50,20,25,65,95],L1,L2).
```

Результат работы программы с данной целью Вы можете получить на компьютере.

```
/* Разделение списка на два.  */
domains
 middle = integer
 list = integer *
predicates
 split(middle,list,list,list)
clauses
 split(Middle,[Head|Tail],[Head|L1],L2) :-
      Head <= Middle,
      split(Middle,Tail,L1,L2).
 split(Middle,[Head|Tail],L1,[Head|L2]) :-
      split(Middle,Tail,L1,L2),
      Head > Middle.
      split(_,[],[],[]).
/*          конец программы          */
```

Задание к работе

1. Реализовать набор функций для обработки списков:

- Добавление элемента к списку.
- Удаление элемента из списка.
- Конкатенация списков.
- Определение длины списка.
- Определение принадлежности элемента списку.

2. Решить задачи согласно варианту:

I=N(MOD 25)+1, J=(I+12)(MOD 25)+1.

используя построенные функции в пункте 1.

3. Подобрать тестовые данные и оформить отчет.

Варианты заданий

1. Два списка U и V имеют одинаковое число элементов. Составить программу, выполняющую правило объединить**(U,V,List),** которое последовательно помещает соответствующие элементы U и V в List.
2. ПРИМЕР: [0,1,2],[3,4,5] => [0,3,1,4,2,5]
3. Задан список L ,содержащий четное число элементов.
4. Построить списки M и N, содержащие элементы, находящиеся на четных и нечетных позициях соответственно.
5. Определить принадлежность элемента X списку L.
6. Определить отношение перевод**(**Список**1,**Список**2)** для перевода списка чисел от 0 до 9 в список соответствующих слов нуль, один и т.д.
7. Определить, что список не содержит повторяющихся элементов.
8. Определить, что список M является началом списка L.
9. Определить, что все элементы списка M содержатся в L (в любом порядке).
10. Определите два предиката четнаядлина**(**Список**)** и нечетнаядлина**(**Список**)** таким образом, чтобы они были истинными, если их аргументом является список четной или нечетной длины соответственно. Например, список [a,b,c,d] имеет четную длину.
11. Определить предикат суммасписка**(**Список**,**Сумма**)** так, чтобы Сумма равнялась сумме чисел, входящих в Список**.**
12. Задать правило для объединения списков: U \/ V -> L.
13. Задать правило для пересечения списков: U /\ V -> L.
14. Определить предикат упорядоченный**(**Список**)** так, чтобы он принимал значение истина, если Список упорядочен. Например, список [1,3,7,8,19] – упорядочен.
15. Определить отношение **n_**элемент**(N,**Список**,**X**),** которое выполняется,

если X является **N**-ым элементом списка Список.

16. Удалить первый по порядку элемент x, принадлежащий L.

17. Выбрать последний элемент списка L.

18. Подсчитать количество положительных чисел в списке;

19. Удалить повторяющиеся элементы в списке.

20. Вычесть от каждого элемента списка 1.

21. Найти сумму четных элементов списка.

22. Определить длину списка.

23. Найти сумму нечетных элементов списка.

24. Подсчитать число элементов списка больших 10.

25. Выделить из списка все элементы, которые делятся на 2 и на 3;

Содержание отчета

1. Исходные тексты и структуры программ на языке Пролог.
2. Примеры обработки запросов
3. Анализ результатов работы программ в режиме трассировки.
4. Выводы по работе.

Контрольные вопросы

1. Управление процессом выполнения программы. Предикаты: fail, cut, not.
2. Правила использования предиката **not** в ПРОЛОГе.
3. Вопросы эффективности разработки программ в ПРОЛОГе.
4. Понятие составных обьектов. Функторы в ПРОЛОГе.
5. Понятие хвостовой рекурсии. Оптимизация хвостовой рекурсии.
6. Механизм логического вывода в ПРОЛОГе.
7. Обработка списков. Основные процедуры в ПРОЛОГе.

Лабораторная работа № 5

Работа с динамическими базами данных

Цель работы – изучить основные приемы реализации динамических баз данных в языке Пролог.

Порядок выполнения работы

1. Решить задачи соответствующего варианта.

2. Подобрать тестовые данные и протестировать программу на компьютере.

3. Составить отчет о проделанной работе.

Теоретические положения

Реляционная модель данных предполагает, что база данных – это есть описание некоторого множества отношений. Пролог–программу можно рассматривать как такую базу данных, здесь отношения между данными представлены в виде фактов и правил. В Прологе есть специальные средства для модифицирования этой базы данных, то есть добавлять и удалять новые отношения из файла. Для этих целей служат встроенные предикаты.

Чтобы понять, как в Прологе реализуется обращение к БД, рассмотрим запрос на примере БД игроков футбольной команды:

```
dplayer("Bernie Kosar",Team,Pos).
```

В этом утверждении Team и Pos есть переменные, значения которых нужно найти. Когда этот запрос (цель) испытывается, процедуры Пролога просматривают утверждения БД игроков (см.ниже) на предмет сопоставления с утверждением, содержащим Bernie Kosar. Так как в базе данных такое утверждение присутствует, то переменной Team присваивается значение Cleveland Browns, а переменной Pos - QB.

Если трактовать dplayer как предикат БД Пролог, то отсюда следует с необходимостью такое его описание

```
database

    dplayer(name,team,position)
```

Раздел **database** в Прологе предназначен для описания предикатов базы данных, таких как dplayer. Все различные утверждения этого предиката составляют динамическую базу данных Пролога. База данных называется динамической, так во время работы программы из нее можно удалять любые содержащиеся в ней утверждения, а также добавлять новые. В этом

состоит ее отличие от "статических" баз данных, где утверждения являются частью кода программы и не могут быть изменены во время счета. Другая важная особенность динамической базы данных состоит в том, что такая база может быть записана на диск, а также считана с диска в оперативную память.

Иногда бывает предпочтительно иметь часть информации базы данных в виде утверждений статической БД; эти данные заносятся в динамическую БД сразу после активизации программы. Для этой цели используются предикаты **asserta** и **assertz,** которые будут рассмотрены ниже. В общем, предикаты статической БД имеют другое имя, но ту же самую форму представления данных, что и предикаты динамической. Предикат статической БД, соответствующий предикату dplayer динамической базы данных, есть

```
predicates
    player(name,team,position)
clauses
    player("Dan Marino","Miami Dolphins","QB").
    player("Richart Dent","Chicago Bears","DE").
    player("Bernie Kosar","Cleveland Browns","QB").
    player("Doug Cosbie","Dallas Cowboy","TE").
    player("Mark Malone","Pittsburgh Steelers","QB").
```

Заметим, что все отличие предиката dplayer по сравнению с player заключается лишь в одной лишней букве терма. Добавление латинской буквы d - обычный способ различать предикаты динамической и статической баз данных. Правилом для занесения в динамическую БД информации из утверждений предиката player служит

```
assert_database :-
  player(Name,Team,Number),
  assertz( dplayer(Name,Team,Number) ),
  fail.
assert_database :- !.
```

В этом правиле применяется уже знакомый вам метод отката после неудачи, который позволяет перебрать все утверждения предиката player.

Таким образом, можно указать следующее соответствие понятий:

база данных Пролога – реляционная база данных,

предикат БД – отношение,

объект – атрибут,

отдельное утверждение – элемент отношения,

количество утверждений – мощность.

Пример использования динамической базы данных в Прологе.

При создании динамической базы данных Пролога будут очень полезны многие уже известные вам вещи. Так, здесь применим почти весь материал, касающийся предикатов и утверждений Пролога. Можно создавать правила для работы с информацией БД, можно также использовать введенные предикаты, осуществляющие обращение к файлам, файлам БД, записанным на диск.

В Прологе имеются и специальные встроенные предикаты для работы с динамической базой данных. Таковыми являются **asserta, assertz, retract, save, consult,** и **findall.**

Предикаты **asserta, assertz** и **retract** позволяют занести факт в заданное место динамической БД и удалить из нее уже имеющийся факт.

Предикат **asserta** заносит новый факт в базу данных, располагающуюся в оперативной памяти компьютера (резидентная БД). Новый факт помещается перед всеми уже внесенными утверждениями данного предиката. Этот предикат имеет такой синтаксис:

asserta(Clause).

Таким образом, чтобы поместить в БД утверждение

dplayer("Bernie Kosar","Cleveland Browns","QB").

перед уже имеющимся там утверждением

dplayer("Doug Cosbie","Dallas Cowboy","TE"),

стоящим в настоящий момент в базе данных на первом месте, необходимо следующее предикатное выражение:

asserta(dplayer("Bernie Kosar","Cleveland Browns","QB")).

Теперь БД содержит два утверждения, причем утверждение со

сведениями о Kosar предшествует утверждению со сведениями о Cosbie:

dplayer("Bernie Kosar","Cleveland Browns","QB").

dplayer("Doug Cosbie","Dallas Cowboy","TE").

Крайне важно отдавать себе отчет в том, что в динамической базе данных могут содержаться только факты (не правила).

Предикат **assertz** так же, как и **asserta**, заносит новые утверждения в базу данных. Однако он помещает новое утверждение за всеми уже имеющимися в базе утверждениями того же предиката. Синтаксис предиката столь же прост:

assertz(Clause).

Для добаления к двум уже имеющимся в БД утверждениям третьего

dplayer("Mark Malone","Pittsburgh Steelers","QB")

потребуется следующее предикатное выражение:

assertz(dplayer("Mark Malone","Pittsburgh Steelers","QB")).

после чего БД будет содержать третьего утверждения

dplayer("Bernie Kosar","Cleveland Browns","QB").

dplayer("Doug Cosbie","Dallas Cowboy","TE").

dplayer("Mark Malone","Pittsburgh Steelers","QB").

Третье, новое, только что занесенное утверждение, следует за двумя старыми.

Предикат **retract** удаляет утверждение из динамической БД (еще раз напомним, что динамическая БД содержит факты, но не правила.) Его синтаксис таков:

retract(Existing_clause).

Предположим, что вы хотите удалить из базы данных второе утверждение. Для этого необходимо написать выражение

retract(dplayer("Doug Cosbie","Dallas Cowboy","TE")).

и БД будет состоять уже только из двух утверждений:

dplayer("Bernie Kosar","Cleveland Browns","QB").

dplayer("Mark Malone","Pittsburgh Steelers","QB").

Так же, как **asserta** и **assertz, retract** применим только в отношении фактов.

Предикаты **save** и **consult** применяются для записи динамической БД в файл на диск и для загрузки содержимого файла в динамическую БД.

Предикат **save** сохраняет находящуюся в оперативной памяти базу данных в текстовом файле. Синтаксис этого предиката

save(DOS_file_name),

где DOS_file_name есть произвольное допустимое в MS DOS или PC DOS имя файла.

Для того, чтобы сохранить содержимое футбольной БД в файле с именем football.dba, требуется предикат

save("football.dba").

В результате все утверждения находящейся в оперативной памяти динамической БД будут записаны в файл football.dba.

Файл БД может быть считан в память (загружен) при помощи предиката **consult**, систаксис которого таков:

consult(DOS_file_name).

Для загрузки файла футбольной БД требуется выражение

consult("football.dba").

Предикат **consult** неуспешен, если файл с указанным именем отсутствует на диске, или если этот файл содержит ошибки, как, например, в случае несоответствия синтаксиса предиката из файла описаниям из раздела программы **database**, или если содержимое файла невозможно разместить в памяти ввиду отсутствия места.

Предикат **findall** позволяет собрать все имеющиеся в базе данные в список, который может быть полезен при дальнейшей работе. Так, findall можно использовать для получения списка имен всех игроков.

После того, как успешным будет предикат

findall(Name,dplayer(Name,_,_),Name_list)

переменная Name_list содержит список имен всех игроков.

Пример программы добавления, вывода и удаления из базы данных фактов и значений.

```
Add :– write(‘Введите имя
      мужчины: ‘), read_string(S),
      assertz(man(S)).

Out :– write(‘Список имен мужчин:\n’),
      man(S),write(S),nl.

Del :– write('Введите имя мужчины: '),

      read_string(S), retractall(man(S)).

Main :– consult(‘db.pro’),add,out,del,out.
```

Файл db.pro первоначально содержит следующие факты:

```
man(oleg).
woman(zina).
man(alexandr).
man(petr).
```

Задание к работе

Составить программу, реализующую следующие операции с базой данных:

1. Добавление записей
2. Удаление конкретной записи по заданному значению.
3. Поиск заданной записи.
4. Вывод данных из базы данных.

Варианты заданий

Выполнить вариант для заданной предметной области: **I=N(MOD 10)+1.**

1.Библиотека	6.Авиакасса.
2.Отдел кадров	7.Магазин.
3.Отдел договоров.	8.Склад.
4.Деканат.	9.Автотранспортное предприятие.
5.Кафедра.	10. Фирма по продаже компьютеров.

Содержание отчета

1. Исходные тексты и структуры программ на языке Пролог.

2. Примеры обработки запросов

3. Анализ результатов работы программ в режиме трассировки.

4. Выводы по работе.

Контрольные вопросы

1 . Назовите способы представления баз данных в ПРОЛОГ.

2 . Какие есть типы отношений между термами в ПРОЛОГ.

3 . В чем отличие структуры программы для работы с базой данных.

4 . Назовите основные предикаты при работе с базами данных.

5 .Какие типовые операции над базами данных Вам известны.

Лабораторная работа №6

Типовые операции с бинарными деревьями

Цель работы - научиться описывать рекурсивные типы данных, выработать навыки представления структур в виде дерева и реализации типовых процедур с деревьями.

Порядок выполнения работы

1. Решить задачу соответствующего варианта.

2.Описать тестовые примеры и протестировать программу на компьютере.

3. Составить отчет о проделанной работе.

Теоретические положения

Типы данных являются рекурсивными, если они допускают, чтобы в его структуре данных была рекурсия.

Примером рекурсивного типа данных является бинарное дерево. Бинарное дерево либо пусто, либо состоит из следующих трех частей:

корень;

- левое поддерево;

правое поддерево.

Где левое и правое поддеревья сами являются бинарными деревьями. Таким образом, рекурсивность бинарного дерева заложена уже в его определении.

Бинарное дерево упорядочено, если в нем все вершины левого поддерева меньше корня, все вершины правого поддерева больше корня, и оба поддерева также упорядочены. Такое дерево называется бинарным справочником. Преимущество упорядочивания состоит в том, что для поиска некоторого объекта в бинарном справочнике достаточно просмотреть не более одного поддерева.

Стоит заметить, что поиск в двоичном справочнике наиболее эффективен для сбалансированного дерева, то есть дерева, в котором два поддерева содержат примерно равное количество элементов. Если же дерево полностью разбалансировано, то поиск в нем так же неэффективен, как и поиск в списке. Логические программы, работающие с бинарными деревьями, подобны программам, работающим со списками. Как и в случаях натуральных чисел и списков, мы начнем с определения типа бинарного дерева. Оно дается программой 5.1. Отметим, что в программе имеется двойная рекурсия, то есть в теле рекурсивного правила имеются две цели с тем же предикатом, что и в заголовке правила. Этот эффект возникает благодаря двойной рекурсивной природе бинарных деревьев и может быть замечен в остальных программах данного раздела.

Давайте напишем некоторые программы обработки деревьев. Наш первый пример состоит в проверке, появляется ли некоторый элемент в дереве. Реляционная схема – **tree_member(Element,Tree).** Отношение выполнено, если элемент Element является одной из вершин дерева Tree. В программе 5.2 приведено определение. Декларативное понимание программы: "X - элемент дерева, если X находится в вершине (ввиду факта) или X - элемент левого или правого поддерева (ввиду двух рекурсивных правил)".

Две ветви бинарного дерева различны, но во многих приложениях это различие несущественно. Следовательно, возникает полезное понятие изоморфизма, которое определяет, являются ли два неупорядоченных дерева по существу одинаковыми. Два бинарных дерева T1 и T2 изоморфны, если T2 может быть получено из T1 изменением порядка ветвей в поддеревьях.

Изоморфизм является отношением эквивалентности с простым рекурсивным определением. Два пустых дерева изоморфны. В случае непустых деревьев, два дерева изоморфны, если элементы в вершинах дерева совпадают и, или оба левых поддерева и оба правых поддерева изоморфны или левое поддерево одного дерева изоморфно правому поддереву другого и два других поддерева тоже изоморфны.

Программа **5.1.** Определение бинарного дерева

```
% binary_tree(Tree) Tree - бинарное дерево.

binary_tree(empty).
binary_tree(tree(Element,Left,Right)):
— binary_tree(Left),
binary_tree(Right).
```

Программа **5.2.** Проверка принадлежности дереву

```
% tree_member(Element,Tree) Element является
  элементом бинарного дерева Tree.

tree_member(X,tree(X,Left,Right)).

tree_member(X,tree(Y,Left,Right));—
tree_member(X,Left).
tree_member(X,tree(Y,Left,Right));—
tree_member(X,Right).
```

Программа **5.3.** Определение изоморфизма деревьев

```
% isotree(Tree1,Tree2) бинарные деревья Tree1 и Tree2 изоморфны.

isotree(empty, empty).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)):—

   isotree(Left1,Left2), isotree(Right1,Right2).
isotree(tree(X,Left1,Right1),tree(X,Left2,Right2)):—

   isotree(Left1,Right2), isotree(Right1,Left2).
```

Программа 5.3 определяет предикат isotree(Tree1, Tree2), который истинен, если дерево Tree1 изоморфно дереву Tree2. Аргументы входят в предикат симметрично.

Программа **5.4.** Подстановка терма в дерево

```
%substitute(X,Y,TreeX,TreeY) бинарное дерево TreeY - результат
                             замены всех вхождений X в бинарном
                             дереве TreeX на Y.

substitute(X,Y,empty,empty).
substitute(X,Y,tree(X,Left,Right),tree(Y,Left1,Right1)):—

 substitute(X,Y,Left,Left1),

 substitute(X,Y,Right,Right1).
```

```
substitute(X,Y,tree(Z,Left,Right),tree(Z,Left1,Right1)):–

    X=Z,

    substitute(X,Y,Left,Left1),

    substitute(X,Y,Right,Right1).
```

Программы, относящиеся к бинарным деревьям, используют двойную рекурсию, по одной на каждую ветвь дерева. Двойная рекурсия может проявляться двумя способами. В программе могут рассматриваться два отдельных случая, как в программе 5.2 для отношения tree_member. В отличие от этого, программа, проверяющая принадлежность элемента списку, содержит лишь один рекурсивный случай. При другом способе, в теле рекурсивного правила содержатся два рекурсивных вызова, как в каждом рекурсивном правиле для isotree в программе 5.3.

Ранее, одно из заданий состояло в том, чтобы написать программу подстановки элементов в списки. Аналогичная программа может быть написана для подстановки элементов в бинарные деревья. Предикат **substitute(X, Y, OldTree, NewTree)** выполнен, если дерево **NewTree** получается из дерева **OldTree** заменой всех вхождений элемента X на Y. Аксиоматизация отношения substitute/4 приведена в программе 5.4.

Во многих приложениях, использующих деревья, требуется доступ к элементам, указанным в вершинах. Основной является идея обхода дерева в предписанном порядке. Имеется три возможности линейного упорядочения при обходе:

– сверху-вниз, когда сначала идет значение в вершине, далее вершины левого поддерева, затем вершины правого поддерева;

– слева-направо, когда сначала приводятся вершины левого поддерева, затем вершина дерева и далее вершины правого поддерева;

– снизу-вверх, когда значение в вершине приводится после вершин левого и правого поддерева.

Определение каждого из этих трех обходов приведено в программе 5.5.

Программа **5.5.** Обходы бинарного дерева

```
%pre_order(Tree,Pre) Pre - обход бинарного дерева Tree  сверху вниз.

pre_order(tree(X,L,R),Xs):–

   pre_order(L,Ls), pre_order(R,Rs), append([X|Ls],Rs,Xs).
pre_order(empty,[]).
```

```
%in_order(Tree,In)In-обходбинарногодереваTreeслеваналраво.

in_order(tree(X,L,R),Xs):-

  in_order(L,Ls),in_order(R,Rs),
append(Ls,[X|Rs],Xs).in_order(empty,[]).
```

% post_order(Tree,Post) Post - обход бинарного дерева Tree снизу вверх.

```
post_order(tree(X,L,R),Xs):–

  post_order(L,Ls),

  post_order(R,Rs),

  append(Rs,[X],Rs1),

  append(Ls,Rs1,Xs).
post_order(empty,[]).
```

Рекурсивная структура определений одинакова, единственное отличие состоит в порядке элементов, полученных применением целей вида append.

Опишем предикаты для создания и модификации бинарного дерева. Бинарное дерево задается с помощью функтора

tree(K, LeftT, RightT), где K – элемент, находящийся в вершине;

LeftT и RightT – левое и правое поддерево соответственно.

```
create_tree(A, tree(A, empty, empty)).          %созданиедерева

insert_left(X, tree(A, _, B), tree(A, X, B)).   % включение
```
элемента данных A, как левого поддерева B

```
insert_right(X, tree(A, B, _), tree(A, B, X)).  % включение
```
элемента данных A, как правого поддерева B

append – это процедура **append(LeftList, RightList, ListRes),** где ListRes является результатом слияния списков LeftList, RightList.

Фрагмент программы печатает все элементы, проходя его «в глубину».

```
show_tree(nil).

show_tree(tree(X,Left,Right)):-write(X),show_tree(Left),   show_tree(Right).
```

Задание к работе

1. Исследовать на конкретных тестовых данных примеры программ 5.1–

5.5.

2. Решить задачи согласно варианту:

I=N(MOD 7)+1.

3. Подобрать тестовые данные и оформить отчет.

Варианты заданий.

1. Определите предикаты

двдерево(Объект)
справочник(Объект)

распознающие, является ли Объект двоичным деревом или двоичным справочником соответственно. Используйте обозначения, введенные в данном разделе.

2. Определите процедуру

глубина(ДвДерево, Глубина)

вычисляющую глубину двоичного дерева в предположении, что глубина пустого дерева равна 0, а глубина одноэлементного дерева равна 1.

3. Определите отношение

линеаризация(Дерево, Список)

соответствующее "выстраиванию" всех вершин дерева в список.

4. Определите отношение

максэлемент(Д, Элемент)

таким образом, чтобы переменная Элемент приняла значение наибольшего из элементов, хранящихся в дереве Д.

5. Составьте программу subtree(S,T), определяющую, является ли S поддеревом T.

6. Определите отношение sum_tree(TreeOfIntegers,Sum), выполненное, если число Sum равно сумме целых чисел, являющихся вершинами дерева TreeOfIntegers.

7. Определите отношение ordered(Tree), выполненное, если дерево Tree является упорядоченным деревом целых чисел, то есть число стоящее в любой вершине дерева больше любого элемента в левом поддереве и меньше любого элемента в правом поддереве.

(Указание: Определите два вспомогательных отношения **ordered_left(X,Tree)** и **ordered_right(X,T),** выполненных если X меньше (больше) корня дерева Tree и дерево Tree упорядоченно).

Содержание отчета

1. Исходные тексты и структуры программ на языке Пролог в соответствии с заданным вариантом.

2. Примеры обработки запросов

3. Анализ результатов работы программ в режиме трассировки.

4. Выводы по работе.

Контрольные вопросы

1. Дайте определение бинарного дерева.

2. В чем отличия между упорядоченными и сбалансированными деревьями.

3 .Что такое двоично-троичные деревья, AVL-деревья.

4 .Типовые операции над деревьями.

5 .Как представляются графы в ПРОЛОГ, деревья И/ИЛИ.

6 .Чем отличается нисходящий от восходящего вывода в ПРОЛОГ.

7 .Назовите основные стратегии решения задач в ПРОЛОГ.

Библиографический список

1. Брюховецкий А.А. Методические указания к лабораторным занятиям для студентов направления «Компьютерная инженерия - 0915» по курсу «Логическое программирование» на тему «Функциональное программирование». / А.А.Брюховецкий. – СевНТУ, 2002.-14с.

2. Братко И. Программирование на языке Пролог для искусственного интеллекта. – М.: Мир, 1990.

3. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог. – М.: Мир, 1990.

4. Метакидес Г., Нероуд А. Принципы логики и логического программирования. – М.: Факториал, 1998

5. Хоггер К. Введение в логическое программирование. – М.: Мир, 1988.

6. Малпас Дж. Реляционный язык Пролог и его применение. /Малпас Дж. ; Пер. с англ.; — М.: Наука, 1990. –574с.

7. Ин Ц., Соломон Д. Использование Пролога. – М., 1990.

8. Чень Ч., Ли Р. Математическая логика и автоматическое доказательство теорем. – М.: Наука, 1983.

Ответственный за выпуск зав. кафедрой кибернетики
и вычислительной техники Скатков А.В. д.т.н., профессор

Допущено научно-методическим центром университета
в качестве методических указаний