

**Логическое программирование на языке Visual Prolog  
Учебное пособие**

**Составители:  
О.П. Солдатова, И.В.Лёзина**

**Самара 2010**

**УДК 004.89(075.8)**

**Рецензенты:**

- заведующий кафедрой «Программные системы» Самарского государственного аэрокосмического университета имени академика С.П.Королёва, д.т.н., профессор А.Н. Коварцев
- заведующий кафедрой «Прикладная математика и вычислительная техника» Самарского государственного архитектурно-строительного университета, д.т.н., профессор С.А. Пиявский

О.П. Солдатова, И.В. Лёзина

Логическое программирование на языке Visual Prolog: учебное пособие – Самара: СНЦ РАН, 2010 –81 с., ил.

**ISBN – 978-593424-486-7**

Данное пособие предназначено для студентов направления 010400 «Информационные технологии» и специальности 230102 «Автоматизированные системы обработки информации и управления», и может быть использовано при изучении дисциплин «Логическое программирование», «Интеллектуальные системы», «Системы искусственного интеллекта» и «Основы экспертных систем».

*Печатается по решению издательского совета  
Самарского научного центра Российской академии наук*

**ISBN - 978-593424-486-7**

© О.П. Солдатова, И.В. Лёзина, 2010

## Содержание

Предисловие.....	5
1 Логическое программирование и аксиоматические системы .....	5
1.1 Общие положения .....	5
1.2 Автоматизация доказательства в логике предикатов.....	7
1.2.1 История вопроса.....	7
1.2.2 Скулемовские стандартные формы.....	8
1.2.3 Метод резолюций в исчислении высказываний.....	11
1.2.4 Правило унификации в логике предикатов.....	13
1.2.5 Метод резолюций в исчислении предикатов .....	15
2 Введение в язык логического программирования ПРОЛОГ.....	16
2.1 Общие положения .....	16
2.2 Основы языка программирования Пролог .....	17
2.3 Использование дизъюнкции и отрицания.....	21
2.4 Унификация в Прологе.....	21
2.5 Вычисление цели. Механизм возврата.....	22
2.6 Управление поиском решения.....	24
2.7 Процедурность Пролога.....	25
2.8 Структура программ Пролога.....	26
2.9 Использование составных термов.....	28
2.10 Использование списков .....	30
2.11 Применение списков в программах .....	32
2.11.1 Поиск элемента в списке .....	32
2.11.2 Объединение двух списков .....	33
2.11.3 Определение длины списка.....	34
2.11.4 Поиск максимального и минимального элемента в списке.....	35
2.11.5 Сортировка списков .....	35
2.11.6 Компоновка данных в список .....	37
2.12 Повторение и рекурсия в Прологе .....	38
2.12.1 Механизм возврата.....	38
2.12.2 Метод возврата после неудачи .....	39
2.12.3 Метод повтора, использующий бесконечный цикл .....	41
2.13 Методы организации рекурсии.....	42
2.14 Создание динамических баз данных .....	45
2.15 Использование строк в Прологе.....	49
2.16 Преобразование данных в Прологе.....	51
2.17 Представление бинарных деревьев .....	52
2.18 Представление графов в языке Пролог.....	55
2.19 Поиск пути на графе.....	57
2.20 Метод “образовать и проверить” .....	59
3 Основные стратегии решения задач. Поиск решения в пространстве состояний .....	62
3.1 Понятие пространства состояния .....	62
3.2 Основные стратегии поиска решений в пространстве состояний .....	64

3.2.1 Поиск в глубину .....	64
3.2.2 Поиск в ширину .....	67
3.3 Сведение задачи к подзадачам и И/ИЛИ графы. ....	71
3.4 Решение игровых задач в терминах И/ИЛИ- графа .....	74
3.5 Минимаксный принцип поиска решений .....	76
Литература .....	80

## Предисловие

Язык Пролог был создан как язык программирования для решения задач искусственного интеллекта. Языку Пролог посвящались многие книги и статьи в журналах, однако все они описывали разные версии языка, и только одна монография была посвящена языку Visual Prolog – книга А.Н. Адаменко и А.М.Кучукова «Логическое программирование и Visual Prolog» – вышла в 2003 году и в настоящее время стала библиографической редкостью.

За последние годы в стандарты многих специальностей и направлений подготовки бакалавров и магистров, связанных с информационными технологиями, введены курсы, освоение которых предполагает получение навыков логического программирования. Для студентов, изучающих логическое программирование, и предназначено данное учебное пособие.

В пособии содержится краткое описание математических основ логического программирования, введение в язык программирования Visual Prolog и изложение наиболее известных методов и алгоритмов решения интеллектуальных задач. В пособии приведено множество примеров программ на языке Visual Prolog, иллюстрирующих описываемые методы и алгоритмы.

### 1 Логическое программирование и аксиоматические системы

#### 1.1 Общие положения

Теория формальных систем и, в частности, математическая логика являются формализацией человеческого мышления и представления наших знаний. Если предположить, что можно аксиоматизировать наши знания и можно построить алгоритм, позволяющий реализовать процесс вывода ответов на запрос из знаний, то в результате можно получить формальный метод для получения неформальных результатов.

Логическое программирование возникло в эру ЭВМ как естественное желание автоматизировать процесс логического вывода, поэтому оно является ветвью теории формальных систем.

Логическое программирование (в широком смысле) представляет собой семейство таких методов решения задач, в которых используются приемы логического вывода для манипулирования знаниями, представленными в декларативной форме [1]. Как писал Джордж Робинсон в 1984 году, в основе идеи логического программирования лежит описание задачи совокупностью утверждений на некотором формальном логическом языке и получение решения с помощью вывода в некоторой формальной (аксиоматической) системе. Такой аксиоматической системой являются исчисление предикатов первого порядка, поэтому в узком смысле логическое программирование понимается как использование исчисления предикатов

первого порядка в качестве основы для описания предметной области и осуществления резолюционного логического вывода.

*Аксиоматической системой* называется способ задания множества путем указания исходных элементов (аксиом исчисления) и правил вывода, каждое из которых описывает, как строить новые элементы из исходных элементов.

Любая аксиоматическая система должна удовлетворять следующим требованиям:

1. Непротиворечивость: невозможность вывода отрицания уже доказанного выражения (которое считается общезначимым);
2. Независимость (минимальность): система не должна содержать бесполезных аксиом и правил вывода. Некоторое выражение *независимо* от аксиоматической системы, если его нельзя вывести с помощью этой системы. В минимальной системе каждая аксиома независима от остальной системы, то есть, не выводима из других аксиом.
3. Полнота (взаимность адекватности): любая тавтология выводима из системы аксиом. В адекватной системе аксиом любая выводимая формула есть тавтология, то есть верно, что  $\vdash P \rightarrow \vdash P$ . Соответственно в *полной системе* верно:  $\vdash P \rightarrow \vdash \neg P$ .

Исчислениями называют наиболее важные из аксиоматических логических систем – исчисление высказываний и исчисление предикатов. Исчисление высказываний и исчисление предикатов первого порядка являются *полными аксиоматическими системами*.

Под *аксиоматическим методом* [1] понимают способ построения научной теории, при котором за ее основу берется ряд основополагающих, не требующих доказательств положений этой теории, называемыми аксиомами или постулатами.

Аксиоматический метод зародился в работах древнегреческих геометров. Вплоть до начала XIX века единственным образцом применения этого метода была геометрия Евклида.

В начале XIX века Н.И.Лобачевский и Я.Больяй, независимо друг от друга, открыли новую неевклидову геометрию, заменив пятый постулат о параллельных прямых на его отрицание. Их открытие стало отправной точкой для развития аксиоматического метода, который лег в основу теории формальных систем.

Формальная теория строится как четко определенный класс выражений, формул, в котором некоторым точным способом выделяется подкласс теорем данной формальной системы. При этом формулы формальной системы непосредственно не несут в себе никакого содержательного смысла, они строятся из произвольных знаков или символов, исходя лишь из соображений удобства.

*Формальная теория (система)*, задается четверкой вида  $M = \langle T, S, A, B \rangle$ . Множество  $T$  есть множество *базовых элементов*, например слов из

некоторого словаря, или деталей из некоторого набора. Для множества  $T$  существует некоторый способ определения принадлежности или непринадлежности произвольного элемента к данному множеству. Процедура такой проверки может быть любой, но она должна давать ответ на вопрос, является ли  $x$  элементом множества  $T$  за конечное число шагов. Обозначим эту процедуру  $P(T)$ .

Множество  $S$  есть множество синтаксических правил. С их помощью из элементов  $T$  образуют синтаксически правильные совокупности. Например, из слов словаря строятся синтаксически правильные фразы, а из деталей собираются конструкции. Существует некоторая процедура  $P(S)$ , с помощью которой за конечное число шагов можно получить ответ на вопрос, является ли совокупность  $X$  синтаксически правильной.

Во множестве синтаксически правильных совокупностей выделяется некоторое подмножество  $A$ . Элементы  $A$  называются аксиомами. Как и для других составляющих формальной системы, должна существовать процедура  $P(A)$ , с помощью которой для любой синтаксически правильной совокупности можно получить ответ на вопрос о принадлежности ее к множеству  $A$ .

Множество  $B$  есть множество правил вывода. Применяя их к элементам  $A$ , можно получать новые синтаксически правильные совокупности, к которым снова можно применять правила из  $B$ . Так формируется множество выводимых в данной формальной системе совокупностей. Если имеется процедура  $P(B)$ , с помощью которой можно определить для любой синтаксически правильной совокупности, является ли она выводимой, то соответствующая формальная система называется *разрешимой*. Это показывает, что именно правила вывода являются наиболее сложной составляющей формальной системы.

Исчисление высказываний является *разрешимой формальной системой*, а исчисление предикатов первого порядка – *неразрешимой формальной системой*.

С накоплением опыта построения формальных теорий и попытками аксиоматизации арифметики, предпринятыми Дж. Пеано, возникла *теория доказательств*. Теория доказательств – это раздел современной математической логики и предшественница логического программирования.

## **1.2 Автоматизация доказательства в логике предикатов.**

### **1.2.1 История вопроса**

Поиск общей разрешающей процедуры для проверки общезначимости формул был начат Лейбницем в XVII веке. Затем исследования продолжились в XX веке, а в 1936 году Черч и Тьюринг независимо друг от друга доказали, что не существует никакой общей разрешающей процедуры, никакого алгоритма, проверяющего общезначимость формул в логике предикатов первого порядка.

Тем не менее, существуют алгоритмы поиска доказательства, которые могут подтвердить, что формула общезначима, если она на самом деле общезначима (для необщезначимых формул эти алгоритмы, вообще говоря, не заканчивают свою работу).

Очень важный подход к автоматическому доказательству теорем был дан Эрбраном в 1930 году. По определению общезначимая формула есть формула, которая истинна при всех интерпретациях. Эрбран разработал алгоритм нахождения интерпретации, которая опровергает данную формулу. Однако, если данная формула действительно общезначима, то никакой интерпретации не существует и алгоритм заканчивает работу за конечное число шагов. Метод Эрбрана служит основой для большинства современных автоматических алгоритмов поиска доказательства.

Гилмор в 1959 году одним из первых реализовал процедуру Эрбрана. Его программа была предназначена для обнаружения противоречивости отрицания данной формулы, так как формула общезначима тогда и только тогда, когда ее отрицание противоречиво. Однако программа Гилмора оказалась неэффективной и в 1960 году метод Гилмора был улучшен Девисом и Патнемом. Но их улучшение оказалось недостаточным, так как многие общезначимые формулы логики предикатов все еще не могли быть доказаны на ЭВМ за разумное время.

Главный шаг вперед сделал Робинсон в 1965 году, который ввел так называемый метод резолюций, который оказался много эффективней, чем любая описанная ранее процедура. После введения метода резолюций был предложен ряд стратегий для увеличения его эффективности. Такими стратегиями являются *семантическая резолюция, лок-резолюция, линейная резолюция, стратегия предпочтения единичных и стратегия поддержки*.

### **1.2.2 Скулемовские стандартные формы.**

Процедуры доказательства по Эрбрану или методу резолюций на самом деле являются процедурами опровержения, то есть вместо доказательства общезначимости формулы доказываемся, что ее отрицание противоречиво. Кроме того, эти процедуры опровержения применяются к некоторой стандартной форме, которая была введена Девисом и Патнемом. По существу они использовали следующие идеи:

1. Формула логики предикатов может быть сведена к ПНФ, в которой матрица не содержит никаких кванторов, а префикс есть последовательность кванторов.
2. Поскольку матрица не содержит кванторов, она может быть сведена к конъюнктивной нормальной форме.
3. Сохраняя противоречивость формулы, в ней можно исключить кванторы существования путем использования скулемовских функций.



Алгоритм преобразования формулы в ПНФ известен. При помощи законов эквивалентных преобразований логики высказываний можно свести матрицу к КНФ.

*Алгоритм преобразования формул в ДНФ и КНФ.*

*Шаг 1.* Используем законы эквивалентных преобразований исчисления высказываний для того, чтобы исключить логические связки импликации и эквивалентности.

*Шаг 2.* Многократно используем закон двойного отрицания, и законы де Моргана, чтобы внести знак отрицания внутрь формулы.

*Шаг 3.* Несколько раз используем дистрибутивные законы и другие законы, чтобы получить НФ.

Алгоритм преобразования формулы  $(K_1x_1)...(K_nx_n) (M)$ , где каждое  $(K_ix_i)$ ,  $i = 1, \dots, n$ , есть или  $(\forall x_i)$  или  $(\exists x_i)$ , и  $M$  есть КНФ в сколемовскую нормальную форму (СНФ) приведен ниже.

*Алгоритм преобразования ПНФ в ССФ.*

*Шаг 1.* Представим формулу в ПНФ  $(K_1x_1)...(K_nx_n) (M)$ , где  $M$  есть КНФ. Пусть  $K_r$  есть квантор существования в префиксе  $(K_1x_1)...(K_nx_n)$ ,  $1 \leq r \leq n$ .

*Шаг 2.* Если никакой квантор всеобщности не стоит левее  $K_r$  – выберем новую константу  $c$ , отличную от других констант, входящих в  $M$ , заменим все  $x_r$  в  $M$  на  $c$  и вычеркнем  $K_rx_r$  из префикса. Если  $K_1, \dots, K_i$  – список всех кванторов всеобщности, встречающихся в  $M$  левее  $K_r$ ,  $1 < i < r$ , выберем новый  $i$ -местный функциональный символ  $f_i$ , отличный от других функциональных символов, заменим все  $x_r$  в  $M$  на  $f_i(x_1, x_2, \dots, x_i)$  и вычеркнем  $K_rx_r$  из префикса.

*Шаг 3.* Применим шаг 2 для всех кванторов существования в префиксе. Последняя из полученных формул есть *сколемовская стандартная форма* формулы. Константы и функции, используемые для замены переменных квантора существования, называются *сколемовскими функциями*.

*Пример 1.* Получить ССФ для формулы  $(\exists x)(\forall y)(\forall z)(\exists u)(\forall v)(\exists w) (P(x, y, z, u, v, w))$ .

В этой формуле левее  $(\exists x)$  нет никаких кванторов всеобщности, левее  $(\exists u)$  стоят  $(\forall y)$  и  $(\forall z)$ , а левее  $(\exists w)$  стоят  $(\forall y)$ ,  $(\forall z)$  и  $(\forall v)$ . Следовательно, мы заменим переменную  $x$  на константу  $a$ , переменную  $u$  - на двухместную  $f(y, z)$ , переменную  $w$  - на трехместную функцию  $g(y, z, v)$ . Таким образом, мы получаем следующую стандартную форму написанной выше формулы:

$(\forall y)(\forall z)(\forall v)(P(a, y, z, f(y, z), g(y, z, v)))$ .

*Определение 1:* Дизъюнктом называется дизъюнкция литералов. Дизъюнкт, содержащий  $r$  литералов, называется  $r$ -литеральным дизъюнктом. Однолитеральный дизъюнкт называется единичным дизъюнктом. Если дизъюнкт не содержит никаких литералов, то он

называется пустым дизъюнктом-  $\square$  . Так как пустой дизъюнкт не содержит литералов, которые могли бы быть истинными при любых интерпретациях, то пустой дизъюнкт всегда ложен.

*Определение 2:* Множество дизъюнктов  $S$  есть конъюнкция всех дизъюнктов из  $S$  , где каждая переменная в  $S$  считается управляемой квантором всеобщности.

Вследствие последнего определения, ССФ может быть представлена множеством дизъюнктов.

*Пример 2.* Получить скелемовскую стандартную форму формулы

$(\forall x)(P(x) \wedge (\forall y)(\neg(\forall z)Q(x, y) \rightarrow (\exists u)R(u, x, y)))$  и представить её в виде множества дизъюнктов.

1. Исключим связи импликации:

$$(\forall x)(P(x) \wedge (\forall y)(\neg \neg(\forall z)Q(x, y) \vee (\exists u)R(u, x, y))).$$

2. Удалим бесполезные кванторы:

$$(\forall x)(P(x) \wedge (\forall y)(\neg \neg Q(x, y) \vee (\exists u)R(u, x, y))).$$

3. Применим правило двойного отрицания:

$$(\forall x)(P(x) \wedge (\forall y)(Q(x, y) \vee (\exists u)R(u, x, y))).$$

4. Переместим кванторы в начало формулы:

$$(\forall x)(\forall y)(\exists u)(P(x) \wedge (Q(x, y) \vee R(u, x, y))),$$

Получим ПНФ.

$(\forall x)(\forall y)(\exists u)(P(x) \wedge (Q(x, y) \vee R(u, x, y)))$ , у которой матрица находится в КНФ.

Избавимся от кванторов существования в префиксе:

Так как перед  $(\exists u)$  есть  $(\forall x), (\forall y)$  то переменная  $u$  заменяется двухместной функцией  $f(x, y)$ . Таким образом, мы получаем следующую стандартную форму:

$$(\forall x)(\forall y)(P(x) \wedge (Q(x, y) \vee R(f(x, y), x, y))).$$

Получим ССФ.

Отбросим кванторы всеобщности и заменим конъюнкцию на перечисление:

$$\{P(x), (Q(x, y) \vee R(f(x, y), x, y))\}.$$

Получим множество из двух дизъюнктов.

*Пример 3.* Получить скелемовскую стандартную форму формулы

$$(\forall x)(\exists y)(\exists z)((\neg P(x, y) \wedge Q(x, z)) \vee R(x, y, z) \wedge (\neg Q(x, z) \wedge P(x, y))).$$

Сначала сведем матрицу к КНФ:

$$((\neg P(x, y) \vee R(x, y, z)) \wedge (Q(x, z) \vee R(x, y, z)) \wedge (\neg Q(x, z) \wedge P(x, y))).$$

Затем избавимся от кванторов существования в префиксе:

Так как перед  $(\exists y)(\exists z)$  есть  $(\forall x)$ , то переменные  $y, z$  заменяются соответственно одноместными функциями  $f(x), g(x)$ . Таким образом, мы получаем следующую стандартную форму:

$$(\forall x)((\neg P(x, f(x)) \vee R(x, f(x), g(x))) \wedge (Q(x, g(x)) \vee R(x, f(x), g(x))) \wedge (\neg Q(x, g(x)) \wedge P(x, f(x)))).$$

Представим полученную ССФ в виде множества дизъюнктов:

$\{-P(x, f(x)) \vee R(x, f(x), g(x)), Q(x, g(x)) \vee R(x, f(x), g(x)), \neg Q(x, g(x)) \wedge P(x, f(x))\}$ .

*Теорема 1.* Пусть  $S$  – множество дизъюнктов, которые представляют ССФ формулы  $F$ . Тогда  $F$  противоречива в том и только в том случае, когда  $S$  противоречиво.

*Теорема 2.* Пусть даны формулы  $F_1, F_2, \dots, F_n$  и формула  $G$ .  $G$  есть логическое следствие  $F_1, F_2, \dots, F_n$  тогда и только тогда, когда формула  $((F_1 \wedge F_2 \wedge \dots \wedge F_n) \rightarrow G)$  общезначима.

*Теорема 3.* Пусть даны формулы  $F_1, F_2, \dots, F_n$  и формула  $G$ .  $G$  есть логическое следствие  $F_1, F_2, \dots, F_n$  тогда и только тогда, когда формула  $(F_1 \wedge F_2 \wedge \dots \wedge F_n \wedge \neg G)$  противоречива.

*Замечание.*

Для того чтобы доказать, что данная формула является тавтологией, достаточно доказать, что ее отрицание является противоречием:

$$\begin{aligned} \neg((F_1 \wedge F_2 \wedge \dots \wedge F_n) \rightarrow G) &= \\ \neg(\neg(F_1 \wedge F_2 \wedge \dots \wedge F_n) \vee G) &= \\ (F_1 \wedge F_2 \wedge \dots \wedge F_n) \wedge \neg G. & \end{aligned}$$

На основании теорем 1 и 3 можно сделать вывод, что формула  $G$  является логическим следствием формулы  $F$  тогда, когда противоречива конъюнкция множества  $S$  и формулы  $\neg G$ , то есть противоречива формула  $S_1 \wedge S_2 \wedge \dots \wedge S_n \wedge \neg G$ . Таким образом, если в множество  $S$  добавить негативный литерал  $\neg G$  и доказать, что полученное множество противоречиво, то тем самым можно доказать выводимость  $G$  из множества  $S$ .

### 1.2.3 Метод резолюций в исчислении высказываний.

Основная идея метода резолюций состоит в том, чтобы проверить, содержит ли множество дизъюнктов пустой дизъюнкт. Если множество содержит пустой дизъюнкт, то оно противоречиво (невыполнимо). Если множество не содержит пустой дизъюнкт, то проверяется следующий факт: может ли пустой дизъюнкт быть получен из данного множества. Множество содержит пустой дизъюнкт, тогда и только тогда, когда оно пустое. Если множество можно свести к пустому, то тем самым можно доказать его противоречивость. В этом и состоит метод резолюций, который часто рассматривают как специальное правило вывода, используемое для порождения новых дизъюнктов из данного множества.

*Определение 3:* Если  $A$  атом, то литералы  $A$  и  $\neg A$  контрарны друг другу, и множество  $\{A, \neg A\}$  называется контрарной парой.

Отметим, что дизъюнкт есть тавтология, если он содержит контрарную пару.

*Определение 4:* Правило резолюций состоит в следующем:

Для любых двух дизъюнктов  $C_1$  и  $C_2$ , если существует литерал  $L_1$  в  $C_1$ , который контрарен литералу  $L_2$  в  $C_2$ , то вычеркнув  $L_1$  и  $L_2$  из  $C_1$  и  $C_2$  соответственно и построив дизъюнкцию оставшихся дизъюнктов, получим резолюцию (резольвенту)  $C_1$  и  $C_2$ .

Пример 4: рассмотрим следующие дизъюнкты:

$$C_1: P \vee R,$$

$$C_2: \neg P \vee Q.$$

Дизъюнкт  $C_1$  имеет литерал  $P$ , который контрарен литералу  $\neg P$  в  $C_2$ . Следовательно, вычеркивая  $P$  и  $\neg P$  из  $C_1$  и  $C_2$  соответственно, построим дизъюнкцию оставшихся дизъюнктов  $R$  и  $Q$  и получим резольвенту  $R \vee Q$ .

Важным свойством резольвенты является то, что любая резольвента двух дизъюнктов  $C_1$  и  $C_2$  есть логическое следствие  $C_1$  и  $C_2$ . Это устанавливается в следующей теореме.

*Теорема 4.* Пусть даны два дизъюнкта  $C_1$  и  $C_2$ . Тогда резольвента  $C$  дизъюнктов  $C_1$  и  $C_2$  есть логическое следствие  $C_1$  и  $C_2$ .

Если есть два единичных дизъюнкта, то их резольвента, если она существует, есть пустой дизъюнкт  $\square$ . Более существенно, что для невыполнимого множества дизъюнктов многократным применением правила резолюций можно породить  $\square$ .

*Определение 5:* Пусть  $S$  – множество дизъюнктов. Резолютивный вывод  $C$  из  $S$  есть такая конечная последовательность  $C_1, C_2, \dots, C_k$  дизъюнктов, что каждый  $C_i$  или принадлежит  $S$  или является резольвентой дизъюнктов, предшествующих  $C_i$ , и  $C_k = C$ . Вывод  $\square$  из  $S$  называется опровержением (или доказательством невыполнимости)  $S$ .

Пример 5. Рассмотрим множество  $S$ :

$$1. \neg P \vee Q,$$

$$2. \neg Q,$$

$$3. P.$$

Из 1 и 2 получим резольвенту

$$4. \neg P.$$

Из 4 и 3 получим резольвенту

$$5. \square.$$

Так как  $\square$  получается из  $S$  применениями правила резолюций, то согласно теореме 4  $\square$  есть логическое следствие  $S$ , следовательно,  $S$  невыполнимо.

Метод резолюций является наиболее эффективным в случае применения его к множеству Хорновских дизъюнктов.

*Определение 6:* Фразой называется дизъюнкт, у которого негативные литералы размещаются после позитивных литералов в конце дизъюнкта.

$$\text{Пример 6: } P_1 \vee P_2 \vee \dots \vee P_n \vee \neg N_1 \vee \neg N_2 \dots \vee \neg N_m$$

*Определение 28:* Фраза Хорна это фраза, содержащая только один позитивный литерал.

Пример 7: преобразовать фразу Хорна в обратную импликацию.

$$\begin{aligned}
& P \vee \neg N_1 \vee \neg N_2 \dots \vee \neg N_m \\
& \neg N_1 \vee \neg N_2 \dots \vee \neg N_m \equiv \neg (N_1 \wedge N_2 \wedge \dots \wedge N_m) \\
& P \leftarrow (N_1 \wedge N_2 \wedge \dots \wedge N_m) \\
& P \leftarrow N_1, N_2, \dots, N_m
\end{aligned}$$

При представлении дизъюнктов фразами Хорна негативные литералы соответствуют гипотезам, а позитивный литерал представляет заключение. Единичный позитивный дизъюнкт представляет некоторый факт, то есть заключение, не зависящее ни от каких гипотез. Часто задача состоит в том, что надо проверить некоторую формулу, называемую целью, логически выведенную из множества правил и фактов. Резолюция является методом доказательства от противного: исходя из фактов, правил и отрицания цели, приходим к противоречию (пустому дизъюнкту).

#### 1.2.4 Правило унификации в логике предикатов.

Правило резолюций предполагает нахождение в дизъюнкте литерала, контрарного литералу в другом дизъюнкте. Для дизъюнктов логики высказываний это очень просто. Для дизъюнктов логики предикатов процесс усложняется, так как дизъюнкты могут содержать функции, переменные и константы.

*Пример 8. Рассмотрим дизъюнкты:*

$$\begin{aligned}
C_1: & P(y) \vee Q(y), \\
C_2: & \neg P(f(x)) \vee R(x).
\end{aligned}$$

*Не существует никакого литерала в  $C_1$ , контрарного какому-либо литералу в  $C_2$ . Однако, если подставить  $f(a)$  вместо  $y$  в  $C_1$  и  $a$  вместо  $x$  в  $C_2$ , то исходные дизъюнкты примут вид:*

$$\begin{aligned}
C_1': & P(f(a)) \vee Q(f(a)), \\
C_2': & \neg P(f(a)) \vee R(a).
\end{aligned}$$

*Так как  $P(f(a))$  контрарен  $\neg P(f(a))$ , то можно получить резольвенту*

$$C_3': Q(f(a)) \vee R(a).$$

*В общем случае, подставив  $f(x)$  вместо  $y$  в  $C_1$ , получим*

$$C_1'': P(f(x)) \vee Q(f(x)).$$

*Литерал  $P(f(x))$  в  $C_1''$  контрарен литералу  $\neg P(f(x))$  в  $C_2$ . Следовательно, можно получить резольвенту*

$$C_3: Q(f(x)) \vee R(x).$$

Таким образом, если подставлять подходящие термы вместо переменных в исходные дизъюнкты, можно порождать новые дизъюнкты. Отметим, что дизъюнкт  $C_3$  из примера 8 является наиболее общим дизъюнктом в том смысле, что все другие дизъюнкты, порожденные правилом резолюции будут частным случаем данного дизъюнкта.

*Определение 7: Подстановка  $\theta$  это конечное множество вида  $\{t_1/v_1, \dots, t_n/v_n\}$ , где каждая  $v_i$  – переменная, каждый  $t_i$  – терм, отличный от  $v_i$ , все  $v_i$  различны.*

*Определение 8:* Подстановка  $\theta$  называется унификатором для множества  $\{E_1, \dots, E_k\}$  тогда и только тогда, когда  $E_1\theta = E_2\theta = \dots = E_k\theta$ . Множество  $\{E_1, \dots, E_k\}$  унифицируемо, если для него существует унификатор.

Прежде чем применить правило резолюции в исчислении предикатов переменные в литералах необходимо унифицировать.

Унификация производится при следующих условиях:

1. Если термы константы, то они унифицируемы тогда и только тогда, когда они совпадают.
2. Если в первом дизъюнкте терм переменная, а во втором константа, то они унифицируемы, при этом вместо переменной подставляется константа.
3. Если терм в первом дизъюнкте переменная и во втором дизъюнкте терм тоже переменная, то они унифицируемы.
4. Если в первом дизъюнкте терм переменная, а во втором - употребление функции, то они унифицируемы, при этом вместо переменной подставляется употребление функции.
5. Унифицируются между собой термы, стоящие на одинаковых местах в одинаковых предикатах.

*Пример 9.* Рассмотрим дизъюнкты:

1.  $Q(a, b, c)$  и  $Q(a, d, l)$ . Дизъюнкты не унифицируемы.
2.  $Q(a, b, c)$  и  $Q(x, y, z)$ . Дизъюнкты унифицируемы. Унификатор -  $Q(a, b, c)$ .

*Определение 9:* Унификатор  $\sigma$  для множества  $\{E_1, \dots, E_k\}$  будет наиболее общим унификатором тогда и только тогда, когда для каждого унификатора  $\theta$  для этого множества существует такая подстановка  $\lambda$ , что  $\theta = \sigma \circ \lambda$ , то есть  $\theta$  является композицией подстановок  $\sigma$  и  $\lambda$ .

*Определение 10:* Композицией подстановок  $\sigma$  и  $\lambda$  есть функция  $\sigma \circ \lambda$ , определяемая следующим образом  $(\sigma \circ \lambda)[t] = \sigma[\lambda[t]]$ , где  $t$  – терм,  $\sigma$  и  $\lambda$  – подстановки, а  $\lambda[t]$  – терм, который получается из  $t$  путем применения к нему подстановки  $\lambda$ .

*Определение 11:* Множество рассогласований непустого множества дизъюнктов  $\{E_1, \dots, E_k\}$  получается путем выявления первой (слева) позиции, на которой не для всех дизъюнктов из  $E$  стоит один и тот же символ, и выписывания из каждого дизъюнкта терма, который начинается с символа, занимающего данную позицию. Множество термов и есть множество рассогласований в  $E$ .

*Пример 10.* Рассмотрим дизъюнкты:

$\{P(x, f(y, z)), P(x, a), P(x, g(h(k(x))))\}$ .

Множество рассогласований состоит из термов, которые начинаются с пятой позиции и представляет собой множество  $\{f(x, y), a, g(h(k(x)))\}$ .

Алгоритм унификации для нахождения наиболее общего унификатора.

Пусть  $E$  – множество дизъюнктов,  $D$  – множество рассогласований,  $k$  – номер итерации,  $\sigma_k$  наиболее общий унификатор на  $k$ -ой итерации.

*Шаг 1.* Присвоим  $k=0$ ,  $\sigma_k=e$  (пустой унификатор),  $E_k=E$ .

*Шаг 2.* Если для  $E_k$  не существует множества рассогласований  $D_k$ , то остановка:  $\sigma_k$  – наиболее общий унификатор для  $E$ . Иначе найдем множество рассогласований  $D_k$ .

*Шаг 3.* Если существуют такие элементы  $v_k$  и  $t_k$  в  $D_k$ , что  $v_k$  переменная, не входящая в терм  $t_k$ , то перейдем к шагу 4. В противном случае остановка:  $E$  не унифицируемо.

*Шаг 4.* Пусть  $\sigma_{k+1}=\sigma_k \{ t_k / v_k \}$ , заменим во всех дизъюнктах  $E_k$   $t_k$  на  $v_k$ .

*Шаг 5.*  $K=k+1$ . Перейти к шагу 2.

*Пример 11.* Рассмотрим дизъюнкты:

$E=\{P(f(a), g(x)), P(y, y)\}$ .

1.  $E_0=E, k=0, \sigma_0=e$ .

2.  $D_0=\{f(a), y\}, v_0=y, t_0=f(a)$ .

3.  $\sigma_1=\{f(a)/y\}, E_1=\{P(f(a), g(x)), P(f(a), f(a))\}$ .

4.  $D_1=\{g(x), f(a)\}$ .

5. Нет переменной в множестве рассогласований  $D_1$ .

Следовательно, алгоритм унификации завершается, множество  $E$  – не унифицируемо.

### 1.2.5 Метод резолюций в исчислении предикатов

С помощью унификации можно распространить правило резолюций на исчисление предикатов. При унификации возникает одна трудность: если один из термов есть переменная  $x$ , а другой терм содержит  $x$ , но не сводится к  $x$ , унификация невозможна. Проблема решается путем переименования переменных таким образом, чтобы унифицируемые дизъюнкты не содержали одинаковых переменных.

*Определение 12:* Если два или более литерала (с одинаковым знаком) дизъюнкта  $C$  имеют наиболее общий унификатор  $\sigma$ , то  $C\sigma$  – называется склейкой  $C$ .

*Пример 12.* Рассмотрим дизъюнкты:

Пусть  $C = P(x) \vee P(f(y)) \vee \neg Q(x)$ .

Тогда 1 и 2 литералы имеют наиболее общий унификатор  $\sigma = \{f(y)/x\}$ . Следовательно,  $C\sigma = P(f(y)) \vee \neg Q(f(y))$  есть склейка  $C$ .

*Определение 13:* Пусть  $C_1$  и  $C_2$  – два дизъюнкта, которые не имеют никаких общих переменных. Пусть  $L_1$  и  $L_2$  – два литерала в  $C_1$  и  $C_2$  соответственно. Если  $L_1$  и  $\neg L_2$  имеют наиболее общий унификатор  $\sigma$ , то дизъюнкт  $(C_1\sigma - L_1\sigma) \cup (C_2\sigma - L_2\sigma)$  называется резольвентой  $C_1$  и  $C_2$ .

*Пример 13.* Пусть  $C_1 = P(x) \vee Q(x)$  и  $C_2 = \neg P(a) \vee R(x)$ . Так как  $x$  входит в  $C_1$  и  $C_2$ , то мы заменим переменную в  $C_2$  и пусть  $C_2 = \neg P(a) \vee R(y)$ . Выбираем  $L_1 = P(x)$  и  $L_2 = \neg P(a)$ .  $L_1$  и  $L_2$  имеют наиболее общий унификатор  $\sigma = \{a/x\}$ . Следовательно,  $Q(a) \vee R(y)$  – резольвента  $C_1$  и  $C_2$ .

*Пример 14. Доказать, что формула  $R(a, z)$  выводима из множества дизъюнктов  $S = \{-P(x, f(x)) \vee R(x, f(x), g(x)), Q(x, g(x)) \vee R(x, f(x), g(x)), \neg Q(x, g(x)) \wedge P(x, f(x))\}$ .*

*Пусть  $C_1 = -P(x, f(x)) \vee R(x, f(x))$ ,  $C_2 = Q(x, g(x)) \vee R(x, f(x))$ ,  $C_3 = -Q(x, g(x)) \vee P(x, f(x))$ . Добавим к множеству  $S$  единичный дизъюнкт  $C_4 = -R(a, z)$ .*

*Так как в дизъюнктах  $C_1$ ,  $C_2$ ,  $C_3$  есть одинаковая переменная  $x$ , то заменим её в  $C_2$  на  $y$ , а в  $C_3$  на  $u$ . Таким образом, решение исходной задачи сводится к доказательству противоречивости следующего множества дизъюнктов:*

$$C_1 = -P(x, f(x)) \vee R(x, f(x));$$

$$C_2 = Q(y, g(y)) \vee R(y, f(y));$$

$$C_3 = -Q(u, g(u)) \vee P(u, f(u));$$

$$C_4 = -R(a, z).$$

*Найдём резольвенту  $C_5$  дизъюнктов  $C_1$  и  $C_3$  и добавим её к множеству дизъюнктов, для чего произведём унификацию переменных  $x$  и  $u$ , таким образом, что  $u$  заменим на  $x$ , и получим  $C_5 = R(x, f(x)) \vee -Q(x, g(x))$ .*

*Найдём резольвенту  $C_6$  дизъюнктов  $C_2$  и  $C_5$  и добавим её к множеству дизъюнктов, для чего произведём унификацию переменных  $y$  и  $x$ , таким образом, что  $y$  заменим на  $x$ , и получим  $C_6 = R(x, f(x)) \vee R(x, f(x)) = R(x, f(x))$ .*

*Найдём резольвенту  $C_7$  дизъюнктов  $C_2$  и  $C_6$  и добавим её к множеству дизъюнктов, для чего произведём замену переменной  $x$  на константу  $a$ , а переменную  $z$  заменим на функцию  $f(a)$ , таким образом получим  $C_7 = \square$ , то есть пустой дизъюнкт.*

*Алгоритм метода резолюций.*

*Шаг 1. Если в  $S$  есть пустой дизъюнкт, то множество невыполнимо, иначе перейти к шагу 2.*

*Шаг 2. Найти в исходном множестве  $S$  такие дизъюнкты или склейки дизъюнктов  $C_1$  и  $C_2$ , которые содержат унифицируемые литералы  $L_1 \in C_1$  и  $L_2 \in C_2$ . Если таких дизъюнктов нет, то исходное множество выполнимо, иначе перейти к шагу 3.*

*Шаг 3. Вычислить резольвенту  $C_1$  и  $C_2$  и добавить её в множество  $S$ . Перейти к шагу 1.*

## **2 Введение в язык логического программирования ПРОЛОГ.**

### **2.1 Общие положения**

Язык Пролог объединяет два подхода: логический и процедурный.

По мнению Дж. Робинсона, в основе идеи логического программирования лежит принцип описания задачи при помощи совокупности утверждений на некотором формальном логическом языке и получение решения при помощи вывода в некоторой формальной системе. Основой языка Пролог является логика предикатов первого порядка.



Программа на Прологе включает в себя постановку задачи в виде множества фраз Хорна (раздел clauses) и описание цели (раздел goal), - формулировку теоремы, которую нужно доказать, исходя из множества правил и фактов, содержащихся в этой постановке.

Процесс поиска доказательства основан на методе линейной резолюции (дизъюнкты подбираются в порядке их следования в тексте программы).

Проиллюстрируем принцип логического программирования на простом примере: запишем известный метод вычисления наибольшего общего делителя двух натуральных чисел – алгоритм Евклида в виде Хорновских дизъюнктов. При этом примем новую форму записи фразы Хорна, например  $S \vee \neg P \vee \neg Q \vee \neg R$  будем записывать как  $S: - P, Q, R$ . Тогда алгоритм Евклида можно записать в виде трех фраз Хорна:

1.  $NOD(x, x, x): -$ .
2.  $NOD(x, y, z): - B(x, y), NOD(f(x, y), y, z)$ .
3.  $NOD(x, y, z): -B(y, x), NOD(x, f(y, x), z)$ .

Предикат  $NOD$  – определяет наибольший общий делитель  $z$  для натуральных чисел  $x$  и  $y$ , предикат  $B$  – определяет отношение «больше», функция  $f$  – определяет операцию вычитания. Если мы заменим предикат  $B$  и функцию  $f$  обычными символами, то фразы примут вид:

1.  $NOD(x, x, x): -$ .
2.  $NOD(x, y, z): - x > y, NOD((x - y), y, z)$ .
3.  $NOD(x, y, z): -y > x, NOD((x, y - x), z)$ .

Для вычисления наибольшего общего делителя двух натуральных чисел, например 4 и 6, добавим к описанию алгоритма четвертый дизъюнкт:

4.  $\square: - NOD(4, 6, z)$ .

Последний дизъюнкт – это цель, которую мы будем пытаться вывести из первых трех дизъюнктов.

## 2.2 Основы языка программирования Пролог

Язык программирования Пролог (PROgramming LOGic) предполагает получение решения задачи при помощи логического вывода из ранее известных фактов. Программа на языке Пролог не является последовательностью действий – она представляет собой набор фактов и правил, обеспечивающих получение логических заключений из данных фактов. Поэтому Пролог считается декларативным языком программирования.

Пролог базируется на *фразах (предложениях) Хорна*, являющихся подмножеством формальной системы, называемой *логикой предикатов* [2].

Пролог использует упрощенную версию синтаксиса логики предикатов, он прост для понимания и очень близок к естественному языку.

Пролог имеет механизм вывода, который основан на сопоставлении образцов. С помощью подбора ответов на запросы Пролог извлекает

хранящуюся информацию. Пролог пытается ответить на запрос, запрашивая информацию, о которой уже известно, что она истинна.

Одной из важнейших особенностей Пролога является то, что он ищет не только ответ на поставленный вопрос, но и все возможные альтернативные решения. Вместо обычной работы программы на процедурном языке от начала и до конца, Пролог может возвращаться назад и просматривать все остальные пути при решении всех частей задачи.

Программист на Прологе описывает объекты и отношения, а также правила, при которых эти отношения являются истинными.

Объекты рассуждения в Прологе называются *термами* – синтаксическими объектами одной из следующих категорий:

- константы,
- переменные,
- функции (составные термы или структуры), состоящие из имени функции и списка аргументов-термов, имена функций начинаются со строчной буквы.

*Константа* в Прологе служит для обозначения имен собственных и начинается *со строчной буквы*.

*Переменная* в Прологе служит для обозначения объекта на который нельзя сослаться *по имени*.

Пролог не имеет оператора присваивания.

***Переменные в Прологе инициализируются при сопоставлении с константами в фактах и правилах.***

До инициализации переменная свободна, после присвоения ей значения она становится связанной. Переменная остается связанной только то время, которое необходимо для получения решения по запросу, затем Пролог освобождает ее и ищет другое решение.

Переменные в Прологе предназначены для установления соответствия между термами предикатов, действующих в пределах одной фразы (предложения), а не местом памяти для хранения данных. Переменная начинается с прописной буквы или знаков подчеркивания.

В Прологе программист свободен в выборе имен констант, переменных, функций и предикатов. Исключения составляют резервированные имена и числовые константы. Переменные от констант отличаются первой буквой имени: у констант она строчная, у переменных – заглавная буква или символ подчеркивания.

Область действия имени представляет собой часть программы, где это имя имеет один и тот же смысл:

- для переменной областью действия является предложение (факт, правило или цель), содержащее данную переменную;
- для остальных имен (констант, функций или предикатов) – вся программа.

Специальным знаком «\_» обозначается анонимная переменная, которая используется тогда, когда конкретное значение переменной не существенно

для данного предложения. Анонимные переменные не отличаются от обычных при поиске соответствий, но не принимают значений и не появляются в ответах. *Различные вхождения знака подчеркивания означают различные анонимные переменные.*

Отношения между объектами в Прологе называются фактами. Факт соответствует фразе Хорна, состоящей из одного положительного литерала.

Факт – это простейшая разновидность предложения Пролога.

Любой факт имеет соответствующее значение истинности и определяет отношение между термами.

Факт является простым предикатом, который записывается в виде функционального термина, состоящего из имени отношения и объектов, заключенных в круглые скобки, например:

мать( мария, анна).

отец( иван, анна).

Точка, стоящая после предиката, указывает на то, что рассматриваемое выражение является фактом.

Вторым типом предложений Пролога является вопрос или *цель*. *Цель* – это средство формулировки задачи, которую должна решать программа. Простой вопрос (цель) синтаксически является разновидностью факта, например:

Цель: мать (мария, юлия).

В данном случае программе задан вопрос, является ли мария матерью юлии. Если необходимо задать вопрос, кто является матерью юлии, то цель будет иметь следующий вид:

Цель: мать( X, юлия).

Сложные цели представляют собой конъюнкцию простых целей и имеют следующий вид:

*Цель:  $Q_1, Q_2, \dots, Q_n$ , где запятая обозначает операцию конъюнкции, а  $Q_1, Q_2, \dots, Q_n$  – подцели главной цели.*

Конъюнкция в Прологе истинна только при истинности всех компонент, однако, в отличие от логики, в Прологе учитывается *порядок оценки истинности компонент* (слева направо).

*Пример 15.*

*Пусть задана семейная БД при помощи перечисления родительских отношений в виде списка фактов:*

*мать( мария, анна).*

*мать(мария, юлия).*

*мать( анна, петр).*

*отец( иван, анна).*

*отец( иван, юлия).*

*Тогда вопрос, является ли иван дедом петра, можно задать в виде следующей цели:*

*Цель: отец( иван, X), мать(X, петр).*

На самом деле БД Пролога включает не только факты, но и правила. Факты и правила представляют собой не множество, а список. Для получения ответа БД просматривается по порядку, то есть в порядке следования фактов и предикатов в тексте программы.

Цель достигнута, если в БД удалось найти факт или правило, который (которое) удовлетворяет предикату цели, то есть превращает его в истинное высказывание. В нашем примере первую подцель удовлетворяют факты *отец( иван, анна)*. и *отец( иван, юлия)*. Вторую подцель удовлетворяет факт *мать( анна, петр)*. Следовательно, главная цель удовлетворена, переменная *X* связывается с константой *анна*.

Третьим типом предложения является *правило*. Правило позволяет вывести один факт из других фактов. Иными словами, правило – это заключение, для которого известно, что оно истинно, если одно или несколько других найденных заключений или фактов являются истинными.

*Правила – это предложения вида*

*H: - P<sub>1</sub>, P<sub>2</sub>, ..., P<sub>n</sub>.*

Символ «: -» читается как «если», предикат *H* называется заключением, а последовательность предикатов *P<sub>1</sub>, P<sub>2</sub>, ..., P<sub>n</sub>* называется посылками. Приведенное правило является аналогом хорновского дизъюнкта  $H \vee \neg P_1 \vee \neg P_2, \dots, \vee \neg P_n$ . Заключение истинно, если истинны все посылки. В посылках переменные связаны квантором существования, а в заключении - квантором всеобщности.

*Пример 16.*

*Добавим в БД примера 15 правила, задающие отношение «дед»:*

*мать( мария, анна).*

*мать(мария, юлия).*

*мать( анна, петр).*

*отец( иван, анна).*

*отец( иван, юлия).*

*дед (X, Y): - отец(X, Z), мать(Z, Y).*

*дед (X, Y): - отец(X, Z), отец(Z, Y).*

*Тогда вопрос, является ли иван дедом петра, можно задать в виде следующей цели:*

*Цель: дед( иван, петр).*

Правила - самые общие предложения Пролога, факт является частным случаем правила без правой части, а цель – правило без левой части.

Все предложения для одного предиката связаны между собой отношением «или».

Очень часто правила в Прологе являются рекурсивными. Например, для нашей семейной БД предикат «предок» определяется рекурсивно:

*предок(x, y): - мать(x, y).*

*предок(x, y): - отец(x, y).*

*предок(x, y): - мать(x, z), предок(z, y).*

*предок(x, y): - отец (x, z), предок(z, y).*

Рекурсивное определение предиката обязательно должно содержать нерекурсивную часть, иначе оно будет логически некорректным и программа заикнется. Чтобы избежать заикливания, следует также позаботиться о порядке выполнения предложений, поэтому практически полезно, а порой и необходимо придерживаться принципа: «*сначала нерекурсивные выражения*».

Программа на Прологе - это конечное множество предложений.

### **2.3 Использование дизъюнкции и отрицания.**

Чистый Пролог разрешает применять в правилах и целях только конъюнкцию, однако, язык, используемый на практике, допускает применение дизъюнкции и отрицания в телах правил и целях. Для достижения цели, содержащей дизъюнкцию, Пролог-система сначала пытается удовлетворить левую часть дизъюнкции, а если это не удастся, то переходит к поиску решения для правой части дизъюнкции. Аналогичные действия производятся при выполнении тела правил, содержащих дизъюнкцию. Для обозначения дизъюнкции используется символ « ; ».

В Прологе отрицание имеет имя «*not*» и для представления отрицания какого-либо предиката  $P$  используется запись  $not(P)$ . *Цель  $not(P)$  достижима тогда и только тогда, когда не удовлетворяется предикат (цель)  $P$* . При этом переменным значения не присваиваются. В самом деле, если достигается  $P$ , то не достигается  $not(P)$ , значит надо стереть все присваивания, приводящие к данному результату. Наоборот, если  $P$  не достигается, то переменные не принимают никаких значений.

### **2.4 Унификация в Прологе.**

Общий принцип выполнения программ на Прологе прост: производится поиск ответа на вопросы, задаваемые БД, состоящей из фактов и правил, то есть проверяется соответствие предикатов вопроса предложениям из БД. Это частный случай метода резолюций.

Отметим, что в Прологе не проводится синтаксического различия между предикатом и функцией (составным термом), а также между нечисловой константой и функцией без аргументов. Следовательно, суть действия состоит в том, что ищется попарное соответствие между термами, один из которых является целью, а другой принадлежит БД.

Установление соответствия между термами является основной операцией при вычислении цели. Она осуществляется следующим образом: на каждом шаге выбирается очередной терм и отыскивается соответствующее выражение в БД. При этом переменные получают или теряют значения. Этот процесс можно описать в терминах текстуальных подстановок: «подставить терм  $t$  вместо переменной  $Y$ ». Свободными переменными в Прологе называются переменные, которым не были присвоены значения, а все остальные переменные называются связанными

переменными. Переменная становится связанной только во время унификации, переменная вновь становится свободной, когда унификация оказывается неуспешной или цель оказывается успешно вычисленной. В Прологе присваивание значений переменным выполняется внутренними подпрограммами унификации. Переменные становятся свободными, как только для внутренних подпрограмм унификации отпадает необходимость связывать некоторое значение с переменной для выполнения доказательства подцели.

### *Правила унификации.*

1. Если  $x$  и  $y$ -константы, то они унифицируемы, только если они равны.

2. Если  $x$ - константа или функция, а  $Y$ -переменная, то они унифицируемы, при этом  $Y$  принимает значение  $x$ .

3. Если  $x$  и  $y$  -функции, то они унифицируемы тогда и только тогда, когда у них одинаковые имена функций (функторы) и набор аргументов и каждая пара аргументов функций унифицируемы.

Есть особый предикат «= $\Rightarrow$ », который используется в Прологе для отождествления двух термов. Использование оператора «= $\Rightarrow$ » поможет лучше понять процесс означивания переменной. В Прологе оператор «= $\Rightarrow$ » интерпретируется как оператор присваивания или как оператор проверки на равенство в зависимости от того, являются ли значения термов свободными или связанными.

*Пример 17:  $X=Y$ , если  $X$  и  $Y$  – связанные переменные, то производится проверка на равенство, например: если  $X=5$  и  $Y=5$ , то результат ДА (истина); если  $X=6$  а  $Y=5$ , то результат НЕТ(ложь). Если одна из переменных  $X$  или  $Y$  – свободная, то ей будет присвоено значение другой переменной, для Пролога несущественно слева или справа от знака «= $\Rightarrow$ » стоит связанная переменная.*

Оператор «= $\Rightarrow$ » ведет себя точно так, как внутренние подпрограммы унификации при сопоставлении целей или подцелей с фактами и правилами программы.

### **2.5 Вычисление цели. Механизм возврата.**

Каноническая форма цели (вопроса) является конъюнкцией атомарных предикатов, то есть последовательностью подцелей, разделенных запятыми [1]:

$$Q=Q_1, Q_2, \dots, Q_n.$$

Пролог пытается вычислить цель при помощи унификации термов предикатов подцелей с соответствующими элементами в фактах и заголовках правил. Поиск ответа на вопрос напоминает поиск пути в лабиринте: следует поворачивать налево в каждой развилке лабиринта до тех пор, пока не попадете в тупик. В этом случае следует вернуться к последней развилке и

повернуть направо, после чего опять следует повернуть налево и так далее. Унификация выполняется слева направо, как и поиск пути в лабиринте.

Некоторые подцели при унификации с некоторыми фактами или правилами могут оказаться неуспешными, поэтому Прологу требуется способ запоминания точек отката, в которых он может продолжить альтернативные поиски решения. Прежде чем реализовать один из возможных путей вычисления подцели, Пролог фактически помещает в программу указатель, который определяет точку, в которую может быть выполнен возврат, если текущая попытка поиска цели будет неудачной.

Если некоторая подцель оказывается неуспешной, то Пролог возвращается влево к ближайшей точке возврата. С этой точки Пролог начинает попытку найти другое решение для неуспешной цели. До тех пор, пока следующая цель на данном уровне не будет успешной, Пролог будет повторять возврат к ближайшей точке возврата. Эти попытки выполняются внутренними подпрограммами унификации и механизмом возврата.

**Замечание:** *единственным способом освободить переменную, связанную в предложении является откат при поиске с возвратом.*

Алгоритм вычисления цели – частный случай правила резолюции применительно к дизъюнктам Хорна. Вопрос  $Q$  является правилом без заголовка, аналогом выражения  $\neg Q$ . Пусть  $D$  – база данных (множество дизъюнктов). На вопрос  $Q$ , следует найти такую подстановку  $\sigma$ , для которой множество  $\sigma[D \cup (\neg Q)]$  невыполнимо. Стратегия выбора очередной пары дизъюнктов для резолюции здесь очень проста: подцели и предложения просматриваются в текстуальном порядке.

*Пример 18: пусть есть БД семья:*

1. *мать(мария, анна).*
2. *мать(мария, юлия).*
3. *мать(анна, петр).*
4. *отец(иван, анна).*
5. *отец(иван, юлия).*
6. *дед(X, Y): - отец(X, Z), мать(Z, Y).*
7. *дед(X, Y): - отец(X, Z), отец(Z, Y).*
8. *бабка(X, Y): - мать(X, Z), мать(Z, Y).*
9. *бабка(X, Y): - мать(X, Z), отец(Z, Y).*

*Зададим сложную цель:*

$Q1, Q2 = \text{отец}(X, Y), \text{мать}(\text{мария}, Y)$ .

*Подцель  $Q1 = \text{отец}(X, Y)$  соответствует четвертому предложению БД: отец (иван, анна). Это дает подстановку  $\sigma_1 = \{X = \text{иван}, Y = \text{анна}\}$ . Затем найденная подстановка применяется к  $\sigma_1[Q2] = \text{мать}(\text{мария}, \text{анна})$ . Этой подцели соответствует 1 предложение БД, что дает пустую подцель по правилу резолюции, следовательно ответ найден:  $X = \text{иван}, Y = \text{анна}$ .*

*Для получения нового ответа в БД ищется новая унификация для  $\sigma_1[Q2]$ . Так как в БД нет соответствующего предложения, то вычисления*

прекращаются, система вновь рассматривает последовательность  $Q_1$ ,  $Q_2$  и для  $Q_1$  ищется новая унификация в БД, начиная с пятого предложения. Это и есть возврат. Пятое предложение сразу же дает желаемую унификацию с подстановкой  $\sigma_2 = \{X=иван, Y=юлия\}$ . Вновь найденная подстановка применяется к  $\sigma_1[Q_2] = \text{мать(мария, юлия)}$ . Этой подцели соответствует второе предложение БД. Далее указанная процедура повторяется и в итоге имеем:

Цель: отец( $X$ ,  $Y$ ), мать(мария,  $Y$ ).

2 решения:  $X=иван$ ,  $Y=анна$

$X=иван$ ,  $Y=юлия$ .

**Это описание объясняет, как работает утилита *TestGoal* в *Visual Prolog*.**

При реализации механизма возврата выполняются следующие правила:

1. Подцели вычисляются слева-направо.
2. Предложения при вычислении подцели проверяются в текстуальном порядке, то есть сверху-вниз.
3. Если подцель сопоставима с заголовком правила, то должно быть вычислено тело этого правила, при этом тело правила образует новое подмножество подцелей для вычисления.
4. Цель считается успешно вычисленной, когда найден соответствующий факт для каждой подцели.

**Если в *Visual Prolog* создать программу для автономного исполнения (то есть создать свой *project*-файл для разрабатываемой программы), то поиск цели в ней будет вестись так же, как для внутренней цели в *PDC Prolog*, то есть до первого успешного решения.**

## 2.6 Управление поиском решения.

Встроенный в Пролог механизм поиска с возвратом может привести к поиску ненужных решений, в результате чего снижается эффективность программы в случае, если надо найти только одно решение. В других случаях бывает необходимо продолжить поиск, даже если решение найдено.

Пролог обеспечивает два встроенных предиката, которые дают возможность управлять механизмом поиска с возвратом: предикат *fail* – используется для инициализации поиска с возвратом и предикат *отсечения* ! – используется для запрета возврата.

***Предикат fail* всегда имеет ложное значение!**

Отсечение так же, как и *fail* помещается в тело правила. **Однако, в отличие от *fail* предикат отсечения имеет всегда истинное значение.**

При этом выполняется обращение к другим предикатам в теле правила, следующим за отсечением. Следует иметь в виду, что невозможно произвести возврат к предикатам, расположенным в теле правила перед отсечением, а также невозможен возврат к другим правилам данного предиката.



Существует только два случая применения предиката отсечения:

1. Если заранее известно, что определенные посылки никогда не приведут к осмысленным решениям – это так называемое «зеленое отсечение».
2. Если отсечения требует сама логика программы для исключения альтернативных подцелей – это так называемое «красное отсечение».

*Пример 19: Использование предикатов ! и fail. Для примера 16 можно добавить правила для печати всех пар «мать-ребёнок», которые есть в БД:*

*1.печать\_1(X,Y):-мать(X,Y), write(X," есть мать",Y),nl.*

*goal*

*печать\_1(X,Y).*

*В результате будет выдано 3 решения:*

*X=мария, Y= анна.*

*X=мария, Y= юлия.*

*X=анна, Y= петр.*

*2.печать\_2:-мать(X,Y), write(X," есть мать",Y),nl,fail.*

*goal*

*печать\_2.*

*В результате будет выдано 3 решения:*

*X=мария, Y= анна.*

*X=мария, Y= юлия.*

*X=анна, Y= петр.*

*3.печать\_3(X,Y):-мать(X,Y), write(X," есть мать",Y),nl,!.*

*goal*

*печать\_3(X,Y).*

*В результате будет выдано 1 решение:*

*X=мария, Y= анна.*

## **2.7 Процедурность Пролога.**

Пролог – декларативный язык. Описывая задачу в терминах фактов и правил, программист предоставляет Прологу самому искать способ решения. В процедурных языках программист должен сам писать процедуры и функции, которые подробно «объясняют» компьютеру, какие шаги надо сделать для решения задачи.

Тем не менее, рассмотрим Пролог с точки зрения процедурного программирования:

1. Факты и правила можно рассматривать как определения процедур.
2. Использование правил для условного ветвления программы. Правило, в отличие от процедуры, позволяет задавать множество альтернативных определений одной и той же процедуры.

Поэтому, правило можно считать аналогом оператора *case* в Паскале.

3. В правиле может быть выполнено сравнение, как в условных операторах.
4. Отсечение можно считать аналогом *go to*.
5. Возврат вычисленного значения производится аналогично процедурам. В Прологе это делается путем связывания свободных переменных при сопоставлении цели с фактами и правилами.

## 2.8 Структура программ Пролога.

Программа, написанная на Прологе, состоит из шести основных разделов: раздел описания доменов, раздел базы данных, раздел описания предикатов, раздел описания предложений и раздел описания цели. Ключевые слова *domains*, *constants*, *facts (database)*, *predicates*, *clauses* и *goal* отмечают начала соответствующих разделов. Назначение этих разделов таково:

- раздел *domains* содержит объявления доменов, которые описывают различные типы данных, используемых в программе, если в программе не требуется объявления доменов, то этот раздел может быть опущен;
- раздел *constants* используется для объявления символических констант, используемых в программе, если в программе не требуется объявления символических констант, то этот раздел может быть опущен;
- раздел *facts (database)* содержит описания предикатов внутренней базы данных Пролога, если программа такой базы данных не требует, то этот раздел может быть опущен;
- раздел *predicates* служит для описания предикатов, не принадлежащих внутренней базе данных, если в программе не требуется объявления предикатов, то этот раздел может быть опущен;
- в раздел *clauses* заносятся факты и правила самой программы, если в программе используются только встроенные предикаты, то этот раздел может быть опущен;
- в разделе *goal* на языке Пролог формулируется назначение создаваемой программы. **Данный раздел программы является обязательным.** Составными частями при этом могут являться некие подцели, из которых формируется единая цель программы.

В Visual Prolog разрешает объявление разделов *domains*, *facts*, *predicates*, *clauses* как глобальных разделов, то есть с ключевым словом *global*.

Пролог имеет следующие встроенные типы доменов:

Тип данных	Ключевое слово	Диапазон значений	Примеры использования
Символы	char	Все возможные символы	'a', 'b', '#', 'B', '%'
Целые числа	integer	От -32768 до 32767	-63, 84, 2349
	byte	От 0 до 255	
	word	От 0 до 65535	
	dword	От 0 до	
Действительные числа	real	От +1E-307 до +1E308	360, - 8324, 1.25E23, 5.15E-9
	short	16 битов со знаком	
	ushort	16 битов без знака	
	long	32 бита со знаком	
	ulong unsigned	32 бита без знака 16 или 32 бита без знака	
Строки	string	Последовательность символов (не более 250)	«today», «123», «school_day»
Символические имена	symbol	1. Последовательность букв, цифр, символов подчеркивания; первый символ – строчная буква. 2. Последовательность любых символов, заключенная в кавычки.	flower, school_day  «string and symbol»
Ссылочный тип	ref		
Файлы	file	Допустимое в DOS имя файла	mail.txt, LAB.PRO

Если в программе необходимо использовать новые домены данных, то они должны быть описаны в разделе *domains*.

*Пример 20:*

```
domains
number=integer
name, person=symbol.
```

Различие между `symbol` и `string` - в машинном представлении и выполнении, синтаксически они не различимы.

Visual Prolog выполняет автоматическое преобразование типов между доменами `string` и `symbol`. Однако, по принятому соглашению, символическую строку в двойных кавычках нужно рассматривать как `string`, а без кавычек – как `symbol`:

`Symbol` - имена, начинающиеся с символа нижнего регистра и содержащие только символы, цифры, и символы подчеркивания.

`String` – в двойных кавычках могут содержать любую комбинацию символов, кроме `#0`, который отмечает конец строки.

Visual Prolog поддерживает и другие типы стандартных доменов данных, например, для работы с внешними БД или объектами.

Предикаты описываются в разделе `predicates`. Предикат представляет собой строку символов, первым из которых является строчная буква. Предикаты могут не иметь аргументов, например «`go`» или «`repeat`». Если предикаты имеют аргументы, то они определяются при описании предикатов в разделе `predicates`:

*Пример 21:*

*predicates*

*mother (symbol, symbol)*

*father (symbol, symbol).*

Факты и правила определяются в разделе `clauses`, а вопрос к программе задается в разделе `goal` – в этом случае цель называется внутренней целью. В Visual Prolog раздел `goal` в тексте программы является обязательным. Разница в режимах исполнения программы состоит в разном использовании утилиты `Test Goal`. Если утилита создается для запуска любой программы, то при этом ищутся все решения, если утилита создается для автономного запуска программы – то ищется одно решение.

## 2.9 Использование составных термов

В Прологе функциональный терм или предикат можно рассматривать как структуру данных, подобную записи в языке Паскаль. Терм, представляющий совокупность термов, называется составным термом или структурой. Составные структуры данных в Прологе объявляются в разделе `domains`. Если термы структуры относятся к одному и тому же типу доменов, то этот объект называется однодоменной структурой данных. Если термы структуры относятся к разным типам доменов, то такая структура данных называется многодоменной структурой данных. Использование доменной структуры упрощает структуру предиката.

Аргументами составного терма данных могут быть простые типы данных, составные термы или списки.

Синтаксически составной терм выглядит так же, как и предикат: у терма есть функтор и список аргументов, заключенных в круглые скобки.

Составной терм может быть унифицирован с простой переменной или составным объектом (при этом переменные могут быть использованы как часть внутренней структуры терма). Это означает, что составной объект

можно использовать для того, чтобы передавать целый набор значений, как единый объект, а затем применять унификацию для их разделения.

*Пример 22: Необходимо создать БД, содержащую сведения о книгах из личной библиотеки. Зададим составной терм с именем `personal_library`, имеющим следующую структуру: `personal_library = book (title, author, publisher, year)`, и предикат `collection (collector, personal_library)`. Терм `book` называется функтором структуры данных. Пример программы, использующей составные термы для описания личной библиотеки и поиска информации о книгах, напечатанных в 1990 году, выглядит следующим образом:*

```
domains
collector, title, author, publisher = symbol
year = integer
personal_library = book (title, author, publisher, year)
predicates
collection (collector, personal_library)
clauses
collection (irina, book («Using Turbo Prolog», «Yin with Solomon»,
«Moscow, World», 1993)).
collection (petr, book («The art of Prolog», «Sterling with Shapiro»,
»Moscow, World», 1990)).
collection (anna, book («Prolog: a relation language and its applications»,
«John Malpas», »Moscow, Science», 1990)).
goal
collection (X, book( Y,_, _, 1990))
```

В данном случае переменная  $Y$  используется для унификации части составного терма. Если цель задать в виде:

`collection (X, Z), Z=book( Y,_, _, 1990)`, то простая переменная  $Z$  унифицируется с составным термом `book`.

Представление данных часто требует наличия большого числа структур. В Прологе эти структуры должны быть описаны. Для более компактного описания структур данных в Прологе предлагается использование альтернативных описаний доменов.

*Пример 23: Необходимо создать БД, содержащую сведения о книгах и аудиозаписях из личной библиотеки.*

```
domains
person, title, author, artist, album, type = symbol
thing = book (title, author); record (artist, album, type)
predicates
owns (person, thing)
clauses
owns (irina, book («Using Turbo Prolog», «Yin with Solomon»)).
owns (petr, book («The art of Prolog», «Sterling with Shapiro»)).
```

*owns (anna, book («Prolog: a relation language and its applications», «John Malpas»)).*

*owns (irina, record («Elton John», «Ice Fair», «popular»)).*

*owns (petr, record («Benny Goodman», «The King of Swing», »jazz»)).*

*owns (anna record («Madonna», «Madonna», «popular»)).*

*goal*

*owns (X, record( \_, \_ , «jazz»))*

Visual Prolog позволяет конструировать многоуровневые составные термы. Например, в терме *record («Elton John», «Ice Fair», «popular»)* вместо фамилии артиста можно использовать новую структуру, которая будет описывать артиста более детально:

*artist=art(family,name),*

*family,name=symbol,*

*при этом терм будет выглядеть следующим образом: record (art(“Elton”, ”John ”),«Ice Fair», «popular»).*

## **2.10 Использование списков**

Список является составной рекурсивной структурой данных, хотя явно и не объявляется как рекурсивная структура. Список – это упорядоченный набор объектов одного и того же типа. Элементами списка могут быть целые числа, действительные числа, символы, строки, символические имена и структуры. Порядок расположения элементов в списке играет важную роль: те же самые элементы списка, упорядоченные иным способом, представляют уже совсем другой список [4].

Совокупность элементов списка заключается в квадратные скобки ([ ]), элементы друг от друга отделяются запятыми. Список может содержать произвольное число элементов, единственным ограничением является объем оперативной памяти. Количество элементов в списке называется его длиной. Список может содержать один элемент и даже не содержать ни одного элемента. Список, не содержащий элементов, называется пустым или нулевым списком.

Непустой список можно рассматривать как список, состоящий из двух частей: головы – первого элемента списка; и хвоста – остальной части списка. Голова является элементом списка, хвост является списком. Голова списка – это неделимое значение, хвост представляет собой список, составленный из того, что осталось от исходного списка в результате «отделения головы». Этот новый список обычно можно делить и дальше. Если список состоит из одного элемента, то его можно разделить на голову, которой будет этот самый элемент, и хвост, являющийся пустым списком.

***Пустой список нельзя разделить на голову и хвост!***

Операция деления списка на голову и хвост обозначается при помощи вертикальной черты (|):

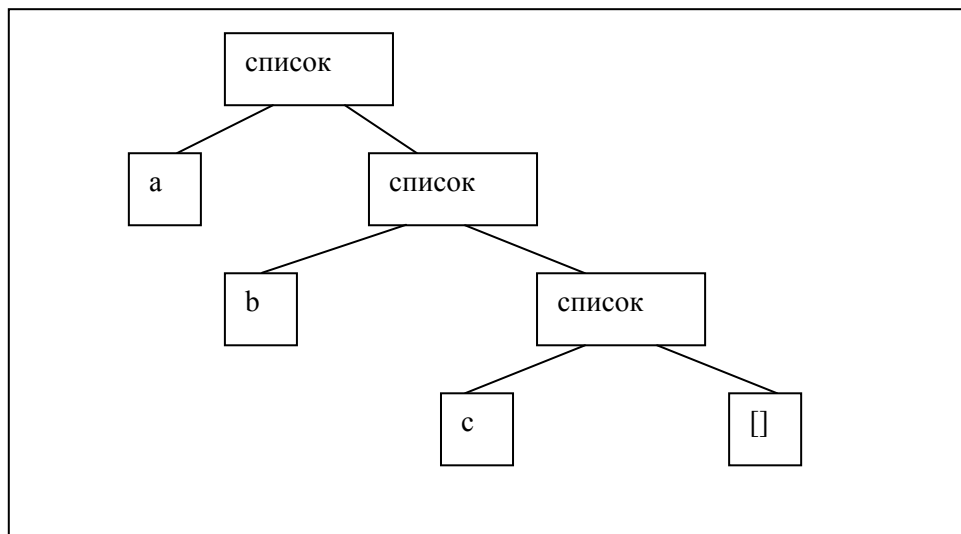
[Head | Tail].

**Голова списка всегда имеет тип элемента списка, хвост списка – тип списка!**

Head здесь является переменной для обозначения головы списка, переменная Tail обозначает хвост списка (для имен головы и хвоста списка пригодны любые допустимые Прологом имена).

Данная операция также присоединяет элемент в начало списка, например, для того, чтобы присоединить X к списку S следует написать [X|S].

В концептуальном плане, список имеет структуру дерева, как и другие составные термы. Так, например, список [a,b,c] можно представить в виде структуры:



Отличительной особенностью описания списков является наличие звездочки (\*) после имени домена элементов.

*Пример 24: объявление списков, состоящих из элементов стандартных типов доменов или типа структуры.*

```
domains
list1=integer*
list2=char*
list3=string*
list4=real*
list5=symbol*
personal_library = book (title, author, publisher, year)
list6= personal_library*
list7=list1*
list8=list5*
```

В первых пяти объявлениях списков в качестве элементов используются стандартные домены данных, в шестом типе списка в качестве элемента используется домен структуры personal\_library, в седьмом и восьмом типе списка в качестве элемента используется ранее объявленный список.

Пример 25: демонстрация разделения списков на голову и хвост.

Список	Голова	Хвост
[1, 2, 3, 4, 5]	1	[2, 3, 4, 5]
[6.9, 4.3]	6.9	[4.3]
[cat, dog, horse]	Cat	[dog, horse]
['S', 'K', 'Y']	'S'	['K', 'Y']
[«PIG»]	«PIG»	[]
[]	Не определена	Не определен

Visual Prolog допускает объявление и использование составных списков. Составной список – это список, в котором используется более чем один тип элемента. Для создания такого списка, необходимо использовать структуры, так как домен может содержать более одного типа данных только для структуры.

Пример 26: объявление списка, который может содержать символы, целые числа или строки:

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

## 2.11 Применение списков в программах

Для применения списков в программах на Прологе необходимо описать домен списка в разделе *domains*, предикаты, работающие со списками необходимо описать в разделе *predicates*, задать сам список можно либо в разделе *clauses* либо в разделе *goal*.

Над списками можно реализовать различные операции: поиск элемента в списке, разделение списка на два списка, присоединение одного списка к другому, удаление элементов из списка, сортировку списка, создание списка из содержимого БД и так далее.

### 2.11.1 Поиск элемента в списке

Поиск элемента в списке является очень распространенной операцией. Поиск представляет собой просмотр списка на предмет выявления соответствия между объектом поиска и элементом списка. Если такое соответствие найдено, то поиск заканчивается успехом, в противном случае поиск заканчивается неуспехом. Стратегия поиска при этом будет состоять в рекурсивном выделении головы списка и сравнении ее с объектом поиска.

Пример 27: поиск элемента в списке.

```
domains
list=integer*
predicates
```



*member (integer, list)*

*clauses*

*member (Head, [Head | \_]).*

*member (Head, [\_ | Tail ]):- member (Head, Tail).*

*goal*

*member (3, [1, 4, 3, 2]).*

Правило поиска может сравнить объект поиска и голову текущего списка, эта операция записана в виде факта предиката *member*. Этот вариант предполагает наличие соответствия между объектом поиска и головой списка. Отметим, что хвост списка в данном случае не важен, поэтому хвост списка присваивается анонимной переменной. Если объект поиска и голова списка различны, то в результате исполнения первого предложения будет неуспех, происходит возврат и поиск другого правила или факта, с которыми можно попытаться найти соответствие. Для этой цели служит второе предложение, которое выделяет из списка следующий по порядку элемент, то есть выделяет голову текущего хвоста, поэтому текущий хвост представляется как новый список, голову которого можно сравнить с объектом поиска. В случае исполнения второго предложения, голова текущего списка ставится в соответствие анонимной переменной, так как значение головы с писка в данном случае не играет никакой роли.

Процесс повторяется до тех пор, пока первое предложение даст успех, в случае установления соответствия, либо неуспех, в случае исчерпания списка. В представленном примере предикат *find* находит все совпадения объекта поиска с элементами списка. Для того, чтобы найти только первое совпадение следует модифицировать первое предложение следующим образом:

*member (Head, [Head | \_ ]):- !.*

Отсечение отменяет действие механизма возврата, поэтому поиск альтернативных успешных решений реализован не будет.

### 2.11.2 Объединение двух списков

Слияние двух списков и получение, таким образом, третьего списка принадлежит к числу наиболее полезных при работе со списками операций. Обозначим первый список  $L1$ , а второй список -  $L2$ . Пусть  $L1 = [1, 2, 3]$ , а  $L2 = [4, 5]$ . Предикат *append* присоединяет  $L2$  к  $L1$  и создает выходной список  $L3$ , в который он должен переслать все элементы  $L1$  и  $L2$ . Весь процесс можно представить следующим образом:

1. Список  $L3$  вначале пуст.
2. Элементы списка  $L1$  пересылаются в  $L3$ , теперь значением  $L3$  будет  $[1, 2, 3]$ .
3. Элементы списка  $L2$  пересылаются в  $L3$ , в результате чего тот принимает значение  $[1, 2, 3, 4, 5]$ .

Тогда программа на языке Пролог имеет следующий вид:

*Пример 28: объединение двух списков.*

```

domains
list=integer*
predicates
append(list, list, list)
clauses
append([], L2, L2).
append([H|T1], L2, [H|T3]):- append(T1, L2, T3).
goal
append([1, 2, 3], [4, 5], L3).

```

Основное использование предиката *append* состоит в объединении двух списков, что делается при помощи задания цели вида *append* (*[1, 2, 3], [4, 5], L3*). Поиск ответа на вопрос типа: *append* (*L1, [3, 4, 5], [1, 2, 3, 4, 5]*) – сводится к поиску такого списка *L1=[1, 2]*, который при слиянии со списком *L2 = [3, 4, 5]* даст список *L3 = [1, 2, 3, 4, 5]*. При обработки цели *append* (*L1, L2, [1, 2, 3, 4, 5]*) ищутся такие списки *L1* и *L2*, что их объединение даст список *L3 = [1, 2, 3, 4, 5]*.

### 2.11.3 Определение длины списка

Число элементов в списке можно подсчитать при помощи рекурсивных предикатов *count\_list1* и *count\_list2*. Предикат *count\_list1* ведёт подсчёт числа элементов в списке на прямом ходе рекурсии, начиная от головы списка, при этом первый параметр типа *byte* является текущим счётчиком, а второй - необходим для возвращения результата при выходе из рекурсии. Предикат *count\_list2* ведёт подсчёт числа элементов в списке на обратном ходе рекурсии, начиная с последнего элемента, при этом параметр типа *byte* является текущим счётчиком и результатом одновременно. Два варианта решения задачи приводятся в примере 29.

*Пример 29: определение длины списка.*

```

domains
list=integer*
predicates
count_list1(list,byte,byte)
count_list2(list,byte)
clauses
count_list1([],N,N).
count_list1([_|T],N,M):- N1=N+1, count_list1(T,N1,M).
count_list2([],0).
count_list1([_|T],N):- count_list1(T,N1), N=N1+1.
goal
count_list1([0,-1,2,6,-9],0,N), count_list2([4,3,-8],K).

```

### 2.11.4 Поиск максимального и минимального элемента в списке

Найти максимальный или минимальный элемент в списке можно при помощи рекурсивных предикатов *max\_list* и *min\_list*. Предикат *max\_list* ищет максимальный элемент в списке на прямом ходе рекурсии, начиная от головы списка, при этом первый параметр типа *integer* является текущим максимумом, а второй - необходим для возвращения результата при выходе из рекурсии. Предикат *min\_list* ищет минимальный элемент в списке на обратном ходе рекурсии, начиная с последнего элемента, при этом параметр типа *integer* является текущим минимумом и результатом одновременно. Два варианта решения задачи приводятся в примере 26.

*Пример 30: поиск максимального и минимального элемента в списке.*

*domains*

*list=integer\**

*predicates*

*max\_list(list, integer, integer)*

*min\_list(list, integer)*

*clauses*

*max\_list ([],M,M).*

*max\_list ([H|T],N,M):- H>N, max\_list(T,H,M).*

*max\_list ([H|T],N,M):- H<=N, max\_list(T,N,M).*

*min\_list ([H|[]],H).*

*min\_list ([H|T],M):- min\_list(T,M1), H>M1, M=H.*

*min\_list ([H|T],M):- min\_list(T,M1), H<=M1, M=M1.*

*goal*

*L=[1,5,3,-6,8,-4],L=[H|T],max\_list(T,H,Max), min\_list([4,3,-8,6],Min).*

### 2.11.5 Сортировка списков

Сортировка представляет собой переупорядочение элементов списка определенным образом. Назначением сортировки является упрощение доступа к нужным элементам. Для сортировки списков обычно применяются три метода:

- метод перестановки,
- метод вставки,
- метод выборки.

Также можно использовать комбинации указанных методов.

Первый метод сортировки заключается в перестановке элементов списка до тех пор, пока он не будет упорядочен. Второй метод осуществляется при помощи неоднократной вставки элементов в список до тех пор, пока он не будет упорядочен. Третий метод включает в себя многократную выборку и перемещение элементов списка.

Второй из методов, метод вставки, особенно удобен для реализации на Прологе.

*Пример 31: сортировка списков методом перестановки (пузырька).*

```
domains
list=integer*
predicates
puz(list,list)
perest(list,list)
clauses
puz(L1,L2):-perest(L1,L3),!,puz(L3,L2).
puz(L1,L1).
perest([X,Y|T],[Y,X|T]):-X>Y.
perest([Z|T],[Z|T1]):-perest(T,T1).
goal
puz([1,3,4,5,2],L).]
```

*Пример 32: сортировка списков методом вставки.*

```
domains
list=integer*
predicates
insert_sort(list,list)
insert(integer,list,list)
asc_order(integer,integer)
clauses
insert_sort([],[]).
insert_sort([H1|T1],L2):-insert_sort(T1,T2),
                           insert(H1,T2,L2).
insert(X,[H1|T1],[H1|T2]):-asc_order(X,H1),!,
                           insert(X,T1,T2).
insert(X,L1,[X|L1]).
asc_order(X,Y):-X>Y.
goal
insert_sort([4,7,3,9],L).
```

Для удовлетворения первого правила оба списка должны быть пустыми. Для того, чтобы достичь этого состояния, по второму правилу происходит рекурсивный вызов предиката *insert\_sort*, при этом значениями *H1* последовательно становятся все элементы исходного списка, которые затем помещаются в стек. В результате исходный список становится пустым и по первому правилу выходной список также становится пустым.

После того, как произошло успешное завершение первого правила, Пролог пытается выполнить второй предикат *insert*, вызов которого содержится в теле второго правила. Переменной *H1* сначала присваивается первое взятое из стека значение 9, а предикат принимает вид *insert(9, [], [9])*.

Так как теперь удовлетворено все второе правило, то происходит возврат на один шаг рекурсии в предикате *insert\_sort*. Из стека извлекается 3 и по третьему правилу вызывается предикат *asc\_order*, то есть происходит попытка удовлетворения пятого правила *asc\_order(3, 9):- 3>9*. Так как

данное правило заканчивается неуспешно, то неуспешно заканчивается и третье правило, следовательно, удовлетворяется четвертое правило и 3 вставляется в выходной список слева от 9:  $insert(3, [9], [3, 9])$ .

Далее происходит возврат к предикату  $insert\_sort$ , который принимает следующий вид:  $insert\_sort([3, 9], [3, 9])$ .

На следующем шаге рекурсии из стека извлекается 7 и по третьему правилу вызывается предикат  $asc\_order$  в виде  $asc\_order(7, 3):- 7>3$ . Так как данное правило заканчивается успешно, то элемент 3 убирается в стек и  $insert$  вызывается рекурсивно еще раз, но уже с хвостом списка – [9]:  $insert(7, [9], \_)$ . Так как правило  $asc\_order(7, 9):- 7>9$  заканчивается неуспешно, то выполняется четвертое правило, происходит возврат на предыдущие шаги рекурсии сначала  $insert$ , затем  $insert\_sort$ .

В результате 7 помещается в выходной список между элементами 3 и 9:  $insert(7, [3, 9], [3, 7, 9])$ .

При возврате еще на один шаг рекурсии из стека извлекается 4 и по третьему правилу вызывается предикат  $asc\_order$  в виде  $asc\_order(4, 3):- 4>3$ . Так как данное правило заканчивается успешно, то элемент 3 убирается в стек и  $insert$  вызывается рекурсивно еще раз, но уже с хвостом списка – [7, 9]:  $insert(4, [7, 9], \_)$ . Так как правило  $asc\_order(4, 7):- 4>7$  заканчивается неуспешно, то выполняется четвертое правило, происходит возврат на предыдущие шаги рекурсии сначала  $insert$ , затем  $insert\_sort$ .

В результате 4 помещается в выходной список между элементами 3 и 7:

$insert(4, [3, 7, 9], [3, 4, 7, 9])$ .  
 $insert\_sort[4, 7, 3, 9], [3, 4, 7, 9])$ .

### 2.11.6 Компоновка данных в список

Иногда при программировании определенных задач возникает необходимость собрать данные из фактов БД в список для последующей их обработки. Пролог содержит встроенный предикат  $findall$ , который позволяет выполнить данную операцию. Описание предиката  $findall$  выглядит следующим образом:

$Findall(Var\_ , Predicate\_ , List\_ )$ , где  $Var\_$  обозначает имя для терма предиката  $Predicate\_$ , в соответствии с типом которого формируются элементы списка  $List\_$ .

Пример 33: использование предиката  $findall$ .

$domains$   
 $d=integer$   
 $predicates$   
 $decimal(d)$   
 $write\_decimal$   
 $clauses$   
 $decimal(0)$

```

decimal (1)
decimal (2)
decimal (3)
decimal (4)
decimal (5)
decimal (6)
decimal (7)
decimal (8)
decimal (9)
write_decimal:- findall(C, decimal (C), L), write (L).
goal
write_decimal.

```

## 2.12 Повторение и рекурсия в Прологе

Очень часто в программах необходимо выполнить одну и ту же операцию несколько раз. В программах на Прологе повторяющиеся операции обычно выполняются при помощи правил, которые используют возврат и рекурсию [3]. Существует четыре способа построения итеративных и рекурсивных правил:

- механизм возврата;
- метод возврата после неудачи;
- правило повтора, использующее бесконечный цикл;
- обобщенное рекурсивное правило.

Правила повторений и рекурсии должны содержать средства управления их выполнением. Встроенные предикаты Пролога *fail* и *cut (!)* используются для управления возвратами, а условия завершения используются для управления рекурсией. Правила выполняющие повторения, используют возврат, а правила, выполняющие рекурсию, используют самовывоз.

### 2.12.1 Механизм возврата

Возврат является автоматически иницируемым системой процессом, если не используются специальные средства управления им. Если в предикате цели есть хотя бы одна свободная переменная, то механизм возврата переберет все возможные способы связывания этой переменной, то есть реализует итеративный цикл.

*Пример 34: распечатать все десятичные цифры.*

```

domains
d=integer
predicates
decimal (d)
write_decimal (d)
clauses

```

```

decimal (0).
decimal (1).
decimal (2).
decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (7).
decimal (8).
decimal (9).
write_decimal (C):- decimal (C), write (C), nl.
goal
write_decimal (C).

```

### 2.12.2 Метод возврата после неудачи

Метод возврата после неудачи может быть использован для управления вычислением внутренней цели при поиске всех возможных решений. Данный метод либо использует внутренний предикат Пролога *fail*, либо условие, которое порождает ложное значение.

*Пример 35: распечатать все десятичные цифры.*

```

domains
d=integer
predicates
decimal (d)
write_decimal
clauses
decimal (0)
decimal (1).
decimal (2).
decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (7).
decimal (8).
decimal (9).
write_decimal:- decimal (C), write (C), nl, fail.
goal
write_decimal.

```

В программе есть 10 предикатов, каждый из которых является альтернативным предложением для предиката *decimal (C)*. Во время попытки вычислить цель внутренние подпрограммы унификации связывают переменную *C* с термом первого предложения, то есть с цифрой 0. Так как существует следующее предложение, которое может обеспечить

вычисление подцели *decimal* (C), то помещается указатель возврата, значение 0 выводится на экран. Предикат *fail* вызывает неуспешное завершение правила, внутренние подпрограммы унификации выполняют возврат и процесс повторяется до тех пор, пока не будет обработано последнее предложение.

Пример 36: подсчитать значения квадратов всех десятичных цифр.

```
domains
d=integer
predicates
decimal(d)
s(d,d)
cicl
clauses
decimal(0).
decimal(1).
decimal(2).
decimal(3).
decimal(4).
decimal(5).
decimal(6).
decimal(7).
decimal(8).
decimal(9).
s(X,Z):-Z=X*X.
cicl:-decimal(I),s(I,S),write(S),nl,fail.
goal
not(cicl).
```

Пример 37: необходимо выдать десятичные цифры до 5 включительно.

```
domains
d=integer
predicates
decimal(d)
write_decimal.
make_cut(d)
clauses
decimal(0).
decimal(1).
decimal(2).
decimal(3).
decimal(4).
decimal(5).
decimal(6).
decimal(7).
```



```

decimal (8).
decimal (9).
write_decimal:- decimal (C), write (C), nl, make_cut (C).
make_cut (C):-C=5.
goal
write_decimal.

```

Предикат ! используется для того, чтобы выполнить отсечение в указанном месте. Неуспешное выполнение предиката make\_cut порождает предикат fail, который используется для возврата и доступа к цифрам до тех пор, пока цифра не будет равна 5.

Пример 38: необходимо выдать из БД первую цифру, равную 5.

```

domains
d=integer
predicates
decimal (d)
write_decimal
clauses
decimal (0).
decimal (5).
decimal (2).
decimal (3).
decimal (4).
decimal (5).
decimal (6).
decimal (5).
decimal (8).
decimal (9).
write_decimal:- decimal (C), C=5, write (C), nl, !.
goal
write_decimal.

```

Если из тела правила убрать предикат !, то будут найдены все три цифры 5, что является результатом применения метода возврата после неудачи. При внесении отсечения будет выдана единственная цифра 5.

### 2.12.3 Метод повтора, использующий бесконечный цикл

Вид правила повторения, порождающего бесконечный цикл:

```

repeat.
repeat:- repeat.

```

Первый repeat является предложением, объявляющим предикат repeat истинным. Однако, поскольку имеется еще один вариант для данного предложения, то указатель возврата устанавливается на первый repeat. Второй repeat – это правило, которое использует само себя как компоненту (третий repeat). Второй repeat вызывает третий repeat, и этот вызов вычисляется успешно, так как первый repeat удовлетворяет подцели repeat.

Предикат *repeat* будет вычисляться успешно при каждой новой попытке его вызвать после возврата. Факт будет использоваться для выполнения всех подцелей. Таким образом, *repeat* это рекурсивное правило, которое никогда не бывает неуспешным. Предикат *repeat* широко используется в качестве компонента других правил, который вызывает повторное выполнение всех следующих за ним компонентов.

*Пример 39: ввести с клавиатуры целые числа и вывести их на экран. Программа завершается при вводе числа 0.*

```
domains
d=integer
predicates
repeat
write_decimal
do_echo
check (d)
clauses
repeat.
repeat:- repeat.
write_decimal:-nl, write(«Введите, пожалуйста, цифры»), nl,
write(«Для останова, введите 0»), nl.
do_echo:- repeat, readln (D), write(D), nl, check (D), !.
check (0):- nl, write («-OK!»).
check(_):- fail.
goal
write_decimal, do_echo.
```

Правило *do\_echo* – это конечное правило повтора, благодаря тому, что предикат *repeat* является его компонентом и вызывает повторение предикатов *readln*, *write*, и *check*. Предикат *check* описывается двумя предложениями. Первое предложение будет успешным, если вводимая цифра 0, при этом курсор сдвигается на начало следующей строки и на экране появляется сообщение «OK!» и процесс повторения завершается, так как после предиката *check* в теле правила следует предикат отсечения. Если введенное значение отличается от 0, то результатом выполнения предиката *check* будет *fail* в соответствии со вторым предложением. В этом случае произойдет возврат к предикату *repeat*. *Repeat* повторяет посылки в правой части правила, находящиеся правее *repeat* и левее условия выхода из бесконечного цикла.

### 2.13 Методы организации рекурсии

*Рекурсивная процедура* – это процедура, которая вызывает сама себя. В рекурсивной процедуре нет проблемы запоминания результатов ее выполнения потому, что любые вычисленные значения можно передавать из одного вызова в другой, как аргументы рекурсивного предиката.

Например, в приведенной ниже программе вычисления факториала (пример 48), приведен пример написания рекурсивного предиката. При этом Пролог создает новую копию предиката *factorial* таким образом, что он становится способным вызвать сам себя как самостоятельную процедуру.

**При этом не происходит копирования кода выполнения, но все аргументы и промежуточные переменные копируются в стек, который создается каждый раз при вызове рекурсивного правила.**

Когда выполнение правила завершается, занятая стеком память освобождается и выполнение продолжается в стеке правила-родителя.

*Пример 40: написать программу вычисления факториала.*

```
predicates
factorial (byte, word)
clauses
factorial (0, 1).
factorial (1, 1):-!.
factorial (N, R):- N1=N-1, factorial (N1, R1), R=N*R1.
goal
f (7, Result).
```

Для вычисления факториала используется последовательное вычисление произведения ряда чисел. Его значение образуется после извлечения значений соответствующих переменных из стека, используемых как список параметров для последнего предиката в теле правила. Этот последний предикат вызывается после завершения рекурсии.

*Пример 41: написать программу, генерирующую числа Фибоначчи до заданного значения.*

```
predicates
f (byte, word)
clauses
f (1, 1).
f (2, 1).
f (N, F):- N1=N-1, f (N1, F1), N2=N1-1, f (N2,F2), F=F1+F2.
goal
f (10, Fib).
```

У рекурсии есть три основных преимущества:

- логическая простота по сравнению с итерацией;
- широкое применение при обработке списков;
- возможность моделирования алгоритмов, которые нельзя эффективно выразить никаким другим способом (например, описания задач, содержащих в себе подзадачи такого же типа).

У рекурсии есть один большой недостаток – использование большого объема памяти. Всякий раз, когда одна процедура вызывает другую, информация о выполнении вызывающей процедуры должна быть сохранена для того, чтобы вызывающая процедура могла, после завершения вызванной процедуры, возобновить выполнение на том месте, где остановилась.

Рассмотрим специальный случай, когда процедура может вызвать себя без сохранения информации о своем состоянии.

Предположим, что процедура вызывается последний раз, то есть после вызванной копии, вызывающая процедура не возобновит свою работу, при этом стек вызывающей процедуры должен быть заменен стеком вызванной копии. Тогда аргументам процедуры просто присваиваются новые значения, и выполнение возвращается на начало вызывающей процедуры. С процедурной точки зрения этот процесс напоминает обновление управляющих переменных в цикле.

***Эта операция в Visual Prolog называется оптимизацией хвостовой рекурсии или оптимизацией последнего вызова.***

Создание хвостовой рекурсии в программе на Прологе означает, что:

- вызов рекурсивной процедуры является самой последней посылкой в правой части правила;
- до вызова рекурсивной процедуры в правой части правила не было точек отката.

Приведем примеры хвостовой рекурсии.

*Пример 42: рекурсивный счетчик с оптимизированной хвостовой рекурсией.*

```
count(100).
count(N):-write(N),nl,N1=N+1,count(N1).
goal
nl, count(0).
```

*Модифицируем этот пример так, чтобы хвостовой рекурсии не стало.*

*Пример 43: рекурсивный счетчик без хвостовой рекурсии.*

```
count1(100).
count1(N):-write(N),N1=N+1,count1(N1),nl.
goal
nl, count1(0).
```

Из-за вызова предиката *nl* после вызова рекурсивного предиката должен сохраняться стек.

*Пример 44: рекурсивный счетчик без хвостовой рекурсии.*

```
count2(100).
count2(N):-write(N),nl,N1=N+1,count2(N1).
count2(N):-N<0, write("N – отрицательное число").
goal
nl, count2(0).
```

Здесь есть непроверенная альтернатива до вызова рекурсивного предиката (третье правило), которое должно проверяться, если второе правило завершится неудачно. Таким образом, стек должен быть сохранен.

*Пример 45: рекурсивный счетчик без хвостовой рекурсии.*

```
count3(100).
count3(N):-write(N),nl,N1=N+1,check(N1),count3(N1).
```

```

check(Z):-Z>=0.
check(Z):-Z<0.
goal
nl, count3(0).

```

Здесь тоже есть непроверенная альтернатива до вызова рекурсивного предиката (предикат *check*). Случаи в примерах 44 и 45 хуже, чем в примере 42, так как они генерируют точки возврата.

Очень просто сделать рекурсивный вызов последним в правой части правила, но как избежать альтернатив? Для этого следует использовать предикат отсечения, который предотвращает возвраты в точки, левее предиката отсечения. Модифицируем 44 и 45 примеры так, чтобы была хвостовая рекурсия.

*Пример 46: рекурсивный счетчик из примера 41 с хвостовой рекурсией.*

```

count4(100).
count4(N):-N>0,!,write(N),N1=N+1,count4(N1).
count4(N):-write("N – отрицательное число").
goal
nl, count4(0).

```

*Пример 47: рекурсивный счетчик из примера 42 с хвостовой рекурсией.*

```

count5(100).
count5(N):-write(N),N1=N+1 check(N1),!, count5(N1).
check(Z):-Z>=0.
check(Z):-Z<0.
goal
nl, count5(0).

```

## 2.14 Создание динамических баз данных

В Прологе существуют специальные средства для организации внутренних и внешних баз данных. Эти средства рассчитаны на работу с реляционными базами данных. Внутренние подпрограммы унификации осуществляют автоматическую выборку фактов из внутренней (динамической) базы данных с нужными значениями известных параметров и присваивают значения неопределенным параметрам.

Раздел программы *facts* в *Visual Prolog* предназначен для описания предикатов динамической (внутренней) базы данных. База данных называется динамической, так как во время работы программы из нее можно удалять любые факты, а также добавлять новые факты. В этом состоит ее отличие от статических баз данных, где факты являются частью кода программы и не могут быть изменены во время исполнения.

Иногда бывает полезно иметь часть информации базы данных в виде фактов статической БД - эти данные заносятся в динамическую БД сразу после активизации программы. В общем случае, предикаты статической БД имеют другое имя, но ту же самую форму представления данных, что и предикаты динамической БД. Добавление латинской буквы *d* к имени

предиката статической БД - обычный способ различать предикаты динамической и статической БД.

Следует отметить два ограничения, объявленные в разделе *facts* :

- в динамической базе данных Пролога могут содержаться только факты;
- факты базы данных не могут содержать свободные переменные.

Допускается наличие нескольких разделов *facts* , тогда в описании каждого раздела *facts* нужно явно указать его имя, например *facts – mydatabase*. В двух различных разделах *facts* нельзя использовать одинаковые имена предикатов. Также нельзя использовать одинаковые имена предикатов в разделах *facts* и *predicates*. Если имя базы данных не указывается, то ей присваивается стандартное имя *dbasedom*. Программа может содержать локальные безымянные разделы фактов, если она состоит из единственного модуля, который не объявлен как часть проекта. Среда разработки компилирует программный файл как единственный модуль только при использовании утилиты *TestGoal*. Иначе безымянный раздел фактов должен быть объявлен глобальным, то есть как *global facts*.

В Прологе есть специальные встроенные предикаты для работы с динамической базой данных:

- *assert*;
- *asserta*;
- *assertz*;
- *retract*;
- *retractall*;
- *save*;
- *consult*.

Предикаты *assert*, *asserta*, *assertz*, - позволяют занести факт в БД, а предикаты *retract*, *retractall* - удалить из нее уже имеющийся факт.

Предикат *assert* заносит новый факт в БД в произвольное место, предикат *asserta* добавляет новый факт перед всеми уже внесенными фактами данного предиката, *assertz* добавляет новый факт после всех фактов данного предиката.

Предикат *retract* удаляет из БД первый факт, который сопоставляется с заданным фактом, предикат *retractall* удаляет из БД все факты, которые сопоставляются с заданным фактом.

Предикат *save* записывает все факты динамической БД в текстовый файл на диск, причем в каждую строку файла заносится один факт. Если файл с заданным именем уже существует, то старый файл будет затерт.

Предикат *consult* записывает в динамическую БД факты, считанные из текстового файла, при этом факты из файла дописываются в имеющуюся БД. Факты, содержащиеся в текстовом файле должны быть описаны в разделе *domains*.

*Пример 48: Написать программу, генерирующую множество 4-разрядных двоичных чисел и записывающих их в динамическую БД.*

*facts*  
*dbin (byte, byte, byte, byte)*  
*predicates*  
*cifra (byte)*  
*bin (byte, byte, byte, byte)*  
*clauses*  
*cifra (0).*  
*cifra (1).*  
*bin (A, B, C, D):- cifra (A), cifra (B), cifra (C), cifra (D),*  
*assert (bin (A, B, C, D)).*  
*goal*  
*bin (A, B, C, D).*

Пример 49: Написать программу, подсчитывающую число обращений к программе.

*facts*  
*dcount (word)*  
*predicates*  
*modcount*  
*clauses*  
*dcount (0).*  
*modcount:- dcount (N), M=N+1, retract (dcount (N)),asserta (dcount (M)).*  
*goal*  
*modcount.*

Пример 50: Написать программу, определяющую родственные отношения.

*facts*  
*dsisters(symbol,symbol)*  
*dbrothers(symbol,symbol)*  
*predicates*  
*parents(symbol,symbol)*  
*pol(symbol,symbol)*  
*sisters(symbol,symbol)*  
*brothers(symbol,symbol)*  
*clauses*  
*parents (anna, olga).*  
*parents (petr, olga).*  
*parents (anna, irina).*  
*parents (petr, irina).*  
*parents (anna, ivan).*  
*parents (petr, ivan).*  
*pol(olga, w).*  
*pol(anna, w).*  
*pol(petr, m).*  
*pol(irina, w).*

```

pol(ivan, m).
sisters (X, Y):-dsisters(X, Y).
sisters (X, Y):- parents (Z, X), parents (Z,Y),pol(X,w),
pol(Y,w),not(X=Y),not(dsisters(X,Y)),
asserta(dsisters(X, Y)).
brothers (X, Y):-dbrothers(X, Y).
brothers (X, Y):- parents (Z,X), parents l(Z,Y),pol(X,m),
pol(Y,m),not(X=Y),not(dbrothers(X,Y)),
asserta(dbrothers(X,Y)).
goal
sisters (X, Y), save ("mybase.txt").

```

Пример 51: Для базы данных, содержащей сведения о книгах из личной библиотеки, создадим внутреннюю базу данных, куда будем записывать результаты запросов:

```

domains
collector, title, author, publisher = symbol
year = integer
personal_library = book (title, author, publisher, year)
predicates
collection (collector, personal_library)
q1(collector, title, year)
q2(year)
facts
dq1(collector, title, year)
count(year,byte)
clauses
collection (irina, book («Using Turbo Prolog», «Yin with Solomon»,
«Moscow, World», 1993)).
collection (irina, book («The art of Prolog», «Sterling with Shapiro»,
»Moscow, World», 1990)).
collection (petr, book («The art of Prolog», «Sterling with Shapiro»,
»Moscow, World», 1990)).
collection (petr, book («Prolog: a relation language and its applications»,
«John Malpas», »Moscow, Science», 1990)).
collection (anna, book («Prolog: a relation language and its applications»,
«John Malpas», »Moscow, Science», 1990)).
q1(C,T,Y):-dq1(C,T,Y), write(;' ',C, ' ',T, ' ',Y),nl, fail.
q1(C,T,Y):- not(dq1(C,T,_)),collection (C, book( T,_, _, 1990)),
assert(dq1(C,T,Y)), write(C, ' ',T, ' ',Y),nl.
count(2100,0).
q2(Y):-collection (_, book(_, _, _, Y)), count(Yold,Nold), N=Nold+1,
assert(count(Y,N)), write(Y, ' ',N),nl, fail.
goal
%q1(C,T,1990).

```



*q2(1990);count(Y,N).*

## 2.15 Использование строк в Прологе.

Строка – это набор символов. При программировании на Прологе символы могут быть «записаны» при помощи алфавитно-цифрового представления или при помощи их *ASCII*-кодов. Обратный слэш (\), за которым непосредственно следует *ASCII*-код (*N*) символа, интерпретируется как символ. Для представления одиночного символа выражение \N должно быть заключено в апострофы ('N'). Для представления строки символов *ASCII*-коды помещаются друг за другом и вся строка заключается в кавычки («\NNN»).

Операции, обычно выполняемые над строками, включают:

- объединение строк для образования новой строки;
- разделение строки для создания двух новых строк, каждая из которых содержит некоторые из исходных символов;
- поиск символа или подстроки внутри данной строки.

Для удобства работы со строками Пролог имеет несколько встроенных предикатов, манипулирующих со строками:

- *str\_len* – предикат для нахождения длины строки;
- *concat* – предикат для объединения двух строк;
- *frontstr* – предикат для разделения строки на две подстроки;
- *frontchar* – предикат для разделения строки на первый символ и остаток;
- *fronttoken* – предикат для разделения строки на лексему и остаток.

Синтаксис предиката *str\_len*:

*str\_len (Str\_value, Str\_length)*, где первый терм имеет тип *string*, а второй терм имеет тип *integer*.

*Пример 52:*

*str\_len («Today», L)*- в данном случае переменная *L* получит значение 5;

*str\_len («Today», 5)* – в данном случае будет выполнено сравнение длины строки «Today» и 5. Так как они совпали, то предикат выполнится успешно, если бы длина строки не была равна 5, то предикат вылился бы неуспешно.

Синтаксис предиката *concat*:

*concat (Str1, Str2, Str3)*, где все термы имеют тип *string*.

*Пример 53:*

*concat («Today», «Tomorrow», S3)*- в данном случае переменная *S3* получит значение «TodayTomorrow»;

*concat (S1, «Tomorrow», «TodayTomorrow»)* – в данном случае *S1* будет присвоено значение «Today»;

*concat («Today», S2, «TodayTomorrow»)* – в данном случае *S2* будет присвоено значение «Tomorrow»;

*concat* («Today», «Tomorrow», «TodayTomorrow»)- будет проверена возможность склейки строк «Today» и «Tomorrow» в строку «TodayTomorrow».

Синтаксис предиката *frontstr*:

*frontstr* (*Number*, *Str1*, *Str2*, *Str3*), где терм *Number* имеет тип *integer*, а остальные термы имеют тип *string*. Терм *Number* задает число символов, которые должны быть скопированы из строки *Str1* в строку *Str2*, остальные символы будут скопированы в строку *Str3*.

Пример 54:

*frontstr* (6, «Expert systems», *S2*, *S3*)- в данном случае переменная *S2* получит значение «Expert», а *S3* получит значение « systems».

Синтаксис предиката *frontchar*:

*frontchar* (*Str1*, *Char\_*, *Str2*), где терм *Char\_* имеет тип *char*, а остальные термы имеют тип *string*.

Пример 55:

*frontchar* («Today », *C*, *S2*)- в данном случае переменная *C* получит значение «T», а *S2* получит значение «oday»;

*frontchar* («Today », 'T', *S2*) – в данном случае *S2* будет присвоено значение «oday»;

*frontchar* («Today», *C*, «oday») – в данном случае *C* будет присвоено значение «T»;

*frontchar* (*S1*, «T», «oday») – в данном случае *S1* будет присвоено значение «Today»;

*frontchar* («Today», «T», «oday»)- будет проверена возможность склейки строк «T» и «oday» в строку «Today».

Синтаксис предиката *fronttoken*:

*fronttoken* (*Str1*, *Lex*, *Str2*), где все термы имеют тип *string*. В терм *Lex* копируется первая лексема строки *Str1*, остальные символы будут скопированы в строку *Str2*. Лексема – это имя в соответствии с синтаксисом языка Пролог или строчное представление числа или отдельный символ (кроме пробела).

Пример 56:

*fronttoken* («Expert systems», *Lex*, *S2*)- в данном случае переменная *Lex* получит значение «Expert», а *S2* получит значение « systems».

*fronttoken* («\$Expert», *Lex*, *S2*)- в данном случае переменная *Lex* получит значение «\$», а *S2* получит значение «Expert».

*fronttoken* («Expert systems», *Lex*, « systems»)- в данном случае переменная *Lex* получит значение «Expert»;

*fronttoken* («Expert systems», «Expert», *S2*)- в данном случае переменная *S2* получит значение « systems»;

*fronttoken* (*S1*, «Expert», « systems»)- в данном случае переменная *S1* получит значение «Expert systems»;

*fronttoken* («Expert systems», «Expert», « systems»)- в данном случае будет проверена возможность склейки лексемы и остатка в строку «Expert systems».

## 2.16 Преобразование данных в Прологе

Для преобразования данных из одного типа в другой Пролог имеет следующие встроенные предикаты:

*upper\_lower*;

*str\_char*;

*str\_int*;

*str\_real*;

*char\_int*.

Все предикаты преобразования данных имеют два термина. Все предикаты имеют два направления преобразования данных в зависимости от того, какой терм является свободной или связанной переменной.

*Пример 57:*

*upper\_lower* («STARS», S2).

*upper\_lower* (S1, «stars»).

*str\_char* («T», C).

*str\_char* (S, 'T').

*str\_int* («123», N).

*str\_int* (S, 123).

*str\_real* («12.3», R).

*str\_real* (S, 12.3).

*char\_int* ('A', N).

*char\_int* (C, 61).

В Прологе нет встроенных предикатов для преобразования действительных чисел в целые и наоборот, или строк в символы. На самом деле, правила преобразования данных типов очень просты и могут быть заданы в программе самими программистом.

*Пример 58:*

*predicates*

*conv\_real\_int* (real, integer)

*conv\_int\_real* (integer, real)

*conv\_str\_symb* (string, symbol)

*clauses*

*conv\_real\_int* (R, N):- R=N.

*conv\_int\_real* (N, R):- N=R.

*conv\_str\_symb* (S, Sb):- S=Sb.

*goal*

*conv\_real\_int* (5432.765, N). (N= 5432).

*conv\_int\_real* (1234, R). (R=1234).

*conv\_str\_symb* («Visual Prolog», Sb). (Sb=Visual Prolog).

Пример 59: преобразование строки в список символов с использованием предиката *frontchar*.

```
domains
list=char*
predicates
convert (string, list)
clauses
convert («», []).
convert (Str, [H\T]):- frontchar(Str, H, Str1),
                        convert(Str1, T).
```

## 2.17 Представление бинарных деревьев

Одной из областей применения списков является представление множества объектов. Недостатком представления множества в виде списка является неэффективная процедура проверки принадлежности элемента множеству. Используемый для этой цели предикат *member* неэффективен при использовании больших списков.

Для представления множеств могут использоваться различные древовидные структуры, которые обеспечивают более эффективную реализацию проверки принадлежности элемента множеству, в частности, в данном разделе для этой цели рассматриваются бинарные деревья.

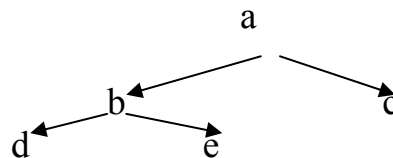
Представление бинарных деревьев основано на определении рекурсивной структуры данных, использующей функцию типа *tree (Top, Left, Right)* или *tree (Left, Top, Right)*, где *Top* - вершина дерева, *Left* и *Right* - соответственно левое и правое поддерево. Пустое дерево обозначим термом *nil*. Объявить бинарное дерево можно следующим образом:

Пример 60:

```
domains
treetype1=tree(symbol, treetype1, treetype1);nil
treetype2=tree(treetype2, symbol, treetype2);nil
```

Пример 61:

Пусть дано дерево следующего вида:



Такое дерево может быть задано следующим образом:

1. левое поддерево: *tree (b, tree (d, nil, nil), tree (e, nil, nil))*.
2. правое поддерево: *tree (c, nil, nil)*.
3. все дерево: *tree (a, tree (b, tree (d, nil, nil), tree (e, nil, nil)), tree (c, nil, nil))*.

*Пример 62: написать программу проверки принадлежности вершины бинарному дереву.*

```
domains
treetype = tree(symbol, treetype, treetype);nil()
predicates
in(symbol, treetype)
clauses
in(X, tree(X,_,_)).
in(X, tree(_L,_) :- in(X, L).
in(X, tree(_R,_) :- in(X, R).
goal
in(d,tree(a, tree(b, tree(d, nil, nil),
                tree(e, nil, nil)),
            tree(c, nil, nil))).
```

Поиск вершины в неупорядоченном дереве также неэффективен, как и поиск элемента в списке. Если ввести отношение упорядочения между элементами множества, то процедура поиска элемента становится гораздо эффективнее. Можно ввести отношение *упорядочения слева направо непустого дерева*  $tree(X, Left, Right)$  следующим образом:

1. Все узлы в левом поддереве *Left* меньше *X*.
2. Все узлы в правом поддереве *Right* больше *X*.
3. Оба поддерева также являются упорядоченными.

Преимуществом упорядочивания является то, что для поиска любого узла в дереве достаточно провести поиск не более, чем в одном поддереве. В результате сравнения узла и корня дерева из рассмотрения исключается одно из поддеревьев.

*Пример 63: написать программу проверки принадлежности вершины упорядоченному слева направо бинарному дереву.*

```
domains
treetype = tree(byte, treetype, treetype);nil()
predicates
in(byte, treetype)
clauses
in(X, tree(X,_,_)).
in(X, tree(Root,L,R) :- Root > X, in(X, L).
in(X, tree(Root,L,R) :- Root < X, in(X, R).
goal
in(6,tree(4, tree(2, nil, tree(3, nil, nil)),
          tree(5,tree(1,nil,nil),nil)),tree(8,tree(7,nil,nil),tree(9,nil,tree(10,nil,nil)
)))
```

*Пример 64: написать программу печати вершин бинарного дерева, начиная от корневой и следуя правилу левого обхода дерева.*

```
domains
treetype = tree(symbol, treetype, treetype);nil()
```

```

predicates
print_all_elements(treetype)
clauses
print_all_elements(nil).
print_all_elements(tree(X, Y, Z)) :-
    write(X), nl, print_all_elements(Y),
    print_all_elements(Z).
goal
print_all_elements(tree(a, tree(b, tree(d, nil, nil),
    tree(e, nil, nil)),
    tree(c, nil, nil))).

```

Пример 65: написать программу проверки изоморфности двух бинарных деревьев.

```

domains
treetype = tree(symbol, treetype, treetype);nil
predicates
isotree (treetype, treetype)
clauses
isotree (T, T).
isotree (tree (X, L1, R1), tree (X, L2, R2)):- isotree (L1, L2), isotree (R1,
R2).
isotree (tree (X, L1, R1), tree (X, L2, R2)):- isotree (L1, R2), isotree (L2,
R1).

```

Пример 66: написать предикаты создания бинарного дерева из одного узла и вставки одного дерева в качестве левого или правого поддеревья в другое дерево.

```

domains
treetype = tree(symbol, treetype, treetype);nil
predicates
create_tree(symbol, tree)
insert_left(tree, tree, tree)
insert_righth(tree, tree, tree)

clauses
create_tree(N, tree(N,nil,nil)).
insert_left(X, tree(A, _,B), tree(A,X,B)).
insert_righth(X,tree(A,B, _), tree(A,B,X)).
goal
create_tree(a, T1), insert_left(tree(b, nil, nil), tree(c, nil, nil), T2),
insert_righth(tree(d, nil, nil), tree(e, nil, nil), T3).

```

В результате:  $T1 = \text{tree}(a, \text{nil}, \text{nil})$ ,  $T2 = \text{tree}(c, \text{tree}(b, \text{nil}, \text{nil}), \text{nil})$ ,  $T3 = \text{tree}(d, \text{nil}, \text{tree}(e, \text{nil}, \text{nil}))$ .

## 2.18 Представление графов в языке Пролог

Для представления графов в языке Пролог можно использовать три способа:

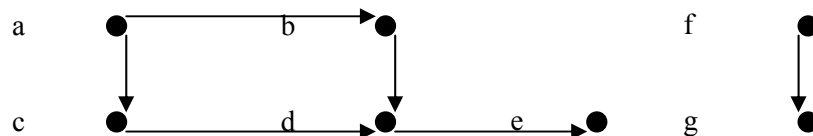
1. Использование факта языка Пролог для описания дуг или рёбер графа.
2. Использование списка или структуры данных для объединения двух списков: списка вершин и списка рёбер.
3. Использование списка списков: каждый подсписок в качестве головы содержит вершину графа, а в качестве хвоста - список смежных вершин.

Две самые распространённые операции, которые выполняются над графами:

- Поиск пути между двумя вершинами;
- Поиск в графе подграфа, который обладает некоторыми заданными свойствами.

*Пример 67:*

*Определить наличие связи между вершинами графа, представленного на рисунке:*



*Две вершины графа связаны, если существует последовательность ребер, ведущая из первой вершины во вторую. Используем для описания структуры графа факты языка Пролог.*

```
domains
top=symbol
predicates
edge (top, top)
/* аргументы обозначают имена вершин */
path( top,top)
/*Предикат connected(symbol, symbol) определяет отношение
связанности вершин.*/
clauses
edge (a, b).
edge (c, d).
edge (a, c).
edge (d, e).
edge (b, d).
edge (f, g).
```

*path* (*X*, *X*).

*path* (*X*, *Y*):- *edge* (*X*, *Z*), *path* (*Z*, *Y*).

Пример 68:

Решить задачу из примера 67, используя списочные структуры для представления графа. Граф задается двумя списками: списком вершин и списком ребер. Ребро представлено при помощи структуры *edge*.

```
domains
edge= e(symbol, symbol)
/* аргументы обозначают имена вершин */
list1=symbol*
list2=edge*
graf = g(list1, list2)
predicates
path(graf, symbol, symbol)
/*Предикат path(graf, symbol, symbol) определяет отношение
связанности вершин в графе.*/
clauses
path (g([],[]),_,_).
path (g([X|_],[e(X,Y)|_]),X,Y).
path (g([X|T],[e(X,_)|T1]),X,Y):-
path (g([X|T],T1),X,Y).
path (g([X|T],[e(_,_)|T1]),X,Y):-
path (g([X|T],T1),X,Y).
path (g([X|T],[e(X,Z)|T1]),X,Y):-
path (g([X|T],T1),Z,Y).
goal
path (g([a, b, c, d],[e(a, b),e(b, c),e(a, d),e(c, b)]), a, c).
```

Пример 69:

Решить задачу из примера 67, используя списочные структуры для представления графа. Граф задается списком списков: в каждом подсписке голова является вершиной, а хвост – списком смежных вершин.

```
domains
edge= e(symbol, symbol)
/* аргументы обозначают имена вершин */
list1=symbol*
list2=edge*
graf = g(list1, list2)
predicates
path(graf, symbol, symbol)
/*Предикат path(graf, symbol, symbol) определяет отношение
связанности вершин в графе.*/
clauses
path (g([],[]),_,_).
path (g([X|_],[e(X,Y)|_]),X,Y).
```



```

%path (g([X|T],[e(X,_)|T1]),X,Y):-
%path (g([X|T],T1),X,Y).
path (g([X|T],[e(X,Z)|T1]),X,Y):-
path (g(T,T1),Z,Y).
path (g([Z|T],T1),X,Y):-Z<>X,
path (g(T,T1),X,Y).
path (g([X|T],[e(_,_)|T1]),X,Y):-
path (g([X|T],T1),X,Y).
goal
path (g([a, b, c, d],[e(a, b),e(b, c),e(a, d),e(c, e)]), b, e).

```

## 2.19 Поиск пути на графе.

Программы поиска пути на графе относятся к классу так называемых недетерминированных программ с неизвестным выбором альтернативы, то есть в таких программах неизвестно, какое из нескольких рекурсивных правил будет выполнено для доказательства до того момента, когда вычисление будет успешно завершено. По сути дела такие программы представляют собой спецификацию алгоритма поиска в глубину для решения определенной задачи. Программа проверки изоморфности двух бинарных деревьев, приведенная в примере 65 относится к задачам данного класса.

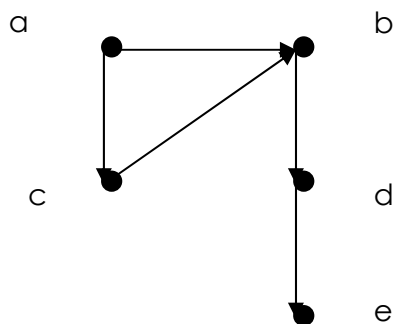
*Пример 70:*

*Определить путь между вершинами графа, представленного на рисунке:*

*A- переменная обозначающая начало пути*

*B- вершина в которую нужно попасть*

*P -ациклический путь на графе (ациклический путь- это путь не имеющий повторяющихся вершин).*



*domains*

*top=symbol*

*listtop=top\**

*predicates*

*edge (top, top)*

*/\* аргументы обозначают имена вершин \*/*

```

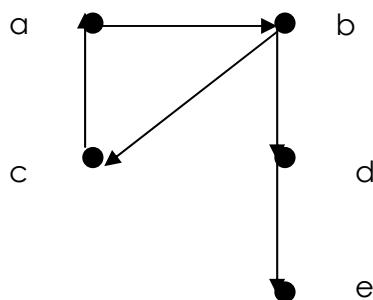
path (top, top, listtop)
/*Предикат path( top, top, listtop) создает список из вершин,
составляющих путь.*/
clauses
edge (a, b).
edge (b, c).
edge (c, a).
edge (b, d).
edge (d, e).
path (A, A, [A]).
path (A, B, [A\P]):-edge(A, N), path(N, B, P).

```

С помощью поиска в глубину осуществляется корректный обход любого конечного дерева или ориентированного ациклического графа. Однако, встречаются задачи, где требуется производить обход графа с циклами. В процессе вычислений может образоваться бесконечный программный цикл по одному из циклов графа.

Неприятностей с закливанием можно избежать двумя способами: ограничением на глубину поиска или накоплением пройденных вершин. Последний способ можно реализовать при помощи модификации отношения path. К аргументам отношения добавляется дополнительный аргумент, используемый для накопления уже пройденных при обходе вершин, для исключения повторного использования одного и того же состояния применяется проверка.

*Пример 71: написать программу обхода конечного ориентированного графа, представленного на рисунке.*



```

domains
top=symbol
listtop=top*
predicates
edge(top,top)
path (top,top,listtop,listtop)
path (top,top)
member(top,listtop)
reverse(listtop,listtop,listtop)
clauses

```

```

edge(a,b).
edge(b,c).
edge(c,a).
edge(b,d).
edge(d,e).
member(A,[A|_]):-!.
member(A,[_|T]):-member(A,T).
reverse([],T2,T2).
reverse([H|T],T1,T2):-reverse(T,[H|T1],T2).
path(A,B,P,[B|P]):-edge(A,B).
path(A,B,P,P2):-edge(A,N),not(member(N,P)),
                P1=[N|P], path (N,B,P1,P2).
path(A,B):-path(A,B,[A],P),reverse(P,[],Res),write(Res).
goal
path(a,e).

```

## 2.20 Метод “образовать и проверить”

Метод “образовать и проверить” – общий прием, используемый при проектировании алгоритмов и программ. Суть его состоит в том, что один процесс или программа генерирует множество предполагаемых решений задачи, а другой процесс или программа проверяет эти предполагаемые решения, пытаясь найти те из них, которые действительно являются решением задачи.

Используя вычислительную модель Пролога, легко создавать логические программы, реализующие метод “образовать и проверить”. Обычно такие программы содержат конъюнкцию двух целей, одна из которых действует как генератор предполагаемых решений, а вторая проверяет, являются ли эти решения приемлемыми. В Прологе метод “образовать и проверить” рассматривается как метод недетерминированного программирования. В такой недетерминированной программе генератор вносит предположение о некотором элементе из области возможных решений, а затем просто проверяется, корректно ли данное предположение генератора.

Для написания программ недетерминированного выбора конкретного элемента из некоторого списка в качестве генератора обычно используют предикат *member* из примера 27, порождающий множество решений. При задании цели *member* (*X*, [1,2,3,4]) будут даны в требуемой последовательности решения *X*=1, *X*=2, *X*=3, *X*=4.

*Пример 72: проверить существование в двух списках одного и того же элемента.*

```

domains
list=integer*
predicates
member (integer, list)

```

```

intersect(list,list)
clauses
member (Head, [Head | _]).
member (Head, [_ | Tail ]):- member (Head, Tail).
intersect (L1, L2):- member(X, L1), member(X, L2).
goal
intersect ([1, 4, 3, 2], [2, 5,6]).

```

Первая подцель *member* в предикате *intersect* генерирует элементы из первого списка, а с помощью второй подцели *member* проверяется, входят ли эти элементы во второй список. Описывая данную программу как недетерминированную, можно говорить, что первая цель делает предположение о том, что *X* содержится в списке *L1*, а вторая цель проверяет, является ли *X* элементом списка *L2*.

Следующее определение предиката *member* с использованием предиката *append*:

```

member(X, L):- append(L1, [X|L2], L)

```

само по существу является программой, в которой реализован принцип «образовать и проверить». Однако, в данном случае, два шага метода сливаются в один в процессе унификации. С помощью предиката *append* производится расщепление списка и тут же выполняется проверка, является ли *X* первым элементом второго списка.

Еще один пример преимущества соединения генерации и проверки дает программа для решения задачи об *N* ферзях: требуется разместить *N* ферзей на квадратной доске размером *NxN* так, чтобы на каждой горизонтальной, вертикальной или диагональной линии было не больше одной фигуры. В первоначальной формулировке речь шла о размещении 8 ферзей на шахматной доске таким образом, чтобы они не угрожали друг другу. Отсюда пошло название задачи о ферзях.

Эта задача хорошо изучена в математике. Для *N=2* и *N=3* решения не существует; два симметричных решения при *N=4* показаны на рисунке. Для *N=8* существует 88 (а с учетом симметричных – 92) решений этой задачи.

		Q	
Q			
			Q
	Q		

	Q		
			Q
Q			
		Q	

Приведенная в *примере 73* программа представляет решение задачи об  $N$  ферзях. Решение задачи представляется в виде некоторой перестановки списка от 1 до  $N$ . Порядковый номер элемента этого списка определяет номер вертикали, а сам элемент – номер горизонтали, на пересечении которых стоит ферзь. Так решение  $[2, 4, 1, 3]$  задачи о четырех ферзях соответствует первому решению, представленному на рисунке, а решение  $[3, 1, 4, 2]$  – второму решению. Подобное описание решений и программа их генерации неявно предполагают, что в любом решении задачи о ферзях на каждой горизонтали и на каждой вертикали будет находиться по одному ферзю.

*Пример 73: программа решения задачи об  $N$  ферзях.*

```
domains
list=integer*
predicates
range (integer, integer, list)
/* предикат порождает список, содержащий числа в заданном
интервале*/
queens (list, list, list)
/* предикат формирует решение задачи о  $N$  ферзях в виде списка
решений, при этом первый список – текущий вариант списка размещения
ферзей, второй список промежуточное решение, третий список -
результат*/
select (integer, list, list)
/*предикат удаляет из списка одно вхождение элемента*/
attack (integer, list)
/*предикат преобразует attack, чтобы ввести начальное присваивание
разности в номерах горизонталей */
attack (integer, integer, list)
/*предикат проверяет условие атаки ферзя другими ферзями из
списка, два ферзя находятся на одной и той же диагонали, на расстоянии  $M$ 
вертикалей друг от друга, если номер горизонтали одного ферзя на  $M$ 
больше или на  $M$  меньше номера горизонтали другого ферзя*/
fqueens (integer, list)
clauses
```

```

range (M, N, [M|T]):- M<N, M1=M+1, range (M1, N, T).
range (N, N, [N]):-!.
select(X,[X|T1],T1).
select (X, [Y|T1], [Y|T2]):-select (X, T1, T2).
attack1 (X, L):- attack(X, 1, L).
attack( X, N, [Y|T2]):-N=X-Y; N=Y-X.
attack( X, N, [Y|T2]):-N1=N+1, attack (X, N1, T2).
queens (L1, L2, L3):-select (X, L1, L11),
                        not (attack1 (X,L2)),
                        queens (L11, [X|L2], L3).
queens ([], L, L).
fqueens(N,L):-range (1, N, L1),
               queens(L1,[],L).
goal
fqueens (4,L),write(L).

```

При таком задании цели, будет выдано второе решение, представленное на рисунке, если задать внешнюю цель, то будут выданы оба решения.

В данной программе реализован принцип «образовать и проверить», так как сначала с помощью предиката *range* генерируется список, содержащий числа от 1 до N. Предикат *select* перебирает все элементы из полученного списка для размещения очередного ферзя, при этом корректность размещения проверяется при помощи предиката *attack*. Таким образом, генератором является предикат *select*, а проверка реализуется при помощи отрицания предиката *attack*. Чтобы проверить, в безопасном положении находится новый ферзь, необходимо знать позиции ранее размещенных ферзей. В данном случае для хранения промежуточных результатов используется второй параметр предиката *queens*, так как решение задачи находится на прямом ходе рекурсии, для закрепления результата при выходе из рекурсии используется третий параметр.

### 3 Основные стратегии решения задач. Поиск решения в пространстве состояний

#### 3.1 Понятие пространства состояния

Пространство состояний – это граф, вершины которого соответствуют ситуациям, встречающимся в задаче («проблемные ситуации»), а решение задачи сводится к поиску путей на этом графе. На самом деле, задача поиска пути на графе и задача о N ферзях - это задачи, использующие одну из стратегий перебора альтернатив в пространстве состояний, а именно – стратегию поиска в глубину.

Рассмотрим другие задачи, для решения которых можно использовать в качестве общей схемы решения пространство состояний.

К таким задачам относятся следующие задачи:  
задача о восьми ферзях;  
переупорядочение кубиков, поставленных друг на друга в виде столбиков;  
головоломка «игра в восемь»;  
головоломка о «ханойской башне»;  
задача о перевозке через реку волка, козы и капусты;  
задача о двух кувшинах;  
задача о коммивояжере;  
другие оптимизационные задачи.  
Со всеми задачами такого рода связано два типа понятий:  
проблемные ситуации;  
разрешенные ходы или действия, преобразующие одни проблемные ситуации в другие.

Проблемные ситуации вместе с возможными ходами образуют направленный граф, называемый пространством состояний.

Пространство состояний некоторой задачи определяет «правила игры»: вершины пространства состояний соответствуют ситуациям, а дуги – разрешенным ходам или действиям, или шагам решения задачи. Конкретная задача определяется:

пространством состояний;  
стартовой вершиной;  
целевым условием или целевой вершиной.

Каждому разрешенному ходу или действию можно приписать его стоимость. Например, в задаче о коммивояжере ходы соответствуют переездам из города в город, ясно, что стоимость хода в данном случае – это расстояние между соответствующими городами.

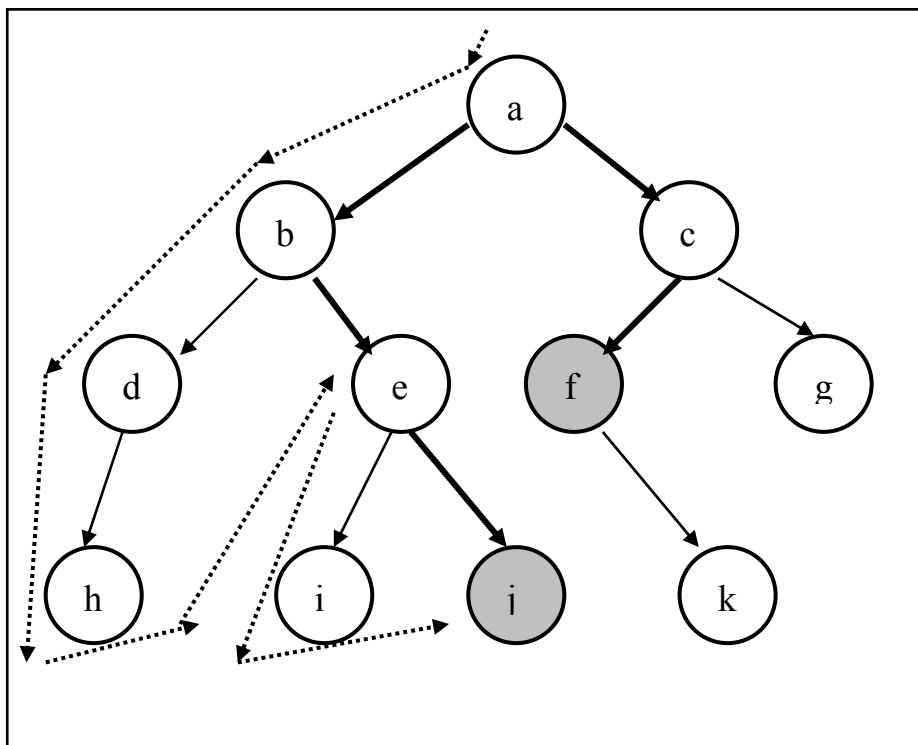
В тех случаях, когда ход имеет стоимость, программист заинтересован в отыскании решения минимальной стоимости. Стоимость решения – это сумма стоимостей дуг, из которых состоит «решающий путь» – путь из стартовой вершины в целевую. Даже если стоимости не заданы, все равно может возникнуть оптимизационная задача: требуется найти кратчайшее решение.

В представленной в примере 73 программе о  $N$  ферзях проблемная ситуация (вершина в пространстве состояний) описывается в виде списка из  $N$   $X$ -координат ферзей, а переход из одной вершины в другую генерирует предикат *queens*, причем начальная ситуация генерируется предикатом *range*, а целевая ситуация определяется при помощи предиката *attack*.

## 3.2 Основные стратегии поиска решений в пространстве состояний

### 3.2.1 Поиск в глубину

Программа решения задачи о N ферзях реализует стратегию поиска в глубину. Под термином «в глубину» имеется в виду тот порядок, в котором рассматриваются альтернативы в пространстве состояний. Всегда, когда алгоритму поиска в глубину надлежит выбрать из нескольких вершин ту, в которую следует перейти для продолжения поиска, он предпочитает самую «глубокую» из них. Самая глубокая вершина – это вершина, расположенная дальше других от стартовой вершины. На следующем рисунке показан пример, который иллюстрирует работу алгоритма поиска в глубину. Этот порядок в точности соответствует результату трассировки процесса вычислений при поиске решения.



Порядок обхода вершин указан пунктирными стрелками, а – начальная вершина, j и f – целевые вершины.

Поиск в глубину наиболее адекватен рекурсивному стилю программирования, принятому в Прологе. Причина этого состоит в том, что обрабатывая цели, Пролог сам просматривает альтернативы именно в глубину.

На Прологе переход от одной проблемной ситуации к другой может быть представлен при помощи предиката *after* ( $X, Y, C$ ), который истинен тогда, когда в пространстве состояний существует разрешенный ход из вершины  $X$  в вершину  $Y$ , стоимость которого равна  $C$ . Предикат *after* может быть задан в программе явным образом в виде фактов, однако такой принцип



оказывается непрактичным, если пространство состояний является достаточно сложным. Поэтому отношение следования *after* обычно определяется неявно, при помощи правил вычисления вершин, следующих за некоторой заданной вершиной.

Другой проблемой при описании пространства состояний является способ представления самих вершин, то есть самих состояний.

В качестве первого примера решения таких задач рассмотрим задачу о ханойских башнях. Есть три стержня и набор дисков разного диаметра. В начале игры все диски надеты на левый стержень. Цель игры заключается в переносе всех дисков на правый стержень по одному стержню за раз, при этом нельзя ставить диск большего диаметра на диск меньшего диаметра. Для этой игры есть простая стратегия:

1. Один диск перемещается непосредственно;
2.  $N$  дисков переносятся в 3 этапа:
  - Перенести  $N-1$  диск на средний стержень;
  - Перенести последний диск на правый стержень;
  - Перенести  $N-1$  диск со среднего на правый стержень.

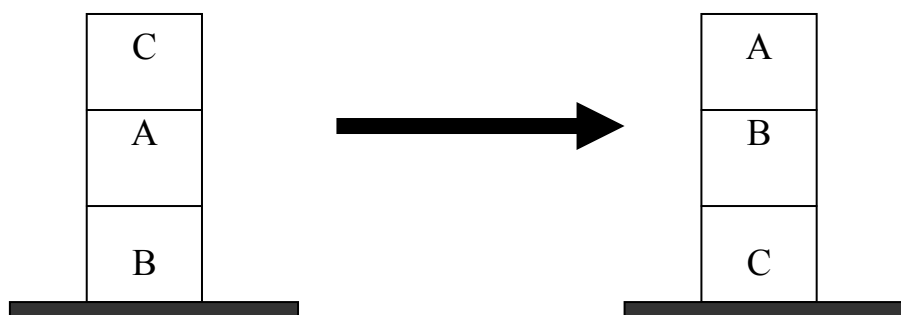
В программе на языке Пролог есть 3 предиката:

- *hanoi* – запускаящий предикат, указывает сколько дисков надо переместить;
- *move* – описывает правила перемещения дисков с одного стержня на другой;
- *inform* – указывает на действие с конкретным диском.

*Пример 74: решение задачи о ханойских башнях.*

```
domains
loc=right;middle;left
% описывает состояние стержней
predicates
hanoi(integer)
% определяет размерность задачи
move(integer,loc,loc,loc)
% определяет переход из одной вершины пространства состояния в
другую, то есть описывает правила перекладывания дисков
inform(loc,loc)
% распечатывает действия с дисками
clauses
hanoi(N):-move(N,left,middle,right).
move(1,A,_,C):-inform(A,C),!.
move(N,A,B,C):-N1=N-1, move(N1,A,C,B),
inform(A,C), move(N1,B,A,C).
inform(Loc1,Loc2):-nl, write("Move a disk from ",Loc1," to ", Loc2).
goal
hanoi(3).
```

В качестве второго примера решения таких задач рассмотрим задачу нахождения плана переупорядочивания кубиков, представленную на следующем рисунке.



На каждом шаге разрешается переставлять только один кубик. Кубик можно взять только тогда, когда его верхняя поверхность свободна. Кубик можно поставить либо на стол, либо на другой кубик. Для того, чтобы получить требуемое состояние, необходимо получить последовательность ходов, реализующую данную трансформацию. В качестве примера будет рассмотрен общий случай данной задачи, когда имеется произвольное число кубиков в столбиках. Число столбиков ограничено некоторым максимальным значением.

Проблемную ситуацию можно представить как список столбиков. Каждый столбик, в свою очередь, представляется списком кубиков, из которых он составлен. Кубики упорядочены в списке таким образом, что самый верхний кубик находится в голове списка. «Пустые» столбики изображаются как пустые списки. Таким образом, исходную ситуацию на рисунке можно записать как терм  $[[c, a, b], [], []]$ .

Целевая ситуация- это любая конфигурация кубиков, содержащая столбик, составленный из имеющихся кубиков в указанном порядке. Таких ситуаций три:

$[[a, b, c], [], []]$ ;

$[[], [a, b, c], []]$ ;

$[[], [], [a, b, c]]$ .

*Пример 75: решение задачи о перемещении кубиков.*

*domains*

*list=symbol\**

*% описывает состояние одного столбика кубиков*

*sit=list\**

*% описывает состояние всех столбиков*

*sits=sit\**

*% описывает путь из начальной вершины в целевую вершину*

*predicates*

*after(sit,sit)*

*% определяет переход из одной вершины пространства состояния в другую, то есть описывает все возможные правила перекладывания кубиков*

```

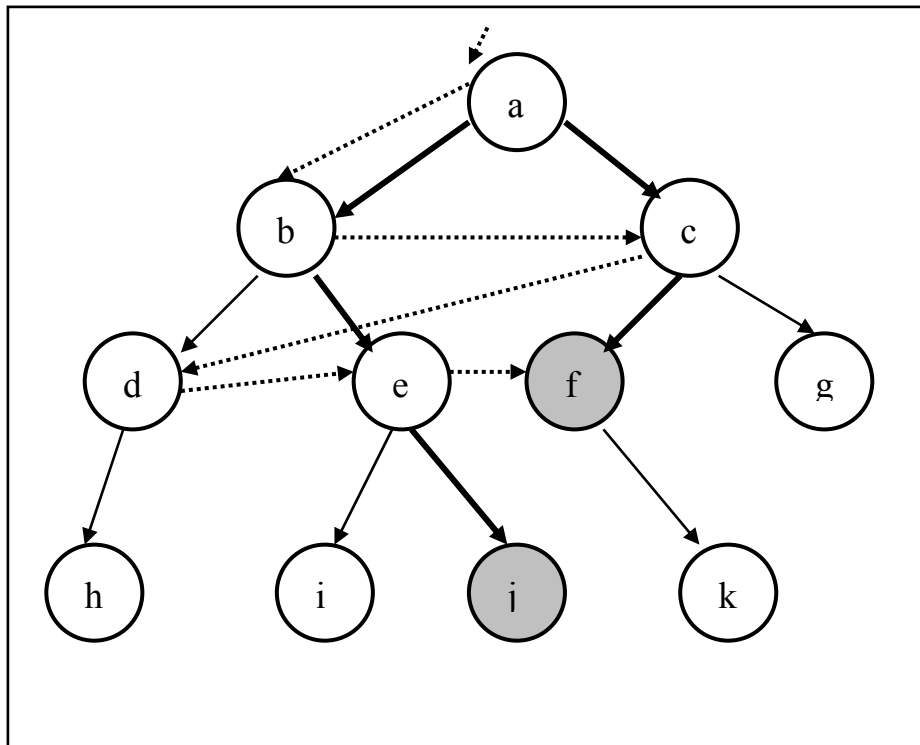
solve (sit, sit, sits, sits)
% определяет путь для решения задачи
member (list, sit)
% первый предикат ищет список в списке списков
member1 (sit, sits)
% второй предикат ищет список списков в списке списков списков
writesp(sits)
% распечатывает путь
clauses
member(X, [X|_]):-!.
member(X,[_|T]):-member(X,T).
member1(X, [X|_]):-!.
member1(X,[_|T]):-member1(X,T).
after([St11,St12,St13],S):-St13=[H3|T3],S=[St11,[H3|St12],T3].
after([St11,St12,St13],S):-St13=[H3|T3],S=[[H3|St11],St12,T3].
after([St11,St12,St13],S):-St12=[H2|T2],S=[[H2|St11],T2,St13].
after([St11,St12,St13],S):-St12=[H2|T2],S=[St11,T2,[H2|St13]].
after([St11,St12,St13],S):-St11=[H1|T1],S=[T1,[H1|St12],St13].
after([St11,St12,St13],S):-St11=[H1|T1],S=[T1,St12,[H1|St13]].
solve(S,S1,Sp,[S1|Sp]):-after(S,S1), member([a,b,c],S1).
solve(S,S2,Sp,Sp2):-after(S,S1), not(member([a,b,c],S1)),
not(member1(S1,Sp)),solve(S1,S2,[S1|Sp],Sp2).
writesp([]).
writesp([H|T]):-writesp(T),write(H),nl.
goal
solve([[c,a,b],[],[[]],S,[[c,a,b],[],[[]],Sp),writesp(Sp).

```

В данном примере реализован усовершенствованный алгоритм поиска в глубину, в котором добавлен алгоритм обнаружения циклов. Предикат *solve* включает очередную вершину в решающий путь только в том случае, если она еще не встречалась раньше.

### 3.2.2 Поиск в ширину

В противоположность поиску в глубину стратегия поиска в ширину предусматривает переход в первую очередь к вершинам, ближайшим к начальной вершине. На следующем рисунке показан пример, который иллюстрирует работу алгоритма поиска в ширину.



Поиск в ширину программируется не так легко, как поиск в глубину. Причина состоит в том, что приходится сохранять все множество альтернативных вершин-кандидатов, а не только одну вершину как при поиске в глубину. Более того, если при помощи процесса поиска необходимо получить решающий путь, то следует хранить не множество вершин-кандидатов, а множество путей-кандидатов. Для представления множества путей-кандидатов обычно используют списки, однако при таком способе одинаковые участки путей хранятся в нескольких экземплярах. Избежать подобной ситуации можно, если представить множество путей-кандидатов в виде дерева, в котором общие участки путей хранятся в его верхней части без дублирования. При реализации стратегии поиска в ширину решающие пути порождаются один за другим в порядке увеличения их длин, следовательно, стратегия поиска в ширину гарантирует получение кратчайшего решения первым.

Представленные выше стратегии поиска в глубину и поиска в ширину не учитывают стоимости, приписанной дугам в пространстве состояний. Если критерием оптимальности является минимальная стоимость пути, а не его длина, то в данном случае поиск в ширину не решает поставленную задачу.

Еще одна проблема, возникающая при решении задачи поиска – это проблема *комбинаторной сложности*. Для сложных предметных областей число альтернатив столь велико, что проблема сложности часто принимает критический характер, так как длина решающего пути (тем более, если их множество, как при реализации поиска в ширину) может привести к *экспоненциальному росту длины в зависимости от размерности задачи*, что приводит к ситуации, называемой *комбинаторным взрывом*. Стратегии

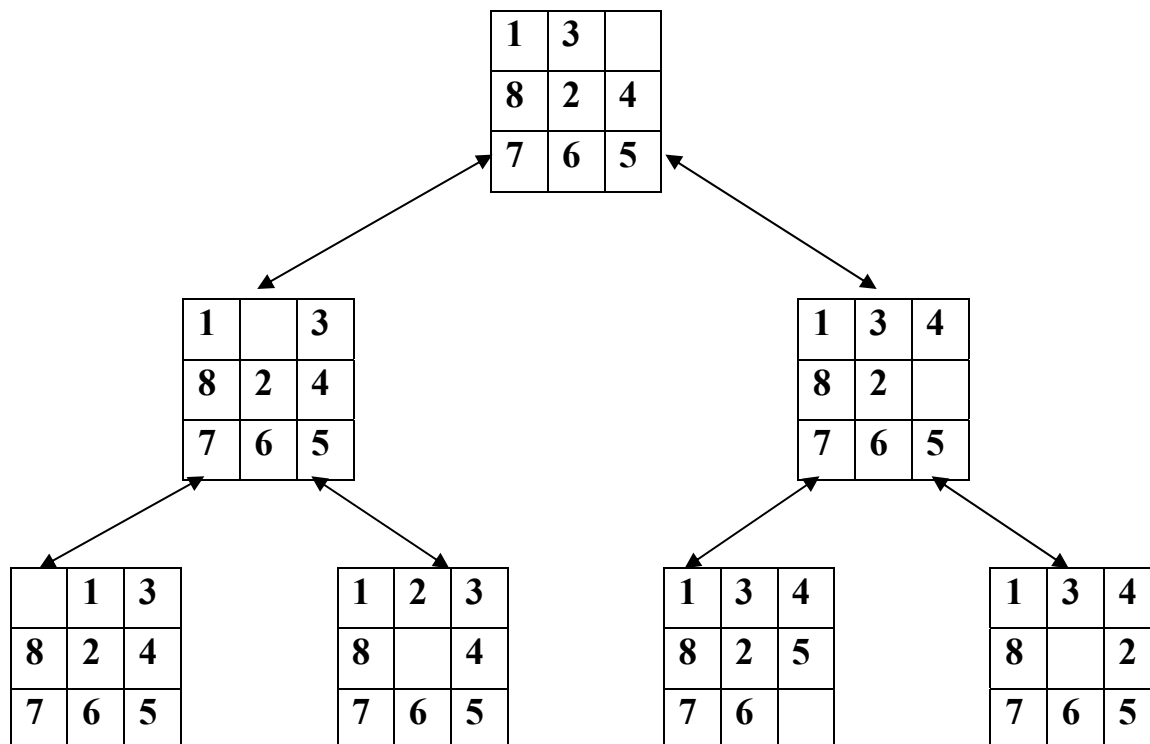
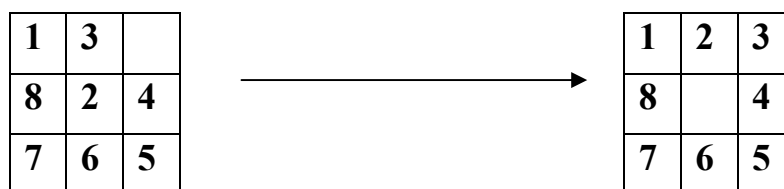
поиска в глубину и в ширину недостаточно «умны» для борьбы с такой ситуацией, так как все пути рассматриваются как одинаково перспективные.

По-видимому, процедуры поиска должны использовать какую-либо информацию, отражающую специфику данной задачи, с тем, чтобы на каждой стадии поиска принимать решения о наиболее перспективных путях поиска. В результате процесс будет продвигаться к целевой вершине, обходя бесполезные пути. Информация, относящаяся к конкретной решаемой задаче и используемая для управления поиском, называется *эвристикой*. Алгоритмы поиска, *использующие эвристики*, называют *эвристическими алгоритмами*.

Одним из эвристических алгоритмов решения задач является алгоритм  $A^*$ , который является усовершенствованным алгоритмом поиска в ширину. Это, так называемый, алгоритм поиска по заданному критерию. Для каждого возможного перехода за один шаг определяется эвристическая оценка и для продолжения поиска решающего пути выбирается наилучшая в соответствии с данной оценкой вершина.

Предполагается, что для всех дуг в пространстве состояний определена функция стоимости перемещения от вершины к вершинам-преемникам. Допустим, для эвристической оценки применяется функция  $f(n)$  такая, что для каждой вершины  $n$  она служит для оценки «сложности достижения  $n$ ». Соответственно наиболее перспективной является та вершина, для которой значение функции  $f(n)$  является минимальным. Рассмотрим алгоритм  $A^*$  на примере решения задачи «игра в восемь».

На следующем рисунке представлена задача «игра в восемь» в виде задачи поиска пути в пространстве состояний. В головоломке используется восемь перемещаемых фишек, пронумерованных цифрами от 1 до 8. Фишки располагаются в девяти ячейках, образующих матрицу  $3 \times 3$ . Одна из ячеек всегда пуста, любая смежная с ней фишка может быть передвинута в эту ячейку. Конечная ситуация – это некоторая заранее заданная конфигурация фишек.



При этом можно выделить четыре основных оператора:

1. Перемещение пустой фишки вниз;
2. Перемещение пустой фишки вверх;
3. Перемещение пустой фишки влево;
4. Перемещение пустой фишки вправо.

Оценочная функция  $f(n)$  формируется как стоимость оптимального пути к цели из начального состояния через  $n$  вершин дерева поиска. Значение оценочной функции для вершины  $n$  равно:  $f(n)=g(n)+h(n)$ , где  $g(n)$  – стоимость оптимального пути от начальной вершины до  $n$ -ой, а  $h(n)$  – стоимость оптимального пути от  $n$ -ой вершины до целевой. Понятно, что  $h(n)$  это эвристическая гипотеза, основанная на имеющейся информации о данной конкретной задаче.

При этом  $g(n)$  принимается равной глубине пройденного пути на дереве поиска от начальной вершины до  $n$ -ой, а  $h(n)$  – расстояние Хемминга от  $n$ -ой вершины до целевой (в данном случае оно равно числу фишек, стоящих не на своих местах). Существует модификация алгоритма  $A^*$ , в которой представляет сумму манхэттеновских расстояний (считается как сумма расстояний в горизонтальном и вертикальном направлениях) от

каждой фишки до её «целевой клетки» плюс утроенное значение «оценки упорядоченности». Оценка упорядоченности определяет степень упорядоченности фишек текущей позиции по отношению к целевой позиции и высчитывается по следующим правилам:

- Фишка в центре имеет оценку 1;
- Фишка в другой позиции имеет оценку 0, если за ней в направлении по часовой стрелке следует соответствующий ей преемник;
- Фишка в другой позиции имеет оценку 2, если за ней в направлении по часовой стрелке не следует соответствующий ей преемник.

Стратегия выбора следующей вершины в пространстве состояний – минимальное значение оценочной функции.

Формулировка алгоритма:

1. Рассматриваем все варианты перемещения пустой фишки за один шаг и выбираем вариант с минимальной оценкой  $h(n)$ .
2. Переходим в новое состояние.
3. Создаём вершины следующего уровня иерархии.
4. Выбираем состояние с минимальной оценкой  $h(n)$ .
5. Повторяем до тех пор, пока не достигнем цели.

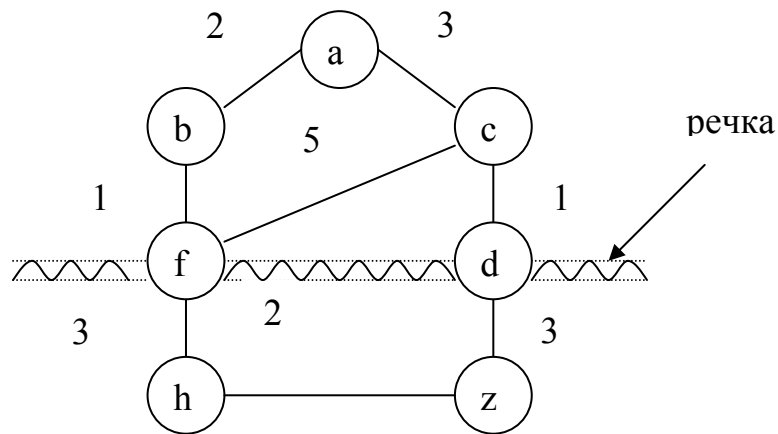
Цель не будет достигнута до тех пор, пока число перемещений меньше числа фишек, находящихся не на своих местах.

Для задачи, изображённой на рисунке, алгоритм находит решение за два шага. От начальной вершины возможен переход в два состояния с оценочной стоимостью  $f(1)=n+h(1)=1+4+3*(1+2)=14$  и  $f(2)=1+3+3*(1+2+2)=19$ . Выбираем минимальную стоимость и переходим в состояние 1. Далее генерируем вершины 3 и 4 с оценочными стоимостями соответственно  $f(3)=2+2+3*(1+2+2)=19$  и  $f(4)=2+0=2$ . Последняя вершина является целевой.

В соответствии с модифицированным алгоритмом оценочные стоимости вершин на первом шаге равны  $f(1)=n+h(1)=1+2=3$  и  $f(2)=1+3=4$

### 3.3 Сведение задачи к подзадачам и И/ИЛИ графы.

Для некоторых категорий задач более естественным решением является разбиение задачи на подзадачи. Разбиение на подзадачи дает преимущество в том случае, когда подзадачи взаимно независимы, и, следовательно, решать их можно независимо друг от друга. Проиллюстрируем это на примере решения задачи поиска на карте дорог маршрута между заданными городами как показано на рисунке.



Вершинами  $a, b, c, d, f, h, z$  – представлены города. Расстояние между городами обозначено весом дуги из одной вершины графа в другую. На карте есть река. Допустим, что переправиться через реку можно только по двум мостам: один находится в городе  $f$ , а второй – в городе  $d$ . Очевидно, что искомый маршрут обязательно должен проходить через один из мостов, а значит должен проходить либо через  $f$ , либо через  $d$ . Таким образом, мы имеем две главные альтернативы:

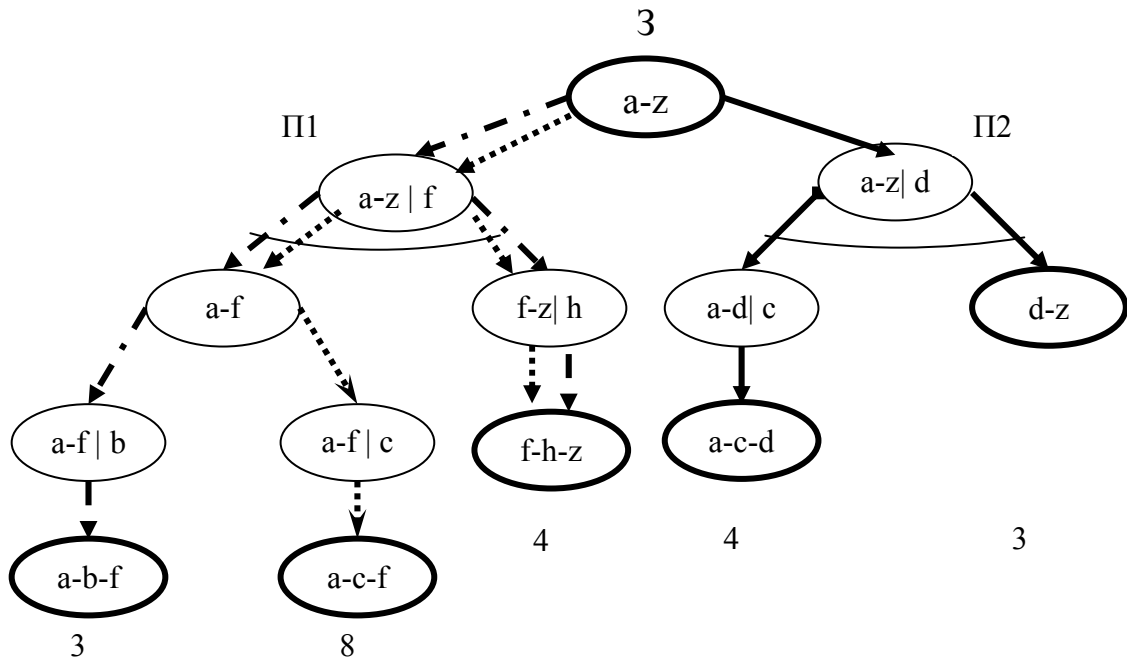
- путь из  $a$  в  $z$ , проходящий через  $f$ ;
- путь из  $a$  в  $z$ , проходящий через  $d$ .

Затем, каждую из этих двух альтернативных задач можно, в свою очередь, разбить следующим образом:

1. для того, чтобы найти путь из  $a$  в  $z$  через  $f$ , необходимо:
  - найти путь из  $a$  в  $f$  и
  - найти путь из  $f$  в  $z$ .
2. для того, чтобы найти путь из  $a$  в  $z$  через  $d$ , необходимо:
  - найти путь из  $a$  в  $d$  и
  - найти путь из  $d$  в  $z$ .

Таким образом, в двух альтернативах мы получили четыре подзадачи, которые можно решать независимо друг от друга. Полученное разбиение исходной задачи можно изобразить в форме И/ИЛИ – графа, представленного на рисунке.





Круглые дуги на графе указывают на отношение **И** между соответствующими подзадачами. Задачи более низкого уровня называются задачами-преемниками.

И/ИЛИ-граф – это направленный граф, вершины которого соответствуют задачам, а дуги – отношениям между задачами.

Между дугами также существуют свои отношения – это отношения **И** и **ИЛИ**, в зависимости от того, должны ли мы решить только одну из задач-преемников или же несколько из них. В принципе из вершины могут выходить дуги, находящиеся в отношении **И** вместе с дугами, находящимися в отношении **ИЛИ**. Тем не менее, будем предполагать, что каждая вершина имеет либо только **И**-преемников, либо только **ИЛИ**-преемников, так как в такую форму можно преобразовать любой И/ИЛИ-граф, вводя в него при необходимости вспомогательные ИЛИ-вершины. Вершину, из которой выходят только **И**-дуги называются **И**-вершиной; вершину, из которой выходят только **ИЛИ**-дуги, – **ИЛИ**-вершиной.

Решением задачи, представленной в виде И/ИЛИ-графа является решающее дерево, так как решение должно включать в себя все подзадачи **И**-вершин.

Решающее дерево **T** определяется следующим образом:

исходная задача **P** – это корень дерева **T**;

если **P** является **ИЛИ**-вершиной, то в **T** содержится только один из ее преемников (из И/ИЛИ-графа) вместе со своим собственным решающим деревом;

если **P** – это **И**-вершина, то все ее преемники (из И/ИЛИ-графа) вместе со своими решающими деревьями содержатся в **T**.

На представленном выше И/ИЛИ- графе представлены три решающих дерева, обозначенных штих-пунктирной, пунктирной и сплошной линиями. Соответственно, стоимости данных деревьев составляют 7, 12, 7. В данном случае стоимости определены как суммы стоимостей всех дуг дерева. Иногда стоимость решающего дерева определяется суммой стоимостей всех его вершин. В соответствии с заданным критерием, из всех решающих деревьев выбирается оптимальное.

### 3.4 Решение игровых задач в терминах И/ИЛИ- графа

Такие игры, как шахматы или шашки, естественно рассматривать как задачи, представленные И/ИЛИ- графами. Игры такого рода называются играми двух лиц с полной информацией. Будем считать, что существует только два возможных исхода игры:

- выигрыш;
- проигрыш.

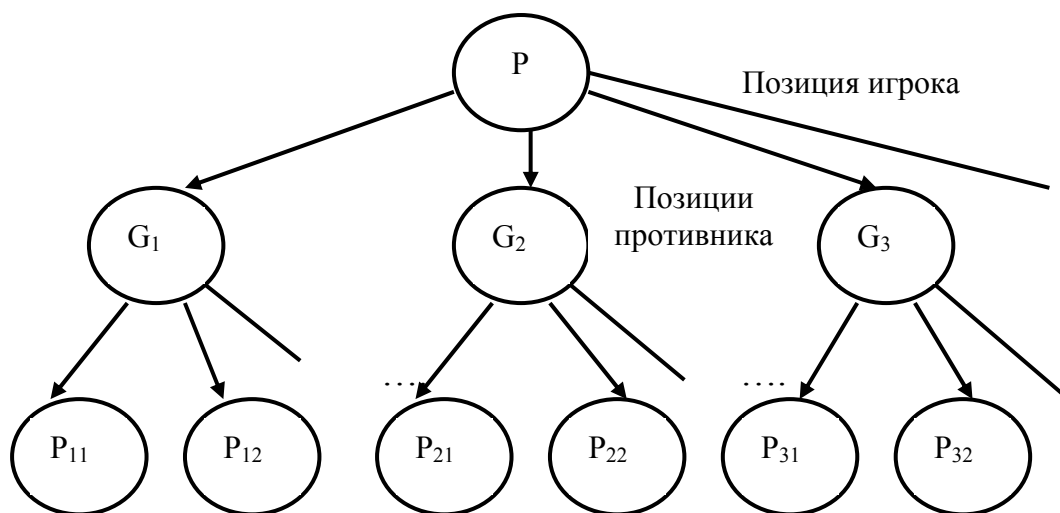
Игры с тремя возможными исходами можно свести к играм с двумя исходами, считая, что есть: выигрыш и невыигрыш.

Так как участники игры ходят по очереди, то выделим два вида позиций, в зависимости от того, чей ход:





- позиция игрока;
- позиция противника.

Допустим, что начальная позиция  $P$  – это позиция игрока. Каждый вариант хода игрока в этой позиции приводит к одной из позиций противника  $G_1, G_2, G_3$  и так далее. Каждый вариант хода противника в позиции  $G_i$  приводит к одной из позиций игрока  $P_{ij}$ .

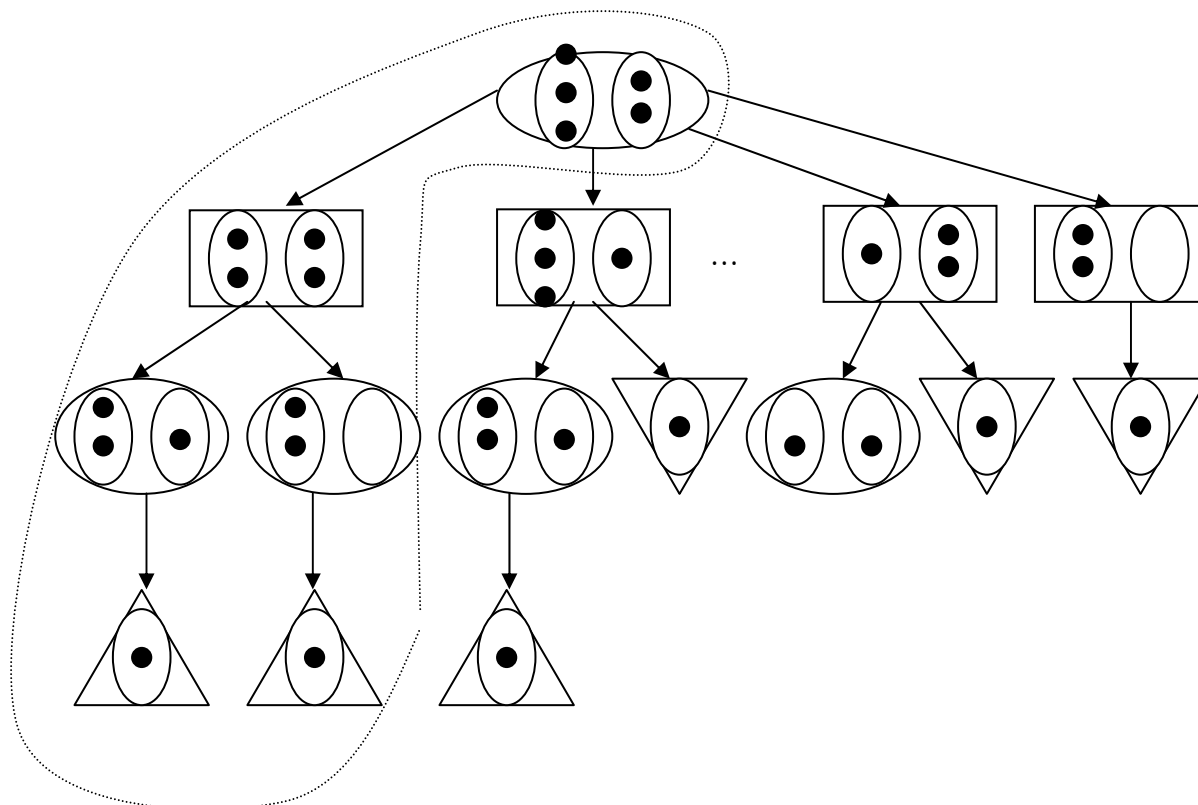
В И/ИЛИ- дереве, показанном на рисунке, вершины соответствуют позициям, а дуги – возможным ходам. Уровни позиций игрока чередуются в дереве с уровнями позиций противника. Игрок выигрывает в позиции  $P$ , если он выигрывает в  $G_1, G_2, G_3$  и так далее. Следовательно,  $P$  – это ИЛИ-вершина. Позиции  $G_i$  – это позиции противника, поэтому если в этой позиции выигрывает игрок, то он выигрывает и после каждого варианта хода противника, то есть игрок выигрывает в  $G_i$ , если он выигрывает во всех позициях  $P_{ij}$ . Таким образом, все позиции противника – это И-вершины. Целевые позиции – это позиции, выигранные согласно правилам игры. Для того, чтобы решить игровую задачу, мы должны построить решающее дерево, гарантирующее победу игрока независимо от ответов противника. Такое дерево задает полную стратегию достижения выигрыша: для каждого возможного продолжения, выбранного противником, в дереве стратегии есть ответный ход, приводящий к победе.



Рассмотрим решение подобных задач на примере игры в «2 лунки». Игрок или его противник может взять из одной любой лунки любое количество камешков. Проигрывает тот, кто берет последний камешек.

-  - позиция противника
-  - позиция игрока
-  - выигрыш игрока
-  - проигрыш игрока

Дерево решений этой игры представлено на рисунке.



Пунктирной линией обозначена оптимальная стратегия игрока, которая приведет к выигрышу.

### **3.5 Минимаксный принцип поиска решений**

Алгоритмы поиска пути на И/ИЛИ- графах могут использовать стратегии поиска в глубину и ширину, однако, для большинства игр, дерево игры имеет большое количество позиций, что приводит к комбинаторному взрыву при реализации просмотра всех вершин дерева решений.

Основной подход к организации поиска на игровых деревьях использует оценочные функции. Оценочная функция используется для вычисления оценки текущего состояния игры.

Для выбора следующего хода используется простой алгоритм:

найти всевозможные состояния игры, которые могут быть достигнуты за один ход;

используя оценочную функцию, вычислить оценки состояний;

выбрать ход, ведущий к позиции с наивысшей оценкой.

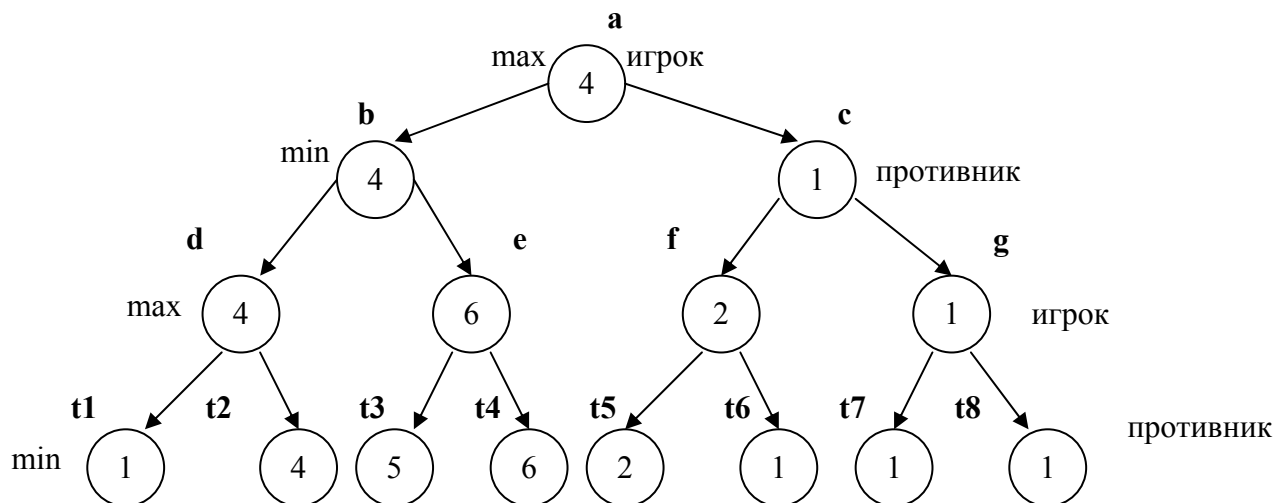
Если оценочная функция была бы совершенной, то есть ее значение отражало бы какие позиции ведут к победе, а какие – к поражению, то достаточно было бы просмотра вперед на один шаг. Обычно совершенная оценочная функция неизвестна, поэтому стратегия выбора хода на основе просмотра на один шаг вперед не дает хорошего результата, поэтому используется стратегия просмотра на несколько шагов вперед.

Стандартный метод определения оценки позиции, основанный на просмотре вперед нескольких слоев игрового дерева, называется минимаксным алгоритмом.

Минимаксный алгоритм предполагает, что противник из нескольких возможных ходов сделает выбор, лучший для себя, то есть худший для игрока. Поэтому целью игрока является выбор такого хода, который даст максимальную оценку позиции, возможной после лучшего хода противника, то есть минимизирующую оценку позиции противника. Отсюда название – минимаксный алгоритм. Число слоев игрового дерева, просматриваемых при поиске, зависит от доступных ресурсов. На последнем слое используется оценочная функция.

В предположении, что оценочная функция выбрана разумно, алгоритм будет давать тем лучшие результаты, чем больше слоев просматривается при поиске.

Пусть мы имеем следующее дерево игры:



Задана некая оценочная функция  $\varphi(P_k)$ , где  $P_k$ - некоторая игровая ситуация.

Предположим, что игрок максимизирует свой выигрыш, а противник минимизирует свой проигрыш. Вариант решения, образованный минимаксной стратегией движения по дереву игры, будем называть основным вариантом решения.

Если существует оценочная функция, то можно ввести внутреннюю функцию  $\varphi(P_k)$  такую, что:

$$\varphi(p_k) = \begin{cases} \max \varphi(p_k) \rightarrow p_k - \text{max вершина} \\ \min \varphi(p_k) \rightarrow p_k - \text{min вершина} \\ \varphi(p_k) \rightarrow p_k - \text{терминальная вершина} \end{cases}$$

Пример 76:

*trace*

*domains*

*pozic = symbol*

*spoz = pozic\**

*database*

*xod (pozic, spoz)*

*xod\_min (pozic)*

*xod\_max (pozic)*

*predicates*

*minmax (pozic, pozic, integer)*

*best (spoz, pozic, integer)*

*oc\_term (pozic, integer)*

*vibor (pozic, integer, pozic, integer, pozic, integer)*

*clauses*

```

xod (a, [b,c]).
xod (b, [d,e]).
xod (c, [f,g]).
xod (d, [t1,t2]).
xod (e, [t3,t4]).
xod (f, [t5,t6]).
xod (g, [t7,t8]).
xod_max (a).
xod_max (d).
xod_max (e).
xod_max (f).
xod_max (g).
xod_min (b).
xod_min (c).
xod_min (t1).
xod_min (t2).
xod_min (t3).
xod_min (t4).
xod_min (t5).
xod_min (t6).
xod_min (t7).
xod_min (t8).
oc_term (a,4).
oc_term (b,4).
oc_term (c,1).
oc_term (d,4).
oc_term (e,6).
oc_term (f,2).
oc_term (g,1).
oc_term (t1,1).
oc_term (t2,4).
oc_term (t3,5).
oc_term (t4,6).
oc_term (t5,2).
oc_term (t6,1).
oc_term (t7,1).
oc_term (t8,1).
minmax (Poz, BestPoz, Oc):-
    xod (Poz, SpPoz),!,
    best(SpPoz, BestPoz, Oc);
    oc_term(Poz, Oc).
best ([Poz], Poz, Oc):- minmax (Poz, _, Oc), !.
best ([Poz1| T], BestPoz, BestOc):-
    minmax (Poz1, _, Oc1),

```

```
best (T, Poz2, Oc2),
vibor(Poz1,Oc1,Poz2,Oc2,BestPoz,BestOc).
vibor(Poz0, Oc0, Poz1, Oc1, Poz0, Oc0):-
    xod_min (Poz0), Oc0>Oc1,!;
    xod_max (Poz0), Oc0<Oc1,!.
vibor(Poz0, Oc0, Poz1, Oc1, Poz1, Oc1).
goal
minmax(a,BestPoz,Oc),write(BestPoz),write(Oc).
```

## Литература

1. Адаменко А.Н., Кучуков А. Логическое программирование и Visual Prolog – Спб.: БХВ – Петербург, 2003.
2. Братко И. Алгоритмы искусственного интеллекта на языке Prolog. М.: Вильямс, 2004. – 637 с.
3. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог: Пер. с англ. М.: Мир.1990.
4. Солдатова О. П., Лёзина И.В. Программирование на языке ПРОЛОГ: метод. указания / О. П. Солдатова, И.В.Лёзина.-Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2008- 52 с.



О. П. Солдатова, И. В. Лёзина

**ЛОГИЧЕСКОЕ ПРОГРАММИРОВАНИЕ  
НА ЯЗЫКЕ VISUAL PROLOG  
УЧЕБНОЕ ПОСОБИЕ**

Компьютерный набор и верстка: О. П. Солдатова, И. В. Лёзина  
Лицензия ЛР №040910 от 10.08.98

Подписано в печать: 16.06.2010 г. Формат 60x84  
Бумага офсетная. Печать офсетная.  
Гарнитура Times New Roman. Объем: 5 усл. печ. л.  
Тираж 100 экз. Заказ № 362

Самарский научный центр Российской академии наук  
443001, г. Самара, Студенческий переулок, 3а

Отпечатано в типографии АНО «Издательство СНЦ РАН»  
443001, г. Самара, Студенческий переулок, 3а  
Тел.: 242-37-07  
443086, Самара, Московское шоссе, 34