Московский государственный университет
имени М. В. Ломоносова

Факультет вычислительной математики и кибернетики

Е.И. Большакова, Н.В. Груздева

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ПРОЛОГ

Учебное пособие

Москва – 2013

УДК
ББК

Рецензенты: доцент, к.ф.-м.н. В.В. Малышко
ст. научный сотрудник, к.ф.-м.н. Ю.С. Владимирова

Большакова Елена Игоревна,
Груздева Надежда Валерьевна
Программирование на языке Пролог: Учебное пособие. – М.: Издательский отдел факультета ВМК МГУ имени М.В.Ломоносова (лицензия ИД № 05899 от 24.09.2001); МАКС Пресс, 2013 – 112 с.

В учебном пособии излагаются основы программирования на языке Пролог, используемом для решения задач искусственного интеллекта и обработки сложных символьных структур. Разбираются методы и средства логического программирования, приводятся примеры пролог-программ. В пособие включен набор задач на логическое программирование, а также дается описание заданий практикума по программированию на языке Пролог, проводимого для студентов 4 курса факультета ВМК МГУ.

Печатается по решению Редакционно-издательского совета факультета вычислительной математики и кибернетики МГУ им. М. В. Ломоносова

ISBN -----

© Издательский отдел факультета вычислительной математики и кибернетики МГУ им. М. В. Ломоносова, 2013
© Большакова Е.И., Груздева Н.В., 2013

Содержание

1 Базовый Пролог ... 6
 1.1 Предикаты, факты и вопросы ... 6
 1.2 Пролог-правила, принципы вычисления ... 11
 1.3 Рекурсивные предикаты ... 17
 1.4 Термы и их сопоставление ... 22
 1.5 Пролог и логика предикатов ... 28
 1.6 Особенности логических программ ... 30
 1.7 Списки и их обработка ... 33
 1.8 Отсечение и бектрекинг ... 40
 1.9 Процедура доказательства целей ... 43
2 Методы и средства программирования ... 46
 2.1 Арифметические вычисления ... 46
 2.2 Использование отсечения ... 50
 2.3 Предикаты работы с базой данных ... 58
 2.4 Оптимизация вычислений ... 61
 2.5 Разностные списки и накапливающий параметр ... 64
 2.6 Предикаты второго порядка ... 67
 2.7 Предикаты ввода/вывода, циклы ... 74
 2.8 Графическая библиотека системы SWI-Prolog ... 78
3 Задачи на логическое программирование ... 88
4 Задания практикума по языку Пролог ... 94
5 Литература ... 110
 Приложение: Встроенные предикаты языка Пролог ... 112

Предисловие

Язык Пролог – один из известных языков программирования для задач искусственного интеллекта и первый язык *логического программирования*, получивший широкое распространение. Работы над языком были начаты в начале 1970-х гг. в научной группе Марсельского университета, занимавшейся задачей машинного перевода. В 1973 г. этой группой на языке Фортран была разработана программа автоматического доказательства теорем, примененная для обработки текстов на ограниченном подмножестве естественного языка [10, 20]. Программа получила название Prolog (*Programmation en Logique*, англ.: *Programming in Logic*) и послужила прообразом современного языка Пролог. Однако эта программа работала достаточно медленно, что дало толчок дальнейшим теоретическим и практическим поискам, завершившимся созданием в 1977 г. в Эдинбургском университете эффективно реализованной версии языка Пролог [22]. Эта реализация языка, известная как *эдинбургская версия,* послужила прототипом для многих последующих версий этого языка и ныне де факто служит стандартом языка Пролог.

С момента возникновения языка было реализовано множество его версий. К числу наиболее известных реализаций, которые поддерживают все основополагающие свойства Пролога, восходящие к его эдинбургской версии, относятся Arity Prolog [15], MicroProlog [6], а также относительно новые реализации: GNU Prolog, SWI-Prolog [21], Strawberry Prolog. В то же время получили распространение и такие неклассические версии, как Turbo Prolog и PDC Prolog [4, 18, 19]: в них для большей эффективности вычислений были допущены существенные отступления от исходных принципов Пролога. В 1996 г. для этих реализаций языка появилась первая интегрированная среда разработки программ Visual Prolog [1].

Язык Пролог представляет *парадигму логического программирования* [8, 14, 17]. Ключевым понятием языка является понятие *предиката* – отношения, связывающего определенные объекты. Логическая программа строится как набор утверждений, описывающих некоторое множество объектов, их отношений и свойств. Вычисление в логической парадигме представляет собой *доказательство* для поиска ответа на вопрос относительно описанных в программе объектов и отношений, причем в случае успешного доказательства в общем случае находятся объекты с нужными свойствами.

Таким образом, для логической парадигмы характерна *декларативность* программ [7]. Программируя задачу в декларативном стиле, программист описывает, ЧТО является её решением (т.е. указывает свойства решения) – в противоположность процедурному, императивному стилю, когда указывается, КАК оно получается, т.е. задаётся последовательность действий для этого.

Высокий уровень и мощность языка Пролог определяют следующие механизмы, встроенные в интерпретатор языка:

- древовидное представление структур данных;
- сопоставление древесных структур (термов);
- автоматический возврат в вычислениях (бэктрекинг).

Как язык программирования для искусственного интеллекта [1, 3, 11], Пролог ориентирован на обработку сложных символьных структур, и в таковом качестве он превосходит многие другие языки. Разработка приложений по символьной обработке данных во многих случаях выполняется на порядок быстрее, а результирующие программы более компактны. К прикладным задачам, решаемым на Прологе, относятся разработка экспертных систем, автоматическое доказательство теорем, создание баз знаний с возможностью логического вывода, планирование и проектирование [1, 3, 13].

В основу данного учебного пособия легли лекции, семинары и практические занятия по программированию на языке Пролог, проводившиеся в течение многих лет на кафедре алгоритмических языков факультета ВМК МГУ. Основное внимание в пособии уделяется изучению принципов логического программирования и базовых средств языка Пролог, представленных во всех его классических версиях. Рассматриваются также дополнительные средства работы с графическими объектами, доступные в версии SWI-Prolog, одной из наиболее удачных современных реализаций этого языка. В пособие включен дополнительный набор задач на логическое программирование, а также дано описание заданий практикума по программированию на языке Пролог типичных задач искусственного интеллекта.

Языку Пролог посвящен ряд книг и учебников, но многие из них труднодоступны. В книгах [2, 3, 7, 17] подробно рассматриваются возможности языка и приёмы программирования на нём, а в [9, 12, 15, 18] содержатся полезные задачи и упражнения по программированию на Прологе.

1 Базовый Пролог

В данном разделе рассматриваются основные средства *логического программирования* на языке Пролог, представленные во всех его клачссических версиях.

1.1 Предикаты, факты и вопросы

Программы на языке Пролог строятся на основе ***предикатов***, фиксирующих отношения между некоторыми объектами. При записи предиката указывается его *имя* (имя отношения), за которым в скобках, через запятую записываются его *аргументы* (объекты, вступающие в это отношение). Число аргументов предиката может быть произвольным.

Рассмотрим в качестве примера предиката с двумя аргументами отношение «быть родителем», выполняющееся для определенных членов некоторой семьи. Имя этого предиката – `parent`, в качестве его первого аргумента должен указываться родитель человека, задаваемого вторым аргументом.

Составим простейшую ***пролог-программу***, включающую 8 пролог-предложений и описывающую отношение родительства между 9 членами семьи. Соответствующая информация может быть представлена генеалогическим деревом – см. Рис. 1. Заметим, что представленная информация неполна, т.к. для многих членов рассматриваемой семьи не указывается их мать.

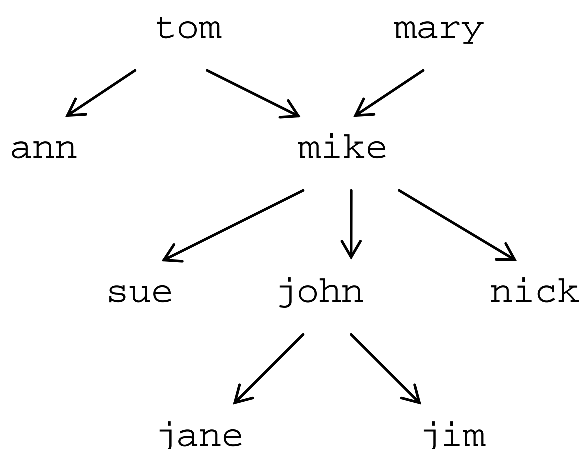

Рис. 1. Генеалогическое дерево

```
/* пролог-программа: факты родительства */
parent(tom,ann).
parent(tom,mike).
parent(mary,mike).
parent(mike,sue).
parent(mike,john).
parent(mike,nick).
parent(john,jane).
parent(john,jim).
```

Первая строка этой программы – *комментарий*. В Прологе строки коментариев заключаются в пары символов `/*` и `*/`, а комментарии, умещающиеся на одной строке, можно записывать после символа `%` и продолжать до конца текущей строки. Далее в этой программе идут пролог-предложения, каждое из которых отображает один известный факт родительства, записанный с помощью предиката `parent` и соответствующих аргументов. Согласно синтаксическим правилам языка Пролог все пролог-предложения заканчиваются точкой. Имена членов семьи в ***предложениях-фактах*** рассматриваемой программы записаны как ***атомы***. Атомы Пролога – простейшие данные, используемые в первую очередь как имена предикатов и их аргументов – конкретных объектов. Поскольку синтаксические правила требуют записи атомов со строчной (маленькой) буквы, то и имена родственников в нашем случае начинаются со строчных букв.

Исполнитель пролог-программ обычно называется *пролог-интерпретатором*, или *пролог-системой*. После ввода программы в интерпретатор можно задавать ***вопросы*** и получать вычисленные ответы (решения). Вопросы суть ***цели*** (целевые утверждения), которые интерпретатор пытается доказать (достигнуть). Простые вопросы записываются по тем же синтаксическим правилам, что и факты-предложения. В данном учебном пособии мы будем начинать запись вопросов с символов **`?-`**, т.к. эти символы обычно выводит пролог-интерпретатор в качестве приглашения к вводу. Ответ интерпретатора будем указывать после стрелки **`=>`** . К примеру, на вопрос `?- parent(tom,mike). => yes`
интерпретатор даст положительный ответ, поскольку в приведенной программе есть подтверждающий (доказывающий) факт. Ответ же на вопрос `?- parent(john,bob). => no`
будет отрицательным (подтверждающего предложения нет).

В вопросах в качестве аргументов могут фигурировать не только атомы, но и ***переменные***, которые в отличие от атомов записываются с заглавной буквы и обозначают не конкретный, а произвольный объект. В приведенных ниже примерах вопросов переменные X и Y обозначают некоторых членов семьи; содержательная интерпретация этих вопросов дана в виде комментария, начинающегося с символа %.

```
?- parent(X,sue).  % Кто родитель sue?
                  => yes, X =mike
?-parent(john,X).  % Кто дети john?
                  => yes, X=jane
                 ; => yes, X=jim
                 ; => no  % Больше решений нет
```

Для второго вопроса в программе есть несколько подтверждающих фактов и, соответственно, несколько ответов. Пролог-интерпретатор сначала находит ответ X=jane (согласно первому найденному подтверждающему факту), но позволяет найти и другие решения. Для поиска альтернативных решений пролог-интерпретатор применяет ***бектрекинг*** (поиск с возвратом). В ряде пролог-систем в диалоговом режиме работы пользователю дается возможность инициировать бектрекинг для поиска новых решений, например, путём ввода с клавиатуры символа ;. Далее мы будем предполагать именно такой режим работы. Заметим однако, что ряд других пролог-систем в подобных случаях автоматически инициируют бектрекинг для поиска всех возможных решений, после чего выдают их все пользователю.

В последнем примере вопроса пользователь дважды запрашивал новое решение (инициируя бектрекинг), и в результате системой был найден ещё один ребенок для john, но затем был получен отрицательный ответ, т.к. детей у него только двое. Порядок найденных решений соответствует порядку расположения в пролог-программе подтверждающих фактов, поскольку при доказательстве целей-вопросов интерпретатор последовательно просматривает предложения программы, начиная с первого. При этом примененное для доказательства предложение помечается – с тем, чтобы при возникшем потом бектрекинге возобновить просмотр программы с предложения, непосредственно следующего за примененным.

Пролог допускает вопросы с несколькими переменными, например: ?- parent(X,Y).

```
          % Найти X и Y такие, что X – родитель Y
```

В режиме бектрекинга будут последовательно найдены 8 решений – все возможные пары X и Y.

Возможны также ***составные цели***, т.е. вопросы из нескольких простых вопросов-целей. Знак запятой в них означает *логическую конъюнкцию*, а каждое вхождение одной переменной подразумевает один и тот же объект. К примеру:

```
?- parent(X, jim),parent(Y, X).
   % Кто родитель jim и родитель его родителя?
```

В результате доказательства указанной составной цели будет получен ответ: X=john, Y=mike.

Следующий вопрос также состоит из двух целей:

```
?-parent(X,ann),parent(X,mike).
    /* Есть ли общий родитель у ann и mike ,
    т.е. являются ли они братом и сестрой? */
```

Система последовательно докажет входящие в эту составную цель подцели и выдаст ответ: => yes, X=tom.

Вот ещё пример составного вопроса-цели:

```
?-parent(mike,X),parent(X,Y).
    /* Кто из детей mike имеет своих детей? */
            => yes, X=john, Y=jane
         ;  => yes, X=john, Y=jim
         ;  => no    % Больше решений нет
```

Интерпретатор на основе бектрекинга найдёт два варианта ответа на этот вопрос. Ход вычислений показан на Рис. 2 в виде *дерева доказательства*. В таком дереве дуги соответствуют шагам доказательства, а в вершинах записываются цели, которые осталось доказать. На дугах указываются номера предложений программы, примененных при доказательстве, и найденные при этом значения переменных. Несколько дуг, выходящих из одной вершины-цели дерева, представляют альтернативные способы доказательства этой цели; такие дуги рисуются слева направо в соответствии с порядком применения соответствующих предложений программы. В листьях дерева расположены либо найденные решения для успешно доказанных целей (решения подчеркнуты и содержат строку yes), либо же цели, доказать которые не возможно. В корневой вершине дерева находится исходная цель, и если она успешно доказана, то в дереве есть путь от корня к листу с пометой yes. В общем случае в дереве доказательства может быть несколько таких путей.

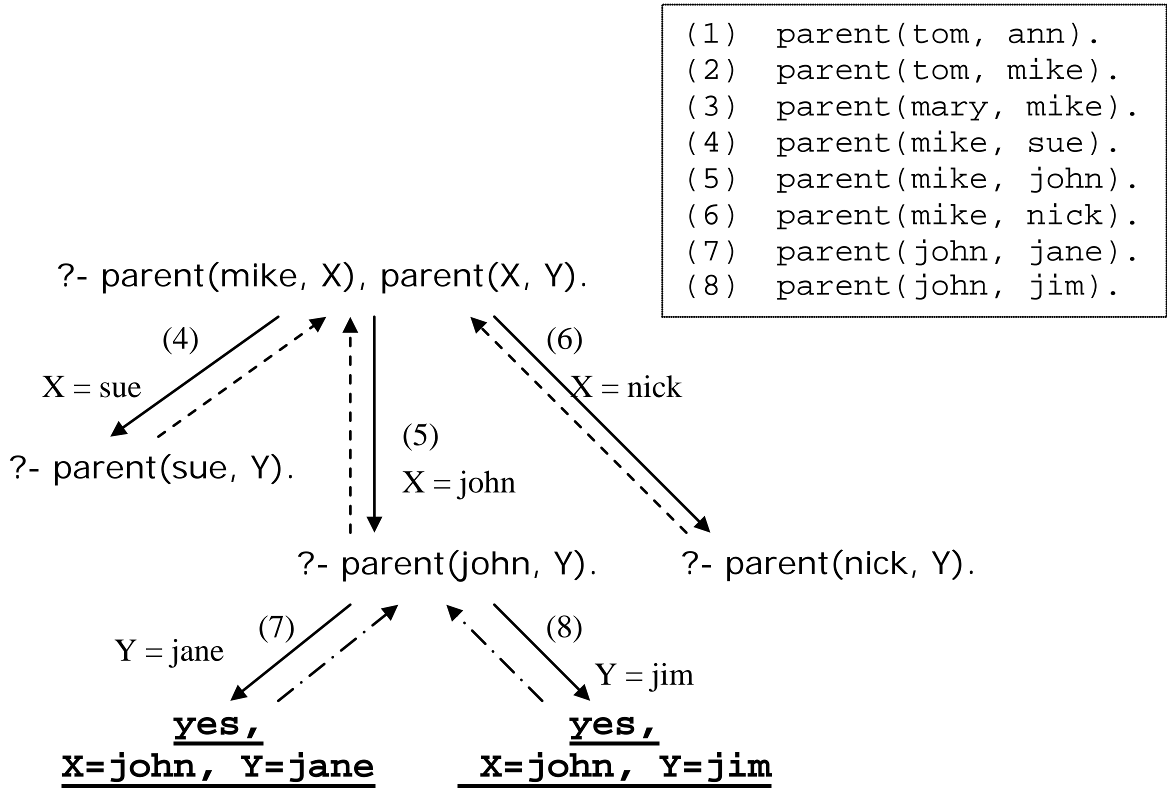

Рис. 2. Доказательство составной цели

Для рассматриваемой цели `parent(mike,X),parent(X,Y)` сначала с помощью предложения (4) пролог-программы успешно доказывается первая подцель, и при этом конкретизируется значение пpеременной: `X=sue`. Это значение подставляется вместо `X` во вторую подцель, и остаётся доказать `parent(sue,Y)`, т.е. найти детей `sue`. Но поскольку в пролог-программе нет подтверждающих фактов, доказательство подцели неуспешно, и пролог-интерпретатор автоматически инициирует бектрекинг (изображаемый в дереве пунктирной линией), т.е. реализует возврат к доказательству цели `parent(mike,X)` для поиска другого её решения. Предложение (5) программы дает альтернативное решение `X=john`, после чего остается доказать `parent(john,Y)`, и доказательство даёт решение `Y=jane`. В случае, если пользователь инициирует поиск ещё одного решения (инициируемый им бектрекинг показан прерывистой линией другого вида), то в результате возврата к цели `parent(john,Y)`будет найдено и выдано решение `Y=jim`. Запрос пользователя на поиск ещё одного решения уже не увенчается успехом, поскольку оба ребенка `john` уже рассмотрены, и

возобновлённый бектрекингом поиск вновь вернется к цели parent(mike,X). Хотя у этой цели есть еще решение: X=nick, однако у nick нет детей, и доказательство цели parent(nick,Y) неуспешно, так что в качестве ответа пролог-система выдаст no .

Отметим, что в некоторых пролог-системах в случае успешного доказательства цели вместо ответа yes выдается true, а в случае неуспеха доказательства вместо no выдается false. Однако вариант yes/no представляется нам более удачным по следующей причине. В то время как ответ yes действительно соответствует логическому значению true, ответ no не эквивалентен логическому false и не означает неверности утверждения-цели: ответ no всего лишь свидетельствует, что цель не удалось доказать (возможно, по причине недостатка информации в программе).

1.2 Пролог-правила, принципы вычисления

Факты и вопросы представляют два разных вида предложений пролог-программ. Ещё один вид предложений – ***правила***, они служат для определения новых отношений на основе уже известных предикатов (отношений).

Определим, к примеру, отношение «быть внуком или внучкой». Предикат с именем grandchild истинен, если его первым аргументом выступает человек, являющийся внуком или внучкой для человека, указанного в качестве второго аргумента (т.е. последний выступает бабушкой или дедушкой для первого). Приведённое ниже пролог-правило для определения этого отношения включает две части, разделенные знаками **:-** . Правая часть, или ***тело правила*** задаёт условия, при котором верно отношение, записанное в левой части, или ***заголовке правила***:

```
grandchild(X,Z) :- parent(Y,X), parent(Z,Y).
```

Данное правило читается так: для всех X и Z объект X – внук или внучка для Z, если найдётся объект Y такой, что Y – родитель для X, и Z – родитель для Y. Таким образом, знаки **:-** в правиле по сути обозначают *логическую импликацию*.

В общем случае в заголовке пролог-правила записывается предикат, а в теле – составная цель: *P* **:-** $G_1, G_2,\ldots G_n$. Для всех переменных правила, фигурирующих в его левой и правой части, мыслится квантор общности (читается как «для всех…»). Для тех же

переменных правила, которые встречаются только в правой части, подразумевается квантор существования (читается как «существует…»).

Сделаем важное уточнение относительно области видимости переменных, используемых в пролог-предложениях. Область действия любой переменной в пролог-программе – ровно одно предложение, в котором оно записано; употребление такой же переменной в другом правиле будет означать, вообще говоря, уже другой объект. Так, например, переменная `X` в определении предиката `grandchild` в общем случае означает вовсе не то же лицо, что в правиле, определяющем отношение `has_child` (свойство `has_child` истинно, если у человека `X` есть ребенок):

```
has_child(X):- parent(X,Y).
```

Отметим важное отличие пролог-правил от пролог-фактов: факт есть безусловно истинное утверждение, в то время как пролог-правило в своей левой части описывает утверждение, которое истинно (доказуемо), только если выполнены все цели в правой части правила.

Уточним теперь **общие принципы доказательства целей пролог-системой:**

I. Цели, образующие **составную цель** $G_1, G_2, \dots G_n$ (она возможна в вопросах и телах пролог-правил), доказываются последовательно, слева направо, т.е. сначала достигается первая цель G_1, затем вторая G_2 и т.д.

II. Для доказательства **простой цели** G выполняется просмотр предложений пролог-программы, начиная с первого и далее, с целью поиска применимого предложения. *Применимое предложение* – это

- либо предложение-факт, *сопоставимый* (согласующийся) с целью G;
- либо предложение-правило, заголовок которого *сопоставим* (согласуем) с G.

Цель ***сопоставима*** с фактом или заголовком правила, если у входящих в них предикатов совпадают имена и арность (количество аргументов), а также попарно сопоставимы аргументы этих предикатов.

При попарном сопоставлении аргументов предикатов возможны следующие варианты пар: атом-атом, атом-переменная, переменная-переменная:

a. атомы сопоставимы, если они тождественно равны;
b. переменная сопоставима с любым атомом – в этом случае *переменная конкретизируется значением* сопоставленного атома; далее этим значением конкретизируется все вхождения этой переменной в доказываемые цели (т.е. в ходе доказательства везде вместо этой переменной подставляется её значение);
c. переменная сопоставима с любой переменной – в этом случае переменные становятся *сцепленными*; и если в ходе дальнейшего доказательства одна из связанных переменных конкретизируется некоторым значением, то такое же значение получает и другая переменная.

Простая цель *G* **достижима** (доказуема), если найденное применимое предложение:

- либо является пролог-фактом;
- либо является пролог-правилом, и при этом достижимы все цели тела этого правила при соответствующих (найденных при сопоставлении *G* и заголовка правила) конкретизациях входящих в них переменных.

III. Если при доказательстве очередной простой цели G_i в составе сложной цели $G_1, G_2, \dots G_n$ оказывается, что G_i не достижима, то пролог-система инициирует **бектрекинг** и осуществляет возврат к строго предыдущей цели G_{i-1} и пытается её доказать вновь, найдя другой вариант решения.

Важно, что при бектрекинге теряют свои значения переменные, конкретизированные при доказательстве цели G_{i-1} и в ходе неуспешного доказательства цели G_i; в случае успеха повторного доказательства G_{i-1} в общем случае будут найдены другие конкретизирующие значения этих переменных.

Покажем ход доказательства цели-вопроса с предикатом `grandchild` – соответствующее дерево доказательства изображено на Рис. 3. Как и ранее, вершины дерева – доказываемые цели и конъюнкции целей. Дуги соответствуют применению правил и фактов программы на соответствующих шагах доказательства. На каждой

```
(*) grandchild(X,Z) :- parent(Y,X), parent(Z,Y).
```

```
?- grandchild(nick, tom).
        (*) | X = nick
            ↓ Z = tom
?- parent(Y, nick), parent(tom, Y).
        (6) | Y = mike
            ↓
?- parent(tom, mike).
        (2) ↓
          yes
```

```
(1) parent(tom, ann).
(2) parent(tom, mike).
(3) parent(mary, mike).
(4) parent(mike, sue).
(5) parent(mike, john).
(6) parent(mike, nick).
(7) parent(john, jane).
(8) parent(john, jim).
```

Рис. 3. Доказательство цели с отношением grandchild

дуге записывается номер (или метка *) примененного предложения, а также конкретизации переменных, найденные на этом шаге.

Дерево на Рис. 3 вырождено: построена только одна ветвь, сразу ведущая к ответу. Можно заметить, что применение пролог-правила при доказательстве порождает новые цели (в общем случае увеличивая их число в вершинах дерева), а применение факта сокращает число подцелей, которые осталось доказать. В частности, лист в дереве возникает, когда единственная цель в вершине дерева доказывается некоторым фактом.

Для определения многих отношений родства требуется информация о поле членов семьи. Её можно задать с помощью фактов на основе унарных предикатов man и woman. При наличии таких фактов можно определить пролог-правило для отношения «быть сестрой» (первый аргумент указывает на лицо, являющееся сестрой для второго аргумента):

```
sister(X, Y) :- parent(Z, Y), parent(Z, X),  %(**)
                              woman(X).
```

Это правило проверяет пол человека X, а также существование у людей X и Y общего родителя. На Рис. 4 показано дерево доказательства для вопроса

```
?- sister(W,jim).  => yes, W=jane
```

```
?- sister(W, jim).
     |  (**)
     |  X=W, Y=jim
     v
?- parent(Z, jim), parent(Z, W), woman(W).
     |  (8)
     |  Z=john
     v
?- parent(john, W), woman(W).
     |  (7)   W=jane
     v
?- woman(jane).
     |
     v
yes, W=jane
```

```
man(tom).
man(mike).
man(john).
man(nick).
man(jim).
woman(mary).
woman(ann).
woman(sue).
woman(jane).
```

```
(1) parent(tom, ann).
(2) parent(tom, mike).
(3) parent(mary, mike).
(4) parent(mike, sue).
(5) parent(mike, john).
(6) parent(mike, nick).
(7) parent(john, jane).
(8) parent(john, jim).
```

Рис. 4. Доказательство цели `sister(W,jim)`

Если же задать вопрос

```
?- sister(W,jane).  => yes, W=jane
```

то вместо ожидаемого ответа `no` получим все тот же ответ `yes` – см. дерево доказательства на Рис. 5 – однако этот ответ противоречит общепринятому пониманию сестры. Нетрудно догадаться, что такой результат связан с тем, что переменные `X` и `Y` в нашем определении отношения `sister` могут означать один и тот же объект, т.е. конкретизироваться одним и тем же значением. Чтобы исправить правило для предиката `sister`, необходимо потребовать, чтобы значения этих переменных были не равны, например, с помощью встроенного предиката **`dif`** (в разных версиях языка Пролог он может обозначаться по-разному; но как мы увидим далее, этот предикат можно запрограммировать базовыми средствами языка). Исправленная версия определения предиката выглядит так:

```
sister(X,Y):- parent(Z,X), parent(Z,Y),
              dif(X,Y), woman(X).
```

```
man(tom).
man(mike).
man(john).
man(nick).
man(jim).
woman(mary).
woman(ann).
woman(sue).
woman(jane).
```

```
?- sister(W, jane).
      | (**)
      | X=W, Y=jane
      v
?- parent(Z, jane), parent(Z, W), woman(W).
      | (7)
      | Z=john
      v
?- parent(john, W), woman(W).
      | (7) W=jane
      v
?- woman(jane).
      |
      v
yes, W=jane
```

```
(1) parent(tom, ann).
(2) parent(tom, mike).
(3) parent(mary, mike).
(4) parent(mike, sue).
(5) parent(mike, john).
(6) parent(mike, nick).
(7) parent(john, jane).
(8) parent(john, jim).
```

Рис. 5. Доказательство цели `sister(W,jane)`

Итак, программирование на Прологе состоит в определении **отношений** (предикатов) и постановке **вопросов**, касающихся этих отношений. Отношение определяется **фактами** и/или **правилами**, задающими условия-**цели**, которые должны быть выполнены для верности отношения. Правила суть наиболее общий вид пролог-предложений. Факты – это правила с пустым телом, т.е. безусловно верные утверждения. Вопросы – правила без заголовка, и входящие в вопрос цели требуется доказать. Таким образом, программирование ведется в терминах целей, а запуск логической программы осуществляется некоторым целевым утверждением.

Процесс получения ответа на заданный вопрос подразумевает достижение всех требуемых целей и в общем случае включает исследование различных вариантов их доказательства с помощью бектрекинга. Бектрекинг инициируется пролог-системой автоматически в случаях невозможности доказать какую-либо цель/подцель (что скрыто от пользователя системы). В то же время после получения ответа на вопрос пользователь может инициировать бектрекинг для поиска других, альтернативных решений.

1.3 Рекурсивные предикаты

Пролог-правила позволяют определять рекурсивные отношения. В качестве примера рассмотрим бинарное отношение «быть предком»: первый аргумент предиката ancestor будет означает предка для лица, фигурирующего в качестве второго аргумента. Для того, чтобы определить это отношение *рекурсивно*, т.е. через него самого, воспользуемся соображением о том, что непосредственный предок – это родитель, а любой более отдалённый предок (дедушка, бабушка, прадедушка или прабабушка и т.д.) предполагает цепочку лиц, связанных отношением родительства. Более точно: X есть отдаленный предок для Z, если он является родителем некоторого человека, который в свою очередь является для Z предком (но уже более близким). Два соответствующих пролог-правила:

```
ancestor(X,Z):- parent(X,Z).    % Правило (*1)
ancestor(X,Z):- parent(X,Y),ancestor(Y,Z).
                                 % Правило (*2)
```

Второе правило получилось рекурсивным. Заметим, что рекурсию можно было записать иным способом:

```
ancestor(X,Z):-parent(Y,Z),ancestor(X,Y).
```

В этом правиле цепочка лиц, связанная отношением родительства, обрабатывается с противоположной стороны, со стороны потомка Z.

Пример доказательства цели с предикатом ancestor показан на Рис. 6. Пунктирной линией в дереве доказательства изображен бектрекинг – возврат к повторному доказательству более ранней цели при неуспехе доказательства более поздней цели (находящейся ниже в дереве доказательства).

Из большинства вершин дерева на Рис.6 выходит по две дуги, соответствующих применению альтернативных правил (*1) и (*2) для предиката ancestor. Согласно принципам вычисления пролог-программ, сначала пробуется правило (*1), а затем при бектрекинге правило (*2), и дуга дерева для правила, применяемого раньше, рисуется левее. Вследствие рекурсивности предиката указанные правила применяются при доказательстве несколько раз. Ясно, что второе применение правил (*1) и (*2) никак не связано с их первым применением. Вообще, при каждом новом применении этих правил переменные X,Y,Z означают другие объекты, отличные от предыдущего применения правила, т.е. эти переменные при разных

```
(*1)  ancestor(X,Z):- parent(X,Z).
(*2)  ancestor(X,Z):- parent(X,Y), ancestor(Y,Z).
```

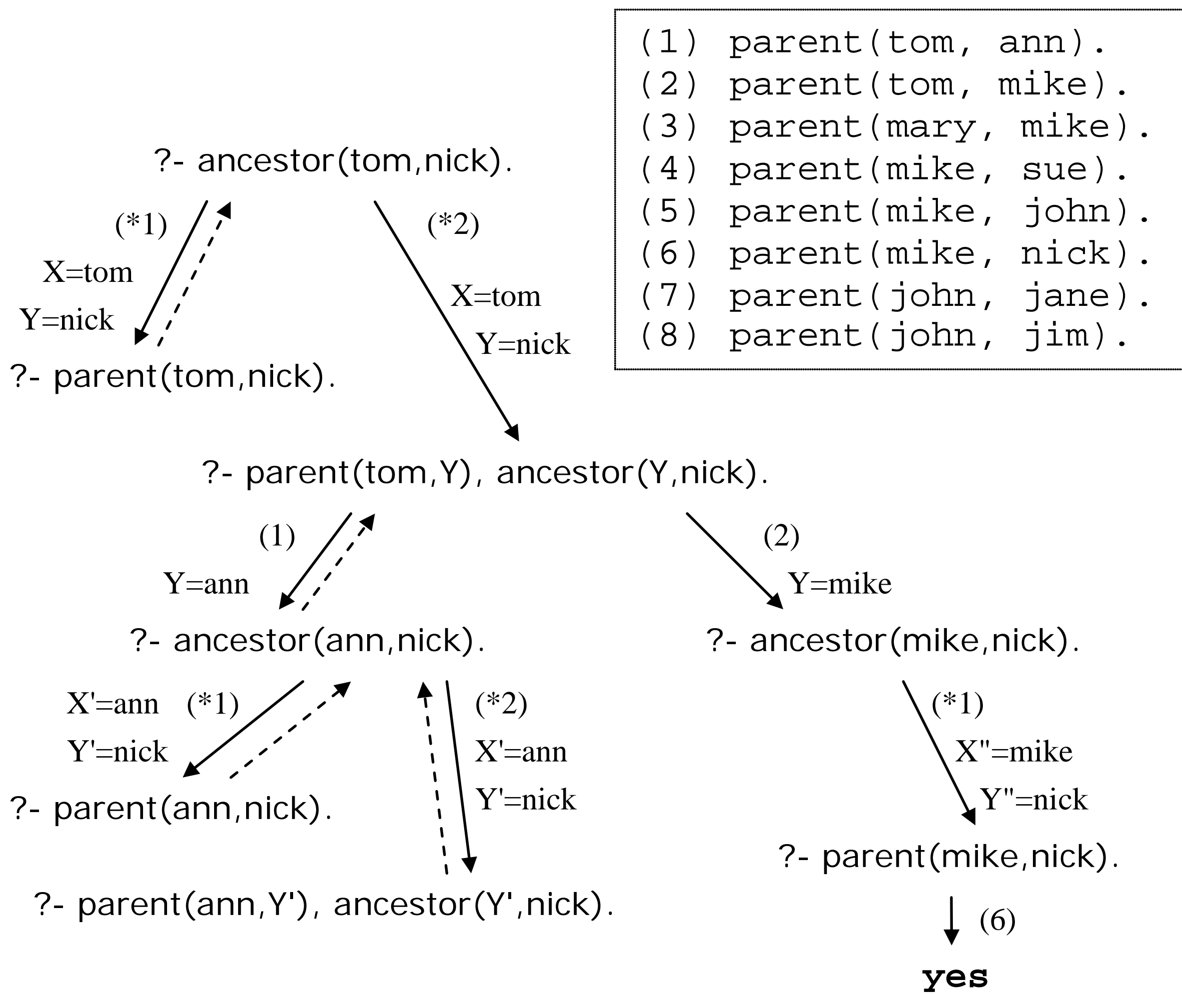

Рис. 6. Дерево доказательства цели `ancestor(tom,nick)`

применениях правил конкретизируются в общем случае разными значениями.

Чтобы запомнить различные конкретизации переменных правила при нескольких его применениях, пролог-система систематически переименовывает переменные правила всякий раз, когда использует его. Строго говоря, переименование выполняется при каждом применении пролог-предложения с переменными. Для исключения коллизий имён система генерирует специальные имена, которые не могут встретиться в исходных пролог-программах (например, имена вида: `_143`, `_143` и т.п.). В дереве доказательства на Рис. 6 каждое необходимое переименование переменных

выполнено путём приписывания дополнительного знака штриха к именам переменных.

Отметим, что объём вычислений при доказательстве некоторой цели (и, как следствие, размер дерева доказательства), зависят от:

- порядка записи фактов и правил в пролог-программе, поскольку поиск применимого предложения идёт от её начала к концу;
- порядка целей в телах правил, поскольку они доказываются последовательно.

Например, изменение порядка правил в предикате `ancestor` потребует большего объема вычислений при доказательстве цели `ancestor(tom,nick)` – в этом можно убедиться, нарисовав соответствующее дерево доказательства. Существенен также порядок целей в теле правила для предиката `sister`: если поменять местами первую и вторую цель, то объём вычислений при доказательстве цели `sister(W,jim)` увеличится.

От порядка предложений в программе и целей в правилах может зависеть также полученный при доказательстве результат, а в случае нескольких возможных решений – порядок их получения.

На Рис. 7 показано еще одно дерево доказательства – для цели `ancestor(mike,Y)`, которая имеет несколько решений, их поиск выполняется в режиме бектрекинга.

Каждый раз, выполняя просмотр предложений программы для доказательства текущей цели, пролог-интерпретатор помечает то предложение, которое оказалось применимым и используется для доказательства. Такая помета, называемая ***точкой бектрекинга***, необходима для того, чтобы в случае неуспеха дальнейшего доказательства можно было вернуться к помеченному предложению и возобновить от него поиск следующего применимого предложения. Если такое предложение находится, то точка бэктрекинга помечает уже его, в ином случае она уничтожается. В общем случае каждая доказываемая цель может иметь свою точку бектрекинга (в дереве на Рис. 7 точка бектрекинга есть у каждой вершины-цели с несколькими исходящими дугами), что позволяет полностью обойти все дерево доказательства и получить все решения. Важно, что выполняя возврат от некоторой вершины-цели дерева в более раннюю вершину, интерпретатор отменяет конкретизации переменных, сделанные ранее при движении между этими вершинами.

```
(*1)  ancestor(X,Y):- parent(X,Y).
(*2)  ancestor(X,Y):- parent(X,Z), ancestor(Z,Y).
```

```
(1)  parent(tom, ann).
(2)  parent(tom, mike).
(3)  parent(mary, mike).
(4)  parent(mike, sue).
(5)  parent(mike, john).
(6)  parent(mike, nick).
(7)  parent(john, jane).
(8)  parent(john, jim).
```

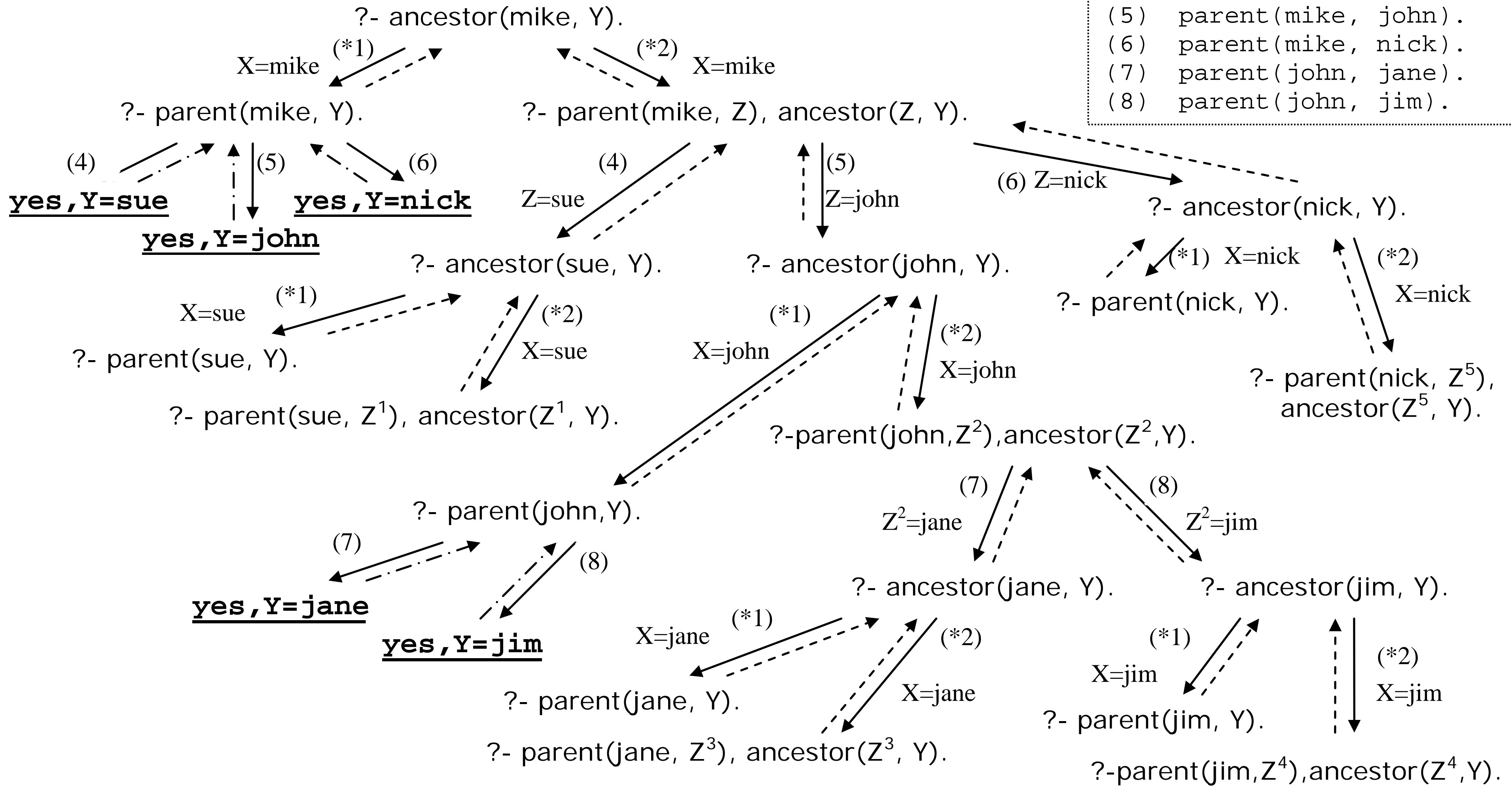

Рис. 7. Дерево доказательства цели `ancestor(mike,Y)`

Рассмотрим ещё один пример рекурсивного пролог-предиката. Предикат `connected(X,Y)` проверяет связанность двух вершин ориентированного графа (*орграфа*), т.е. наличие пути между ними по цепочке ребер, стрелки которых соответствуют направлению пути. Рис. 8 показывает пример орграфа, в нем не все вершины связаны.

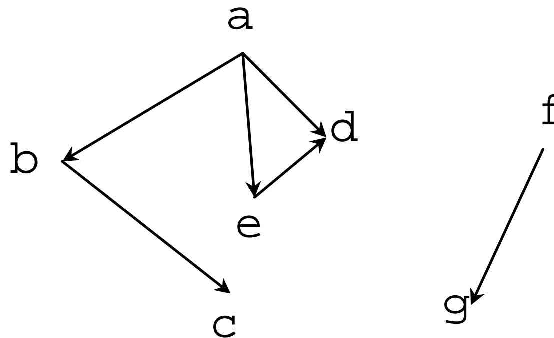

Рис. 8. Пример ориентированного графа

Пусть граф представляется пролог-фактами вида `edge(V,W)`. Такой факт устанавливает наличие ребра, ведущего от вершины `V` к вершине `W`. Аргументы предиката `edge` – атомы, помечающие вершины графа, к примеру: `edge(a,b)`.

Ясно, что две вершины `X` и `Y` орграфа связаны, когда есть ребро, выходящее из `X` и входящее в вершину `Y`. Вершины также связаны, если есть выходящее из `X` ребро и существует путь (последовательность ребёр нужной ориентации), ведущий из вершины-конца этого ребра в вершину `Y`. Тем самым получаем два определяющих правила для предиката `connected`, записанные в нижеследующей пролог-программе:

```
/* Ребра ориентированного графа */
edge(a, b).
edge(a, d).
edge(a, e).
edge(b, c).
edge(e, d).
edge(f, g).
/* Определение связанности вершин орграфа */
connected(X,Y):-edge(X,Y).
connected(X,Y):-edge(X,Z), connected(Z,Y).
```

Заметим, что в пролог-программах предложения, определяющие один и тот же предикат, принято располагать вместе. Эти предложения иногда называют *пролог-процедурой*. В общем случае пролог-программа состоит из фактов и набора процедур. Для обозримости программы процедуры можно выделять строками комментария.

Для сокращения числа предложений программы оба правила для предиката `connected` можно записать одним предложением, воспользовавшись знаком **;** , который в этом случае обозначает *логическую дизъюнкцию*:

```
connected(X,Y):- edge(X, Y) ;
                 edge(X, Z),connected(Z,Y).
```

Приведем примеры вопросов с предикатом `connected`:

```
?- connected(a,c).   => yes
?- connected(b,d).   => no
?- connected(a,V).
       % С какими вершинами связана вершина a?
                  => yes, V=b
               ;  => yes, V=d
               ;  => yes, V=e
               ;  => yes, V=c
               ;  => no  % других ответов нет
```

1.4 Термы и их сопоставление

До сих пор в пролог-предикатах записывались в качестве аргументов и обрабатывались простые данные – ***атомы***. Синтаксически атомы суть идентификаторы (имена), начальная буква которых должна быть строчной: `nick`, `a`, `b`… и т.п. К простым данным относятся также ***числа*** и ***переменные***. Кроме простых данных в языке Пролог есть составные данные – ***термы*** – см. Рис. 9.

Атомы и числа и считаются константами. Пролог обычно допускает как целые, так и вещественные числа: `3.14`, `1`, `-97` и т.п. (хотя необходимость в вещественных числах при символьной обработке невелика).

Переменные в Прологе синтаксически также являются идентификаторами, но в отличие от атомов записываются с заглавной буквы: `X`, `Y`, `Nick`, `Num`… и т.п. Обычно при записи переменных (как и атомов) допускается знак подчеркивания.

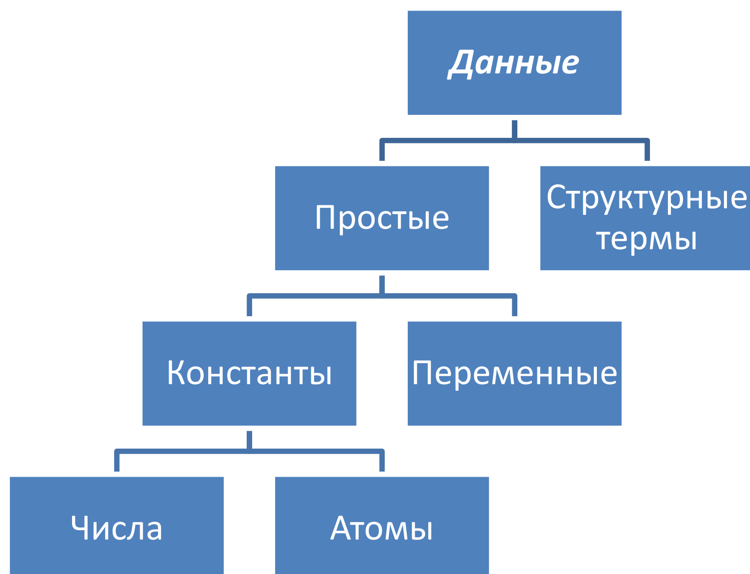

Рис. 9. Обрабатываемые данные языка Пролог

Кроме чисто синтаксического отличия в первой букве есть ещё одно важное отличие переменных от атомов: диапазон действия атомов – вся пролог-программа (т.к. они константы), в то время как для переменных область действия – одно предложение (и в разных предложениях переменная с одним и тем же именем обозначает разные объекты).

Среди переменных выделяется так называемая ***анонимная переменная***, записываемая одним знаком подчерка: _ . Её удобно использовать, когда не важно, какой именно объект ей соответствует при доказательстве, например:

```
has_child(X):- parent(X,_).
```

Ссылки на такие объекты не нужны, а значит, не нужны и их имена. Значения анонимных переменных не сохраняются и в ответах на вопросы не выводятся. Анонимная переменная может использоваться в одном пролог-предложении несколько раз, но при этом есть важное отличие от других переменных. Если любая другая переменная в рамках одного пролог-предложения обозначает один и тот же объект, то разные вхождения анонимной переменной в это предложение обозначают, вообще говоря, разные объекты (фактически, каждое ее вхождение – это новая переменная). К примеру, цель

```
connected(_,_).
```

достижима, если в графе есть хотя бы одна пара связанных (путём или ребром) вершин. проверить

Таким образом, применение анонимной переменной избавляет от необходимости придумывать имена объектам, на которые ссылаются один раз.

Сложные данные в языке Пролог представляются в виде ***структурных термов***. Запись терма начинается с указания имени функтора, за которым в скобках указываются его аргументы, например:

```
point(X,Y,Z)
family(Husband, Wife, Cildren)
```

Функтор, или функциональный символ определяется именем и арностью – количеством аргументов. Тем самым функторы с одинаковыми именами, но разной арности рассматриваются пролог-системой как разные функторы. Имёнами функторов могут быть атомы, а в качестве аргументов могут выступать переменные, числа, атомы, а также и структурные термы – это значит, что термы могут вкладываться друг в друга, причем на любую глубину. К примеру, терм вида

```
lecture(Subject, time(Week_Day,Pair_Number),
        Room, lector(Fname))
```

служит для записи информации о лекции, входящей в расписание студенческих занятий. Вот пример записи о конкретной лекции:

```
lecture(math_analysis, time(monday,3),
          609, lector(smirnov))
```

В этом терме указывается название предмета, время проведения лекции (день недели и порядковый номер пары), номер аудитории, фамилия лектора. В такой записи есть внутренние термы (подтермы) с внутренними функторами (их имена: `lecture, time, lector`). Внешний, не вложенный в другие термы функтор называется *главным функтором*, в данном примере это функтор `lecture`.

Заметим, что в частном случае терм может иметь один аргумент (функтор `lector`) и даже не иметь аргументов – в этом случае терм вырождается в атом.

По сути, структурный терм языка Пролог – это простой и гибкий способ группировки данных в единую структуру. Чтобы объединить разрозненные данные в единый объект, следует выбрать имя функтора, определить количество его аргументов, и записать нужные данные в качестве аргументов.

Структурные термы, как и простые данные (атомы, числа, переменные) могут быть аргументами пролог-предикатов, и тем самым участвовать в процедуре сопоставления, проводимой пролог-интерпретатором в процессе доказательства целей.

Строго говоря, понятие терма в Прологе (и в математической логике) трактуется расширенно: частными случаями терма являются атомы, числа, переменные (их можно считать вырожденными случаями терма). Чтобы отличить от них собственно термы, последние называют *структурными термами* или *структурами*. Заметим попутно, что синтаксически предикаты языка Пролог также являются термами, что позволяет применять к термам и предикатам общую процедуру сопоставления.

Определим понятие сопоставимости двух термов Пролога.

Два терма *S* и *T* ***сопоставимы***, если:

- они не содержат переменных и тождественно равны, например:
  `S=jane, T=jane` или
  `S=lector(jane,smith), T=lector(jane,smith)`
- переменные в термах *S* и *T* можно конкретизировать (приписав им значения) таким образом, что *S* и *T* становятся тождественно равными после соответствующей подстановки значений вместо переменных, например:
  `time(D1,N)` и `time(D2,3)` сопоставимы, т.к. существует конкретизация `D1=monday, D2=monday, N=3`.

На самом деле в данном случае возможно несколько разных конкретизаций переменных, приводящих к сопоставимости термов, к примеру, допустима такая конкретизация: `D1=friday, D2=friday, N=3`. Вообще для сопоставимости рассматриваемых термов, кроме `N=3` необходимо, чтобы `D1` и `D2` имели одно и то же значение, однако само это значение может быть произвольным.

Пролог-система в процессе сопоставления всегда в подобных случаях выбирает наиболее общую конкретизацию – т.е. ту, что ограничивает переменные в наименьшей степени, оставляя им наибольшую свободу для дальнейшего доказательства (в приведенном примере переменные `D1` и `D2` при сопоставлении становятся сцепленными). Обычно переменным по мере доказательства целей приписываются всё более и более конкретные значения.

Пролог-интерпретатор выполняет **сопоставление двух термов** *S* и *T* (в расширенном их понимании) **по следующим правилам**:

1. Если *S* и *T* – константы, то *S* и *T* сопоставимы только тогда, когда они тождественно равны.
2. Если *S* и *T* – неконкретизированные переменные, то они сопоставимы и эти переменные становятся *сцепленными* (как только одна из них в процессе доказательства конкретизируется некоторым значением, то и вторая переменная принимает то же значение).
3. Если *S* – переменная, а *T* – константа или структурный терм, не содержащий вхождений *S*, то они сопоставимы, и *S* получает в качестве значения терм *T*.
4. Если *T* – переменная, а *S* – константа или структурный терм, не содержащий вхождений *T*, то они сопоставимы, и *T* получает в качестве значения терм *S*.
5. Если *S* и *T* – структурные термы, то они сопоставимы, только если:
   - *S* и *T* имеют одинаковый главный функтор;
   - все аргументы этих функторов в термах *S* и *T* попарно сопоставимы.
6. Во всех остальных случаях сопоставление неуспешно.

К примеру, будет успешным сопоставление термов

S= `lecture(english,time(X,2),Y,Z)` и

T= `lecture(english,time(friday,2),71,lector(kim))`

и будут конктретизированы: `X=friday`, `Y=71`, `Z=lector(kim)`.

Важно, что сформулированные выше правила дополняют принципы доказательства целей пролог-системой, изложенные в разделе 1.2. Эти правила лежат в основе применяемой системой рекурсивной **процедуры сопоставления**.

Рассмотрим использование сопоставления термов на примере задачи из [2], касающейся представления геометрических объектов на плоскости. Точка на плоскости задаётся двумя своими координатами, отрезок – двумя точками, треугольник же можно задать тремя точками. Соответственно, для их представления можно использовать бинарные функторы `point`, `segment` и тернарный функтор `triangle`. Вот примеры представления конкретных геометрических объектов: `point(2,3)` `point(1,1)`

```
    segment(point(2,3),point(1,1))
triangle(point(2,3),point(1,1),point(5,2))
```

Введенные функторы позволяют определить унарные предикаты `vertical` и `horizontal` для распознавания вертикальности и горизонтальности заданных отрезков. Учитывая, что вертикальность означает совпадение абсцисс концов заданного отрезка, а горизонтальность – совпадение ординат, получаем пролог-программу:

```
 vertical(segment(point(X,_),point(X,_))).
horizontal(segment(point(_,Y),point(_,Y))).
```

Эта программа формально состоит из двух пролог-фактов, однако эти факты достаточно общие, поскольку их аргументами выступают термы с переменными. Приведём примеры вопросов с этими предикатами:

```
?-vertical(segment(point(2,3),point(1,1))). => no
?-vertical(segment(point(2,Z),point(1,1))).
                                        =>yes, Z=1
```

В первом вопросе проверяется вертикальность конкретного отрезка, а во втором – может ли быть вертикальным отрезок, у которого не задана ордината одной точки. Первый вопрос дает отрицательный ответ (т.к. ординаты концов отрезка не совпадают). Доказательство второго вопроса-цели успешно выполняется за один шаг (применением правила для `vertical`), в ходе которого включается процедура соспоставления терма-аргумента из правила и терма из доказываемой цели. Последовательно (попарно) сопоставляются аргументы главного функтора `segment`, что приводит к попарному сопоставлению аргументов внутреннего функтора `point`.

Следующий вопрос – это попытка найти вертикальный отрезок с заданным началом:

```
?-vertical(segment(point(2,3),P).
                           => yes,P=point(2,_17)
```

В результате успешного сопоставления определяются координаты конца искомого отрезка (`_17` – это переменная, сгенерированная пролог-системой).

В следующем составном вопросе запрашивается, существует ли отрезок, одновременно вертикальный и горизонтальный:

```
?-vertical(Seg),horizontal(Seg). => yes,
   Seg=segment(point(_23,_24),point(_23,_24))
```

Согласно установленным определениям предикатов вертикальности и горизонтальности, такой отрезок существует и представляет собой точку (т.е. отрезок, у которого совпадают начало и конец; переменные вида `_23` сгенерированы пролог-интерпретатором).

Заметим, что поскольку структурные термы могут быть изображены в виде деревьев, вершины которых помечаются именами функторов, атомами, числами, переменными, а дуги соответствуют аргументам этих функторов, то Пролог можно считать языком обработки деревьев.

1.5 Пролог *и логика предикатов*

Язык Пролог восходит к математической логике, предоставляя процедурную трактовку предикатов первого порядка [8, 9].

Синтаксис формул в языке Пролог существенно отличается от используемой в математической логике традиционной записи формул логики предикатов. В Прологе используется специальная клаузальная форма записи – запись в виде набора *клауз Хорна* [8, 10], в которых не более одного положительного дизъюнкта, а кванторы общности и существования опускаются, что облегчает чтение формул. При этом логические связки конъюнкция, дизъюнкция и импликация имеют тот же приоритет и кодируются соответственно знаками **,** **;** и **:-**.

В частности, пролог-правило вида `P:-Q,R,T.` представляет собой формулу логики предикатов $Q\&R\&T \rightarrow P$, равносильную формуле $\neg Q \vee \neg R \vee \neg T \vee P$ – дизъюнкту, в котором все слагаемые кроме одного отрицательны. Подобные формулы называются *хорновскими*. Таким образом, любое пролог-предложение является хорновским диъюнктом, а программирование на Прологе – программирование в хорновских дизъюнктах.

В языке Пролог при записи пролог-предложений дизъюнкция не является обязательной и используется лишь для сокращения записи предложений, к примеру, запись `P :- Q ; R.`
где `P`, `Q`, `R` – простые цели, эквивалентна последовательности двух предложений, в которых дизъюнкция задана неявно:

```
P :- Q.
P :- R.
```

Достижение цели в Прологе – это доказательство истинности утверждения-цели в предположении, что истинны все отношения, определённые в пролог-программе, или, иными словами, доказательство, что она логически следует из фактов и правил программы. При этом в ходе доказательства конкретизируются значения переменных, входящих в доказываемую цель, т.е. находятся конкретные объекты, входящие в отношения-предикаты.

Тем самым пролог-система выполняет доказательство теоремы на основе аксиом, при этом в качестве аксиом выступают факты и правила пролог-программы, а доказываемый вопрос-цель – это теорема. Примечательно, что доказательство проводится в обратном порядке: пролог-система начинает доказательство не с аксиом (известных фактов и правил), а наоборот – с целей, которые надо доказать, и, применяя правила, заменяет текущие цели новыми до тех пор, пока эти новые цели не окажутся фактами. Доказательство основано на известном методе резолюций [16] автоматического доказательства теорем, точнее, на более эффективной его разновидности – линейной, или SLD-резолюции для хорновских дизъюнктов [8]. При доказательстве пролог-программ проводится сопоставление термов, что соответствует (с некоторыми ограничениями) процедуре унификации двух термов в математической логике.

К числу основных логических связок исчисления предикатов, кроме конъюнкции и дизъюнкции, относится **логическое отрицание**, поэтому важен вопрос о выразимости отрицания в языке Пролог. В Прологе реализована, в виде *встроенного предиката* **`not`** **ограниченная форма отрицания**, которая не в точности соответствует отрицанию в математической логике.

Пролог-предикат **`not`** реализует отрицание как безуспешное выполнение. Конкретно, если `G` – некоторая цель (в общем случае, составная), то `not(G)` доказуемо (доказательство успешно завершается с ответом `yes`) тогда, когда цель `G` не может быть доказана, т.е. когда при её доказательстве возникает неуспех (ответ `no`). Если же доказательство цели `G` успешно, то предикат `not(G)` вырабатывает неуспех.

Вот пример предиката с `not`, определяющего самого старшего члена семьи («патриарха»): проверить

```
patriarch(X):-parent(X,_),not(parent(_,X)).
```

В большинстве реализаций языка Пролог требуется, чтобы к моменту доказательства цели G она не содержала неконкретизированных переменных (за исключением анонимных) – во избежание некорректностей, связанных с логической трактовкой значений переменных, полученных в результате доказательстве.

Непрямое доказательство истинности, выполняемое предикатом `not`, опирается на так называемое *предположение о замкнутости мира*, а именно: все, что в нём существует, либо указано в программе, либо может быть из неё выведено; если же для какого-то отношения это не так, то оно не истинно, и, следовательно, истинно его отрицание. Это позволяет понять кажущиеся противоречивыми ответы пролог-системы на вопросы:

```
?- not(woman(alice)).   => yes
?- not(man(alice)).     => yes
```

Здесь для доказательства взяты пролог-факты из раздела 1.2, устанавливающие пол для членов семьи, среди которых нет `alice`.

Приведём пример применения предиката `not` для определения неравенства двух пролог-объектов (такой предикат требовался в разделе 1.2 для уточнения предиката `sister`).

```
dif(X,Y):- not(X=Y).
```

Приведённый предикат `dif` опирается на явное обращение к процедуре сопоставления двух термов (аргументов предиката): `X=Y`, неуспех их сопоставления и означает их неравенство.

1.6 Особенности логических программ

Программирование на языке Пролог отличает ряд особенностей, не свойственных ни традиционным императивным языкам программирования (Паскаль, Си и др.), ни менее традиционным функциональным языкам (Лисп, Схема и др.). К этим особенностям относят недетерминированность и инвертируемость логических предикатов, а также различие декларативной и процедурной семантики пролог-программ.

Свойство инвертируемости, или *обратимости* (*invertibility*) означает, что один и тот же аргумент пролог-предиката может быть как входным (при одних вычислениях), так и выходным (при других вычислениях), что связано с декларативным пониманием предиката как отношения между объектами. В процедурах и функциях

большинства языков программирования фиксируется, какие параметры (аргументы) являются *входными* (*input*), т.е. служат для передачи исходных данных, а какие являются *выходными* (*output*), т.е. служат для возврата вычисленных данных. Напротив, пролог-предикаты обычно допускают многообразное использование своих аргументов, не фиксируя направления передачи их значений.

Инвертируемость демонстрируют многие предикаты, в том числе ранее рассмотренные `parent`, `ancestor`, `connected`. Зафиксированные направления передачи значений аргументов предиката (внутрь/вовне: *input/output*) иногда называют *образцом* или *прототипом* передачи (*flow pattern*); при записи прототипа используют буквы *i* и *o*, а также скобки. В частности, для предиката `connected` из раздела 1.3 его прототипы записываются как *(i, i)*, *(o, i)*, *(i, o)*, *(o, o)*. Вот примеры разных вычислений этого предиката:

```
прототип (i, i):   ?-connected(a,c).   => yes
прототип (i, o):   ?-connected(b,Y).   => yes, Y=c
прототип (o, i):   ?-connected(X,c).    => yes, X=b
                                      ; => yes, X=a
прототип (o, o):   ?-connected(X,Y). => yes, X=a, Y=b
                                   ; => yes, X=a, Y=d
                                   ; => yes, X=a, Y=e
                                   ; => yes, X=b, Y=c
                                   ; => yes, X=e, Y=d
                                   ; => yes, X=f, Y=g
                                   ; => yes, X=a, Y=c
                                   ; => yes, X=a, Y=d
```

Таким образом, инвертируемый предикат – это предикат, допускающий несколько прототипов (образцов передачи аргументов).

Еще одна особенность многих предикатов Пролога – недетерминированность, или множественность решений. Предикат является ***недетерминированным***, если доказательство цели с этим предикатом может дать более одного решения. Легко заметить, что недетерминированными являются ранее рассмотренные предикаты `parent`, `sister`, `ancestor`, `connected`. Точнее говорить о недерминированности (или же детерминированности) вычислений для определенных прототипов (образцов передачи) предиката. Например, для предиката `connected` недетерминированными являются образцы *(o, i)* , *(i, o)* и *(o, o)*.

Ещё одной отличительной особенностью логического программирования является то, что для пролог-предикатов и программ различают ***декларативную и процедурную семантику***.

Декларативный смысл программы касается только логических отношений, заданных в ней, и определяет, что должно быть её результатом (решением). Процедурный смысл определяет, ещё и как этот результат получается (т.е. как цели программы доказываются пролог-системой).

Поясним это на примере пролог-правила

```
P:-Q,R.
```

где `P`, `Q`, `R` – цели-предикаты. Декларативная интерпретация правила звучит так: `P` – истинно, если `Q` и `R` истинны (или: из `Q` и `R` следует `P`). Его процедурная интерпретация читается так: чтобы достичь цель `P`, необходимо сначала достичь цель `Q`, а затем достичь `R`. Тем самым процедурная семантика определяет не только логические связи между целями `P`, `Q` и `R`, но и порядок их обработки.

По этой причине не редки случаи, когда декларативная семантика некоторой пролог-программы может быть правильной, но при этом её процедурная семантика неверна, т.е. на основе такой программы нельзя получить правильный ответ на заданный вопрос-цель. Продемонстрируем это на примере предиката `ancestor`, рассмотренного в разделе 1.3 и включающего два правила. Второе правило рекурсивно, его тело состоит из конъюнкции двух целей. Ясно, что с точки зрения логики (декларативной семантики) порядок доказательства этих целей не важен (т.к. конъюнкция – коммутативная операция), и цели в теле правила можно поменять местами. Аналогично, не важен (с точки зрения логики) порядок самих предложений (поскольку соединяющая их дизъюнкция – коммутативная операция). Тем самым возможны ещё 3 варианта определения этого предиката, логически эквивалентные исходному. Ниже приводятся эти варианты, первым даётся определение из раздела 1.3:

```
 I.  ancestor(X, Z) :- parent(X, Z).
     ancestor(X, Z) :- parent(X, Y),ancestor(Y, Z).
 II. ancestor(X, Z) :- parent(X, Y),ancestor(Y, Z).
     ancestor(X, Z) :- parent(X, Z).
III. ancestor(X, Z) :- parent(X, Z).
     ancestor(X, Z) :- ancestor(Y, Z),parent(X, Y).
```

```
IV. ancestor(X, Z) :- ancestor(Y, Z),parent(X, Y).
    ancestor(X, Z) :- parent(X, Z).
```

Хотя все приведённые варианты предиката имеют одинаковую декларативную семантику (она не зависит от порядка целей в предложениях и порядка самих предложений), их процедурная семантика различна.

Предикат варианта IV всегда зацикливается, не выдавая решения (поскольку леворекурсивно первое правило, всегда пробующееся в ходе доказательства первым). Вариант III зацикливается на некоторых видах вопросов, например, на вопросе `?-ancestor(ann,jane)`, но иногда может найти верный ответ, как для вопроса `?-ancestor(mary,jane)`. Таким образом, эти оба варианта имеют неверную процедурную семантику.

Варианты I и II предиката `ancestor` являются правильными как логически, так и процедурно, в то же время вариант II довольно неэффективен, т.к. осуществляет значительно больше возвратов при поиске (из-за того, что рекурсивное правило стоит первым). Хороший эвристический принцип составления пролог-предикатов: помещать рекурсивные правила после рекурсивных, а факты – перед правилами.

1.7 Списки и их обработка

Список, или упорядоченная последовательность элементов – одна из широко используемых структур в программировании. Поскольку список можно рассматривать как вырожденный случай дерева, термы языка Пролог позволяют представлять и обрабатывать в пролог-программах списки из произвольного числа разных элементов (атомов, чисел, термов).

Структура списка определяется рекурсивно: список является либо *пустым* (без элементов), и обозначается **[ ]**, либо состоит из двух компонент: первого элемента, называемого *головой списка* (*Head*), и *остатка-хвоста* (*Rest*), являющегося списком. Список-хвост представляет собой исходный список без первого его элемента, т.е. это всегда список, возможно, пустой. Пустой список же не имеет ни головы, ни хвоста.

Списки в Прологе реализуются с помощью встроенного бинарного функтора «точка», соединяющего голову и хвост списка: **.***(Head, Rest)*. Любой список представляется как терм с вложенными функторами «точка» (`[ ]` – пустой список), например:

```
    .(1, .(3, .(5, .(7, .(9, [])))))
    .(the, .(cat, .(likes, .(milk, []))))
.(point(2,3), .(point(5,17), .(point(1,4), [])))
```

Подобная запись громоздка, поэтому для списков в пролог-программах используется более удобная скобочная форма записи, при которой элементы списка разделяются запятыми и заключаются в скобки. Вот скобочная запись приведенных выше списков:

`[1,3,5,7,9]` – список первых пяти нечётных чисел;

`[the,cat,likes,milk]` – список английских слов;

`[point(2,3),point(5,7),point(1,4)]` – список трёх точек.

Отметим, что в качестве элементов списка могут браться любые термы, в том числе атомы, числа, переменные, списки. Также возможна вложенность списков:

```
[a,point(3,5),[b,[c]],children([tim,pam]),d]
```

Так как операция расщепления на голову и хвост списка часто используется при обработке списков, в языке Пролог допускается ещё одно удобное синтаксическое средство, называемое иногда *конструктором* – это вертикальная черта, позволяющая соединить голову и хвост списка и записать их в квадратных скобках:

```
.(Head, Rest) ≡ [Head | Rest]
```

Допускается одновременное использование вертикальной черты со скобочной формой записи списка, к примеру:

```
[the, Y, likes | X]
```

– в этом списке указаны первые три элемента (второй элемент `Y` неопределен), а переменная `X` после вертикальной черты обозначает остаток списка, т.е. список-хвост; подчеркнем, что значением `X` не может быть атом или число, а только список.

В целом, запись с вертикальной чертой удобна для представления неопределённых и частично определённых списков. Проиллюстрируем на примере связь рассмотренных средств записи списков:

```
[a,b,c] ≡ [a|[b,c]] ≡ [a,b|[c]] ≡ [a,b,c|[]]
```

При работе со списками часто используется предикат проверки, является ли некоторый объект элементом списка, а также предикат соединения (конкатенации) двух списков.

Приведём примеры вопросов с предикатом **member(E,L)**, реализующим *отношение принадлежности* элемента списку. Предикат истинен, если пролог-объект E является элементом списка L на верхнем его уровне:

```
?-member(c,[a,b,c]). => yes
?-member(c,[a,b,[c]]). => no
```

Этот предикат можно определить двумя предложениями: E входит в список L, либо когда E является головой списка, либо когда E принадлежит хвосту списка L. Вот соответствующие факт и правило:

```
member(E,[E|R]).                    %Правило (1)
member(E,[H|R]):- member(E,R).      %Правило (2)
```

Построенный предикат рекурсивен и инвертируем. Кроме прототипа *(i, i)* он допускает вариант использования *(o ,i)*, когда задан только второй аргумент, и результат вычислений – объект, который входит в заданный список как элемент верхнего уровня. В этом прототипе предикат недетерминированный: решениями, которые можно получить в режиме бектрекинга, являются всевозможные элементы верхнего уровня исходного списка:

```
?-member(E,[a,b,c]).     => yes, E = a
                       ; => yes, E = b
                       ; => yes, E = c
                       ; => no
```

Ход доказательства этой цели показан на Рис. 10. Поскольку предложения предиката member применяются при доказательстве несколько раз, при каждом применении входящие в них переменные переименовываются путем приписывания им верхнего индекса, равного порядковому номеру применения предложений предиката.

Оказывается, рассмотренный предикат member работает и в прототипе *(i,o)*, когда задан лишь первый его аргумент; и в этом случае предикат также недетерминирован, например:

```
?-member(b,Y)     => yes, Y = [b|_596]
                ; => yes, Y = [_595, b|_599]
                ; => yes, Y = [_595,_598, b|_602]
             ; => yes, Y = [_595,_598,_601, b|_602]
                   …      …
```

В режиме бектрекинга получаем всевозможные частично конкретизированные списки, в которых атом b является первым,

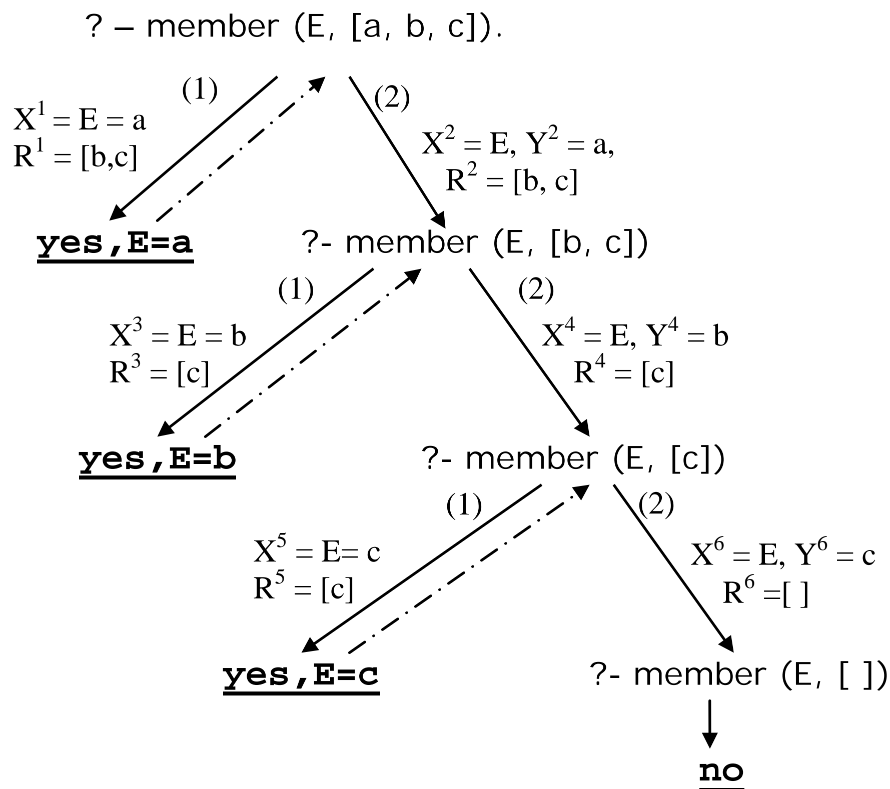

Рис. 10. Дерево доказательства цели с предикатом `member`

вторым, третьим и т.д. элементом (сгенерированные пролог-системой переменные вида `_595` обозначают произвольные объекты).

Предикат `member` обычно является встроенным, как и предикат **`append(L1,L2,L3)`**, реализующий отношение *конкатенации* (*соединения*) двух списков, а именно: список `L3` состоит из элементов списка `L1`, за которыми следуют элементы списка `L2`, например:

```
?- append([a,b,c],[d,e],[a,b,c,d,e]).    => yes
?- append([a,b,c],[d,c],[a,b,c,a,d,c]). => no
```

Это отношение определяется отдельно для двух случаев:

- если `L1` пуст, то `L3` совпадает с `L2` (т.е. они представляют один и тот же список `L`);
- если же `L1` не пуст, то он имеет вид: `[X|R1]`, и тогда результирующий список `L3` есть список с головой `X` и хвостом `R3` – результатом соединения хвоста `R1` первого списка и списка `L2`; схематическое изображение соединения показано на Рис. 11.

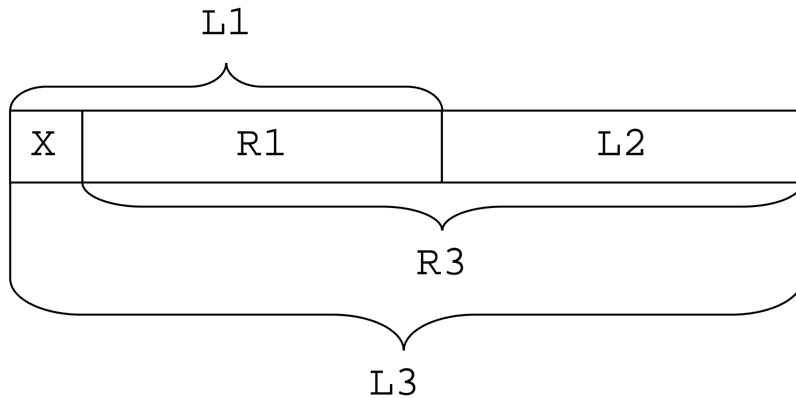

Рис. 11. Схема конкатенации (соединения) двух списков

Соответственно, получаем два предложения: пролог-факт и рекурсивное пролог-правило:

```
    append([],L,L).          %Правило (1)
    append([X|R1],L2,[X|R3]) :- append(R1,L2,R3).
                             %Правило (2)
```

Полученный предикат инвертируем и работает в прототипах:

(i, i, i) – проверка, является ли `L3` конкатенацией списков `L1` и `L2`;

(i, i, o) – соединение (конкатенация) двух заданных списков `L1` и `L2`, дерево доказательство такой цели показано на Рис.12;

(o, o, i) – расщепление заданного списка `L3` на два списка `L1` и `L2`;

(o, i, i) и *(i, o, i)* – отщепление от заданного списка `L3` справа или слева элементов другого списка: `L1` или `L2`. Например:

```
?-append([a,b,c],[l,k],Z). =>  yes, Z=[a,b,c,l,k]
```

Отметим, что в прототипе *(o, o, i)* предикат недетерминирован:

```
?-append(X,Y,[a,b,c,d]).
            => yes, X=[], Y=[a,b,c,d]
         ;  => yes, X=[a], Y=[b,c,d]
         ;  => yes, X=[a,b], Y=[c,d]
         ;  => yes, X=[a,b,c], Y=[d]
         ;  => yes, X=[a,b,c,d], Y=[]
         ;  => no
```

```
?- append([a, b], [c, d, e], Z).
        (2) | X = a, R1 = [b], L2 = [c, d, e],
            | Z = [X | R3]=[a | R3]
            v
?- append([b], [c, d, e], R3).
        (2) | X' = b, R1' = [], L2' = [c, d, e],
            | R3 = [X | R3']=[b | R3']
            v
?- append([], [c, d, e], R3').
        (1) | L = [c, d, e] = R3'
            v
yes, Z=[a,b,c,d,e]
```

Сборка результата:
Z = [a | R3]
= [a, b | R3']
= [a, b, c, d, e]
[a | R3] =
[a | [b | R3']] =
[a | [b | [c, d, e]]]

Рис. 12. Дерево доказательства цели с `append` в прототипе *(i,i,o)*

Рассмотренный предикат `append` для конкатенации списков реализует довольно мощное преобразование, и его можно использовать, к примеру, для определения предиката `member`:

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

Это описание более декларативно, чем рассмотренное выше рекурсивное определение `member`, однако процедурный смысл обработки списка не столь ясен. В качестве другого примера приведём использование `append` в прототипе *(o, o, i)* для поиска в заданном списке `L` двух подряд стоящих одинаковых элементов:

```
?-append(_,[X,X|Rest],L).
```

В результате достижения этой цели будут найдены значения `X` и `Rest`. Подобным образом можно искать в списке некоторую комбинацию подряд стоящих элементов.

С помощью `append` можно также реализовать предикат `last` от двух аргументов: предикат истинен, если последний элемент заданного списка оказался равным первому аргументу предиката:

```
last(X,L):-append(_,[X],L).
```

На Рис. 13 показан пример дерева доказательства цели с предикатом `append` в прототипе *(o, o, i)* .

```
(1)   append([ ], L, L).
(2)   append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

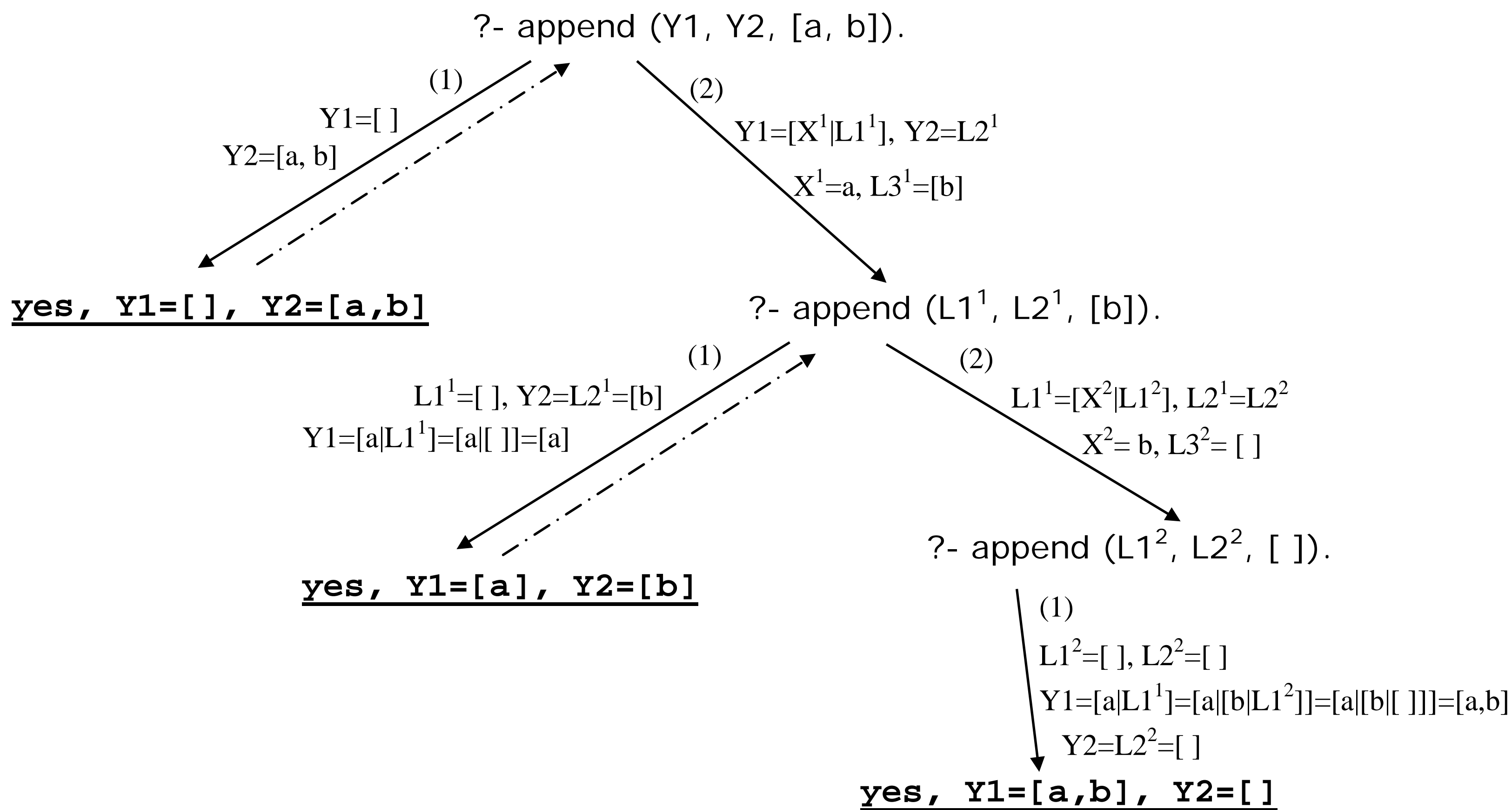

Рис. 13. Дерево доказательства цели с предикатом `append` в прототипе *(o, o, i)*

1.8 Отсечение и бектрекинг

В ходе вычислений пролог-программ точки бектрекинга создаются и обрабатываются пролог-интерпретатором автоматически, поэтому при программировании некоторых отношений полезны средства управления механизмом бектрекинга. К этим средствам относятся два встроенных пролог-предиката без аргументов: предикаты **fail** и **cut**.

Предикат **fail** **всегда** ложен, т.е. вырабатывает *неуспех* и тем самым инициирует бектрекинг, в ходе которого пролог-интерпретатор выполняет возврат к последней (по времени возникновения) точке бектрекинга. Важно, что при этом автоматически восстанавливается вся операционная обстановка этой точки вычислений (включая значения переменных), после чего возобновляется доказательство текущей цели. Поскольку предикат **fail** всегда инициирует бектрекинг, то стоящие после него (в теле пролог-правила или в пролог-вопросе) цели никогда не будут доказываться, а потому этот предикат следует помещать последним в списке доказываемых целей. В качестве примера приведем предикат list, истинный, если его аргумент – непустой список:

```
list([]):-fail.
list([_|_]). проверить
```

В случае, когда аргументом предиката list является пустой список, вырабатывается неуспех, возвращающий вычисления ко второму предложению предиката, которое реализует нужную проверку.

Заметим, что поскольку предикат **fail** всегда ложен, с тем же результатом вместо него можно использовать предикат с любым другим именем, для которого в пролог-программе отсутствуют определяющие его предложения.

Встроенный предикат **cut**, называемый ***отсечением*** и обозначаемый знаком восклицания **!**, всегда истинен. Его основное назначение – уничтожение по определённому принципу уже возникших точек бектрекинга, тем самым частично предотвращая бектрекинг, и в этом смысле его действие противоположно действию **fail**. В целом, предикат **!** служит для сокращения дерева доказательства путём отсечения некоторых его ветвей.

Рассмотрим действие отсечения **!** в пролог-предложении общего вида: $P:- R_1,...,R_k,!,R_{k+1},...,R_n.$ где $0 \le k \le n$.

Пусть это предложение используется при доказательстве цели G; само предложение относится к пролог-процедуре P (напомним, что под пролог-процедурой P понимается набор из всех предложений, в левой части которых стоит предикат P). В результате отсечения происходит уничтожение всех последних (по времени возникновения) точек бектрекинга, возникших с момента входа в процедуру P , т.е. начиная с момента поиска предложения этой процедуры, применимого для доказательства.

Таким образом, при выполнении отсечения, во-первых, отбрасываются все точки бектрекинга, которые возникли при доказательстве целей $R_1,...,R_k$, предшествующих отсечению (и тем самым отбрасываются все альтернативные решения конъюнкции целей $R_1,...,R_k$). Во-вторых, уничтожается также точка бектрекинга, связанная с самой процедурой P и соответствующая другим применимым предложениям этой процедуры, и в результате будут отброшены возможные альтернативы доказательства цели G.

В то же время выполненное отсечение не влияет на цели $R_{k+1},...,R_n$, расположенные правее отсечения **!** в теле рассматриваемого предложения. Возникающие при их доказательстве точки бектрекинга остаются, и тем самым, эти цели могут порождать более одного решения.

Если же доказательство конъюнкции $R_{k+1},...,R_n$ неуспешно, т.е. возник неуспех (*fail*), и процесс возврата, исчерпав все альтернативы в появившихся в ходе этого доказательства точках бектрекинга, достиг точки отсечения, то далее он распространяется до последней по времени точки бектрекинга, возникшей ещё до входа в процедуру P (и доказательство конъюнкции целей $R_1,...,R_k$ до отсечения не пересматривается).

В качестве примера рассмотрим введение отсечения в предикат member (определенный в разделе 1.8):

```
member_c(E,[X|_]):- !.
member_c(E,[_|R]):- member_cut(E,R).
```

В общем случае искомый элемент E может входить в список (второй аргумент предиката), несколько раз. Вычисление member_c в прототипе *(o, i)* приведёт к поиску только первого вхождения E в заданный список, т.к. сразу после применения первого правила предиката member_c будет отсечена точка бектрекинга текущей цели, и с ней – возможность применения второго правила. Поэтому

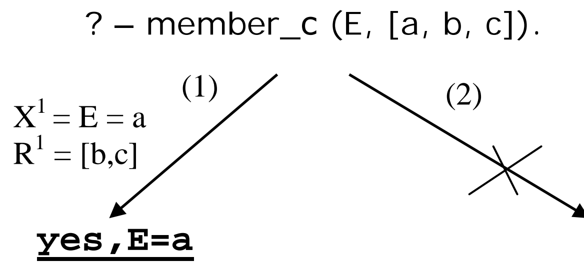

Рис. 14. Дерево доказательства цели с предикатом member_c

результат вычислений будет отличаться от результата исходного предиката member без отсечения:

```
?-member_c(E,[a,b,c]).     => yes, E = a
                           ; => no
```

Такой предикат целесообразен в случаях, когда требуется найти лишь одно решение, если оно вообще существует. На Рис. 14 показана отсеченная ветвь дерева доказательства.

Отсечение – мощное средство управления бектрекингом и, в частности, с помощью него (вместо применения предиката not) может быть реализовано отношение dif неравенства двух термов:

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

Проверка равенства двух термов-аргументов производится при попытке интерпретатора применить первое предложение-правило предиката dif, точнее при обработке заголовка этого пролог-правила встроенной процедурой сопоставления. В случае успеха сопоставления вырабатывается неуспех (fail), но перед этим происходит отсечение точки бектрекинга, связанной с этими двумя предложениями, и как следствие, второе предложение (которое применимо всегда) применяться уже не будет. В случае же неуспеха сопоставления двух термов срабатывает как раз второе правило предиката, возвращающее ответ yes.

1.9 Процедура доказательства целей

Декларативная семантика доказываемой цели $G = G_1, G_2 \ldots G_n$ определяет, является ли это целевое утверждение истинным, исходя из заданной пролог-программы *PP*, и если оно истинно, то при каких значениях входящих в него переменных. Цель *G* истинна, если истинны (доказуемы) все её подцели G_i при одинаковых конкретизациях входящих в них переменных.

Процедурная семантика цели *G* есть, по сути, процедура доказательства (достижения) целей $G_1, G_2 \ldots G_n$ в контексте данной программы *PP*, зависящая от порядка G_i и предложений в программе.

Уточним выполняемые пролог-интерпретатором основные шаги процедуры доказательства составной цели $G = G_1, G_2, \ldots G_n$ при заданной пролог-программе *PP*.

- Вход процедуры: список целевых утверждений $G_1, G_2, \ldots G_n$
- Выход: ответ `yes/no` , а в случае ответа `yes` выводится также найденные в ходе доказательства конкретизации переменных доказанной цели *G*.

Процедура *Prove* ($G_1, G_2, \ldots G_n$).

Шаг 1. Если `n=0` (т.е. список целей пуст), то доказательство завершено, выход из процедуры с ответом `yes`.

Шаг 2. Если $n \neq 0$, то попытка доказать цель G_1, для этого:

поиск *применимого предложения* пролог-программы *PP* путём последовательного, от начала к концу программы, просмотра всех её предложений. Предложение *C* применимо, либо если это факт вида `A.` , и он сопоставим с доказываемой целью G_1 , либо же это правило вида $A{:}\text{-}B_1,\ldots,B_m.$ и его заголовок-предикат `A` сопоставим с доказываемой целью G_1.

Если применимых предложений в программе нет, то *неуспех* доказательства, выход из процедуры с ответом `no`.

Шаг 3. Если применимые предложения есть, то для цели G_1 устанавливается *точка бектрекинга*, её назначение – запомнить выбранное для применения предложение *C* и контекст вычислений.

Шаг 4. Выбирается первое по порядку применимое предложение, либо же – в случае возврата при бектрекинге на этот шаг процедуры – следующее по порядку применимое предложение.

Если применимых предложений больше нет, то уничтожение точки бектрекинга, неуспех доказательства, выход из процедуры с ответом `no`.

Шаг 5. Для выбранного предложения *C*:

Производится переименование входящих в него переменных так, чтобы получить эквивалентный его вариант *C'* , в котором нет переменных, встречающихся в целях $G_1, G_2 \dots G_n$.

Предложение *C'* имеет вид $A'\texttt{:-}B_1',\dots,B_m'$**.** (пролог-правило) или `A'.` (пролог-факт).

Выполняется *сопоставление* терма-цели G_1 с термом *A '* , в ходе которого определяется *конкретизация S* входящих в термы переменных.

Шаг 6. Применяется предложение *C'* :

Если *C'* – пролог-правило $A'\texttt{:-}B_1',\dots, B_m'$, то в исходном списке целей $G_1, G_2, \dots G_n$ первая цель G_1 заменяется на тело правила *C'*, в котором выполнена *подстановка*: конкретизированные на предыдущем шаге переменные заменяются своими значениями; в остальных целях $G_2, \dots G_n$ производится такая же подстановка значений вместо переменных, в соответствии с конкретизацией *S*.
В итоге получается список целей $B_1'',\dots B_m'', G_2',\dots G_n'$.

Если *C'*– пролог-факт `A'`, то первая цель G_1 исходного списка целей $G_1, G_2 \dots,G_n$ доказана, а в остальных целях $G_2,\dots,G_n$ выполняется *подстановка*: конкретизированные на предыдущем шаге переменные заменяются своими значениями, в соответствии с конкретизацией *S*) .
В итоге получается список целей $G_2',\dots, G_n'$.

Шаг 7. Процедура доказательства рекурсивно вызывается для полученного на шаге 6 списка целей:

Prove $(B_1'',\dots B_m'', G_2',\dots G_n')$ или ***Prove*** $(G_2',\dots, G_n')$.

Шаг 8. Если результат рекурсивного вызова – `yes`, то процедура доказательства для входного списка целей $G_1,G_2,\dots,G_n$ успешно завершается с ответом `yes`, и объединение конкретизаций

переменных, найденных в ходе последовательного доказательства всех возникающих целей, есть *результирующая конкретизация переменных* входной цели $G = G_1 , G_2, \ldots, G_n$.

Шаг 9. В ином случае, если результат рекурсивного вызова – неуспех, то пролог-система производит *бектрекинг*: аннулирует результаты шагов 5-7 (включая найденные конкретизации переменных) и выполняет возврат к шагу 4 (после чего начинает с этого шага новую попытку доказательства, уже с помощью следующего применимого предложения программы).

Сделаем несколько важных замечаний к описанной рекурсивной процедуре доказательства целей.

На *Шаге 2* при поиске применимого предложения пролог-интерпретатор использует процедуру сопоставления термов, поскольку синтаксически термы и предикаты имеют одинаковую структуру.

На *Шаге 6* при применении пролог-предложений программы, являющихся правилами, в общем случае список подлежащих доказательству целей увеличивается. Сокращение этого списка происходит, когда применяемое предложение – пролог-факт. В случае же, когда пролог-факт применяется для доказательства единственной цели списка ($G = G_1$), список подлежащих доказательству целей становится пустым, что завершает при последующем рекурсивном вызове всю процедуру доказательства.

Инициируемый на *Шаге 9* бектрекинг гарантирует систематическую проверку пролог-интерпретатором всех возможных альтернативных путей доказательства – до тех пор, пока не будет найден путь, ведущий к успешному доказательству исходной цели, или же не окажется, что все пути приводят к неуспеху (ответу `no`). Фактически Пролог реализует определенную стратегию обхода дерева доказательства – *Depth_First_Search*, или поиск вглубь с возвратами [2, 3, 11].

Бектрекинг, инициируемый пользователем в ходе диалога с пролог-системой (путём ввода с клавиатуры символа `;`) фактически означает возврат в последнюю по времени возникновения, но ещё не уничтоженную точку бектрекинга для продолжения доказательства и поиска нового решения (см. *Шаг 4*).

2 Методы и средства программирования

В данном разделе рассматриваются типичные приёмы программирования на языке Пролог, а также описываются встроенные предикаты, расширяющие возможности языка и поэтому полезные при решении практических задач. В их число входят предикаты для работы с базой данных, предикаты ввода/вывода и др. Поскольку эти предикаты не имеют логической интерпретации, они называются *металогическими*. Отдельные детали работы этих предикатов могут зависеть от конкретной реализации Пролога.

2.1 Арифметические вычисления

Хотя в основном язык Пролог ориентирован на нечисловые задачи, тем не менее в нём возможны арифметические вычисления. Язык допускает выполнение всех основных арифметических операций над числами (числа – частный вид термов), однако в записи арифметических выражений и их вычислении есть особенности.

Все знаки арифметических операций: `+`, `-`, `*`, `/` , `**` (возведение в степень) и др. в Прологе являются встроенными функторами, и арифметическое выражение – это структурный терм, представленный в принятой для термов *префиксной форме* записи, например: `+(3,*(4,5))`. Для удобства в языке допускается привычная *инфиксная форма* записи арифметических выражений, и разрешается даже смешивать обе эти формы при записи одного и того же арифметического выражения. К примеру, приведённое выше арифметическое выражение можно записать в виде следующих *арифметических термов*:

`3+4*5` – инфиксная форма, общепринятая для арифметических выражений;

`+(3,*(4,5))` – запись в префиксной форме;

`3+ *(4,5)` и `+(3,4*5)` – смешанные формы.

Важно, что арифметические функторы сами по себе не вызывают выполнения соответствующих арифметических операций, и запись арифметических выражений ещё не предполагает их автоматическое вычисление. Арифметические выражения являются в первую очередь термами-структурами, которые обрабатываются

пролог-интерпретатором по установленным правилам сопоставления термов. Поэтому если в пролог-программе есть факт `expr(3+4*5).` то в результате следующих вопросов будет получено:

```
?-expr(3+4*5). => yes
?-expr(+(3,*(4,5))). => yes
?-expr(23).     => no
?-expr(4*5+3). => no
?-expr(X+Y).    => yes, X=3,Y=4*5
```

Если же при решении некоторой задачи возникает необходимость вычисления арифметического терма, то для этого следует использовать встроенный двуместный **оператор `is`**. Этот оператор имеет вид `E1 is E2`,
где `E2` – арифметическое выражение, а `E1` – произвольное выражение. Оператор записывается как некоторая цель пролог-программы, и её доказательство инициирует вычисление арифметического выражения `E2`. Более точно, доказательство цели-оператора `is` происходит следующим образом:

1. Вычисление выражения `E2` в соответствии с правилами арифметики; `E2` не должно содержать неконкретизированных переменных (т.е. должны быть известны значения всех переменных в `E2`), в ином случае пролог-интерпретатор прерывает работу с сообщением об ошибке;
2. Сопоставление (по правилам Пролога) выражения `E1` и вычисленного значения выражения `E2`; результат сопоставления есть результат выполнения оператора `is`.

На этапе сопоставления возможны такие случаи:

- в качестве `E1` выступает число или другая константа, результат сопоставления констант зависит от их тождественности;
- `E1` – переменная без значения, сопоставление успешно и `E1` конкретизируется вычисленным значением арифметического выражения (это основной случай использования оператора **`is`**);
- `E1` – переменная, конкретизированная некоторым числовым значением, сопоставление успешно, если это значение тождественно значению арифметического выражения;
- в качестве `E1` выступает структурный терм (в том числе – арифметическое выражение), сопоставление неуспешно.

Приведём примеры целей с оператором `is`:

```
?-3 is 1+2. => yes
?-4 is 1+2. => no
?-N is 1+2. => yes, N=3.
?-1+2 is 1+2. => no
?-is(M, 3+4). => M=7.
```

Другой способ инициировать вычисление арифметических выражений – использовать их в качестве операндов предикатов, которые выполняют сравнение значений двух арифметических выражений. Эти предикаты записываются, как правило, в инфиксной форме: `E1` *предикат* `E2`.

Пролог обычно включает следующие **встроенные предикаты сравнения**:

`=:=` (проверка на равенство),

`=\=` (проверка на неравенство),

`<` (проверка на меньше), **`>`** (проверка на больше),

`=<` (проверка на не больше), **`>=`** (проверка на не меньше).

Эти предикаты вычисляют оба своих аргумента-выражения `E1` и `E2`, а затем сравнивают их значения. Все входящие в эти выражения переменные должны быть конкретизированы числовыми значениями, в ином случае пролог-интерпретатор прерывает работу с сообщением об ошибке. Ошибка возникает также, если `E1` и `E2` не являются арифметическими выражениями.

Приведём пример использования этих предикатов:

```
?-5*4+3 =:= 3+4*5.  => yes
```

Отметим, что проверка на равенство двух числовых значений может быть выполнена как с помощью предиката **`=:=`** , так и оператора `is`, к примеру:

```
?- 23 =:= 23  => yes
?-17 is 17    => yes
```

Сравнение чисел на равенство может реализовать также **встроенный предикат `=`** , однако его семантика существенно отлична от предиката **`=:=`**.

Предикат `=` обычно записывается в инфиксной форме и является, по сути, явным обращением к процедуре сопоставления, реализуемой пролог-интерпретатором, т.е. сравнение `E1 = E2`

успешно, если термы `E1` и `E2` сопоставимы. Поэтому, например:

```
?- 1+2=2+1   => no
```

т.к. термы `+(1,2)` и `+(2,1)` несопоставимы, а их вычисление не производится. В то же время:

```
?- N is 1+2. => yes, N=3
?- N = 3. => yes, N=3
?- N = 3, 4 = N+1. => no
```

В качестве ещё одного примера применения оператора `is` рассмотрим определение предиката `length`: он истинен, если его второй аргумент-число есть количество элементов (на верхнем уровне) списка, являющегося его первым аргументом:

```
length([],0). % в пустом списке 0 элементов
length([X|T],N):-length(T,N1), N is N1+1.
        /* длина непустого списка на 1 больше
        длины его хвоста */
```

Предикат допускает использование в двух прототипах: *(i, i)* и *(i, o)* :

```
?- length([a,b,c,d,e],5)  => yes
?- length([a,b,c,d,e],N)  => yes, N=5
```

Однако если в определении оператор `is` заменить на предикат = :

```
length1([],0).
length1([X|T],N):-length1(T,N1),N = N1+1.
```

то в прототипах *(i, i)* и *(i, o)* будут получаться другие результаты:

```
?- length1([a,b,c,d,e],5)  => no
?- length1([a,b,c,d,e],N)
                => yes, N=0+1+1+1+1+1
```

Ниже приведены еще три предиката, использующих `is`: `slength` – для вычисления длины отрезка; `eq_side_triangle` – для определения, является ли треугольник, образованный тремя точками, равносторонним; `romb` – для определения, является ли четырёхугольник, заданный четырьмя точками, ромбом:

```
slength(segment(point(X1,Y1), point(X2,Y2)), Z) :-
        Z is sqrt((X1–X2)**2 +(Y1–Y2)**2).

eq_side_triangle(P1,P2,P3)) :-
            slength(segment(P1,P2),Z),
            slength(segment(P1,P3),Z)),
            slength(segment(P2,P3),Z).
```

```
romb(P1, P2, P3, P4):-
            slength(segment(P1,P2),L),
            slength(segment(P2,P3), L),
            slength(segment(P3,P3), L),
            slength(segment(P4,P1), L).
```

В этих определениях используются функторы `segment` и `point`, введенные в разделе 1.4, а также встроенная арифметическая функция `sqrt` извлечения квадратного корня.

2.2 Использование отсечения

Чисто декларативного программирования в терминах целей недостаточно при решении многих задач – для построения корректных и эффективных пролог-программ требуется учитывать их процедурную семантику, которая усложняется при использовании предикатов `fail` и `cut (!)`.

Отсечение **`cut (!)`** – достаточно мощное средство управления бектрекингом: этот предикат служит для уничтожения точек бектрекинга и тем самым – отсечения ветвей дерева доказательства целей, что может привести к потери части решений, т.е. к изменению декларативного значения логической программы.

По семантике отсечения подразделяются на так называемые зелёные и красные [17].

Зелёные отсечения не изменяют множество возможных решений логической программы, т.е. её декларативное значение. Это означает, что уничтожаются лишние точки бектрекинга и отсекаются лишь ненужные ветви дерева доказательства целей – в результате вычисления завершаются быстрее, а также экономится испольуемая пролог-интерпретатором память (вся связанная с точками бектрекинга информация запоминается в стеке).

Если логическая программа (предикат) детерминирована (у неё всего одно решение), то любые добавляемые в неё отсечения являются зелёными, при условии их корректности (т.е. если они не отсекают это единственное решение) – примером может служить предикат для определения минимума двух чисел:

```
min(X,Y,X):- X<=Y,!.
min(X,Y,Y):- X>Y.
```

Действительно, в случае истинности арифметического условия в первом предложении, оказывается невозможным выполнение условия во втором предложении, и поэтому эту альтернативу (и точку бектрекинга, связанную с процедурой `min`) можно отсечь. Приведенный пример – частный случай введения отсечения в пролог-процедуре вида

```
A:- B,C.          ⇒   A:- B,!,C.
A:- not(B),D.     ⇒   A:- D.
```

где `A`, `B`, `C`, `D` – некоторые цели. В исходном варианте процедуры присутствует неэффективное дублирование цели `B`, ведь для успешного доказательства цели `A` может потребоваться две попытки доказательства `B` (в случае неуспеха доказательства цели `C` при применении первого правила и затем при бектрекинге и применении второго предложения), что существенно в случае сложности цели `B`.

В отличие от зелёных, *красные отсечения* изменяют декларативное значение программы. Они обычно применяются, когда в недетерминированной программе необходимо по каким-либо причинам отбросить часть решений. Примером красного отсечения является отсечения в предикате `member_c` (см. раздел 1.8):

```
member_c(E,[E|_]):- !.
member_c(E,[_|R]):- member_c(E,R).
```

В этом примере происходит поиск только первого вхождения `E` в список, являющийся вторым аргументом предиката, и тем самым в прототипе *(o, i)* будет найдено только одно решение (выдан только первый элемент списка). Однако применяемое отсечение имеет смысл, если заранее известно, что предикат будет использоваться только в прототипе *(i, i)*.

Приведённые выше примеры предикатов демонстрируют отсечения, применяемые после проверки условий (предикат `min`), и отсечения после сопоставления (предикат `member_c`).

Заметим, что устранение ненужной недетерминированности может осуществляться также путём введения явных условий:

```
member_d(E,[E|_]).
member_d(E,[H|R]):- dif(E,H), member_d(E,R).
```

Вариант member_d в прототипе *(o, i)* недетерминирован, но в отличие от исходного member (без отсечения и без dif – см. раздел 1.7) он выдаст только различные элементы списка:

```
?- member(E,[a,b,a,c]).   => yes, E=a
                        ; => yes, E=b
                        ; => yes, E=a
                        ; => yes, E=c
                        ; => no
?- member_d(E,[a,b,a,c]).   => yes, E=a
                          ; => yes, E=b
                          ; => yes, E=c
                          ; => no
```

В целом, предикат отсечения можно считать слабой формой логического отрицания. Так, циклы вида *while <условие> do B* можно запрограммировать в Прологе на базе рекурсии и отсечения:

```
while:- <условие>,B, while.
while:-!.
```

Важно однако, что введение отсечения нарушает модульность пролог-программы, т.е. становится принципиальным порядок предложений в программе, а кроме того, отсечение очень часто ведёт к потере инвертируемости предиката. Таким образом, программы с отсечениями менее гибки и менее декларативны, чем их аналоги без отсечений.

Укажем типичные случаи использования отсечения:

1) Если фиксирован прототип использования пролог-процедуры, и в ходе доказательства всегда следует применить определенное её предложение – как в примере рекурсивного предиката для вычисления факториала:

   ```
   factorial(1,1):- !.
   factorial(N,F):- N1 is N-1, factorial(N1,F1),
                    F is F1*N.
   ```

   Здесь отсечение исключает из рассмотрения второе правило, но вместо него можно было бы также записать проверку условия N>1 в начале второго предложения.

2) Если доказательство цели в определённый момент зашло в тупик и надо рассмотреть другие пути доказательства. Именно так с помощью отсечения и предиката fail может быть смоделирован

предикат логического отрицания `no`:

```
not(G):- G,!,fail.
not(G).
```

где `G` обозначает некоторую цель. Отсечение исключает возможность применения второго правила, но только в случае успеха доказательства `G`. Предикат `fail` стоит в конце тела правила (обычное его использование), обеспечивая ответ `no` в качестве результата доказательства. Второе предложение применяется только в случае невозможности доказать `G`.

3) Если уже найдено единственное решение задачи и нет необходимости искать другое, к примеру:

```
child(X):-parent(_,X),!.
```

Этот предикат служит для поиска только одного (любого) лица, для которого известен родитель (в данном случае отсекаются точки бектрекинга, связанные с предикатом `parent`).

Проиллюстрируем использование отсечения при составлении предиката `delete(E,L1,L2)`: истинного, когда `L2` – список, полученный из `L1` удалением одного элемента, совпадающего с `E`.

Недетерминированный предикат

```
delete_one1(X,[X|T],T).
delete_one1(X,[Y|T],[Y|TY]):-delete_one1(X,T,TY).
```

реализует удаление произвольного вхождения `E` в список `L1`, его вычисление в прототипе *(i, i, o)* показано на Рис. 15. В ходе доказательства сначала найдено решение-список, в котором удалено первое вхождение `E` в список, затем найден список, в котором удалено второе вхождение и т.д.

Если же поменять местами предложения в определении этого предиката:

```
delete_one2(X,[Y|T],[Y|TY]):- delete_one2(X,T,TY).
delete_one2(X,[X|T],T).
```

то множество решений будет тем же, но решения будут находиться в обратном порядке: сначала получено решение, в котором удалено последнее вхождение `E`, затем предпоследнее и т.д. – см. Рис. 16.

Вставить сюда рис. 15 и 16 !!!

```
(1)  delete_one1(X,[X|T],T).
(2)  delete_one1(X,[Y|T],[Y|TY]):-delete_one1(X,T,TY).
```

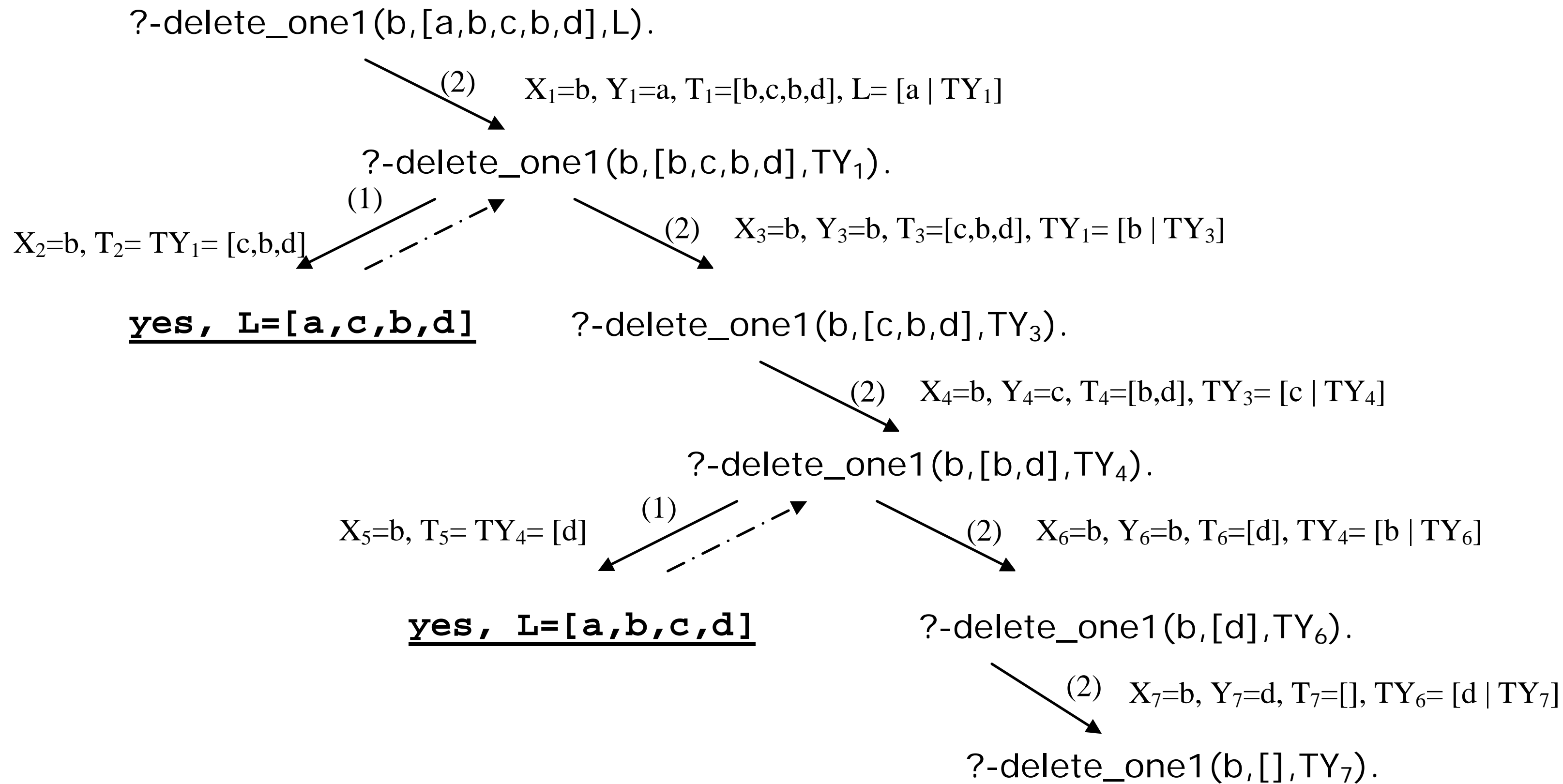

Рис. 15. Доказательство цели с предикатом `delete_one1`

```
(1)  delete_one2(X,[Y|T],[Y|TY]):-delete_one2(X,T,TY).
(2)  delete_one2(X,[X|T],T).
```

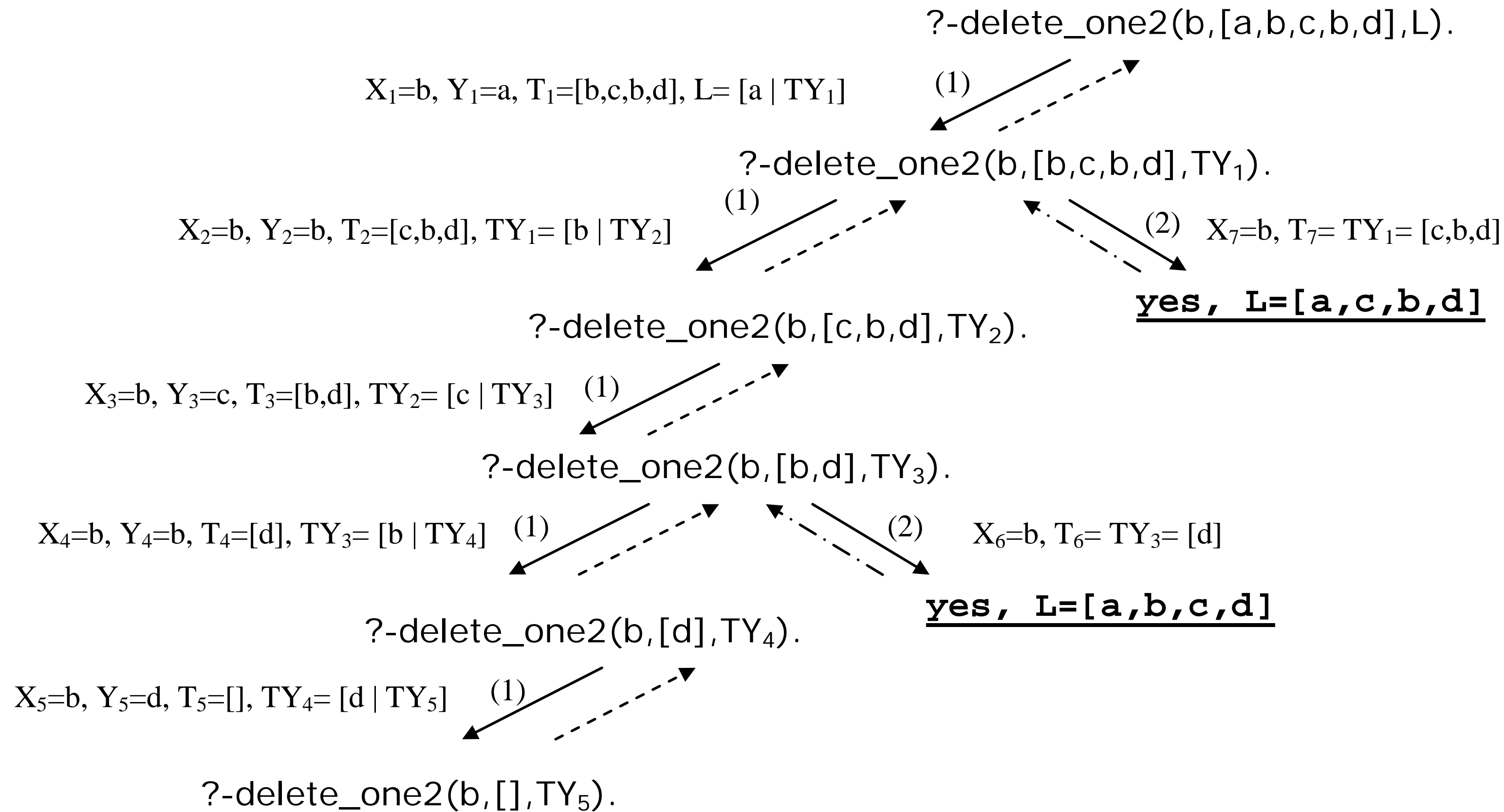

Рис. 16. Доказательство цели с предикатом `delete_one2`

Применение отсечения устраняет в рассмотренных предикатах недетерминизм. Из предиката delete_one1, путем введения отсечения ! получается предикат, удаляющий только первое вхождение E в список:

```
delete_first(X,[X|T],T):- !.
delete_first(X,[Y|T],[Y|TY]):-
                          delete_first(X,T,TY).
```

Доказательство цели с этим предикатом продемонстрировано на Рис. 17: отсечение (всех решений, кроме первого) показано в виде перечёркнутой стрелки.

```
(1)   delete_first(X,[X|T],T):-!.
(2)   delete_first(X,[Y|T],[Y|TY]):-
                     delete_first(X,T,TY).
```

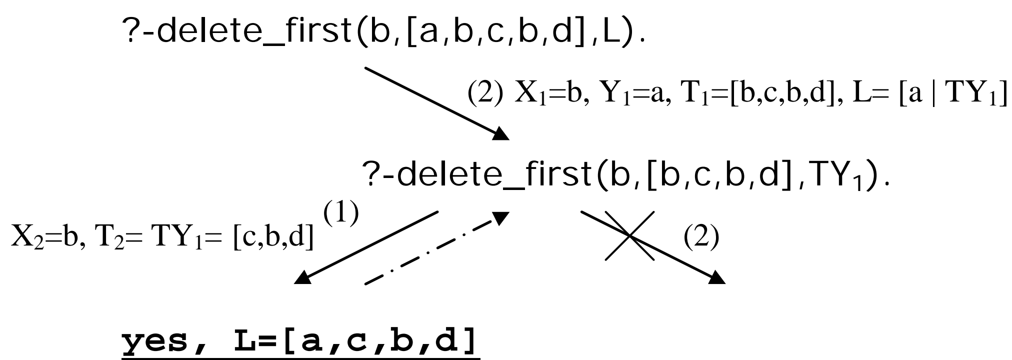

Рис. 17. Доказательство цели с предикатом delete_first

Аналогично, из предиката delete_one2, введением отсечения в конце первого предложения, получается предикат delete_last:

```
delete_last(X,[Y|T],[Y|TY]):-
                          delete_last(X,T,TY),!.
delete_last(X,[X|T],T).
```

Он удаляет последнее вхождение E в список. Вычисление предиката показано на Рис. 18; в результате выполнения первого же отсечения будут уничтожены все точки бектрекинга, начиная с точки, помеченной (*), что показано перечёркнутыми стрелками.

Прокомментировать ряды восклиц.знаков на рисунке

```
(1)   delete_last(X,[Y|T],[Y|TY]):-delete_last(X,T,TY),!.
(2)   delete_last(X,[X|T],T).
```

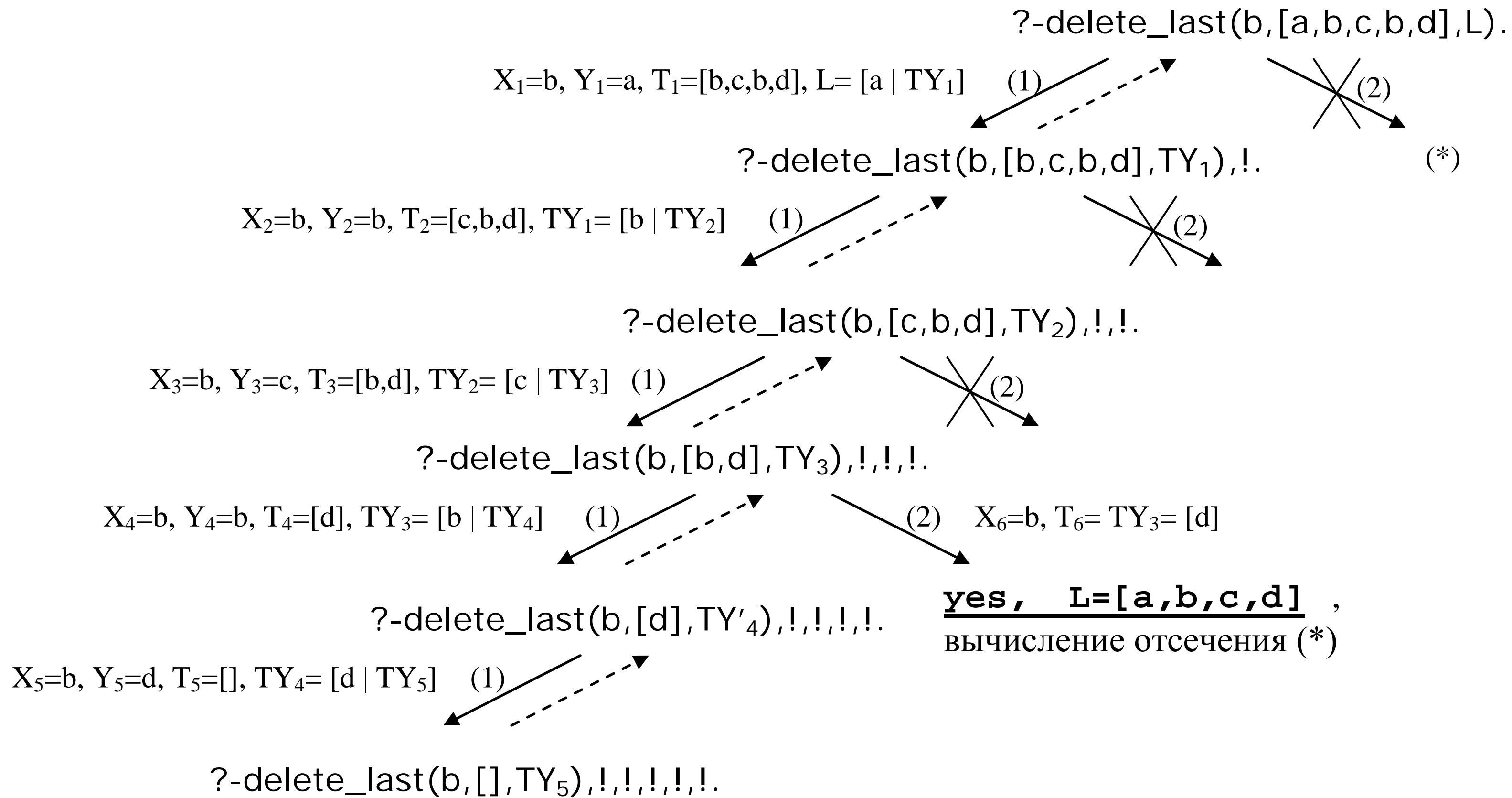

Рис. 18. Доказательство цели с предикатом `delete_last`

2.3 Предикаты работы с базой данных

Пролог-программа, состоящая из набора предложений (фактов и правил) и загруженная в пролог-интерпретатор, называется ***базой данных***.

Базу данных можно изменять во время работы программы – для этого служат встроенные предикаты `assert`, `asserta`, `assertz`, `retract` и `retractall`.

`assert(S)` и **`assertz(S)`** добавляют предложение `S` в конец базы данных;

`asserta(S)` добавляет предложение `S` в начало базы данных.

`retract(S)` удаляет из базы первое предложение, которое согласуется (сопоставимо) с предложением `S`.

а если такого предложения нет??

`retractall(S)` удаляет из базы данных все предложения, согласующиеся (сопоставимые) с предложением `S`.

Рассмотрим пример: пусть в некоторый момент пролог-программа (база данных) содержит три предложения-факта:

```
parent(tom, ann).
parent(jim, jane).
parent(jim, nick).
```

и доказывается составная цель:

```
?- asserta(parent(mary,jim)),
   assert(parent(nick,liz)),
   assertz(parent(nick,bob)),
   asserta(parent(tom,jim)),
   retract(parent(jim, _)).
```

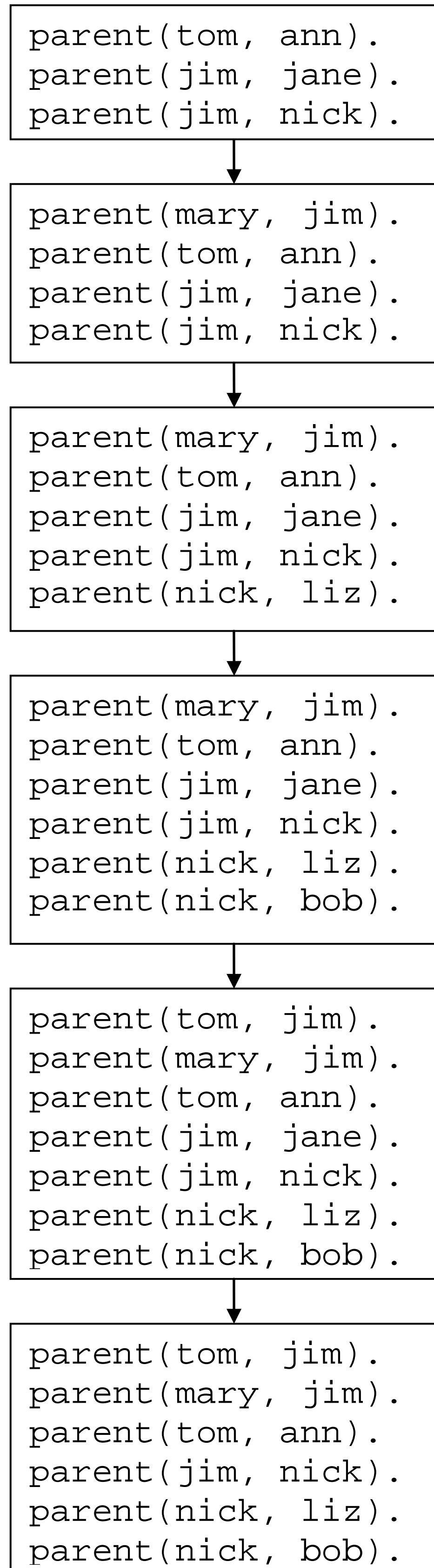

Рис. 19.

Состояние базы данных после выполнения каждой из перечисленных простых целей показано на Рис. 19.

Важно, что предикат `retract` является недетерминированным, т.е. при его выполнении порождается точка бектрекинга, и в случае возврата к ней в ходе дальнейших вычислений из базы будет удалено следующее по порядку предложение, сопоставимое с предложением `S` – в нашем примере будет удалён факт `parent(jim,nick)`. Однако при этом ранее удаленный факт уже не восстанавливается в базе. После удаления подобным образом всех согласующихся с `S` предложений предикат выработает неуспех (`no`).

В отличие от `retract` предикат `retractall` детерминирован. Он может быть запрограммирован следующими предложениями:

```
retractall(F):- retract(F), fail.
retractall(_).
```

Заметим, что при записи аргумента `S` во всех описанных выше предикатах работы с базой точка перед закрывающейся скобкой не ставится.

Пролог-предложение `S`, являющееся аргументом указанных предикатов, может быть не только фактом, но и пролог-правилом – в этом случае оно должно быть заключено в дополнительные скобки. К примеру, цель для добавления в базу определения предиката `has_child` следует записать так:

```
assert((has_child(X):- parent(X,_))).
```

Итак, пролог-интерпретатор допускает изменение пролог-программы (базы данных) в процессе её вычисления. В некоторых реализациях Пролога, в частности, в SWI-Prolog, изменяемые (с помощью `assert` и `retract`) предикаты должны быть объявлены в тексте программы *динамическими* – для этого служит директива

`:-dynamic` <имя_предиката>`/`<число_аргументов_предиката>`.`

К примеру, для динамического изменения предиката `parent` следует записать пролог-предложение `:- dynamic parent/2.`

Отметим, что любой предикат, внесенный в базу данных предикатом `assert`, автоматически рассматривается как динамический.

В практике программирования на Прологе иногда необходимы *глобальные переменные*. Глобальные переменные моделируются в Прологе с помощью базы данных и предикатов `assert` и `retract`. Например, запись в базу факта вида `a(Value)` можно трактовать как присвоение глобальной переменной `a` значения `Value`.

Другой полезный приём программирования с использованием предиката `assert` – это так называемая техника *мемоизации* [7, 18], при которой уже вычисленные факты добавляются в базу и затем используются, что повышает эффективность вычислений. При очередном вычислении предиката сначала проверяется, нет ли в базе данных уже вычисленного значения (факта), и если оно есть, то оно берётся из базы. В противном случае нужное значение вычисляется предписанным образом и записывается в базу для дальнейшего использования.

Покажем применение этой техники на примере предиката для вычисления чисел Фибоначчи:

```
/* пролог-программа вычисления чисел Фибоначчи */
:- dynamic fib_db/2. % объявление динамичности
    fib(0,1):- !.
    fib(1,1):- !.
    fib(N,F):- fib_db(N,F),!. % поиск в базе
    fib(N,F):-
        /* вычисление предыдущего числа: */
               N1 is N-1, fib(N1,F1),
        /* вычисление предпредыдущего числа: */
               N2 is N-2, fib(N2,F2),
    /*вычисление результата и его запись в базу */
            F is F1+F2, asserta(fib_db(N,F)).
```

Вычисленные числа Фибоначчи сохраняются в базе данных как факты с именем `fib_db`. В третьем предложении предиката выполняется поиск в базе нужного числа Фибоначчи, и в случае успеха используется отсечение, позволяющее не рассматривать (при возможном бектрекинге) следующее, четвёртое предложение. Использованный приём позволяет избежать неэффективности прямых вычислений чисел Фибоначчи по определяющей их рекуррентной формуле.

2.4 Оптимизация вычислений

В целом, эффективность реализуемых пролог-программами вычислений определяется расходуемыми памятью и временем [5]. Расходы по памяти зависят:

- от количества и размера используемых древовидных структур-термов, которые хранятся пролог-интерпретатором в области памяти, называемой *кучей* (*heap*);
- от глубины вложенных рекурсивных вызовов предикатов, которые сохраняются пролог-интерпретатором в стеке (*stack*);
- от количества точек бектрекинга – вся связанная с ними информация, необходимая для возврата и возобновления доказательства от нужной точки, также запоминается в стеке.

Поскольку области памяти, отведенные пролог-интерпретатору под стек и кучу, ограничены, их нерациональное использование может привести к их переполнению, после чего работа программы аварийно завершается. Стек растёт с увеличением числа точек бектрекинга и рекурсивных вызовов, поэтому для предотвращения его переполнения в ряде случаев полезно уничтожение ненужных точек бектрекинга предикатом отсечения `!`.

Заметим, что в настоящее время с ростом общей памяти компьютеров расходы по памяти стали менее критичны, чем затраты по времени (и переполнение стека и кучи для правильных пролог-программ возникает всё реже). Таким образом, время работы – главный оцениваемый параметр пролог-программ, тем более что именно по времени работы они часто проигрывают программам на императивных языках.

Так же как затраты по памяти, затраты по времени выполнения зависят от глубины рекурсии и размера обрабатываемых структур. Менее явной, но в ряде случаев более существенной является зависимость:

- от числа возвратов, выполняемых при доказательстве целей в режиме бектрекинга;
- от общего объёма работы по сопоставлению термов, включая удачные и неудачные попытки сопоставления.

Основу пролог-интерпретатора составляют встроенные механизмы сопоставления и бектрекинга, и построение эффективных пролог-программ предполагает эффективное использование этих

механизмов. Так, предикаты `fail` и `cut` необходимы для написания более оптимальных программ. Общий принцип таков: максимально скорейший отказ от ненужного перебора и от бесполезных вариантов.

Укажем некоторые полезные эвристические приёмы, позволяющие (хотя и не всегда) оптимизировать вычисления. Сразу отметим, что эти приёмы могут противоречить друг другу, так что оптимизация пролог-программы программистом должна учитывать сразу несколько факторов.

Поскольку время поиска решения зависит от порядка предложений в пролог-программе, а также от порядка доказываемых целей, который в свою очередь зависит от целей в телах пролог-правил, то целесообразно:

- ставить в начало определений предикатов более простые предложения (обрабатывающие более простые случаи), в частности, нерекурсивные предложения помещать перед рекурсивными;
- как можно раньше выполнять в телах пролог-правил проверки условий и распознавать неуспех (неуспешные ветви доказательства);
- ставить первыми в телах пролог-правил цели-предикаты с наибольшим количеством конкретизированных переменных, уменьшая тем самым время, затрачиваемое на сопоставление и перебор.

Ещё одно эвристическое правило оптимизации пролог-предикатов и программ можно сформулировать так: лучше больше переменных, чем предикатов и определяющих их предложений. Сокращая общее число предложений пролог-программы, мы уменьшаем время её просмотра при поиске применимого предложения на каждом шаге доказательства.

Кроме переупорядочивания самих предложений и целей в них при составлении пролог-программы в ряде случаев полезно пересмотреть порядок аргументов предикатов. Порядок аргументов важен, поскольку при реализации Пролога обычно применяется метод оптимизации, называемый *индексированием предложений*. Он заключается в том, что ещё до фактического выполнения сопоставления доказываемой цели-предиката G с заголовками правил программы, пролог-интерпретатор производит анализ G и сужает множество предложений, которые будут участвовать в

сопоставлении. Обычно индексирование производится для первого аргумента предикатов, и поэтому в качестве первого аргумента следует выбирать тот, что наиболее ограничивает выбор применимых предложений. Таким свойством обладает, к примеру, первый аргумент предиката append:

```
append([],L2,L2).
append([E|L1],L2,[E|L3]):-append(L1,L2,L3).
```

Действительно, если применимо первое предложение, то неприменимо второе и наоборот, так что при доказательстве цели append([a,b,c],[d,e],R) в ходе индексирования определяется, что применимо лишь второе предложение и точка бектрекинга не создается.

Ещё один метод оптимизации, применяемый пролог-интерпретатором, касается рекурсивных предикатов, представляющих так называемую *хвостовую рекурсию*. Примером хвостовой рекурсии является вышеприведенный предикат append. Ключевая идея этого метода состоит в том, что рекурсивная пролог-процедура выполняется так, как если бы она была итерационной, т.е. она вызывает сама себя без стека (за счёт чего экономится время и память). Такая оптимизация применима далеко не ко всякой рекурсивной пролог-процедуре. Для возможности преобразования рекурсивного пролог-правила процедуры A

$$A :- B_1,\ B_2,\ ...,\ B_n.$$

в её итерационный аналог необходимо соблюдение следующих условий (которые и распознаёт интерпретатор):

- В указанном предложении рекурсивный вызов единственен и стоит в конце тела правила, т.е. является последней его целью: B_n=A (отсюда произошло название: хвостовая рекурсия).
- С момента входа в пролог-процедуру A до вычисления указанного рекурсивного вызова B_n=A не осталось или не возникало точек бектрекинга, т.е. вычисление конъюнкции всех целей B_i , кроме последней цели B_n, было детерминированным и не осталось иных применимых предложений процедуры A.

Если же указанные условия не соблюдены, то полезно попробовать провести перестройку рекурсивного предиката: переставить рекурсивное обращение в конец пролог-предложений, и вставить перед ними предикат отсечения (тем самым становятся

выполнены условия оптимизации). Заметим, что оптимизация хвостовой рекурсии применяется обычно вместе с индексированием предложений.

Эффективность пролог-программ зависит также от использования предиката `not(G)`. Ясно, что в общем случае успешное доказательство составных целей с отрицанием высокозатратно, поскольку для этого надо доказать неуспешность цели `G`, что требует просмотра в режиме бектрекинга всех вариантов-путей доказательства входящих в `G` целей. Поэтому обычно рекомендуется ограничить использование предиката `not`, особенно в случаях, когда `G` – сложная составная цель.

Заметим попутно, что иногда при решении задач подходит как предикат `not`, так и отсечение. Если же выбирать между этими предикатами, то, несмотря на меньшую эффективность, использование `not` предпочтительнее, чем менее понятная конструкция с отсечением

2.5 Разностные списки и накапливающий параметр

Рассмотрим ещё один приём построения более эффективных пролог-программ – использование *разностных структур*: списков, очередей и др. [17, с.190-197]. Разностные структуры позволяют избегать ненужного копирования и просмотра термов в памяти за счёт использования встроенного механизма сопоставления пролог-интерпретатора.

Наиболее часто используются *разностные списки*, идея которых связана с тем, что каждый список можно представить как разность двух других списков, причём несколькими способами. Например, для списка `[1,2,3]` возможны следующие варианты:

```
dl([1,2,3,4,5],[4,5])
dl([1,2,3|Y],Y)
dl([1,2,3],[])
```

где `dl` – бинарный функтор для представления разностного списка, т.е. разности двух списков: первый его аргумент – уменьшаемое (укорачиваемое), второй – вычитаемое.

Таким образом, разностный список – другое представление обычного списка, при котором вычитаемый список (второй аргумент функтора `dl`) фиксирует конец обычного списка, и он становится

сразу доступен в процессе сопоставления, без просмотра обычного списка от начала до конца. Вследствие этого, если два обычных списка соединяются в один список предикатом `append` за время, линейное от длины первого аргумента, то два неполных разностных списка могут быть соединены в разностный список за константное время. Реализует это преобразование предикат `append_dl` , или *разностный* `append` , состоящий из одного пролог-факта:

```
append_dl(dl(Z1,Z2),dl(Z2,Z3),dl(Z1,Z3)).
```

Его действие поясняет схематическое изображение на Рис. 20.

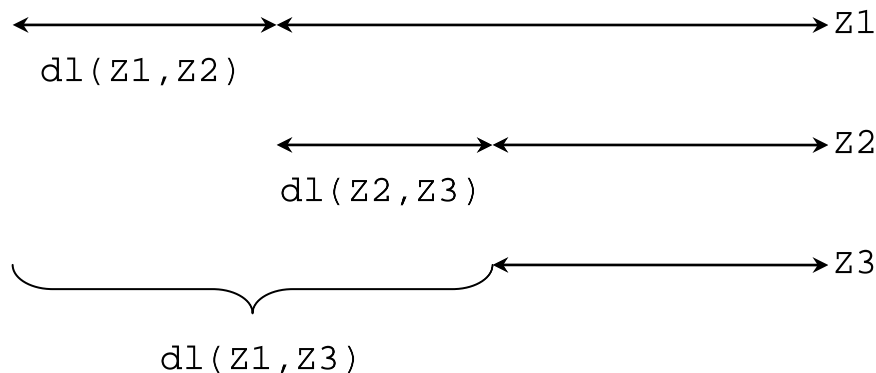

Рис. 20. Схема соединения разностных списков

Подчеркнём, что соединение разностных списков происходит неявно, в процессе сопоставления, поэтому поведение пролог-программ с разностными списками труднее для понимания, так что при их отладке имеет смысл использовать трассировку.

Заметим, что имя функтора `dl` выбрано произвольно и может быть заменено на любой другой бинарный функтор, или даже опущено – тогда первый и второй аргументы разностного списка становятся двумя отдельными аргументами предиката, использующего разностный список. Рассмотренное выше определение разностного `append` от трёх аргументов превращается таким образом в предложение

```
append_del(Z1,Z2,Z2,Z3,Z1,Z3).
```

Приведём пример вычисления конкатенации списков `[a,b,c]` и `[d,e]` с использованием этого предиката с шестью аргументами:

```
?- append_del([a,b,c|T1],T1,[d,e|T2],T2,L,[]).
```

В результате унификации этой цели с приведённым выше определением append_del за один шаг доказательства получаем:

```
Z1=[a,b,c|T1], Z2=T1, Z2=[d,e|T2], Z3=T2, Z1=L, Z3=[]
    L=[a,b,c|T1]=[a,b,c|[d,e|T2]]=
                 [a,b,c|[d,e|[]]]=[a,b,c,d,e]
```

Как правило, пролог-программы с обращениями к обычному предикату append могут быть переработаны в более эффективные за счёт использования разностных списков и разностного предиката append, что в ряде случаев по сути равносильно другому приёму построения более эффективных пролог-программ – использованию *накапливающего параметра*.

Проиллюстрируем технику накапливающего параметра на примере предиката reverse(L1,L2), истинного, если L2 – перевёрнутый список L1, например:

```
?- reverse([a,b,c],L).  => yes, L=[c,b,a]
```

Этот предикат можно запрограммировать на основе append:

```
reverse([],[]).
reverse([X|T],R):- reverse(T,T1),
                   append(T1,[X],R).
```

Однако такое решение неэффективно, поскольку во время рекурсии происходит многократное вычисление append (и при каждом его вычислении проход по его первому аргументу-списку) для подсоединения первого элемента списка в конец.

Более эффективное решение связано с последовательным накоплением обращённого списка в дополнительном аргументе:

```
reverse1(L1,L2):- rev(L1,[],L2).
rev([],L,L).
rev([X|T],Ac,L):-rev(T,[X|Ac],L).
```

В этом решении потребовался дополнительный предикат rev.

Ещё один типичный пример использования накапливающего параметра – предикат flatten_list(L1,L2) для выравнивания многоуровневого списка. Он истинен, если L2 – одноуровневый список из элементов списка L1 (т.е. из L1 убраны все внутренние скобки), к примеру:

```
?- flatten_list([[a,[b]],c],L).
                                =>yes, L=[a,b,c]
```

Определение этого предиката:

```
flatten_list(L, R):-flat(L, [], R).

flat([], Res, Res):-!.
flat([X|Y], Ac, Res):- !, flat(Y, Ac, Res1),
                         flat(X, Res1, Res).
flat(X, Res, [X|Res]).
```

Для формирования выровненного списка основной предикат `flatten_list` обращается к вспомогательному предикату `flat`, использующему накапливающий параметр (как второй аргумент). При обращении к `flat` параметр инициализируется пустым списком. Определение `flat` состоит из трёх правил, соответствующих трём взаимоисключающим случаям (и поэтому в первом и втором правиле стоит отсечение):

1. Если первый аргумент `flat` – пустой список, то накапливающий параметр содержит результат – выровненный список.
2. Если первый аргумент `flat` – непустой список, то сначала выравнивается хвост списка, с использованием накапливающего параметра `Ac` и занесением результата выравнивания в `Res1`; затем `Res1` используется в качестве накапливающего параметра для выравнивания первого элемента списка.
3. В ином случае первый аргумент `flat` не является списком, а значит, это атом или число, и его надо включить в начало накапливающего параметра, получая тем самым результирующий выровненный список.

2.6 Предикаты второго порядка

Предикатами второго порядка называются предикаты, у которых один из аргументов представляет собой предикат-цель, которая должна быть доказана.

Встроенные в Пролог предикаты второго порядка `bagof`, `setof` и `findall` служат для формирования списка всех решений внутренней цели-аргумента `Goal`. Кроме `Goal` в качестве аргумента должна быть задана переменная `Var`, фигурирующая в цели `Goal`, найденные значения этой переменной и есть те решения, которые собираются в выходном аргументе-списке `Vlist`. Рассмотрим эти предикаты подробнее.

bagof(Var,Goal,Vlist) порождает при доказательстве список Vlist тех значений переменной Var, для которых доказуема цель Goal, причём Var указана в качестве одного из аргументов предиката Goal.

Если в базе данных (в пролог-программе) содержатся факты:

```
parent(tom, ann).
parent(tom, mike).
parent(mary, mike).
parent(mike, sue).
parent(mike, john).
parent(mike, nick).
parent(john, jane).
parent(john, jim).
```

то в результате вопроса

```
?-bagof(Y, parent(mike,Y), Children).
        => yes, Children=[sue, john, nick]
```

будет получен список имён всех детей mike. Если же искомых решений нет, то вырабатывается неуспех:

```
?-bagof(Y,parent(sue,Y),Children). => no
```

В общем случае предикат bagof недетерминированный: если цель Goal имеет кроме Var другие переменные, то по отдельности для всех возможных комбинаций значений этих переменных будет вычислен Vlist. Так, результатом запроса

```
?- bagof(Y,parent(X,Y),Children).
```

будут четыре решения-списка: для каждого возможного родителя (значения X) будет сформирован список его детей:

```
  => yes, X=john, Children=[jane, jim]
; => yes, X=mary, Children=[mike]
; => yes, X=mike, Children=[sue, john, nick]
; => yes, X=tom,  Children=[ann, mike]
```

Вопрос ?- bagof(X,parent(X,_),Parents).

даст (в режиме бектрекинга) семь решений-списков: для каждого ребёнка – список его родителей:

```
Parents=[tom]; Parents=[john]; Parents=[john];
Parents=[mike]; Parents=[tom, mary];
Parents=[mike]; Parents=[mike].
```

При необходимости можно собрать все найденные подобным образом решения в один список, отмечая с помощью **знака** ^ (квантор существования) переменные (отличные от переменной Var), значения которых нас не интересуют:

```
?- bagof(X,Y^parent(X,Y),Parents). => yes,
       Parents=[tom, tom, mary, mike, mike,
                   mike, john, john].
```

Действие предиката **setof(Var,Goal,Vlist)** аналогично bagof, но результирующий список Vlist упорядочивается лексикографически (алфавитно или на основе отношения **<** для чисел), а дубликаты элементов удаляются (т.е. строится множество):

```
?- setof(X,Y^parent(X,Y),Parents). => yes,
                Parents=[john, mary, mike, tom].
```

Рассмотренные предикаты работают и в более общих случаях: в качестве их первого аргумента может быть задан произвольный терм, имеющий общие переменные с целью Goal. К примеру, можно получить список термов-пар вида *родитель/ребёнок* или *ребёнок/родитель* (в качестве разделителя пары используется знак деления):

```
?- bagof(X/Y,parent(X,Y),L).  => yes,
  L=[tom/ann, tom/mike, mary/mike, mike/sue,
   mike/john, mike/nick, john/jane, john/jim].
```

Предикат **findall(Var,Goal,Vlist)** аналогичен bagof, но он детерминирован: в список Vlist собираются все значения Var, для которых доказуема цель Goal при различных конкретизациях входящих в неё переменных, отличных от Var. При этом список не упорядочивается и повторные элементы не устраняются. К примеру:

```
?- findall(Y,parent(X,Y),Children).  => yes,
  Children=[ann,mike,mike,sue,john,nick,jane,jim].

?-findall(X,append(X,_,[a,b,c]),Xlist). => yes,
                     Xlist =[[],[a],[a,b],[a,b,c])
```

Ещё одно важное отличие предиката findall от bagof: в случае, если нет ни одного решения цели Goal, предикат findall всё равно истинен, а результирующий список – пустой:

```
?- bagof(Y,parent(ann,Y),Ch). => no
?- findall(Y,parent(ann,Y),Ch). => yes, Ch=[]
```

Предикат `findall` фактически доказывает цель `Goal` в режиме бектрекинга: после каждого успешного доказательства `Goal` найденное значение `Var` добавляется в список `Vlist` и автоматически вырабатывается неуспех – до тех пор, пока не будут рассмотрены все варианты доказательства. Работа `findall` может быть смоделирована с использованием предикатов `fail` , `cut` (`!`), `assert` и `retract` [2, с. 235].

В качестве развернутого примера использования предикатов `setof` и `findall` рассмотрим решение на Прологе задачи о коммивояжёре, являющейся классической переборной задачей.

Дана карта городов и дорог между ними, известно также расстояние между городами. Необходимо, выехав из города `a`, вернуться в него, объехав все города по разу – таким образом, маршрут коммивояжёра должен начинаться и заканчиваться городом `a` и включать все остальные города ровно по одному разу.

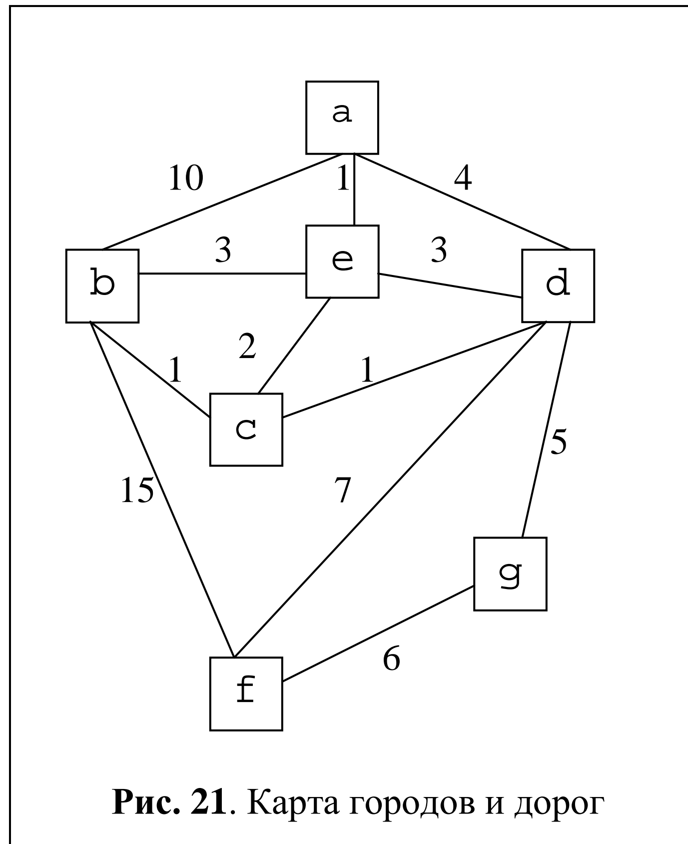

Рис. 21. Карта городов и дорог

Карта городов и дорог может быть представлена графом. Задаче поиска маршрута коммивояжера в теории графов соответствует задача поиска *гамильтонова цикла* в графе [15].

Заметим, что искомый маршрут не всегда существует, т.е. возможны такие карты дорог (графы), для которых нет нужного «кругового» маршрута. Рисунок сдвинуть вниз страницы

Естественное усложнение рассмотренной задачи – найти не просто какой-нибудь маршрут с указанными свойствами, а самый короткий из них, т.е. минимальный по длине проделанного пути. Длина пути подсчитывается как сумма пройденных расстояний между городами. В этом случае поиск осуществляется в нагруженном графе, где для каждого ребра указана его стоимость (положительное число).

Для карты городов, представленной на Рис. 21, существует четыре различных маршрута коммивояжёра, два из них являются минимальными по длине.

Рассмотрим пролог-программу для решения рассмотренной задачи. Граф городов и дорог, а также расстояния между городами представлены пролог-фактами вида `road(T1,T2,S).`
где `T1` и `T2` – названия городов, а `S` – расстояние между ними.

Основной предикат **`com_way`** для заданного начального города (первый его аргумент) строит маршрут коммивояжёра (второй аргумент) и подсчитывает его длину (третий аргумент). Маршрут представляется списком названий городов. Этот предикат недетерминированный, т.е. может найти все допустимые решения. Например, для графа, показанного на Рис. 21, будут получены следующие маршруты и их длины:

```
?- com_way(a,Way,Cost).
          =>  yes, Way=[a,e,c,d,g,f,b,a], Cost=40
      ;   =>  yes, Way=[a,e,c,b,f,g,d,a], Cost=34
      ;   =>  yes, Way=[a,d,g,f,b,c,e,a], Cost=34
      ;   =>  yes, Way=[a,b,f,g,d,c,e,a], Cost=40
      ;   =>  no
```

Для построения маршрутов предикат `com_way` использует список-множество еще непосещённых городов, который он формирует в начале вычислений с помощью предиката `setof`.

При поиске очередного маршрута предикат `com_way` опирается на предикат `path_com`, который собственно и строит путь-маршрут без циклов между заданными городами, проходящими через все остальные города. В свою очередь, `path_com` использует базовый вспомогательный рекурсивный предикат `path`, реализующий технику накапливающего параметра (для формируемого списка-пути и его длины-стоимости).

Предикат **`best_com_way`** с помощью предикатов `findall` и `com_way` собирает в список все возможные маршруты коммивояжёра, начинающиеся в заданном городе, и выбирает из них минимальный по длине путь (второй аргумент предиката).

```
?- best_com_way(a,W).
              =>  yes, W=[a,e,c,b,f,g,d,a]
         ;    =>  yes, W=[a,d,g,f,b,c,e,a]
         ;    =>  no
```

Приведём полный текст пролог-программы, решающей рассмотренную задачу.

```
/* Решение задачи коммивояжера */
/* Карта дорог: */
road(a,b,10).
road(a,d,4).
road(a,e,1).
road(b,e,3).
road(d,e,3).
road(c,e,2).
road(b,c,1).
road(d,c,1).
road(b,f,15).
road(d,f,7).
road(d,g,5).
road(f,g,6).
/* Учет двунаправленности каждой дороги: */
road_new(X,Y,Z):-road(X,Y,Z) ; road(Y,X,Z).

/* Основной предикат нахождения пути коммивояжера:
Start - город выезда коммивояжера
Way - список городов - путь коммивояжера
WayCost - длина этого пути */
com_way(Start,Way,WayCost):-
       /*Собираем список-множество всех городов:*/
    setof(X,Y^Z^road_new(X,Y,Z),City_set),
        /*Находим любой соседний город Finish:*/
    road_new(Finish,Start,Cost),
     /*Удаляем Finish из множества всех городов:*/
   delete_first(Finish,City_set,City_set1),
   /*Находим путь без циклов между Start и Finish,
              проходящий через все города:*/
   path_com(Finish,Start,Way1,WayCost1,City_set1),
         /*Пересчитываем путь и его длину:*/
   WayCost is WayCost1+Cost, Way=[Start|Way1].
```

```
/* Вспомогат. предикат: удаление элемента списка*/
delete_first(X,[X|T],T):-!.
delete_first(X,[Y|T],[Y|TY]):-
                    delete_first(X,T,TY).

/* Предикат поиска пути Way без циклов, длина пути
WayCost, между городами Start и Finish, проходящий
через все города из множества City_set */
path_com(Start,Finish,Way,WayCost,City_set):-
    path(Start,Finish,Way,WayCost,City_set,
                        [Finish],0).

/* Базовый вспомогательный предикат поиска пути,
последние два аргумента – накапливающие параметры
для пути и его стоимости */
        /*Когда все города уже посещены: */
path(Finish,Finish,Way,WayCost,[],Way,WayCost).
       /*Иначе: ищем путь из X в Finish через Y */
path(X,Finish,Way,WayCost,
         City_set,PartWay,PartCost):-
    road_new(X,Y,Cost),  %Есть дорога в город Y
    member(Y,City_set),  %Y ещё не посещен
  /*Удаляем Y из множества непосещённых городов:*/
   delete_first(Y,City_set,City_set1),
        /*Пересчёт длины пройденного пути:*/
   PartCost1 is PartCost+Cost,
       /*Рекурсивно ищем путь из Y в Finish:*/
   path(Y,Finish,Way,WayCost,City_set1,
                   [X|PartWay],PartCost1).

/*Предикат нахождения самого короткого пути */
best_com_way(Start,Way):-
        /*Находим все пути с их длинами:*/
    findall(WayCost1/Way1,
            com_way(Start,Way1,WayCost1),AllWays),
        /*Находим минимальную длину пути:*/
    min_cost(AllWays,MinCost),
        /*Выбор любого пути минимальной длины:*/
    member(MinCost/Way,AllWays).
```

```
/*Вспомогат. предикат вычисления минимальной длины
пути: если путь один, то его длина – минимальная,
иначе выбор минимума из длины первого пути и длины
минимального пути хвоста списка */
min_cost([WayCost/_],WayCost):-!.
min_cost([WayCost1/_ |Tail],MinCost):-
    min_cost(Tail,MinCost1),
    minCost is min(WayCost1,MinCost1).
```

2.7 Предикаты ввода/вывода, циклы

Рассмотрим средства ввода и вывода информации, встроенные в Пролог в виде предикатов. Текстовые файлы, участвующие во вводе/выводе рассматриваются как входные и выходные потоки. В любой момент вычислений возможен только один *активный* (*текущий*) *входной* и один *активный* (*текущий*) *выходной поток*. В начале выполнения программы оба этих потока соответствуют пользовательскому терминалу, именуемому `user`. Изменение активных файлов выполняют следующие предикаты:

`see(F)`, где `F` – имя файла, делает этот файл активным для чтения, например: `see(datafile)`.

`tell(F)`, где `F` – имя файла, делает этот файл активным для записи, например: `tell(outputfile)`.

Цели `see(user)` и `tell(user)` переключают текущие входной и выходной потоки на пользовательский терминал.

Предикаты **`seen`** и **`told`** выполняют закрытие соответственно текущего входного и текущего выходного файла, после чего ввод осуществляется с клавиатуры, а вывод – на экран.

Заметим, что перед работой с текстовым файлом лучше переименовать таким образом, чтобы его имя не содержало точку и начиналось с маленькой буквы (т.е. было бы атомом Пролога), либо имя файла записывать в одинарных кавычках (в виде строки).

Следующие предикаты реализуют собственно ввод и вывод из текущего файла и позволяют работать как с символами, так и с термами.

`get0(C)` – считывание из текущего входного потока очередного символа, при этом переменная `C` конкретизируется значением кода ASCII этого символа.

get(C) – считывание очередного печатаемого символа, т.е. при считывании пропускаются символы пробела, табуляции, конца строки и т.д.

put(C) – вывод в текущий выходной поток символа `C`, точнее, его кода ASCII (число от 0 до 127).

tab(N), где `N` – целое число, вывод `N` пробелов.

nl – вывод в текущий файл управляющего символа конца строки.

read(Var) – чтение из текущего входного потока очередного терма, и в результате переменная `Var` конкретизируется этим термом. После записи каждого терма в файле обязательно должна стоять точка, а также символ пробела либо символ конца строки.

write(T) – вывод в текущий выходной поток терма `T` в стандартной синтаксической форме (так, как они отображаются на экране компьютера); если в дальнейшем предполагается ввод информации из этого файла с помощью предиката `read`, то необходимо выводить после каждого терма символ точки и символ пробела (или же конца строки).

Предикаты ввода в случае, когда ввод данных выполняется в конце файла, в качестве ответа возвращают атом `end_of_file`(конец файла).

Заметим, что аргументы предикатов `assert` и `retract` имеют ту же синтаксически форму, что и структурные термы. Поэтому набор пролог-предложений, записанных в текстовом файле, может быть считан из него и загружен в базу данных посредством обращения к предикатам `read` и `assert`. Считывание очередного предложения и добавление его в базу выполняется двумя последовательными целями: `read(X), assert(X)`.

Кроме рассмотренных предикатов ввода/вывода полезны предикаты `consult` и `reconsult`, осуществляющие ввод из файла в базу данных сразу всей пролог-программы.

consult (F)– считывание в базу данных всех пролог-предложений из заданного файла с именем `F`, причём если база была уже не пуста, то считанные предложения помещаются в её конец (что по сути, эквивалентно множественному применению предиката `assertz`).

Заметим, что предикат `consult` не во всех реализациях языка Пролог работает вышеописанным образом. В частности, в системе SWI-Prolog его действие эквивалентно предикату `reconsult`.

`reconsult (F)`– считывание в базу данных пролог-предложений из заданного файла `F`, но при этом если в `F` есть предложения с предикатами, которые уже определены в базе данных, то старые определения заменяются на новые из файла `F`.

Для анализа пролог-программой введённой информации может быть полезен предикат `name`, выполняющий декомпозицию и синтез атомов:

`name(A,L)` истинен, если `L` – список ASCII-кодов символов атома `A`. Например:

```
?- name(name15, [110,97,109,101,49,52]).=> yes
```

В прототипе *(i, o)* `name` разбивает атом на отдельные символы, а в прототипе *(o, i)* по списку кодов символов строит атом:

```
?- name(name15, L).
             => yes, L=[110, 97, 109, 101, 49, 52]
?- name(A, [110, 97, 109, 101, 49, 52]).
             => yes, A=name15
```

Иногда удобно воспользоваться предикатами, проверяющими тип значения переменной (`X` – имя переменной):

`atom(X)`: истинен, если `X` в настоящее время конкретизирована атомом;

`atomic(X)`: `X` в настоящее время конкретизирована числом или атомом;

`number(X)`: `X` в настоящее время конкретизирована числом;

`integer(X)`: `X` в настоящее время конкретизирована целым числом;

`float(X)`: `X` в настоящее время конкретизирована вещественным числом;

`compound(X)`: `X` в настоящее время конкретизирована структурным термом.

В процессе вычислений переменная пролог-программы может быть либо свободной (без значения) либо конкретизированной (связанной), для распознавания этого служат предикаты:

`var(X)`: `X` – свободная переменная;

nonvar(X): X – это терм, отличный от переменной, или же конкретизированная переменная.

В практике программирования на языке Пролог, например, при вводе/выводе данных нередко используются *циклы, управляемые неуспехом* – они получили такое название потому, что для порождения шагов цикла используется не рекурсия, а механизм бектрекинга, включаемый неуспехом fail.

Рассмотрим, к примеру, рекурсивную организацию ввода и вывода символов до символа звёздочки (предикат begin служит для входа в цикл):

```
begin : - readchar(C), while(C).
while(C): - C='*'.
while(C): - write(C), readchar(Z), while(Z).
```

Такой цикл можно запрограммировать иначе, если использовать специальный недетерминированный предикат **repeat** без аргументов, который порождает при бектрекинге бесконечные вычисления:

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

Цикл ввода/вывода будет выглядеть так:

```
begin (X): - repeat,readchar(C),while (C),!.
while(C):- C='*', !.
while(C):- write(C),fail.
```

В этом решении предикат while вычисляется успешно лишь в момент выхода из цикла (применение первого предложения). Отсечение в первом предложении while предохраняет от незавершающихся вычислений (оно отсекает точку бектрекига, связанную с предикатом while), а отсечение в процедуре begin исключает повторный вход в цикл repeat, т.к. отсекает точку бектрекинга, возникающую при выполнении цели repeat.

Циклы, управляемые неуспехом, не имеют логической интерпретации, и в этом смысле они хуже рекурсивных циклов. Заметим, что вместо встроенного fail можно использовать предикат с любым именем, для которого в программе нет определяющих предложений.

В заключение приведём пример предиката, организующего на основе repeat цикл загрузки в базу данных фактов-термов из файла; имя файла задаётся в качестве аргумента предиката. Предполагается, что в файле после каждого факта стоит точка и символ пробела.

```
load_file(Data_file):- see(Data_file),
                repeat,read(F),
                    ( F=end_of_file, !, seen ;
                       assert(F), fail ).
```

В этом предикате после открытия файла для чтения запускается цикл repeat, на каждом шаге которого из файла считывается факт в переменную F. Если файл закончился, то F в качестве значения получит атом end_of_file, и в этом случае отсечение уничтожит точки бектрекинга (созданные для предиката repeat, а также для альтернативы-дизъюнкции, записанной в скобках), после чего происходит закрытие файла. Если же файл ещё не закончился, и из него считан факт F, то он записывается в базу данных, после чего fail вырабатывает неуспех. В результате происходит возврат в точку бектрекинга repeat, и повтор всех действий цикла. При этом возврате действия предикатов read и assert не отменяются, и поэтому происходит считывание из файла следующего факта и его запись в базу данных.

Выполнение рассмотренного цикла схематически показано на Рис. 22, для краткости предикаты see и seen опущены, а вместо атома end_of_file использован атом end.

2.8 Графическая библиотека системы SWI-Prolog

Система SWI-Prolog включает средства объектно-ориентированной библиотеки XPCE (X-windows Process Control Engineering) [24], предназначенной для создания графического пользовательского интерфейса. Для SWI-Prolog ряд средств библиотеки синтаксически оформлен в виде предикатов, так что вызов некоторых методов классов, представленных в библиотеке, имеет вид пролог-целей.

Рассмотрим часть базовых возможностей этой библиотеки, позволяющих создать диалоговое окно с такими стандартными элементами графического интерфейса как кнопки, области для ввода/вывода текста и чисел, области для вывода графических фигур.

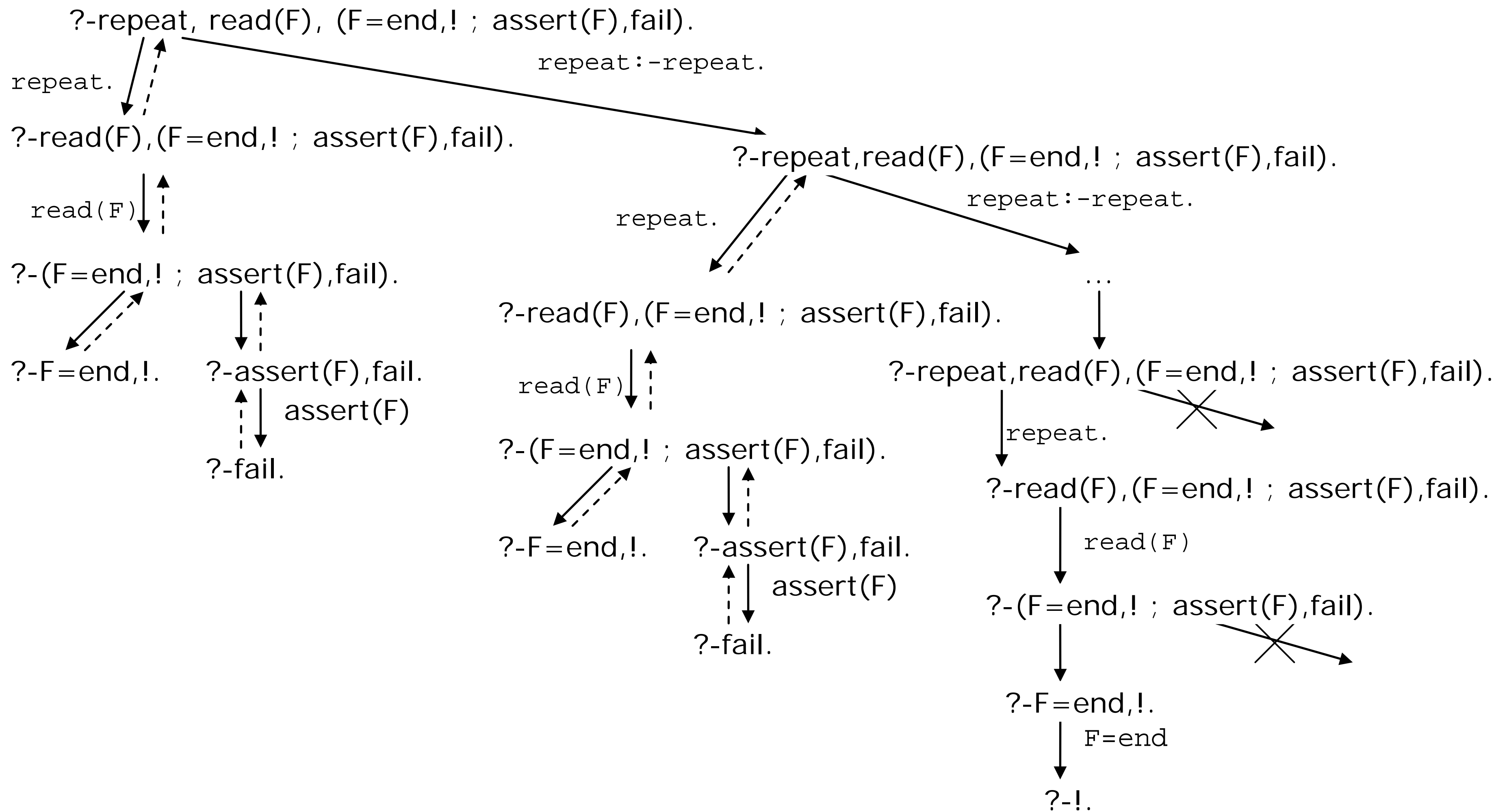

Рис. 22. Схема вычисления цикла с использованием `repeat`

Кроме средств, рассматриваемых в данном разделе, библиотека включает возможности заливки цветом замкнутых фигур, скругление углов прямоугольника и др.

Детальное описание всех классов и методов библиотеки содержится в справочном руководстве, вызов которого реализует цель, заданная в диалоговом окне системы SWI-Prolog:

```
?-manpce.
```

Подключение библиотеки XPCE

Для операционных систем семейства Unix подключение библиотеки XPCE выполняет предикат:

```
use_module(library(pce)).
```

Для ОС MS Windows загрузка библиотеки XPCE происходит автоматически при запуске интерпретатора SWI-Prolog.

Классы и объекты библиотеки XPCE

Наиболее часто при создании графического интерфейса пролог-приложений используются следующие классы объектов:

dialog – диалоговое окно;

button – кнопка;

int_item – поле для ввода/вывода числа;

text_item – поле для ввода/вывода строки;

text – класс текст, предназначенный для вывода текста;

size(width,height) – размер объекта (ширина, высота);

message – класс для создания и посылки сообщения объекту;

picture – область для вывода графических объектов;

point(X,Y) – точка с координатами X и Y;

box(width,hight) – прямоугольник заданной ширины и высоты;

ellipse(width,hight) – эллипс заданной ширины и высоты;

circle(diameter) – окружность заданного диаметра;

arc(radius,start,size) – дуга окружности заданного радиуса, start – угол начальной точки в градусах, size – величина угла, образующего дугу;

**line(start_x, start_y, end_x, end_y,
arrows:={none,first,second,both})** – отрезок линии

с началом и концом в заданных точках, `arrows` – наличие стрелки на концах линии;

`path` – рисование сложных фигур по заданным точкам;

`dialog_group` – создание группы объектов диалогового окна для их последующего совместного вывода в диалоговое окно.

Более подробную информацию о классах библиотеки можно получить, вызвав в диалоговом окне системы SWI-Prolog предикат **`help(Class)`**, где `Class` – имя класса.

Основные предикаты и методы классов

Пусть `Obj` – ссылка на объект, а `Class` – терм, главный функтор которого – имя какого-либо класса библиотеки.

Предикат **`new(Obj,Class)`** – создание объекта, являющегося экземпляром данного класса; в аргументе `Class` возможно задание некоторых параметров объекта. Параметры делятся на обязательные и необязательные. При отсутствии обязательного параметра вырабатывается ошибка. Если необязательные параметры объекта не заданы при его создании, им присваиваются значения по умолчанию, заданные в описании класса. Если в качестве `Obj` указывается переменная, она получает в качестве значения ссылку на созданный объект, эта ссылка генерируется системой и имеет вид *`@число`*. Пользователь может сам задать имя для ссылки на создаваемый объект, оно должно иметь вид *`@атом`*.

Примеры применения предиката:

`new(DW,dialog('Геометрические фигуры'))` – создание диалогового окна с заголовком "Геометрические фигуры", переменная `DW` будет конкретизирована ссылкой на созданное диалоговое окно.

`new(@pict,picture('Головоломка'))` – создание графической области для вывода рисунка с заголовком "Головоломка" (заголовок может быть не задан). Имя `@pict` задается для доступа к созданной области.

`new(Ln,line(0,0,15,15,second))` – создание объекта-линии с началом в точке `(0,0)` и концом со стрелкой в точке `(15,15)`.

`new(@box,box(Widht,Height))` – создание прямоугольника с размерами, заданными значениями переменных `Widht` и `Height`.

Предикат **get(Obj,TypeInf,Var)** – получить от объекта, заданного ссылкой Obj, информацию вида TypeInf, которая связывается с переменной Var. Информация может представлять собой ссылку на объект какого-либо класса (в примере ниже Sz связывается с ссылкой на объект класса size, содержащий информацию о созданном диалоговом окне DW). Полученную ссылку на объект можно использовать в последующих get для получения новой информации, например:

get(DW,size,Sz),get(Sz,width,W),get(Sz,height,H) – получение информации о размере созданного ранее диалогового окна DW.

Предикат **send(Obj, Me(Param))** – послать объекту Obj сообщение (вызвать метод) Me. Me имеет вид терма, соответственно у него могут быть заданы параметры Param (параметры могут и отсутствовать). Использование send возможно также в виде send(Obj, Me, Param), где параметры сообщения заданы в виде третьего и последующих аргументов (общее количество параметров не более 10). Пример использования send (в двух возможных видах):

send(DW, width(800)),send(DW, height,650) – установить новые размеры уже созданного диалогового окна DW.

send(@box,radius,R) – скругление углов созданного ранее прямоугольника @box, R – радиус скругления углов.

Предикат **send_list(Obj,Me,MeList)**, где MeList – список параметров, эквивалентен последовательным send, у которых два первых аргумента – Obj и Me, а третий аргумент – очередной элемент списка MeList (пример см. ниже).

Предикат **free(Obj)** – удаляет из базы объектов XPCE созданный ранее объект Obj (уничтожает объект).

Для работы с диалоговым окном и графическими объектами наиболее часто используются следующие сообщения без параметров:

open – визуализация объекта;

destroy – уничтожение визуализированного объекта. Например:

send(DW,open) – отображение на экране созданного ранее окна DW.

send(DW, destroy) – уничтожение созданного ранее объекта – диалогового окна DW. Замечание: уничтожается только сам объект DW,

а все добавленные в него объекты (поля, кнопки и т.д.) перестают отображаться на экране, но остаются в базе XPCE. Для их уничтожения можно воспользоваться, к примеру, предикатом `free`.

Для диалоговых окон часто используется сообщение

append – добавление в окно элемента графического интерфейса, например:

`send(DW,append,@pict)` – добавление в диалоговое окно `DW` созданной ранее области `@pict` для вывода графического рисунка.

Параметрами подобного сообщения выступают добавляемый объект (в примере: `@pict`) и, опционально, одно из следующих значений: `below`, `above`, `left`, `right`, `next_row`, указывающих на его положение относительно последнего уже добавленного объекта. Если положение нового объекта не указано, он добавляется в направлении по умолчанию (к примеру, поле для ввода – ниже, кнопка – правее предыдущей).

Сообщение `append` часто используют в предикате `send_list`.

Например, следующее сообщение:

```
send_list(DW, append,
       [@pict, button(exit,message(DW,destroy))])
```

добавляющее в диалоговое окно область для вывода графического рисунка и кнопку завершения работы (с надписью `exit`), уничтожающую диалоговое окно, эквивалентно последовательности двух сообщений:

```
send(DW, append, @pict),
send(DW,append, button(exit,message(DW,destroy))).
```

Можно группировать созданные объекты графического интерфейса, добавляя их в созданный объект класса `dialog_group`. При добавлении объекта в группу возможно указать его расположение относительно предыдущего добавленного объекта: `bellow`, `right`, `next_row`. Созданную группу объектов можно затем добавить в диалоговое окно, указав её расположение (например, правее предыдущего добавленного объекта). При этом все элементы группы будут расположены вместе, их взаимное расположение будет таким, каким оно было задано при создании группы. Приведём пример:

```
% Создаём поле для ввода чисел
new(Pole_int,int_item(поле,default:=50)),
% Создаём группу объектов
new(Group,dialog_group(box)),
% Добавляем в группу созданное поле
send(Group,append,Pole_int),
% Добавляем в группу 2 кнопки: кнопка1 и кнопка2
send_list(Group, append,
            [button(кнопка1,message(…)),
             button(кнопка2,message(…))]),
% Добавляем в группу кнопку3 ниже двух предыдущих
send(Group, append,
        button(кнопка3,message(…)), next_row),
% Добавляем кнопку4 - она будет правее кнопки3
send(Group, append, button(кнопка4,message(…))).
```

В результате в созданной группе Group будут 5 объектов диалогового окна: вверху поле для ввода чисел, под ним 4 кнопки. Кнопки будут расположены следующим образом: первый ряд кнопок – кнопка1, правее кнопка2, второй ряд кнопок – кнопка3 (под кнопкой1), правее кнопка4. Созданную группу после этого можно добавить, например, в диалоговое окно, указав её расположение относительно последнего добавленного объекта. При необходимости можно создавать группу, в состав которой входят другие ранее созданные группы объектов.

Для графической области часто используются сообщения:

clear – очистить область от всех визуализированных объектов.

display – визуализация графического объекта, который выступает как обязательный параметр сообщения; необязательным параметром являются координаты point(X,Y) точки левого верхнего угла области, в которой отображается объект. Например:

send(@pict,display,circle(10),point(10,10)) – вывод в графическую область @pict окружности диаметром в 10 пикселей в квадрате с координатами (10,10)левого верхнего угла.

Метод **message(Obj, Me, Params)** используется для посылки сообщения Me с параметрами Params объекту Obj, но отложенного по времени. Т.е. он эквивалентен вычислению send(Obj, Me, Params), но произведённому позже. Обычно

используется при описании кнопки, соответствующее сообщение посылается при нажатии на эту кнопку.

Бинарный инфиксный оператор **?** используется для получения информации об объекте, но его выполнение также отложено по времени. К примеру, вычисление `X?selection` позволяет получить значение, введённое в поле для ввода числа (`X` – ссылка на объект, представляющий это поле), т.е. производит вычисление, аналогичное `get`, но отложенное по времени. Это вычисление происходит в момент формирования сообщения, заданного в `message`.

Заметим, что с точки зрения XPCE Пролог является объектом, ссылка на который связана с именем `@prolog`, а вычисление прологовского предиката происходит посредством посылки сообщения объекту `@prolog`, где в качестве метода выступает прологовский предикат, а в качестве параметров – аргументы этого предиката.

Для графических объектов используются следующие сообщения:

`colour` – задание цвета линии прорисовки фигуры;

`fill_pattern` – заливка замкнутой фигуры цветом.

Приведём в качестве примера фрагмент кода, позволяющий создать треугольник с углами в заданных трёх точках, цвет треугольника – красный, внутренняя заливка – зелёным цветом.

```
/* Создаём объект класса path */
new(@triangle,path),
/* Задаём точки - углы треугольника */
send_list(@triangle,append,
        [point(X1,Y1),point(X2,Y2),point(X3,Y3)]),
/* Делаем фигуру замкнутой */
send(@triangle,close,@on),
% Задаём цвет фигуры - красный
send(@triangle,colour,colour(red)),
% Задаём цвет заливки - зелёный
send(@triangle,fill_pattern,colour(green)),
/* Выводим треугольник в созданную ранее
графическую область @pict */
send(@pict,display,@triangle).
```

Ниже приведена пролог-программа, в которой создаётся диалоговое окно с заголовком "Геометрические фигуры". Окно

содержит поле для вывода графического объекта, кнопки "Нарисовать прямоугольник", "Нарисовать эллипс", "Стереть", "exit", а также четыре поля: для ввода высоты и ширины рисуемой фигуры, X и Y-координат левого верхнего угла поля для вывода фигуры. Программа включает основной предикат `start` и вспомогательные предикаты `mybox` и `myellipse` для рисования фигур.

```
/* Запуск программы */
start :- /* создаём диалоговое окно */
         new(DW, dialog('Окно моей программы')),
/* создаём поле Picture для вывода графических
фигур и задаём его размеры */
         new(Picture, picture),
         send(Picture, width(350)),
         send(Picture, height(350)),
/*добавляем   в   это   диалоговое   окно   список
объектов*/
         send_list(DW,append,
/*графическое поле*/
            [Picture,
/*поле для ввода ширины фигуры: заданы имя поля,
значение по умолчанию и min и мах ограничения */
            new(Width, int_item(width, low := 10,
                   high := 300, default := 150)),
/* аналогичное поле для высоты фигуры */
            new(Height,int_item(height, low := 10,
                  high := 300, default := 150))]),
/*добавляем поле для ввода X-координаты */
         send(DW, append, new(X, int_item(x_coord,
                                default := 10))),
/*добавляем поле для ввода Y-координаты */
         send(DW, append, new(Y, int_item(y_coord,
                                 default :=10))),
/*добавляем список объектов*/
         send_list(DW,append,
/* кнопка "Нарисовать прямоугольник", при нажатии
происходит вычисление предиката mybox с пятью
аргументами, последние четыре получают значения из
соответствующих полей ввода */
               [button('нарисовать прямоугольник',
```

```
                    message(@prolog,mybox,Picture,
                       Width?selection,
                       Height?selection,
                       X?selection, Y?selection)),
/* аналогичная кнопка "Нарисовать эллипс" */
                 button('нарисовать эллипс',
                      message(@prolog, myellipse,
                       Picture,
                       Width?selection,
                       Height?selection,
                       X?selection, Y?selection)),
/*кнопка "стереть" очищает графическую область*/
                 button('стереть',
                        message(Picture,clear))]),
/*добавляем кнопку "exit" завершения работы
программы – уничтожает диалоговое окно */
       send(DW, append,
          button(exit, and(message(DW, destroy),
                      message(Picture, destroy)))),
/*делаем объекты DW и Picture активными */
         send(Picture, open),
         send(DW, open).

/* вспомогательный предикат рисования
прямоугольника заданной ширины и высоты, с
заданными координатами левого верхнего угла */
mybox(Picture,Width,Height,X,Y) :-
         send(Picture, display,
         new(_,box(Width,Height)), point(X,Y)).

/*аналогичный предикат рисования эллипса */
myellipse(Picture,Width,Height,X,Y) :-
          send(Picture, display,
          new(_,ellipse(Width,Height)),
                    point(X,Y)).
```

3 Задачи на логическое программирование

В задачах данного раздела предполагается использовать только логические средства языка Пролог, описанные в разделе 1; запрещается использовать металогические предикаты с побочными эффектами: `free`, `bound`, `read`, `assert`, `retract` и др. При логическом программировании предикатов целесообразно определить их детерминированность/недетерминированность и возможные прототипы работы. Для недетерминированных предикатов следует исключить возможность получения дважды одного и того же решения.

Проблемно-ориентированные предикаты

1. В пролог-программе заданы факты вида `parent(a,b)`: `a` – родитель для `b`, факты вида `man(c)`: `c` – мужчина, и факты `woman(d)`: `d` – женщина. Используя эти предикаты, составить предикат `niece_f(X,Y)`, истинный, если `X` – племянница для `Y` по отцовской линии. Нарисовать дерево доказательство цели `?-niece_f(V,nick)` на основе пролог-программы, определённой в разделе 1.2.
2. В пролог-программе заданы факты вида `parent(a,b)`: `a` – родитель для `b`, и факты вида `man(c)`: `c` – мужчина. Используя эти предикаты, определить:

   а) отношение `grand_nephew(X,Y)`: `X` является внучатым племянником для `Y` (т.е. `X` – внук сестры или брата `Y`).

   б) отношение `has_cousin_Kim(X)`: у человека `X` есть двоюродный брат или сестра по имени `Kim`.
3. Для предиката , определенного пролог-предложениями
   ```
   ancestor(X, Y) :- parent(X, Y).
   ancestor(X, Y) :- parent(X, Z),ancestor(Z, Y).
   ```
   нарисовать дерево доказательства цели
   ```
   ?-ancestor(V,john).
   ```
4. В пролог-программе заданы факты из мира разноцветных кубиков одинакового размера, которые можно ставить на стол и один на другой: `on_table(a)` – кубик `a` лежит на столе; `on_box(a,b)` – кубик `a` лежит на кубике `b`; `clear(a)` – верхняя поверхность кубика `a` свободна.

Заданы также факты о цвете каждого кубика: red(c) или blue(c). Используя эти предикаты, определить:

а) отношение base_4(Z): Z – это нижний кубик в столбике из четырёх поставленных друг на друга кубиков;

б) отношение stack_2(X,Y): кубики X и Y образуют одноцветный столбик.

5. Определить предикат Rect с аргументами P1,P2,P3,P4 – верный, когда четыре точки плоскости P1,P2,P3,P4, будучи соединённые отрезками прямой (в том порядке, как они задаются в качестве аргументов), образуют на плоскости прямоугольник, стороны которого параллельны осям координат. Для задания двух координат точек использовать функтор point(X, Y).

Предикаты работы со списками

1. Определить на основе предиката append:

   а) предикат two_times(E,L): объект E входит в список L не менее двух раз;

   б) предикат neib(E1,E2,L): E1 и E2 являются соседними элементами в списке L .

2. На языке Пролог определено отношение D(X,Y):

```
D(X,[X]).
D(X,[X|Z]):- D(X,Z).
```

   Сформулировать словесное описание этого отношения, привести примеры целей (вопросов) с этим отношением и нарисовать дерево доказательства одной из этих целей.

3. Составить предикат number(E,N,L): N – порядковый номер элемента E в списке L. Привести примеры вопросов с этим предикатом, показывающие его работу в разных прототипах, и получающиеся ответы (решения).

4. Дано определение бинарного отношения D(X,Y):

```
D(X,[X,X]).
D(X,[Y|Z]):- D(X,Z).
```

   Словесно описать это отношение и привести примеры вопросов-целей с ним.

5. Составить предикат length(N,L): N – количество элементов в списке L на верхнем уровне; проверить работу предиката в разных прототипах.

6. Определить рекурсивно предикат `last(A,L)`: `A` является последним элементом списка `L` (на верхнем его уровне).
7. Составить предикат `sublist(L1,L2)`: `L1` – произвольный подсписок списка `L2`, т.е. непустой отрезок из подряд идущих элементов `L2`.
8. Определено отношение `D(X,Y)`:

```
D(X,[]).
D(X,[X]).
D(X,[X,Y|Z]):- D(X,Z).
```

   Сформулировать словесное описание этого отношения, привести примеры вопросов-целей с этим отношением и нарисовать дерево доказательства одной из целей.
9. Составить предикат `sort(L1,L2)`: `L2` – отсортированный по неубыванию список чисел из `L1`.
10. Определить отношение `no_doubles(L1,L2)`: `L2` – список, являющийся результатом удаления из `L1` всех повторяющихся элементов (на верхнем уровне);

Предикаты работы с множествами

`M1`, `M2`, `M3` – множества, представленные в виде списков элементов без повторений, порядок элементов в них не существенен, тип элементов – произволен.

Составить пролог-предикаты:

1. `subset(M1,M2)`: множество `M1` является подмножеством `M2`;
2. `union(M1,M2,M3)`: множество `M3` – объединение `M1` и `M2`;
3. `intersection(M1,M2,M3)`: `M3` – пересечение `M1` и `M2`;
4. `subtraction(M1,M2,M3)`: `M3` – разность `M1` и `M2`;
5. `decart(M1,M2,M3)`: `M3` – декартово произведение `M1` и `M2`.

Замечание: для обеспечения инвертируемости предикатов можно использовать предикат `no_doubles`.

Предикаты работы с деревьями

Бинарные деревья представляются в виде термов, записываемых с помощью константы `nil` (пустое дерево) и тернарного функтора `tree`: `tree(`*<левое поддерево>*`,` *<правое поддерево>*`,` *<метка узла дерева>*`)`.

Например:

```
tree(tree(nil,nil,f),tree(tree(nil,nil,p),
                          tree(nil,nil,r),k),g)
```

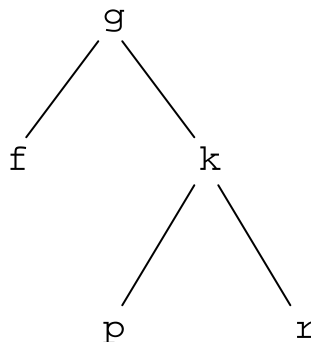

Рис. 23. Пример бинарного дерева

представляет дерево, изображенное на Рис. 23. Метками узлов могут быть атомы или числа.

Составить пролог-предикаты:

1. `tree_depth(T,N)`: `N` – глубина дерева `T` (количество дуг в самой длинной его ветви);
2. `label_tree(V,T)`: метка `V` есть среди меток узлов дерева `T`;
3. `sub_tree(T1,T2)`: дерево `T1` является непустым поддеревом дерева `T2`;
4. `replace(T1,V1,T2,V2)`: `T2` – дерево, полученное путем замены всех меток `V1`, встречающихся в узлах дерева `T1`, на метку `V2`;
5. `del_trees(T1,V,T,T2)`: `T2` - дерево, полученное из дерева `T1` удалением всех поддеревьев с меткой `V` в корне;
6. `flatten_tree(T,L)`: `L` – список меток всех узлов дерева `T` (т.е. «выровненное» дерево), например:

```
flatten_tree(tree(tree(nil,nil,f),
                  tree(tree(nil,nil,p),
                      tree(nil, nil,r),k),g),L)
              =>  yes, L=[f,g,p,k,r]
```

Написать два варианта этого предиката: с применением предиката `append` и с использованием накапливающего параметра.

Предикаты для работы с графами

Граф может быть представлен несколькими способами, в том числе:

I. набором фактов вида `edge(P,R,N)`, каждый из которых устанавливает наличие ребра между вершинами `P` и `R` со стоимостью `N` (целое неотрицательное число);

II. в виде единой структуры – структурного терма, включающего список рёбер (заданных аналогично с помощью тернарного функтора `edge`) и список вершин графа, объединяемых бинарным функтором `graph`.

Например, ориентированный граф, показанный на Рис. 24, может быть задан фактами:

```
edge(a,c,8).
edge(a,b,3).
edge(c,d,12).
edge(b,d,0).
edge(d,e,9).
```

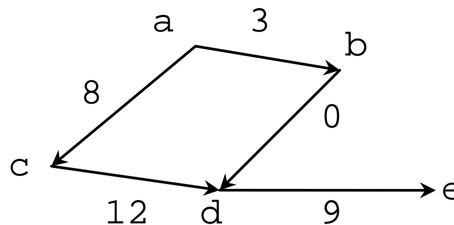

Рис. 24. Пример графа

или же структурным термом:

```
graph([edge(a,c,8),edge(a,b,3),edge(c,d,12),
       edge(b,d,0),edge(d,e,9)],[a,b,c,d,e]).
```

Отметим, что первый способ не позволяет представлять графы с одиночными вершинами (без исходящих дуг). Второй способ – менее экономный (т.к. информация частично дублируется), но более универсальный. Возможны и другие способы задания графов, например, задание графа как списка пар вида

`pair(`*<вершина>*`,` *<список вершин, смежных с ней>*`)`,

где `pair` – бинарный функтор. Для рассмотренного примера графа таким представлением будет список

```
[pair(a,[b,c]),pair(b,[a,d]),pair(c,[a,d]),
    pair(d,[b,c,e]),pair(e,[d])].
```

В нижеследующем описании предикатов предполагается первый способ задания графа. При использовании второго способа представления графа предикаты должны содержать дополнительный аргумент – сам граф `G`. Например: `path(G,X,Y,L)`, где `G` – структура, представляющая граф.

Составить пролог-предикаты:

1. `connected(X,Y)`, выражающий отношение связности двух вершин `X` и `Y` ациклического неориентированного графа;
2. `path(X,Y,L)`: `L` – путь без петель между вершинами `X` и `Y` (т.е. список вершин между этими вершинами); граф предполагается связным;
3. `min_path(X,Y,L)`: `L` – путь между вершинами `X` и `Y`, имеющий минимальную стоимость (стоимость пути равна сумме стоимостей входящих в него рёбер); граф предполагается связным;
4. `short_path(X,Y,L)`: `L` – самый короткий путь между вершинами `X` и `Y` (длина пути равна количеству входящих в него рёбер); граф предполагается связным;
5. `cyclic` (предикат без аргументов): граф является циклическим (т.е. не является деревом или лесом);
6. `is_connected` : граф является связным (т.е. для любых двух его вершин существует связывающий их путь).

Замечание: несмотря на внешнюю похожесть определений предикатов `min_path` и `short_path` их эффективная реализация на Прологе требует применения разных алгоритмов перебора вершин в графе.

Нахождение минимального по стоимости пути (предикат `min_path`) предполагает полный просмотр графа и путей в нём. Такой просмотр целесообразно реализовать на основе алгоритма поиска вглубь *Depth_First_Search*, поскольку он по сути встроен в Пролог (пролог-интерпретатор при доказательстве целей просматривает и обрабатывает альтернативы именно стратегией в глубину). При поиске вглубь всегда для продолжения поиска выбирается вершина графа, наиболее удалённая от исходной вершины (от которой был начат поиск) [2, с.330-335; 17, с.224-232].

В отличие от `min_path` нахождение самого короткого пути (предикат `short_path`) в общем случае не требует полного просмотра графа; более быстрое обнаружение такого пути гарантируется другим алгоритмом перебора вершин графа – поиском вширь *Breadth_First_Search*. При поиске вширь всегда для продолжения поиска выбирается одна из вершин, наиболее близких к исходной.

4 Задания практикума по языку Пролог

Задания практикума служат для освоения языка Пролог как практического языка программирования. Задания включают разработку пролог-программ решения типичных задач из области искусственного интеллекта и предполагают применение методов этой области, включая эвристический перебор, формальное представление знаний, анализ естественного языка. При программировании допускается использование любых встроенных предикатов языка, в том числе металогических.

Во всех вариантах заданий разрабатываемая пролог-программа должна быть организована как законченная система с удобным и понятным пользовательским интерфейсом, предусматривающим возможность:

- считывания исходных данных из внешнего файла и сохранения их в файле после обработки;
- редактирования данных для неоднократного выполнения основных функций системы;
- выдачи пользователю системы необходимых в текущий момент подсказок и пояснений.

В вариантах заданий, предполагающих проведение перебора для поиска нужного решения, для более эффективной его реализации следует использовать эвристики.

Система, отвечающая на вопросы о родственных отношениях

Основным модулем системы должна быть ***база знаний***, в которой хранится информация о членах семьи (не менее 20 человек) из нескольких (не менее четырех) поколений и родственных отношениях между ними (не менее 12-15 различных отношений).

Базовые отношения родства, к которым относятся 4 понятия: *родитель*, *супруги*, *мужчина*, *женщина*, должны быть представлены в базе явно, в виде соответствующих фактов Пролога. Все остальные родственные отношения (внук, дядя, невестка и т.п.) определяются на основе базовых отношений и для конкретных лиц должны выводиться пролог-системой исходя из базовых фактов. Таким образом, база знаний о родственных отношениях является дедуктивной (т.е. обладает возможностями дедуктивного вывода).

Отметим, что базовый набор отношений может быть выбран не единственным способом: например, вместо бинарного отношения *родитель* могут быть взяты отношения *мать* и *отец*.

Основная функция системы – ответы на запросы двух видов:

- Определить для двух конкретных членов семьи, в каком родственном отношении они находятся, например: *В каком родстве Елена и Петр?*
- Определить для заданного члена семьи, кто состоит с ним в конкретном родственном отношении: например: *Кто является сестрой Ольги?* или *Кто внуки Андрея?*

Система также должна допускать модификацию базы знаний в диалоге с пользователем: введение в неё новой информации о членах семьи и коррекцию старой информации. При этом в случае ввода новых утверждений о родственных отношениях, не являющихся базовыми, они должны быть «разложены» системой на более простые (базовые), которые и заносятся в базу знаний.

Заметим, что в случае вопросов о родственном отношении двух членов семьи, не находящихся в близком родстве, необходимо в ответе произвести «синтез» искомого отношения из нескольких известных системе отношений, например: *Елена - тётя внука Петра* или *Елена - тётя Николая, внука Петра*.

Для корректной работы системы все члены семьи должны иметь разные имена. Чтобы исключить противоречия в базе знаний, желательно, чтобы при вводе в неё новой информации система проверяла её непротиворечивость по отношению к текущему состоянию базы и осуществляла только ввод допустимых фактов.

Интерфейс с пользователем может быть организован с помощью стандартных средств: меню, форм для ввода/вывода, или же простых фраз естественного языка (в этом случае необходим синтаксический анализ запросов пользователя путём выделения в них ключевых слов). Для корректного вывода русских имён в ответах системы можно встроить в неё список имён, с указанием их именительного и родительного падежей (именно эти падежи используются в запросах указанных выше видов и ответах на них).

Укажем некоторые отношения родства, которые могут быть предствлены в системе: *зять* – муж дочери; *золовка* – сестра мужа; *свояченица* – сестра жены; *шурин* – брат жены; *деверь* – брат мужа; *сноха* – жена сына (для его отца) или жены двух братьев друг другу.

Программа синтаксического анализа предложений естественного языка

Предлагается рассмотреть подмножество грамматически правильных предложений одного из европейских языков (русский, английский, французский или др.). В общем случае предложения состоят из группы подлежащего (именной группы) и группы сказуемого (глагольной группы). Группа подлежащего в свою очередь состоит из нескольких прилагательных и/или числительных, существительного или местоимения, а также, возможно, артикля. Группа сказуемого включает глагол в одной из личных форм и нескольких дополнений или обстоятельств в виде именных групп с предлогом или без, например: *Маленький мальчик долго играл с большой собакой в саду*.

Необходимо описать структуру предложений с помощью расширенной контекстно-свободной грамматики [7, с. 234-255; 17, с. 203-210]. Грамматика должна учитывать:

- единственное и множественное число существительных и глаголов;
- возможность нескольких прилагательных в качестве определений существительных;
- употребление наречий при глаголах и прилагательных;
- согласование в лице и числе именной группы в роли подлежащего и глагольной группы в роли сказуемого;
- при необходимости – согласование составных частей самой именной группы и правильное употребление артиклей.

Построенная на базе указанной грамматики пролог-программа должна выполнять ***синтаксический анализ (разбор)*** поступающего на вход предложения естественного языка методом рекурсивного спуска [7, с. 234-255]. В результате для грамматически правильного предложения долно быть построено и визуализировано его дерево разбора; а для грамматически неверных предложений выданы диагностические сообщения. Если согласно грамматике возможно несколько альтернативных вариантов разбора исходного предложения, все деревья разбора должны быть построены и визуализированы.

Встроенный в программу словарь для анализируемого подмножества естественного языка должен включать не менее 25 единиц для каждой части речи (существительное, прилагательное,

глагол и т.п.) и допускать расширения. Синтаксический разбор следует реализовать достаточно эффективно, с использованием разностных списков, без использования предиката `append` и порождения лишних точек бектрекинга.

Усложнение рассмотренного варианта задания предполагает создание *системы автоматического перевода* с одного естественного языка на другой (например, с английского на французский). В этом случае результатом работы программы является не дерево разбора исходного предложения, а эквивалентное по смыслу предложение целевого естественного языка. Реализация такой программы проще, если брать пару родственных языков (русский – украинский, испанский – итальянский и т.п.). Кроме описывающей исходный язык формальной грамматики потребуется также набор продукций вида: *грамматическая конструкция исходного языка* $\rightarrow$ *конструкция целевого языка*.

Экспертная система на основе продукций

Основная функция ***экспертной системы*** (ЭС) – поиск решения задачи на основе заложенных в систему предметных знаний, а также ответов пользователя об особенностях решаемой задачи в ходе диалога с ним [13].

Разрабатываемая экспертная система определяет класс (тип) некоторого объекта по его наблюдаемым признакам (диагностирует, классифицирует его). Для этого система инициирует диалог с пользователем, в ходе которого запрашивает у него информацию о нужных признаках диагностируемого объекта, а затем на основе его ответов принимает решение, к какому классу принадлежит этот объект. К примеру, на основе внешних признаков музыкального инструмента (его размера, формы, наличия клавишей, струн, смычка и т.п.) система определяет вид музыкального инструмента (скрипка, балалайка, рояль и др.).

Основными модулями разрабатываемой экспертной системы являются [2, с. 414]:

- *база знаний*, состоящая из правил *продукций* (правил вывода) вида: *условие* $\rightarrow$ *следствие*; каждая продукция представляет некоторый фрагмент знания, необходимый для классификации объектов в рассматриваемой предметной области;

- *механизм вывода* (*решатель*), который осуществляет поиск решения, т.е. нужной (решающей) цепочки продукций, представляющей последовательные шаги заключения о классе (типе) объекта.

Полученное системой решение о классе объекта при необходимости может быть пояснено пользователю – это одна из отличительных особенностей ЭС. Для этого экспертная система должна уметь отвечать на вопросы вида *Как?* и *Почему?* Вопрос *Как?* (т.е. как получено указанное решение) может быть задан пользователем после получения решения, и в качестве ответа система визуализирует решающую цепочку продукций. Вопрос *Почему?* (т.е. почему запрашивается именно этот признак объекта) может быть задан в ходе диалога, и в ответ система показывает правило продукции, которое она пробует в текущий момент применить.

Вывод решения (поиск решающей цепочки) может производиться двумя путями [2, 4]:

- в прямом направлении – от известных фактов, т.е. наблюдаемых признаков объекта, к следствиям – заключениям о классе/подклассе объекта (так называемый *прямой вывод*);
- в обратном направлении – от гипотез о возможном классе/подклассе объекта к фактам, их подтверждающим (так называемый *обратный вывод*).

Поскольку обычно в процессе вывода выявляются недостающие для диагностики (классификации) факты, они запрашиваются у пользователя. Тем самым, ход диалога пользователя с системой определяется фактически механизмом вывода и набором правил базы знаний. В силу этого интерфейс с пользователем часто не выделяют в отдельный модуль, а относят к механизму вывода. По аналогичным причинам также часто не выделяют подсистему объяснения ЭС, реализующую ответы на вопросы пользователя. В целом, инициируемый системой диалог не должен содержать повторяющиеся или непонятные вопросы пользователю, число вопросов должно быть разумным.

Мощность построенной базы знаний должна позволять ЭС распознавать не менее 25 различных классов объектов по не менее 5-7 признакам. Важно, что база знаний ЭС должна допускать расширение, т.е. добавление описания нового объекта/класса (по зафиксированной системе признаков), что позволяет затем ЭС классифицировать его по указанным признакам.

В качестве предметной области (области экспертизы) ЭС может быть взята классификация (диагностика):

- плодов фруктов или ягод (признаки: цвет, форма, вкус, количество косточек и др.);
- грибов (признаки: цвет и форма шляпки, толщина ножки, места появления и др.);
- пород собак (признаки: вид и длина шерсти, окрас, размер и тип ушей и др.);
- а также классификация транспорта, оружия, бабочек, автомобилей, напитков и т.п.

Программа генерации геометрических головоломок

В головоломках рассматриваемого вида требуется найти закономерность в наборе из нескольких *составных геометрических фигур*. Такие фигуры могут состоять из нескольких простых фигур и их частей, например: точка, квадрат, треугольник, прямоугольник, круг, дуга, прямые и ломаные отрезки линий и др. Составляющие фигуры могут быть соединены различным образом: могут быть вложены друг в друга, могут пересекаться или соприкасаться. Кроме того, их контуры могут быть разного цвета, а их внутренняя часть – заштрихованной или цветной.

Примеры составных фигур показаны на Рис. 25. Возможны и более сложные фигуры – например, схематические изображения человечков или животных.

Кроме обнаружения закономерности в заданном наборе фигур головоломка предполагает также определение некоторой недостающей фигуры. Например, в головоломке, приведенной на Рис. 25, требуется по закономерности построения двух первых (левых) рядов из трех фигур найти фигуру, обозначенную вопросительным знаком и завершающую последний ряд фигур согласно найденной закономерности. Искомая фигура содержится среди *вариантов ответа* – нескольких пронумерованных составных фигур (на рисунке показаны справа), которые также включаются в головоломку. Среди вариантов ответа всегда есть правильный; для показанной на рисунке головоломки правильным ответом является фигура 6. В общем случае возможны несколько правильных ответов, тогда их все целесообразно включить в головоломку в качестве вариантов ответа.

Кроме головоломки, показанной на Рис. 25, возможны другие виды геометрических головоломок [17, с. 183].

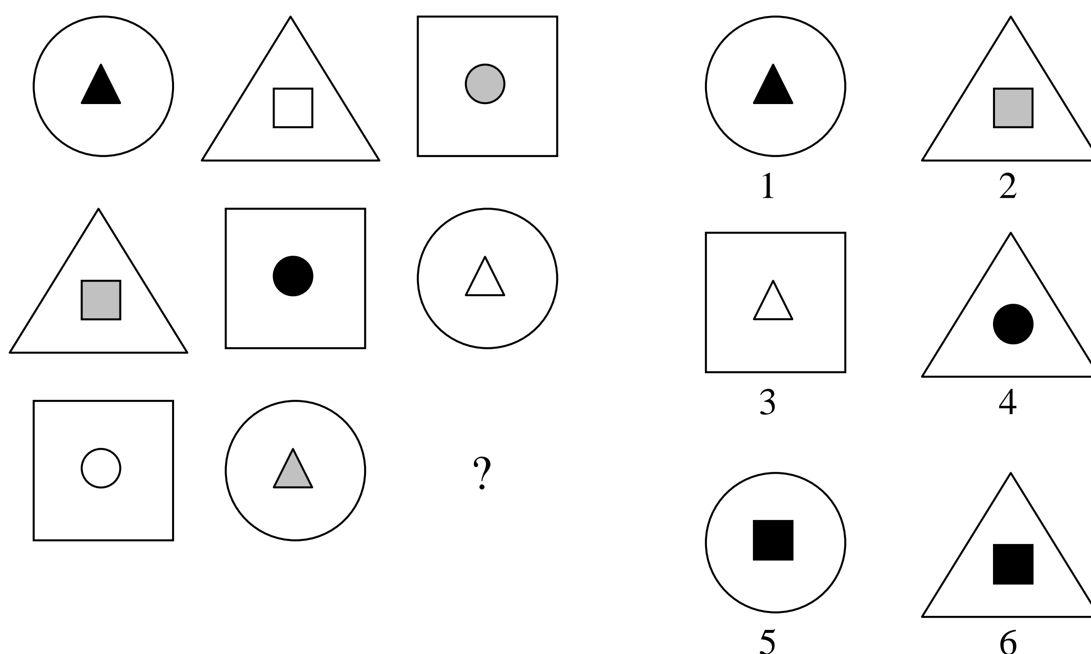

Рис. 25. Геометрическая головоломка

Программа генерации геометрических головоломок должна строить набор составных фигур без некоторой недостающей фигуры, а также набор вариантов ответа, включающий эту недостающую фигуру. Построенные наборы должны быть показаны пользователю, и он имеет возможность:

- либо запросить решение головоломки, если он затрудняется решить её сам, и тогда ему показывается её правильное решение (или несколько решений, если такое возможно);
- либо самостоятельно решить головоломку – при этом проверяется правильность его решения с последующей выдачей ему нужного сообщения; далее в случае неверного решения ему показывается правильное решение (решения), а в случае верного решения ему могут быть показаны другие возможные решения (если таковые есть).

Важно, чтобы сгенерированные программой составные фигуры, предлагаемые как варианты ответа на головоломку, были в достаточной степени похожи на фигуру, представляющую правильный ответ (иначе решение головоломки существенно упрощается). Для этого можно брать в качестве вариантов ответа

фигуры, отличающиеся всего одним-двумя элементами от фигуры, являющейся решением головоломки.

Сложность головоломки, как и возможное их количество, зависит от числа возможных параметров-элементов, образующих составные геометрические фигуры. Необходимо предусмотреть в генерируемых составных фигурах не менее 5-7 различных типов параметров-элементов: простых фигур, их взаимного расположения, а также их закраски, включая цвет контура и штриховку/заливку. Целесообразно ввести несколько уровней сложности генерируемых головоломок, в зависимости от используемого количества этих параметров-элементов в фигурах – с тем, чтобы пользователь мог устанавливать нужный ему уровень.

Требуется предусмотреть возможность решения пользователем за сеанс работы с программой нескольких головоломок – для этого программа при каждом обращении к ней должна порождать разные задачи, причём в разные сеансы последовательности генерируемых задач должны быть разными. Чтобы этого достичь, следует в некоторые моменты процесса генерации задачи производить случайный выбор как параметров-элементов, определяющих составную фигуру, используя для этого датчик случайных величин и доступное программе значение, заранее неизвестное: дату и/или текущее время.

Пользовательский интерфейс должен быть понятным, а для этого он должен включать необходимые подсказки о допустимых в текущий момент действиях.

Вопросно-ответная система по онтологии предметной области

Центральным модулем системы должна быть ***база знаний***, в которой представлена онтологическая информация об объектах некоторой предметной области и их классах. ***Онтология предметной области*** представляет собой инвентарь сущностей (объектов и понятий) этой области, их описаний и описаний их связей [11]. Объекты онтологии обычно объединяются в *классы*. К числу основных связей, представленных в онтологиях, относятся связь *род-вид* (класс-подкласс), связь *часть-целое*, связь *класс-экземпляр класса* (класс-объект).

Онтологии, фиксирующие только отношение род-вид, называются *таксономиями*. Классической таксономией является

система классов в биологии – растений и животных; в виде таксономии могут быть представлены знания о музыкальных инструментах, пищевых продуктах и напитках, оружии и т.п.

Объекты любого класса характеризуются определенным набором *атрибутов* (свойств) и их значений, причем атрибуты класса часто наследуются объектами подклассов, и потому такие наследуемые атрибуты, как правило, описываются в самом общем классе цепочки наследования. Наследование свойств происходит обычно по связям *род-вид* (класс-подкласс) и *класс-экземпляр*, и в ряде случаев – по связи *часть-целое*.

Многие атрибуты объектов задаются явно своими значениями (например, цвет животного), значения других атрибутов могут вычисляться по значениям других атрибутов. Для некоторых атрибутов в некоторых классах может быть установлено специальное *значение по умолчанию* – обычно это стандартное значение для объектов описываемого класса (например, по умолчанию, птицы умеют летать).

Функции разрабатываемой вопросно-ответной системы включают:

- Вычисление ответов на запросы следующих видов:
  - для конкретного объекта/класса онтологии указать его атрибуты (свойства);
  - определить значение нужного свойства у конкретного объекта/класса;
  - для двух заданных объектов/классов определить отличающие их свойства.
- Расширение в диалоге с пользователем базы знаний: введение в неё новых объектов и классов и/или коррекция старой информации.

Мощность итоговой базы знаний должна включать не менее 15 различных классов объектов, различающихся по не менее 5-7 признакам.

Вопросно-ответная система должна обладать возможностями *логического вывода*, который позволяет вычислять значения атрибутов (свойств) объектов и классов, которые не представлены явно в их описаниях. Необходимо реализовать две различные *стратегии наследования атрибутов*, различающиеся порядком движения по цепочке подкласс-класс и рассмотрения свойств по

умолчанию [23, с. 255-262]. При первой стратегии сначала делается попытка найти или отнаследовать нужное свойство по цепочке классов наследования, без использования значений по умолчанию. При второй стратегии находится самое ближайшее, представленное в цепочке классов наследования значение нужного свойства, включая значение по умолчанию.

Усложнение рассмотренного варианта задания предполагает реализацию *множественного наследования*, которое характерно для ряда предметных областей и требует разработки дополнительных стратегий логического вывода.

Пользовательский интерфейс вопросно-ответной системы может быть организован с помощью меню, форматов для ввода/вывода, или же простых фраз естественного языка. В последнем случае требуется проведение частичного синтаксического анализа запросов пользователя путем выделения в них ключевых слов.

Программа построения лабиринта

Рассматриваются лабиринты, расположенные в прямоугольной области $M \times N$ и состоящие из стенок внутри и на границе этой области, причем стенки являются сторонами квадратов (клеток) покрывающей эту область равномерной сетки. Каждый лабиринт имеет один вход на одной из сторон прямоугольной области и один выход на противоположной стороне – см. Рис. 26. Такой лабиринт может быть получен из равномерной сетки стенок в результате выбивания ровно двух граничных стенок на противоположных сторонах рассматриваемого прямоугольника и удаления некоторого количества внутренних стенок .

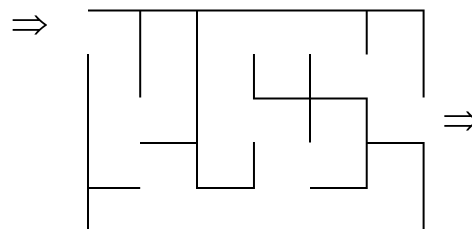

Рис. 26. Пример построенного лабиринта

Лабиринт проходим, если внутри него между стенками существует путь, соединяющий вход и выход. Будем говорить в этом случае, что лабиринт имеет решение. *Решение единственно*, если среди всех таких путей есть минимальный по длине путь, являющийся частью всех остальных путей. *Клетка* лабиринта *достижима*, если существует путь, соединяющий её с входом или выходом. Тот факт, что все клетки лабиринта достижимы, означает, что в лабиринте отсутствуют замкнутые области из клеток.

Требуемая пролог-программа должна по заданным M и N ($5 \leq M \leq 30,\ 5 \leq N \leq 40$) строить прямоугольный лабиринт, который имеет единственное решение и состоит только из достижимых клеток. Программа визуализирует построенный лабиринт, отмечает в нём вход и выход и по указанию пользователя высвечивает путь между ними. Желательно, чтобы построенный лабиринт был интересным – содержал достаточно большое число внутренних стенок и почти замкнутых комнат (областей), чтобы путь в нём был достаточно извилистым.

Программа должна уметь строить в одном сеансе несколько лабиринтов разных размеров, причем лабиринты одной и той же размерности должны быть разными. Для этого необходимо при порождении лабиринтов использовать случайные величины. Лабиринты больших размеров (порядка 500 клеток и больше) должны строиться за приемлемое время.

Заметим, что стратегии генерации лабиринта могут быть различны: удаление внутренних стенок прямоугольной области или же их добавление; при этом можно сначала построить путь, а затем достроить оставшуюся часть лабиринта. При любой стратегии единственность решения (пути), как и достижимость клеток лабиринта, целесообразно проверять как можно раньше, либо же гарантировать выполнение этих требований самим методом построения лабиринта.

Компьютерная игра

Предлагается выбрать для реализации игру из класса ***игр двух лиц*** (игроков) ***с полной информацией*** [2, 3, 11] – к этому классу относятся, к примеру, шахматы и шашки. Игра должна быть достаточно сложной, т.е. исключающая (или сильно затрудняющая) практическую возможность полного просмотра дерева игры и обнаружения выигрышной стратегии (если таковая существует).

С другой стороны, игра не должны быть слишком сложной, с сильно ветвистым деревом игры (как, например, полная игра в шашки или шахматы). Рекомендуется выбирать шашки без дамок, калах, шахматный эндшпиль, реверси, крестики-нолики на неограниченной доске и т.п.

В играх двух лиц с полной информацией для выбора компьютером очередного хода используется так называемая *статическая оценочная функция*, оценивающая позицию игры как таковую без учёта её продолжений. Также применяется *минимаксная процедура* поиска достаточно хорошего хода от заданной игровой позиции, а чаще – её более эффективная модификация, известная как ***альфа-бета процедура***.

Альфа-бета процедура основана на частичном просмотре возможных продолжений игры на заданное количество D ходов игроков, т.е. просмотре дерева игры от заданной игровой позиции на глубину дерева D, и оценки возможных игровых позиций с помощью статической оценочной функции. В отличие от минимаксной процедуры, решающей ту же задачу, альфа-бета процедура выполняет одновременный просмотр дерева и оценку его вершин, что позволяет сократить часть работы.

В реализуемой компьютерной игре необходимо предусмотреть возможность изменения (перед началом игры) *глубины просмотра дерева игры* альфа-бета процедурой: $2 \leq D \leq 5$, шаг глубины соответствует ходу одного игрока. Просмотр дерева игры на 2-3 хода вперед должен выполняться за приемлемое время. Целесообразно реализовать случайный выбор хода из нескольких равноценных (чтобы с программой было интересно играть).

Игровая программа должна уметь играть как против человека (пользователя), так и против самой себя или другой подобной игровой программы. Пользовательский интерфейс должен быть понятным и удобным и включать установку уровня сложности игры (глубины просмотра дерева игры), визуализацию текущей позиции игры, показ предыдущих позиций.

Программа построения головоломок крисс-кросс

Рассматриваемая головоломка крисс-кросс похожа на кроссворд, но предлагает задачу более простую. Для заданного набора слов и заданной схемы, подобной сетке кроссворда, требуется вписать в схему все слова из набора. Схема состоит из пересекающихся, но не соприкасающихся линий (столбцов и строк) из клеток; схема должна быть *связной*, т.е. каждая линия должна иметь хотя бы одно пересечение с другой – см. Рис 27.

Программа составления головоломки, получая на вход набор из 7-12 различных слов одного из естественных языков (русского или английского) должна построить подходящую для них схему или несколько схем. *Схема построена правильно*, если она является связной и все слова набора могут быть вписаны в нее (после чего не остается пустых клеток), причем возможен лишь один вариант заполнения схемы (головоломка имеет единственное решение).

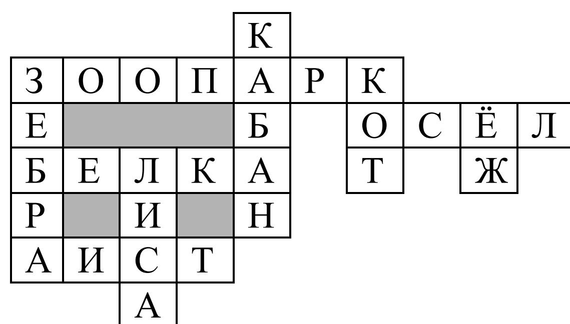

Рис. 27. Пример заполненной схемы в головоломке крисс-кросс

Если для входного набора слов не существует решения-схемы, то программа должна сообщить об этом, указав по возможности причину неудачи (например, наличие в наборе слова, не имеющего общих букв с другими словами). В остальных случаях построенная схема должна быть изображена на экране компьютера, и по запросу пользователя может быть показано решение головоломки, т.е. заполнение схемы исходными словами.

В случаях, когда для заданного набора слов возможно несколько схем, предпочтение следует отдавать более компактным и связным схемам. Связность схем зависит от количества пересечений в ней; компактность схемы определяется площадью наименьшего объемлющего ее прямоугольника. Эти критерии оценки схемы

рекомендуется формализовать тем или иным способом (в виде эвристической функции) и использовать при построении правильных схем.

Отметим, что длина слова из набора и количество слов одинаковой длины могут быть ключом как к разгадке самой головоломки, так и к написанию программы их составления. Эффективность перебора вариантов схем при построении правильной схемы может зависеть от порядка, в каком рассматриваются слова исходного набора. В любом случае при построении схемы возникает существенный перебор, и имеет смысл применять эвристическое отсечение некоторых вариантов частично построенных схем, поскольку в ином случае построение всей схемы может оказаться слишком длительным (программа должна строить схему для 10-12 слов за приемлемое время). По этой же причине проверку условия единственности решения головоломки целесообразно проводить как можно раньше в ходе построения схемы.

Система составления учебного расписания

Назначение системы – составление недельного расписания занятий для курса, включающего N ($7 \le N \le 10$) студенческих групп. Исходной информацией для составления расписания являются:

- учебный план курса, определяющий названия изучаемых предметов (не более 7 разных), и количество учебных занятий в неделю для изучения каждого предмета (не более 5 занятий в неделю);
- список-распределение преподавателей по группам, в котором для каждой учебной группы и каждого изучаемого предмета указывается фамилия преподавателя, который будет вести этот предмет в этой группе.

Известно, что суммарное количество учебных занятий по учебному плану не превышает 17 занятий в неделю, а недельная нагрузка каждого преподавателя не превышает 8 занятий.

При генерации расписания для студенческих групп желательно более или менее равномерное распределение занятий по дням недели (от понедельника до субботы включительно). Оптимальным следует считать 2-3 занятия в день. В расписании должны быть учтены следующие требования:

- В каждый день недели у учебной группы не может быть более четырех занятий, а у преподавателя – не более трех занятий;
- В любой день недели в расписании группы не может быть больше одного «окна» (перерыва) между занятиями, причём его протяженность – не более чем одно занятие;
- В недельном расписании нагрузки преподавателя не должно быть более одного дня лишь с одним занятием, а «окна» между занятиями в каждый учебный день в сумме не должны быть больше 2 занятий.

В построенном системой учебном расписании для каждого занятия любого дня недели должны быть определены: номер и время начала занятия, предмет, фамилия преподавателя. Возможные номера занятий (от 1 до 6) задают порядок следования занятий в течение дня.

Пользовательский интерфейс системы должен быть удобен для просмотра исходных данных (учебный план, список-распределение преподавателей) и построенного расписания учебных групп, а также определяемого этим расписанием индивидуального расписания каждого преподавателя. Это расписание для каждого дня указывает, есть ли занятия в этот день, и если есть, то их номер и время, предмет, номер учебной группы.

Учитывая сложность составления расписания, удовлетворяющего всем требованиям, целесообразно предусмотреть возможность ослабления перед началом его генерации некоторых требований (например, ограничений на количество занятий в день).

Программа разработки маршрутов транспортных перевозок

Рассматривается сеть одноколейных железнодорожных путей, соединяющих N городов ($5 \le N \le 12$); известны длины всех путей (дорог) в километрах. Задан список из M заявок ($3 \le M \le 9$) на грузоперевозки по этой железной дороге в течение текущих суток. Каждая заявка включает в общем случае следующую информацию:

- пункт (город) отправления грузового состава;
- пункт (город) назначения грузового состава;
- максимальную скорость движения грузового состава по путям (не более 70 км в час);
- час суток, не позднее которого груз должен прибыть в пункт назначения.

Требуемая пролог-программа должна составить, исходя из заданного списка заявок, расписание движения грузовых составов на текущие сутки. Расписание включает *M* маршрутов; каждый маршрут соответствует заявке и фиксирует кроме пунктов (городов) отправления и назначения следующее:

- время (в часах) отправления из исходного пункта;
- время (в часах) прибытия в конечный пункт;
- промежуточные пункты, через которые проходит маршрут, и время остановки грузового состава в них (в тех пунктах, где остановка делается);
- скорость движения на каждом участке пути между двумя городами, входящими в маршрут.

Считается, что скорость движения грузового состава на любом участке пути постоянна. Она может быть разной для разных составов, но не выше указанной в заявке и не ниже 20 км в час. Каждый грузовой состав может делать остановки на промежуточных пунктах маршрута, но они должны быть ограничены по длительности.

Дополнительное требование, предъявляемое к составлению маршрутов перевозок – протяжённость каждого маршрута между заданными городами отправления и назначения не должна превышать минимальное расстояние между ними по этой железной дороге более чем в 1,5 раза.

Пользовательский интерфейс должен допускать просмотр построенного расписания маршрутов и списка исходных заявок. Желательно визуализировать карту дорог, на которой можно отметить найденные маршруты или же в динамике показать прохождение грузовых составов по этим маршрутам в течение суток, выбрав подходящий шаг времени для пересчёта местонахождения составов.

5 Литература

1. Адаменко А.Н., Кучуков А.М. Логическое программирование и Visual Prolog. – СПб.: БХВ-Петербург, 2003.
2. Братко И. Программирование на языке Пролог для искусственного интеллекта – М.: Мир, 1990.
3. Братко И. Алгоритмы искусственного интеллекта на языке PROLOG. – М., Вильямс, 2004.
4. Ин Ц., Соломон Д. Использование Турбо-Пролога – М.: Мир, 1993.
5. Ильинский Н.И., Козинцев И.Б. Язык Пролог и методы его реализации // Искусственный интеллект – В 3-х кн. Кн. 3. Программные и аппаратные средства: Справочник./ Под ред. В.Н. Захарова, В.Ф. Хорошевского – М.: Радио и связь, 1990, с.33-47.
6. Кларк К., Маккейб Ф. Введение в логическое программирование на микро-прологе – М.: Радио и связь, 1987.
7. Клоксин У., Меллиш К. Программирование на языке Пролог – М.: Мир, 1987.
8. Ковальски Р. Логическое программирование // Логическое программирование: Пер. с англ. и фр. – М.: Мир, 1988, с.134-166.
9. Ковальски Р. Логика в решении проблем: пер. с англ. – М.: Наука, 1990. – 280 с.
10. Колмероэ А., Кануи А., ванн Канеген М. Пролог – теоретические основы и современное развитие // Логическое программирование: Пер. с англ. и фр. – М.: Мир, 1988, с.27-133.
11. Люгер Дж. Искусственный интеллект: стратегии и методы решения сложных проблем. М., Вильямс, 2003.
12. Малпас Дж. Реляционный язык Пролог и его применение – М.: Наука, 1990.
13. Марселлус Д. Программирование экспертных систем на Турбо-Прологе – М.: Финансы и статистика, 1994.
14. Метадикес Г., Нероуд А. Принципы логики и логического программирования – М.: Факториал, 1998.
15. Набебин А. А. Логика и Пролог в дискретной математике – М.: Издательство МЭИ, 1996.

16. Робинсон Дж. А. Машинно-ориентированная логика, основанная на принципе резолюций // Кибернетический сборник, Вып. 7, 1970.
17. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог – М.: Мир, 1990.
18. Шрайнер П.А. Основы программирования на языке Пролог: курс лекций. Учеб. пособие – М.: Интернет-университет информационных технологий, 2005.
19. Янсон А. Турбо-Пролог в сжатом изложении. – М.: Мир, 1991.
20. Colmeraurer A., Kanoui H., Pasero R. Un systeme de communication homme-machine en francais. Raport CRI 72-18, Groupe Intelligence Artificielle, Universite Aux – Marseille II, 1973.
21. SWI Prolog \\ www.swi-prolog.org оформить по правилам
22. Warren D. Implementing Prolog – compiling predicate logic programs. – D. A. I. Research Report N 39, 40. Edinburgh: University of Edinburgh, 1977.
23. Winston P. Artificial Intelligence (Third Edition). Addison-Wesley Publishing Co., 1993.
24. XPCE Documantation оформить по правилам http://www.swi-prolog.org/download/xpce/doc/userguide/userguide.pdf

Приложение: Встроенные предикаты языка Пролог

append
assert
atom
atomic
bagof
compound
consult
dif
fail
findall
float
get0
get0
integer
is
length
member
name
nl
nonvar
number
put
read
reconsult
repeat
retract
reverse
see
seen
setof
tab
tell
told
var
write