Новосибирский государственный технический университет
Кафедра вычислительной техники

Ю. В. Новицкая

Основы логического и функционального программирования

Учебное пособие

Новосибирск
2004 г.

Основы логического программирования

Введение

Рассмотрим пример программы на Prolog'е:

Например, необходимо выявить все объекты, имеющие свойство быть птицей.

Предположим, имеются следующие исходные данные:

- журавль – это птица;
- у синицы есть крылья;
- синица умеет летать;
- у пингвина есть крылья;
- пингвин умеет плавать;
- некто является птицей при условии, что у него есть крылья и он умеет летать.

Первые пять предложений будем считать не подлежащими сомнению и назовем фактами, шестое предложение – правилом вывода, так как это предложение формулирует правило, по которому можно сделать вывод о том, является ли некто птицей или нет. Некто будет птицей при условии, что он умеет летать и у него есть крылья. Условное обозначение :- читается: «при условии» или «если». Так как условия «умеет летать» и «есть крылья» перечислены через запятую, они должны быть выполнены оба. Запятая означает операцию «логическое И». Все предложения должны заканчиваться точкой.

Программа на языке Prolog будет выглядеть следующим образом:

```
птица (журавль).
есть_крылья (синица).
умеет_летать (синица).
есть_крылья (пингвин).
умеет_плавать (пингвин).
птица (Объект):- есть_крылья (Объект), умеет_летать (Объект).
```

Следует оговорить, что программа записана не по всем правилам, но для первого знакомства с языком Prolog вполне можно допустить несоблюдение некоторых правил. В разделе «Основные стандартные домены» этот пример будет приведен с соблюдением всех правил синтаксиса Prolog'а.

Теперь к программе можно обращаться с вопросами (запросами):

1. Кто является птицей? (Ответы: журавль, синица)
2. Кто умеет летать? (Ответ: синица)
3. У кого есть крылья? (Ответы: синица, пингвин)
4. и т.д.

Рассмотрим, каким образом работает программа. Предположим, нужно найти ответ на первый вопрос – кто является птицей? Вопрос будет иметь такой вид:

```
Goal: птица (Кто).
```

Кто – это имя переменной, которая в начале работы программы не имеет никакого значения.

Каким образом будет находиться первое решение? Будут последовательно просматриваться все строки программы и первая же строка (первый факт) дает первое решение – журавль.

Но язык Prolog своеобразен и существенно отличается от других языков программирования. В частности, если у задачи есть несколько решений, они все будут найдены. В рассматриваемом примере есть еще возможные решения, поэтому выполнение программы не прекращается после нахождения первого решения.

Для нахождения следующих возможных решений продолжается просмотр строк программы и обнаруживается правило вывода с подходящей для нахождения следующего решения левой частью. Переменные Кто и Объект начинают обозначать один и тот же объект, то есть, как только переменная Объект получит какое-нибудь значение, то же самое значение сразу же получит переменная Кто.

Теперь, для того, чтобы правило вывода дало второе решение, необходимо, чтобы были выполнены все условия, записанные в его правой части. Первое условие выполнится, если будет найден объект, у которого есть крылья. Другими словами, в тексте программы нужно найти факт, говорящий об этом. При просмотре программы с самого начала такой факт обнаруживается – у синицы есть крылья. Переменная Объект немедленно принимает значение «синица». Проверка выполнения второго условия начинается с учетом того, что переменная Объект уже имеет значение «синица», то есть второе условие в правиле вывода имеет вид – умеет_летать (синица). Но второе условие еще нельзя считать выполненным. Недостаточно того, что переменная Объект приняла значение «синица», нужно найти факт, подтверждающий, что синица действительно умеет летать. А для этого вновь просмотреть предложения программы с самого начала. Факт находится. Найдено второе решение «синица». Больше решений обнаружить не удастся, так как больше нет фактов, описывающий птиц, у которых есть крылья и которые умеют летать.

Итак, полученные решения:

1. журавль (в соответствии с фактом)
2. синица (по правилу вывода)

Как видно в рассмотренном примере, основой для находимых решений являются факты, записанные в тексте программы. Если дописать в программу еще факты, например:

```
есть_крылья (самолет).
умеет_летать (самолет).
```

будет найдено третье решение – самолет.

Предложения: факты и правила

Программа на языке Prolog состоит из предложений, которые можно разделить на две группы: факты и правила вывода.

В виде фактов в программе записываются данные, которые принимаются за истину и не требуют доказательства. Данные в фактах могут быть использованы для логического вывода. Факт может описывать некоторые свойства объекта или отношения между объектами. Можно дать следующее определение для факта: факт – это свойство объекта или отношение между объектами, для которого известно, что они истинны.

Примеры фактов:

```
%объект кот обладает свойством – черным цветом или, проще, кот – черный
black (cat).

%города Новосибирск и Омск связаны железной дорогой
railway (novosibirsk, omsk).
```

Второй тип предложений – правила вывода. Правило вывода состоит из двух частей, разделенных условным обозначением :- , которое читается как «если», или «при условии, что». Левая часть правила вывода называется заголовком или головной целью. Правая часть правила вывода называется хвостом или хвостовой частью. Хвостовая часть может состоять из нескольких условий (хвостовых целей), перечисленных через запятую или точку с запятой. Запятая означает операцию «логическое И», точка с запятой – операцию «логическое ИЛИ». Использовать скобки в хвостовой части правила вывода нельзя.

Головная цель правила вывода считается доказанной, если доказаны все хвостовые цели в правой части правила вывода.

Пример правил вывода:

```
%Некоторый объект, обозначенный переменной Bird, является пингвином,
%если объект умеет плавать и у него есть крылья
penguin (Bird):- swim (Bird), wings(Bird).

%Некоторый объект, обозначенный переменной Bird, является страусом,
%если объект не умеет летать, у него длинные ноги и есть крылья
ostrich (Bird):- not_fly (Bird), long_legs (Bird), wings(Bird).
```

В предложениях используются переменные для обобщенной формулировки правил вывода. Все предложения обязательно заканчиваются точкой.

Запросы

Для того чтобы программа, написанная на языке Prolog, начала работу, к ней нужно обратиться с запросом. Запросы могут быть двух видов: внутренние и внешние. Внутренние запросы записываются непосредственно в тексте программы, для использования внешних запросов нужна программная оболочка.

Вне зависимости от того, являются запросы внутренними или внешними, выглядят они одинаково. Запрос может быть простым (состоящим из одной цели) или составным (состоящим из нескольких целей). Выполнение программы заключается в доказательстве целей, входящих в запрос. Программа считается успешно выполненной (завершенной), если доказаны цели, из которых состоит запрос.

Предикаты

Предикат – это имя свойства или отношения между объектами с последовательностью аргументов.

При записи факта или цели с использованием некоторого предиката сначала записывается имя предиката, а затем в скобках, через запятую, его аргументы.

Например, факт black (cat). записан с использованием предиката black, имеющего один аргумент. Факт railway (novosibirsk, omsk). записан с использованием предиката railway, имеющего два аргумента.

Число аргументов предиката называется арностью предиката и обозначается black/1 (предикат black имеет один аргумент, его арность равна единице). Предикаты могут не иметь аргументов, арность таких предикатов равна 0.

Переменные

Работа с переменными в Prolog’е достаточна своеобразна. Если в других, алгоритмических, языках программирования значение переменной, которое было ей присвоено, не изменяется до тех пор, пока не будет выполнено переприсваивание значения, то в Prolog’е переменная может получить некоторое значение в процессе поиска решения и потерять его, когда начнется поиск нового решения.

Имя переменной дается по следующим правилам: имя должно начинаться с заглавной латинской буквы или символа подчеркивания, после которых могут следовать латинской буквы, цифры или символы подчеркивания.

Пример:

```
First_list
X
Person
City
```

Переменная, не имеющая значения, называется свободной, переменная, имеющая значение – конкретизированной.

В Prolog’е нет оператора присваивания, его роль выполняет оператор равенства =.

Если записать следующую цель:

…, X=5, …

то как эта цель будет рассматриваться, как сравнение или как присваивание, все зависит от того, получила ли какое-либо значение переменная X к моменту доказательства этой цели. Если переменная X имеет значение (например, равное 6), то оператор равенства = работает как оператор сравнения. Если же переменная X свободна (не имеет никакого значения), то оператор равенства = работает как оператор присваивания. При этом совершенно неважно, слева или справа от знака равенства находится имя переменной. Главное, чтобы она была свободной. С точки зрения программы на Prolog’е следующие две строки совершенно одинаковы:

…, X=5, …
…, 5=X, …

Самое важное, чтобы переменная X не имела значения. Из вышесказанного вытекает следующая особенность использования переменных в Prolog’е, нельзя записывать вот так:

…, X=X+5, …

В любом случае такая цель будет ошибочной. Действительно, если переменная X имеет, например, значение равное 10, то предыдущая цель сводится к доказательству цели:

…, 10=10+5, …

что, естественно, доказать нельзя.

Если же переменная X свободна, то нельзя к переменной, не имеющей никакого значения, прибавить 5, и присвоить эту неопределенность той же самой переменной. Как же тогда быть, если нужно изменить значение переменной? Ответ один – использовать новое имя, поскольку переменная, которая появляется в тексте программы впервые, считается свободной, и может быть конкретизирована некоторым значением:

…, Y=X+5, …

При этом опять же не важен порядок записи. Такая цель также будет правильной, и будет выполнять присваивание (конечно, если переменная X конкретизирована некоторым числом):

…, X+5=Y, …

Еще существуют специальные переменные, называемые анонимными. Их имя состоит только из знака подчеркивания. Анонимные переменные используются в случаях, когда значение переменной несущественно, но переменная должна быть использована.

Рассмотрим пример:

```
parent (“Владимир”, “Михаил”).
parent (“Владимир”, “Светлана”).
parent (“Анна”, “Михаил”).
parent (“Анна ”, “ Светлана”).
```

Факты описывают родителей и их детей (первый аргумент – имя родителя, второй – имя ребенка). Теперь, если нужно узнать только имена родителей, но не нужны имена детей, к программе можно обратиться с внешним запросом, использовав анонимную переменную:

```
Goal: parent (Person, _).
```

Решение в данном случае будет избыточным, поскольку есть четыре факта.

```
Person=Владимир
Person=Владимир
Person=Анна
Person=Анна
4 Solutions
```

Сравните, если использовать запрос:

```
Goal: parent (Person, Child).
```

какими будут результаты:

```
Person=Владимир, Child=Михаил
Person=Владимир, Child=Светлана
Person=Анна, Child=Михаил
Person=Анна, Child=Светлана
4 Solutions
```

Основные секции программы

Как правило, программа на Prolog'е состоит из нескольких секций, ни одна из которых не является обязательной. Вот основные секции:

1. DOMAINS – секция описания доменов (типов). Секция применяется только, если в программе используются нестандартные домены.
2. PREDICATES – секция описания предикатов. Секция применяется, если в программе используются нестандартные предикаты.
3. CLAUSES – секция предложений. Именно в этой секции записываются предложения: факты и правила вывода.
4. GOAL – секция цели. В этой секции записывается внутренний запрос.

На первый взгляд, без секций DOMAINS, PREDICATES и GOAL действительно можно обойтись, но как написать программу без секции CLAUSES? Конечно, такая программа не обладает большим количеством возможностей, но принципиально такую программу написать можно. Например:

```
GOAL
write ("Введите Ваше имя: "), readln (Name), write ("Здравствуйте, ", Name, "!").
```

Вот пример программы, состоящей только из секции GOAL. Используются только стандартные домены, следовательно, отпадает необходимость использовать секцию DOMAINS, нет нестандартных предикатов, следовательно, отпадает необходимость использовать секцию PREDICATES, и, наконец, все цели записаны непосредственно в секции GOAL, следовательно, нет необходимости использовать секцию CLAUSES.

Основные стандартные домены

Доменом в Prolog'е называют тип данных. В Prolog'е, как и других языках программирования, существует несколько стандартных доменов, перечислим их:

1. integer – целые числа.
2. real – вещественные числа.
3. string – строки (любая последовательность символов, заключенная в кавычки).
4. char – одиночный символ, заключенный в апострофы.
5. symbol – последовательность латинских букв, цифр и символов подчеркивания, начинающаяся с маленькой буквы или любая последовательность символов, заключенная в кавычки.

Для примера приведем программу из введения, оформленную по всем правилам:

```
PREDICATES
bird (string)
has_wings (string)
can_fly (string)
can_swim (string)

CLAUSES
bird ("журавль").
```

```
has_wings (“синица”).
has_wings (“пингвин”).
can_fly (“синица”).
can_swim (“пингвин”).
bird (Object):- has_wings (Object), can_fly (Object).

GOAL
bird (Who).
```

Несколько замечаний. Поскольку в программе не использовались нестандартные домены, не было необходимости использовать секцию описания доменов DOMAINS. В отличие от примера из введения, где был использован внешний запрос, в данной программе запрос записан в секции GOAL, то есть является внутренним. В таком случае находится только первое решение. Как находить все существующие решения, если используется внутренний запрос, более подробно описано в разделе «Управление поиском с возвратом: предикаты ! и fail»

Поиск с возвратом

Поиск с возвратом (backtracking) – это один из основных приемов поиска решений поставленной задачи в Prolog’е.

Каким образом работает поиск с возвратом? Это достаточно хорошо можно пояснить, вот на каком примере.

Предположим, для достижения некоторой цели человеку необходимо последовательно принять несколько решений и выполнить некоторые действия в соответствии с принятыми решениями. Первоначально человек без колебаний и раздумий принимает несколько решений, но при решении очередной проблемы у него возникают сомнения, поскольку возможных решений, предположим, имеется два, и человеку они кажутся одинаково правильными. Какое-либо из двух решений человек все равно принимает (но запоминает, в какой момент он сомневался, и какое из двух решений все же выбрал) и продолжает свое движение к поставленной цели.

Но, в какой-то момент оказывается, что решение, выбранное из двух, все же оказалось неправильным. Тогда человек вернется в точку принятия неверного решения, и пойдет по альтернативному пути. Не факт, что вновь выбранный путь окажется правильным, но человек попробует все возможные варианты нахождения решения.

Еще одна аналогия. Поиск с возвратом можно сравнить с поиском выхода из лабиринта. Нужно войти в лабиринт и на каждой развилке сворачивать влево, до тех пор, пока не найдется выход или тупик. Если впереди оказался тупик, нужно вернуться к последней развилке и свернуть направо, затем снова проверять все левые пути.

В конце концов, выход (если он есть) будет найден. Подобным образом работаем и механизм поиска с возвратом в языке Prolog.

Благодаря механизму поиска с возвратом Prolog в состоянии находить все возможные решения, имеющиеся для данной задачи.

Рассмотрим на примере, каким образом выполняется поиск всех возможных решений с применением поиска с возвратом.

```
PREDICATES
little (symbol)
middle (symbol)
big (symbol)
strong (symbol)
powerful (symbol)
```

```
CLAUSES
little (cat).
little (wolf).
middle (tiger).
middle (bear).
big (elephant).
big (hippopotamus).
strong (tiger).
powerful (Animal):- middle (Animal), strong (Animal).
powerful (Animal):- big (Animal).
```

Итак, обратимся к программе с запросом – какое животное можно назвать мощным?

Запрос будет выглядеть следующим образом:

```
Goal: powerful (Animal).
```

Проследим по шагам, каким образом будут находиться все возможные решения.

Доказательство цели, сформулированной в запросе, начинается с последовательного просмотра всех предложений, имеющихся в тексте программы. В данном примере цель powerful (Animal) может быть сопоставлена с заголовком первого правила вывода, что и происходит, но при этом помечается, что в тексте программы имеется еще одно правило точно с таким же заголовком, то есть устанавливается первая точка возврата (назовем ее *1).

Так как было выбрано первое правило вывода, теперь необходимо последовательно доказать все цели, перечисленные в теле правила. Для доказательства цели middle (Animal) вновь начинается просмотр всех предложений, имеющихся в тексте программы, и находится факт middle (tiger). Но! Поскольку имеется еще один факт, описывающий животное средних размеров middle (bear). , устанавливается вторая точка возврата (назовем ее *2).Переменная Animal получает значение tiger. Первая цель в теле правила успешно доказана.

Теперь выполняется переход к доказательству цели strong (tiger). Переменная Animal получила значение tiger при доказательстве предыдущей цели. Чтобы доказать цель strong (tiger), вновь начинается просмотр всех предложений, имеющихся в тексте программы, и находится факт strong (tiger), успешно доказывающий цель strong (tiger). Точка возврата не устанавливается, так как в тексте программы нет больше фактов strong, описывающих сильных животных.

Так как доказаны все цели в теле правила, считается успешно доказанной головная цель правила, и, следовательно, цель powerful (Animal), записанная в исходном запросе.

Найдено первое решение: Animal=tiger.

Поскольку должны быть найдены все возможные решения, вступает в действие поиск с возвратом, который возвращает выполнение программы к последней установленной точке возврата – *2, то есть к цели middle (Animal), которая может быть передоказана. Вновь начинается просмотр всех предложений, но не с самого первого, а с того, на котором была установлена точка возврата *2 и цель middle (Animal) успешно передоказывается фактом middle (bear). Следует отметить, что переменная Animal, получившая при нахождении первого решения значение tiger, потеряла это значение, когда поиск с возвратом вернулся к передоказательству цели middle (Animal), то есть, нет никаких препятствий к тому, чтобы переменная Animal получила теперь значение bear.

Точка возврата *2 удаляется и вновь не устанавливается, так как нет более фактов middle, описывающих животных средних размеров. Итак, успешно передоказана первая цель в теле правила, и восстанавливается исходный порядок действий, то есть выполняется переход к доказательству второй цели в теле правила, но только теперь это цель strong (bear). Найти факт, доказывающий данную цель, не удается, то есть считается недоказанной вторая цель в теле правила, следовательно, вновь в действие вступает поиск с возвратом и происходит возврат к ближайшей точке возврата, а это точка *1. Точка возврата *1 свидетельствует о

том, что вновь начинается просмотр предложений в тексте программы, но не с самого начала, а с предложения, помеченного этой самой точкой *1. При просмотре обнаруживается, что цель в запросе powerful (Animal) может быть передоказана с помощью второго правила вывода.

Так как выполнен возврат к последней точке возврата, переменная Animal вновь теряет свое значение, и, так как больше возможностей для передоказательства цели в запросе powerful (Animal) нет, точка возврата *1 более не устанавливается.

Теперь вновь повторяются действия, похожие на действия, происходившие, когда для доказательства использовалось первое правило вывода, только действий будет немного меньше, так как в теле второго правила всего одна цель.

Итак, цель запроса powerful (Animal) была сопоставлена с заголовком второго правила, что привело к необходимости доказать единственную цель в теле правила – big (Animal). Для этого вновь начинается просмотр предложений в тексте программы с самого начала и обнаруживается факт big (elephant). Вновь устанавливается точка возврата, назовем ее *3, говорящая о том, что цель big (Animal) в дальнейшем может быть передоказана. Факт big (elephant). успешно доказывает цель big (Animal) и переменная Animal принимает значение elephant. Так как успешно доказаны все (в данном случае одна) цели в теле правила, считается успешно доказанной и заголовочная цель правила, что приводит к успеху в доказательстве цели в запросе, и вот оно, второе решение:

Animal=elephant.

Так как успешно найдено очередное решение, возобновляются действия по поиску следующих возможных решений, ведь есть точка возврата *3, к которой можно вернуться и передоказать цель в теле правила big (Animal) (в процессе поиска с возвратом переменная Animal вновь теряет свое значение). Точка возврата *3 свидетельствует о том, что вновь начинается просмотр предложений в тексте программы, но не с самого начала, а с предложения, помеченного точкой *3. При просмотре обнаруживается, что цель в теле правила big (Animal) может быть передоказана с помощью факта big (hippopotamus). , что и делается, переменная Animal получает значение hippopotamus, точка возврата *3 удаляется и более не устанавливается, так как больше нет фактов, описывающих больших животных, и считается найденным очередное, третье, решение:

Animal=hippopotamus.

Так как все возможные решения найдены, выполнение программы заканчивается.

Далее приводится пример трассировки (пошагового выполнения) только что рассмотренного примера.

Условные обозначения для трассировки:

1. CALL – цель, которую нужно доказать.
2. RETURN – цель, которая успешно доказана.
3. REDO – поиск с возвратом.
4. FAIL – неудача в доказательстве.
5. * – точка возврата.
6. _ – переменная, не имеющая значения.

```
CALL:   powerful (_)
CALL:   middle (_)
RETURN: *middle ("tiger")
CALL:   strong ("tiger")
RETURN: strong ("tiger")
RETURN: *powerful ("tiger")
REDO:   middle (_)
RETURN: middle ("bear")
```

```
CALL:    strong ("bear")
FAIL:    strong ("bear")
REDO:    powerful (_)
CALL:    big (_)
RETURN: *big ("elephant")
RETURN: powerful ("elephant")
REDO:    big (_)
RETURN: big ("hippopotamus")
RETURN: powerful ("hippopotamus")
```

Теперь можно сформулировать основные правила поиска с возвратом:

1. Цели должны быть доказаны по порядку, слева, направо.
2. Для доказательства некоторой цели предложения просматриваются в том порядке, в каком они появляются в тексте программы.
3. Для того, чтобы доказать головную цель правила, необходимо доказать цели в теле правила. Тело правила состоит, в свою очередь из целей, которые должны быть доказаны.
4. Цель считается доказанной, если с помощью соответствующих фактов доказаны все цели, находящиеся в листьевых вершинах дерева целей.

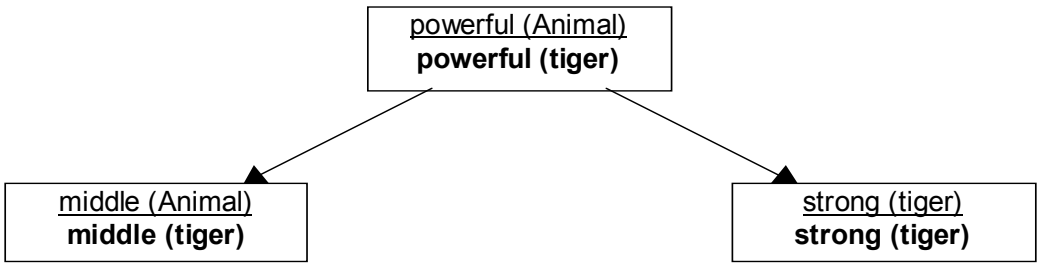
Для последнего правила следует пояснить, что называется деревом целей. Ход решения программы удобно представлять в виде дерева, которое называется деревом целей.

Пример дерева целей для ранее рассмотренного примера, для случая нахождения первого решения Animal=tiger.

Условные обозначения в дереве целей:

1. powerful (Animal) – цель, которую нужно доказать.
2. **powerful (tiger)** – цель, которая успешно доказана.

В данном дереве целей две цели: middle (Animal) и strong (tiger), находящиеся в листьевых вершинах, и обе они доказаны.



Управление поиском с возвратом: предикаты ! и fail

Управление поиском с возвратом заключается в решении двух задач: включении поиска с возвратом при его отсутствии, и отключении поиска с возвратом при его наличии.

Для решения этих задач используются два стандартных предиката:

1. предикат fail, включающий поиск с возвратом.
2. предикат ! (этот предикат еще называют «отсечение»), предотвращающий поиск с возвратом.

Рассмотрим, как работает предикат fail. Для начала следует напомнить, что поиск с возвратом начинает свою работу только в том случае, если не удается доказать какую-либо цель. Поэтому действует предикат fail очень просто – цель с использованием данного предиката НИКОГДА не доказывается, а, следовательно, всегда включается поиск с возвратом.

Для получения такого же эффекта можно записать, например, вот такую цель: 2=3. Эффект будет абсолютно тем же самым. Предикат fail используется в тех случаях, когда в программе

есть внутренняя цель, и необходимо позаботиться о нахождении всех возможных решений. Рассмотрим простой пример: вывод названий всех стран, перечисленных в фактах.

```
PREDICATES
country (string)
print

CLAUSES
country (“Финляндия”).
country (“Швеция”).
country (“Норвегия”).
print:- country (Country_name), write (Country_name), nl.

GOAL
print.
```

Результатом работы программы будет вывод только названия первой страны из перечня фактов – Финляндии. Произойдет это из-за того, что при использовании внутренней цели находится только первое решение задачи.

Для того чтобы в процессе выполнения программы был выведен полный перечень названий стран, необходимо, чтобы цель country (Country_name) была передоказана столько раз, сколько имеется фактов в секции CLAUSES. Эта цель достигается очень просто. Предложение для предиката print нужно лишь переписать следующим образом:

```
print:- country (Country_name), write (Country_name), nl, fail.
```

В таком случае данное правило будет работать следующим образом: первый раз цель country (Country_name) будет успешно доказана с помощью факта country (“Финляндия”). и переменная Country_name будет конкретизирована значением “Финляндия”. Затем будет выведено значение переменной Country_name и наступит черед цели fail, которая никогда не доказывается.

Естественно, будет инициализирован поиск с возвратом, и возврат будет выполнен к ближайшей цели, которую можно передоказать. (Следует отметить, что ближайшей считается цель, встреченная при возврате, то есть при движении справа налево.) Эта цель – country (Country_name). Так как заработал поиск с возвратом, переменная Country_name теряет свое значение, и ничто не препятствует успешному передоказательству цели country (Country_name) фактом country (“Швеция”). Подобные действия будут повторяться до тех пор, пока не будут исчерпаны все факты, используемые для доказательства.

В результате будет выведен список всех стран, то есть программа выполнит те действия, которых от нее ждали. Однако следует сделать небольшое, но важное замечание. Несмотря на то, что программа выполнила все ожидаемые действия, в итоге выполнения программы завершится с неуспехом, поскольку цель print из секции GOAL доказана не будет. Недоказательство цели print произойдет вот по какой причине. Когда цель country (Country_name) будет в последний раз успешно доказана фактом country (“Норвегия”). в действие вновь вступает цель fail. Но передоказать цель country (Country_name) более нельзя, все факты исчерпаны и, так как не удалось доказать цель в теле правила и головная цель правила считается недоказанной, следовательно будет считаться недоказанной цель print из секции GOAL.

Избежать этого изъяна в работе программы очень легко, следует всего лишь добавить еще одно предложение для предиката print. Следующий вариант примера будет выполнять все необходимые действия, и выполнение программы будет завершаться с успехом.

```
PREDICATES
country (string)
print
```

```
CLAUSES
country (“Финляндия”).
country (“Швеция”).
country (“Норвегия”).
print:- country (Country_name), write (Country_name), nl, fail.
print.

GOAL
print.
```

В данном примере наличие второго предложения для предиката print создает еще одну точку возврата. Когда цель fail в очередной раз своим недоказательством включает поиск с возвратом, передоказать цель country (Country_name) более нельзя, все факты исчерпаны, вот тогда поиск с возвратом и возвращается к передоказательству цели print, что с успехом делается с помощью второго предложения для предиката print.

Еще один пример, показывающий, как можно управлять поиском с возвратом без предиката fail. В приведенном примере выполняется вывод положительных чисел до первого встреченного отрицательного числа. В этом случае работа программы прекращается.

```
PREDICATES
number (integer)
output

CLAUSES
number (2).
number (1).
number (0).
number (-1).
number (-2).
output:- number (Positive_number), write (Positive_number), nl, Positive_number<0.
output.

GOAL
output.
```

В данном случае цель Positive_number<0 играет роль, если так можно выразиться, условного предиката fail. Пока цель number (Positive_number) будет доказываться фактами, содержащими положительные числа, цель Positive_number<0 доказываться не будет, и, сдедовательно, будет работать поиск с возвратом. Как только переменная Positive_number будет конкретизирована значением –1, цель Positive_number<0 будет успешно доказана, завершится успехом доказательство всего правила и цели output в секции GOAL. В этом примере второе предложение для предиката output требуется только на тот случай, если бы все факты number содержали бы только положительные числа.

Еще одно средство для управления поиском с возвратом – это стандартный предикат ! (отсечение). Действие этого предиката прямо противоположно действию предиката fail. Если предикат fail всегда включает поиск с возвратом, то отсечение поиск с возвратом прекращает.

Рассмотрим, как работает отсечение, на примере. Пусть имеется набор фактов, описывающих некоторые числа.

```
PREDICATES
tens (string)
ones (string)
numbers
```

```
CLAUSES
tens ("двадцать").
tens ("тридцать").
ones ("два").
ones ("три").
numbers:- tens (Tens_number), ones(Ones_number),
          write (Tens_number, " ", Ones_number), nl, fail.
numbers.

GOAL
numbers.
```

Результатом работы программы будет вывод следующих строк:

```
двадцать два
двадцать три
тридцать два
тридцать три
(выполнение программы завершится с успехом)
```

В данном случае в программе будет две точки возврата, которые и дадут этот эффект. Вместо подробного описания работы программы ниже дается трассировка (с некоторыми несущественными сокращениями), из которой становится ясной логика работы программы. Следует заметить, что выполнение программы завершится с успехом.

```
CALL:    numbers ()
CALL:    tens (_)
RETURN: *tens ("двадцать")
CALL:    ones (_)
RETURN: *ones ("два")
         write ("двадцать", " ", "два")
REDO:    ones (_)
RETURN: ones ("три")
         write ("двадцать", " ", "три")
REDO:    tens (_)
RETURN: tens ("тридцать")
CALL:    ones (_)
RETURN: *ones ("два")
         write ("тридцать", " ", "два")
REDO:    ones (_)
RETURN: ones ("три")
         write ("тридцать", " ", "три")
REDO:    numbers ()
RETURN: numbers ()
```

Если теперь добавить в правило вывода отсечение,

```
numbers:- tens (Tens_number), !, ones (Ones_number),
          write (Tens_number, " ", Ones_number), nl, fail.
```

то это существенно изменит результат работы программы:

```
двадцать два
двадцать три
(выполнение программы завершится с неуспехом)
```

Почему отсечение произвело такой эффект? Рассмотрим, как работает отсечение. Когда выполняется последовательное доказательство целей слева направо, то цель ! (отсечение)

доказывается всегда и при этом выполняется еще одно очень важное действие – отсечение уничтожает все точки возврата, которые остались слева от отсечения.

Следовательно, когда цель ! была успешно доказана, точка возврата для цели tens (Tens_number) была уничтожена, а с точкой возврата для цели ones (Ones_number) ничего не произошло.

Когда предикат fail инициализировал поиск с возвратом, «в живых» осталась только одна точка возврата, для цели ones (Ones_number), то есть только для этой цели перебирались все возможные решения. Другими словами, после того, как доказательство целей миновало отсечение, поиск с возвратом возможен только справа от отсечения (нужно отметить, что, до тех пор, пока цель отсечение не доказана, поиск с возвратом возможен и слева от цели !). Пример трассировки для второго варианта правила.

```
CALL:    numbers ()
CALL:    tens (_)
RETURN: *tens ("двадцать")
CALL:    ones (_)
RETURN: *ones ("два")
         write ("двадцать", " ", "два")
REDO:    ones (_)
RETURN: ones ("три")
         write ("двадцать", " ", "три")
```

В целом выполнение программы завершается с неуспехом, так как отсечение уничтожило не только возможность возврата к передоказательству цели tens (Tens_number), но и цели numbers. Перестановка отсечения в правиле будет существенно изменять результаты работы программы, и при наличии отсечения приведенный пример всегда будет завершаться с неуспехом.

```
numbers:- tens (Number1), ones (Number2), !,
          write (Number1, " ", Number2), nl, fail.
```

```
двадцать два
(выполнение программы завершится с неуспехом)
```

```
numbers:- !, tens (Number1), ones (Number2),
          write (Number1, " ", Number2), nl, fail.
```

```
двадцать два
двадцать три
тридцать два
тридцать три
(выполнение программы завершится с неуспехом)
```

Отсечение весьма полезно при организации ветвления с помощью нескольких правил с одной и той же целью в заголовке правила. Пример программы, которая проверяет, делится ли введенное число нацело на 2, на 3 или на 5.

```
PREDICATES
division (integer)
start (string)

CLAUSES
division (N):- N mod 2 = 0, !, write (N, “делится на 2 без остатка.”), nl.
division (N):- N mod 3 = 0, !, write (N, “делится на 3 без остатка.”), nl.
division (N):- N mod 5 = 0, !, write (N, “делится на 3 без остатка.”), nl.
division (_):- write (“Ваше число не делится нацело ни на 2, ни на 3, ни на 5!!!”).
```

```
GOAL
write (“Ваше число? ”), readint (Number), division (Number).
```

В рассмотренном примере отсечение убирает совершенно ненужные точки возврата. Предположим, пользователь ввел число 1023. В этом случае в первом правиле для предиката division условие N mod 2 = 0 доказано не будет, следовательно, будет выполнен откат ко второму правилу. Препятствий для отката нет, так как отсечение в первом правиле не сработало. Во втором правиле условие N mod 3 = 0 успешно доказывается, и в этом случае доказательство проходит через отсечение. Как уже упоминалось, отсечение всегда доказывается с успехом, и будут уничтожены все точки возврата, проставленные к моменту доказательства цели !, оказавшиеся совершенно ненужными. Действительно, ведь если условие N mod 3 = 0 верно, а то, что N не делится нацело на 2, было проверено с помощью первого правила, третье и четвертое правила для предиката division точно не понадобится, то есть и не нужно сохранять бесполезную точку возврата.

Если при написании программы есть возможность выносить проверку некоторого условия непосредственно в заголовок правила, лучше это сделать. Например, программа, проверяющая, является ли введенное число равным 1, 2 или 3, может быть написана следующим образом:

```
…
choice (N):- N=1, !, write (“Это единица.”).
choice (N):- N=2, !, write (“Это двойка.”).
choice (N):- N=3, !, write (“Это тройка.”).
choice (N):- write (“Это не единица, не двойка, не тройка!!!”).
…
```

Более кратко правила можно записать с проверкой значения N не отдельным условием, а непосредственно в заголовке правила:

```
…
choice (1):- !, write (“Это единица.”).
choice (2):- !, write (“Это двойка.”).
choice (3):- !, write (“Это тройка.”).
choice (_):- write (“Это не единица, не двойка, не тройка!!!”).
…
```

Анонимная переменная в заголовке последнего правила говорит о том, что значение переменной роли не играет. Действительно, если дело дошло до последнего правила, значит, значение переменной не равно 1, 2 или 3.

Всегда следует убирать ненужные точки возврата как можно раньше.

Итак, поиск с возвратом возможен только в случае, если в предложении есть цели, которые можно передоказать. Как же быть, если поиск с возвратом необходим для решения задачи, но нет целей, которые можно передоказывать? В такой ситуации приходиться создавать точку возврата искусственно, используя специальный предикат, для которого должны быть определены два предложения. Цель, записанная с использованием данного предиката, должна соответствовать двум условиям: не выполнять никаких видимых действий и ВСЕГДА генерировать точку возврата. Такой специальный предикат определяется следующим образом (предикат не является стандартным и его имя может быть выбрано совершенно произвольно):

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

Рассмотрим пример: вывод приглашения «<>», ввод с клавиатуры строки и вывод ее на экран до тех пор, пока не будет введена строка stop.

```
PREDICATES
```

```
repeat
echo

CLAUSES
repeat.
repeat:- repeat.
echo:- repeat, write (“> ”), readln (String), write (String), nl, String=”stop”.

GOAL
echo.
```

Если написать правило вывода без цели repeat,

```
echo:- write (“> ”), readln (String), write (String), nl, String=”stop”.
```

будет выполнен ввод и вывод только первой и единственной введенной строки, неравной “stop”. Действительно, после ввода и вывода строки будет выполнена проверка String=”stop”, которая завершится неуспехом, что приведет к включению поиска с возвратом, но ни одну из целей в правиле передоказать нельзя (точки возврата не были проставлены), поэтому выполнение программы завершится неуспехом.

Рассмотрим вариант правила с использованием цели repeat. При первом доказательстве цели repeat она успешно доказывается с помощью факта repeat. , при этом проставляется точка возврата, так что, если в дальнейшем не будет доказана какая-либо цель, можно будет вернуться к цели repeat и передоказать ее с помощью правила вывода repeat:- repeat.

Далее выполняется успешное последовательное доказательство всех целей, вплоть до цели String=”stop”, которая, в случае, если была введена строка, неравная ”stop”, не доказывается. Как известно, недоказательство некоторой цели приводит к включению поиска с возвратом. В данном примере цели write (“> ”), readln (String), write (String) и nl передоказать нельзя, предикат repeat же определен таким образом, что всегда может быть передоказан.

Цель repeat успешно передоказывается (вновь с генерацией точки возврата) и возобновляется естественный порядок доказательства целей, слева направо. Таким образом, организовываются повторяющиеся действия с помощью поиска с возвратом в том случае, когда исходно не было целей, дающих точку возврата.

Рекурсия

Рекурсия – это второе средство для организации повторяющихся действий в Prolog’e. Рекурсивная процедура – это процедура, вызывающая сама себя до тех пор, пока не будет соблюдено некоторое условие, которое остановит рекурсию. Такое условие называют граничным. Рекурсия – хороший способ для решения задач, содержащих в себе подзадачу такого же типа.

Пример рекурсии: найти факториал n!.

Задача нахождения значения факториала n! очень хорошо решается с помощью рекурсии, поскольку может быть сведена к решению аналогичной подзадачи, которая, в свою очередь, сводится к решению аналогичной подподзадачи и т.д.

Действительно, чтобы найти значение факториала n!, можно найти значение факториала (n–1)! и умножить найденное значения на n. Для нахождения значения факториала (n–1)! можно пойти по уже известному пути – найти значение факториала (n–2)! и умножить найденное значения на n–1. Так можно действовать до тех пор, пока не доберемся до нахождения значения факториала (n–n)! или другими словами, факториала 0!. Значение факториала 0! известно – это 1. Вот это и будет граничное условие, которое позволит остановить рекурсию. Все, что теперь остается – это умножить полученную 1 на (n-(n-1)), затем на (n-(n-2)) и т.д. столько раз, сколько было рекурсивных вызовов. Результат n! получен.

Вот как выглядит программа, которая проделывает вычисление n! (нужно заметить, что предложения Prolog-программы достаточно точно повторяют формулировку задачи на естественном языке).

```
PREDICATES
factorial (integer, integer)

CLAUSES
%факториал 0! равен 1
factorial (0, 1):- !.
%факториал n! равен факториалу (n–1)!, умноженному на n
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M),
        Factorial_N=Factorial_M*N.

GOAL
write ("Для какого числа Вы хотите найти факториал? "), readint (Number),
        factorial (Number, Result), write (Number, "!=", Result).
```

Результат работы программы: 3!=6

Каким же образом работает программа?

Выполнение программы начинается с последовательного доказательства целей, записанных в секции GOAL. Доказательство первых двух целей обеспечивает вывод подсказки и ввод значения Number (пусть будет введено значение 3). Эти цели успешно доказываются и очередь доходит до цели, собственно обеспечивающей вычисление факториала. С учетом того, что переменная Number уже конкретизирована значением 3, цель, которую нужно доказать, будет иметь вид factorial (3, Result).

Для доказательства поставленной цели будет выполнен последовательный перебор предложений и определено, что для доказательства можно использовать второе предложение

```
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M),
        Factorial_N=Factorial_M*N.
```

Цель factorial (Number, Result) сопоставляется с заголовком правила factorial (N, Factorial_N) и выполняется унификация, в результате которой переменные Number и N, Result и Factorial_N становятся сцепленными, что приводит к тому, что переменная N также получает значение 3.

Чтобы заголовок правила считался доказанным, необходимо, чтобы были доказаны все хвостовые цели правила вывода, то есть последовательно нужно доказать цели M=N–1, factorial (M, Factorial_M) и Factorial_N=Factorial_M*N.

Первая цель доказывается успешно: 2=3–1, переменная M конкретизируется значением, равным 2. Доказательство следующей цели factorial (2, Factorial_M) приводит в действие рекурсию. Цель factorial (2, Factorial_M) вновь сопоставляется с заголовком того же самого правила factorial (N, Factorial_N), и теперь переменные M и N, Factorial_M и Factorial_N также становятся сцепленными. Но только переменные, используемые в рекурсивном вызове, являются самостоятельными переменными, несмотря на то, что для них используются те же самые имена. Для того, чтобы различать переменные на различных уровнях рекурсии, к их именам будет добавляться апостроф, вот так – Factorial_M'.

Следует отметить, что доказательство цели Factorial_N=Factorial_M*N пока откладывается до тех пор, пока не будет доказана цель factorial (2, Factorial_M).

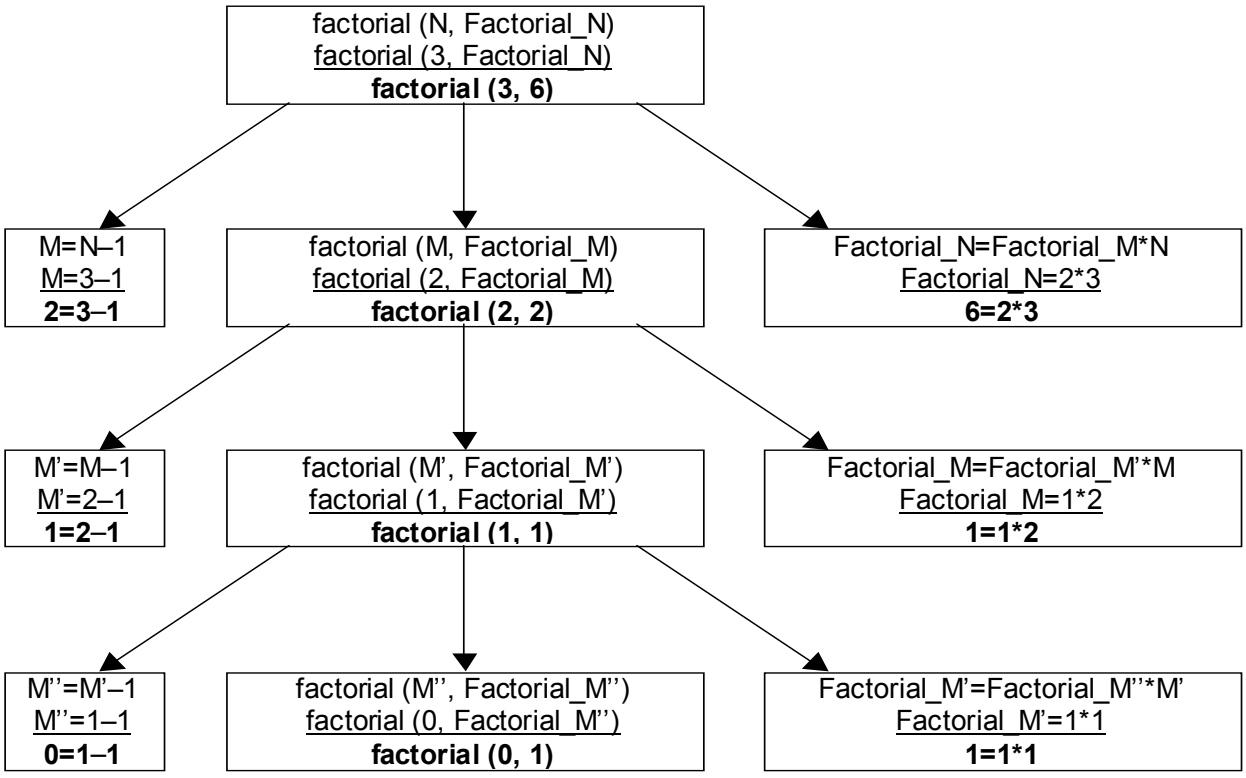
Поскольку в действие вступает рекурсия, все повторяется вновь: успешное доказательство цели 1=2–1, рекурсивный вызов factorial (1, Factorial_M'), отложенная цель Factorial_M=Factorial_M'*M. Далее: успешное доказательство цели 0=1–1, рекурсивный вызов factorial (0, Factorial_M''). НО! вот теперь наступил момент для остановки рекурсии, поскольку впервые значение факториала уменьшилось до 0, то есть впервые для

доказательства цели factorial (0, Factorial_M'') используется первое правило и наконец-то первый раз рекурсивная цель factorial (0, Factorial_M'') успешно доказывается фактом factorial (0, 1)!

Это событие позволяет впервые пройти к доказательству цели Factorial_M'=Factorial_M''*M', которая на всех предыдущих уровнях рекурсии оставалась отложенной. Цель успешно доказывается 1=1*1. В самом последнем рекурсивном вызове впервые доказаны все три хвостовые цели правила, что позволяет вернуться в рекурсии на один уровень выше и считать доказанной цель factorial (1, 1). Теперь можно пройти к доказательству отложенной цели 2=1*2. И вновь – доказаны все три хвостовые цели правила на этом уровне рекурсии, что позволяет вернуться в рекурсии еще на один уровень выше и считать доказанной цель factorial (2, 2). Снова можно пройти к доказательству отложенной цели 6=2*3. Выполнение программы практически завершено, поскольку доказаны все три хвостовые цели на самом верхнем уровне рекурсии, что дает возможным считать доказанной головную цель правила factorial (3, 6), и, далее, доказанной цель из секции GOAL factorial (3, 6). На завершающем этапе успешно доказывается последняя цель из секции GOAL, которая выводит на экран полученный результат. Выполнение программы с успехом завершено.

Обратите внимание, что результат рассчитывается в процессе возврата из рекурсивных вызовов и доказательства отложенных целей. В момент, когда рекурсия останавливается граничным условием, то есть при доказательстве цели factorial (0, Factorial_M'') значение факториала равно 1 (factorial (0, 1)).

Описание работы программы получилось достаточно громоздким, более наглядно работу программы можно представить в виде дерева целей.



Непременно нужно отметить очень важную роль отсечения в первом предложении программы. Видимых действий здесь отсечение не производит, если его убрать, то есть записать первое предложение как факт

```
factorial (0, 1).
```

результат работы программы останется тем же.

В чем же тогда смысл отсечения в этом примере? Отсечение убирает совершенно ненужную точку возврата, которая в дальнейшем может доставить много хлопот, если на нее не обратить внимания и оставить.

Это утверждение можно пояснить на примере. Выполнение программы начинается с попытки доказательства цели, например, factorial (3, Result). Для доказательства данной цели будет использовано второе правило вывода, так как в первом правиле нет совпадения по первому аргументу (3<>0), что приведет, естественно, к последовательному доказательству хвостовых подцелей правила. В процессе этого доказательства нужно будет доказать рекурсивную цель factorial (2, Factorial_M), что вновь приведет к использованию второго правила вывода (в действие вступает рекурсия), так как первое правило по прежнему не подходит для доказательства из-за несовпадения первого аргумента. Подобные действия будут продолжаться до тех пор, пока рекурсивная цель не примет вид factorial (0, Factorial_M''). Вот теперь наступил ключевой момент!

Впервые для доказательства рекурсивной цели будет использовано первое правило. При этом цель factorial (0, Factorial_M'') будет успешно, с помощью первого правила, доказана, но при этом остается потенциальная возможность использовать для доказательства той же самой цели второе правило. Другими словами, будет поставлена точка возврата. Но никакого передоказательства в дальнейшем не потребуется! Значение 0! всегда равно 1!.

Вот часть трассировки выполняемой программы для случая БЕЗ отсечения:

```
factorial (0, 1).
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M),
        Factorial_N=Factorial_M*N.
…
CALL:   factorial (2, _)
REDO:   factorial (2, _)
        1=1
CALL:   factorial (1, _)
REDO:   factorial (1, _)
        0=0
CALL:   factorial (0, _)
RETURN: *factorial (0, 1)       вот он, ключевой момент! цель доказана,
…                               но * говорит о том, что поставлена точка возврата
```

Теперь часть трассировки выполняемой программы для случая С отсечением:

```
factorial (0, 1):- !.
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M),
        Factorial_N=Factorial_M*N.
…
CALL:   factorial (2, _)
REDO:   factorial (2, _)
        1=1
CALL:   factorial (1, _)
REDO:   factorial (1, _)
        0=0
CALL:   factorial (0, _)
RETURN: factorial (0, 1)        цель доказана,
…                               но точка возврата НЕ поставлена
```

Каковы могут быть последствия, если точку возврата не убрать с помощью отсечения. Предположим, предикат factorial используется в каком-нибудь правиле вывода, например:

```
…
factorial (0, 1).
```

```
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M),
        Factorial_N=Factorial_M*N.
calculate (N, Res):- factorial (N, Res), Res<1000, !, write (“Все в порядке!”).
calculate (_, _):- write (“Слишком большое число!”).
…
```

Что же произойдет, если будет рассчитываться, например, факториал 7! (7!=5040)? Цель factorial (N, Res) будет успешно доказана, но следующая цель Res<1000 доказана не будет, то есть начнется поиск с возвратом, который как известно, возвращается к ближайшей точке возврата. В данном примере эта точка оставлена в предикате factorial. То есть, вместо того, чтобы вывести сообщение “Слишком большое число!”, как задумал автор программы, программа проваливается в бесконечную рекурсию и выполнение программы прекращается из-за переполнения стека.

Для иллюстрации вышесказанного приводится выдержка из трассировки:

```
…
CALL:    calculate(7, _)
CALL:    factorial (7, _)
FAIL:    factorial (7, _)
REDO:    factorial (7, _)
         6=6
CALL:    factorial (6, _)
FAIL:    factorial (6, _)
REDO:    factorial (6, _)
…
REDO:    factorial (1, _)
         0=0
CALL:    factorial (0, _)
RETURN: *factorial (0,1)          оставшаяся точка возврата
         1=1
RETURN: factorial (1,1)
…
RETURN: factorial (6,720)
         5040=5040
RETURN: factorial (7,5040)
         5040<1000
FAIL:    calculate(7, _)          неудача, начинается поиск с возвратом
REDO:    factorial (0, _)         а вот и последствия, возвращаемся и
         -1=-1                    проваливаемся в бесконечную рекурсию
CALL:    factorial (-1, _)
FAIL:    factorial (-1, _)
REDO:    factorial (-1, _)
         -2=-2
CALL:    factorial (-2, _)
FAIL:    factorial (-2, _)
REDO:    factorial (-2, _)
         -3=-3
CALL:    factorial (-3, _)
…
```

Теперь, для сравнения, как все происходит, если точка возврата ликвидирована с помощью отсечения:

```
…
factorial (0, 1):- !.
```

```
factorial (N, Factorial_N):- M=N–1, factorial (M, Factorial_M),
        Factorial_N=Factorial_M*N.
calculate (N, Res):- factorial (N, Res), Res<1000, !, write (“Все в порядке!”).
calculate (_, _):- write (“Слишком большое число!”).
…
```

Вновь пример трассировки:

```
…
REDO:   factorial (1, _)
        0=0
CALL:   factorial (0, _)
RETURN: factorial (0,1)                          точки возврата нет!!!
        1=1
RETURN: factorial (1,1)
…
RETURN: factorial (6, 720)
        5040=5040
RETURN: factorial (7, 5040)
        5040<1000
FAIL:   calculate (7, _)
REDO:   calculate (7, _)
        write("Слишком большое число!")   работает, как и было задумано!!!
…
```

Вывод из всего вышеизложенного – никогда не оставлять ненужные точки возврата, убирая их с помощью отсечения. Если программа объемная, то достаточно сложно понять, куда выполняется откат и где оставлена лишняя точка возврата.

Рассмотрим еще один пример рекурсивной программы. Поскольку в PDC Prologe’e нет стандартного предиката для возведения в степень, приходится эту операцию реализовывать самостоятельно. Для решения этой задачи также очень удобно использовать рекурсию, так как для того чтобы вычислить x^n, можно вычислить $x^{(n-1)}$ и умножить полученный результат на x. Для вычисления $x^{(n-1)}$ воспользоваться тем же способом: вычислить $x^{(n-2)}$ и умножить полученный результат на x. Продолжать эти действия до тех пор, пока не придет черед вычисления $x^{(n-n)}$, что, как известно равно 1 (любое число в нулевой степени равно единице $x^0=1$). Вот как это реализуется на Prolog’e:

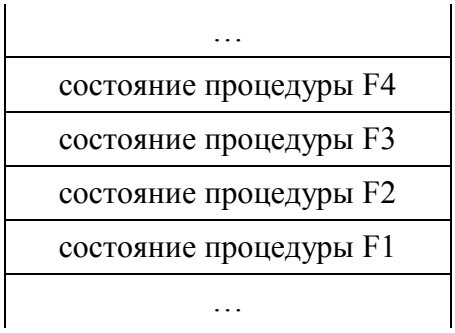
```
PREDICATES
%первый аргумент – основание степени, второй – показатель степени,
% третий – результат
power (real, integer, real)

CLAUSES
%любое число в нулевой степени равно 1
power (_, 0, 1):- !.
%x^n=x^(n–1)*x
power (X, N, X_powerN):- M=N–1, power (M, X_powerM),
        Xpower_N=Xpower_M*X.

GOAL
write (“Основание степени? ”), readreal (X),
        write (“Показатель степени? ”), readint (N),
        power (X, N, Result), write (X, “ в степени ”, N“ =”, Result).
```

Результат работы программы: 3 в степени 2=9

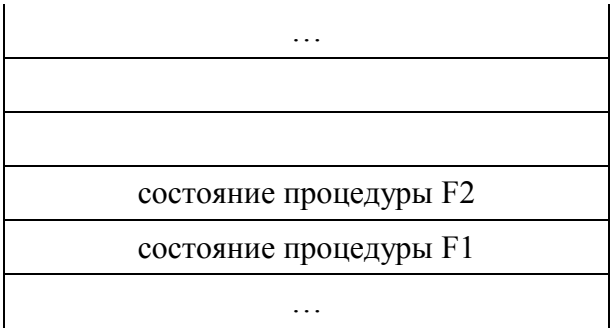
Как видно, рекурсивные задачи, решаемые с помощью Prolog'а, отличаются краткостью и приближенностью к естественному языку.

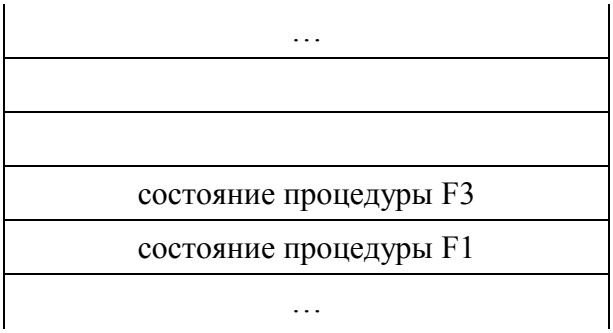
Но таким образом организованная рекурсия имеет один, но существенный недостаток. При достаточно глубокой рекурсии переполняется стек вызовов, и выполнение программы аварийно завершается. Можно ли избежать такой ситуации? Да, это возможно, при специальным образом организованной рекурсии. Такая рекурсия называется хвостовой. Прежде чем перейти к рассмотрению хвостовой рекурсии, рассмотрим для начала следующий пример.

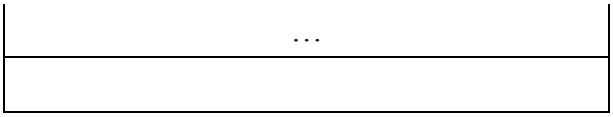
Предположим, имеются некоторые абстрактные процедуры F1, F2, F3 и F4. Пусть в процессе выполнения процедуры F1 выполняется вызов процедуры F2, при выполнении которой, в свою очередь, вызывается процедура F3, которая, в свою очередь, вызывает F4. В этом случае стек вызовов будет выглядеть следующим образом:

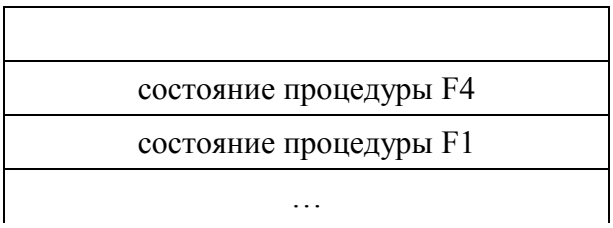


Сохранять состояние вызывающей процедуры необходимо для того, чтобы продолжить ее выполнение после завершения вызова. Но, а если вызов процедур F2, F3 и F4 будет последним действием в вызывающих процедурах? Тогда можно не сохранять состояние вызывающей процедуры, так как в ней после завершения вызова больше ничего выполняться не будет. Можно хранить в стеке вызовов только состояние последней вызванной процедуры, другими словами подменять состояние бывшей процедуры состоянием новой процедуры.



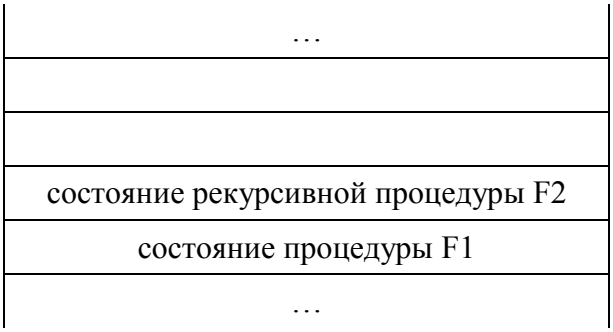






То есть считать, что процедура F1 вызывает процедуру F4 непосредственно. В этом случае, не расходуется стек вызовов.

Теперь перейдем к варианту с рекурсивными вызовами процедур. Пусть в процессе выполнения процедуры F1 выполняется вызов процедуры F2, при выполнении которой, в свою очередь, рекурсивно вызывается процедура F2, которая, в свою очередь, вновь рекурсивно вызывает саму себя, то есть опять вызывает F2. В этом случае стек вызовов будет хранить только состояние последнего рекурсивного вызова, то есть стек вновь не будет переполняться!



Выполнение таким образом организованной рекурсии не будет завершаться аварийно из-за переполнения стека, сколько бы ни было рекурсивных вызовов! Аварийное завершение по другим причинам, конечно, не исключается.

Можно на практике убедиться, что это действительно так. Попробуйте запустить следующую программу и посмотреть, в течение какого времени она будет благополучно работать:

```
PREDICATES
tail_recursion

CLAUSES
%хвостовая рекурсия
tail_recursion:- write ("*"), tail_recursion.

GOAL
tail_recursion.
```

Теперь можно сформулировать условия, при соблюдении которых рекурсия в Prolog'е становится хвостовой, то есть не расходует стек при неограниченном количестве рекурсивных вызовов:

1. рекурсивный вызов должен быть последней целью в хвостовой части правила вывода.
2. перед рекурсивным вызовом не должно быть точек возврата (это условие хвостовой рекурсии специфично для Prolog'а).

Если первое условие очевидно, то необходимость выполнения второго условия может быть, на первый взгляд, не совсем понятна. Чтобы рекурсия была хвостовой, необходимо, чтобы доказательство рекурсивного вызова было действительно последним действием в хвостовой части правила. А если до рекурсивного вызова имеется цель, которую можно передоказать, то есть имеется точка возврата? Тогда придется сохранять в стеке состояние вызывающего

правила! Вдруг в дальнейшем, где-то в глубинах рекурсии какая-либо цель не будет доказана? Тогда будет включен поиск с возвратом к ближайшей точке возврата, для чего и нужно сохранять состояние в стеке соответствующего правила (чтобы знать, куда вернуться).

Если соблюдение первого условия сложности не представляет (легко проконтролировать, чтобы рекурсивный вызов был последней целью в теле правила), то как быть уверенным в соблюдении второго условия, в отсутствии точек возврата до рекурсивного вызова?

Соблюсти второе условие очень просто. Достаточно перед рекурсивным вызовом поставить отсечение. Только и всего! Конечно, использовать отсечение следует как можно раньше в теле правила, но, в крайнем случае, его можно использовать в качестве предпоследней цели (последняя цель, естественно, рекурсивный вызов)

Примеры нехвостовой рекурсии и ее преобразования в хвостовую:

```
%пример нехвостовой рекурсии
PREDICATES
counter (integer)

CLAUSES
counter (N):- write (“N=”, N), nl, New_N=N+1, counter (New_N), nl.

GOAL
counter (0).
```

Естественно, приведенный пример не является примером хвостовой рекурсии, однако получить хвостовую рекурсию достаточно просто: нужно всего лишь убрать цель nl (переход на новую строчку) после рекурсивного вызова.

```
%пример хвостовой рекурсии
PREDICATES
counter (integer)

CLAUSES
%выполнение программы аварийно завершится из-за переполнения
%разрядной сетки для integer, но не из-за переполнения стека
counter (N):- write (“N=”, N), nl, New_N=N+1, counter (New_N).

GOAL
counter (0).
```

В рассмотренном примере соблюдены оба условия хвостовой рекурсии: рекурсивный вызов последний в теле правила и нет точек возврата перед рекурсивным вызовом. Действительно, цели write (“N=”, N), nl и New_N=N+1 передоказать нельзя, соответственно, нет и точки возврата.

```
%пример нехвостовой рекурсии
PREDICATES
counter (integer)

CLAUSES
counter (N):- N>0, write (“N=”, N), nl, New_N=N–1, counter (New_N).
counter (N):- write (“Отрицательное N=”, N).

GOAL
counter (1000).
```

В приведенном примере рекурсия хвостовой не является из-за оставленной точки возврата. Пока N будет положительным, для доказательства рекурсивной подцели будет использоваться первое правило вывода, но ведь остается неиспользованным второе правило, что и дает точку возврата. Преобразовать нехвостовую рекурсию в хвостовую можно, убрав точку возврата с помощью отсечения. Первое правило нужно переписать следующим образом:

```
counter (N):- N>0, !, write (“N=”, N), nl, New_N=N–1, counter (New_N).
```

Если N – положительное число, второе правило заведомо не понадобится.

```
%пример хвостовой рекурсии
PREDICATES
counter (integer)

CLAUSES
counter (N):- N>0, !, write (“N=”, N), nl, New_N=N–1, counter (New_N).
counter (N):- write (“Отрицательное N=”, N).

GOAL
counter (1000).

%пример нехвостовой рекурсии
PREDICATES
counter (integer)
check (integer)

CLAUSES
counter (N):- check (N), write (“N=”, N), nl, New_N=N–1, counter (New_N).
check (N):- N>0, write (“Положительное ”).
check (_):- write (“Отрицательное ”).

GOAL
counter (1000).
```

В приведенном примере рекурсия также хвостовой не является вновь из-за оставленной точки возврата. Пока N будет положительным, для доказательства рекурсивной подцели будет использоваться первое правило вывода для предиката check, но ведь остается неиспользованным второе правило для того же самого предиката, что и дает точку возврата. Преобразовать нехвостовую рекурсию в хвостовую можно, убрав точку возврата с помощью отсечения, как и в первом примере. Первое правило для предиката counter можно переписать следующим образом:

```
counter (N):- check (N), !, write (“N=”, N), nl, New_N=N–1, counter (New_N).
```

Но более правильно, с точки зрения хорошего стиля программирования на Prolog’е, точку возврата лучше убрать в предложении для предиката check.

```
%пример хвостовой рекурсии
PREDICATES
counter (integer)
check (integer)

CLAUSES
counter (N):- check (N), write (“N=”, N), nl, New_N=N–1, counter (New_N).
check (N):- N>0, !, write (“Положительное ”).
check (_):- write (“Отрицательное ”).
```

```
GOAL
counter (1000).
```

Не всегда так легко и просто можно преобразовать нехвостовую рекурсию в хвостовую. Иногда для этого требуется полностью переписать программу. Рассмотрим уже известный пример вычисления факториала. Рекурсия в приведенном ранее примере, очевидно, хвостовой не является. Только с помощью применения отсечения добиться хвостовой рекурсии нельзя. Придется полностью преобразовать программу.

Пример: вычисление n! с применением хвостовой рекурсии.

```
PREDICATES
factorial (integer, integer)
factorial_aux (integer, integer, integer, integer)

CLAUSES

factorial (N, Factorial_N):- factorial_aux (N, Factorial_N, 1, 1).

factorial_aux (N, Factorial_N, Counter, Product):- Counter<=N, !,
            New_Product=Product*Counter, New_Counter=Counter+1,
            factorial_aux (N, Factorial_N, New_Counter, New_Product).
factorial_aux (_, Factorial_N, _, Factorial_N).

GOAL
write (“Для какого числа Вы хотите найти факториал? ”), readint (Number),
factorial (Number, Result), write (Number, “!=”, Result).
```

В данном случае при организации рекурсии были использованы вспомогательный предикат factorial_aux с четырьмя параметрами. Переменные N и Factorial_N служат для тех же целей, что и в примере с нехвостовой рекурсией, переменная N конкретизируется значением, для которого нужно вычислить факториал, Factorial_N – собственно полученный результат. Переменные Counter и Product – вспомогательные, Counter – счетчик рекурсивных вызовов, Product – постепенно накапливающийся результат.

Рассмотрим более подробно работу программы. Единственное предложение для предиката factorial служит для того, чтобы перейти от предиката с двумя параметрами к вызову предиката с четырьмя параметрами. Первое предложение для предиката factorial_aux выполняет основные вычисления. При этом результат постепенно формируется при выполнении рекурсивных вызовов и достигает нужного значения в момент, когда текущий счетчик Counter становится больше значения N. В этот момент переменная Product конкретизируется значением готового результата, то есть в момент, когда рекурсия должна быть остановлена, значение факториала уже рассчитано.

В приведенном примере результат готов в тот момент, когда условие Counter<=N более не выполняется (соответственно цель Counter<=N не доказывается), переменная Product конкретизируется значением рассчитанного факториала. Теперь необходимо передать полученное значение из четвертого параметра во второй. Для этого служит второе предложение для предиката factorial_aux. Цель Counter<=N не доказывается и выполняется поиск с возвратом ко второму предложению для предиката factorial_aux. Использование одноименных переменных для второго и четвертого параметров в этом предложении как раз и обеспечивает переприсваивание полученного значения. Все, что теперь остается сделать – это передать полученное значение факториала из рекурсивных вызовов, никак не изменяя его.

Следует отметить применение отсечения в первом предложении для предиката factorial_aux. Отсечение в данном случае служит для того, чтобы убрать точку возврата, если цель Counter<=N успешно доказывается, соответственно второе предложение для предиката

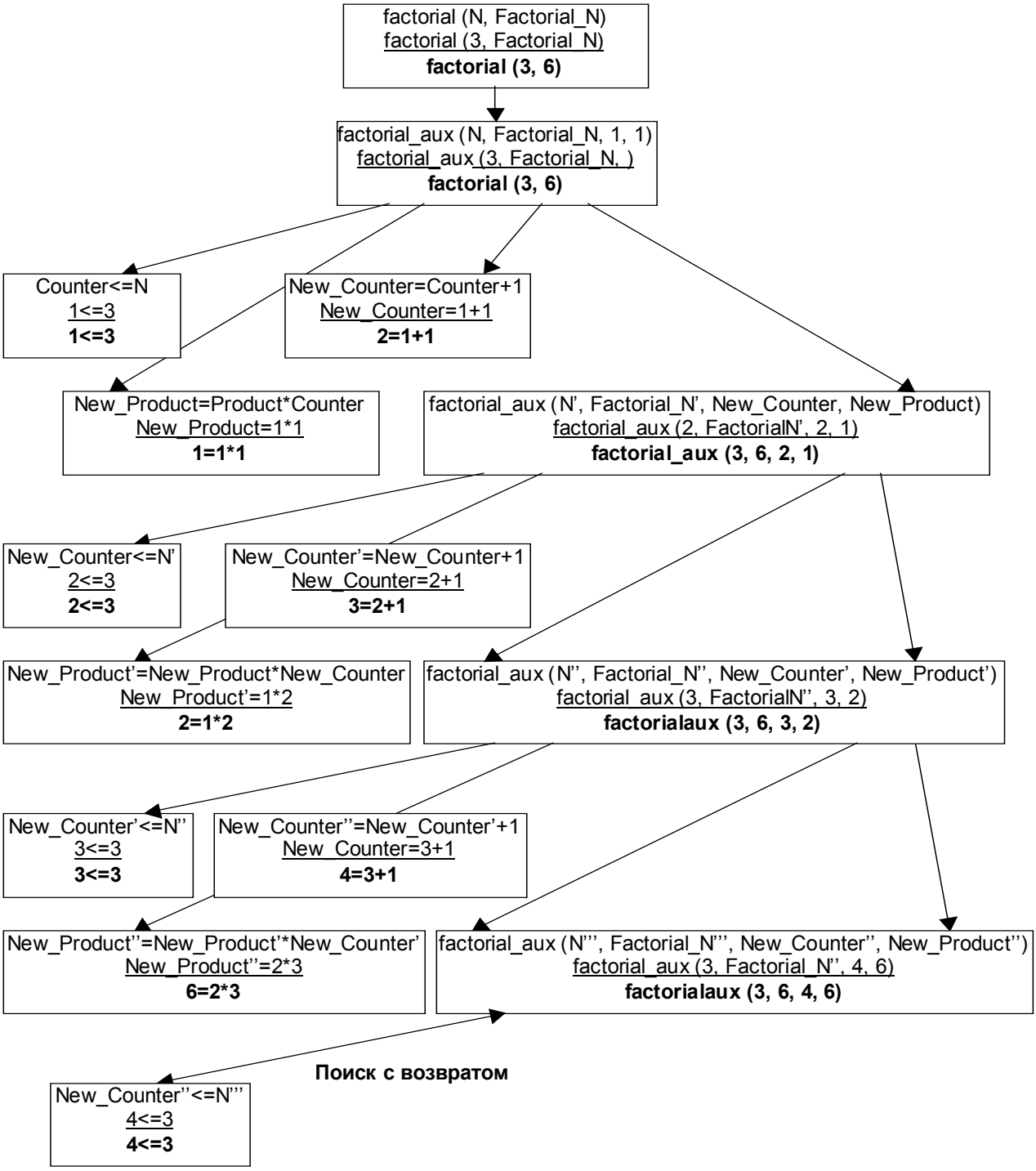
factorial_aux использовано заведомо не будет. Во втором предложении также нет проверки условия Counter>N, так как это излишне. Переход ко второму предложению будет выполнен, только если Counter будет больше, чем N. В качестве первого и третьего параметров используются анонимные переменные, так как в данном случае не важно ни значение переменной N, ни значение текущего счетчика Counter.

Второе предложение можно переписать в более подробном варианте,

```
factorial_aux (_, Factorial_N, _, Product):- Factorial_N=Product.
```

однако так, как это предложение записано в примере выше, с точки зрения стиля программирования на Prolog'е, будет более грамотно.

Дерево целей для случая вычисления факториала 3!.



Составные объекты

Составные объекты данных позволяют рассматривать информацию, состоящую из нескольких частей, как единое целое, в то же время предоставляя возможность получать доступ к отдельным частям этой информации.

Например, дата рождения человека может быть представлена как единый объект, и в то же время, дата состоит из трех частей, числа, месяца и года. Такую составную структуру можно представить как единое целое, использовав функтор, например, birthday. Имя функтора выбирается произвольно. Тогда структура будет выглядеть следующим образом:

```
birthday (25, “мая”, 1980)
```

Секция описания доменов будет выглядеть следующим образом:

```
DOMAINS
day=birthday (integer, symbol, integer)
```

Такое описание домена позволит определить предикат, в котором в качестве аргумента можно использовать данные, принадлежащие домену day. Далее следует пример программы с использованием данного домена.

```
DOMAINS
day=birthday (integer, string, integer)
name=string

PREDICATES
congratulation (name, day)
print
print2

CLAUSES
congratulation (“Анна”, birthday (25, “мая”, 1980)).
congratulation (“Иван”, birthday (17, “декабря”, 1957)).
congratulation (“Петр”, birthday (30, “июля”, 2001)).

print:- congratulation (Name, birthday (Day, Month, Year)), write (“С днем рождения
        ”, Day, Month, Year, ”, ”, Name,”!”), nl, fail.
print.
print2:- congratulation (Name, Birthday), write (“С днем рождения ”, Birthday, ”, ”,
        Name,”!”), nl, fail.
print2.

GOAL
print, print2.
```

Как видно из вышеприведенного примера к составному объекту можно обратиться как единому целому, так и получить доступ к его составным элементам.

Списки

Список – это объект, который содержит конечное число других объектов. Список в Prolog’е можно приблизительно сравнить с массивами в других языках, но для списков нет необходимости заранее объявлять размерность.

Список в Prolog’е заключается в квадратные скобки и элементы списка разделяются запятыми. Список, который не содержит ни одного элемента, называется пустым списком.

Примеры списков:

```
список, элементами которого являются целые числа
[1, 2, 3]
```

список, элементами которого являются символы
[one, two, three]
список, элементами которого являются строки
[“One”, “Two”, “Three”]
пустой список
[ ]

Для работы со списками с Prolog’е не существует отдельного домена, для того, чтобы работать со списком, необходимо объявить списочный домен следующим образом:

```
DOMAINS
listdomain=elementdomain*
elementdomain=…
```

listdomain – это произвольно выбранное имя нестандартного списочного домена, elementdomain – имя домена, которому принадлежит элемент списка, звездочка * как раз и обозначает, что выполнено объявление списка, состоящего из элементов домена element. При работе со списками нельзя включать в список элементы, принадлежащие разным доменам. В таком случае нужно воспользоваться составным доменом.

Примеры объявления списочных доменов:

```
DOMAINS
%элементы списка – целые числа
intlist=integer*
%элементы списка – символы
symlist=symbol*
%элементы списка – строки
strlist=string*
%элементы списка – или целые числа или символы или строки
mixlist=mixdomain*
mixdomain=int(integer); sym(symbol); str(string)
```

Обратите внимание, что при объявлении составного домена были использованы функторы, так как объявление вида mixdomain=integer; symbol; string привело бы к ошибке.

Список является рекурсивным составным объектом, состоящим из двух частей. Составные части списка:

1. Голова списка – первый элемент списка;
2. Хвост списка – все последующие элементы, являющиеся, в свою очередь списком.

Примеры голов и хвостов списков:

```
[1, 2, 3]   голова списка – 1, хвост списка – [2, 3]
[1]         голова списка – 1, хвост списка – [ ]
[ ]         пустой список нельзя разделить на голову и хвост
```

В Prolog’е используется специальный символ для разделения списка на голову и хвост – вертикальная черта |.

Например:

```
[1, 2, 3] или [1 | [2, 3]] или [1 | [2| [3]]] или [1 | [2 | [3 | [ ]]]]
[1] или [1 | [ ]]
```

Вертикальную черту можно использовать не только для отделения головы списка, но и для отделения произвольного числа начальных элементов списка:

```
[1, 2, 3] или [1, 2 | [3]] или [1, 2, 3 | [ ]]
```

Примеры сопоставления и унификации в списках:

```
[1, 2, 3]=[Elem1, Elem2, Elem3]       Elem1=1, Elem2=2, Elem3=3
[1, 2, 3]=[Elem1, Elem2, Elem3 | T]   Elem1=1, Elem2=2, Elem3=3, T=[ ]
[1, 2, 3]=[Head | Tail]               Head=1, Tail=[2, 3]
```

[1, 2, 3]=[Elem1, Elem2 | T]	Elem1=1, Elem2=2, T=[3]
[1, 2, 3]=[4 | T]	ошибка
[ ]=[H | T]	ошибка

Так как список является рекурсивной структурой данных, то для работы со списками используется рекурсия. Основной метод обработки списков заключается в следующем: отделить от списка голову, выполнить с ней какие-либо действия и перейти к работе с хвостом списка, являющимся в свою очередь списком. Далее у хвоста списка отделить голову и так далее до тех пор, пока список не останется пустым. В этом случае обработку списка необходимо прекратить. Следовательно, предикаты для работы со списками должны иметь по крайней мере два предложения: для пустого списка и для непустого списка.

Пример: поэлементный вывод списка, элементами которого являются целые числа, на экран (нужно отметить, что этот пример приводится в учебных целях, список вполне может быть выведен на экран как единая структура, например, GOAL List=[1, 2, 3], write(List)).

```
DOMAINS
intlist=integer*

PREDICATES
printlist (intlist)

CLAUSES
%если список пустой, то делать ничего не нужно
printlist ([ ]):- !.
%если список непустой, то отделить от списка голову, напечатать ее и
%использовать тот же самый предикат для печати хвоста списка, то есть
%выполнить рекурсивный вызов предиката printlist, передав в качестве
%аргумента хвост
printlist ([H | T]):- write (H), nl, printlist (T).

GOAL
printlist ([1, 2, 3])
```

Результат работы программы:

```
1
2
3
```

Еще один пример работы со списком – подсчет числа элементов списка или, другими словами, определение длины списка. Для того, чтобы определить длину списка, вновь нужно рассмотреть два случая: для пустого и непустого списков. Если список пуст, то его длина равна 0, если же список не пуст, то определить его длину можно следующим образом: разделить список на голову и хвост, подсчитать длину хвоста списка и увеличить это длину хвоста на единицу (то есть учесть отделенную предварительно голову списка.)

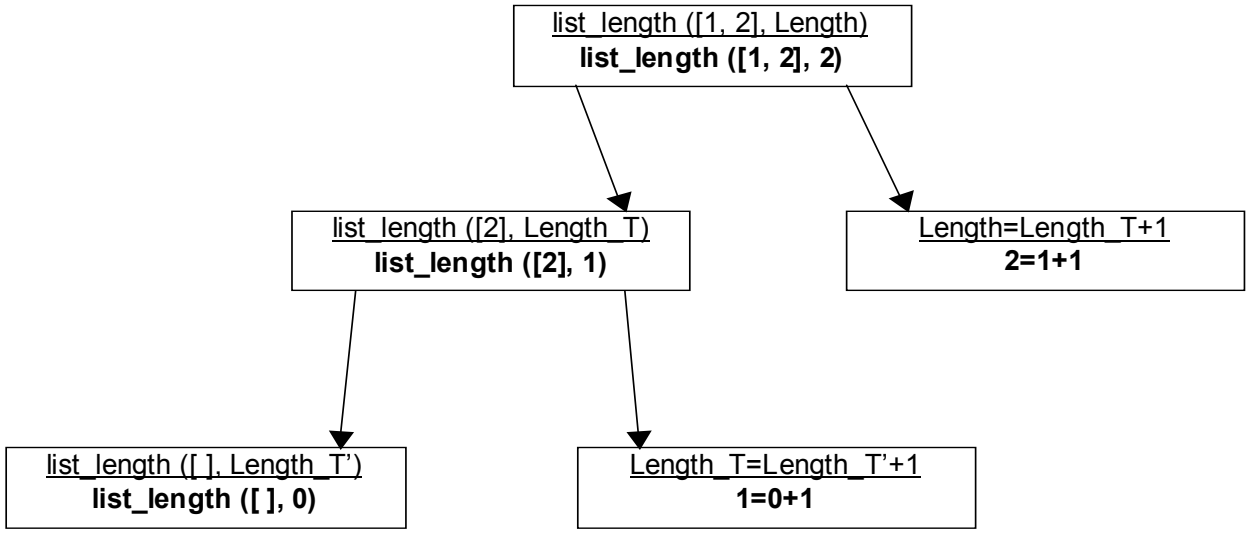
```
DOMAINS
intlist=integer*

PREDICATES
list_length (intlist, integer)

CLAUSES
%если список пустой, то его длина равна 0
list_length ([ ], 0):- !.
%если список непустой, то его длина равна длине хвоста, увеличенной на 1
list_length (List, Length):- List=[H | T], list_length (T, Length_T), Length=Length_T+1.
```

```
%более кратко второе правило вывода можно записать, перенеся разделение
%списка на голову и хвост в заголовок предложения и избавиться от лишней
%переменной List
%list_length ([H | T], Length):- list_length (T, Length_T), Length=Length_T+1.*/

GOAL
listlength ([1, 2], Length), write("Length=", Length).
```

Результат работы программы:

```
Length=2
```

Для того, чтобы лучше понять, каким образом работает приведенный пример, удобно воспользоваться деревом целей. В данном примере следует обратить внимание на то, каким образом формируется выходное значение – длина списка. Счетчик для подсчета числа элементов в списке обнуляется в момент, когда рекурсия останавливается, то есть список разбирается до пустого списка-хвоста и результат насчитывается при выходе из рекурсии.



Достаточно часто необходимо обработать список поэлементно, выполняя некоторые действия с элементами списка, в зависимости от соблюдения некоторых условий. Предположим, необходимо преобразовать список, элементами которого являются целые числа, инвертируя знак элементов списка, то есть положительные числа преобразовать в отрицательные, отрицательные в положительные, для нулевых значений никаких действий не предпринимать. Вновь придется рассмотреть два случая – для непустого и пустого списков. Преобразование пустого списка дать в результате также пустой список. Если же список не пуст, то следует рекурсивно выполнять отделение головы списка, ее обработку и рассматривать полученный результат как голову списка-результата.

```
DOMAINS
intlist=integer*

PREDICATES
inverting (intlist, intlist)
processing (integer, integer)

CLAUSES
%обработка пустого списка дает, в результате, тоже пустой список
inverting ([ ], [ ]):- !.
%если список непустой, отделить голову, обработать ее,
%и добавить в качестве головы списка-результата
```

```
inverting ([H | T], [Inv_H | Inv_T]):- processing (H, Inv_H), inverting (T, Inv_T).
%предикат processing выполняет действия по обработке элемента списка в
%зависимости от его знака, предикат имеет два предложения, так как нужно
%рассмотреть два варианта: ненулевое и нулевое значения
processing (0, 0):- !.
processing (H, Inv_H):- Inv_H=-H.

GOAL
inverting ([-2, -1, 0, 1, 2], Inv_List), write("Inv_List=", Inv_List).
```

Результат работы программы:

```
Inv_List=[2, 1, 0, -1, -2]
```

Следующий пример рассматривает достаточно часто встречающуюся задачу – определение принадлежности элемента списку. Проверка принадлежности элемента списку выполняется достаточно просто – отделением головы списка и сравнением ее с искомым элементом. Если сравнение завершилось неудачей, продолжается поиск элемента в хвосте списка. Признаком наличия искомого элемента в списке будет успешное доказательство цели, если же цель не была доказана, значит, такого элемента в списке нет.

```
DOMAINS
strlist=string*

PREDICATES
member (string, strlist)
search (string, strlist)

CLAUSES
%искомый элемент найден, его значение совпало со значением головы списка,
%хвост списка обозначен анонимной переменной, так как теперь хвост списка
%не важен
member (Elem, [Elem | _]):- !.
%если элемент пока не обнаружен, попробовать найти его в хвосте списка,
%теперь для головы списка использована анонимная переменная, поскольку ее
%значение не важно, так как оно точно не равно искомому значению
member (Elem, [_ | T]):- member (Elem, T).
%предикат search служит для двух целей: во-первых, чтобы сообщить о
%результатах поиска, и, во-вторых, чтобы при любом исходе поиска программа
%всегда заканчивала свое выполнение с успехом
search (Elem, List):- member (Elem, List), !, write ("Элемент найден! :-) ").
search (_, _):- write ("Элемент не найден! :-( ").

GOAL
Cityes=["Москва", "Санкт-Петербург", "Омск", "Новосибирск", "Томск"],
        City="Новосибирск", search (City, Cityes).
```

Результат работы программы:

```
Элемент найден! :-)
```

Последний пример, который будет рассмотрен, это решение задачи соединения двух списков. Итак, каким образом можно объединить два списка? Предположим, имеется два двухэлементных списка: [1, 2] и [3, 4]. Объединение нужно начать с последовательного отделения голов первого списка до тех пор, пока первый список не станет пустым. Как только первый список станет пустым, его легко можно будет объединить со вторым, непустым, никоим образом к этому моменту не изменившимся списком. В результате, естественно, будет получен список, полностью совпадающий со вторым. Все, что осталось

сделать, это добавить головы, отделенные от первого списка, ко второму списку. Вот как это выглядит в пошаговом описании:

1. отделяется голова от первого списка – [1, 2] –> [1 | [2]] (голова – 1, хвост – [2])
2. продолжается выполнение отделение головы, только теперь от полученного хвоста – [2] –> [2 | [ ]] (голова – 2, хвост – [ ])
3. когда первый список разобран до пустого, можно приступить к его объединению со вторым, непустым списком, объединяются пустой хвост [ ] и непустой второй список [3, 4] – получается тоже непустой список – [3, 4], теперь можно приступать к формированию объединенного списка, так как основа для списка-результата заложена, это его будущий хвост – [3, 4]
4. к хвосту списка-результата [3, 4] добавляется последняя отделенная от первого списка голова 2, что дает следующий список – [2, 3, 4]
5. все, что осталось сделать, это добавить к списку [2, 3, 4], который получился на предыдущем шаге, голову 1, которая была отделена самой первой и получается объединенный список [1, 2, 3, 4]

Теперь собственно текст программы:

```
DOMAINS
intlist=integer*

PREDICATES
append (intlist, intlist)

CLAUSES
%объединение пустого и непустого списков
append ([ ], List, List):- !.
%объединение двух непустых списков
append ([H | T], List, [H | App_T]):- append (T, List, App_T).

GOAL
append ([1, 2], [3, 4], App_List), write("App_List=", App_List).
```

Результат работы программы:

```
App_List=[1, 2, 3, 4]
```

Строки

PDC Prolog поддерживает различные стандартные предикаты для обработки строк. Основными предикатами для работы со строками можно назвать предикат frontchar (String, Char, StringRest), позволяющий разделить строку String на первый символ Char и остаток строки StringRest и предикат fronttoken (String, Lexeme, StringRest), который работает аналогично предикату frontchar, но только отделяет от строки String лексему Lexeme. Лексемой называется последовательность символов, удовлетворяющая следующим условиям: имя в соответствии с синтаксисом Prolog'a, число или отличный от пробела символ.

Пример: преобразование строки в список символов (Два варианта. Варианты отличаются друг от друга граничным условием рекурсии. В первом варианте остановка рекурсии происходит, когда от строки будет отделен последний символ и строка станет пустой строкой. Во втором варианте остановка рекурсии происходит в момент попытки отделения от пустой строки очередного символа, что сделать не удается, и происходит откат ко второму предложению).

```
%1 вариант
DOMAINS
charlist=char*
```

```
PREDICATES
string2charlist (string, charlist)

CLAUSES
string2charlist (“”, [ ]):- !.
string2charlist (Str, [H|T]):- frontchar (Str, H, Str_Rest), string2charlist (Str_Rest, T).

GOAL
string2charlist (“abcde”, List), write (“List=”, List).

%2 вариант
DOMAINS
charlist=char*

PREDICATES
string2charlist (string, charlist)

CLAUSES
string2charlist (Str, [H|T]):- frontchar (Str, H, Str_Rest), !, string2charlist (Str_Rest, T).
string2charlist (_, [ ]).

GOAL
string2charlist (“abcde”, List), write (“List=”, List).
```

Результат работы программы:

```
List=[‘a’, ‘b’, ‘c’, ‘d’, ‘e’]
```

Более подробно стандартные предикаты для работы со строками можно посмотреть в файле Prolog.hlp в разделах “STRING HANDLING” и “CONVERSIONS”.

Основы функционального программирования

Введение

Язык Lisp является языком функционального программирования. В Lisp’е как программы, так и данные, представляются одинаково – в виде списков. Например, запись (+ 1 2) может толковаться, в зависимости от контекста, как список, состоящий из трех элементов (данные), или как вызов функции суммирования с двумя аргументами (программа).

О такой особенности Lisp’а говорит и название языка, ведь аббревиатура Lisp произведена от слов LISt Processing (обработка списков). То есть, программы, написанные на Lisp’е, могут обрабатывать и преобразовывать как другие программы, так и самих себя.

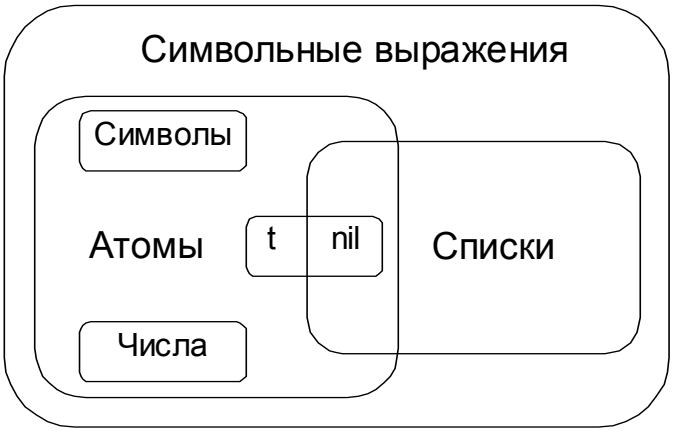
Символьные выражения

При написании программ на Lisp’е используются символы и создаваемые на основе символов символьные выражения. Символ в Lisp’е аналогичен переменной в традиционном языке программирования – это имя, состоящее из букв латиницы, цифр и некоторых специальных литер. Символ, как и переменная, может иметь какое-либо значение, то есть представлять какой-либо объект.

Наряду с символами, в Lisp’е используются также:

1. числа, целые и вещественные;
2. специальные константы t и nil, обозначающие логические значения true (истина) и false (ложь);
3. списки.

Символы, числа и специальные константы представляют простейшие объекты, на основе которых строятся все прочие объекты данных. Поэтому для них используется обобщающее название – атомы. Все вышеперечисленные объекты (атомы и списки) называют символьными выражениями. Отношения между различными символьными выражениями можно представить следующим образом:



Списки

Список – это основной тип данных в Lisp. Список заключается в круглые скобки, элементы списка разделяются пробелами. Пустой список обозначается парой скобок – (). Для обозначения пустого списка используется также специальная константа nil.

Примеры списков:

```
;пятиэлементный список
(1 2 3 4 5)
;четырехэлементный список
(1 2 ((3) 4) 5)
;одноэлементный список
((1 2 3 4 5))
```

Первый элемент списка называется головой списка, все прочие элементы, кроме первого, представленные как список, называются хвостом списка.

Примеры разделения списка на голову и хвост:

Список	Голова	Хвост
(1 2 3 4 5)	1	(2 3 4 5)
(1 2 ((3) 4) 5)	1	(2 ((3) 4) 5)
((1 2 3 4 5))	(1 2 3 4 5)	()
(1)	1	()

Функции

В Lisp’е для вызова функции принята единообразная префиксная форма записи, при которой как имя функции, так и ее аргументы записываются в виде элементов списка, причем имя функции – это всегда первый элемент списка.

В Lisp’е передача параметров в функцию осуществляется по значению. Параметры используются для того, чтобы передать данные в функцию, а результат функции возвращается как ее значение, а не как значение параметра, передаваемого по ссылке.

Например:

```
;возвращаемое значение 8
```

```
(+ 3 5)
;возвращаемое значение 12
(* (+ 1 2) (+ 3 4))
;возвращаемое значение 0
(sin 3.14)
```

Список может рассматриваться и не как вызов функции, а как перечень равноправных элементов. Для блокирования вызова функции используется, в свою очередь, функция quote.

Например, список

```
(+ 3 5)
```

будет восприниматься как вызов функции суммирования с аргументами 3 и 5. Если же использовать данный список в качестве аргумента функции quote

```
(quote (+ 3 5))
```

то список воспринимается именно как список. То есть применение функции quote блокирует вызов функции и ее имя воспринимается как обычный элемент списка. Или, иначе говоря, если список является аргументом функции quote, то первый элемент списка не считается именем функции, а все прочие элементы не считаются аргументами функции.

Для функции quote существует сокращенная форма записи, вместо имени функции quote используется апостроф перед открывающейся скобкой списка. Например:

```
‘(+ 3 5)
```

Если предложить интерпретатору следующие примеры, он вернет соответствующие значения (значение, возвращаемое интерпретатором, написано прописными символами):

```
> (+ 3 5))
8
> (quote (+ 3 5))
(+ 3 5)
> ‘(+ 3 5)
(+ 3 5)
> (quote (quote (+ 3 5)))
(QUOTE (+ 3 5))
```

Для выполнения операции присваивания используется функция set. Формат функции (set variable value). Причем, если не требуется вычисления аргументов, их нужно предварить апострофами. Например:

```
> (set ‘x 5)
5
> x
5
> (set ‘y (+6 12))
18
> y
12
> (set 'a 'b)
B
> (set a 'c)
C
> a
B
> b
C
> c
error: unbound variable - C
```

Вместо функции set можно использовать функцию setq, также выполняющую присваивание, но при ее использовании нет необходимости предварять первый аргумент апострофом. Для первого аргумента блокировка вычисления выполняется автоматически.

```
> (setq x 5)
5
> x
5
> (setq y (+6 12))
18
> y
18
```

Функцию, которая в качестве значения возвращает только константы nil или t, называют предикатом.

Для определения собственной функции можно воспользоваться стандартной функцией defun (сокращение от DEfine FUNction). Эта стандартная функция позволяет создавать собственные функции, причем не запрещается переопределять стандартные функции, то есть в качестве имени собственной функции использовать имя стандартной функции. Функция defun выглядит следующим образом: (defun name (fp1 fp2 … fpN) (form1 form1 … formN)).

Name – это имя новой функции, (fp1 fp2 … fpN) – список формальных параметров, а (form1 form1 … formN) – тело функции, то есть последовательность действий, выполняемых при вызове функции.

Пример – функция для вычисления суммы квадратов:

```
(defun squaresum (x y) (+ (* x x) (* y y)))
```

Результат работы:

```
>(squaresum 3 4)
25
>(squaresum -2 -4)
20
```

Еще один пример: функция deftype, определяющая тип выражения (пустой список, атом или список).

```
(defun deftype(arg)   (cond ((null arg) ‘emptylist) ((atom arg) ‘atom) (t ‘list)))
```

Результат работы:

```
>(deftype ())
EMPTYLIST
>(deftype 'abc)
ATOM
>(deftype '(a b c))
LIST
```

Базовые функции

В Lisp’е существует пять основных функций, называемых базовыми:

1. (car list) – отделяет голову списка (первый элемент списка);
2. (cdr list) – отделяет хвост списка (все элементы, кроме первого, представленные в виде списка);
3. (cons head tail) – соединяет элемент и список в новый список, где присоединенный элемент становится головой нового списка;
4. (equal object1 object2) – проверяет объекты на равенство;
5. (atom object) – проверяет, является ли объект атомом.

Так как функции equal и atom возвращают значения nil или t, их можно назвать базовыми предикатами.

Рассмотрим примеры для перечисленных функций:

```
> (car ‘(one two three))
ONE
> (car ‘(one))
ONE
> (car ‘())
NIL
> (cdr ‘(first second third))
(SECOND THIRD)
> (cdr ‘(first))
NIL
> (cdr ‘())
NIL
> (cons 1 ‘(2 3))
(1 2 3)
> (cons 1 nil)
(1)
> (cons nil nil)
(NIL)
> (equal ‘(1 2 3) ‘(1 2 3))
T
> (equal ‘(1 2 3) ‘(3 2 1))
NIL
> (equal ‘one ‘two)
NIL
> (atom ‘one)
T
> (atom 1)
T
> (atom (1))
NIL
```

Управляющие структуры (предложения)

Для организации различных видов программ в Lisp’е служат разнообразные управляющие структуры, которые позволяют организовывать ветвление, циклы, последовательные вычисления. Перечислим их:

1. предложение let – служит для одновременного присваивания значений нескольким символам;
2. предложения prog1, prog2, prong – используются для организации последовательных вычислений;
3. предложение cond – используется для организации ветвления, и, следовательно, очень важно для организации рекурсии;
4. предложение do – традиционный цикл.

Рассмотрим перечисленные предложения подробнее.

Предложение let служит для одновременного присваивания значений нескольким переменным. Формат предложения let:

(let ((var_1 value_1) (var_2 value_2) … (var_n value_n)) form_1 form_2 … form_m)

Работает предложение следующим образом: переменным var_1, var_2, … var_n присваиваются (параллельно!) значения value_1, value_2, … value_n, а затем вычисляются

(последовательно!) значения форм form_1 form_2 … form_m. В качестве значения всего предложения let возвращается значение последней вычисленной формы form_m.

```
> (let ((x 3) (y 4)) (sqrt (+ (* x x) (* y y))))
5.0
```

Так как присваивание значений переменным выполняется параллельно, то в следующем примере будет выдано сообщение об ошибке.

```
> (let ((x 3) (y (+ 1 x))) (sqrt (+ (* x x) (* y y))))
error: unbound variable - X
```

После завершения выполнения предложения let переменные var_1, var_2, … var_n получают значения, которые они имели до использования в этом предложении, то есть предложение let выполняет локальные присваивания.

```
> (setq x ‘three)
THREE
> (setq y ‘four)
FOUR
> (let ((x 3) (y 4)) (sqrt (+ (* x x) (* y y))))
5.0
> x
THREE
> y
FOUR
```

Для выполнения последовательных действий можно использовать предложения prog1, prog2, prong. Их общий формат:

```
(prog* form_1 form_2 … form_n)
```

Все три предложения работают одинаково, последовательно вычисляются значения форм form_1, form_2, …, form_n. Различие между предложениями проявляется только в тех значениях, которые они возвращают: предложение prog1 возвратит значение первой формы form_1, предложение prog2 возвратит значение второй формы form_2, а предложение progn возвратит значение последней формы form_n. Во всем остальном эти предложения ничем не отличаются.

```
> (prog1 (setq x 2) (setq x (* x 2)) (setq x (* x 2)) (setq x (* x 2)))
2
> (prog2 (setq x 2) (setq x (* x 2)) (setq x (* x 2)) (setq x (* x 2)))
4
> (progn (setq x 2) (setq x (* x 2)) (setq x (* x 2)) (setq x (* x 2)))
16
```

Предложение cond предназначено для организации ветвления (это предложение является аналогом оператора выбора – переключателя switch в языке C). Формат предложения cond выглядит следующим образом:

```
(cond
   (predicate1 form1)
   (predicate2 form21 form22 … form2M)
   (predicate3)
   …
   (predicateN formN)
)
```

При выполнении предложения cond последовательно вычисляются значения предикатов, обозначенных как predicate. Если предикат возвращает значение t, тогда вычисляется значение вычислимой формы form и полученное значение возвращается в качестве значения

всего предложения cond. Другими словами, идет последовательный перебор предикатов до тех пор, пока не встретиться предикат, который будет истинен.

Для некоторого предиката может отсутствовать вычислимая форма. В таком случае предложение cond возвратит значение самого предиката. Для некоторого предиката вычислимых форм может быть несколько. В таком случае формы будут вычислены последовательно, и значение последней формы будет возвращено как значение всего предложения cond.

Пример для предложения cond – определение отрицательного, равного нулю или положительного числа (в этом примере предложения cond вложены одно в другое):

```
(defun snumberp (num)
   (cond
      ((numberp num)
         (cond
            ((< num 0) ‘neg_number)
            ((= num 0) ‘zero)
            ((> num 0) ‘pos_number)
         )
      )
      (t ‘not_number)
   )
)
```

Результат работы программы:

```
> (snumberp 1)
POS_NUMBER
> (snumberp -1)
NEG_NUMBER
>(snumberp 0)
ZERO
> (snumberp 'a)
NOT_NUMBER
```

Как быть в случае, если ни один из предикатов predicate в предложении cond не вернет значение, отличное от nil? Тогда используется прием, как в последнем примере. В качестве последнего предиката predicate используется константа t, что гарантирует выход из предложения cond с помощью последней ветви (использование константы t в качестве предиката predicate аналогично использованию метки default в переключателе switch).

Предложение do, также как и предложение cond, является аналогом оператора цикла for в языке C. Формат предложения do:

```
(do
   ((var_1 value_1) (var_2 value_2) … (var_n value_n))
   (condition form_yes_1 form_yes_2 … form_yes_m)
   form_no_1 form_no_2 … form_yes_k
)
```

Предложение do работает следующим образом: первоначально переменным var_1, var_2, …, var_n присваиваются значения value_1, value_2, …, value_n (параллельно, как в предложении let). Затем проверяется условие выхода из цикла condition. Если условие выполняется, последовательно вычисляются формы form_yes_1, form_yes_2, …, form_yes_m, и значение последней вычисленной формы form_yes_m возвращается в качестве значения всего предложения do. Если же условие condition не выполняется, последовательно вычисляются формы form_no_1, form_no_2, …, form_yes_k, и вновь выполняется переход в проверке условия выхода из цикла condition.

Пример использования предложения do: для возведения x в степень n с помощью умножения определена функция power с двумя аргументами x и n: x – основание степени, n – показатель степени.

```
> (defun power (x n)
   (do
;присваивание начального значения переменной result
      ((result 1))
;условие выхода их цикла
      ((= n 0) result)
;повторяющиеся действия
      (setq result (* result x)) (setq n (- n 1))))
POWER
> (power 2 3)
8
```

Простая рекурсия

Несмотря на то, что в языке Lisp есть предложение для организации циклических действий, все же основным методом решения задач остается метод с использованием рекурсии, то есть с применением рекурсивных функций.

Функция является рекурсивной, если в ее определении содержится вызов этой же функции. Рекурсия является простой, если вызов функции встречается в некоторой ветви лишь один раз. Простой рекурсии в процедурном программировании соответствует обыкновенный цикл. Например, задача нахождения значения факториала n! сводится к нахождению значения факториала (n-1)! и умножения найденного значения на n.

Пример: нахождение значения факториала n!.

```
> (defun factorial (n)
   (cond
;факториал 0! равен 1
      ((= n 0) 1)
;факториал n! равен (n-1)!*n
      (t (* (factorial (- n 1)) n))))
FACTORIAL
>(factorial 3)
6
```

Для отладки программы можно использовать возможности трассировки. Трассировка позволяет проследить процесс нахождения решения.

Для того чтобы включить трассировку можно воспользоваться функцией trace:

Например:

```
> (trace factorial)
(FACTORIAL)

> (factorial 3)
Entering: FACTORIAL, Argument list: (3)
   Entering: FACTORIAL, Argument list: (2)
      Entering: FACTORIAL, Argument list: (1)
         Entering: FACTORIAL, Argument list: (0)
         Exiting: FACTORIAL, Value: 1
      Exiting: FACTORIAL, Value: 1
   Exiting: FACTORIAL, Value: 2
Exiting: FACTORIAL, Value: 6
```

```
6
```

Для отключения трассировки можно воспользоваться функцией untrace:

Например:

```
> (untrace factorial)
NIL
```

Можно говорить о двух видах рекурсии: рекурсии по значению и рекурсии по аргументу. Рекурсия по значению определяется в случае, если рекурсивный вызов является выражением, определяющим результат функции. Рекурсия по аргументу существует в функции, возвращаемое значение которой формирует некоторая нерекурсивная функция, в качестве аргумента которой используется рекурсивный вызов.

Приведенный выше пример рекурсивной функции вычисления факториала является примером рекурсии по аргументу, так как возвращаемый результат формирует функция умножения, в качестве аргумента которой используется рекурсивный вызов.

Вот несколько примеров простой рекурсии.

Возведение числа x в степень n с помощью умножения (рекурсия по аргументу):

```
> (defun power (x n)
   (cond
;x^0=1 (любое число в нулевой степени равно 1)
      ((= n 0) 1)
;x^n=x^(n-1)*n (значение x в степени n вычисляется возведением x в степень n-1
;и умножением результата на n)
      (t (* (power (- n 1)) n))))
> (power 2 3)
8
> (power 10 0)
1
```

Копирование списка (рекурсия по аргументу):

```
> (defun copy_list (list)
   (cond
;копией пустого списка является пустой список
      ((null list) nil)
;копией непустого списка является список, полученный из головы и копии
;хвоста исходного списка
      (t (cons (car list) (copy_list (cdr list))))))
COPY_LIST
>(copy_list '(1 2 3))
(1 2 3)
>(copy_list ())
NIL
```

Определение принадлежности элемента списку (рекурсия по значению):

```
> (defun member (el list)
   (cond
;список просмотрен до конца, элемент не найден
      ((null list) nil)
;очередная голова списка равна искомому элементу, элемент найден
      ((equal el (car list)) t)
;если элемент не найден, продолжить его поиск в хвосте списка
      (t (member el (cdr list)))))
MEMBER
> (member 2 '(1 2 3))
```

```
T
> (member 22 '(1 2 3))
NIL
```

Соединение двух списков (рекурсия по аргументу):

```
> (defun append (list1 list2)
   (cond
;соединение пустого списка и непустого дает непустой список
      ((null list1) list2)
;соединить голову первого списка и хвост первого списка,
;соединенный со вторым списком
      (t (cons (car list1) (append (cdr list1) list2)))))
APPEND
> (append '(1 2) '(3 4))
(1 2 3 4)
> (append '(1 2) ())
(1 2)
> (append () '(3 4))
(3 4)
> (append () ())
NIL
```

Реверс списка (рекурсия по аргументу):

```
> (defun reverse (list)
   (cond
;реверс пустого списка дает пустой список
      ((null list) nil)
;соединить реверсированный хвост списка и голову списка
      (t (append (reverse (cdr list)) (cons (car list) nil)))))
REVERSE
> (reverse '(one two three))
(THREE TWO ONE)
> (reverse ())
NIL
```

Другие виды рекурсии

Рекурсию можно назвать простой, если в функции присутствует лишь один рекурсивный вызов. Такую рекурсию можно назвать еще рекурсией первого порядка. Но рекурсивный вызов может появляться в функции более, чем один раз. В таких случаях можно выделить следующие виды рекурсии:

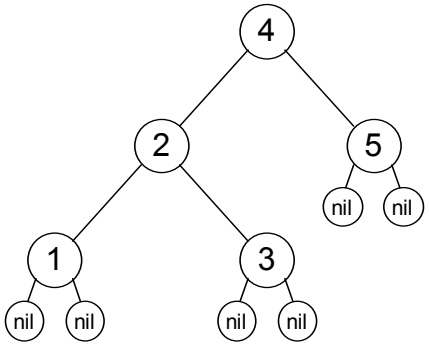
1. параллельная рекурсия – тело определения функции function_1 содержит вызов некоторой функции function_2, несколько аргументов которой являются рекурсивными вызовами функции function_1.
   (defun function_1 … (function_2 … (function_1 …) … (function_1 …) …) …)
2. взаимная рекурсия – в теле определения функции function_1 вызывается некоторая функции function_2, которая, в свою очередь, содержит вызов функции function_1.
   (defun function_1 … (function_2 …) …)
   (defun function_2 … (function_1 …) …)
3. рекурсия более высокого порядка – в теле определения функции аргументом рекурсивного вызова является рекурсивный вызов.
   (defun function_1 … (function_1 … (function_1 …) …) …)

Рассмотрим примеры параллельной рекурсии. В разделе, посвященном простой рекурсии, уже рассматривался пример копирования списка (функция copy_list), но эта функция не

выполняет копирования элементов списка в случае, если они являются, в свою очередь также списками. Для записи функции, которая будет копировать список в глубину, придется воспользоваться параллельной рекурсией.

```
> (defun full_copy_list (list)
   (cond
;копией пустого списка является пустой список
      ((null list) nil)
;копией элемента-атома является элемент-атом
      ((atom list) list)
;копией непустого списка является список, полученный из копии головы
;и копии хвоста исходного списка
      (t (cons (full_copy_list (car list)) (full_copy_list (cdr list))))))
FULL_COPY_LIST
> (full_copy_list '(((1) 2) 3))
(((1) 2) 3)
> (full_copy_list ())
NIL
```

Не обойтись без параллельной рекурсии при работе с бинарными деревьями. Бинарное дерево, как и все прочие данные, представляются в Lisp’е в виде списков. Например, упорядоченное бинарное дерево



можно представить в виде списка (4 (2 (1 nil nil) (3 nil nil)) (5 nil nil)). Константы nil представляют пустые деревья. В таком представлении первый элемент списка – это узел дерева, второй элемент списка – левое поддерево и третий элемент списка – правое поддерево. Другой вариант представления дерева– (((nil 1 nil) 2 (nil 3 nil)) 4 (nil 5 nil)). В таком представлении первый элемент списка – это левое поддерево, второй элемент списка – узел дерева и третий элемент списка – правое поддерево. Можно использовать и другие варианты представления деревьев. Рассмотрим простой пример работы с бинарным деревом – обход дерева и подсчет числа узлов дерева. Для работы с элементами дерева, которые являются, по сути, элементами списка, очень удобно использовать стандартные функции Lisp’a, для получения первого, второго и третьего элементов списка – fist, second и third, соответственно.

```
> (defun node_counter (tree)
   (cond
;число узлов пустого дерева равно 0
      ((null tree) 0)
;число узлов непустого дерева складывается из: одного корня,
;числа узлов левого поддерева и числа узлов правого поддерева
      (t (+ 1 (node_counter (second tree)) (node_counter (third tree))))))
NODE_COUNTER
> (node_counter '(4 (2 (1 nil nil) (3 nil nil)) (5 nil nil)))
5
```

```
> (node_counter nil)
0
```

Пример взаимной рекурсии – реверс списка. Так как рекурсия взаимная, в примере определены две функции: reverse и rearrange. Функция rearrange рекурсивна сама по себе.

```
> (defun reverse (list)
   (cond
      ((atom list) list)
      (t (rearrange list nil)))))
REVERSE
> (defun rearrange (list result)
   (cond
      ((null list) result)
      (t (rearrange (cdr list) (cons (reverse (car list)) result)))))
REARRANGE
> (reverse '(((1 2 3) 4 5) 6 7))
(7 6 (5 4 (3 2 1)))
```

Пример рекурсии более высокого порядка – второго. Классический пример функции с рекурсией второго порядка – функция Аккермана.

Функция Аккермана определяется следующим образом:

```
B (0, n) = n+1
B (m, 0) = B (m-1, 0)
B (m, n) = B (m-1, B (m, n-1))
```

где m>=0 и n>=0.

```
> (defun ackerman
   (cond
      ((= n 0) (+ n 1))
      ((= m 0) (ackerman (- m 1) 1))
      (t (ackerman (- m 1) (ackerman m (- n 1))))))
ACKERMAN
> (ackerman 2 2)
7
> (ackerman 2 3)
9
> (ackerman 3 2)
29
```

Литература

1. Адаменко А.Н., Кучуков А.М. Логическое программирование и Visual Prolog. – СПб.: БХВ-Петербург, 2003. – 992 С.
2. Братко И. Программирование на языке Пролог для искусственного интеллекта. – М.: Мир, 1990. – 560 С.
3. Ин Ц., Соломон Д. Использование Турбо-Пролога. – М.: Мир, 1993. – 608 С.
4. Доорс Дж., Рейблейн А.Р., Вадера С. Пролог - язык программирования будущего. – М.: ФиС, 1990. – 144 С.
5. Стерлинг Л., Шапиро Э. Искусство программирования на языке Пролог. – М.: Мир, 1990. – 235 С.
6. Клоксин У., Меллиш Д. Программирование на языке Пролог. – М.: Мир, 1987. – 336 С.
7. Стобо Дж. Язык программирования Пролог. – М.: Мир, 1993. – 368 С.
8. Хювёнен Э., Сеппянен Й. Мир Лиспа. - М.: Мир, 1990. – 447 С.
9. Хендерсон П. Функциональное программирование: применение и реализация. - М.: Мир, 1983. – 349 С.
10. Малпас Дж. Реляционный язык Пролог и его применение. – М.: Наука, 1990. – 463 С.
11. Янсон А. Турбо-Пролог в сжатом изложении. – М.: Мир, 1991. – 94 С.
12. Маурер У. Введение в программирование на языке ЛИСП. – М.: Мир, 1978. – 104 С.