5.4. Прерывание поиска с возвратом

В некоторых ситуациях необходимо иметь доступ только к определённой части данных. Это особенно актуально, когда существует большое число условий для выборки. Чтобы управлять поиском решений, удовлетворяющих некоторым условиям, требуется средство управления откатом.

Для ограничения автоматического перебора предусмотрен *предикат прерывания поиска с возвратом* **cut**, который **устанавливает запрет** на поиск альтернативных решений текущей цели. Этот предикат является обратным предикату **fail** и используется для того, чтобы выполнить отсечение в указанном месте с целью устранения **всех** последующих откатов.

Формат предиката прерывания поиска с возвратом:

!

Предикат **cut всегда успешен**, но не имеет эффекта, пока не будет выполнен реально.

Когда процесс поиска решений проходит через отсечение, реализуется обращение к следующей подцели (если таковая имеется).

Однажды пройдя предикат отсечения:

- **невозможно** произвести поиск с возвратом в подцелях, расположенных *перед* отсечением в обрабатываемом предложении;
- **невозможно** возвратиться к *альтернативным* логическим предложениям процедуры, которая определяет обрабатываемый предикат.

Отсечение обеспечивает детерминированное решение при поиске цели.

Детерминированный предикат – предикат, обращение к которому производит только одно единственное решение.

Недетерминированный предикат – предикат, производящий множественные решения.

Пример 5-2. Проверка вхождения элемента в список.

```
domains    list = name*      name = symbol
predicates member (name, list)
clauses    member (Name, [Name | _]) :- !.
           member ( Name, [ _ | T ] ) :-
                 member (Name, T).
goal       readln (X), member ( X, [a, b, c, b, d, b]).
```

Отсечение в первом правиле предиката **member** обеспечивает прерывание автоматического перебора сразу же после первого совпадения очередной головы с искомым элементом. Именно поэтому любой запрос даст *единственное решение*.

Цель 3 из Примера 3-18:

goal X = tom, member (X, [tom, bill, ann, tom]).
X = tom
1 Solution

Цель 4 из Примера 3-18:

goal member (X, [tom, bill, ann, susan]).
X = tom
1 Solution

Цель из приведённого в Примере 5-2 текста программы:

1. Если введённое значение **X = b**

   X = b
   1 Solution

2. Если введённое значение **X = s**

   No Solution

5.5. Зелёное и красное отсечения

Пусть существует процедура:

```
P :- A, B.      %1
P :- C.         %2
```

P истинно тогда и только тогда, когда одновременно истинны **A** и **B**, **ИЛИ** истинно **C**.

Для этого утверждения справедлива логическая формула

$$P \leftarrow (A \wedge B) \vee C$$

Эту формулу следует читать так:

***P** логически следует из формулы **(A ∧ B) ∨ C**, если **P** имеет значение истина в интерпретациях, при которых формула **(A ∧ B) ∨ C** имеет значение истина.*

Можно поменять местами предложения %1 и %2, но *декларативный смысл не изменится*:

$$P \leftarrow C \vee (A \wedge B)$$

Отсечение, применение которого не меняет декларативный смысл программы, называют зелёным.

При чтении программы зелёное отсечение можно игнорировать.

Пусть теперь в правило % 1 добавлено отсечение:

```
P :- A, !, B.   %1
P :- C.         %2
```

Логическая формула процедуры (согласно определению отсечения):

$$P \leftarrow (A \wedge B) \vee (\neg A \wedge C)$$

Если поменять предложения %1 и %2 местами, изменится и логическая формула, и декларативный смысл:

```
P :- C.
P :- A, !, B.
```

Декларативный смысл изменился:

$$P \leftarrow C \vee (A \wedge B)$$

Отсечение, влияющее на декларативный смысл программы, называют красным.

При использовании красного отсечения следует избегать ошибок:

- отсечение путей вычисления, которые нельзя отбрасывать;
- не отсечение тех решений, которые должны были быть отброшены.

Существует две основные причины применения отсечения в программе:

1. Заранее известно, что определённые посылки никогда не приведут к осмысленным решениям. В этом случае говорят о зелёном отсечении, которое соответствует правилу вида:

   ЕСЛИ Условие ТО Утверждение$_1$ ИНАЧЕ Утверждение$_2$

2. Логика программы требует исключения из рассмотрения альтернативных подцелей. В этом случае имеет место красное отсечение.

Следует помнить, когда для поиска решения существуют взаимоисключающие условия, всегда следует пользоваться предикатом отсечения.

Пример 5-3. Фильтрация списка. Зелёное отсечение.

```
domains    list = integer*
predicates disNeg (list, list)
clauses    disNeg ([ ], [ ]).
           disNeg ([H | T], NewList) :-
                 H <= 0,
                 !,
                 disNeg (T, NewList).
           disNeg ([H | T], [H | NewList]) :-
                 disNeg (T, NewList).
goal       disNeg ([2, -45, -3, 468], X ), write (X), exit.
```

Предикат отсечения, использованный после проверки головы **H <= 0**, предотвращает перебор с возвратами, поэтому процедура **disNeg** имеет единственное решение **[2, 468]**. Ранее рассматривался *Пример 3-16*, где

процедура **disNeg** была недетерминированной, и при отсутствии предиката **exit** могла произвести дополнительные, но неверные решения.

Пример 5-4. Поиск максимума. Зелёное отсечение.

```
domains     x, y, max = integer
predicates  max (max, x, y)
clauses     max (MAX, X, Y) :- X >= Y, MAX = X, !.
            max (MAX, X, Y) :- X < Y, MAX = Y.
goal        max (Max, 5, 2).
```

Для поиска решения цели **Max = 5** используется первое предложение процедуры, второе правило из-за отсечения рассматриваться не будет. Если в процедуре **max** поменять местами правила, декларативный смысл программы не изменится.

Пример 5-5. Поиск максимума. Красное отсечение.

```
domains     x, y, max = integer
predicates  max (max, x, y)
clauses     max (MAX, X, Y) :- X >= Y, MAX = X, !.
            max (MAX, _, Y) :- MAX = Y, !.
goal        max (Max, 5, 2).
```

Первое правило проверяет отношение **X >= Y**. Если проверка отношения терпит неудачу, то по второму правилу делается вывод, что максимальным является значение переменной **Y**. Результат этой программы будет верным **Max = 5**. Второе правило не рассматривается из-за отсечения в первом.

Однако если поменять правила местами, для той же самой цели будет получен неверный результат **Max = 2**, и отсечение будет красным.

Пример 5-6. Красное отсечение.

```
predicates  action (integer)
clauses     action (1) :- !, write ("\n1\n").
            action (2) :- !, write ("\n2\n").
            action (3) :- !, write ("\n3\n").
            action (_) :- write ("Excuse me\n").
goal        readint (X), action (X).
```

Порядок правил здесь имеет значение. Необходима уверенность, что не возникнет попытка выполнить правило с заголовком **action (_)** раньше, чем будут испробованы и не выполнят отсечений все предыдущие правила. Кроме того, перемещение правила с анонимной переменной в заголовке в любое место выше делает предикат **action** недетерминированным.

5.6. Отрицание

Пусть необходимо определить на *Прологе* предложение естественного языка: *"Марку нравятся все животные, кроме змей"*.

Первая половина этой фразы легко определяется следующим образом:

"Марку нравится любой **X**, если **X** – животное".

Правило, составленное согласно фразе:

нравится (марк, X) :- животное (X).

Чтобы исключить *змей*, можно воспользоваться другой формулировкой:

"Если **X** – **змея**, то **нравится (марк, X) не есть истина**, иначе если **X** – животное, то Марку нравится **X** ".

Утверждение **"нечто не есть истина"** определяется на *Прологе* при помощи предиката **fail**. Попытка достичь цели **fail** всегда терпит неудачу, заставляя потерпеть неудачу и ту цель, которая является её родителем.

Последняя формулировка, переведённая на язык *Пролог*, имеет вид:

```
нравится (марк, X) :-
        змея (X),     % проверка аргумента на принадлежность к змеям
        !,            % запрет поиска альтернатив
        fail.         % вызов неудачи

нравится (марк, X) :-
        % проверка аргумента на принадлежность к животным
        животное (X).
```

Отношение отрицания реализовано в *Пролог*-системе как унарный предикат отрицания **not**. Формат предиката:

not (GOAL)

, где **GOAL** – отрицаемое утверждение.

Предикат **not** успешен, если *невозможно доказать истинность* входящей в него *цели*.

Для доказательства цели **все переменные** (если таковые имеются) должны быть **связаны**, как того требуют правила согласования цели.

5.7. Оптимизация хвостовой рекурсии

С точки зрения исполнения рекурсия требует объёма памяти, линейно зависящего от числа выполняемых рекурсивных обращений.

Существует метод, приводящий к экономии памяти, который называется *оптимизацией хвостовой рекурсии*. Оптимизированная процедура может вызвать себя без сохранения информации о своём состоянии.

Процедура обладает хвостовой рекурсией, если:

1. Вызов является самой последней подцелью предложения.
2. Ранее в предложении не было точек возврата.

Далее рассматриваются варианты процедуры для печати последовательности натуральных чисел.

Вариант 1. Результаты процедуры не являются безупречными, однако она обладает хвостовой рекурсией, поскольку вызов – последняя подцель правила, и в процедуре отсутствуют точки возврата.

```
predicates  count (integer)
clauses     count (N) :-
                nl, write (N), NewN = N + 1,
                count (NewN).
goal        count (1).
```

Вариант 2. Отсутствие хвостовой рекурсии. Произойдёт переполнение стека, так как из-за небрежности программирования в процедуре есть точка возврата из-за необходимости выполнения предиката **nl** в последней подцели.

```
badCount1 (N) :-
        write (N), NewN = N + 1,
        badCount1 (NewN ),
        nl.
```

Вариант 3. Не оптимизированная процедура. Невозможно освободить стек, так как рекурсивный вызов происходит ещё тогда, когда не проверено второе альтернативное предложение. Происходит истощение оперативной памяти.

```
badCount2 (N) :-
        nl, write (N), NewN = N + 1,
        badCount2 (NewN).
badCount2 (N) :-        % альтернатива
        N < 0,
        write ("\nIs negative ", N).
```

Для оптимизации процедуры используется отсечение:

```
goodCount2 (N) :-
        N > 0,          % проверка
        !,              % отсечение
        nl, write (N),
        NewN = N + 1,
        goodCount2 (NewN).
googCount2 (N) :-       % альтернатива
        N < 0,
        write ("\nIs negative ", N).
```

Вариант 4. Не оптимизированная процедура из-за наличия точки возврата до рекурсивного вызова. В процедуре имеются два предложения для проверки числа. На момент вызова рекурсивной процедуры не будет проверено второе альтернативное правило процедуры **check**.

```
badCount3 (N) :-
        nl, write (N),
        NewN = N + 1,
        check (NewN),          % точка возврата
        badCount3 (NewN).
% 1-ое правило
check (X) :- X > 1.
% 2-ое правило
check (X) :- X < 1.
```

Для оптимизации процедуры используется отсечение:

```
goodCount3 (N) :-
        nl, write (N),
        NewN = N + 1,
        check (NewN),
        !,                     % отсечение
        goodCount3 (NewN).
% 1-ое правило
check (X) :- X > 1.
% 2-ое правило
check (X) :- X < 1.
```

5.8. Процедуры сортировки списка

Рассматриваются процедуры, в которых реализованы разные методы сортировки списка по возрастанию.

Декларативный смысл: элементы исходного списка упорядочены по возрастанию, если каждый следующий элемент больше предыдущего.

Пример 5-7. Метод наивной сортировки.

Процедурный смысл: чтобы отсортировать список, необходимо произвольным образом переставить в нём элементы, и проверить порядок следования элементов в получившемся списке.

```
domains
        list = integer*

predicates
        sort (list, list)                   % сортировка списка
        nondeterm permutation (list, list)  % перемешивание
        check (list)                        % проверка списка
        nondeterm append (list, list, list) % объединение списков
        % проверка порядка следования
        order (integer, integer)
```

```
clauses
        % сортировка списка (список In сортируется в список Out)
        sort (In, Out) :-
                % элементы списка In перемешиваются в Out
                permutation (In, Out),
                % проверяется, отсортирован ли список Out
                check (Out), !.

        % перемешивание элементов списка
        permutation ([ ], [ ]) :- !.
        permutation (L, [H | T]) :-
                append (V, [H | U], L),  % поток данных – (i, [o|o], o)
                append (V, U, W),         % поток данных – (i, i, o)
                permutation (W, T).

        % проверка списка на упорядоченность
        % список из одного элемента всегда отсортирован
        check ([ _ ]).
        % список упорядочен,
        % если первый элемент меньше или равен второму и
        % остаток списка, начиная со второго элемента, тоже упорядочен
        check ([X, Y | T]) :- order (X, Y), check ([Y | T]).

        % процедура проверки порядка следования элементов
        order (X, Y) :- X <= Y.

        % процедура объединения списков
        % используется для перемешивания элементов
        append ([ ], L, L).
        append ([X | L1], L2, [X | L3]) :- append (L1, L2, L3).

goal
        sort ([-299, 5, 8, -7, 1, 22, -8, -100, 55, 18, 99], List),
        write (List),
        exit.
```

Пример 5-8. Метод пузырька.

Декларативный смысл: если в списке нет ни одной пары смежных элементов **X** и **Y** таких, что выполняется условие **X > Y**, то считать, что список отсортирован по возрастанию.

Процедурный смысл: чтобы отсортировать список, необходимо:

- найти в исходном списке такие смежные элементы **X** и **Y**, что **X > Y**;
- поменять их местами и получить новый список;
- отсортировать полученный новый список.

```
domains     list = integer*
predicates
        bubble (list, list)             % сортировка списка
        swap (list, list)               % поиск смежных элементов
        order (integer, integer)        % проверка порядка следования
```

```
clauses
        % сортировка списка
        bubble (In, Out) :-
                swap (In, L), !,    % поиск в списке In смежных элементов
                bubble (L, Out).    % сортировка нового списка
        % в списке нет ни одной пары смежных элементов,
        % значит список отсортирован
        bubble (Out, Out) :- !.

        % поиск смежных элементов (предыдущий больше следующего)
        % если X > Y, то они меняются местами согласно заголовку
        swap ([X, Y | R], [Y, X | R]) :- order (X, Y), !.
        % если X ≤ Y, то X становится головой отсортированного списка
        % сортируется хвост исходного списка
        swap ([X | R], [X | R1]) :- swap (R, R1).

        % проверка порядка следования элементов
        order ( X, Y ) :- X > Y.

goal
        bubble ([-299, 5, 8, -7, 1, 22, -8, -100, 55, 18, 99, -777], List),
        write (List), exit.
```

Пример 5-9. Метод вставки.

Процедурный смысл: чтобы упорядочить непустой список необходимо:

- упорядочить хвост исходного списка в новый список;
- вставить голову исходного списка в упорядоченный хвост, поместив её таким образом, чтобы новый получившийся список остался упорядоченным.

```
domains     list = integer*
predicates
        insort (list, list)             % сортировка списка
        order (integer, integer)        % проверка порядка следования
        % добавление элемента в упорядоченный список
        insortx (integer, list, list)
clauses
        % сортировка списка
        insort ([ ], [ ]).
        insort ([X | Rest], Out) :-
                % сортировка хвоста
                insort (Rest, SortRest),
                % вставка головы списка в упорядоченный хвост
                insortx (X, SortRest, Out).

        % добавление элемента в упорядоченный хвост
        insortx (X, [A | Rest], [A | Out]) :-
                order (A, X), !,
                insortx (X, Rest, Out).
        insortx (X, Rest, [X | Rest]).
```

```
        % проверка порядка следования элементов
        order (X, Y) :- X <= Y.

goal    insort ([-299, 5, 8, -7, 1, 22, -8, -100, 55, 18, 99, -777], List),
        write (List), exit.
```

Пример 5-10. Метод быстрой сортировки (метод Хоара).

Процедурный смысл: чтобы отсортировать список, необходимо:

- отделить голову исходного списка;
- хвост исходного списка разбить на два так, что в 1-ый список помещаются элементы, которые меньше или равны голове, во второй – больше головы;
- полученные списки сортируются точно так же, как исходный список;
- результирующий список получается соединением отсортированного первого списка с головой исходного списка и отсортированным вторым списком.

```
domains     list = integer*
predicates
        sort (list, list)                   % сортировка списка
        split (integer, list, list, list)   % разбиение списка на два списка
        append (list, list, list)           % объединение списков
        order (integer, integer)            % проверка порядка следования
clauses
        % сортировка списка
        sort ([ ], [ ]) :-!.
        sort ([H | T], Out) :-
                split (H, T, Small, Big),  % разбиение хвоста на два списка
                sort (Small, SortSmall),        % сортировка первого списка
                sort (Big, SortBig),            % сортировка второго списка
                % объединение первого отсортированного списка
                % с головой и вторым отсортированным списком
                append (SortSmall, [H | SortBig], Out).

        % разбиение списка на два списка
        split (H, [A | T], [A | Small], Big) :-        % A ≤ H
                order (A, H), !,
                split (H, T, Small, Big).
        split (H, [A | T], Small, [A | Big]) :-        % A > H
                split (H, T, Small, Big).
        split (_, [ ], [ ], [ ]).

        % проверка порядка следования элементов
        order (X, Y) :- X <= Y.

        % объединение списков
        append ([ ], L, L).
        append ([X | L1], L2, [X | L3]) :- append (L1, L2, L3).

goal    sort ([5, 3 , 7, 8, -1, 4, -7, 6], List), write (List), exit.
```