Севастопольский государственный университет
Кафедра «Информационные системы»

Курс лекций по дисциплине

"МЕТОДЫ И СИСТЕМЫ ИСКУССТВЕННОГО ИНТЕЛЛЕКТА"

(МиСИИ)

Лектор: Бондарев Владимир Николаевич

Лекция17

Списки и рекурсия. Управление возвратом. Отрицание. Метаусловия.

В.Бондарев

Списки и рекурсия

Список – это структура данных, составленная их произвольного числа элементов. Элементы списка отделяются друг от друга запятыми и заключаются в квадратные скобки: **[a, b, c, d].**

Для представления списка в виде структуры данных, состоящей из **головы и хвоста**, в Прологе широко используется еще одно обозначение, в котором голова и хвост списка отделяются вертикальной чертой. Например, в записи **[H|T]** переменная **H** – представляет голову списка, а переменная **T** – хвост. Применив символ "|", список **[a, b, c, d]** можно представить следующими различными способами:

[a,b,c,d]=[a| [b,c,d]]=[a, b| [c,d]]=[a, b, c| [d]]=[a, b, c, d| [ ]].

Здесь пара квадратных скобок **[ ]** обозначает пустой список.

? – [X1, X2, X3, X4]=[1, 2, 3, 4].
X1=1, X2=2, X3=3, X4=4.

? – [X1, X2| X3]=[1, 2, 3, 4].
X1=1, X2=2, X3=[3, 4].

? – [X1, X2, X3 | X4]=[1, 2, 3, 4].
X1=1, X2=2, X3=3, X4=[4].

? – [X1 | X2]=[1, 2, 3, 4].
X1=1, X2=[2, 3, 4].

Списки и рекурсия

Над списками часто выполняют следующие **операции**: добавление элемента в список, удаление элемента из списка, объединение списков, поиск элемента в списке.

Добавление элемента в список: **добавить (X, L, [X|L]).**

Здесь **X** – добавляемый элемент; **L** – список, в который добавляется элемент; **[X|L]** – результирующий список. Таким образом, элемент **X** добавляется в начало списка **L**.

Представление списков в виде головы и хвоста, где хвост, в свою очередь, тоже список, является рекурсивным. Поэтому обработка списков часто выполняется с помощью **рекурсивных предикатов**.

Рекурсивными называют предикаты, в определениях которых содержатся ссылки на самих себя.

?-добавить(а, [б, в, г], СПИСОК).
СПИСОК = [а, б, в, г].
?-добавить(а, СПИСОК, [а, б, в, г]).

Списки и рекурсия

?-добавить(а, СПИСОК, [а, б, в, г]).
СПИСОК = [б, в, г].

?-добавить(ЧТО, [б, в, г], [а, б, в, г]).

Списки и рекурсия

?-добавить(ЧТО, [б, в, г], [а, б, в, г]).
ЧТО = а.

Списки и рекурсия

Предикат, проверяющей **вхождение элемента H** в список:

member(H,[H|T]).

member(H,[X|T]):-member(H,T).

С помощью факта задается истинное утверждение о том, что элемент **H** и список **[H|T],** головной элемент которого есть **H**, находятся в отношении **member.** Правило означает, что элемент **H** и список **[X|T]** будут находиться в отношении **member**, если указанное отношение имеет место между элементом **H** и хвостом списка **T**. Иными словами, элемент **H** содержится в списке **[X|T]**, если он входит в хвост **T** этого списка.

?-member(a, [a, b, c]).
true.
?-member(a, [b, c]).
false.
?-member(X, [b, c]).

Списки и рекурсия

```
member(H,[H|T]).
member(H,[X|T]):-member(H,T).
```

```
?-member(X, [b, c]).
X = b;
X = c;
false.
```

Списки и рекурсия

Пусть требуется написать программу, выполняющую **соединение двух списков** и возвращающую в качестве результата третий список. Определим для этого предикат (отношение) **append(X,Y,Z),** где **X** и **Y** – исходные списки, а **Z** – результирующий список.

При описании отношения **append** необходимо учесть два случая:

1) если **X** представляет пустой список, то второй и третий аргумент (т. е. **Y** и **Z**) отношения представляют собой один и тот же список, что выражается в виде факта

append([ ],L,L);

2) если **X** не пустой список, то он имеет голову и хвост и может быть записан в виде **[H|T];** в результате соединения такого списка со списком **Y**, получим новый список **[H|W],** где между хвостом **T** первого списка, списком **Y** и хвостом **W** результирующего списка должно существовать отношение **append(T,Y,W).** Это записывается в виде правила:

append([H|T],Y,[H|W]):-append(T,Y,W).

Списки и рекурсия

Иными словами, списки **[H|T], Y**, и **[H|W]** находятся в отношении **append,** если в этом же отношении находятся списки **T,Y и W**.
Пролог-программа, решающая поставленную задачу:

```
append([ ],L,L).
append([H|T],Y,[H|W]):-append(T,Y,W).
```

?-append([a, b], [c, d], [a, b, c, d]).

?-append([a, b], [c, d], [a, a, c, d]).

?-append([a, b], [c, d], RES).

Списки и рекурсия

```
append([ ],L,L).
append([H|T],Y,[H|W]):-append(T,Y,W).
```

```
?-append([a, b], [c, d], RES).
  RES = [a, b, c, d].
```

```
?-append(X, [c, d], [a, b, c, d]).
```

Списки и рекурсия

```
append([ ],L,L).
append([H|T],Y,[H|W]):-append(T,Y,W).
```

```
?-append(X, [c, d], [a, b, c, d]).
   X = [a, b].
```

```
?-append(X1, X2, [a, d]).
```

Списки и рекурсия

```
append([ ],L,L).
append([H|T],Y,[H|W]):-append(T,Y,W).
```

```
?-append(X1, X2, [a, d]).
  X1 = [],  X2 = [a, d];
  X1 = [a],  X2 = [d];
  X1 = [a, d], X2 = []|;
  false.
```

Списки и рекурсия

В общем случае все такие определения строятся по следующей схеме:

```
предикат(…[  ]…).
предикат(…[Голова|Хвост]…): –
        обработка(Голова),
        предикат(…[Хвост]…).
```

Рекурсивные вызовы прекратятся, когда хвост списка окажется пустым списком. В приведенном определении первое утверждение определяет условие выхода из рекурсии, а второе утверждение – правило, предусматривающее при каждом вызове обработку очередного элемента списка и рекурсивный вызов определяемого предиката, аргументом которого является хвост списка.

Списки и рекурсия

Определим, например, предикат **реверс(X,Y)**, где список **Y** представляет **инверсную копию списка X**:

реверс([ ], [ ]).
реверс([H|T],Y):- реверс(T,Ys), append(Ys,[H],Y).

Для того чтобы выполнить инверсию списка **[H|T]**, необходимо выполнить инверсию хвоста **T** и получить список **Ys**, а затем добавить голову **H** в конец списка **Ys** и получить результирующий список **Y**.

?-реверс([a, b, c], [c, b, a]).

?-реверс([a, b, c], [a, b, c]).

?-реверс([a, b, c], X).

?- реверс(X, [c, b, a]).

Списки и рекурсия

```
реверс([ ], [ ]).
реверс([H|T],Y):- реверс(T,Ys), append(Ys,[H],Y).
```

```
?-реверс([a, b, c], [c, b, a]).
  true.
```

```
?-реверс([a, b, c], [a, b, c]).
  false.
```

```
?-реверс([a, b, c], X).
X = [c, b, a].
```

```
?- реверс(X, [c, b, a]).
X = [a, b, c].
```

Списки и рекурсия

Удаление элемента **X** из списка **L** можно представить в виде предиката **удалить(X, L, L1),** где **L1** – результирующий список. Если **X** является головой списка **L**, то результирующий список **L1** – это хвост списка **L**. Если **X** находится в хвосте списка **L**, то для удаления **X** необходимо рекурсивно вызвать предикат **удалить**, подставив в качестве второго аргумента хвост списка **L**.

```
удалить(X, [X|T], T).
удалить(X, [H|T], [H|T1]): – удалить(X, T, T1).
```

```
?-удалить(a,[a,b,c],[b,c]).

?-удалить(a, [b, c], [b, c]).

?-удалить(a, [a, b, c], X).
```

Списки и рекурсия

```
удалить(X, [X|T], T).
удалить(X, [H|T], [H|T1]): – удалить(X, T, T1).
```

```
?-удалить(a,[a,b,c],[b,c]).
  true.
?-удалить(a, [b, c], [b, c]).
   false.
?-удалить(a, [a, b, c], X).
   X = [b, c].
```

Списки и рекурсия

удалить(X, [X|T], T).
удалить(X, [H|T], [H|T1]): – удалить(X, T, T1).

?-удалить(X, [a, b, c], [b, c]).

?- удалить(X, [a, b, c], [a, b]).

?- удалить(X, [a, b, c], [a, c]).

Списки и рекурсия

удалить(X, [X|T], T).
удалить(X, [H|T], [H|T1]): – удалить(X, T, T1).

?-удалить(X, [a, b, c], [b, c]).
X = a.
?- удалить(X, [a, b, c], [a, b]).
X = c.
?- удалить(X, [a, b, c], [a, c]).
X = b.

Списки и рекурсия

Предикат **удалить** можно использовать и в **обратном порядке:**

?-удалить(a, X, [b, c]).

Списки и рекурсия

Предикат **удалить** можно использовать и в **обратном порядке:**

```
?-удалить(a, X, [b, c]).
  X = [a, b, c];
  X = [b, a, c];
  X = [b, c, a].
```

В данном случае **выполняется вставка** элемента **a** в произвольные позиции списка. В итоге получаются различные списки **X**, исключив из которых элемент **a**, получим список **[b, c].**

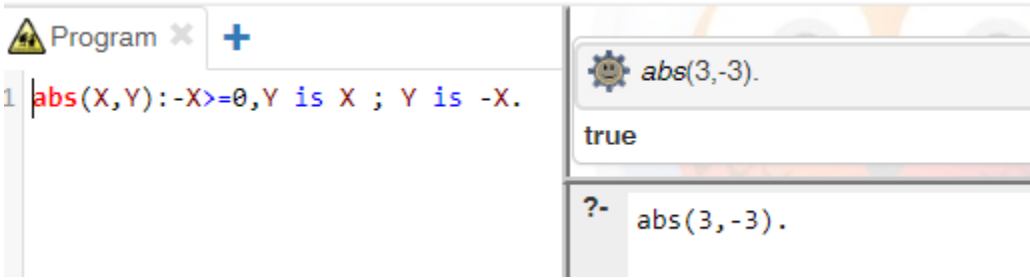
Управление возвратом (отсечение)

В Пролог имеется встроенный предикат, ограничивающий возвраты. С этой целью отсекаются некоторые ветви дерева вывода, к которым возможен возврат.

Предикат, обеспечивающий **отсечение**, обозначается знаком “**!**” .

Определим предикат **abs(X,Y)**, присваивающий переменной **Y** абсолютное значение **X**:

$$Y = \begin{cases} X, & X \geq 0 \\ -X, & иначе. \end{cases}$$



% Определение1
abs(X,Y): – X>=0, Y is X;
Y is –X.

Проверим данное определение на целевом утверждении

? – abs(3, -3).

Управление возвратом (отсечение)

Пролог-система подставит вместо **X** значение **3**, а вместо **Y** – значение **-3** и попытается выяснить выполнимость условий **3>=0, -3 is 3**.

Так как второе условие невыполнимо, то пролог-система выполнит возврат и проверит альтернативную версию **Y is -X**, то есть **-3 is -3**. В этом случае выполнение предиката **is** завершится удачей. Следовательно, в ответ на вопрос **abs(3,-3)** будет получен ошибочный ответ **true (Yes)**.

Попробуем исправить ошибку. Определим предикат **abs(X,Y)** следующим образом:

```
% определение 2
abs(X, Y): –  X>=0, Y is X;
              X<0, Y is –X.
```

Теперь при ответе на вопрос **? – abs(3, -3)** будет получен ответ **false (No)**.

В данном определении выполняются две проверки: **X>=0** и **X<0**. Что не эффективно.

Управление возвратом (отсечение)

```
% определение 3
abs(X, Y) :- X>=0, !, Y is X;
             Y is -X.
```

Управление возвратом (отсечение)

Предикат отсечения применяется также для **прерывания рекурсивных вызовов** и устранения бесконечных циклов.

В качестве примера рассмотрим программу, обеспечивающую вычисление квадратного корня $y = \sqrt{x}$. Для этого воспользуемся итерационной формулой:

$$y_n = y_{n-1} + \frac{1}{2}\left(\frac{x}{y_{n-1}} - y_{n-1}\right), \qquad y_0 = 1, \tag{6.1}$$

где y_n – значение корня на n-ом шаге вычислений,

$y_0 = 1$ – начальное приближение.

Вычисления заканчиваются, когда будет достигнута заданная точность вычислений $\varepsilon = |y_n - y_{n-1}|$.

Пусть $\varepsilon = 10^{-5}$. Определим предикат **квадратный_корень(X,Y)**:

Управление возвратом (отсечение)

квадратный_корень(X,Y): –

X>0, !, поиск_квадратного_корня(X, Корень, 1),
(Y is Корень; Y is -Корень);
X=:=0, Y is 0.

Если **X=0**, то **Y** присваивается ноль. Предикат “**!**” используется для программирования взаимоисключающих вариантов.

поиск_квадратого_корня(X, Корень, Приближение): –

Yn is (X/ Приближение+Приближение)/2,
(abs(Yn - Приближение)< 1.0e-5, !, Корень is Yn;
поиск_квадратного_корня(X, Корень, Yn)).

В рассмотренных определениях использована группа.

Группа – это заключенная в скобки последовательность вариантов, отделенных друг от друга точкой с запятой. Группа позволяет, с одной стороны, рассматривать совокупность вариантов как одно утверждение, а с другой – ограничивать область действия связки ”или”. По существу, группа – это тело правила без головы.

Управление возвратом (отсечение)

Уточним **область действия предиката отсечения**. Рассмотрим пример. Определим предикат **абсолютный_элемент(X,L)**, который будет иметь истинное значение, если список **L** будет содержать числовой элемент, абсолютное значение которого равно **X**. Приведем два определения этого предиката:

```
% Определение 1
абсолютный_элемент(X, L): –
        member(E, L), abs(E, X).
```

```
% Определение 2
абсолютный_элемент(X, L): –
    member(E, L),
    (E>=0, !, X is E;
                X is –E).
```

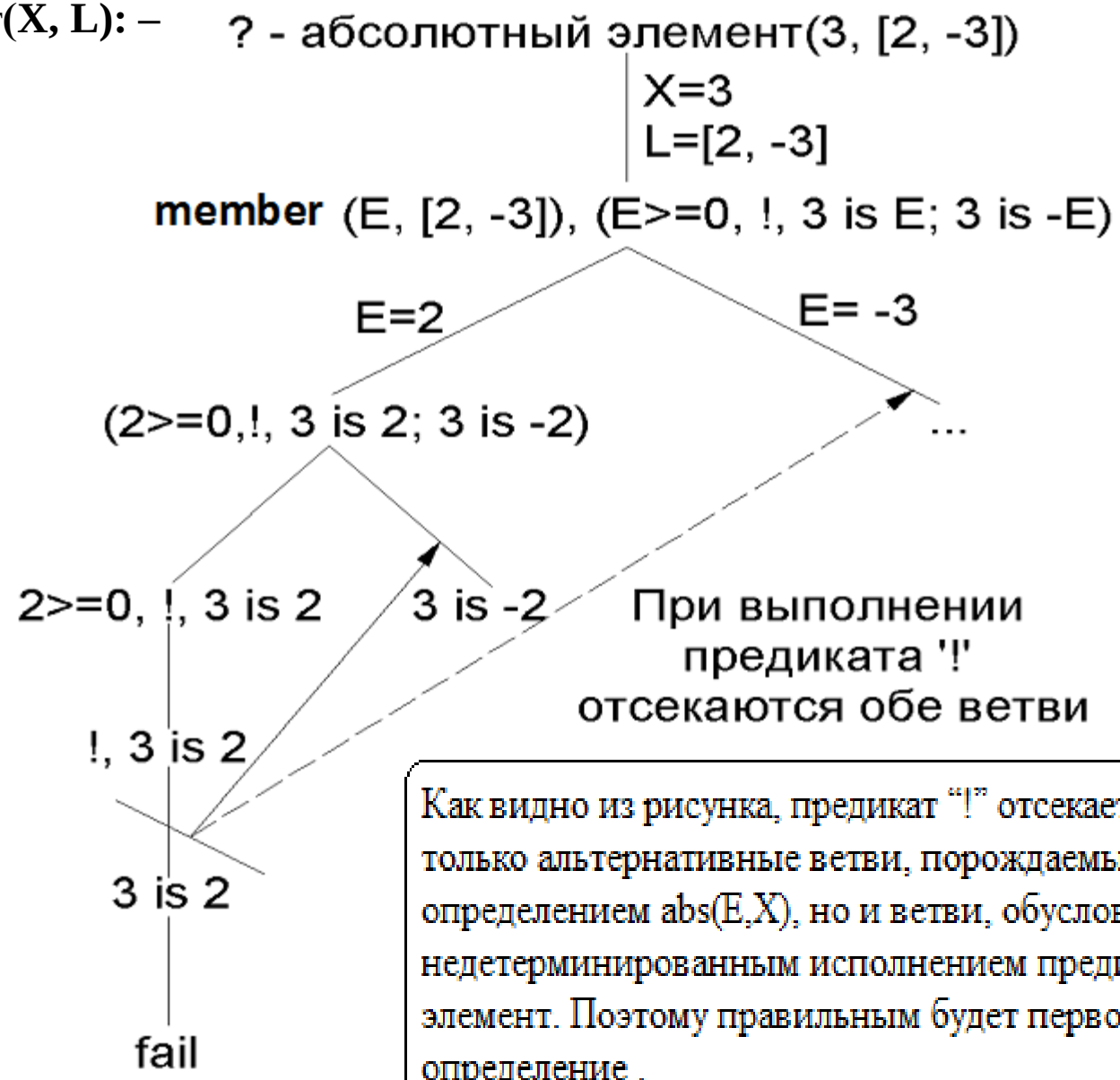
Проверим данные определения на вопросе:

? – абсолютный_элемент(3, [2, -3]).

При использовании первого определения ответ на вопрос будет положительным, а применение второго определения даст отрицательный ответ **false(No)**. Во втором случае **!** расширил область своего действия.

Управление возвратом (отсечение)

```
абсолютный_элемент(X, L): –
        member(E, L),
        (E>=0, !, X is E;
          X is –E).
```



Как видно из рисунка, предикат "!" отсекает не только альтернативные ветви, порождаемые определением abs(E,X), но и ветви, обусловленные недетерминированным исполнением предиката элемент. Поэтому правильным будет первое определение .

Отрицание в языке Пролог

Определим предикат **не_элемент(X,L)**, противоположный предикату **элемент(X, L)**:

```
не_элемент(X, L):  –
        member(X, L),!,fail;
        true.
```

В этом случае доказательство истинности исходного утверждения подменяется доказательством недоказуемости противоположного утверждения. Поэтому рассмотренной схемой построения отрицания необходимо пользоваться осторожно.

Пусть в программе имеются следующие факты и правила:

```
отец('Иван', 'Сергей').
отец('Иван', 'Ольга' ).
отец('Петр', 'Николай').
не_отец(X, Y): –
        отец(X, Y), !, fail;
        true.
```

Отрицание в языке Пролог

При попытке ответа на вопрос

?- не_отец ('Петр', 'Татьяна').

будет получен положительный ответ. Но это означает только то, что база данных не содержит утверждения **отец('Петр', 'Татьяна').**

Многие реализации Пролога содержат встроенный предикат **not**. Тогда предикат **не_отец(X, Y)** можно определить следующим образом:

не_отец(X, Y): – not(отец(X, Y)).

Обычно предикат **not** описывается в виде префиксного оператора. Поэтому цель **not(отец(X, Y))** можно также записать в виде

not отец(X, Y).

Метаусловия

Пролог допускает использование переменных не только в качестве аргументов предикатов, но и вместо условий (предикатов). Такие условия (переменные) называют ***метаусловиями (метапеременными).*** При выполнении метаусловия переменная, представляющая такое метаусловие, должна быть конкретизирована. В приведенном ниже правиле

p(A, B, C): – A, B; C.

переменные **A**, **B**, **C** представляют метаусловия. При вызове предиката **p(A, B, C)** на место аргументов должны подставляться конкретные условия.

Используя метаусловия, определим предикат **not**:

not(P): – P, !, fail;
true.

В данном определении **P** – метаусловие. Подстановка вместо **P** конкретного утверждения обеспечивает успех, если метаусловие **P** не достижимо.

Метаусловия

Введение метаусловий позволяет реализовать управляющую конструкцию **if_then_else:**

if(Если, То, Иначе): – Если, !, То; Иначе.

Здесь **Если, То, Иначе** – метаусловия. Предикат **if** удобно использовать при программировании взаимоисключающих версий. Например, переопределим предикат **abs(X, Y)**:

abs(X, Y): – if(X>=0, Y is X, Y is -X).

Здесь метаусловие **Если** задано отношением **X>=0**. Если оно справедливо, то выполняется условие **Y is X**, иначе условие **Y is -X**.

Подобным образом можно использовать предикат **if** при выборе максимального из двух чисел:

max(X, Y, Z): – if(X>=Y, Z is X, Z is Y).

max(X, Y, Z): – X>=Y,!, Z is X; Z is Y).