Programowanie w logice

PROLOG

Struktury danych - listy

- **Lista** – ciąg uporządkowanych elementów o dowolnej długości.
- Elementy listy mogą być dowolnymi termami: stałymi, zmiennymi, strukturami (w tym listami).
- Lista jest albo **listą pustą**, nie zawierającą żadnych elementów, albo jest strukturą z dwiema składowymi: **głową** i **ogonem**.

Listy

- Lista jest strukturą rekurencyjną (do jej konstrukcji użyto funktora **.** (kropka)
- Listę pustą zapisuje się: []
- Głowa i ogon listy są argumentami funktora **.** (kropka)
- Przykłady:
  - .(a,[]) lista jednoelementowa
  - .(a,.(b,[])) lista o elementach a, b
  - .(a,.(b,.(c,[]))) lista o elementach a, b, c

Przykłady list

Wygodniejszy zapis listy: elementy oddziela się przecinkami i umieszcza między nawiasami [oraz]

Zamiast: .(5,.(8,.(3,[])))

pisze się: [5,8,3]

Przykłady list:

[wydział, informatyki]

[X,lubi,Y]

[autor(adam,mickiewicz),"Pan Tadeusz"]

[[2,3],[5,6,7],[2,8]]

[a,1,[2,3],b]

Lista z głową X i ogonem Y [X|Y]

lista	głowa	ogon
[]	niezdefiniowane	niezdefiniowane
[a]	a	[]
[a,b]	a	[b]
[a,b,c]	a	[b,c]
[[1,2],[3,4],5]	[1,2]	[[3,4],5]
[1,2,3|[a,b]]	1	[2,3,a,b]
[[1,2],[a,b]]	[1,2]	[[a,b]]
[[1,2]|[a,b]]	[1,2]	[a,b]

Unifikacja list

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

true.

2 ?- .(a,.(B,.(C,[]))) = [a,b,c].

B = b,
C = c.

3 ?- [a,V,1,[c,s],p(X)] = [A,B,C,D,E].

V = B,
A = a,
C = 1,
D = [c, s],
E = p(X).

4 ?- [a,V]=[A,B].
V = B,
A = a.

5 ?- [V,a]=[A,B].
V = A,
B = a.

Unifikacja list

```
5 ?- [1,2,3,4] = [1|[2,3,4]].
true.

6 ?- [1,2,3,4] = [1,2|[3,4]].
true.

7 ?- [1,2,3,4] = [1,2,3|[4]].
true.

8 ?- [1,2,3,4] = [1,2,3,4|[]].
true.

9 ?- [1,2,3,4] = [1,2,3,4,[]].
false.

10 ?- [1,2,3,4] = [1|2,3,4].
ERROR: Syntax error: Unexpected comma or bar in rest of list
ERROR: [1,2,3,4] = [1|
ERROR: ** here **
ERROR: 2,3,4] .
```

Unifikacja list

```
10 ?- [H|T]=[1,2,3,4].
H = 1,
T = [2, 3, 4].

11 ?- [A|B]=[[1,2],3,4].
A = [1, 2],
B = [3, 4].

12 ?- [H|T]=[A,b].
H = A,
T = [b].

13 ?- [H|T]=[b].
H = b,
T = [].

14 ?- [A,B|C]=[a|[1,2,3]].
A = a,
B = 1,
C = [2, 3].
```

Przetwarzanie list

- Listy są **strukturami rekurencyjnymi**, do ich przetwarzania służą procedury rekurencyjne.
- **Procedura** – zbiór klauzul zbudowany w oparciu o ten sam predykat.
- **Procedura rekurencyjna** składa się z klauzul:

1. Faktu opisującego sytuację, która powoduje zakończenie rekurencji, np. napotkanie listy pustej,
2. Reguły, która przedstawia sposób przetwarzania listy.
   W jej ciele znajduje się ten sam predykat,
   co w nagłówku, tylko z innymi argumentami.

Przykład procedury rekurencyjnej

Wypisanie na ekranie elementów listy:

```
pisz([]).
pisz([X|Y]):-write(X),nl,pisz(Y).
```

Fakt mówi, że w przypadku napotkania listy pustej (końca listy) nie należy nic robić.
Reguła mówi: podziel listę na głowę i ogon, wydrukuj głowę listy, następnie ją pomiń i zastosuj tę samą metodę do powstałego ogona.

write ozn. wypisanie termu
nl ozn. przejście do nowej linii

Przykłady predykatów wbudowanych działających na listach

- **is_list** (L) - sprawdza, czy L jest listą

Przykład.

```
?- is_list([1,2,a,b]).
   true.
```

- **append** (L1,L2,L3) – łączy listy L1 i L2 w listę L3

Przykład.

```
?- append([1,2],[3,4],X).
   X=[1,2,3,4].
```

```
15 ?- append(A,B,[1,2,3]).
A = [],
B = [1, 2, 3] ;
A = [1],
B = [2, 3] ;
A = [1, 2],
B = [3] ;
A = [1, 2, 3],
B = [] ;
false.
```

- **member**(E,L) – sprawdza, czy element E należy do listy L lub wypisuje elementy listy L

Przykład.

```
?- member(5,[3,6,5,7,6]).
   true
?- member(X,[2,3,4,9]).
   X = 2 ;
   X = 3 ;
   X = 4 ;
   X = 9 ;
   false.
```

- **memberchk**(E,L) - równoważny predykatowi member, ale podaje tylko jedno rozwiązanie (pierwsze)

• **nextto**(X,Y,L) – predykat spełniony, gdy Y występuje bezpośrednio po X

Przykład.

```
?- nextto(X,Y,[2,3,4,5]).
  X = 2,
  Y = 3 ;
  X = 3,
  Y = 4 ;
  X = 4,
  Y = 5 .
?- nextto(3,Y,[2,3,4,5]).
  Y = 4
?- nextto(X,4,[2,3,4,5]).
  X = 3
```

• **delete**(L1,E,L2) – z listy L1 usuwa **wszystkie** wystąpienia elementu E, wynik uzgadnia z listą L2

Przykład.

```
?-delete([1,2,3,2,5,3],3,X).
  X = [1, 2, 2, 5].
```

• **select**(E,L,R) – lista R jest uzgadniana z listą, która powstaje z L po usunięciu wybranego (jednego) elementu.

Przykład.

```
?-select(3,[1,2,3,2,5,3],X).
  X = [1, 2, 2, 5, 3] ;
  X = [1, 2, 3, 2, 5]
```

• **nth1**(N,L,E) – predykat spełniony, jeśli element listy L o numerze N daje się uzgodnić z elementem E

Przykład.

```
?-nth1(2,[a,b,c,d],Y).
  Y = b.
?-nth1(X,[a,d,b,c,d],d).
  X = 2 ;
  X = 5.
```

• **last**(L,E) – ostatni element listy L

Przykład.

```
?-last([a,b,c,d],Y).
  Y = d.
```

• **reverse**(L1,L2) – odwraca porządek elementów listy L1 i unifikuje rezultat z listą L2

Przykład.

```
?-reverse([a,b,c,d],Y).
  Y = [d,c,b,a].
```

• **permutation**(L1,L2) – lista L1 jest permutacją listy L2

Przykład.

```
?- permutation([1,2,3],L).
  L = [1, 2, 3] ;
  L = [2, 1, 3] ;
  L = [2, 3, 1] ;
  L = [1, 3, 2] ;
  L = [3, 1, 2] ;
  L = [3, 2, 1] ;
```

• **sumlist**(L,S) – suma listy liczbowej L

Przykład.

```
?-sumlist([1,4,7,9],S).
  S=21.
```

• **length**(L,N) – liczba elementów listy L

Przykład.

```
?-length([b,2,a,0],N).
  N=4.
```

Operacje na listach

Sprawdzenie, **czy element jest na liście**

Procedura: X jest elementem listy L, jeżeli X jest głową listy L lub X jest elementem ogona listy L.

```
element(X,[X|_]).
element(X,[_|Ogon]) :- element(X,Ogon).
```

„_" to zmienna anonimowa zastępująca głowę listy [_|Ogon], jej nazwa nie ma znaczenia

Przykład.

```
?-element(a,[w,s,d,a,e]).
  true
```

Predykat wbudowany: **member**

Operacje na listach

Łączenie list

Procedura:

- Jeżeli pierwszy element listy jest pusty [], to drugi i trzeci element muszą być takie same (L = L).
- Jeżeli pierwszy element nie jest pusty, to głową listy L3 staje się głową listy L1, a ogonem listy L3 jest ogon listy L1 złączony z listą L2.

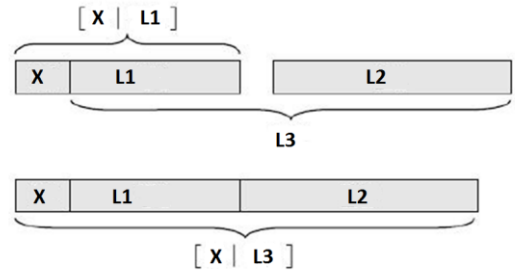
```
polacz([],L,L).
polacz([X|L1],L2,[X|L3]):-polacz(L1,L2,L3).
```

Predykat wbudowany: **append**

Przykład.

?-polacz([1,2,3],[2,3,4],L).

L=[1,2,3,2,3,4].



Operacje na listach

Liczba elementów listy liczbowej

Procedura:

- długość listy pustej jest równa 0 (fakt)
- Długość listy, to długość jej ogona plus jeden (reguła)

```
dlugosc([],0).
dlugosc([G|O],N):-dlugosc(O,N1),N is N1+1.
```

Przykład.

?-dlugosc([a,s,d,f,g],K).

K=5.

Predykat wbudowany: **length**

Operacje na listach

Odwracanie kolejności elementów listy

Procedura:

- odwrotna do listy pustej jest lista pusta (fakt)
- odwrotnością listy jest połączenie odwróconego ogona listy z listą złożoną z głowy listy wejściowej (reguła)

```
odwracanie([],[]).
odwracanie([A|B],C):- odwracanie(B,D),
                      append(D,[A],C).
```

Predykat wbudowany: **reverse**

Operacje na listach

n-ty element listy, element na n-tym miejscu w liście

Procedura:

- głowa listy jest pierwszym elementem listy (fakt)
- n-ty element listy jest n-1-szym elementem ogona (reguła)

```
nty(1,[E|_],E).
nty(N,[_|X],E):-nty(N1,X,E), N is N1+1.
```

```
3 ?- nth1(3,[a,b,c,d,c],X).
X = c.

4 ?- nth1(Y,[a,b,c,d,c,e],c).
Y = 3 ;
Y = 5 ;
false.
```

```
1 ?- nth1d(3,[a,b,c,d,c],X).
X = c ;
false.

2 ?- nth1d(Y,[a,b,c,d,c],c).
Y = 3 ;
Y = 5 ;
false.
```

Predykat wbudowany: **nth1**

Operacje na listach

Porównaj:

```
nth1a(1,[E|_],E).
nth1a(N,[_|T],E):-N is N-1,nth1a(N,T,E).

nth1b(1,[E|_],E).
nth1b(N,[_|T],E):-N1 is N-1,nth1b(N1,T,E).

nth1c(1,[E|_],E).
nth1c(N,[_|T],E):-N is N1+1,nth1c(N1,T,E).

nth1d(1,[E|_],E).
nth1d(N,[_|T],E):-nth1d(N1,T,E),N is N1+1.

nth1e(1,[E|_],E).
nth1e(N,[_|T],E):-nth1e(N1,T,E),N1 is N-1.
```

```
nth1a(1,[E|_],E).
nth1a(N,[_|T],E):-N is N-1,nth1a(N,T,E).

1 ?- nth1a(3,[a,b,c,d,c,e],X).
false.

2 ?- nth1a(Y,[a,b,c,d,c,e],c).
ERROR: is/2: Arguments are not sufficiently instantiated
```

```
nth1b(1,[E|_],E).
nth1b(N,[_|T],E):-N1 is N-1,nth1b(N1,T,E).

3 ?- nth1b(3,[a,b,c,d,c,e],X).
X = c ;
false.

4 ?- nth1a(Y,[a,b,c,d,c,e],c).
ERROR: is/2: Arguments are not sufficiently instantiated
```

Predykat odcięcia „cut” („!”)

„Cut” – bezargumentowy predykat jest interpretowany logicznie jako zawsze prawdziwy i służy do ograniczania nawrotów.

Realizacja tego predykatu, występującego jako jeden z podcelów w ciele klauzuli, uniemożliwia nawrót do któregokolwiek z poprzedzających go podcelów przy próbie znajdowania rozwiązań alternatywnych.

Cut

Wszystkie zmienne, które zostały ukonkretnione podczas realizacji poprzedzających odcięcie podcelów w ciele klauzuli, zachowują nadane im wartości w trakcie realizacji występujących po predykacie odcięcia warunków.

Odcięcie nie ma wpływu na nieukonkretnione zmienne występujące w następujących po nim podcelach.

Wpływ na nawracanie

repeat – generowanie wielu rozwiązań danego problemu poprzez „wymuszanie” nawrotów

Przykład.

```
a(1).
a(2).
a(3).
a(4).
?-repeat, a(X),write(X), X==3,!.
123
X=3
```

Przykłady na wykładzie (odcięcie.pl, stolica.pl)

Przykład wykorzystania odcięć

```
max1(X,Y,X) :- X >= Y.
max1(X,Y,Y) :- X < Y.

max2(X,Y,X) :- X >= Y, !.
max2(_,Y,Y).
```

```
[trace] 54 ?- max1(2,3,X).
   Call: (6) max1(2, 3, _G9371) ? creep
   Call: (7) 2>=3 ? creep
   Fail: (7) 2>=3 ? creep
   Redo: (6) max1(2, 3, _G9371) ? creep
   Call: (7) 2<3 ? creep
   Exit: (7) 2<3 ? creep
   Exit: (6) max1(2, 3, 3) ? creep
X = 3.

[trace] 55 ?- max2(2,3,X).
   Call: (6) max2(2, 3, _G9371) ? creep
   Call: (7) 2>=3 ? creep
   Fail: (7) 2>=3 ? creep
   Redo: (6) max2(2, 3, _G9371) ? creep
   Exit: (6) max2(2, 3, 3) ? creep
X = 3.
```

Predykat „fail”

„fail” powoduje niepowodzenie wykonywania klauzuli. Wykonanie tego predykatu zawsze zawodzi. Najczęściej używany w celu wymuszenia nawrotów.

Użyty w kombinacji z „cut” (!,fail) zapobiega użyciu innej klauzuli przy próbie znalezienia rozwiązań alternatywnych, co oznacza niepowodzenie wykonywania całej procedury.

Przykłady na wykładzie (fail.pl)

Literatura

- W. Clocksin, C. Mellish, „Prolog. Programowanie”
- E.Gatnar, K.Stąpor, „Prolog”
- G.Brzykcy, A.Meissner, „Programowanie w prologu i programowanie funkcyjne”
- M. Ben-Ari, “Logika matematyczna w informatyce”
- http://lpn.swi-prolog.org/lpnpage.php?pageid=online