Programowanie w logice

PROLOG

Rekurencja

```
silnia(0,1).
silnia(N,X):-
    N1 is N-1,
    silnia(N1,X1),
    X is N*X1.
```

```
[trace] 2 ?- silnia(2,L).
   Call: (6) silnia(2, _G3191) ? creep
   Call: (7) _G3266 is 2+ -1 ? creep
   Exit: (7) 1 is 2+ -1 ? creep
   Call: (7) silnia(1, _G3267) ? creep
   Call: (8) _G3269 is 1+ -1 ? creep
   Exit: (8) 0 is 1+ -1 ? creep
   Call: (8) silnia(0, _G3270) ? creep
   Exit: (8) silnia(0, 1) ? creep
   Call: (8) _G3272 is 1*1 ? creep
   Exit: (8) 1 is 1*1 ? creep
   Exit: (7) silnia(1, 1) ? creep
   Call: (7) _G3191 is 2*1 ? creep
   Exit: (7) 2 is 2*1 ? creep
   Exit: (6) silnia(2, 2) ? creep
L = 2 .
```

```
silnia2(N,X):-
    silnia2(N,1,X).
silnia2(N,A,X):-
    N>0,
    A1 is A*N,
    N1 is N-1,
    silnia2(N1,A1,X).
silnia2(0,X,X).
```

```
[trace] 1 ?- silnia2(2,L).
   Call: (6) silnia2(2, _G3019) ? creep
   Call: (7) silnia2(2, 1, _G3019) ? creep
   Call: (8) 2>0 ? creep
   Exit: (8) 2>0 ? creep
   Call: (8) _G3094 is 1*2 ? creep
   Exit: (8) 2 is 1*2 ? creep
   Call: (8) _G3097 is 2+ -1 ? creep
   Exit: (8) 1 is 2+ -1 ? creep
   Call: (8) silnia2(1, 2, _G3019) ? creep
   Call: (9) 1>0 ? creep
   Exit: (9) 1>0 ? creep
   Call: (9) _G3100 is 2*1 ? creep
   Exit: (9) 2 is 2*1 ? creep
   Call: (9) _G3103 is 1+ -1 ? creep
   Exit: (9) 0 is 1+ -1 ? creep
   Call: (9) silnia2(0, 2, _G3019) ? creep
   Call: (10) 0>0 ? creep
   Fail: (10) 0>0 ? creep
   Redo: (9) silnia2(0, 2, _G3019) ? creep
   Exit: (9) silnia2(0, 2, 2) ? creep
   Exit: (8) silnia2(1, 2, 2) ? creep
   Exit: (7) silnia2(2, 1, 2) ? creep
   Exit: (6) silnia2(2, 2) ? creep
L = 2.
```

```
silnia3(N,X):-
    silnia3(0,N,1,X).
silnia3(C,N,A,X):-
    C<N,
    C1 is C+1,
    A1 is A*C1,
    silnia3(C1,N,A1,X).
silnia3(N,N,X,X).
```

```
[trace] 3 ?- silnia3(2,L).
   Call: (6) silnia3(2, _G3564) ? creep
   Call: (7) silnia3(0, 2, 1, _G3564) ? creep
   Call: (8) 0<2 ? creep
   Exit: (8) 0<2 ? creep
   Call: (8) _G3639 is 0+1 ? creep
   Exit: (8) 1 is 0+1 ? creep
   Call: (8) _G3642 is 1*1 ? creep
   Exit: (8) 1 is 1*1 ? creep
   Call: (8) silnia3(1, 2, 1, _G3564) ? creep
   Call: (9) 1<2 ? creep
   Exit: (9) 1<2 ? creep
   Call: (9) _G3645 is 1+1 ? creep
   Exit: (9) 2 is 1+1 ? creep
   Call: (9) _G3648 is 1*2 ? creep
   Exit: (9) 2 is 1*2 ? creep
   Call: (9) silnia3(2, 2, 2, _G3564) ? creep
   Call: (10) 2<2 ? creep
   Fail: (10) 2<2 ? creep
   Redo: (9) silnia3(2, 2, 2, _G3564) ? creep
   Exit: (9) silnia3(2, 2, 2, 2) ? creep
   Exit: (8) silnia3(1, 2, 1, 2) ? creep
   Exit: (7) silnia3(0, 2, 1, 2) ? creep
   Exit: (6) silnia3(2, 2) ? creep
L = 2 .
```

Przechwytywanie wyników

bagof(X,P,L)
buduje listę L, złożoną z takich X, że spełnione jest P.

Podobnie: setof(X,P,L)
powstała lista jest posortowana i nie zawiera ew. duplikatów.

```
1 ?- setof(3,member(3,[2,3,4,4,3]),L).
L = [3].

2 ?- bagof(3,member(3,[2,3,4,4,3]),L).
L = [3, 3].
```

Rekurencyjne definiowanie liczb

```
liczba_naturalna(0).
liczba_naturalna(s(X)):-
    liczba_naturalna(X).
```

Rekurencyjne struktury danych - drzewa

Struktury drzewiaste definiuje się za pomocą stałej reprezentującej drzewo puste i funktora, którego argumentami są korzeń i wychodzące z niego poddrzewa.

Drzewa binarne – posiadają co najwyżej dwa poddrzewa.

Do zapisywania drzew binarnych stosujemy funktory 3-argumentowe:

tree(Element, Left, Right)

nil – drzewo puste
Left – poddrzewo lewe
Right – poddrzewo prawe

Rekurencyjne struktury danych - drzewa

Definicja drzewa binarnego:

```
binary_tree(nil).
binary_tree(tree(E,L,R)):-
        binary_tree(L),
        binary_tree(R).
```

Rekurencyjne struktury danych - drzewa

Przeglądanie „w głąb”

```
preorder(nil,[]).
preorder(tree(E,L,R),N):-
        preorder(L,L1),
        preorder(R,R1),
        append([E|L1],R1,N).
```

Rekurencyjne struktury danych - drzewa

Przeglądanie: najpierw wierzchołki lewego poddrzewa, korzeń i wierzchołki prawego poddrzewa

```
inorder(nil,[]).
inorder(tree(E,L,R),N):-
        inorder(L,L1),
        inorder(R,R1),
        append(L1,[E|R1],N).
```

Rekurencyjne struktury danych - drzewa

Przeglądanie: porządek wsteczny – lewe poddrzewo, prawe poddrzewo i korzeń

```
postorder(nil,[]).
postorder(tree(E,L,R),N):-
        postorder(L,L1),
        postorder(R,R1),
        append(L1,R1,N1),
        append(N1,[E],N).
```

Zastosowania matematyczne Prologu

Przynależność do zbioru (member)

```
nalezy_do_listy(X,[X|_]).
nalezy_do_listy(X,[_|Y]):-
        nalezy_do_listy(X,Y).
```

Zawieranie się zbiorów (subset)

```
podzbior([],Y).
podzbior([A|X],Y):-nalezy_do_listy(A,Y),
        podzbior(X,Y).
```

Zastosowania matematyczne Prologu

Część wspólna zbiorów **(intersect)**

```
czesc_wspolna([],X,[]).
czesc_wspolna([X|R],Y,[X|Z]):-
                nalezy_do_listy(X,Y),!,
                czesc_wspolna(R,Y,Z).
czesc_wspolna([X|R],Y,Z):-
                czesc_wspolna(R,Y,Z).
```

Zastosowania matematyczne Prologu

Suma zbiorów **(union)**

```
suma_zb([],X,X).
suma_zb([X|R],Y,Z):-nalezy_do_listy(X,Y),!,
                suma_zb(R,Y,Z).
suma_zb([X|R],Y,[X|Z]):-suma_zb(R,Y,Z).
```

Zastosowania matematyczne Prologu

Różnica zbiorów **(difference)**

```
roznica([],_,[]).
roznica([X|L],Set,[X|Z]):-
            not(member(X,Set)),!,
            roznica(L,Set,Z).
roznica([_|L],Set,Z):-roznica(L,Set,Z).
```

Różniczkowanie symboliczne

```
pochodna(X,X,1):-!.
pochodna(C,X,0):-atomic(C).
pochodna(-Z,X,-C):-pochodna(Z,X,C).
pochodna(W+Z,X,A+B):-
        pochodna(W,X,A),pochodna(Z,X,B).
pochodna(W-Z,X,A-B):-
        pochodna(W,X,A),pochodna(Z,X,B).
pochodna(C*Z,X,C*A):-
        atomic(C),C\=X,pochodna(Z,X,A),!.
pochodna(W*Z,X,B*W+A*Z):-
        pochodna(W,X,A),pochodna(Z,X,B).
```

Rozwiązywanie równań

```
plus(X,Y,Z):-Z is X+Y.
minus(X,Y,Z):-Z is X-Y.
                X+3=8
                X-Y=3
liczba(0).
liczba(I):-liczba(L), I is L+1.

rownanie(X,Y):-liczba(X),
            plus(X,3,8),liczba(Y),
            minus(X,Y,3),!.
```

Dodawanie macierzy

```
dodm([L1|O1],[L2|O2],[L3|O3]):-
            dodl(L1,L2,L3),
            dodm(O1,O2,O3).
dodm([],[],[]).
dodl([L1|O1],[L2|O2],[L3|O3]):-
            L3 is L1+L2,
            dodl(O1,O2,O3).
dodl([],[],[]).
```

Postacie normalne formuł KRZ

Conjunctive			Disjunctive		
α	α_1	α_2	β	β_1	β_2
$X \wedge Y$	X	Y	$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$\neg(X \vee Y)$	$\neg X$	$\neg Y$	$X \vee Y$	X	Y
$\neg(X \supset Y)$	X	$\neg Y$	$X \supset Y$	$\neg X$	Y
$\neg(X \subset Y)$	$\neg X$	Y	$X \subset Y$	X	$\neg Y$
$\neg(X \uparrow Y)$	X	Y	$X \uparrow Y$	$\neg X$	$\neg Y$
$X \downarrow Y$	$\neg X$	$\neg Y$	$\neg(X \downarrow Y)$	X	Y
$X \not\supset Y$	X	$\neg Y$	$\neg(X \not\supset Y)$	$\neg X$	Y
$X \not\subset Y$	$\neg X$	Y	$\neg(X \not\subset Y)$	X	$\neg Y$

and, or, imp, revimp, uparrow, downarrow, notimp, notrevimp

Przykład - wykład

REGUŁY REDUKCJI

do postaci klauzulowej (KPN):

$$\left| \begin{array}{ccccc} \frac{\neg\neg Z}{Z} & \frac{\neg\top}{\bot} & \frac{\neg\bot}{\top} & \begin{array}{c}\underline{\beta}\\ \beta_1 \\ \beta_2\end{array} & \frac{\alpha}{\alpha_1 \mid \alpha_2} \end{array} \right|$$

dualnej klauzulowej (DPN):

$$\left| \begin{array}{ccccc} \frac{\neg\neg Z}{Z} & \frac{\neg\top}{\bot} & \frac{\neg\bot}{\top} & \begin{array}{c}\underline{\alpha}\\ \alpha_1 \\ \alpha_2\end{array} & \frac{\beta}{\beta_1 \mid \beta_2} \end{array} \right|$$

Aby sprowadzić formułę A do postaci klauzulowej (KPN):

begin
  Niech S oznacza $\langle[A]\rangle$;
  while *jakiś element S zawiera nie-literał* **do**
    wybierz z S element D zawierający nie-literał;
    wybierz z D nie-literał N;
    zastosuj odpowiednią regułę redukcyjną do N;
    niech S oznacza nowo utworzoną formułę
  end
end

Aby sprowadzić formułę A do postaci dualnej klauzulowej (DPN):

begin
  Niech S oznacza $[\langle A\rangle]$;
  while *jakiś element S zawiera nie-literał* **do**
    wybierz z S element C zawierający nie-literał;
    wybierz z C nie-literał N;
    zastosuj odpowiednią regułę redukcyjną do N;
    niech S oznacza nowo utworzoną formułę
  end
end

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”