Programowanie w Prologu, cz.3

Struktury otwarte i listy różnicowe

Listy otwarte

reprezentacja:

– lista pusta: `Zmienna`
– lista niepusta: `[ E1, ..., En | Zmienna ]`

Przykłady

1. Dodanie elementu do listy.

```
dodaj( Elem, Lista ) :-  var( Lista ), !,
                         Lista = [ Elem | _ ].
dodaj( Elem, [ _ | Lista ] ) :- dodaj( Elem, Lista ).
```

2. Scalenie list (L1 = konkatenacja L1 i L2).

```
extend( L1, L2 ) :-  var( L1 ), !, L1 = L2.
extend( [ _ | L1 ], L2 ) :-   extend( L1, L2 ).
```

3. Zamknięcie listy otwartej.

```
close( Lista ) :-  var( Lista ), !, Lista = [].
close( [ _ | Lista ] ) :-  close( Lista ).
```

Drzewa otwarte

reprezentacja:

– drzewo puste: `Zmienna`
– drzewo niepuste: `tree( Lewe, Wartość, Prawe )`

Przykład — dodanie elementu do drzewa BST.

```
dodaj( E, Drzewo ) :-  var( Drzewo ), !,
                       Drzewo = tree( _, E, _ ).
%dodaj( E, tree( _, E, _ ) ) :-  !.
dodaj( E, tree(L, K, P) ) :- E < K, !, dodaj( E, L ).
dodaj( E, tree(L, K ,P) ) :- E > K, !, dodaj( E, P ).
```

Listy różnicowe

lista różnicowa = lista otwarta + zmienna (koniec listy)

reprezentacja:

- lista pusta: `Zmienna -- Zmienna`
  (lub `lr( Zmienna, Zmienna)`)
- lista niepusta: `[ E1, ..., En | Zmienna ] -- Zmienna`

Przykłady

1. Dodanie elementu do listy różnicowej.

```
dodaj( Elem, Lista -- Koniec, NLista -- NKoniec) :-
       NLista = Lista,
       Koniec = [ Elem | NKoniec].
```

czyli:

```
dodaj( E, Lista -- [ E | Koniec], Lista -- Koniec ).
```

2. Konkatenacja list różnicowych.

```
extend( L -- E1, E1 -- E2, L -- E2 ).
```

3. Zamknięcie listy różnicowej.

```
close( Lista -- [], Lista ).
```

4. Lista wierzchołków drzewa (wszerz).

```
wszerz( Drzewo, Lista ) :-
       wszerzp( [ Drzewo | R ] -- R, Lista ).

wszerzp( X -- X, L ) :-  var( X ),  !,  L = [].
wszerzp([ nil | R ]--K,  L) :- wszerzp( R -- K, L ).
wszerzp([tree(L,K,P) | R]--[L,P | NK], [K|Lista]) :-
        wszerzp( R -- NK, Lista ).
```

Uwaga: klauzula nr 1 `wszerzp` — niezbedne przeniesienie `[]` do treści (wpp. sukces np. dla `wszerz(drzewo(nil,1,nil),[1,2])`).

Implementacja kolejki (FIFO) przy użyciu list różnicowych

```
% operacje na kolejce:
%    init( Kolejka )
%    put( Element, Kolejka, NowaKolejka)
%    get( Element, Kolejka, NowaKolejka)
%    empty( Kolejka )

init( Kolejka -- Kolejka ).            % pusta
put( Elem, Kolej -- [Elem | Koniec], Kolej -- Koniec ).
get( Elem, [Elem | Kolej] -- Koniec, Kolej -- Koniec ).
empty( Kolejka -- Koniec ) :-  var( Kolejka ).
```

(definicja procedur **get**, **put** — jedna klauzula unarna)

- Przykład użycia, wstawianie do kolejki.

```
:-  init(Kol), put(3, Kol, Kol1), put(5, Kol1, Kol2).

                     % po pierwszej operacji put
 Kol1 = [ 3 | Koniec ] -- Koniec
 Kol  = [ 3 | Koniec ] -- [ 3 | Koniec ]    % pusta

                     % po drugiej operacji put
 Kol2 = [ 3, 5 | K ] -- K
 Kol1 = [ 3, 5 | K ] -- [ 5 | K ]         % niepusta
 Kol  = [ 3, 5 | K ] -- [ 3, 5 | K ]      % pusta
```

```
init( Kolejka -- Kolejka ).            % pusta
put( Elem, Kolej -- [Elem | Koniec], Kolej -- Koniec ).
get( Elem, [Elem | Kolej] -- Koniec, Kolej -- Koniec ).
```

- Pobieranie elementów z pustej (!) kolejki.

```
:-  init(Kol), get(E1, Kol, Kol1), put(3, Kol1, Kol2).


 % po operacji get

 Kol1 = Koniec -- [ E1 | Koniec ]     % lista ujemna
 Kol  = [ E1 | Koniec ] -- [ E1 | Koniec ]   % pusta

 % po operacji put

 E1 = 3
 Kol2 =  Koniec -- Koniec
 Kol1 =  Koniec -- [ 3 | Koniec ]
 Kol  =  [ 3 | Koniec ] -- [ 3 | Koniec ]
```

Struktura termów, działania na termach

Predykaty standardowe:

- functor(Term, Funktor, Arność)
- arg(Numer, Term, Argument)
- Term =.. [Nazwa | Argumenty] (wym. univ)

Np. `:- functor( tree( L, W, P ), tree, 3 ).`

```
:- arg( 2, f(a, b), b ).
:- tree( nil, 5, nil ) =..  [ tree, nil, 5, nil ].
```

podterm(T, Term) wiw gdy T jest podtermem ustalonego termu Term

— rozwiązanie z użyciem `functor` oraz `arg`

```
podterm( T, T ).
podterm( P, T ) :-  functor( T, _, N ),
                    pterm( N, P, T ).

pterm( N, P, T ) :-  N > 0,
                     arg( N, T, TN ),
                     podterm( P, TN ).
pterm( N, P, T ) :-  N > 1,
                     N1 is N - 1,
                     pterm( N1, P, T ).
```

— rozwiązanie z użyciem univ (`=..`)

```
podterm( T, T ).
podterm( P, T ) :-  T =.. [ Nazwa | Args ],
                    pterm( P, Args ).

pterm( P, [ Arg | Reszta ] ) :-  podterm( P, Arg ).
pterm( P, [ _ | Reszta ] ) :-  pterm( P, Reszta ).
```

Dostęp do predykatów

predykat standardowy `clause( Nagłówek, Treść )`

Np.
```
clause( podterm( T, T ), true )
clause( podterm( P, T ),
        (functor( T, _, N ), pterm( N, P, T )) )
```

wydruk(Nazwa/Arność) — wydruk treści predykatu

```
wydruk( Nazwa/Arno"s"c ) :-
        functor( Nag"l"owek, Nazwa, Arno"s"c ),
        clause( Nag"l"owek, Tre"s"c ),
        write( (Nag"l"owek :- Tre"s"c) ),
        put( ~. ),  nl,
        fail.
wydruk( _ ).
```

wywołanie zmiennej — `call( X )` lub `X`

once(P) — P odnosi sukces co najwyżej jeden raz

```
once( P ) :-  call( P ),  !.
```

test(P) — czy P odnosi sukces?

```
test( P ) :-  not not call( P ).
```

alternatywa — ;

```
p( X, Y ) :-  X > 100,  q( Y ).
p( X, Y ) :-  X < 0,  q( Y ).

p( X, Y ) :-  ( X > 100 ; X < 0 ),  q( Y ).
```

Dynamiczna modyfikacja programu

Predykaty standardowe:

- `assert( Klauzula )` — dodanie klauzuli
- `asserta( Klauzula )` — na początek definicji
- `assertz( Klauzula )` — na koniec definicji
- `retract( Klauzula )` — usunięcie klauzuli (jeśli istnieje klauzula, którą można uzgodnić z argumentem — punkt nawrotu)

abolish(Nazwa, Arność) — usunięcie całej definicji predykatu

```
abolish( Nazwa, Arno"s"c ) :-
        functor( Term, Nazwa, Arno"s"c ),
        retract( (Term :- Tre"s"c) ),
        fail.
abolish( _, _ ).      % zawsze sukces
```

poprawne zastosowania:

— dodawanie lematów, usuwanie redundantnych formuł
— globalne flagi

Przykład — wieże Hanoi

```
% hanoi( N, Sk"ad, Dok"ad, Pom, SekwencjaRuch"ow )

hanoi( 1, A, B, C, [ A na B ] ).
hanoi( N, A, B, C, Sekwencja ) :-
      N > 1, N1 is N - 1,
      lemma( hanoi( N1, A, C, B, Op1 ) ),
      hanoi( N1, C, B, A, Op2 ),
      append( Op1, [ A na B | Op2 ], Sekwencja ).

lemma( P ) :-  P,  asserta( (P :- !) ).
```

```
test_hanoi( N, Nazwy, Wynik ) :-
           hanoi( N, X, Y, Z, Wynik ),
           Nazwy = [ X, Y, Z ].

:- test_hanoi( 10, [ a, b, c ], JakPrze"lo"ry"c ).
```

Generowanie nazw zmiennych/stałych

— przekazywanie wartości przez parametry
— globalne flagi

```
gensym( Prefix, Symbol ) :-
       var( Symbol ),  atom( Prefix ),
       old_value( Prefix, N ),
       N1 is N + 1,
       set_flag( gensym( Prefix, N1 ) ),
       concatenate( Prefix, N1, Symbol ),
       !.

old_value( Prefix, N ) :-
          flaga( gensym( Prefix ), N ),  !.
old_value( Prefix, 0 ).  % pocz"atkowa warto"s"c

set_flag( Nazwa, W ) :-
          nonvar( Nazwa ),
          retract( flaga( Nazwa, _ ) ),  !,
          asserta( flaga( Nazwa, W ) ).
set_flag( Nazwa, W ) :-
          nonvar( Nazwa ),
          asserta( flaga( Nazwa, W ) ).
```

Wszystkie rozwiązania

- `setof( Wzorzec, Pytanie, Zbiór )`
- `bagof( Wzorzec, Pytanie, Worek )`

`Zbiór`/`Worek` — lista wszystkich odpowiedzi (ukonkretnień termu `Wzorzec`) powstałych na podstawie dowodu `Pytania`

Przykład.

```
   lubi( jan, piwo ).     lubi( jan, cola ).
   lubi( piotr, wino ).
   lubi( ewa, pepsi ).    lubi( ewa, cola ).

?- setof( CoLubi, lubi( ewa, CoLubi ), CoLubiEwa ).
CoLubiEwa = [ pepsi, cola ]

?- setof( Nap"oj, lubi( Kto"s, Nap"oj ), Lubiane ).
Lubiane = [ piwo, cola, wino, pepsi ]

?- setof( (Osoba, Lubi),
          setof( Nap"oj, lubi( Osoba, Nap"oj ), Lubi ),
          Wynik ).
Wynik = [ (jan, [piwo, cola]), (piotr, [wino]),
          (ewa, [pepsi, cola]) ]

?- setof( (Nap"oj, KtoLubi),
          setof( Kto, lubi( Kto, Nap"oj ), KtoLubi ),
          Wynik ).
Wynik = [ (piwo, [jan]), (cola, [jan, ewa]),
          (wino, [piotr]), (pepsi, [ewa]) ]
```