Programowanie w Prologu, cz.2

Reprezentacja struktur danych

- termy (stałe, funktory)
- klauzule

Drzewa binarne — reprezentacja, podstawowe operacje

Reprezentacja: — `nil` – puste drzewo
— `tree( Lewe, Wartość, Prawe)`

3.1. drzewo(D) wiw gdy D jest drzewem binarnym

```
drzewo( nil ).
drzewo( tree( L, _, P ) ) :- drzewo( L ), drzewo( P ).
```

3.2. insert(D, E, N) wiw gdy N jest drzewem powstałym przez dodanie elementu E do drzewa D (drzewa BST)

```
insert( nil, Elem, tree( nil, Elem, nil ) ).
insert( tree( L, W, P ), E, tree( NL, W, P ) ) :-
       E <= W,
       insert( L, E, NL ).
insert( tree( L, W, P ), E, tree( L, W, NP ) ) :-
       E > W,
       insert( P, E, NP ).
```

3.3. wypisz_BST(D) — wypisanie wszystkich elementów (niemalejąco)

```
wypisz_BST( tree( L, W, P ) ) :-
           wypisz_BST( L ),
           write( W ), write( ', ' ),
           wypisz_BST( P ).
wypisz_BST( nil ).
```

3.4. delete(D, E, N) wiw gdy N jest drzewem powstałym przez usunięcie elementu E z drzewa D (drzewa BST)

```
delete( tree( nil, Elem, P ), Elem, P ).
delete( tree( L, Elem, nil ), Elem, L ).
delete( tree( L, W, P ), Elem, tree( NL, W, P ) ) :-
       Elem < W,
       delete( L, Elem, NL ).
delete( tree( L, W, P ), Elem, tree( L, W, NP ) ) :-
       Elem > W,
       delete( P, Elem, NP ).
delete( tree( L, El, P ), El, tree( NL, Max, P ) ) :-
%      L \= nil, P \= nil,
       rem_max( L, Max, NL ).

rem_max( tree( L, Max, nil ), Max, L ).
rem_max( tree( L, W, P ), Max, tree( L, W, NP ) ) :-
%       P \= nil,
        rem_max( P, Max, NP ).
```

3.5. stwórz_drzewo(Lista, Drzewo)

```
   stworz_drzewo( [], nil ).
   stworz_drzewo( [ E | L ], D ) :-
                  stworz_drzewo( L, LD ),
                  insert( LD, E, D ).
% lub
   stworz_drzewo( L, D ) :-  lst_drzewo( L, nil, D ).

   lst_drzewo( [], D, D ).
   lst_drzewo( [ E | L ], T, D ) :-
              insert( T, E, NT ),
              lst_drzewo( L, NT, D ).
```

3.6. liście(D, L) wiw gdy L jest listą liści drzewa D

```
liscie( nil, [] ).
liscie( tree( nil, E, nil ), [ E ] ).     % lisc
liscie( tree( L, E, P ), LL ) :-
%         ( L \= nil ; P \= nil ),
          liscie( L, L1 ),
          liscie( P, L2 ),
          append( L1, L2, LL ).
```

Wersja bez append, z akumulatorem (kolejność: od lewej do prawej).

```
liscie( D, LL ) :- liscie( D, LL, [] ).

liscie( nil, L, L ).
liscie( tree( nil, E, nil ), [ E | L ], L ).
liscie( tree( L, E, P ), LL, TL ) :-
        liscie( L, LL, L1 ),
        liscie( P, L1, TL ).
```

3.7. wszerz(D, LW) wiw gdy LW jest listą wierzchołków drzewa D przy przejściu wszerz

```
wszerz( D, LW ) :-  wszerz_pom( [ D ], LW ).

wszerz_pom( [], [] ).
wszerz_pom( [ nil | KD ], LW ) :-
           wszerz_pom( KD, LW ).
wszerz_pom( [ tree( L, W, P ) | KD ], [ W | LW ] ) :-
           append( KD, [ L, P ], NKD ),
           wszerz_pom( NKD, LW ).
```

Grafy - reprezentacja, podstawowe operacje

- graf — lista krawędzi, np. `[ a<->b, b<->c, ... ]`
  `[ kr(a, b, et1), kr(b, c, et2), ... ]`

- klauzulowa reprezentacja struktur danych

`edge( W1, W2 ).` — krawędź od W1 do W2

`edge( W1, W2, Etyk ).` — krawędzie etykietowane

4.1. connected(X, Y) wiw gdy istnieje ścieżka od X do Y (graf skierowany)

```
connected( X, Y ) :-  edge( X, Y ).
connected( X, Y ) :-  edge( X, Z ),
                      connected( Z, Y ).
```

4.2. path(X, Y, L) wiw gdy L jest ścieżką od X do Y

```
path( X, Y, [ X, Y ] ) :-  edge( X, Y ).
path( X, Y, [ X | L ] ) :-  edge( X, Z ),
                            path( Z, Y, L ).
```

4.3. connected(X, Y) z wykrywaniem cykli w grafie

```
connected( X, Y ) :-  connected( X, Y, [] ).

connected( X, Y, Visited ) :-  edge( X, Y ).
connected( X, Y, Visited ) :-
          edge( X, Z ),
          \+ member( Z, Visited ),
          connected( Z, Y, [ X | Visited ] ).
```

4.4. Graf Eulera — Euler(Graf, Ścieżka_Eulera)

```
euler( [ W1 <-> W2 ], [ W1, W2 ] ).
euler( [ W1 <-> W2 ], [ W2, W1 ] ).
euler( G, [ W1, W2 | Dalej ] ) :-
      pobierz( G, W1 <-> W2, G1 ),
      euler( G1, [ W2 | Dalej ] ).

pobierz( [ W1 <-> W2 | G ], W1 <-> W2, G ).
pobierz( [ W1 <-> W2 | G ], W2 <-> W1, G ).
pobierz( [ E | G ], Kr, [ E | NG ] ) :-
        pobierz( G, Kr, NG ).
```

Kolorowanie mapy (zmienne logiczne)

Dana pewna mapa (np. Europy). Pokoloruj mapę tak, aby każde dwa sąsiednie kraje były pokolorowane różnymi kolorami (4 kolory).

Reprezentacja danych:

- tabelka sąsiedztwa kolorów (klauzule unarne)
- predykat (lub zapytanie) z informacją o sąsiedztwie krajów.

```
obok( zielony, czerwony ).
obok( zielony, niebieski ).
obok( zielony, zolty ).
      obok( niebieski, zielony )
      obok( niebieski, czerwony ).
      obok( niebieski, zolty ).
...
mapa( Polska, Czechy, Ukraina, ... ) :-
     obok( Polska, Czechy ), obok( Polska, Ukraina ), ...

?- mapa( Polska, Czechy, ... ).
```

Technika — generuj i testuj

— generator kandydatów
— sprawdzenie poprawności kandydata
(test jak najgłębiej w generatorze)

```
find( X ) :- generate( X ), test( X ).
```

Przykład — problem N królowych

rozwiązanie problemu — lista numerów wierszy (kolejne kolumny)

Wersja 1: niezależny generator i test.

```
queens( N, Ustaw ) :-  range( 1, N, Lista ),
                       permutation( Lista, Ustaw ),
                       safe( Ustaw ).

safe( [ Q | Qs ] ) :- safe( Qs ), \+ attack( Q, Qs ).
safe( [] ).

attack( Q, Qs ) :-  attack( Q, 1, Qs ).
attack( Q, N, [ X | Xs ] ) :-  Q is X + N; Q is X - N.
attack( Q, N, [ _ | Xs ] ) :-  N1 is N + 1,
                               attack( Q, N1, Xs ).
```

Wersja 2: test wewnątrz generatora.

```
queens( N, Ustaw ) :-  range( 1, N, Lista ),
                       queens( Lista, [], Ustaw ).

queens( Nieustaw, Dobre, Wynik ) :-
       select( Q, Nieustaw, ResztaNieust ),
       not attack( Q, Dobre ),
       queens( ResztaNieust, [ Q | Dobre ], Wynik ).
queens( [], Dobre, Dobre ).
```