Elementy logiki i teorii mnogości
Prolog
laboratorium 2

Zadanie 1.

Zastanowić się, a potem sprawdzić, czy poniższe cele zostaną spełnione i (ewentualnie) jak zostaną ukonkretnione zmienne:

[1,2,3,4]=[A|B].
[A,B]=[A|C].
[W,Z]=[1,2].
[W,Z]=[1,[2]].
[W,Z]=[1|[2]].
[1,[A],2]=[1,0,2].
[1,2,3]=[1,2|[3]].
[1,2,3]=[1|[2|[3]]].
[[A],B,C]=[[a,b,c],[d,e,f],1].
[a,[B]|C]=[Z,[C],1].
[a,[B]|C]=[Z,[C],[1]].
[a|[b,c]]=[X,Y|Z].
[a,b|C]=[W|[a,b,c]].

Zadanie 2.

Sprawdzić działanie procedur działających na listach:

is_list (L) - sprawdza, czy L jest listą
Sprawdzić np.

```
is_list([1,2,3,c,d]).
is_list(5).
is_list([5]).
```

append (L1,L2,L3) – łączy listy L1 i L2 w listę L3
Sprawdzić np.

```
append([b,c,d],[e,f,g,h],X).
append([a],[b],[a,b]).
append(L1,L2,[b,c,d]).
```

member(E,L) – sprawdza, czy element E należy do listy L
Sprawdzić np.

```
member(a,[b,c,[s,a],a]).
member(a,[b,c,[s,a]]).
member([s,a],[b,c,[s,a]]).
member(X,[a,b,c]).
member(a, X).
```

memberchk(E,L) - równoważny predykatowi member, ale podaje tylko jedno rozwiązanie
Sprawdzić np.

```
member(Y,[1,2,3,4]).
memberchk(Y,[1,2,3,4]).
```

nextto(X,Y,L) – predykat spełniony, gdy Y występuje bezpośrednio po X
Sprawdzić np.

```
nextto(X,Y,[a,c,d,r]).
nextto(w,Y,[q,w,e,r]).
nextto(X,4,[2,3,4,5]).
```

delete(L1,E,L2) – z listy L1 usuwa wszystkie wystąpienia elementu E, wynik uzgadnia z listą L2
Sprawdzić np.

```
delete([1,2,3,4],4,M).
delete([2,1,2,1,2,1],1,K).
```

select((E,L,R) – z listy L wybiera element, który daje się uzgodnić z E. Lista R jest uzgadniana z listą, która powstaje z L po usunięciu wybranego elementu
Sprawdzić np.

```
select(1,[2,1,2,1],K).
select(X,[1,2,3],K).
select(0,X,[1,2,3,4]).
```

nth0(I,L,E) – predykat spełniony, jeśli element listy L o numerze I daje się uzgodnić z elementem E
Sprawdzić np.

```
nth0(2,[a,b,c,d],X).
nth0(X,[a,b,c,d],2).
nth0(X,[a,b,c,d],c).
```

nth1(I,L,E) – predykat podobny do nth0. Sprawdzić różnicę!

last(L,E) – ostatni element listy L
Sprawdzić np.

```
last([1,2,3,4],L).
last(X,2).
```

reverse(L1,L2) – odwraca porządek elementów listy L1 i unifikuje rezultat z listą L2
Sprawdzić np.

```
reverse([1,2,3,4],X).
reverse(Y,[a,b,c,d,e,f]).
```

permutation(L1,L2) – lista L1 jest permutacja listy L2
Sprawdzić np.

```
permutation([1,2,3],L).
permutation(M,[4,5,6,7]).
```

sumlist(L,S) – suma listy liczbowej L
Sprawdzić np.

```
sumlist([1,2,3,4],X).
sumlist([1,2,3,4],10).
```

numlist(M,N,L) – jeśli M,N są liczbami całkowitymi takimi, że M<N, to L zostanie zunifikowana z listą [M,M+1,…,N]
Sprawdzić np.

```
numlist(2,8,L).
numlist(-3,5,X).
```

length(L,I) – liczba elementów listy L
Sprawdzić np.

```
length([1,3,4,23,21,8],L).
length([a,e,[a],[x,y],l],T).
```

Zadanie 3.

Dana jest procedura **połacz:**

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

Napisać kolejne wywołania (poszukiwanie odpowiedzi w Prologu) tej procedury dla zapytania:

```
polacz([1,2],[a,b],X).
```

Zadanie 4.

Zdefiniować predykat **ostatni** znajdujący ostatni element listy.

```
?-ostatni([2,3,2,4,3,2],O).
  O=2.
?-ostatni([1,2,1,4,3],6).
  false.
```

Zadanie 5.

Zdefiniować predykat **rosnacy** który sprawdza, czy kolejne elementy listy L tworzą ciąg ściśle rosnący.

```
?-rosnący([3,6,7,12,29]).
  true.
?-rosnący([3,2,7,12,2]).
  false
```

Zadanie 6.

Zdefiniować procedurę, która dla dwóch list takiej samej długości tworzy nową w taki sposób, że element na *i*-tym miejscu jest sumą *i*-tych elementów list składowych.

Np. ?-sm([1,2,3,4],[3,4,5,6],X).
X=[4,6,8,10].

Zadanie 7.

Zdefiniować procedurę, która sprawdza, czy dwie listy mają taką samą liczbę elementów, na dwa sposoby:

a) z wykorzystaniem predykatu *length,*
b) bez użycia l*ength*.

Np. ?-rowne([a,b,c],[2,3,4]).
true.
?-rowne([],[1,2,3,4]).
false.
?-rowne([x,y,z,x],[0,1]).
false.