Kurs języka Prolog 2019

Lista zadań nr 3

Na zajęcia 20 marca 2019

Zadanie 1 (6 pkt). Zaprogramuj w Prologu predykaty:

1. `exp(+Base, +Exp, ?Res)`, spełniony, gdy `Res` unifikuje się wynikiem podniesienia liczby `Base` do potęgi `Exp`.
2. `exp(+Base, +Exp, +Mod, ?Res)`, spełniony, gdy `Res` unifikuje się z resztą dzielenia przez `Mod` wyniku podniesienia liczby `Base` do potęgi `Exp`.
3. `factorial(+N, ?M)`, spełniony, gdy `M` unifikuje się z silnią liczby `N`.
4. `concat_number(+Digits, ?Num)`, spełniony, gdy lista `Digits` zawiera ciąg cyfr rozwinięcia dziesiętnego liczby, która unifikuje się z `Num`.
5. `decimal(+Num, ?Digits)`, spełniony, gdy `Digits` unifikuje się z z ciągiem cyfr rozwinięcia dziesiętnego liczby `Num`. Program powinien zwracać poprawny wynik dla liczb nieujemnych.
6. `filter(+LNum, ?LPos)`, spełniony, gdy `LPos` unifikuje się z podlistą listy `LNum` zawierającą wszystkie nieujemne elementy listy `LNum`.

Zadanie 2 (1 pkt). Zaprogramuj w Prologu predykat `count(+Elem, +List, ?Count)`, spełniony, gdy `Elem` unifikuje się z dokładnie n elementami listy `List` i `Count` unifikuje się z liczbą n. Jakie powinny być odpowiedzi na zapytanie

```
?- count(f(X), [f(Y), f(a), c, f(b), f(Z)], N).
```

Zadanie 3 (1 pkt). Zaprogramuj w Prologu predykat `length/2`, który nie zapętla dla żadnego trybu użycia. W szczególności cel `length(X,5)` powinien być spełniony dokładnie na jeden sposób, a pod `X` powinna zostać podstawiona lista `[_,_,_,_,_]`. Zauważ, że drugi argument może być tylko termem arytmetycznym albo zmienną. Możesz rozróżnić te dwa przypadki za pomocą predykatu sprawdzającego typ termu, np. `var/1`.

Zadanie 4 (2 pkt). Zaprogramuj predykat `prime/1` implementujący sito Eratostenesa, który działa poprawnie zarówno w przypadku generowania (tj. wywołany z nieukonkretnioną zmienną jako parametrem), jak i sprawdzania. W tym drugim przypadku jego argumentem może być dowolny term arytmetyczny (niekoniecznie literał całkowitoliczbowy). W przypadku wywołania z innym parametrem powinien zostać zgłoszony błąd arytmetyczny. Podczas przesiewania kandydat na liczbę pierwszą powinien być porównywany z wcześniej wygenerowanymi liczbami pierwszymi w kolejności od najmniejszej do największej.

Zadanie 5 (1 pkt). Napisz predykat `append/2`, który, podobnie jak `append/3` łączy listy, jednak nie dwie, tylko dowolną ich liczbę. Listy do połączenia są elementami listy będącej jego pierwszym parametrem, np.:

```
?- append([[1,2,3], [4,5], [6,7,8,9]], Y).
Y = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Zadbaj o to, żeby predykat działał efektywnie.

Zadanie 6 (1 pkt). Napisz zapytanie prologowe, które rozwiąże następującą łamigłówkę:

```
          U  S  A
+      U  S  S  R
-----------------
=  P  E  A  C  E
```

Zapytanie powinno zawierać zmienne `A`, `C`, `E`, `P`, `R`, `S` i `U` oraz powinno sprawdzić wszystkie możliwe podstawienia pod te zmienne parami różnych cyfr `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8` i `9`. Cyfry podstawione pod zmienne `U` i `P` nie mogą być zerem. W zapytaniu możesz użyć predykatów standardowych `permutation/2`, `length/2`, `is/2`, `(\=)/2`, i predykatów `concat_number/2` i `sublist/2` z poprzednich zadań. Zapytanie powinno zwrócić dokładnie jedną odpowiedź, bez powtórzeń:

```
?- zapytanie (pominięte).
A = 2, C = 7, E = 0, P = 1, R = 8, S = 3, U = 9;
false.
?-
```

Możesz też rozwiązać inne kryptarytmy, np.:

```
                    COCA + COLA = OASIS
                SPOT + A + TOP = GHOST
          YES + YES + YES + YES = EVER
               LEW + WILL + BE = ABLE
                    USED + SEX = WORDS
                   MEMO + FROM = HOMER
                   SEEN + SOME = BONES
                   MEET + MOST = TEENS
     TELL + TALE + TELL + TALE = LATE
WOW + WOW + WOW + WOW + WOW = MEOW
      TOO + TOO + TOO + TOO = GOOD
                   WHAT + THAT = HERE
                 LEAH + LOVES = RUSSIA
                   THIS + SIZE = SHORT
               MAKE + A + CAKE = EMMA
```

Zadanie 7 (2 pkt). Rozważmy predykat

```
sum(X,Y,Z) :-
   Z is X + Y.
```

Ten predykat działa poprawnie tylko w trybie `(+,+,?)`. Zaprogramuj predykat, który działa poprawnie w każdym trybie, w tym `(-,-,-)`, np.

```
?- sum(2,3,X).
X = 5
?- sum(X,4,6).
X = 2
?- sum(X,Y,10).
X = 0, Y = 10;
X = 1, Y = 9;
X = -1, Y = 11;
X = 2, Y = 8;
X = -2, Y = 12
?- sum(X,Y,Z).
X = 0, Y = 0, Z = 0;
X = 1, Y = 0, Z = 1
```

Predykat powinien generować wszystkie rozwiązania całkowitoliczbowe (jest ich nieskończenie wiele). Np. w przypadku zapytania `sum(X,Y,Z)`, gdzie `X`, `Y` i `Z` są nieukonkretnionymi zmiennymi, predykat powinien wygenerować każdą trójkę takich liczb (x, y, z), że $x + y = z$.

Zadanie 8 (2 pkt). Rozważany w zadaniu z poprzedniej listy predykat `halve/4`

```
halve(T,[],[],T) :-
   !.
halve(T,[_],[],T) :-
   !.
halve([H|T],[_,_|S],[H|L],R) :-
   halve(T,S,L,R).
```

kopiuje pierwszą połowę listy (i współdzieli drugą połowę). Aby uniknąć korzystania z predykatu `halve/4` (i związanego z nim narzutu) można uogólnić nieco predykat `merge_sort` dodając dodatkowy parametr wejściowy `N` i wyjściowy `T`. Cel `merge_sort(+X,+N,-Y,-T)` powinien posortować `N` pierwszych elementów listy `X`, zunifikować wynik ze zmienną `Y` a „resztę" listy `X` zunifikować ze zmienną `T`. Aby posortować `N` elementów listy `X` należy teraz posortować rekurencyjnie $\lfloor N/2 \rfloor$ pierwszych elementów listy `X` oraz $N - \lfloor N/2 \rfloor$ pozostałych elementów tej listy, a następnie scalić oba wyniki sortowania. Zaprogramuj taką wersję predykatu `merge_sort`.

Zadanie 9 (2 pkt). W poprzednich zadaniach rozważaliśmy algorytm *Mergesort* w wersji „z góry na dół" (*top down*). Algorytm *Mergesort* w wersji „z dołu do góry" (*bottom up*) działa następująco: tworzymy listę jednoelementowych list zawierających sortowane elementy. Następnie scalamy te listy parami tworząc listę list dwuelementowych, następnie scalamy listy dwuelementowe parami tworząc listy czteroelementowe itd., aż otrzymamy pojedynczą posortowaną listę. Zaprogramuj taką wersję predykatu `merge_sort`.

Zadanie 10 (2 pkt). Niech predykat `connection/2` (zdefiniowany przez zbiór faktów) oznacza, że istnieje bezpośrednie połączenie między dwoma miastami. Zdefiniuj predykat `trip/3`, który znajduje połączenie między dwoma miastami z dowolną liczbą przesiadek i nie zapętla się nawet wówczas, gdy w grafie połączeń są cykle. Pierwsze dwa parametry tego predykatu (wejściowe), to miasta początkowe i końcowe. Trzeci parametr (wyjściowy), to lista miast od początkowego do końcowego, przez które przebiega podróż, np.:

```
?- trip(gliwice, warszawa, T).
T = [gliwice, wroclaw, warszawa] ;
T = [gliwice, wroclaw, katowice, warszawa] ;
false.
```

Wskazówka: zdefiniuj wpierw predykat `trip/4`, którego dodatkowy parametr jest „akumulatorem" — listą zawierającą już odwiedzone miasta. Użyj predykatu `member/2` by sprawdzić, czy miasto, do którego chcemy przejść, było już wcześniej odwiedzone. Ścieżkę buduj od końca ku początkowi, by uniknąć konieczności odwracania listy po znalezieniu rozwiązania.