Metody programowania 2015

Lista zadań nr 5

Na zajęcia 30 marca – 2 kwietnia 2015

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

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

Predykat musi działać efektywnie. W szczególności wywołania rekurencyjne powinny być ogonowe. *Wskazówka:* użyj otwartych struktur danych.

Zadanie 2 (1 pkt). Napisz predykat `flatten/2`, który „spłaszcza" listę:

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

Zadbaj o to, by Twój predykat nie generował nieużytków!

Zadanie 3 (1 pkt). Napisz predykat `halve(+X,-L,-R)` który kopiuje pierwszą połowę listy `X` do listy `L` i unifikuje zmienną `R` z drugą połową listy `X`. Lista `L` powinna zostać skopiowana, zaś lista `R` — współdzielona. Jeśli lista `X` ma nieparzystą liczbę elementów, to dłuższa powinna być lista `R`. *Wskazówka:* Nie musisz znać długości listy, żeby zlokalizować jej środek! Ustaw dwa wskaźniki (powiedzmy Y_1 i Y_2) na początek listy. W każdym kroku iteracji kopiuj element wskazywany przez Y_1 i przesuwaj Y_1 o jeden element listy, zaś Y_2 o dwa elementy listy. Zakończ iterację, gdy Y_2 dojdzie do końca listy.

Zadanie 4 (1 pkt). Zaprogramuj predykat `merge/3` który, zachowując porządek, łączy dwie listy liczb całkowitych uporządkowane rosnąco. Zadbaj o to, by predykat działał deterministycznie, a rekursja była ogonowa (tam, gdzie to niezbędne, użyj odcięć). *Wskazówka:* buduj listę poczynając od głowy i wybieraj jako jej kolejny element mniejszą spośród głów scalanych list. Gdy jedna z list się opróżni, zakończ iterację.

Użyj predykatów `halve/3` i `merge/3` do zdefiniowania predykatu `merge_sort/2` implementującego algorytm sortowania *Mergesort*.

Zadanie 5 (1 pkt). W poprzednim zadaniu predykat `halve/3` kopiuje pierwszą połowę listy (i współdzieli drugą połowę). Aby uniknąć korzystania z predykatu `halve/3` (i związanego z nim narzutu) można uogólnić nieco predykat `merge_sort` dodając dodatkowy parametr wejściowy `N`. Cel `merge_sort(+X,+N,?Y)` powinien posortować `N` pierwszych elementów listy `X` i wynik zunifikować ze zmienną `Y`. 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 6 (1 pkt). W poprzednich zadaniach rozważaliśmy alogorytm *Mergesort* w wersji „z góry na dół" (*top down*). Ten algorytm 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 7 (1 pkt). Zdefiniuj predykat

```
split(+List, +Med, -Small, -Big)
```

rozdzielający listę `List` liczb całkowitych na listy: `Small` — elementów mniejszych i `Big` — elementów nie mniejszych niż liczba `Med`. Wykorzystaj go do zdefiniowania predykatu `qsort/2` implementującego algorytm *Quicksort*. Uwaga: w fazie łączenia posortowanych list nie należy generować nieużytków — nie używaj więc do tego celu predykatu append/3:

```
qsort([], []).
qsort([H|T], R) :-
   split(T, H, S, B),
   qsort(S, S1),
   qsort(B, B1),
   append(S1, [H|B1], R).  % Źle! S1 staje
                           % się nieużytkiem!
```

Dodatkowy akumulator powinien wystarczyć. Jak zwykle uogólnij problem i zaprogramuj predykat `qsort(L,A,R)` tak, by `R` była połączeniem wyniku sortowania listy `L` z listą `A`.

Zadanie 8 (1 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. Użyj listy różnicowej aby sprawnie zaimplementować wstawianie liczby na koniec listy.

Zadanie 9 (1 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 $x + y = z$.