Programowanie Funkcyjne 2025

Lista zadań nr 2

Na zajęcia 23 i 24 października 2025

Zadanie 1 (2p). Wyraź standardowe funkcje `length`, `rev`, `map`, `append`, `rev_append`, `filter`, `rev_map` operujące na listach za pomocą funkcji `List.fold_left` albo `List.fold_right` (wybierz bardziej odpowiednią do każdego z przypadków).

Zadanie 2 (1p). Zdefiniuj funkcję `sublists` znajdującą wszystkie podlisty (rozumiane jako podciągi, niekoniecznie kolejnych elementów) listy zadanej jako argument. Zadbaj o to by Twoja funkcja nie generowała nieużytków.

Zadanie 3 (1p). Zdefiniuj funkcję generującą wszystkie sufiksy danej listy. Na przykład dla listy `[1; 2; 3]` Twoja funkcja powinna zwrócić listę `[[1; 2; 3]; [2; 3]; [3]; []]`. Następnie, zdefiniuj funkcję generującą wszystkie prefiksy danej listy. Na przykład dla listy `[1; 2; 3]` Twoja funkcja powinna zwrócić listę `[[]; [1]; [1; 2]; [1; 2; 3]]`.

Zadanie 4 (4p). Sortowanie przez scalanie.

1. Zdefiniuj funkcję `merge`, która łączy dwie listy posortowane rosnąco w pewnym porządku tak, by wynik działania funkcji był także listą posortowaną rosnąco w tym samym porządku. Argumentami funkcji `merge` powinny być: funkcja `cmp: 'a -> 'a -> bool` (reprezentująca porządek) oraz dwie listy elementów typu `'a`. Na przykład

   ```
   merge (<=) [1; 2; 5] [3; 4; 5] = [1; 2; 3; 4; 5; 5].
   ```

2. Zapisz tę samą funkcję używając rekursji ogonowej, a następnie porównaj działanie obu funkcji na odpowiednich przykładach. Która z nich jest lepsza?
3. Zdefiniuj funkcję `halve` dzielącą listę w połowie. Postaraj się nie wyliczać jawnie długości listy i użyć tylko $n/2$ wywołań rekurencyjnych (gdzie n jest długością listy).
4. Wykorzystaj funkcję `merge` do napisania funkcji `mergesort` sortującej listę przez scalanie.

Zadanie 5 (2p). Zaproponuj alternatywną implementację algorytmu sortowania przez scalanie, w której funkcja `merge` jest ogonowa, ale nie wykonuje odwracania list. Nie przejmuj się, jeżeli otrzymasz algorytm sortowania, który nie jest stabilny. Porównaj szybkość działania tej implementacji z definicją z poprzedniego zadania.

Wskazówka: Twoja funkcja `merge` może jednocześnie scalać i odwracać listy. Zastanów się jak przy jej pomocy posortować listę.

Zadanie 6 (3p). Zdefiniuj funkcję zwracającą listę wszystkich permutacji zadanej listy. Można zrobić to na dwa istotnie różne sposoby: przez wybieranie i przez wstawianie. Czy potrafisz zaimplementować każdy z nich?

Zadanie 7 (4p). Kopiec lewicowy, to drzewo binarne (etykietowane w węzłach), które ma następujące dwie własności:

- zachowany jest porządek kopcowy, tzn. dla każdego węzła przechowywana wartość jest nie większa niż każda wartość w jego dzieciach,
- dla każdego węzła, długość prawego kręgosłupa lewego syna jest nie mniejsza niż długość prawego kręgosłupa prawego syna (długość prawego kręgosłupa, to odległość od wierzchołka do liścia idąc cały czas w prawo).

Zaproponuj reprezentację drzew lewicowych tak, aby dla każdego wierzchołka dało się policzyć długość prawego kręgosłupa w czasie stałym. Następnie zaimplementuj następujące operacje.

- Tworzenie węzła drzewa lewicowego z dwóch drzew i podanej etykiety (tzw. *smart-constructor*). Możesz założyć, że podana etykieta jest nie większa niż wszystkie etykiety w podanych drzewach.
- Złączanie dwóch drzew lewicowych w jedno. W tym celu należy wybrać drzewo o najmniejszym korzeniu, rozbić je na etykietę (nowy korzeń) i dwa poddrzewa, z których prawe należy złączyć z nie wybranym drzewem.
- Wstawianie elementu do drzewa lewicowego. Najłatwiej to zrobić przy pomocy już zaimplementowanego złączania.
- Usuwanie najmniejszego elementu z drzewa. Powstałe poddrzewa należy złączyć w jedno.

Do porównywania elementów użyj wbudowanej funkcji porównującej `(<)`.

Zadanie 8 (1p). Mając implementację drzew lewicowych z poprzedniego zadania zaimplementuj prosty moduł dostarczający kolejkę priorytetową (kopiec). Co należy ujawnić w pliku `.mli`, by nie dało się naruszyć niezmienników struktury danych, ale dało się modułu używać, np. do implementacji sortowania?

Zadanie 9 (2p). Struktura list jest bardzo podobna do struktury liczb naturalnych reprezentowanych unarnie (ciąg następników nałożonych na zero). Jedyna różnica polega na tym, że w listach do następników (zwanych *consami*), przyczepiamy dane — elementy listy. Na poprzedniej liście zadań widzieliśmy reprezentację Churcha liczb naturalnych. W podobny sposób możemy przedstawić listy. Jak można się spodziewać, jedyna różnica polega na tym, że iterowana funkcja przyjmuje dodatkowy argument — element listy.

```
type 'a clist = { clist : 'z. ('a -> 'z -> 'z) -> 'z -> 'z }
```

W ten sposób lista $[a_1, a_2, \ldots, a_n]$, jest reprezentowana przez funkcję dwuargumentową, która dla argumentów `f` oraz `z` obliczy `f` a_1 `(f` a_2 `( ... (f` a_n `z) ... ))`.

Zaimplementuj następujące operacje na listach w kodowaniu Churcha:

- `cnil : 'a clist` — lista pusta,
- `ccons : 'a -> 'a clist -> 'a clist` — dołączanie głowy do listy,
- `map : ('a -> 'b) -> 'a clist -> 'b clist` — nakładania podanej funkcji na wszystkie wartości listy,
- `append : 'a clist -> 'a clist -> 'a clist` — konkatenacja list,
- `clist_to_list : 'a clist -> 'a list`
  oraz `clist_of_list : 'a list -> 'a clist` — konwersja między listami Churcha, a standardowymi listami.

Zauważ, że implementacja operacji `cnil`, `ccons` oraz `append` jest bardzo podobna do implementacji `zero`, `succ` oraz `add` dla liczebników Churcha. Zaimplementuj funkcję `prod : 'a clist -> 'b clist -> ('a * 'b) clist`, która będzie analogiczna do mnożenia. Co robi ta funkcja? Czy potrafisz zaimplementować funkcję analogiczną do potęgowania?