Lista zagadnień nr 4

Ćwiczenie 1.

Zaimplementuj następujące procedury w dwóch wersjach: (1) używając rekursji i (2) procedury `foldr`.

- `(define (elem x xs) ... )` – sprawdza czy element `x` znajduje się na liście `xs` (użyj predykatu `eq?` by porównać równość elementów). Np.:

```
> (elem #\a (string->list "dobranoc"))
#t
> (elem #\a (string->list "dzien dobry"))
#f
```

- `(define (filter p xs) ... )` – tworzy z zachowaniem kolejności listę tych elementów listy `xs`, które spełniają predykat `p`, np.:

```
> (filter odd? (list 1 5 0 7 1 4 1 0))
'(1 5 7 1 1)
```

- `(define (maximum xs) ... )` – ujawnia największy element na liście (względem predykatu `>`). Zwraca błąd, jeśli `xs` jest listą pustą (błąd można zgłosić procedurą `error`), np.:

```
> (maximum (list 1 5 0 7 1 4 1 0))
7
> (maximum (list))
ERROR: pamietaj, cholero, maximum nie dziala z lista dlugosci zero
```

- `(define (zip xs ys) ... )` – bierze jako argumenty dwie listy równej długości i paruje odpowiadające sobie elementy, np.

```
> (zip '(1 2 3 4) (string->list "abcd"))
'((1 . #\a) (2 . #\b) (3 . #\c) (4 . #\d))
```

- `(define (every-other xs) ... )` – zwraca listę elementów znajdujących się na nieparzystych pozycjach (rozpoczynając od 1) listy `xs`, np.

```
> (every-other (list 1 5 0 7 1 4 1 0))
'(1 0 1 1)
> (every-other (list 0 1 5 0 7 1 4 1 0))
'(0 5 7 4 0)
```

- `(define (tails xs) ... )` – zwraca listę zawierającą wszystkie „ogony" listy `xs`, czyli samą listę `xs`, jej ogon, ogon jej ogona, ogon ogona ogona, ogon ogona ogona ogona itd. Np.:

```
> (tails (list 0 1 2 3))
'((0 1 2 3) (1 2 3) (2 3) (3) ())
> (map list->string (tails (string->list "metody")))
'("metody" "etody" "tody" "ody" "dy" "y" "")
```

Ćwiczenie 2.

Zaimplementuj procedury:

- `(define (insert x xs) ... )` – wstawia do posortowanej listy `xs` wartość `x` tak, by lista wynikowa też była posortowana, np.

```
> (insert 4 (list 0 1 1 5 7))
'(0 1 1 4 5 7)
```

- `(define (insertion-sort xs) ... )` – rozpoczynając od pustej listy, wkłada po kolei elementy używając procedury `insert`, np.:

```
> (insertion-sort '(1 5 0 7 1 4 1 0))
'(0 0 1 1 1 4 5 7)
```

Ćwiczenie 3.

Zaimplementuj poniższe procedury. Ile generują nieużytków? Ile razy przechodzą listę? (Uwaga: jeśli jakaś procedura zwraca więcej niż jedną wartość poprzez użycie pary lub listy stałej długości, nie liczymy tego jako nieużytków!)

- `(define (palindrome? xs) ... )` – czy lista `xs` jest palindromem? Np.:

```
> (palindrome? (string->list "kajak"))
#t
> (palindrome? (string->list "zakopanenapokaz"))
#t
> (palindrome? (string->list "metody"))
#f
```

Wskazówka: Istnieje rozwiązanie, w którym przechodzimy listę tylko raz i nie generujemy nieużytków.

- `(define (rotate-left n xs) ... )` – przesuwa listę o `n` pozycji w lewo, przy czym elementy, które „wypadły" z lewej, pojawiają się po prawej, np.:

```
> (rotate-left 2 (list 1 2 3 4 5))
'(3 4 5 1 2)
> (rotate-left 7 (list 1 2 3 4 5))
'(3 4 5 1 2)
> (rotate-left -2 (list 1 2 3 4 5))
'(4 5 1 2 3)
> (rotate-left -11 (list 1 2 3 4 5))
'(5 1 2 3 4)
```

Wskazówka: Autor zadania nie wie, jakie rozwiązanie jest optymalne. Najlepsze, co udało mu się osiągnąć to:

  - Brak nieużytków
  - Jeśli rotacja następuje w lewo o mniej kroków niż długość listy, przechodzimy listę raz. W przeciwnym razie przechodzimy ją dwa razy.

Ćwiczenie 4.

Permutacją ciągu $x_1, x_2, \ldots, x_n$ nazywamy dowolny ciąg $x_{i_1}, x_{i_2}, \ldots, x_{i_n}$, gdzie $i_1, \ldots i_n$ są parami różnymi liczbami z przedziału $1, \ldots, n$. Permutacje powstają zatem przez przestawienie kolejności elementów na liście.

Zdefiniuj procedurę `permi` zwracającą listę wszystkich permutacji danej listy, działającą zgodnie z poniższym schematem:

- Jedyną permutacją listy pustej jest lista pusta.
- Aby wygenerować permutację niepustej listy wygeneruj permutację jej ogona i wstaw jej głowę w dowolne miejsce uzyskanej permutacji.

Ćwiczenie 5.

Udowodnij, że dla dowolnej listy `xs` oraz procedur `f` i `g` zachodzi

$$\texttt{(map (compose f g) xs)} \equiv \texttt{(map f (map g xs))}$$

Ćwiczenie 6.

Zdefiniuj iteracyjną wersję procedury `length` i udowodnij jej równoważność z wersją podaną na wykładzie.

Ćwiczenie 7.

Udowodnij tzw. prawo fuzji: jeśli dla każdego `x` i `y` zachodzi

$$\texttt{(f (g x y))} \equiv \texttt{(h x (f y))}$$

to dla każdej listy `xs`:

$$\texttt{(f (foldr g a xs))} \equiv \texttt{(foldr h (f a) xs)}$$