Lista zadań nr 2

Zadanie 1. (2 pkt)

Ciąg Fibonacciego definiuje się rekurencyjnie w następujący sposób:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad \text{gdy } n > 1 \end{aligned}$$

Inspirując się dwoma implementacjami silni przedstawionymi na wykładzie, zaimplementuj dwie procedury obliczające wartość F_n:

- `fib` – wersję rekurencyjną, obliczającą wartość zgodnie z definicją powyżej,
- `fib-iter` – wersję iteracyjną, wykorzystującą pomocniczą procedurę z dwoma dodatkowymi argumentami, reprezentującymi dwie poprzednie wartości ciągu Fibonacciego względem aktualnie obliczanej.

Porównaj czas trwania obliczeń obydwu implementacji dla różnych wartości n. Wyjaśnij w intuicyjny sposób zaobserwowaną różnicę, odwołując się do podstawieniowego modelu obliczeń poznanego na wykładzie.

Zadanie 2. (2 pkt)

Zdefiniuj typ danych macierzy `matrix` o wymiarze 2×2 przy użyciu formy specjalnej `define-struct`. (Macierze tego rozmiaru mają 4 elementy, które można nazwać np. `a`, `b`, `c`, `d`.) Dla tego typu danych zdefiniuj:

- `(define (matrix-mult m n)...)` – iloczyn dwóch macierzy.
- `(define matrix-id ...)` – macierz identycznościowa.
- `(define (matrix-expt m k)...)` – podnosi macierz `m` do `k`-tej potęgi (naturalnej). Potęgowanie można obliczać przez wielokrotne mnożenie.

Korzystając z tych definicji, zdefiniuj procedurę `fib-matrix` obliczającą k-tą liczbę Fibonacciego F_k na podstawie zależności:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^k = \begin{bmatrix} F_{k+1} & F_k \\ F_k & F_{k-1} \end{bmatrix}$$

Zadanie 3.

Zdefiniuj procedury `matrix-expt-fast` i `fib-fast` analogiczne do tych z poprzedniego zadania, ale stosujące algorytm szybkiego potęgowania. Algorytm ten wykorzystuje poniższą zależność dla wykładników parzystych:

$$M^{2k} = (M^k)^2$$

Porównaj wydajność procedury `fib-fast` z procedurą `fib-matrix` lub `fib-iter` (z poprzednich zadań).

Zadanie 4.

Zaimplementuj procedurę `(elem? x xs)` sprawdzającą, czy element `x` znajduje się na liście `xs` (użyj predykatu `equal?`). Przykład:

```
> (elem 2 (list 1 2 3))
#t
> (elem 4 (list 1 2 3))
#f
```

Zadanie 5.

Zaimplementuj procedurę `(maximum xs)` znajdującą największy element na liście (względem predykatu `>`). Jeśli lista `xs` pusta, zwracana jest wartość `-inf.0` (minus nieskończoność). Przykład:

```
> (maximum (list 1 5 0 7 1 4 1 0))
7
> (maximum (list))
-inf.0
```

Zadanie 6.

Zaimplementuj procedurę `(suffixes xs)` zwracającą wszystkie sufiksy listy `xs` – czyli takie listy, które zawierają, w kolejności i bez powtórzeń, elementy listy `xs` od zadanego elementu aż do końca listy. Listę pustą uznajemy za sufiks dowolnej listy. Przykład:

```
> (suffixes (list 1 2 3 4))
'((1 2 3 4) (2 3 4) (3 4) (4) ())
```

Zadanie 7.

Zaimplementuj procedurę `(sorted? xs)` sprawdzającą, czy zadana lista jest posortowana niemalejąco.

Zadanie 8. (2 pkt)

Na wykładzie przedstawiono implementację algorytmu sortowania przez wstawianie. Zaimplementuj inny znany algorytm sortowania w czasie $O(n^2)$: sortowanie przez wybór. Dokładniej, zaimplementuj następujące procedury:

- `(select xs)` – zwraca parę składającą się z najmniejszego elementu listy `xs` oraz listy wszystkich elementów `xs` oprócz najmniejszego. Można też myśleć o tej procedurze, że zwraca ona taką permutację listy `xs`, w której najmniejszy element jest na pierwszej pozycji, a kolejność pozostałych elementów pozostała niezmieniona. Przykład:

```
> (select (list 4 3 1 2 5))
'(1 4 3 2 5)
```

- `(select-sort xs)` – sortuje listę algorytmem sortowania przez wybór. Dla list niepustych, procedura ta znajduje najmniejszy element używając procedury `select`. Znaleziony element staje się pierwszym elementem listy wynikowej. Pozostałe elementy sortowane są tą samą metodą. Przykład:

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

Zadanie 9.

Zaimplementuj algorytm sortowania przez złączanie. Dokładniej, zaimplementuj następujące procedury:

- `(split xs)` – zwraca parę dwóch list różniących się długością o co najwyżej 1, oraz zawierających wszystkie elementy listy `xs`. Przykład:

```
> (split (list 8 2 4 7 4 2 1))
'((8 4 4 1) 2 7 2)
; albo: '((8 2 4 7) 4 2 1)
```

- `(merge xs ys)` – dla argumentów będących posortowanymi listami zwraca posortowaną listę wszystkich elementów `xs` i `ys`. Przykład:

```
> (merge (list 1 4 4 8) (list 2 2 7))
'(1 2 2 4 4 7 8)
```

- `(merge-sort xs)` – sortuje listę algorytmem sortowania przez złączanie. Dla list długości większej niż 1, procedura ta dzieli listę wejściową na dwie prawie równe części, sortuje je rekurencyjnie, a następnie złącza posortowane wyniki.

Czy procedura `merge-sort` jest strukturalnie rekurencyjna?