Paradigmata programovaní 1

Instrukce:

- Řešení v podobě zdrojového kódu ve Scheme zasílejte v jediném souboru na e-mail

  `eduard.bartl@upol.cz`.

- Jako předmět e-mailové zprávy uveďte

  `PAPR1 -- reseni ukolu c.  2`.

- Do zdrojového kódu nepište ani příklady užití uvedené v zadání ani Vaše testy.
- Soubor se zdrojovým kódem pojmenujte jako

  `UzivatelskeJmeno.ss` (nebo `UzivatelskeJmeno.rkt`),

  kde `UzivatelskeJmeno` je uživatelské jméno ve Vaší univerzitní e-mailové adrese.

Příklad: Student Jiří Novák s univerzitní e-mailovou adresou `jiri.novak02@upol.cz` si ví rady pouze s prvními třemi příklady v níže uvedeném zadání. Vytvoří tedy soubor `jiri.novak02.ss` (nebo `jiri.novak02.rkt`) s následujícím obsahem:

```
> (map-index-pred
     (lambda(i) (>= i 2))
     (lambda (x) (+ x 1))
     '(1 2 3 4 5))
(1 3 4 5 6)

> (divlist 4)
(#t #t #f #t)
```

Student nedostane žádný bod, protože oba příklady jsou špatně. První procedura nedává správné výsledky, druhá procedura (bez ohledu na to, zdali funguje) nebude otestována, protože není navázána na symbol, který je uveden v zadání (viz níže).

Upozornění: Nedodržení instrukcí může znamenat neuznání celého úkolu.

Úkol č. 2

datum zadaní: 18. listopadu 2012
termín odevzdání: 28. listopadu 2012

1. **(9 bodů)** Vytvořte proceduru vyššího řádu `my-cons` realizující konstrukci páru. Tato procedura bude přijímat dva argumenty odpovídající prvkům vytvářeného páru. Aplikací vznikne procedura jednoho argumentu. Je-li tímto argumentem `#f`, bude vrácen druhý prvek páru, jinak bude vrácen první prvek páru. Dále vytvořte procedury `my-car` a `my-cdr`. Aplikací těchto procedur na pár vytvořený pomocí `my-cons` bude vrácen první, resp. druhý prvek prvek daného páru.

   Příklady použití:

   ```
   > (define p1 (my-cons 3 8))
   > (p1 #t)
   3
   > (p1 #f)
   8
   > (my-car p1)
   3
   > (my-cdr p1)
   8
   ```

2. **(4 body)** Implementujte proceduru `switch`, která vymění prvky páru vytvořeného pomocí procedury `my-cons`.

   Příklady použití:

   ```
   > (define p1 (my-cons 3 8))
   > (define p2 (switch p1))
   > (my-car p2)
   8
   > (my-cdr p2)
   3
   ```

3. **(2 body)** Implementujte vlastní aritmetiku komplexních čísel. Komplexní číslo $a+ib$ bude reprezentováno tečkovým párem `(a . b)`. Konstruktor `make-c` vytvoří pár reprezentující komplexní číslo, selektory

`real` a `imag` vratí reálnou a imaginární část, procedura `conj` vrátí komplexně sdružené číslo, a procedury `c+`, `c-`, `c*`, `c/` vypočítají součet, rozdíl, součin a podíl dvou komplexních čísel.

Příklady použití:

```
> (define c1 (make-c 1 2))
> (define c2 (make-c 3 -1))
> c1
(1 . 2)

> (real c1)
1

> (imag c1)
2

> (conj c1)
(1 . -2)

> (c+ c1 c2)
(4 . 1)

> (c* c1 c2)
(5 . 5)
```

4. **(1 bod)** Vytvořte proceduru `singletons` očekávající jako argument seznam a vracející seznam singletonů (tzn. jednoprvkových seznamů) tvořených prvky vstupního seznamu.

   Příklady použití:

   ```
   > (singletons '(2))
   ((2))

   > (singletons '(2 -1 3))
   ((2) (-1) (3))
   ```

5. **(3 body)** Vytvořte proceduru `roots-of-unity` s jedním argumentem

◁n▷, která vrátí seznam délky n obsahující všechny kořeny rovnice

$$x^n = 1.$$

Pro výpočet k-tého kořenu, kde $k = 0, 1, \ldots, n-1$, využijte známý vztah

$$x_k = \cos\frac{2\pi k}{n} + i \cdot \sin\frac{2\pi k}{n},$$

kde i je imaginární jednotka. Jak víme, kořenem může být komplexní číslo. Každý kořen upravte pomocí následující aplikace `(rationalize (inexact->exact x) 1/10)`, kde `x` je reálná nebo imaginární část kořene (viz str. 30 výukového textu).

Příklady použití:

```
> (roots-of-unity 2)
(1 -1)

> (roots-of-unity 6)
(1 1/2+4/5i -1/2+4/5i -1 -1/2-4/5i 1/2-4/5i)
```

6. **(3 body)** Vytvořte proceduru `div-list` s jedním argumentem ◁n▷, která vrátí seznam délky n obsahující pravdivostní hodnoty `#t` a `#f` podle toho, jestli číslo dané pozicí v seznamu dělí číslo n.

   Příklady použití:

```
> (div-list 2)
(#t #t)

> (div-list 3)
(#t #f #t)

> (div-list 4)
(#t #t #f #t)

> (div-list 5)
(#t #f #f #f #t)

> (div-list 12)
(#t #t #t #t #f #t #f #f #f #f #f #t)
```

7. **(4 body)** Naprogramujte proceduru `make-palindrom`, která vytváří z prvků vstupního seznamu palindrom. Je zakázáno použít procedury `reverse` a `append`.

   Příklady použití:

   ```
   (make-palindrom '(a n))
   > (a n n a)

   (make-palindrom '(r o t))
   > (r o t o r)

   (make-palindrom '(d e n n i s))
   > (d e n n i s s i n n e d)

   (make-palindrom '(n e p o t))
   > (n e p o t o p e n)
   ```

8. **(5 bodů)** Naprogramujte proceduru `map-index-pred` se třemi argumenty: predikátem ◁*pred*?▷, procedurou ◁*f*▷ a seznamem ◁*l*▷ = (*a*$_1$ *a*$_2$ ...*a*$_n$). Procedura `map-index-pred` vrací seznam (*b*$_1$ *b*$_2$ ...*b*$_n$), jehož prvky b_i jsou $f(a_i)$ pro indexy splňující predikát ◁*pred*?▷ a a_i pro indexy nesplňující predikát ◁*pred*?▷. Např:

   ```
   > (map-index-pred odd? sqr '(2 3 4 5))
   (2 9 4 25)
   ```

   Prvky seznamu na lichých indexech (tedy prvky 3 a 5) jsou ve výsledném seznamu umocněny. Protože indexy prvků 2 a 4 jsou 0 a 2 (nejsou tedy liché), jsou tyto prvky ve výsledném seznamu stejné jako v původním seznamu. Další příklad:

   ```
   > (map-index-pred (lambda(i) (< i 2)) - '(1 2 3 4 5))
   (-1 -2 3 4 5)
   ```

   Prvky na indexech menší než 2 jsou ve výsledném seznamu nahrazeny jejich opačnou hodnotou, ostatní zůstavají stejné.

9. **(4 body)** Naprogramujte proceduru `swap-index` se třemi argumenty: dva reprezentují indexy a jeden seznam. Procedura vrací seznam, ve kterém jsou prvky na uvedených indexech zaměněny.

   Příklady použití:

```
(swap-index 0 3 '(1 2 3 4 5))
> (4 2 3 1 5)

(swap-index 1 3 '(1 2 3 4 5))
> (1 4 3 2 5)
```