Lista zadań nr 8

Zadanie 1.

Zdefiniuj procedurę `mreverse!`, która odwraca listę mutowalną „w miejscu”, czyli nie tworzy nowych bloczków `mcons`-em, a odpowiednio przepina wskaźniki.

Zadanie 2. (2 pkt)

Wzorując się na implementacji kolejek z wykładu zaimplementuj kolejki dwukierunkowe, czyli takie w których można wstawiać i usuwać element zarówno z jednej jak i z drugiej strony kolejki. Do implementacji kolejek dwukierunkowych użyj list dwukierunkowych, czyli takich w których każdy węzeł ma wskaźnik na następny i poprzedni węzeł listy.

Twoja implementacja powinna znajdować się w osobnym module, a eksportowane procedury powinny mieć odpowiednie kontrakty.

Zadanie 3. (2 pkt)

Podziel moduł `parser.rkt` z wykładu na dwa moduły:

- `parsing.rkt` — zawierający ogólne definicje dotyczące parsowania,
- `expparser.rkt` — zawierający konkretne zastosowanie modułu `parsing.rkt` do parsowania wyrażeń arytmetycznych.

Zwróć uwagę na to, które definicje powinny zostać wyeksportowane, a które powinny zostać prywatne dla modułu. W module `parsing.rkt` zadbaj o odpowiednie kontrakty dla eksportowanych procedur.

Zadanie 4.

Zmodyfikuj parser wyrażeń arytmetycznych z wykładu tak, by nie konstruował *abstrakcyjnego drzewa rozbioru* (drzewa typu `Exp`), ale od razu obliczał podane wyrażenie do liczby. Zauważ, że wystarczy zmodyfikować tylko akcje semantyczne podane w poszczególnych regułach.

Zadanie 5. (2 pkt)

Parser pokazany na wykładzie nie jest jeszcze idealny, ale poprawimy go na następnym wykładzie. Jednak jedną z jego wad możemy poprawić teraz. W opisie gramatyki obecnie znajduje się następująca definicja wyrażeń.

```
("expression"
 (("simple-expr" "operator" "simple-expr")
      ,(lambda (e1 op e2) (exp-op op e1 e2)))
 (("simple-expr") ,(lambda (e) e)))
```

Zauważ, że jeśli wyrażenie jest wyrażeniem prostym, to parser zacznie od pierwszej reguły, gdzie sparsuje podane wyrażenie proste, ale zawiedzie gdy nie uda mu się znaleźć operatora. Wtedy przejdzie do drugiej reguły, której wykonanie się powiedzie. W efekcie, parsowane wyrażenie będzie dwukrotnie odwiedzane przez parser. Jeśli problem będzie się powtarzał z każdym wywołaniem rekurencyjnym, to możemy się zderzyć z wykładniczym spowolnieniem programu.

```
> (run-exp-parser '((((((((((((((((((((((((42))))))))))))))))))))))))
```

Można temu problemowi zaradzić modyfikując gramatykę wyrażeń: wyrażenie to wyrażenie proste po którym opcjonalnie występuje operator i kolejne wyrażenie proste, co w naszym języku opisu gramatyk można zapisać następująco:

```
("expression"
 (("simple-expr" "expr-rest") ,(...)))

("expr-rest"
 (("operator" "simple-expr") ,(...))
 (()                         ,(...)))
```

Uzupełnij akcje semantyczne tak, by zmodyfikowany parser dalej poprawnie konstruował wyrażenia. Pamiętaj, że wartością funkcji może też być funkcja.

Zadanie 6. (2 pkt)

Napisz funkcję tłumaczącą napis złożony z liter, cyfr, białych znaków i wybranych znaków interpunkcyjnych na kod Morse'a. Kropkę koduj jako znak kropki (`#\.`), kreskę jako znak podkreślenia (`#\_`), odstęp między znakami jako pojedynczą spację, zaś dowolny ciąg białych znaków jako dwie spacje. Oto przykładowe wywołanie takiej procedury:

```
> (morse-code "Metody Programowania")
"__ . _ ___ _.. _.__  .__. .._. ___ __. ._. ._ __ ___ .__ ._ _. .. ._"
```

Przydatne mogą okazać się procedury `string->list`, `list->string` oraz `char-whitespace?`. Postaraj się zredukować złożoność kodu poprzez zakodowanie części algorytmu za pomocą danych.

Zadanie 7. (2 pkt)

Napisz funkcję tłumaczącą kod Morse'a na zwykły napis, np.

```
> (morse-decode "__ .__.  ..___ _____ ..___ ..___")
"MP 2022"
```

Również postaraj się zakodować część algorytmu za pomocą odpowiednich danych.