Lista zagadnień nr 8

Przed zajęciami

Ćwiczenie 1.

Zdefiniuj leniwą listę:

1. kolejnych liczb Fibonacciego,
2. zawierającą wszystkie liczby całkowite.

Na zajęciach

UWAGA: Zadania do rozwiązania podczas zajęć znów wymagają napisania większej ilości kodu niż zadania z pierwszych list. Więc być może znów dobrym pomysłem jest rozwiązywanie ich (w grupach) przy komputerach? Proszę pamiętać, że celem tych i poprzednich ćwiczeń jest głównie opanowanie struktury i działania interpretera (dla coraz bogatszego języka), więc jeśli ktoś nie czuje się zbyt pewnie w temacie, może też wrócić do zadań z poprzedniej listy.

Ćwiczenie 2.

Napisz w języku interpretowanym procedury `append`, `map` i `reverse`. Spróbuj zrobić to też dla leniwych list.

Ćwiczenie 3.

By zdefiniować w Rackecie wieloargumentową lambdę, wystarczy nie pisać nawiasów wokół zmiennej, np.

```
(lambda x (+ (second x) (third x)))
```

W ciele tak zdefiniowanej procedury zmiennej `x` przyporządkowana jest **lista** wszystkich argumentów. Tak więc powyższa procedura dodaje do siebie wartości swojego drugiego i trzeciego argumentu, ignorując wszystkie inne argumenty, np.

```
((lambda x (+ (second x) (third x))) 1 2 3 4 5 6)
```

oblicza się do wartości 5. Dodaj do języka interpretowanego takie wieloargumentowe lambdy.

Ćwiczenie 4.

Chyba wszyscy się zgadzają, że „car" i „cdr" to nie są nazwy, które łatwo skojarzyć z tym, co robią te procedury. Nazwy te były używane w pierwszym LISP-ie i jakoś tak zostało. Mają jednak pewną zaletę: można budować z nich zwięzłe wyrażenia, dające dostęp do elementów bardziej złożonych struktur. Na przykład procedura `caddar` jest zdefiniowana jako:

```
(define (caddar a)
  (car (cdr (cdr (car a)))))
```

Jest więc to procedura, której wartością jest głowa ogona ogona głowy listy, np.

```
> (caddar '((1 2 3 4) (5 6 7)))
3
```

Jest to wygodne, ale Racket ma pod tym względem pewne ograniczenie: każda z tych procedur jest zdefiniowana osobno, więc jest ich jedynie skończenie wiele – a dokładniej, zdefiniowane są wszystkie, które mają nie więcej niż cztery litery pomiędzy `c` i `r` (ile ich jest w sumie?). Tak więc:

```
> (caaddar '((1 2 (3 4) 5) (6 7)))
ERROR! caaddar: undefined;
```

My w naszym interpretowanym języku nie popełnimy podobnego błędu! Dodaj do składni i interpretera obsługę formy specjalnej `cxxxr`, gdzie `xxx` jest dowolnie długim ciągiem składającym się z liter `a` i `d`. Do rozwiązania tego zadania przydać się może następująca procedura, która zmienia symbol na listę jednoliterowych symboli (kod dostępny jest też na SKOSie):

```
(define (split-symbol s)
  (define (char->symbol a)
    (string->symbol (list->string (list a))))
  (map char->symbol (string->list (symbol->string s))))
```

Na przykład:

```
> (split-symbol 'caddar)
(list 'c 'a 'd 'd 'a 'r)
```

Wersja dla odważnych: W Rackecie są procedury `first`, `second` i tak aż do `tenth`. A gdzie `eleventh`? Gdzie `twelfth`?! Spraw, by w naszym interpretowanym języku działało przynajmniej do `millionth`.

Ćwiczenie 5.

Rozszerz język interpretowany o wzajemnie rekurencyjne formy `define` na najbardziej zewnętrznym poziomie. Program w takim języku to lista, która najpierw zawiera definicje, a potem jedno wyrażenie, które jest wartością całego programu. Formalnie, program zadany jest przez następujący predykat:

```
(define (define? t)
  (and (tagged-tuple? 'define 3 t)
       (list? (second t))
       (andmap symbol? (second t))
       (> (length (second t)) 0)))

(define (program? t)
  (or (and (list? t)
           (= (length t) 1)
           (expr? (car t)))
      (and (pair? t)
           (define? (car t))
           (program? (cdr t)))))
```

Opisz, jak zmienić strukturę interpretera. Czy da się to zrobić nie modyfikując procedury `eval-env`?