# Lista zagadnień nr 8

## Zadania na ćwiczenia

### Wartości boolowskie

#### Ćwiczenie 1.

Fakt, że reprezentujemy zarówno *stałą* `true` jak i *wartość* `true` w postaci racketowego `true` (czy też `#t`) może wywołać trochę zamieszania. Być może warto więc odróżnić te trzy sposoby istnienia prawdy i fałszu. Zmień implementację w pliku `boolean.rkt` tak, by:

- W składni abstrakcyjnej prawda była reprezentowana jako `(const 'true)`, a nie jak dotychczas `(const true)` (upewnij się, że rozumiesz różnicę). Analogicznie z fałszem.
- Do reprezentowania wartości będącej wynikiem interpretacji, użyj liczb: niech `0` reprezentuje fałsz, a `1` reprezentuje prawdę.

Przetestuj nowe rozwiązanie, żeby upewnić się, że dostosował[e/a]ś wszystkie elementy systemu do nowej reprezentacji. Jakie widzisz wady i zalety takiego rozwiązania?

#### Ćwiczenie 2.

Dodaj do języka konstrukcje (w Rackecie nazwalibyśmy je „formami specjalnymi") `boolean?` i `number?`, które pozwalają sprawdzić czy dana wartość jest liczbą czy boolem. Zwróć uwagę, że w Rackecie te dwie konstrukcje to procedury. Ty też możesz uczynić je procedurami, a nie odrębnymi konstrukcjami.

#### Ćwiczenie 3.

Nasze operatory `and` i `or` są gorliwe, bo nasz interpreter zawsze oblicza oba argumenty, jak dla *każdego* operatora binarnego (zastanów się, czy na pewno rozumiesz dlaczego tak się dzieje). Dodaj do języka `and` i `or` jako formy specjalne,

które będą odpowiednio leniwe w stylu Racketa. Czy musisz zmieniać składnię abstrakcyjną? Czy chcesz zmieniać składnię abstrakcyjną?

## Pary

### Ćwiczenie 4.

Dodaj do naszego języka z pliku `pair.rkt` formę specjalną `pair?` z semantyką taką jak w Rackecie.

### Ćwiczenie 5.

Rozszerz język o konstrukcję `unop` dla operatorów unarnych (jednoargumentowych). Rozszerz ewaluator o operator jednoargumentowy `not` i spraw, by `car` i `cdr` nie były formami specjalnymi a operatorami unarnymi. Jakie widzisz zalety i wady takiego rozwiązania (w szczególności w kontekście funkcji wyższego rzędu)? Jak napisać procedurę `parse`, żeby ewaluator nie pomylił wywołania funkcji z operatorem unarnym?

### Ćwiczenie 6.

Jedną z nielicznych zalet nazw `car` i `cdr` w Rackecie jest wygodna składnia służąca do składania tych procedur. Można więc napisać np. `cadar` żeby uzyskać wartość pierwszego elementu drugiego elementu pierwszego elementu jakieś pary. Wadą (?) racketowej implementacji jest to, że konstrukcje `c[a/d]+r` to zwykłe procedury zdefiniowane w bibliotece standardowej, więc jest ich skończona liczba. Możemy więc użyć procedury `cadaar`, ale procedury `cadaadr` już w bibliotece nie znajdziemy.

Rozbuduj nasz język o formy specjalne postaci `c[a/d]+r` dowolnej długości, poprawiając w ten sposób racktowe podejście. Prawdopodobnie przydadzą Ci się racketowe procedury `symbol->string` i `string->list`. Pojedyncze znaki możemy porównywać procedurą `eq?`, np.

```
> (symbol->string 'cadaadadr)
"cadaadadr"
> (string->list (symbol->string 'cadaadadr))
'(#\c #\a #\d #\a #\a #\d #\a #\d #\r)
> (eq? #\c #\a)
#f
> (eq? #\c #\c)
#t
```

## Listy

### Ćwiczenie 7.

Do języka w pliku `list.rkt` dodaj lukier syntaktyczny `list` z semantyką taką jak w Rackecie. Oznacza to, że procedura `parse` powinna zmienić wyrażenie

```
(list e1 e2 e3 e4)
```

na składnię abstrakcyjną odpowiadającą wyrażeniu

```
(cons e1 (cons e2 (cons e3 (cons e4 null))))
```

### Ćwiczenie 8.

Rozważmy trochę trudniejszą wersję Zadania 6. Racket oferuje wygodne procedury do wyciągania z listy kolejnych elementów: `first`, `second`, `third` i tak dalej aż do... dziesięciu. To oznacza, że w bibliotece standardowej nie znajdą Państwo procedury `eleventh`. Nasz język nie będzie miał tej słabości. Dodaj do niego konstrukcje `first`, `second` i tak dalej aż do 999 miliardów, łącząc pojedyncze słowa liczebników dywizem, np.

```
(nine-hundred-ninety-nine-billion-nine-hundred-ninety-nine-million-nine-
hundred-ninety-nine-thousand-nine-hundred-ninety-ninth xs)
```

Powyższy program powinien obliczyć 999999999999-ty element listy `xs`.

## Funkcje i domknięcia

### Ćwiczenie 9.

Zdefiniuj w naszym języku funkcję `not`.

### Ćwiczenie 10.

Zdefiniuj w nasyzm języku funkcje `curry` i `uncurry`.

### Ćwiczenie 11.

Dodaj do naszego języka konstrukcję `let-lazy`, która działa podobnie do zwykłego let-wyrażenia, ale wartość liczona jest leniwie, czyli dopiero w momencie, gdy jest potrzebna. Proszę pamiętać o statycznym wiązaniu zmiennych! Wynik nie musi być spamiętany, więc w programie

```
(let-lazy [x (+ 3 5)]
  (+ x x))
```

dodawanie `3` do `5` może być wykonane dwa razy. Wartość zmiennej powinna być obliczana przy każdym użyciu zmiennej, w szczególności aplikacja funkcji do argumentu powinna być gorliwa, np. program

```
(let-lazy [x (+ 3 5)]
  ((lambda (a) 1) x)
```

powinien obliczyć się do wartości `1` a wartość zmiennej `x` powinna być obliczona w momencie aplikacji.

**Ćwiczenie 12.**

Dodaj so naszego języka z pliku `fun.rkt` (tego z funkcjami) strumienie. Strumień możemy utworzyć formą specjalną `unfold`, która produkuje kolejne elementy nieskończonego strumienia wartości przy uzyciu ziarna (ang. *seed*), np.

```
(unfold seed step)
```

gdzie `seed` to początkowa wartość ziarna a `step` to funkcja, która jako argument przyjmuje aktualne ziarno, a zwraca parę zawierającą kolejno: wartość elementu w strumieniu i nowe ziarno. Zdefiniuj formy specjalne (a może operatory unarne? A może funkcje?):

- `scar` – zwraca pierwszy element strumienia
- `scdr` – zwraca ogon strumienia

Na przykład wyrażenie

```
(unfold 0 (lambda (x) (cons (* x x) (+ 1 x))))
```

tworzy strumień kwadratów kolejnych liczb naturalnych, np.

```
> (eval (parse
    '(scar (unfold 0 (lambda (x) (cons (* x x) (+ 1 x)))))))
0
> (eval (parse
    '(scar (scdr (unfold 0 (lambda (x) (cons (* x x) (+ 1 x))))))))
1
> (eval (parse
    '(scar (scdr (scdr (unfold 0 (lambda (x) (cons (* x x) (+ 1
        x)))))))))
4
> (eval (parse
    '(scar (scdr (scdr (scdr (unfold 0 (lambda (x) (cons (* x x) (+ 1
        x))))))))))
9
> (eval (parse
```

```
    '(scar (scdr (scdr (scdr (scdr (unfold 0 (lambda (x) (cons (* x x)
        (+ 1 x))))))))))
16
```

# Zadania domowe

## Zadanie 14.

Dodaj do języka z pliku `fun.rkt` formę dwuargumentową formę specjalną (albo, wedle uznania, operator binarny) `apply`, który aplikuje funkcję do listy argumentów, np. (wykorzystując składnię dodaną w Zad. 7):

```
> (eval (parse
    '(apply (lambda (x y) (+ x y))
            (list 1 2))))
3
> (eval (parse
    '(apply (lambda (x y z) (+ x (+ y z)))
            (list 1 2))))
#<clo>
> (eval (parse
    '(apply (lambda (x y) (lambda (z) (+ x (+ y z))))
            (list 1 2 3))))
6
> (eval (parse
    '(apply (lambda (x y) (+ x y))
            (list 1 2 3))))
ERROR
```

## Zadanie 15.

Zmodyfikuj język z pliku `fun.rkt` tak, by w naszym języku argumenty funkcji i formy specjalnej cons były liczone leniwie. Np.

```
> (eval (parse
    '((lambda (x) (+ 3 3)) (/ 5 0))))
6
> (eval (parse
    '(let [if-fun (lambda (b t e) (if b t e))]
       (if-fun true 4 (/ 5 0)))))
4
> (eval (parse
    '(car (cons (+ 2 2) (/ 5 0)))))
4
```

*Wskazówka:* W pliku `fun.rkt` explicite liczymy wartość arguementu. Zamiast tego, można utworzyć odroczone obliczenie tak jak w rozwiązaniu Zadania 11.