Kolokwium nr 2

Kolokwium rozpoczyna się o godz. **16:00** i trwa **150 minut**. Rozwiązanie każdego z zadań powinno być do godz. **18:30** przesłane w **oddzielnym** pliku **tekstowym** o nazwie **zadanie_<*nr-zadania*>.rkt**, z wykorzystaniem przeznaczonej dla danego zadania aktywności w SKOSie.[1]

Przesłane rozwiązania powinny być wynikiem **całkowicie samodzielnej pracy**. Przy opracowywaniu rozwiązań dozwolone jest korzystanie z własnych notatek, materiałów z wykładu i ćwiczeń, a także z podręcznika.

Język LetF

W pierwszych dwóch zadaniach zajmiemy się językiem LetF — czyli językiem wyrażeń arytmetycznych i logicznych z let-wyrażeniami z wykładu rozszerzonym o funkcje deklarowane na początku programu (podobnie jak w C czy Pascalu). Oznacza to, że nasze programy będą składały się z ciągu procedur i wyrażenia, którego wartość jest wartością całego programu. Definicja procedury składać się będzie z nazwy, listy parametrów (zmiennych, które są związane w jej ciele) i ciała, które jest wyrażeniem. Do składni wyrażeń dodajemy również wywołanie funkcji, składające się z nazwy funkcji i listy argumentów, które są, oczywiście, wyrażeniami. Fragment składni abstrakcyjnej zawierający te rozszerzenia przedstawiony jest poniżej:

```
(define (expr? e)
  (match e
    ...
    [(call f es)
     (and (name-fun? f)
          (andmap expr? es))]
    [_ false]))
```

```
(define (function? p)
  (match p
    [(proc f xs e)
     (and (name-fun? f)
          (andmap name-var? xs)
          (expr? e))]
    [_ false]))
```

```
(define (program? p)
  (match p
    [(prog ps e)
     (and (andmap function? ps)
          (expr? e))]
    [_ false]))
```

Żeby móc w naszym języku pisać interesujące programy, zakładamy że wszystkie definicje funkcji mogą być (wzajemnie) rekurencyjne, to znaczy traktujemy nazwy procedur zdefiniowanych w programie jako wiążące wszystkie wystąpienia, zarówno w ciałach procedur (tak własnym jak i pozostałych) jak i w ciele programu. Poniżej przedstawiamy przykładowe programy — składnia konkretna jest naturalnym rozszerzeniem tej którą stosowaliśmy w języku z wykładu:

```
(((fun Fact (n)
    (if (= n 0) 1 (* n (Fact (- n 1))))))
 (Fact 5))
```

```
(((fun Odd  (n) (if (= n 0) false (Even (- n 1))))
  (fun Even (n) (if (= n 0) true  (Odd  (- n 1)))))
 (Odd 33))
```

Zauważmy, że dla odróżnienia nazw zmiennych od nazw funkcji przyjmujemy, że nazwy zmiennych muszą spełniać predykat `name-var?` (a więc zaczynać się małą literą), a nazwy funkcji — `name-fun?` (a więc zaczynać się wielką literą). Zarówno definicja składni abstrakcyjnej, jak i powyższe przykłady znajdują się w dołączonym szablonie.

Zadanie 1. (5 pkt.)

Uzupełnij definicję parsera języka LetF podaną w szablonie. W tym celu uzupełnij definicje procedur `parse-expr`, `parse-fun` i `parse-prog`.

Zadanie 2.

Wersja pełna (10 pkt.). Uzupełnij definicję interpretera języka LetF podaną w szablonie o obsługę funkcji. Zauważ, że procedura `eval-expr` ma dwa środowiska: jedno dla nazw zmiennych (ρ), drugie dla nazw funkcji (ϑ) — ostatecznie w ciele programu mogą wolno występować zarówno zmienne, jak i wywołania funkcji! Częścią zadania jest zdecydowanie *co* powinno być przechowywane w środowisku ϑ: nie komplikuj implementacji bez potrzeby. Pamiętaj też, że (podobnie jak w C czy Pascalu) funkcje *nie są* wartościami!

[1] Zadania 1–2 mają wspólny szablon. Nie trzeba wycinać części niezwiązanych z zadaniem, ale należy zaznaczyć części kodu będące rozwiązaniami poszczególnych zadań.

Wersja *light* (5 pkt.). Jak wyżej, ale zakładamy że ciała funkcji mogą używać tylko funkcji zdefiniowanych wcześniej (w szczególności funkcje *nie mogą* być rekurencyjne). Oznacza to, że nazwa funkcji jest związana dopiero w następujących *po niej* definicjach i w ciele całego programu. Uwaga: wzorcowe rozwiązanie tej wersji nie musi być mniej skomplikowane niż wersji pełnej, ale może okazać się prostsze koncepcyjnie.

Zadanie 3. (5 pkt.)

Zaimplementuj w języku Racket procedury spełniające następujące kontrakty:

```
(parametric->/c [a b] (-> (-> a b) a b))
(parametric->/c [a b c] (-> (-> a b c) (-> (cons/c a b) c)))
(parametric->/c [a b] (-> (listof (-> a b)) (listof a) (listof b)))
(parametric->/c [a b] (-> (-> b (or/c false/c (cons/c a b))) b (listof a)))
(parametric->/c [a] (-> (-> a boolean?) (listof a) (cons/c (listof a) (listof a))))
```

Kontrakt `false/c` jest spełniony wyłącznie przez wartość `#f`.

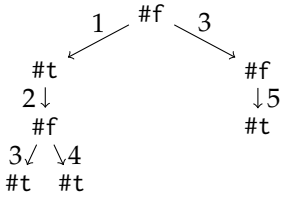
W sytuacji, jeśli istnieje wiele możliwych procedur spełniających dany kontrakt, wybierz taką, która jest w jakiś sposób użyteczna. Napisane procedury powinny się nie pętlić i nie zgłaszać wyjątków.

Zadanie 4. (5 pkt.)

Rozważmy następujący typ w języku Plait:

```
(define-type (PrefixTree 'a)
    (node [exists : Boolean] [children : (Listof ('a * (PrefixTree 'a)))]))
```

Typ ten opisuje tak zwane drzewa prefiksowe. Drzewo prefiksowe jest strukturą danych służącą do przechowywania ciągów elementów. W tej strukturze krawędzie są etykietowane elementami. Każdy przechowywany ciąg tworzy ścieżkę od korzenia drzewa do pewnego wierzchołka `t`, dla którego `(node-exists t)` jest wartością `#t`. Przykładowo, poniższe drzewo zawiera ciągi `(1)`, `(1 2 3)`, `(1 2 4)` oraz `(3 5)`:



Powyższe drzewo można zakodować używając typu `PrefixTree` w następujący sposób:

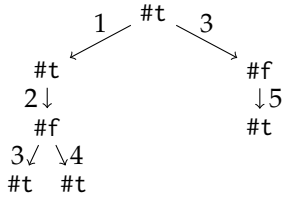
```
(define example-tree
  (node #f (list
    (pair 1 (node #t (list
      (pair 2 (node #f (list
        (pair 3 (node #t empty))
        (pair 4 (node #t empty))))))))
    (pair 3 (node #f (list
      (pair 5 (node #t empty))))))))
```

Zaimplementuj następujące procedury operujące na drzewach prefiksowych:

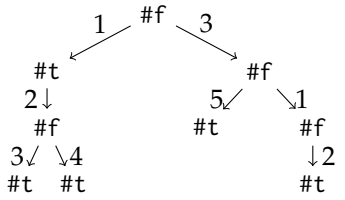
- `lookup` typu `((Listof 'a) (PrefixTree 'a) -> Boolean)` – sprawdzającą, czy ciąg jest zawarty w drzewie,
- `insert` typu `((Listof 'a) (PrefixTree 'a) -> (PrefixTree 'a))` – dodającą ciąg do drzewa,
- `remove` typu `((Listof 'a) (PrefixTree 'a) -> (PrefixTree 'a))` – usuwającą ciąg z drzewa (nie jest wymagane „obcinanie" zbędnych gałęzi).

Poniższe przykłady ilustrują oczekiwany sposób działania tych procedur.

`(insert '() example-tree)`



`(insert '(3 1 2) example-tree)`



`(remove '(3 5) example-tree)`