Kolokwium poprawkowe

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.

Do zaliczenia kolokwium poprawkowego wymagane jest uzyskanie **co najmniej 15 punktów (50%)**.

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.

Zadanie 1. (10 pkt.)

Rozważmy następującą definicję drzew ternarnych z etykietami w liściach:

```
;; Konstruktory:

(define (node t1 t2 t3)
  (list 'node t1 t2 t3))

(define (leaf n)
  (list 'leaf n))

;; Selektory:

(define node-t1 second)
(define node-t2 third)
(define node-t3 fourth)

(define leaf-val second)
```

```
;; Predykaty:

(define (tertree? t)
  (or (and (list? t)
           (= (length t) 2)
           (eq? (first t) 'leaf)
           (number? (second t)))
      (and (list? t)
           (= (length t) 4)
           (eq? (first t) 'node)
           (andmap tertree? (cdr t)))))

(define (node? t)
  (and (pair? t)
       (eq? (car t) 'node)))

(define (leaf? t)
  (and (pair? t)
       (eq? (car t) 'leaf)))
```

Dana jest następująca procedura operująca na drzewach ternarnych:

```
(define (sum t)
  (if (leaf? t)
      (leaf-val t)
      (+ (sum (node-t1 t))
         (sum (node-t2 t))
         (sum (node-t3 t)))))
```

W tym zadaniu:

- Zdefiniuj procedurę `termap`, która aplikuje daną procedurę do wszystkich etykiet, np.

  `(termap (lambda (x) (* 2 x)) (node (leaf 1) (leaf 2) (leaf 3)))` $\equiv$ `(node (leaf 2) (leaf 4) (leaf 6))`

- Sformułuj twierdzenie o indukcji dla typu danych `tertree`.
- Użyj tego twierdzenia, żeby udowodnić, że dla każdego drzewa ternarnego `t` zachodzi:

  `(sum (termap (lambda (x) (* 2 x)) t))` $\equiv$ `(* 2 (sum t))`

Pamiętaj, żeby:

- Sformułować twierdznie możliwie precyzyjne (w szczególności nie powinno ono zawierać zmiennych wolnych),
- Wypisać założenia indukcyjne,
- Jeśli dana równoważność między programami nie wynika z modelu obliczeń (a wynika np. z założenia indukcyjnego lub własności arytmetycznych), należy to zaznaczyć w dowodzie.

Zadanie 2. (10 pkt.)

Rozważmy język FunCore, będący wariantem prostego języka funkcyjnego z wykładu. Wyrażeniami w naszym języku będą stałe liczbowe i boole'owskie, zmienne, wyrażenia warunkowe `if`, funkcje nienazwane `lambda` (podobnie jak na wykładzie wszystkie funkcje są jednoargumentowe) i aplikacje (operacje arytmetyczne traktujemy jako procedury wbudowane). Wartościami naszego języka są oczywiście funkcje, liczby i stałe boole'owskie — od języka z wykładu różni się on przede wszystkim brakiem let-wyrażeń.

Let-wyrażenia są jednak bardzo wygodne, wprowadzimy więc również język Fun*, w którym oprócz powyższych konstrukcji mamy do dyspozycji operację `let*`. Predykaty `expr?` (opisujący składnię abstrakcyjną wyrażeń języka FunCore) i `expr*?` (dla języka Fun*) zamieszczone są poniżej:

```
(define (expr? e)
  (match e
    [(var-expr x) (name? x)]
    [(const v)    (or (number? v) (boolean? v))]
    [(fun x e)    (and (name? x) (expr? e))]
    [(if-expr eb et ef)
     (and (expr? eb) (expr? et) (expr? ef))]
    [(app ef ea)  (and (expr? ef) (expr? ea))]
    [_           false]))
```

```
(define (expr*? e)
  (match e
    [(var-expr x) (name? x)]
    [(const v)    (or (number? v) (boolean? v))]
    [(fun x e)    (and (name? x) (expr*? e))]
    [(if-expr eb et ef)
     (and (expr*? eb) (expr*? et) (expr*? ef))]
    [(app ef ea)  (and (expr*? ef) (expr*? ea))]
    [(let*-expr bds e)
     (and (andmap binding? bds) (expr*? e))]
    [_           false]))

(define (binding? bd)
  (match bd
    [(list x e)  (and (name? x) (expr*? e))]
    [_           false]))
```

Oczywiście, podobnie jak w Rackecie, zakładamy że pierwsza nazwa na liście wiąże zarówno w ciele całego wyrażenia `let*` jak i w pozostałych wyrażeniach na liście, i tak dalej.

Zarówno powyższe predykaty, jak i używane przez nie struktury dostarczone są w szablonie do zadania. Szablon zawiera również parser dla języka Fun* i interpreter dla języka FunCore. **Waszym zadaniem** jest zaimplementowanie jednoargumentowej procedury `deletify` przyjmującej poprawne wyrażenie w języku Fun* i zwracającej odpowiadające mu wyrażenie w języku FunCore, zgodnie z poznaną w pierwszych tygodniach wykładu interpretacją wyrażeń `let*` jako cukru syntaktycznego dla odpowiedniej kombinacji definicji i wywołań funkcji.

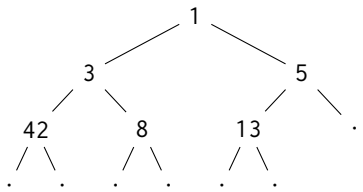
Wersja light (5 p.) Jak wyżej, ale możesz założyć że listy w wyrażeniach `let*` mają zawsze długość 1 (tj. są równoważne wyrażeniom `let` z języka z wykładu).

Zadanie 3. (10 pkt.)

Rozważmy następujący typ w języku Plait:

```
(define-type SkewHeap
  (empty)
  (node [v : Number] [l : SkewHeap] [r : SkewHeap]))
```

Typ ten reprezentuje *kopce skośne* – pewną drzewiastą strukturę danych, umożliwiającą efektywną implementację trzech operacji: dodania elementu, wyjęcia najmniejszego elementu oraz scalenia dwóch kopców. Struktura ta posiada *własność kopca*, która polega na tym, że wartość przechowywana w węźle kopca jest mniejsza lub równa wartościom przechowywanym w jego poddrzewach. Poniższy rysunek przedstawia przykładowy kopiec:



Jego reprezentacja w języku Plait jest następująca:

```
(define example-heap
  (node 1
```

```
(node 3
      (node 42 (empty) (empty))
      (node 8 (empty) (empty)))
(node 5
      (node 13 (empty) (empty))
      (empty))))
```

Najważniejszą operacją dla kopców skośnych jest operacja scalania dwóch kopców. Implementuje się ją następująco:

- Scalenie kopca pustego z dowolnym kopcem zwraca ten kopiec.
- Przy scaleniu dwóch niepustych kopców istotny jest porządek wartości w korzeniach obu kopców. Nazwijmy **pierwszym** kopcem ten, którego wartość w korzeniu jest mniejsza lub równa wartości w korzeniu **drugiego** kopca. Wtedy wynikiem scalenia tych dwóch kopców jest węzeł, którego:
  - wartość – jest wartością w korzeniu **pierwszego** kopca,
  - lewe poddrzewo – jest wynikiem scalenia **drugiego** kopca z **prawym** poddrzewem korzenia **pierwszego** kopca,
  - prawe poddrzewo – jest **lewym** poddrzewem korzenia **pierwszego** kopca.

Zaimplementuj następujące procedury operujące na kopcach skośnych:

- `singleton` typu `(Number -> SkewHeap)` – zwracającą kopiec jednoelementowy,
- `union` typu `(SkewHeap SkewHeap -> SkewHeap)` – scalającą dwa kopce zgodnie z algorytmem opisanym powyżej,
- `insert` typu `(Number SkewHeap -> SkewHeap)` – wstawiającą element do kopca przez scalenie tego kopca z kopcem jednoelementowym,
- `remove` typu `(SkewHeap -> SkewHeap)` – tworzącą kopiec pomniejszony o najmniejszy element przez scalenie lewego i prawego poddrzewa korzenia.