Kolokwium nr 1

Kolokwium rozpoczyna się o godz. **16:00** i trwa **120 minut**. Rozwiązanie każdego z zadań powinno być do godz. **18:00** przesłane w **oddzielnym** pliku **tekstowym** o nazwie **zadanie_<*nr-zadania*>.rkt**, z wykorzystaniem przeznaczonej dla danego zadania aktywności w SKOSie.

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.

Uwaga! Pamiętaj, że w zadaniach o dowodzeniu równoważności programów nie interesuje nas tzw. „machanie rękami", czyli jedynie intuicyjne zrozumienie, co program robi. Interesują nas w miarę formalne dowody indukcyjne. Pamiętaj też, że jeśli jakaś równoważność nie wynika bezpośrednio z ewaluacji (a np. z naszej znajomości praw arytmetyki albo założenia indukcyjnego), należy zaznaczyć to w dowodzie. I wreszcie, że częściowy dowód indukcyjny (np. sformułowanie odpowiedniego lematu, ale pozostawienie go bez dowodu) też jest coś warty.

Zadanie 1. (5 pkt.)

Zdefiniuj poniższe procedury. Nie używaj bezpośrednio rekursji – zamiast tego możesz użyć procedur `append`, `map`, `filter` i `foldr` z biblioteki standardowej Racketa.

- `(define (kwadraty-liczb-parzystych xs) ... )` – z listy `xs` zawierającej liczby wybierz te elementy, które są parzyste i stwórz listę ich kwadratów zachowując kolejność, np.

```
> (kwadraty-liczb-parzystych '(2 3 4 6 7))
'(4 16 36)
```

- `(define (aplikuj-procedury fs x) ... )` – Aplikuje po kolei (od prawej do lewej) procedury znajdujące się na liście `fs` do wartości początkowej `x`, np.

```
> (aplikuj-procedury (list reverse cdr cdr) '(0 1 2 3 4))
'(4 3 2)
> (aplikuj-procedury (list (lambda(x)(* x 2))
                           (lambda(x)(+ x 1))
                           abs)
                     -2)
6
```

- `(define (wstaw-pomiedzy x xs) ... )` – stwórz listę poprzez wstawienie elementu `x` pomiędzy każde dwa elementy na liście `xs` (także na początku i końcu), np.

```
> (wstaw-pomiedzy 0 '(2 3 4 5))
'(0 2 0 3 0 4 0 5 0)
> (wstaw-pomiedzy 0 null)
'(0)
> (wstaw-pomiedzy 0 '(2))
'(0 2 0)
```

- `(define (grupuj xs) ... )` – stwórz listę list, w której wewnętrzne listy powstały poprzez zgrupowanie takich samych elementów na liście `xs`, np.

```
> (grupuj '(1 2 2 3 4 4 4 5 5 6))
'((1) (2 2) (3) (4 4 4) (5 5) (6))
> (grupuj '())
'()
> (grupuj '(1))
'((1))
> (grupuj '(1 2))
'((1) (2))
```

Zadanie 2. (5 pkt.)

Zdefiniuj typ danych (interfejs i reprezentację) drzew, które mają dwa rodzaje wierzchołków: wierzchołek z dwoma poddrzewami i wierzchołek z trzema poddrzewami. Niech każdy z tych rodzajów wierzchołków będzie etykietowany.

Zdefiniuj procedurę `sciezki`, która zwraca listę wszystkich ścieżek (czyli list etykiet w kolejności od korzenia do liścia) z takiego drzewa. Pamiętaj o zachowaniu abstrakcji danych.

Zadanie 3. (5 pkt.)

Przyjmijmy definicje:

```
(define (map f xs)
  (if (null? xs)
      null
      (cons (f (car xs))
            (map f (cdr xs)))))

(define (sum xs)
  (if (null? xs)
      0
      (+ (car xs)
         (sum (cdr xs)))))
```

Zdefiniujmy też procedurę, która łączy dwie powyższe:

```
(define (map-sum f xs)
  (define (iter xs acc)
    (if (null? xs)
        acc
        (iter (cdr xs)
              (+ (f (car xs)) acc))))
  (iter xs 0))
```

Np.

```
> (map-sum abs '(-1 2 -3))
6
```

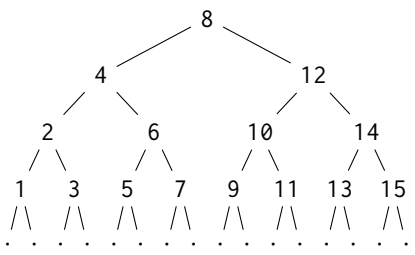
Udowodnij, że dla każdej procedury `f` i listy `xs` zachodzi `(map-sum f xs)` $\equiv$ `(sum (map f xs))`. Jak zwykle w takiej sytuacji zakładamy, że wartościami procedury `f` są zawsze liczby.

Zadanie 4. (5 pkt.)

Przypomnijmy interfejs drzewa binarnego z etykietami w węzłach z wykładu:

```
(define leaf 'leaf)                    ; konstruktor liscia
(define (leaf? t)                      ; predykat
(define (node v l r)                   ; konstruktor wezla
(define (node? t)                      ; predykat
(define (node-val t)                   ; selektory
(define (node-left t)
(define (node-right t)
(define (tree? t)                      ; predykat definiujacy
  (or (leaf? t)                        ; nasz glowny typ danych!
      (and (node? t)
           (tree? (node-left  t))
           (tree? (node-right t)))))
```

Zdefiniuj jednoargumentową procedurę `full-tree` taką, że wynikiem `(full-tree xs)` dla posortowanej listy liczb `xs` długości $2^n - 1$ (gdzie $n \in \mathbb{N}$) jest pełne drzewo BST zawierające elementy z `xs`. Przykładowo, wynikiem dla listy `'(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)` powinno być następujące drzewo:



Uwaga! Postaraj się, żeby Twoja procedura nie generowała nieużytków. Takie rozwiązania będą punktowane wyżej!

Zadanie 5. (5 pkt.)

W tym zadaniu sprawdzamy, czy dane drzewo binarne (zdefiniowane przy pomocy interfejsu z poprzedniego zadania) jest pełne. Można to zrobić tak:

```
(define (height t)
  (if (null? t)
      0
      (+ 1 (max (height (node-left t))
                (height (node-right t))))))

(define (full? t)
  (if (leaf? t)
      true
      (and (full? (node-left t))
           (full? (node-right t))
           (= (height (node-left t))
              (height (node-right t))))))
```

Ale można też poprawić złożoność i zaimplementować procedurę `full?` następująco:

```
(define (full?-2.0 t)
  (define (check t n)
    (if (and (= n 0) (leaf? t))
        true
        (and (check (node-left t) (- n 1))
             (check (node-right t) (- n 1)))))
  (check t (height t)))
```

Udowodnij, że dla dowolnego drzewa `t` zachodzi `(full? t)` $\equiv$ `(full?-2.0 t)`.

Wskazówka: Być może przyda Ci się lemat mówiący, że jeśli `(check t n)` $\equiv$ `true`, to `(height t)` $\equiv$ `n`. Możesz użyć tego lematu i nie musisz go dowodzić.