Lista zagadnień nr 4

Zamiast zajęć

W tym tygodniu będziemy kontynuować pracę nad abstrakcją danych i posługiwaniem się klasycznymi strukturami w programowaniu funkcyjnym: listami i drzewami, pojawi się też pojęcie *cytowania*. Przed zajęciami należy przeczytać ze zrozumieniem **Rozdziały 2.2** i **2.3** podręcznika, potrafić zdefiniować bardziej skomplikowany typ danych za pomocą **predykatów**, a także rozumieć związek między predykatami definiującymi typy danych i **twierdzeniami o indukcji** dla tych typów danych. W szczególności należy przypomnieć sobie twierdzenia o indukcji dla list i zastanowić się nad podobnym twierdzeniem dla drzew binarnych.

Ćwiczenie 1.

Problem n hetmanów polega na rozmieszczeniu na szachownicy o wymiarach $n \times n$ hetmanów tak, aby się wzajemnie nie szachowały. Jeden ze sposobów rozwiązania tego problemu polega na umieszczaniu hetmanów w kolejnych kolumnach (jeśli chcemy ustawić na szachownicy n hetmanów, to oczywiście w każdej kolumnie musi być dokładnie jeden hetman). Ta obserwacja daje nam natychmiast rozwiązanie rekurencyjne problemu, w którym aby stworzyć ustawienie hetmanów dla k pierwszych kolumn rozwiązujemy problem dla $k-1$ pierwszych kolumn, a następnie dostawiamy hetmana w k-tej kolumnie. Jeśli hetmana w k-tej kolumnie dostawimy na wszystkie możliwe sposoby, otrzymamy listę wszystkich rozwiązań problemu.[1]

Poniżej przedstawione jest niemal gotowe rozwiązanie problemu. Brakuje w nim implementacji trzech procedur: `adjoin-position`, `empty-board` i `safe?`. Twoim zadaniem jest zaproponowanie *reprezentacji* częściowego rozwiązania (tj. ustawienia hetmanów w k pierwszych kolumnach szachownicy) i zaimplementowanie brakujących procedur.

```
(define (queens board-size)
  ;; Return the representation of a board with 0 queens inserted
```

[1] Zauważ podobieństwo do zadania z permutacjami z poprzedniej listy!

```
(define (empty-board)
  )
;; Return the representation of a board with a new queen at
;; (row, col) added to the partial representation `rest'
(define (adjoin-position row col rest)
  )
;; Return true if the queen in k-th column does not attack any of
;; the others
(define (safe? k positions)
  )
;; Return a list of all possible solutions for k first columns
(define (queen-cols k)
  (if (= k 0)
      (list (empty-board))
      (filter
       (lambda (positions) (safe? k positions))
       (concatMap
        (lambda (rest-of-queens)
          (map (lambda (new-row)
                 (adjoin-position new-row k rest-of-queens))
               (from-to 1 board-size)))
        (queen-cols (- k 1))))))
(queen-cols board-size))
```

Ćwiczenie 2.

W procedurze `queen-cols` zamień kolejność odwzorowań, tj. zastąp wywołanie `concatMap` następującym kodem:

```
(concatMap
 (lambda (new-row)
   (map (lambda (rest-of-queens)
          (adjoin-position new-row k rest-of-queens))
        (queen-cols (- k 1))))
 (from-to 1 board-size))
```

Jaki jest efekt tej zamiany? Dlaczego?

Ćwiczenie 3.

Rozważmy reprezentację drzew binarnych z etykietami w wierzchołkach zdefiniowaną predykatem `btree?` (odpowiadającym predykatowi `tree?` z dołączonego kodu):

```
(define (btree? t)
  (or (eq? t 'leaf)
      (and (list? t)
           (= 4 (length t))
```

```
            (eq? (car t) 'node)
            (btree? (caddr t))
            (btree? (cadddr t)))))
```

W reprezentacji tej liście są dane symbolem `leaf`, zaś wierzchołki — czwórką, w której pierwszy element jest symbolem `node`, drugi — dowolną etykietą, zaś trzeci i czwarty — odpowiednio lewym i prawym poddrzewem.

Zdefiniuj procedurę `mirror` zwracającą „lustrzane odbicie” danego drzewa. Przykładowo,

```
(mirror '(node a (node b (node c leaf leaf) leaf) (node d leaf leaf))
   ≡ '(node a (node d leaf leaf) (node b leaf (node c leaf leaf))).
```

Ćwiczenie 4.

Dla reprezentacji drzew z poprzedniego zadania zaimplementuj procedurę `flatten` zwracającą listę etykiet w kolejności infiksowej. Zadbaj o to, aby Twoja procedura nie tworzyła pomocniczych list nie będących częścią wyniku (w szczególności — nie używaj procedury `append`).

Ćwiczenie 5.

Zwróć uwagę że powyższa reprezentacja drzew opisuje również drzewa przeszukiwań binarnych z wykładu (tj. każde drzewo BST z wykładu jest drzewem w sensie powyższego predykatu). Użyj procedury `insert` z wykładu i procedury `flatten` żeby zaimplementować procedurę `treesort`, sortującą listę poprzez wstawienie wszystkich jej elementów do drzewa poszukiwań binarnych, a następnie spłaszczenie go z powrotem do listy.

Ćwiczenie 6.

Zaimplementuj procedurę `delete`, zwracającą drzewo z usuniętym danym kluczem, dla reprezentacji drzew przeszukiwań binarnych z wykładu. Wskazówka: Aby stworzyć drzewo przeszukiwań binarnych z którego usunęliśmy korzeń, najlepiej znaleźć (jeśli istnieje) najmniejszy element większy od tego korzenia.

Zadania domowe

Ćwiczenie 7.

Kopce lewicowe (znane też jako drzewa lewicowe) to prosta i efektywna struktura danych implementująca kolejkę priorytetową (którą na wykładzie zaimplementowaliśmy używając nieefektywnej struktury listy posortowanej), zaproponowana w 1972 roku przez Clarka Crane'a i uproszczona rok później przez Donalda Knutha. Podobnie jak w przypadku posortowanej listy, chcemy móc znaleźć najmniejszy element w stałym czasie, jednak chcemy żeby pozostałe operacje (wstawianie, usuwanie minimum i scalanie dwóch kolejek) działały szybko — czyli w czasie logarytmicznym. W tym celu, zamiast listy budujemy *drzewo binarne*, w którym wierzchołki zawierają elementy kopca wraz z wagami. Dodatkowym niezmiennikiem struktury danych, który umożliwi efektywną implementację jest to, że każdemu kopcowi przypisujemy *rangę*, którą jest długość „prawego kręgosłupa" (czyli ranga prawego poddrzewa zwiększona o 1 — lub zero w przypadku pustego kopca), i że w każdym poprawnie sformowanym kopcu ranga lewego poddrzewa jest nie mniejsza niż ranga prawego poddrzewa.

Pozwala to nam zdefiniować następującą implementację:

```
(define (elem-val x)
  (cdr x))

;;; leftist heaps (after Okasaki)

;; data representation
(define leaf 'leaf)

(define (leaf? h) (eq? 'leaf h))

(define (hnode? h)
  (and (tagged-list? 5 'hnode h)
       (natural? (caddr h))))

(define (make-hnode elem heap-a heap-b)
  ;;; XXX: fill in the implementation
  ...)

(define (hnode-elem h)
  (second h))

(define (hnode-left h)
  (fourth h))

(define (hnode-right h)
  (fifth h))

(define (hnode-rank h)
  (third h))

(define (hord? p h)
  (or (leaf? h)
      (<= p (elem-priority (hnode-elem h)))))
```

```
(define (heap? h)
  (or (leaf? h)
      (and (hnode? h)
           (heap? (hnode-left h))
           (heap? (hnode-right h))
           (<= (rank (hnode-right h))
               (rank (hnode-left h)))
           (= (rank h) (inc (rank (hnode-right h))))
           (hord? (elem-priority (hnode-elem h))
                  (hnode-left h))
```

Liście reprezentujemy tu przez symbol `leaf`, wierzchołki zaś jako listy pięcioelementowe w której pierwszy element to symbol `hnode`, drugi to element kopca, trzeci to *ranga* danego wierzchołka (patrz `rank`), a czwarty i piąty — odpowiednio lewe i prawe poddrzewo. Predykat `heap?` sprawdza ponadto czy zachowany jest porządek kopca (używając `hord?`) i czy własność rangi opisana powyżej jest spełniona. Selektory `node-elem`, `node-left` i `node-right` są dostarczone, należy jednak zaimplementować konstruktor `make-node`. Zwróć uwagę, że nie przyjmuje on rangi tworzonego kopca, ale musi ją wyliczyć. Oznacza też, że musimy stwierdzić w procedurze konstruktora który z kopców powinien zostać prawym, a który lewym poddrzewem (możemy natomiast założyć że porządek kopca zostanie zachowany). Zwróć uwagę na użycie procedury `elem-priority` znajdującej priorytet elementu w kopcu (który powinien być liczbą).

Mając powyższą reprezentację danych, możemy zaimplementować operacje na kopcu:

```
(define (rank h)
  (if (leaf? h)
      0
      (hnode-rank h)))

;; operations

(define empty-heap leaf)

(define (heap-empty? h)
  (leaf? h))

(define (heap-insert elt heap)
  (heap-merge heap (make-hnode elt leaf leaf)))

(define (heap-min heap)
  (hnode-elem heap))

(define (heap-pop heap)
  (heap-merge (hnode-left heap) (hnode-right heap)))
```

Wszystkie operacje na kopcach za wyjątkiem scalania są dostarczone: predykat `heap-empty?` sprawdza czy kopiec jest pusty, `heap-insert` wstawia element do kopca, `heap-min` znajduje element o minimalnym priorytecie, zaś `heap-pop` usuwa ten element z kopca. Wstawianie i usuwanie są zaimplementowane przez scalanie, a uzupełnienie tej implementacji jest Twoim zadaniem. Idea scalania kopców jest następująca: jeśli jeden z kopców jest pusty, scalanie jest trywialne (bierzemy drugi kopiec). Jeśli oba są niepuste, możemy znaleźć najmniejszy element każdego z nich. Mniejszy z tych dwóch elementów powinien znaleźć się w korzeniu wynikowego kopca — łatwo go znaleźć. Mamy zatem cztery obiekty:

- element o najniższym priorytecie (nazwiemy go e),
- lewe poddrzewo kopca z którego korzenia pochodzi e — h_l
- prawe poddrzewo kopca z którego korzenia pochodzi e — h_r
- drugi kopiec, h, którego korzeń miał priorytet większy niż e.

Aby stworzyć wynikowy kopiec wystarczy teraz scalić h_r i h (rekurencyjnie), a następnie stworzyć wynikowy kopiec z kopca otrzymanego przez rekurencyjne scalanie, kopca h_l i elementu e. Poprawnie zdefiniowany konstruktor `make-heap` sam zadba o to, żeby otrzymany kopiec spełniał niezmienniki struktury danych.

Ćwiczenie 8.

Kopce (czy, ogólniej, kolejki priorytetowe) dają nam kolejny sposób sortowania list: sortowanie przez kopcowanie. Użyj swojej uzupełnionej implementacji kopców lewicowych aby zdefiniować procedurę `heapsort` sortującą listę przez wstawienie wszystkich elementów listy do kopca, a następnie odbudowanie na jego podstawie listy w rosnącej kolejności elementów. Zadbaj o to, aby korzystać jedynie z *interfejsu* kopca, tak aby procedura sortowania była niezależna od jego wewnętrznej reprezentacji.