Definiowanie własnych typów danych

W Camlu można oczywiście definiować własne algebraiczne typy danych. Definicja takiego typu przypomina definicję rekordu z wariantami w Pascalu. Podaje się w niej listę nazw „wariantów" danego typu wraz z typami wartości, które są związane z tymi wariantami. Nazwy wariantów muszą się zaczynać wielką literą.
Na początek zdefiniujmy listy liczb.

```
# type lista_liczb =
    Pusta | Element of int * lista_liczb ;;
type lista_liczb =
       | Pusta | Element of int * lista_liczb

# Pusta ;;
- : lista_liczb = Pusta

# Element (1, Pusta) ;;
- : lista_liczb = Element (1, Pusta)

# Element (1, Element (2, Pusta));;
- : lista_liczb =
      Element (1, Element (2, Pusta))
```

Umiemy już budować wartości typów algebraicznych, musimy się jeszcze nauczyć je rozbierać. Do tego służy (suprise, suprise!) dopasowanie wzorców.

```
# let rec dlugosc_naszej_listy l =
    match l with
      Pusta -> 0
    | Element (_, l’) ->
        1 + dlugosc_naszej_listy l’;;
val dlugosc_naszej_listy
          : lista_liczb -> int = <fun>

# dlugosc_naszej_listy Pusta ;;
- : int = 0
# dlugosc_naszej_listy
    (Element (1, Element (2, Pusta)));;
- : int = 2
 # let rec lacz a b =
   match (a, b) with
     ([], a) -> a
   | (x :: xs, ys)
         -> x :: (lacz xs ys) ;;
```

Drzewa

Zajmiemy się teraz drzewami (binarnymi albo i nie) jako przykładem sensownej struktury danych, którą chcemy sami zdefiniować.

```
# type 'a tree = Et |
        Node of 'a * 'a tree * 'a tree;;
type 'a tree = | Et |
        Node of 'a * 'a tree * 'a tree


# Et;;
- : 'a tree = Et


# Node ("b",
        Node ("a", Et, Et),
        Node ("c", Et, Et));;
- : string tree = Node ("b",
                        Node ("a", Et, Et),
                        Node ("c", Et, Et))
```

W dalszym ciągu spróbujemy napisać pewną liczbę funkcji manipulujących drzewami (BST albo i nie).

Najpierw zajmiemy się wstawianiem elementu do drzewa BST.

```
# let rec insert_BST e t =
    match t with
      Et -> Node (e, Et, Et)
    |
      Node (x, t1, t2) ->
        if (e < x) then
          Node (x, insert_BST e t1, t2)
        else
          Node (x, t1, insert_BST e t2) ;;
val insert_BST : 'a ->
                 'a tree
                     -> 'a tree = <fun>

# insert_BST 1 Et;;
- : int tree = Node (1, Et, Et)

# insert_BST "a"
         (insert_BST "c"
                 (insert_BST "b" Et)) ;;
- : string tree = Node ("b",
                        Node ("a", Et, Et),
                        Node ("c", Et, Et))
```

Teraz zajmiemy się sprawdzaniem, czy element występuje w drzewie BST.

```
# let rec is_member_BST e t =
    match t with
      Et -> false
    |
      Node (x, t1, t2) ->
        if x = e then true
        else if e < x then
          is_member_BST e t1
        else
          is_member_BST e t1 ;;
val is_member_BST :
       'a -> 'a tree -> bool = <fun>

# is_member_BST 1 Et;;
- : bool = false

# is_member_BST 1
                (Node (2,
                      Node (1, Et, Et),
                      Node (3, Et, Et)));;
- : bool = true
```

```
# is_member_BST 7
                (Node (2,
                      Node (1, Et, Et),
                      Node (3, Et, Et)));;
- : bool = false
```

Operacji usuwania nie będę pisał na slajdzie, bo mi się nie chce, jako, że jej opis jest nieco przydługi i rozbija się na wiele przypadków. Niemniej jest ona równie łatwa do zaprogramowania.

A teraz kilka funkcji typowo rozrywkowych. Najpierw policzymy wysokość drzewa:

```
# let rec tree_hight t =
    match t with
      Et -> 0
    |
      Node (_, t1, t2) ->
        1 + max (tree_hight t1)
                (tree_hight t2) ;;
val tree_hight : 'a tree -> int = <fun>

# tree_hight Et;;
- : int = 0
```

```
# tree_hight
    (Node (2, Node (1, Et, Et),
              Node (3, Et, Et)));;
- : int = 2
```

potem liczbę wierzchołków oraz odbicie lustrzane:

```
# let rec non t =
    match t with
      Et -> 0
    |
      Node (_, t1, t2) ->
        1 + non t1 + non t2;;
val non : 'a tree -> int = <fun>
# non (Node (2, Node (1, Et, Et),
                Node (3, Et, Et)));;
- : int = 3

# let rec mirror t =
    match t with
      Et -> Et
    |
      Node (x, t1, t2) ->
        Node (x, mirror t2, mirror t1) ;;
val mirror : 'a tree -> 'a tree = <fun>
```

Istotnym zadaniem jest obchodzenie drzewa w różnych porządkach. Napiszemy funkcje, które obchodzą drzewo i budują listę odwiedzonych elementów.

```
# let rec inorder t =
    match t with
      Et -> []
    |
      Node (x, t1, t2) ->
        (inorder t1) @ [x] @ (inorder t2) ;;
val inorder : 'a tree -> 'a list = <fun>

# inorder (Node (2, Node (1, Et, Et),
                     Node (3, Et, Et)));;
- : int list = [1; 2; 3]

# let rec preorder t =
    match t with
      Et -> []
    |
      Node (x, t1, t2) ->
        [x] @ (preorder t1) @ (preorder t2) ;;
val preorder : 'a tree -> 'a list = <fun>
```

```
# preorder (Node (2, Node (1, Et, Et),
                     Node (3, Et, Et)));;
- : int list = [2; 1; 3]

# let rec postorder t =
    match t with
      Et -> []
    |
      Node (x, t1, t2) ->
        (postorder t1) @ (postorder t2) @ [x];;
val postorder : 'a tree -> 'a list = <fun>

# postorder (Node (2, Node (1, Et, Et),
                     Node (3, Et, Et)));;
- : int list = [1; 3; 2]
```

Korzystając z funkcji wstawiania do drzewa BST oraz z funkcji spłaszczania tegoż do listy w porządku infiksowym możemy łatwo napisać kolejną funkcję sortującą listę.

```
# let rec treesort l =
    inorder (foldr insert_BST l Et);;
val treesort : 'a list -> 'a list = <fun>

# treesort [];;
- : '_a list = []

# treesort [3; 1; 2];;
- : int list = [1; 2; 3]
```

Typy definiowalne przez użytkownika mogą też służyć do realizacji typów wyliczeniowych (jak w Pascalu):

```
# type kolor = Zielony | Czerwony | Niebieski |
               Czarny | Zolty | Bialy;;
type kolor = | Zielony | Czerwony | Niebieski |
               Czarny | Zolty | Bialy
```

Za pomocą typów algebraicznych można też zdefiniować typ liczb naturalnych i operacje na nich:

```
# type nat = Zero | Succ of nat ;;
type nat = | Zero | Succ of nat

# let rec sum a b =
    match a with
      Zero -> b
     |
      Succ a' ->
        Succ (sum a' b) ;;
val sum : nat -> nat -> nat = <fun>

# sum Zero Zero ;;
- : nat = Zero

# sum (Succ (Succ (Succ Zero)))
      ((Succ (Succ Zero))) ;;
- : nat = Succ (Succ (Succ (Succ (Succ Zero))))
```