Dalsze sekrety list

```
(* długość listy *)
# let rec dlugosc l =
    match l with
      [] -> 0
    |
      (_ :: y) ->
              1 + dlugosc y ;;
val dlugosc :
     'a list -> int = <fun>

# let dlug l =
    let rec dlug' l len =
      match l with
        [] -> len
      |
        (_ :: y) ->
                dlug' y (len + 1)
    in
      dlug' l 0;;
val dlug : 'a list -> int = <fun>
```

```
(* laczenie list *)
# let rec lacz a b =
    match (a, b) with
      ([], a) -> a
    | (x :: xs, ys)
          -> x :: (lacz xs ys) ;;
val lacz :
    'a list -> 'a list
                -> 'a list = <fun>

# let rec odwr l =
    match l with
      [] -> []
    |
      (x :: xs) ->
        lacz (odwr xs) [x] ;;
```

Korzystając z prostych i naturalnych operacji na listach dojść do kilku znanych algorytmów sortowania.

Zacznijmy od wstawiania elementu do posortowanej listy.

```
# let rec wstaw e l =
    match l with
      [] -> [e]
    |
      (x :: xs) ->
            if e < x then e :: x :: xs
            else x :: (wstaw e xs) ;;
val wstaw : 'a -> 'a list -> 'a list = <fun>

# wstaw 5 [2;3;7] ;;
- : int list = [2; 3; 5; 7]
```

Mając tę operację możemy już zdefiniować sortowanie przez wstawianie.

```
# let sortuj_przez_wstawianie l =
    let rec sort' co wynik =
      match co with
        [] -> wynik
      | (x :: xs)
           -> sort' xs (wstaw x wynik)
    in
      sort' l [] ;;
val sortuj_przez_wstawianie
              : 'a list -> 'a list = <fun>

# sortuj_przez_wstawianie
            [2;7;3;1;8;9;2;5;6;
             1;-1;65;12;4];;
- : int list =
         [-1; 1; 1; 2; 2; 3; 4; 5;
           6; 7; 8; 9; 12; 65]
```

Sortowanie przez wstawianie ma bardzo kiepską złożoność obliczeniową. Spróbujmy zaimplementować jakieś lepsze algorytmy.

Najpierw zaprogramujemy quicksort.

```
# let rec qsort l =
    match l with
      [] -> []
    | [x] -> [x]
    | (x :: xs) ->
        let rec podziel (l, p, co) =
          match co with
            [] -> (l, p)
          | (y :: ys) ->
              if y <= x then
                podziel (y :: l, p, ys)
              else
                podziel (l, y :: p, ys)
        in
          let (l, p) = podziel ([], [], xs)
          in
              (qsort l)
            @
              (x :: qsort p) ;;
```

Sortowanie szybkie ma wady. Innym, lepszym i prostym algorytmem jest sortowanie przez łączenie (mergesort).

```
# let rec polacz l p =
    match (l, p) with
      ([], p) -> p
    | (l, []) -> l
    | (x :: xs, y :: ys) ->
        if x < y then
          x :: (polacz xs  (y :: ys))
        else
          y :: (polacz (x :: xs) ys) ;;
val polacz : 'a list ->
                 'a list -> 'a list = <fun>
# let rec podziel (l, p, co) =
    match co with
      [] -> (l, p)
    | [x] -> (x :: l, p)
    | (x :: y :: reszta) ->
        podziel (x :: l, y :: p, reszta) ;;
val podziel : 'a list * 'a list * 'a list
                -> 'a list * 'a list = <fun>
```

Mając te funkcje pomocnicze możemy zapisać właściwy algorytm sortowania.

```
# let rec msort lista =
    match lista with
      [] -> []
    | [x] -> [x]
    | xs ->
        let (l, p) = podziel ([], [], xs)
        in
          polacz (msort l) (msort p) ;;
val msort : 'a list -> 'a list = <fun>

# msort  [2;7;3;1;8;9;2;5;6;
              1;-1;65;12;4];;
- : int list = [-1; 1; 1; 2; 2; 3; 4;
                 5; 6; 7; 8; 9; 12; 65]
```

Umiemy już sortować listy, teraz zajmiemy się wyszukiwaniem danych.

```
(* funkcja biorąca n początkowych elementów *)
# let rec taken n l =
    match l with
      [] -> []
    | (x :: xs) ->
        if n > 0 then
          x :: (taken (n - 1) xs)
        else
          [] ;;

(* porzucanie n początkowych elementów *)
# let rec dropn n l =
    match l with
      [] -> []
    | (x :: xs) ->
        if n > 0 then
          dropn (n - 1) xs
        else
          x :: xs ;;
```

```
(* n-ty element listy *)
# let rec nth n l =
    match l with
      [] -> raise (Failure "nth")
    | (x :: xs) ->
        if n = 0 then
          x
        else
          nth (n - 1) xs ;;

(* wybieranie wszystkich elementów, które
   spełniają warunek
*)
# let filter p l =
    match l with
      [] -> []
    | (x :: xs) ->
        if p x then
          x :: filter p xs
        else
          filter p xs ;;
```

Napiszemy teraz funkcję sprawdzającą, czy element występuje na liście:

```
# let rec member e l =
    match l with
      [] -> false
    | (x :: xs) ->
        if x = e then true
        else member e xs ;;
```

i funkcję sprawdzającą, czy każdy element jednej listy należy do drugiej:

```
let rec zawarta l1 l2 =
  match l1 with
    [] -> true
  | (x :: xs) ->
      member x l2 && zawarta xs l2 ;;
```

Oczywiście łatwo napisać wersję ogonowo–rekurencyjną tej funkcji.

Często potrzebna jest funkcja, która wskazaną funkcję stosuje do kolejnych elementów listy, dając listę wyników:

```
# let rec map f l =
    match l with
      [] -> []
    | (x :: xs) ->
        (f x) :: map f xs;;
val map : ('a -> 'b) ->
          'a list -> 'b list = <fun>
```

Na przykład w celu podniesienia wszystkich elementów listy do kwadratu wystarczy napisać:

```
# map (fun x -> x * x)
      [-12; 1; 13; 5; 23; 7; 16] ;;
- : int list = [144; 1; 169;
                25; 529; 49; 256]
```

Bardzo przydatne są następujące funkcje iteracji:

```
# let rec foldr f l b =
    match l with
      [] -> b
    | (x :: xs) ->
        f x (foldr f xs b) ;;
val foldr : ('a -> 'b -> 'b) ->
            'a list -> 'b -> 'b = <fun>

# let rec foldl f a l =
    match l with
      [] -> a
    | (x :: xs) ->
        foldl f (f a x) xs;;
val foldl : ('a -> 'b -> 'a) ->
            'a -> 'b list -> 'a = <fun>
```

Intuicyjne znaczenie tych funkcji można określić wzorami:

```
foldr f [a1; ...; an] b = f a1 (f a2 (...(f an b) ... )
foldl f a [b1; ...; bn] = f (... (f (f a b1) b2) ... ) bn
```

Funkcje `foldl` i `foldr` mogą wydawać się mało intuicyjne, ale pełnią one bardzo naturalne funkcje i pozwalają na bardzo zwięzły zapis wielu algorytmów. Poniżej pokażemy kilka przykładów takich definicji.

```
(* sumowanie elementów listy *)
# let sumuj l = foldr (+) l 0 ;;
(* mnożenie elementów listy *)
# let mnoz l = foldr (*) l 1 ;;

(* sprawdzanie, czy lista zawiera element *)
# let member' e l =
    foldr (fun x w -> w || e = x) l false ;;

(* sprawdzanie, czy wszystkie elementy listy
   spełniają podany warunek
*)
# let forall p l =
    foldr (fun x w -> w && p x) l true ;;

(* konkatenacja list *)
# let append' l1 l2 =
    foldr (fun x y -> x :: y) l1 l2 ;;
```

Jako przykład konkretnego zastosowania list, napiszemy funkcje do operacji na dowolnie dużych liczbach całkowitych zapisanych jako listy cyfr (dla uproszczenia — dwójkowo).