OCaml PRO Training – Initiation to OCaml for Imperative Programmers
Bloomberg London - March 2015

Zipper**

Topics

- Pattern matching
- Tail-recursion

Introduction

This training session presents a method for data structure traversal, that do not force any pre-establihed traversal order. This method allows local modification of the data structures. We presents two instantiations: the zipper upon lists and the zipper upon binary trees. For more details on the general method you may read: Gérard Huet, The Zipper, Journal of Functionnal Programming, 7(5) :549-554, September 1997.

Exercise 1 – Zipper upon list

In an imperative settings, the easiest way to navigate into a list is to implement double-linked list: given a cell in the list, we are able to move to the next or to the previous cell, and to insert value before or after the current cell. While we may implement double-linked list in OCaml by using mutation, in this exercise we want to implement a persistant structure of list that offers the same primitive operations.

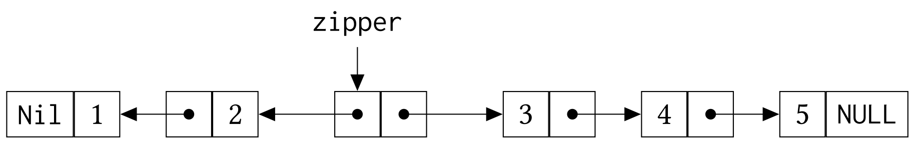
For that purpose, we propose the following type definition:

```ocaml
type 'a list_zipper = { left: 'a list; right: 'a list }
```

Given a initial list l and a position p in this list, the field `right` represents in order all the elements of l that come after the position p. The field `left` represents in *reverse order* all the elements of l that comes before the position p.

For instance, a zipper for the list `[1;2;3;4;5]` at position two will be the floowing value:

```ocaml
let zipper = { left = [2;1]; right = [3;4;5] }
```



Question 1.1 – Write a function `of_list` that receives a list and returns the corresping zipper at position 0.

Question 1.2 – Write the functions `move_right` and `move_left` that receives a zipper at position p and returns a zipper of the same list at position $p+1$ (resp $p-1$).

Question 1.3 – Write the functions `insert_before` and `insert_after` that receives a zipper and an element, and inserts the element in the list before (resp. after) the current position.

Question 1.4 – Write the functions `delete_before` and `delete_after` that receives a zipper and deletes the element just before (resp. after) the current position in the list.

Question 1.5 – Write the function `to_list`, of type `'a list_zipper -> a list`, that returns the list corresponding to a given zipper.

Question 1.6 – Given the following functions to read and write lines of a file, write a function `remove_empty_lines in out` that: read the content of the file `in`; use a zipper to traverse the list of lines and remove all the empty lines; write the result to the file `out`.

```
let read_lines file =
  let ic = open_in file in
  let rec loop acc =
    match input_line ic with
    | line -> loop (line :: acc)
    | exception End_of_file ->
        close_in ic;
        List.rev acc
  in
  loop []

let write_lines file lines =
  let oc = open_out file in
  let output_line line =
    output_string oc line;
    output_char oc '\n' in
  List.iter output_line lines;
  close_out oc
```

Is your function tail-recursive ?

Question 1.7 – Similarly, write a function `wrap_columns in out` that wrap lines longer than 80 characters. Is your function tail-recursive ?

Exercise 2 – Zipper upon binary tree

Following the same idea, we may traverse tree with a zipper by using the following type definition, where `path` represents the path from the current node to the tree's root:

```
type 'a tree =
  | Empty
  | Node of 'a tree * 'a * 'a tree
type 'a path =
  | Top
  | Left of 'a path * 'a * 'a tree
  | Right of 'a tree * 'a * 'a path
type 'a tree_zipper = { path: 'a path; tree: 'a tree }
```

The path also contains the upper subtrees in which we did not descend.

The initial zipper of a tree is:

```
let of_tree tree = { path = Top; tree }
```

Question 2.1 – Write the functions `down_right` and `down_left`, that receives a `tree_zipper` and step down in their respective sons.

Question 2.2 – Write a function `up` that receives a `tree_zipper` and step up into the tree.

Question 2.3 – Write a function `to_tree` that receives a `tree_zipper` and returns the corresponding `tree`.

Question 2.4 – Adapt the `tree_zipper` to the red-black tree, and implement a tail-recursive version of the function `insert`.

Zipper

OLUTIONS – SOLUTIONS – SOLUTIONS – SOLUTIO

Solution to question 1.1

```ocaml
let of_list l = { left = []; right = l }
```

Solution to question 1.2

```ocaml
let move_left z =
  match z.left with
  | [] -> invalid_arg "move_left"
  | x :: left -> { left; right = x :: z.right}
```

Solution to question 1.3

```ocaml
let insert_before x z = { z with left = x :: z.left }
let insert_after x z = { z with right = x :: z.right }
```

Solution to question 1.4

```ocaml
let delete_before z =
  match z.left with
  | [] -> invalid_arg "delete_before"
  | _ :: left -> { z with left }
let delete_after z =
  match z.right with
  | [] -> invalid_arg "delete_after"
  | _ :: right -> { z with right }
```

Solution to question 1.5

```ocaml
let to_list z = List.rev_append z.left z.right
```

Solution to question 1.6

```ocaml
let next z =
  match z.right with
  | [] -> None
  | x :: _ -> Some x
let remove_empty in out =
  let rec loop z =
    match next z with
    | None -> to_list z
    | Some x ->
        if x = "" then loop (delete_after z) else loop (move_after z) in
  let lines = read_file in in
  write_files out (to_list (loop (of_list lines)))
```

Solution to question 2.1

```
let down_left z =
  match z.tree with
  | Empty -> invalid_arg "down_left"
  | Node (l, x, r) -> { path = Left (z.path, x, r); tree = l }
let down_right z =
  match z.tree with
  | Empty -> invalid_arg "down_right"
  | Node (l, x, r) -> { path = Right (z.path, x, l); tree = r }
```

Solution to question 2.2

```
let up z = match z.path with
  | Top -> invalid_arg "top"
  | Left (p, x, r) -> { path = p; tree = N (z.tree, x, r) }
  | Right (p, x, l) -> { path = p; tree = N (l, x, z.tree) }
```

Solution to question 2.3

```
let rec to_tree z = if z.path = Top then z.tree to_tree (up z)
```