L28: Advanced functional programming

Exercise 2

Due on March 2nd

1. The following types represent a balanced binary tree:

```
type z = Z : z
type 'n s = S : 'n -> 'n s

type (_, _, _) eql3 =
  | EqRefl : ('a, 'a, 'a) eql3

type ('a, _) btree =
| Empty : ('a, z) btree
| Tree : ('a, 'm) btree * 'a * ('a, 'n) btree
            * ('m, 'n, 'o) eql3 -> ('a, 'o s) btree
```

by enforcing the constraint that branches of a `Tree` node have equal depth.

An AVL tree is a binary tree where the depth of the two branches of a `Tree` node differ by at most 1. By replacing `eql3` with a type that enforces this constraint, we can create a type – similar to `btree` – which represents AVL trees.

   (i) Fill in the `?` in the following code to create a type `atree` which represents an AVL tree by enforcing the constraint on the depth of branches.

```
type (_,_,_) diff =
  | Less : (?, ?, ?) diff (* Left branch depth one less than right branch *)
  | Same : (?, ?, ?) diff (* Both branches at the same depth *)
  | More : (?, ?, ?) diff (* Left branch depth one more than right branch *)

type ('a, 'd) atree =
| Empty : ('a, z) atree
| Tree : ('a, 'm) atree * 'a * ('a, 'n) atree
            * ('m,'n,'o) diff -> ('a,'o s) atree
```

The following sum type represents the results of comparing two values:

```
type compare = LessThan | Equal | GreaterThan
```

such that a comparison function of type `'a -> 'a -> compare` returns

- `LessThan` if the first argument is less than the second
- `Equal` if the first argument is equal to the second,

- `GreaterThan` if the first argument is greater than the second

(ii) Implement the membership function `member` of type:

```
val member : ('a -> 'a -> compare) -> 'a -> ('a, 'd) atree
               -> bool
```

such that `member cmp x t` returns true iff the value `x` is present in the AVL tree `t` *assuming that the elements of the tree are in order according to the comparison function* `cmp`.

Given

- a binary tree (`l`) of depth n
- a value (`v`)
- a binary tree (`r`) of depth $n+2$

then the binary tree Tree(l, v, r) is not a valid AVL tree because it would break the invariant that the branches' depths differ by at most 1.

However, the following rotation algorithm will create a binary tree that is a valid AVL tree and whose elements are in the same order as Tree(l, v, r):

```
let Tree(rl, rv, rr) = r in
  if depth(rl) <= depth(rr) then Tree(Tree (l, v, rl), rv, rr)
  else
    let Tree(rll, rlv, rlr) = rl in
      Tree(Tree (l, v, rll), rlv, Tree (rlr, rv, rr))
```

(iii) Using the following type for the result of the algorithm:

```
type ('a, 'd) result =
  | SameDepth : ('a, 'd) atree -> ('a, 'd) result
  | Deeper : ('a, 'd s) atree -> ('a, 'd) result
```

Implement the above algorithm as a function `rotate_left` of type:

```
val rotate_left : ('a, 'd) atree -> 'a -> ('a, 'd s s) atree
                    -> ('a, 'd s s) result
```

(iv) Implement the dual operation which rotates a tree to the right as a function `rotate_right` of type:

```
val rotate_right : ('a, 'd s s) atree -> 'a -> ('a, 'd) atree
                     -> ('a, 'd s s) result
```

Insertion of an element into an *ordered* AVL tree is very similar to insertion of an element into an ordered binary tree, except that in some cases `rotate_left` and `rotate_right` are needed to maintain the constraint on branch depth.

(v) Implement the insertion function `insert` of type:

```
val insert : ('a -> 'a -> compare) -> 'a -> ('a, 'd) atree
                -> ('a, 'd) result
```

such that insert cmp x t returns an ordered AVL tree that contains the value `x` and all elements of the ordered AVL tree `t`. *The elements of `t` are assumed to be in order according to the comparison function `cmp`, and the elements of the resulting tree must also be in order according to the comparison function `cmp`.* You can assume that the input trees contain no duplicates and should ensure that the result contains no duplicates.

AVL trees are a good data-structure for implementing sets.

(vi) Implement a functor `Set` of module type:

```
module Set :
  functor (X : sig type t val compare : t -> t -> compare end) ->
    sig
      type t
      val empty : t
      val member : X.t -> t -> bool
      val insert : X.t -> t -> t
    end
```

which implements sets using `atree`.

2. The types in the following module represent fragments of a subset of XHTML:

```
module Untyped = struct

  type element =
    | Data : string -> element (* Alphanumeric strings *)
    | P : t -> element (* <p> body </p> *)
    | Em : t -> element (* <em> body </em> *)
    | A : t -> element (* <a> body </a> *)
    | Table : t -> element (* <table> body </table> *)
    | Tr : t -> element (* <tr> body </tr> *)
    | Td : t -> element (* <td> body </td> *)

  and t = element list

end
```

For example:

```
let example =
  [Untyped.P
    [Untyped.Data "hello ";
     Untyped.Em [Untyped.Data "world"]]]
```

represents "`<p>hello <em>world</em></p>`".

However, XHTML restricts its values to those which obey the following grammar:

```
td ::= <td> flow* </td>

tr ::= <tr> td+ </tr>

flow ::= block | inline

block ::= p | table
p ::= <p> inline* </p>
table ::= <table> tr+ </table>

inline ::= DATA | em | a
em ::= <em> inline* </em>
a ::= <a> inline* </a>
```

where `foo*` is a sequence of 0 or more `foo`s, `foo+` is a sequence of 1 or more `foo`s and `DATA` is a string of alphanumeric characters.

The GADT `t` in the following module represents the classes `tr`, `td`, `inline` and `block` from the grammar:

```
module Kind = struct

  type tr = Tr
  type td = Td

  type inline = Inline
  type block = Block

  type 'k flow =
    | Inline : inline flow
    | Block : block flow

  type 'e t =
    | Tr : tr t
    | Td : td t
    | Flow : 'k flow -> 'k flow t

end
```

and the index of the `t` type reflects its value.

For example, `Flow Inline` represents the `inline` grammar class and has type `inline flow t`.

Using the same type indices as `Kind.t` we can create a type to represent XHTML values that only allows values which obey the XHTML grammar.

(i) Fill in the ? in the following module definition so that `Typed.t` represents the XHTML values which are valid according to the grammar:

```
module Typed = struct

  type 'e element =
    | Data : string -> ? element
    | P : ? t -> ? element
    | Em : ? t -> ? element
    | A : ? t -> ? element
    | Table : ? t -> ? element
    | Tr : ? t -> ? element
    | Td : ? t -> ? element

  and 'e t =
    | Empty : ? t
    | Single : ? element -> ? t
    | Cons : ? element * ? t -> ? t

end
```

(ii) Write a function `verify` with type:

```
val verify : 'k Kind.t -> Untyped.t -> 'k Typed.t option
```

where `verify k u`

- returns `u` converted to a `Typed.t` if `u` is valid and is an instance of the grammar class `k`
- returns `None` otherwise

The following function `index_table` takes a function (`f`) and an untyped fragment of XHTML (`tbl`), and if that fragment represents a table then it returns a new table that has an additional column whose value is filled in by `f`. If `tbl` does not represent a table, or `f` returns XHTML that is not valid as the contents of a table cell, then `index_table` returns `None`.

```
let rec check_inline xs =
  let open Untyped in
    match xs with
    | [] -> true
    | Data _ :: xs -> check_inline xs
    | P _ :: xs -> check_inline xs
    | Em _ :: xs -> check_inline xs
    | A _ :: xs -> check_inline xs
    | Table _ :: xs -> check_inline xs
    | Tr _ :: _ -> false
    | Td _ :: _ -> false

let index_table f tbl =
  let open Untyped in
  let rec loop idx = function
    | [] -> Some []
    | (Tr cols) :: rows -> begin
        let col = f idx in
          if not (check_inline col) then None
          else
            match loop (idx + 1) rows with
            | None -> None
            | Some rows ->
                let row = Tr (Td col :: cols) in
                  Some (row :: rows)
      end
    | _ -> None
  in
    match tbl with
    | Table rows -> begin
        match loop 1 rows with
        | None -> None
```

```
      | Some rows -> Some (Table rows)
    end
  | _ -> None
```

(iii) Write a new version of `index_table` which uses `Typed.t` instead of `Untyped.t`. It should have type:

```
val index_table : (int -> 'a Kind.flow Typed.t) ->
                    Kind.block Kind.flow Typed.element ->
                      Kind.block Kind.flow Typed.element option
```