Important notes about grading:

1. **Compiler errors:** All code you submit must compile. **Programs that do not compile will receive an automatic zero (firm rule this time)**. If you run out of time, it is better to comment out the parts that do not compile, than hand in a more complete file that does not compile.
2. **Late assignments:** Please carefully review the course website's policy on late assignments, as all assignments handed in after the deadline will be considered late. Verify that you have submitted the correct version, before the deadline. Submitting the incorrect version before the deadline and realizing that you have done so after the deadline will **no longer** be accepted.

How to start:

1. Download the source code here.
2. You will need to implement your code in `assignment3/src/assignment3.ml`. We give detailed instructions on the functions that you need to implement below.
3. The test-cases (also seen below) are provided in `assignment3/test/public.ml` (if using `Dune`) and `assignment3/main.ml` (if using `Make`).
4. To compile and run your implementation, we provide two options:
   Option (1): Using `Dune`: type `dune runtest -f` in the command-line window under `assignment3` directory.
   Option (2): Using `Make`: First, to compile the code, type `make` in the command-line window under `assignment3` directory. Second, to run the code, type `./main.byte` in the command-line window under `assignment3` directory.

Mutability and Modules

In this assignment, you will design and implement a couple of mutable data structures and operations on them. You can use library functions in your implementation for this assignment.

Problem 1

Implement more functions for the `doubly linked list` mutable record covered in the lecture.

In [ ]:

```
type 'a element = {
  content : 'a;
  mutable next : 'a element option;
  mutable prev : 'a element option
}
```

The following functions were already covered in the lecture. Please **go over these functions again** before working on this problem.

In [ ]:

```
let create () = ref None

let is_empty t = !t = None

let insert_first l c =
  let n = {content = c; next = !l; prev = None} in
  let _ = match !l with
    | Some o -> (o.prev <- Some n)
    | None -> () in
  let _ = (l := Some n) in
  n

let insert_after n c =
  let n' = {content = c; next = n.next; prev = Some n} in
  let _ =  match n.next with
    | Some o -> (o.prev <- (Some n'))
    | None -> () in
  let _ = (n.next <- (Some n')) in
  n'
```

```
let remove t elt =
  let prev, next = elt.prev, elt.next in
  let _ = match prev with
    | Some prev -> (prev.next <- next)
    | None -> t := next in
  let _ =  match next with
    | Some next -> (next.prev <- prev)
    | None -> () in
  let _ = (elt.prev <- None) in
  let _ = (elt.next <- None) in
  ()       (* return void *)

let iter t f =
  let rec loop node =
    match node with
      | None -> ()
      | Some el ->
        let next = el.next in
        let _ = f el in
        loop (next)
  in
  loop !t
```

Note that the `iter` function above was implemented a little differently than the version in the lecture.

You need to implement the following new fucntions:

In [ ]:

```
(** [dll_of_list l] returns a new doubly linked list with the list of
      elements from the OCaml (singly linked) list [l].
    val dll_of_list : 'a list -> 'a element option ref
*)
let dll_of_list l =
    (* YOUR CODE HERE *)
    raise (Failure "Not implemented")


(** [list_of_dll t] returns a new OCaml (singly linked) list with the list of
      elements from the doubly linked list [t].
      Hint: Use [iter] to implement it.
    val dll_of_list : a element option ref -> 'a list
*)
let list_of_dll t =
    (* YOUR CODE HERE *)
    raise (Failure "Not implemented")


(** Returns the length of the list.
      Hint: Use [iter] to implement it.
*)
let length t =
    (* YOUR CODE HERE *)
    raise (Failure "Not implemented")


(** Given a doubly linked list [t] = [1->2], [duplicate t] returns [1->1->2->2].
      [t] should reference the head of the duplicatd doubly linked list after the function returns
.
      Hint: Use [iter] to implement it.
    val duplicate : 'a element option ref -> unit
*)
let duplicate t =
    (* YOUR CODE HERE *)
    raise (Failure "Not implemented")


(** [reverse t] reverses a doubly linked list [t] in-place.
      [t] should reference the head of the reversed doubly linked list after the function returns.
      Hint: Use [iter] to implement it.
    val reverse : 'a element option ref -> unit
*)
let reverse t =
    (* YOUR CODE HERE *)
    raise (Failure "Not implemented")
```

```
    false (failure "Not implemented")
```

Some Test-cases:

In [ ]:

```
let dll = dll_of_list [] in
assert (is_empty dll = true)

let dll = dll_of_list [1;2;3] in
assert (not(is_empty dll = true))

let dll = dll_of_list [1;2;3] in
let s = ref 0 in
let _ = iter dll (fun c -> s := !s + c.content) in
assert (!s = 6)

let dll = dll_of_list [1;2;3] in
let n1 = match !dll with
    | Some n1 -> n1
    | None -> failwith "impossible" in
let _ = assert (n1.content = 1)
let l = list_of_dll dll in
assert (l = [1;2;3])

let l = dll_of_list [1;2;3] in
let _ = assert (length l = 3) in
let _ = duplicate l in
let _ = assert (length l = 6)
assert (list_of_dll dll = [1;1;2;2;3;3])

let l = dll_of_list [1;2;3;4;5] in
let _ = reverse l in
assert (list_of_dll l = [5;4;3;2;1])
```

Problem 2

Implement more functors for the `Serializable` signature.

A `fold` function is added to the `Serializable` signature. Since the signature serves as the abstraction of many data structures, it would be useful to include a `fold` definition to support useful data structure functions.

In [ ]:

```
module type Serializable = sig
  type t       (* The type of a serializable data structure e.g. list and array *)
  type content (* The type of nodes (e.g. list nodes or array cells) stored in the serializable da
ta structure *)

  val string_of_t : t -> string

  val fold : ('a -> content -> 'a) -> 'a -> t -> 'a
end
```

Observe that the `fold` definition is similar to `List.fold_left`.

In the lecture, we created a `SerializableList` functor. You need to provide an implementation for the `fold` function, that was just introduced, in the `SerializableList` functor. You can use libraray functions in your implementation.

In [ ]:

```
module SerializableList (C : Serializable) = struct
  type t = C.t list      (* The serializable data structure is a list *)
  type content = C.t     (* Each node of the serializable data structure has a type C.t *)

  let string_of_t l =
    let rec loop acc l = match l with
      | [] -> acc
      | [x] -> acc ^ (C.string_of_t x)
```

```
    | x::xs -> loop (acc ^ (C.string_of_t x) ^ ";") xs
  in
  "[" ^ (loop "" l) ^ "]"

let fold f accum l =
  (* YOUR CODE HERE *)
  raise (Failure "Not implemented")
end
```

Similarly, provide a `SerializableArray` functor, which should look very much similar to `SerializableList`. You can find useful array APIs here.

In [ ]:

```
(* Implement the functor SerializableArray *)


module SerializableArray (C : Serializable) = struct
  (* YOUR CODE HERE *)
  raise (Failure "Not implemented")
end
```

You can use the functors to create arbitrarily nested data structures. We start from simple ones like `SerializableFloatList` covered in the lecture.

In [ ]:

```
module SerializableIntArray = SerializableArray (struct
  type t = int
  type content = int

  let string_of_t x = string_of_int x
  let fold f i res = f i res
end)

module SerializableIntList = SerializableList (struct
  type t = int
  type content = int

  let string_of_t x = string_of_int x
  let fold f i res = f i res
end)
```

In the above, a `SerializableIntArray` structure and a `SerializableIntList` structure were created by applying the functors. Each element in the serializable data structuers is an integer structure, which looks familiar to the float structure covered in the lecture.

Observe that the signature of `SerializableIntArray` and `SerializableIntList` are also Serializable.

Some Examples and Test-cases:

In [ ]:

```
(* Behold the power of abstraction by creating nested arrays: *)
module SerializableIntArrayArray = SerializableArray(SerializableIntArray)

(* Behold the power of abstraction by creating arrays of lists: *)
module SerializableIntListArray = SerializableArray(SerializableIntList)

(* Behold the power of abstraction by creating lists of arrays: *)
module SerializableIntArrayList = SerializableList(SerializableIntArray)


(* array to string *)
assert (SerializableIntArray.string_of_t [|1;2;3|] = "[|1;2;3|]")

(* array of arrays to string*)
assert (SerializableIntArrayArray.string_of_t [|[|1|]; [|2;3|]; [|4;5;6|]|] = "[|[|1|];[|2;3|];[|4;
5;6|]|]")
```

```
(* folding in all elements of an array of arrays to a plain list *)
assert (SerializableIntArrayArray.fold (fun xs x -> xs @ [x]) [] [|[|1|]; [|2;3|]; [|4;5;6|]|] = [1
;2;3;4;5;6])

(* array of lists to string *)
assert (SerializableIntListArray.string_of_t [|[7;8;9];[10;11;12];[13]|] = "[|[7;8;9];[10;11;12];[1
3]|]")

(* folding in all elements of an array of lists to a number by adding them together *)
assert (SerializableIntListArray.fold (+) 0 [|[7;8;9];[10;11;12];[13]|] = 70)

(* list of arrays to string *)
assert (SerializableIntArrayList.string_of_t [[|7;8;9|];[|10;11;12|];[|13|]] = "[[|7;8;9|];[|10;11;
12|];[|13|]]")

(* folding in all elements of a list of arrays to a number by adding them together *)
assert (SerializableIntArrayList.fold (+) 0 [[|7;8;9|];[|10;11;12|];[|13|]] = 70)
```

Via these examples, you should convince yourself that you can use functors to create arbitrarily nested data structures without having to write specific functions for each. This is the power of modularity!