CIS 120 Midterm II March 29, 2019

SOLUTIONS

1. **Mutable Lists (abstract types, invariants, mutable state, ASM)**

In this problem we will implement a mutable linked list abstraction, with operations for setting a "current position" in the list, for getting the value at this position, and for inserting new elements.

The interface and a partial implementation of the operations are shown in Appendix A and Appendix B. Look at these appendices now to familiarize yourself with the operations provided and their intended behaviors. **Make sure you read both appendices carefully!**

Now, suppose we execute the following top-level commands:

```
let m = empty ()
;; insert_first m 2
;; insert_first m 3
;; insert_first m 4
;; advance m
```

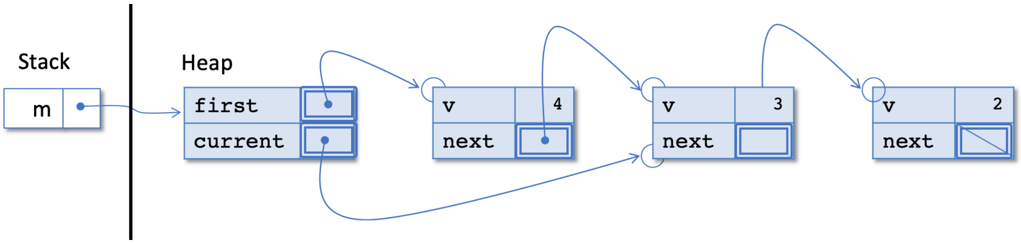
(a) (2 points) After executing these commands, what value will be returned by the expression `current m`?

☐ `None` ☐ `Some 2` ☒ `Some 3` ☐ `Some 4` ☐ error

(b) (2 points) After executing the commands at the top of the page, what value will be returned by the expression `to_list m`?

☐ `[]` ☐ `[3;4]` ☐ `[4;3]` ☐ `[2;3;4]` ☒ `[4;3;2]`

(c) (11 points) Draw the ASM stack and heap after executing the commands at the top of the page.



(d) (8 points) When the programmer first wrote this code, they implemented `empty` like this:

```
let theemptylist : 'a mlist = { first = None; current = None }
let empty () : 'a mlist = theemptylist
```

Write a test case that fails if we use this version of `empty` and succeeds if we use the correct version given in the appendix.

```
;; run_test "test empty" (fun () ->
     let m1 = empty () in
     insert m1 1;
     let m2 = empty () in
     current m2 = None
    )
```

(e) (16 points) Fill in the blanks in the following implementation of the `insert` function.

```
let insert (m: 'a mlist) (x: 'a) : unit =
  begin match m.current with
   None ->
    (* List is empty: insert at the beginning *)
    m.first <- Some { v=x; next=None };
    m.current <- m.first
  | Some c ->
    (* List is nonempty: insert after current *)
    c.next <- Some { v=x; next=c.next };
    m.current <- c.next
  end
```

2. **OCaml ASM and Deques** (12 points)

As we saw in from Homework 4, deques are similar to queues, but with the ability to insert and delete at both ends: they are "double-ended queues." Here is the *representation invariant* for deques, slightly reworded from HW04 and with each clause labeled by a letter:

(a) Either `head` and `tail` are both `None`, or they are both `Some`.

(b) If `head = Some n1` and `tail = Some n2`, then `n2` is reachable from `n1` by following `next` pointers (perhaps zero times—i.e., `n1` and `n2` can be the same node).

(c) If `head = Some n1` and `tail = Some n2`, then `n1.prev = None`.

(d) If `head = Some n1` and `tail = Some n2`, then `n1` is reachable from `n2` by following `prev` pointers (perhaps zero times).

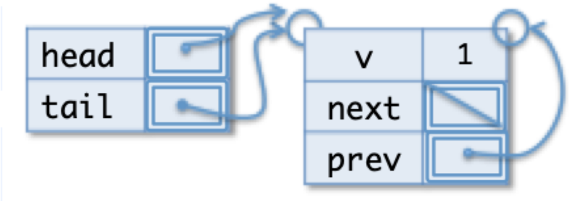
(e) If `head = Some n1` and `tail = Some n2`, then `n2.next = None`.

(f) For every node `n` in the deque:

- if `n.next = Some m` then `m.prev = Some n`, and
- if `n.prev = Some m` then `m.next = Some n`.

For each of the ASM heaps pictured below, indicate whether each part of the invariant is satified (Yes) or not (No).

For example:



(a) Yes ☒ No ☐
(b) Yes ☒ No ☐
(c) Yes ☐ No ☒
(d) Yes ☒ No ☐
(e) Yes ☒ No ☐
(f) Yes ☐ No ☒

(a) Either `head` and `tail` are both `None`, or they are both `Some`.

(b) If `head = Some n1` and `tail = Some n2`, then `n2` is reachable from `n1` by following `next` pointers (perhaps zero times—i.e., `n1` and `n2` can be the same node).

(c) If `head = Some n1` and `tail = Some n2`, then `n1.prev = None`.

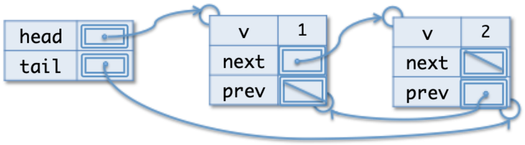
(d) If `head = Some n1` and `tail = Some n2`, then `n1` is reachable from `n2` by following `prev` pointers (perhaps zero times).

(e) If `head = Some n1` and `tail = Some n2`, then `n2.next = None`.

(f) For every node `n` in the deque:

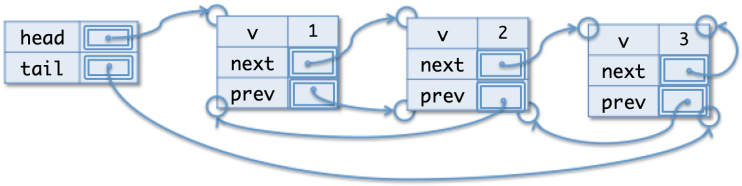
- if `n.next = Some m` then `m.prev = Some n`, and
- if `n.prev = Some m` then `m.next = Some n`.

(i)



(a) ☒ Yes ☐ No
(b) ☒ Yes ☐ No
(c) ☒ Yes ☐ No
(d) ☒ Yes ☐ No
(e) ☒ Yes ☐ No
(f) ☒ Yes ☐ No

(ii)



(a) ☒ Yes ☐ No
(b) ☒ Yes ☐ No
(c) ☐ Yes ☒ No
(d) ☐ Yes ☒ No
(e) ☐ Yes ☒ No
(f) ☐ Yes ☒ No

3. **Program Design: OCaml Objects and Stateful Abstractions**

Step 1: Understand the problem: A *stream* is an abstraction that represents an *infinite* sequence of values. In OCaml, we write the (generic) type of streams as `'a stream`.

Informally, we can write down a stream as a series of values, but, since it is infinite, we can't write down the whole stream. Here are some examples:

- A constant `int stream` of all `0` values:
  `0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...` *(∗ forever ∗)*
- A constant `int stream` of all `42` values:
  `42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 42 ...` *(∗ forever ∗)*
- An interleaved `int stream` of `0`'s and `42`'s:
  `0 42 0 42 0 42 0 42 0 42 0 42 0 42 0 42 0 42 0 42 0 ...` *(∗ forever ∗)*
- A stream of increasing integers:
  `0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 ...` *(∗ forever ∗)*
- An interleaved `int stream` of increasing integers and `42`'s:
  `0 42 1 42 2 42 3 42 4 42 5 42 6 42 7 42 8 42 9 42 10 ...` *(∗ forever ∗)*

Streams, like the widgets from the GUI assignment, are most useful when they encapsulate some state. Also like widgets, there are sensible ways, such as *interleaving*, to combine multiple streams. This problem explores how to implement streams as objects in OCaml.

Step 2: Define the interface Being infinite, it is impossible to compute using all of the elements of a stream at once. Instead, code that uses a stream can ask it to *produce* the next element in the sequence. In fact, the only interesting operation a stream provides is the ability to produce the next element. That leads us to this type definition for stream objects:

```
type 'a stream = { produce : unit -> 'a; }
```

As in the widget library, the stream interface is represented as a *record* containing fields for each method—here, just the method `produce`. Note that the `produce` method can be *stateful*, so successive calls to produce will yield different values.

Given the examples above, we also expect to be able to define these operations for creating streams:

```
val constant_stream : 'a -> 'a stream (* Makes a constant stream *)
val new_increasing : unit -> int stream (* Makes a new stream of increasing integers *)
val interleave : 'a stream -> 'a stream -> 'a stream (* Interleave two streams *)
```

Finally, since we can ask a stream to produce only one element at a time, it is useful to have an operation that "takes" the first n elements from a stream and makes them into a list:

```
val take : int -> 'a stream -> 'a list
```

Steps 3 & 4: Write test cases & Implement As an example, here is a correct implementation of `constant_stream` and a corresponding test case:

```
let constant_stream (x:'a) : 'a stream = {
 produce = (fun () -> x);
}

let test () : bool =
   let s = constant_stream 42 in
   s.produce () = 42 &&
   s.produce () = 42 &&
   s.produce () = 42
 ;; run_test "first three elements of constant 42" test
```

(a) (4 points) An increasing integer stream is just like a `counter` object (as seen in class) except that its method is named "`produce`" instead of "`get`". Consider this code for `new_increasing`, which is intended to generate new increasing stream objects, and a test case:

```
let ctr = { contents = 0 }
let new_increasing () : int stream =
  {
   produce = (fun () ->
      let ans = ctr.contents in
      ctr.contents <- ctr.contents + 1;
      ans);
  }

let test () : bool =
 let s = new_increasing () in
 s.produce () = 0 &&
 s.produce () = 1 &&
 s.produce () = 2 &&
 s.produce () = 3
;; run_test "first four elements of increasing" test
;; run_test "first four elements of increasing" test (* <-- duplicate! *)
```

Notice that the programmer has accidentally duplicated the line that runs the test. Which of the following behaviors would we observe when this program is run?

(Choose one)

☐ Both runs of the test fail.

☒ The first run of the test passes and the second one fails.

☐ The first run of the test fails and the second one passes.

☐ Both runs of the test pass.

(b) (8 points) Here is a (correct) implementation of the `take` operation on streams and one example test case. This version of `take` is written using standard *recursion*:

```
let take (n:int) (s:'a stream) : 'a list =
 let rec loop (i:int) : 'a list =
   if i <= 0 then [] else (s.produce ())::(loop (i-1))
 in
 loop n

let test () : bool =
 let s = constant_stream 42 in
 take 5 s = [42; 42; 42; 42; 42]
;; run_test "take five from the constant 42 stream"
```

Rewrite `take`, by completing the code template below, to use *iteration via tail recursion* instead of plain recursion. Your implementation should use constant stack space and may make use of the library function `List.rev`, which reverses a list. *Hint: you will need to add an extra argument to* `loop`*.*

Answer:

```
let take (n:int) (s:'a stream) : 'a list =
  let rec loop (i:int) (acc:'a list) : 'a list =
    if i <= 0 then List.rev acc else
      loop (i-1) (s.produce () :: acc)
  in
  loop n []
```

(c) (11 points) Now implement the stream combinator `interleave`, which takes two streams and produces a single stream that alternates between producing elements from the first and the second stream (starting with the first). We have given you two test cases, based on the examples from earlier. *Hint: What additional state does interleave need to maintain?*

```
let interleave (s1:'a stream) (s2:'a stream) : 'a stream =
   let first = { contents = true } in
   {
     produce = (fun () ->
         let ans =
           if first.contents then s1.produce () else s2.produce ()
         in
         first.contents <- not (first.contents);
         ans);
   }

let test () : bool =
 take 5 (interleave (constant_stream 0) (constant_stream 42)) =
 [0; 42; 0; 42; 0;]
;; run_test "interleave two streams" test

let test () : bool =
  take 5 (interleave (new_increasing ()) (constant_stream 42)) =
  [0; 42; 1; 42; 2]
;; run_test "interleave ints and 42s streams" test
```

(d) (12 points) Streams, like widgets, can be extended with additional methods. As presented so far, streams can be hard to use because with every call to `produce`, the state of the stream (possibly) changes. One way to fix that is to add a method that lets us "peek" at the next element that will be produced by the stream *without* causing the stream state to advance to the next value. We call such streams "buffered" because they store the next element to produce in a private field, allowing `peek` to be called one or more times, if desired, before `produce` yields the value and advances the stream.

As with the "controller" objects from the GUI assignment, we express the new method as a record and have the constructor function for a buffered stream produce both records. Complete this code for making a buffered stream. We have given you a test that illustrates peek's behavior:

```
type 'a buffered = { peek : unit -> 'a; }

let mk_buffered (s:'a stream) : 'a stream * 'a buffered =
 let buf = {contents = s.produce ()} in
 (
  {
   produce = (fun () ->
       let ans = buf.contents in
       buf.contents <- s.produce ();
       ans)
  },
  {
   peek = (fun () -> buf.contents)
  }
 )

 let test () : bool =
   let (s,b) = mk_buffered (new_increasing ()) in
   b.peek ()    = 0 &&
   b.peek ()    = 0 &&
   b.peek ()    = 0 &&
   s.produce () = 0 &&
   b.peek ()    = 1 &&
   b.peek ()    = 1 &&
   s.produce () = 1 &&
   b.peek ()    = 2
 ;; run_test "peek doesn't change stream; produce updates peek" test
```

4. **Java Basics** (6 points)

   For each of the statement below select the correct option(s):

   (a) Which keyword is used to make a Java variable immutable?

   ☐ `static` ☐ `constant` ☒ `final`

   (b) Which Java type is similar to the `unit` type in OCaml?

   ☒ `void` ☐ `enum` ☐ `null`

   (c) A Java class can implement multiple interfaces.

   ☒ **`true`** ☐ **`false`**

5. **Reference vs. Structural Equality in Java** (8 points)

Consider the following class:

```
public class Tuple{
   private int fst;
   private int snd;

   public Tuple (int fst, int snd){
      this.fst = fst;
      this.snd = snd;
   }

   public boolean equals(Tuple t){
      return this.fst == t.fst && this.snd == t.snd;
   }
}
```

(a) What is the value of `ans` at the end of this program?

```
Tuple t1 = new Tuple(2, 4);
Tuple t2 = new Tuple(2, 4);
boolean ans = (t1 == t2);
```

☐ **`true`** ☒ **`false`**

(b) What is the value of `ans` at the end of this program?

```
Tuple t1 = new Tuple(2, 4);
Tuple t2 = new Tuple(2, 4);
boolean ans = (t1.equals(t2));
```

☒ **`true`** ☐ **`false`**

(c) What is the value of `ans` at the end of this program?

```
Tuple t1 = new Tuple(2, 4);
Tuple t2 = t1;
boolean ans = (t1 == t2);
```

☐ **`true`** ☒ **`false`**

(d) What is the value of `ans` at the end of this program?

```
Tuple t1 = new Tuple(2, 4);
Tuple t2 = t1;
boolean ans = (t1.equals(t2));
```

☒ **`true`** ☐ **`false`**

Feel free to use this page as scratch paper. (If you write anything here that you want us to grade, make sure you <u>clearly</u> indicate this in the answer area earlier in the exam.)

A Appendix: Mutable List Interface

```
module type MLIST = sig

  type 'a mlist

  (* Create a new empty mutable list *)
  val empty : unit -> 'a mlist

  (* Return the value of the current element (or None if the list is empty) *)
  val current : 'a mlist -> 'a option

  (* Reset the "current element" pointer to the beginning of the list *)
  val reset : 'a mlist -> unit

  (* Advance the "current element" pointer one list-element to the
     right, if possible. The values in the list are unchanged. *)
  val advance : 'a mlist -> unit

  (* Insert a new element at the very beginning of the mlist. After
     this operation, the current element pointer points to the newly
     inserted element. *)
  val insert_first : 'a mlist -> 'a -> unit

  (* Insert a new element after the current element. After this
     operation, the current element pointer points to the newly
     inserted element. *)
  val insert : 'a mlist -> 'a -> unit

  (* Convert the whole mlist to a plain list *)
  val to_list : 'a mlist -> 'a list
end
```

B Appendix: Mutable List Implementation

```
module MList : MLIST = struct
  type 'a mlnode =
    { v: 'a;
      mutable next: 'a mlnode option }

  type 'a mlist =
    { mutable first: 'a mlnode option;
      mutable current: 'a mlnode option }

  let empty () : 'a mlist = { first = None; current = None }

  let current (m: 'a mlist) : 'a option =
    begin match m.current with
    | None -> None
    | Some n -> Some n.v
    end

  let reset (m: 'a mlist) : unit =
    m.current <- m.first

  let advance (m: 'a mlist) : unit =
    begin match m.current with
    | None -> ()
    | Some c -> m.current <- c.next
    end

  let insert_first (m: 'a mlist) (x: 'a) : unit =
    m.first <- Some { v=x; next=m.first };
    m.current <- m.first

  let insert (m: 'a mlist) (x: 'a) : unit =
    failwith "implement me!"

  let to_list (m: 'a mlist) : 'a list =
    let rec loop (no: 'a mlnode option) : 'a list =
      begin match no with
      | None -> []
      | Some n -> n.v :: loop n.next
      end in
    loop m.first
end
```