CS 51: Introduction to Computer Programming II
Second Midterm Examination
Spring, 2018

You have 90 minutes to complete this exam.

This is an open-book exam: You are free to use books and notes in preparing your solutions for this exam. However, no electronic devices of any kind may be used.

The exam is in four sections comprised of 17 questions. Numbers in brackets like this ***[nnn points]*** *are the points (out of 75 total) allocated to the problem and may provide a very approximate recommendation for allocating time.*

Write the answers to all problems in the boxes provided. Write with a pen (or a very dark pencil) as we will be scanning your exams for grading. Write clearly, as we can and will only grade what we can unambiguously read. The exam packet is intentionally stapled in the lower left corner to facilitate the scanning process. Do not remove the staple or remove any pages from the exam packet. Anything written outside of the provided boxes will not be graded. If additional room is needed for an answer, make a note in the box provided and write the remainder of your answer in one of the boxes at the back of the examination.

Many of the problems ask you to define something or write code to do something. Throughout the exam, when we ask you to define a value or function or type or class or module, we mean that you should provide a top-level OCaml definition written in well-formed, idiomatic OCaml using the appropriate OCaml definitional construct (`let`, `type`, `class`, `module`, etc.). Except where explicitly noted, you are free to use functions from the standard OCaml libraries such as `Pervasives`, `List`, and `Random`. Your answers will be graded firstly on the well-formedness and correctness of the code, but in keeping with the course's goals, we may also secondarily consider the many other dimensions of code quality – including design and style issues – in evaluating your answers.

To allow for anonymous grading of the exam, please write your name and ID number in the boxes below on this page ***[0.5 points]****, and your ID number (but not your name) in the box provided at the top of all subsequent odd-numbered pages* ***[0.5 points]****.*

By filling in your name and ID number below, you affirm your awareness of the standards of the Harvard College Honor Code.

YOUR NAME:
YOUR HARVARD ID NUMBER:

1. Playing REPL

Suppose you typed the following OCaml expressions into the `ocaml` REPL sequentially. Answer the questions below about the status of the various variables being defined.

```
(1) let p = ref 11 ;;
(2) let r = ref p ;;
(3) let s = ref !r ;;
(4) let t =
      !s := 14;
      !p + !(!r) + !(!s) ;;
(5) let t =
      s := ref 17;
      !p + !(!r) + !(!s) ;;
```

Problem 1. *[2* ***points****] After (1), what is the type of* `p`*?*

Problem 2. *[2* ***points****] After (2), what is the type of* `r`*?*

Problem 3. *[4* ***points****] After (3), which of the following holds? Mark* all *by* clearly *filling in the appropriate box – T for true and F for false.*

(1) T F `p` and `s` have the same type
(2) T F `r` and `s` have the same type
(3) T F `p` and `s` have the same value (in the sense that `p = s` would be `true`)
(4) T F `r` and `s` have the same value (in the sense that `r = s` would be `true`)

Problem 4. ***[2 points]*** *After entering the expressions through (4), what is the value of* `t`*?*

Problem 5. ***[2 points]*** *After entering the expressions through (5), what is the value of* `t`*?*

Now suppose you typed the following OCaml expressions into the `ocaml` REPL sequentially. Answer the questions below about the values of some of the expressions.

(1)
```
class type sample =
  object
    method first : int
    method last : int
    method transform : int -> unit
  end ;;

class top (i : int) : sample =
  object (this)
    val mutable low = i
    val mutable high = i
    method first = low
    method last = high
    method transform inc =
      low <- this#first + inc;
      high <- this#last + this#first
  end ;;

class bottom (i : int) (j : int) : sample =
  object (this)
    inherit top (i + j) as super
    method! last = super#last + j
  end ;;
```

(2) `let o = new bottom 1 3 ;;`

(3) `o#first, o#last ;;`

(4) `o#transform 10 ;;`

(5) `o#first, o#last ;;`

Problem 6. ***[3 points]*** *What is the value of the expression in (3)?*

Problem 7. ***[3 points]*** *What is the value of the expression in (4)?*

Problem 8. ***[3 points]*** *What is the value of the expression in (5)?*

2. Random Walks

A random walk is a mathematical model of the path taken by an entity that takes steps randomly. Imagine you are standing in front of a house at 0 Long Street, an aptly-named street of consecutively numbered houses that proceeds infinitely in both directions. You decide to explore by walking one house left or right, the direction chosen by a random coin flip. Flipping heads, you go left, and find yourself in front of house number −1. After another flip, tails this time, you go right and you're back at 0, then 1, then 0 again, then 1, then 2, and so it goes. (Figure 1 depicts this left and right random walk.) This random process generates an infinite sequence of locations, which we'd like to model as an infinite stream.

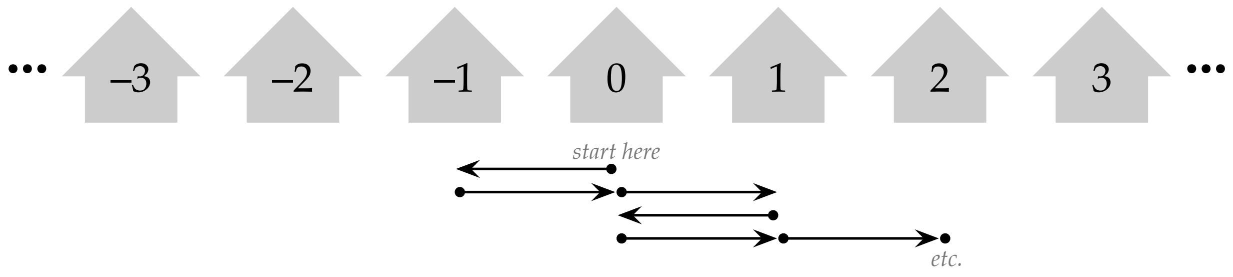

Figure 1. A random walk along Long Street, starting at house 0, and visiting, in order, 0, −1, 0, 1, 0, 1, 2, ...

Start by implementing a simple function `flip : unit -> int` that returns `-1` or `1` randomly with equal probability. For example:

```
# flip () ;;
- : int = -1
# flip () ;;
- : int = -1
# flip () ;;
- : int = 1
```

You may want to use the function `Random.bool : unit -> bool` from the `Random` module, which randomly returns `true` or `false` with equal probability.

Problem 9. *[3* ***points****] Define the function* `flip`*.*

Now, define an `int stream` called `walk`, an infinite stream of integers starting with 0 representing a random walk as described above. The first few elements of the stream might (depending on the random coin flips generated) look like this:

```
# first 20 walk ;;
- : int list = [0; -1; 0; 1; 0; 1; 2; 1; 2; 1; 2; 1; 0; 1; 2; 3; 2; 3; 4; 3]
```

For your reference, Figure 2 on page 10 provides the code for the `NativeLazyStreams` module, which you can assume has been opened for your use.

Here's a hint to think about: Given a random walk stream starting with 0, say, $0, 1, 2, 1, 2, 3, \ldots$, we can extend it one step further by either incrementing or decrementing every element in the stream (that is, adding or subtracting 1) and prepending a zero. For instance, subtracting a 1 gives us the infinite stream $-1, 0, 1, 0, 1, 2, \ldots$, and prepending the zero generates the longer random walk $0, -1, 0, 1, 0, 1, 2, \ldots$. Of course, the decision to increment or decrement needs to be made independently each time the stream is lengthened.

Problem 10. *[5 points] Define `walk : int stream` as described above.*

An interesting question is what is the maximum house number you've ever reached after a certain number of steps of a random walk (that is, how far to the right you've ever reached). You'll define a function `max_reached : int stream -> int -> int` such that `max_reached s n` returns the maximum integer in the first `n` elements in the stream `s`. (We won't be concerned with the value of `max_reached s n` where `n` is zero or less.) For instance,

```
# max_reached walk 10 ;;
- : int = 2
# max_reached walk 100 ;;
- : int = 8
# max_reached walk 1000 ;;
- : int = 17
```

(You'll notice that this maximum grows very slowly, consistent with the theory of random walks.)

Problem 11. *[5 **points**] Define the function `max_reached : int stream -> int -> int`. (No need to worry about efficiency of your code.)*

```
module NativeLazyStreams =
  struct

    type 'a str = Cons of 'a * 'a stream
     and 'a stream = 'a str Lazy.t ;;

    let head (s : 'a stream) : 'a =
      let Cons(h, _t) = Lazy.force s in h ;;

    let tail (s : 'a stream) : 'a stream =
      let Cons(_h, t) = Lazy.force s in t ;;

    let rec first (n : int) (s : 'a stream) : 'a list =
      if n = 0 then []
      else head s :: first (n - 1) (tail s) ;;

    let rec smap (f : 'a -> 'b) (s : 'a stream) : 'b stream =
      lazy (Cons(f (head s), smap f (tail s)));;

    let rec smap2 (f : 'a -> 'b -> 'c)
                  (s1 : 'a stream)
                  (s2 : 'b stream)
                  : 'c stream =
      lazy (Cons(f (head s1) (head s2), smap2 f (tail s1) (tail s2))) ;;

    let rec sfilter (pred : 'a -> bool) (s : 'a stream) : 'a stream =
      lazy (if pred (head s)
            then Cons((head s), sfilter pred (tail s))
            else Lazy.force (sfilter pred (tail s))) ;;

  end
```

Figure 2. The native implementation of lazy streams (from Lab 6)

3. Complexity

Consider the following functions:

$$f(x) = x^3 - x - 4$$
$$g(x) = 5x^2 + 23$$
$$p(x) = min(f(x), g(x))$$
$$q(x) = max(f(x), g(x))$$

Here are some sample values of each of the functions:

x	f	g	p	q
1	−4	28	−4	28
2	0	43	0	43
3	20	68	20	68
4	56	103	56	103
5	116	148	116	148
6	206	203	203	206

Problem 12. *[8 points] Which of the following statements are true or false? Mark* all *by* clearly *filling in the appropriate box – T for true and F for false.*

(1) T F $f \in O(x)$
(2) T F $f \in O(x^2)$
(3) T F $f \in O(x^3)$
(4) T F $f \in O(x^4)$
(5) T F $g \in O(x)$
(6) T F $g \in O(x^2)$
(7) T F $g \in O(x^3)$
(8) T F $g \in O(x^4)$
(9) T F $p \in O(x)$
(10) T F $p \in O(x^2)$
(11) T F $p \in O(x^3)$
(12) T F $p \in O(x^4)$
(13) T F $q \in O(x)$
(14) T F $q \in O(x^2)$
(15) T F $q \in O(x^3)$
(16) T F $q \in O(x^4)$
(17) T F $f \in O(g)$
(18) T F $g \in O(f)$
(19) T F $f \in O(f)$
(20) T F $g \in O(g)$

We've seen algorithms that work by dividing a problem of size n into two equal-sized pieces of size $n/2$, solving them recursively, and combining the results. If the combination can be done in constant time, the complexity of such an algorithm can be modeled by the recurrence equation

$$\begin{aligned} T(n) &= 2 \cdot T(n/2) + c \qquad \text{for } n > 0 \\ T(0) &= c \end{aligned}$$

where c is a constant.

In this problem, however, we'll look at a different recurrence equation, one for algorithms that divide the problem into *unequal* size pieces. Suppose the division is into one piece of size a (a constant) and the remaining piece of size $n-a$. An appropriate recurrence would be

$$T(n) = T(a) + T(n-a) + c \qquad \text{for } n > a$$

(Again, a and c are constants.) For the base case, we can assume that the time required is constant for all problems up to size a:

$$T(n) = c \qquad \text{for } n \leq a$$

Problem 13. ***[8 points]*** *Derive the "big-O" complexity of $T(n)$ as defined by*

$$T(n) = T(a) + T(n-a) + c \qquad \textit{for } n > a$$
$$T(n) = c \qquad \textit{for } n \leq a$$

by unfolding the equations and simplifying, making sure to conclude with a "big-O" characterization of the result. Show your work.

4. A priority queue class

In this section, you will implement an imperative priority queue class, along with an extension of it that can revert (that is, undo) the most recent selection from the priority queue.

For simplicity, we'll stick to integer priority queues, with the highest priority elements being the smallest integers. We provide a class type `prioqueue_t` for integer priority queues:

```
class type prioqueue_t =
  object
    method add : int -> unit    (* add an int to the priority queue *)
    method select : int option (* return the smallest element after
                                  removing it from the priority queue *)
  end ;;
```

Problem 14. *[2 **points**] Why is it appropriate for the `select` method to return an `int option` rather than an `int`?*

Problem 15. *[9 points] Provide an implementation of the* `prioqueue_t` *class type, a* `prioqueue` *class, by filling in the box below. (You can keep the implementation extremely simple. Lists are fine; no need for binary trees or heaps.)*

```
class prioqueue : prioqueue_t =
  object (this)



  end ;;
```

Next, we want to augment the priority queue implementation with an extra method `revert`, which when invoked adds back onto the priority queue the last element (if any) that was selected. (If nothing had ever been selected, the priority queue is left unchanged.) Once an element is reverted back onto the priority queue, it should no longer be available to be reverted again; that is, a second revert leaves the priority queue unchanged. Here is an example of the behavior of the `prioqueue_revert` class:

```
# let p = new prioqueue_revert ;;
val p : prioqueue_revert = <obj>
# let _ = p#add 5;
          p#add 2;
          p#add 4 ;;
- : unit = ()
# p#select ;;
- : int option = Some 2
# p#revert ;;
- : unit = ()
# p#select ;;
- : int option = Some 2
# p#select ;;
- : int option = Some 4
# p#revert ;;
- : unit = ()
# p#select ;;
- : int option = Some 4
# p#select ;;
- : int option = Some 5
# p#revert ;;
- : unit = ()
# p#revert ;;
- : unit = ()
# p#select ;;
- : int option = Some 5
# p#select ;;
- : int option = None
# p#revert ;;
- : unit = ()
# p#select ;;
- : int option = None
```

Problem 16. *[5 **points**] Provide a class type definition* `prioqueue_revert_t` *that extends* `prioqueue_t` *with the* `revert` *method.*

Problem 17. *[8 points] Define a class* `prioqueue_revert` *that satisfies* `prioqueue_revert_t`*.*

End of exam.

Total points: 75

Extra space for answers

Reference this area with “See box 1”

Reference this area with “See box 2”

Reference this area with "See box 3"

Reference this area with "See box 4"