CS 51: Abstraction and Design in Computation
Second Midterm Examination
Spring, 2019

- *You have 90 minutes to complete this exam.*
- *This is a closed-book exam. However, you are free to use up to 10 letter-size pages of notes or other printed materials in preparing your solutions for this exam. No electronic devices of any kind may be used.*
- *The exam is in six sections comprised of 19 questions. Numbers in brackets like this* ***[nnn points]*** *are the points (out of 60 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.). 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.***

1. Short answer questions

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

```
1 let x = ref 42 ;;
2 let y = x ;;
3 let z = y := 33 ;;
4 let f () = !x + !y ;;
5 let y = ref 0 ;;
6 y := (x := 22; y := 23; f ()) ;;
7 let z = !y ;;
8 let y = f () ;;
```

Problem 1. ***[2 points]*** *What is the type of `y` after line 2?*

Solution: `int ref`

```
# let x = ref 42 ;;
val x : int ref = {contents = 42}
# let y = x ;;
val y : int ref = {contents = 42}
```

Problem 2. ***[2 points]*** *What is the type of `z` after line 3?*

Solution: `unit`

```
# let z = y := 33 ;;
val z : unit = ()
```

Problem 3. ***[2 points]*** *What is the type of `f` after line 4?*

Solution: `unit -> int`

```
# let f () = !x + !y ;;
val f : unit -> int = <fun>
```

Problem 4. ***[2 points]*** *What is the value of `z` after line 7?*

Solution: `44`

```
# let y = ref 0 ;;
val y : int ref = {contents = 0}
# y := (x := 22; y := 23; f ()) ;;
- : unit = ()
# let z = !y ;;
val z : int = 44
```

Problem 5. ***[2 points]*** *What is the value of `y` after line 8?*

Solution: `44`

```
# let y = f () ;;
val y : int = 44
```

2. Fill in the blank

Problems 6 through 10 each present an OCaml expression containing a blank. For each one, provide a single well-formed OCaml expression such that when inserted into the blank, the expression as a whole always evaluates to the boolean value `true`. The first problem has been done for you as an example.

(For the last two problems, recall that `Random.int n` returns a different random integer between `0` and `n - 1` each time it is called.)

Problem 6.

```
3 > _____
```

Problem 7. ***[2 points]***

```
______ (3 > 4) (4 > 3)
```

> **Solution:** The key idea here is that what fills the blank is a curried binary function. Lots of things work. Simplest is for the function to ignore its arguments and just return true: `fun x y -> true`. If a built-in operator is desired, logical or `(||)` works. The parentheses are required in the concrete syntax.

Problem 8. ***[3 points]***

```
let x = ref 1 in
while ______ do
  ()
done;
!x <> 1 ;;
```

> **Solution:** The expression `x := 2; false` works. Yes, that counts as a single expression. Of course, the 2 can be replaced by any integer that isn't 1.
>
> ```
> # let x = ref 1 in
> # while x := 2; false do
> #   ()
> # done;
> # !x <> 1 ;;
> - : bool = true
> ```

Problem 9. ***[3 points]***

```
let x = Random.int 100 in
let f y = ______ in
if f x then x > 50 else x <= 50 ;;
```

Solution: The expression `y > 50` works.

```
# let x = Random.int 100 in
# let f y = y > 50 in
# if f x then x > 50 else x <= 50 ;;
- : bool = true
```

Problem 10. ***[3 points]***

```
let x = ref (Random.int 100) in
let f y = ______ in
if f x then !x > 50 else !x < 50 ;;
```

Solution: The expression `y := 60; true` suffices, as does any other similar expresson that sets y to a value greater than 50.

```
# let x = ref (Random.int 100) in
# let f y = y := 60; true in
# if f x then !x > 50 else !x < 50 ;;
- : bool = true
```

The approach in the previous problem, however, does not work. Filling the blank with `!y > 50` fails when y happens to be `50` exactly. (Notice the change in the problem from <= to <.)

3. COMPLEXITY

Consider this function `repeatedly` that repeatedly applies its argument function:

```
# let rec repeatedly f n =
#   if n <= 0 then 1
#   else f (repeatedly f (n-1)) + f (repeatedly f (n-1)) ;;
val repeatedly : (int -> int) -> int -> int = <fun>
```

For example:

```
# repeatedly succ 4 ;;
- : int = 46
# repeatedly succ 20 ;;
- : int = 3145726
```

Problem 11. ***[2 points]*** *What is the value of the expression*

```
repeatedly (fun x -> x) 2
```

?

Solution:

```
# repeatedly (fun x -> x) 2 ;;
- : int = 4
```

Problem 12. ***[2 points]*** *What is the time complexity $T_{rep}(n)$ of* `repeatedly` *in terms of its argument* `n`*? You should assume that the function* `f` *has constant time complexity (that is, $O(1)$). Mark the single appropriate box giving the* tightest *big-O bound for T_{rep}, by* clearly *and* darkly *filling in the* entire *box like this:* ■.

(1) ☐ $T \in O(n)$
(2) ☐ $T \in O(n \log n)$
(3) ☐ $T \in O(2^n)$
(4) ☐ $T \in O(n^2)$
(5) ☐ $T \in O(n^2 \log n)$
(6) ☐ $T \in O(n^3)$
(7) ☐ $T \in O(\log n)$
(8) ☐ $T \in O(3^n)$
(9) ☐ None of the above.

Solution: Since $T_{rep} \in O(2^n)$, only the 2^n line should be checked.

Now consider this alternative definition of `repeatedly`, which stores a call to `f` in a temporary variable and uses it twice.

```
# let rec repeatedly f n =
#   if n <= 0 then 1
#   else let temp = f (repeatedly f (n-1)) in
#        temp + temp ;;
val repeatedly : (int -> int) -> int -> int = <fun>
```

Problem 13. ***[2 points]*** *What is the time complexity of this version of* `repeatedly`*? Again, mark the single appropriate box giving the* tightest *big-O bound for T_{rep}, by* clearly *and* darkly *filling in the* entire *box like this:* ■.

(1) ☐ $T \in O(n)$
(2) ☐ $T \in O(n \log n)$
(3) ☐ $T \in O(2^n)$
(4) ☐ $T \in O(n^2)$
(5) ☐ $T \in O(n^2 \log n)$
(6) ☐ $T \in O(n^3)$
(7) ☐ $T \in O(\log n)$
(8) ☐ $T \in O(3^n)$
(9) ☐ None of the above.

Solution: It's linear.

Problem 14. ***[3 points]*** *Are there cases in which the two versions of* `repeatedly` *above would return different values when applied to identical arguments? If so, explain how such a case could arise. If not, explain why no such example can exist.*

Solution: In a pure language, evaluating an expression twice or evaluating it once and using its value twice are equivalent. But we can easily construct an impure example where the two definitions will differ. Here's one, where the function bumps a counter each time it's called. Since it's called fewer times with the second definition, the results differ.

```
# let rec repeatedly f n =
#   if n <= 0 then 1
#   else f (repeatedly f (n-1)) + f (repeatedly f (n-1)) ;;
val repeatedly : (int -> int) -> int -> int = <fun>

# let rec repeatedly2 f n =
#   if n <= 0 then 1
#   else let temp = f (repeatedly2 f (n-1)) in
#        temp + temp ;;
val repeatedly2 : (int -> int) -> int -> int = <fun>

# repeatedly (let c = ref 1 in
#      fun _x -> c := succ !c; !c) 1 ;;
```

```
- : int = 5

# repeatedly2 (let c = ref 1 in
#      fun _x -> c := succ !c; !c) 1 ;;
- : int = 4
```

4. MULTIPLE STREAMS

The next few questions concern streams, as defined by the `NativeLazyStreams` module that you've used in lab and problem sets. For your reference, Figure 1 on page 15 provides the code for the `NativeLazyStreams` module, which you can assume has been opened for your use.

Recall the definition of the `nats` stream of natural numbers:

```
# let rec nats =
#   lazy (Cons (0, smap succ nats)) ;;
val nats : int stream = <lazy>
# first 10 nats ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

In this section of the exam, we ask you to provide definitions of a different stream, `threeby2`, whose elements start with 3 and increase by 2 thereafter. Thus the definition should generate the following behavior:

```
# first 10 threeby2 ;;
- : int list = [3; 5; 7; 9; 11; 13; 15; 17; 19; 21]
```

We ask you to provide *two* substantively different definitions of `threeby2`. By "substantively different", we mean that the differences are not merely matters of concrete syntax or trivial changes in abstract syntax (such as simple restructuring of arithmetic expressions). You can make use of anything in the `NativeLazyStreams` module and the `nats` stream defined above.

Problem 15. ***[6 points]*** *Provide two substantively different definitions of the stream* `threeby2`*. Place each in a separate box on the next page.*

Solution: Some "substantively distinct" possibilities:

```
# let threeby2 = lazy (Cons (3, smap ((+) 2) threeby2)) ;;
val threeby2 : int stream_internal lazy_t = <lazy>
# first 5 threeby2 ;;
- : int list = [3; 5; 7; 9; 11]
# let threeby2 = nats |> smap (( * ) 2) |> smap ((+) 3) ;;
val threeby2 : int stream = <lazy>
# first 5 threeby2 ;;
- : int list = [3; 5; 7; 9; 11]
# let threeby2 = nats |> smap (fun x -> x * 2 + 3) ;;
val threeby2 : int stream = <lazy>
# first 5 threeby2 ;;
- : int list = [3; 5; 7; 9; 11]
#   let threeby2 = nats |> sfilter (fun x -> x >= 3 && x mod 2
# <> 0) ;;
val threeby2 : int stream = <lazy>
# first 5 threeby2 ;;
- : int list = [3; 5; 7; 9; 11]
# let threeby2 =
#   let rec helper x = lazy (Cons(x, helper (x + 2))) in
#   helper 3 ;;
val threeby2 : int stream = <lazy>
# first 5 threeby2 ;;
- : int list = [3; 5; 7; 9; 11]
```

Some options that are not too distinct from one of the above:

```
# let threeby2 = (tail (tail (tail nats)))
#                |> sfilter (fun x -> x mod 2 <> 0) ;;
val threeby2 : int stream = <lazy>
# first 5 threeby2 ;;
- : int list = [3; 5; 7; 9; 11]
```

5. FISH STORIES

It's a fish-eat-fish world out there. You're tasked with implementing a simulation of fish population dynamics. In this simulation, fish are represented by objects of class type `fish`. The `fish` class type allows for methods to set and to get an integer weight (`set_weight n` and `get_weight`) and a method (`get_level`) that returns an integer that represents the fish's level in the food chain. (As you'll see in Problem 17, fish at lower levels of the food chain are at risk of being eaten by those at higher levels.)

The class type `fish` appropriate for fish objects is defined as

```
class type fish =
  object
    method get_level : int
    method get_weight : int
    method set_weight : int -> unit
  end ;;
```

In the simulation, we'll have guppies (a level 1 fish with initial weight 1) and trout (a level 2 fish with initial weight 10).

Problem 16. ***[8 points]*** *Define classes (`guppy` and `trout`) for these two kinds of fish. (You'll want to use the `fish` class type appropriately. You may of course define other OCaml constructs that aid in defining these classes in keeping with the various course edicts.) Your code should allow for the following behavior:*

Solution: Without the use of inheritance, there'll be a lot of redundancy.

```
# class generic (level : int) (init_weight : int) : fish =
# object
#   val food_level = level
#   val mutable weight = init_weight
#   method get_level = food_level
#   method get_weight = weight
#   method set_weight w = weight <- w
# end ;;
class generic : int -> int -> fish

# class guppy : fish =
# object
#   inherit generic 1 1
# end ;;
class guppy : fish

# class trout : fish =
# object
```

```
      #   inherit generic 2 10
      # end ;;
      class trout : fish
# let a_guppy = new guppy ;;
val a_guppy : guppy = <obj>
# let a_trout = new trout ;;
val a_trout : trout = <obj>
# a_guppy#get_level ;;
- : int = 1
# a_trout#get_weight ;;
- : int = 10
# a_trout#set_weight 15;;
- : unit = ()
# a_trout#get_weight ;;
- : int = 15
```

Problem 17. ***[6 points]*** *Define a function* `attack : fish -> fish -> unit` *that implements the phenomonen of one fish attacking another. If the fish are at different levels, the lower-level fish transfers all its weight to the upper-level fish. For instance,*

Solution: Here's one way.

```
# let rec attack (large : fish) (small : fish) : unit =
#   match (compare large#get_level small#get_level) with
#   | -1 -> attack small large
#   |  0 -> ()
#   |  1 -> large#set_weight
#             (large#get_weight + small#get_weight);
#           small#set_weight 0
#   |  _ -> failwith "Can't happen" ;;
val attack : fish -> fish -> unit = <fun>
```

Other fine possibilities: Use nested conditionals; don't bother with the catchall case; don't do the recursive call trick to reorder (which may lead to redundant code for the two inequality cases unless an auxiliary function is used).

```
# attack a_guppy a_trout ;;
- : unit = ()
# a_guppy#get_weight ;;
- : int = 0
# a_trout#get_weight ;;
- : int = 16
```

6. SEMANTICS

Problem 18. ***[6 points]*** *Show the derivation of the value of the expression* `x := 3; !y` *in an environment* $E_0 = \{\texttt{x} \mapsto l_1; \texttt{y} \mapsto l_1\}$ *and store* $S_0 = \{l_1 \mapsto 1\}$. *That is, complete the derivation that begins*

$$\{\texttt{x} \mapsto l_1; \texttt{y} \mapsto l_1\}, \{l_1 \mapsto 1\} \vdash \texttt{x := 3; !y} \Downarrow \cdots$$

For convenience in writing out your answer, you can use the abbreviations E_0 and S_0 above, for instance,

$$E_0, S_0 \vdash \texttt{x := 3; !y} \Downarrow \cdots \quad ,$$

and you can define any other abbreviations that would be convenient as well.

For your reference, we've provided the rules for the lexical environment semantics with store in Figures 2 and 3 on pages 16 and 17.

Solution: We define the abbreviation $S_1 = \{l_1 \mapsto 3\}$.

$E_0, S_0 \vdash \texttt{x := 3; !y} \Downarrow$

- $E_0, S_0 \vdash \texttt{x := 3} \Downarrow$
  - $E_0, S_0 \vdash \texttt{x} \Downarrow l_1, S_0 \quad \longleftarrow (R_{var})$
  - $E_0, S_0 \vdash \texttt{3} \Downarrow 3, S_0 \quad \longleftarrow (R_{num})$
  - $\Downarrow (), S_1 \quad \longleftarrow (R_{assign})$
- $E_0, S_1 \vdash \texttt{!y} \Downarrow$
  - $E_0, S_1 \vdash \texttt{y} \Downarrow l_1, S_1 \quad \longleftarrow (R_{var})$
  - $\Downarrow 3, S_1 \quad \longleftarrow (R_{deref})$
- $\Downarrow 3, S_1 \quad \longleftarrow (R_{seq})$

Problem 19. ***[3 points]*** *Construct an expression whose evaluation derivation includes as a subderivation the one you've just constructed for the expression* `x := 3; !y` *in the context of the environment* E_0 *and store* S_0 *from Problem 18.*

Solution: The expression has `x := 3; !y` as a subexpression, and must establish an environment in which `x` and `y` both exist and are aliased to the same location with contents 1. The following suffices:

```
# let x = ref 1 in
# let y = x in
# x := 3; !y ;;
- : int = 3
```

End of exam.

```
(*
                              CS51 Lab 15
                          Native Lazy Streams

   A native implementation of lazy streams with some useful functions.
 *)

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

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

let tail (s : 'a stream) : 'a stream =
  let Cons (_, 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))) ;;
```

FIGURE 1. The native implementation of lazy streams, reproduced here from Lab 15.

$$E, S \vdash \overline{n} \Downarrow \overline{n}, S \qquad (R_{int})$$

$$E, S \vdash x \Downarrow E(x), S \qquad (R_{var})$$

$$E, S \vdash \texttt{fun}\ \ x\ \ \texttt{->}\ \ P \Downarrow [E \vdash \texttt{fun}\ \ x\ \ \texttt{->}\ \ P], S \qquad (R_{fun})$$

$$\begin{array}{l} E, S \vdash P\ \ \texttt{+}\ \ Q \Downarrow \\ \qquad \left| \begin{array}{l} E, S \vdash P \Downarrow \overline{m}, S' \\ E, S' \vdash Q \Downarrow \overline{n}, S'' \end{array} \right. \\ \qquad \Downarrow \overline{m+n}, S'' \end{array} \qquad (R_{+})$$

(and similarly for other binary operators)

$$\begin{array}{l} E, S \vdash \texttt{let}\ \ x\ \ \texttt{=}\ \ D\ \ \texttt{in}\ \ B \Downarrow \\ \qquad \left| \begin{array}{l} E, S \vdash D \Downarrow v_D, S' \\ E\{x \mapsto v_D\}, S' \vdash B \Downarrow v_B, S'' \end{array} \right. \\ \qquad \Downarrow v_B, S'' \end{array} \qquad (R_{let})$$

$$\begin{array}{l} E, S \vdash \texttt{let rec}\ \ x\ \ \texttt{=}\ \ D\ \ \texttt{in}\ \ B \Downarrow \\ \qquad \left| \begin{array}{l} E\{x \mapsto \texttt{let rec}\ \ x\ \ \texttt{=}\ \ D\ \ \texttt{in}\ \ x\}, S \vdash D \Downarrow v_D, S' \\ E\{x \mapsto v_D\}, S' \vdash B \Downarrow v_B, S'' \end{array} \right. \\ \qquad \Downarrow v_B, S'' \end{array} \qquad (R_{letrec})$$

$$\begin{array}{l} E_d, S \vdash P\ \ Q \Downarrow \\ \qquad \left| \begin{array}{l} E_d, S \vdash P \Downarrow [E_l \vdash \texttt{fun}\ \ x\ \ \texttt{->}\ \ B], S' \\ E_d, S' \vdash Q \Downarrow v_Q, S'' \\ E_l\{x \mapsto v_Q\}, S'' \vdash B \Downarrow v_B, S''' \end{array} \right. \\ \qquad \Downarrow v_B, S''' \end{array} \qquad (R_{app})$$

FIGURE 2. Lexical environment semantics rules for evaluating expressions, for a functional language with naming and arithmetic (part 1). The remaining rules are shown in Figure 3.

$$E, S \vdash \texttt{ref}\ P \Downarrow$$
$$\Big|\ E, S \vdash P \Downarrow v_P, S' \qquad (R_{ref})$$
$$\Downarrow l, S'\{l \mapsto v_P\} \qquad \text{(where } l \text{ is a new location)}$$

$$E, S \vdash\ !\ P \Downarrow$$
$$\Big|\ E, S \vdash P \Downarrow l, S' \qquad (R_{deref})$$
$$\Downarrow S'(l), S'$$

$$E, S \vdash P\ \texttt{:=}\ Q \Downarrow$$
$$\Big|\ \begin{array}{l} E, S \vdash P \Downarrow l, S' \\ E, S' \vdash Q \Downarrow v_Q, S'' \end{array} \qquad (R_{assign})$$
$$\Downarrow (), S''\{l \mapsto v_Q\}$$

$$E, S \vdash P\ \texttt{;}\ Q \Downarrow$$
$$\Big|\ \begin{array}{l} E, S \vdash P \Downarrow (), S' \\ E, S' \vdash Q \Downarrow v_Q, S'' \end{array} \qquad (R_{seq})$$
$$\Downarrow v_Q, S''$$

FIGURE 3. Lexical environment semantics rules for evaluating expressions, for a functional language with naming and arithmetic (part 2). The remaining rules are shown in Figure 2.

EXTRA SPACE FOR ANSWERS