CS 51: Introduction to Computer Programming II
Second Midterm Examination
Spring, 2017

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. We have provided a separate handout with reference material – a cheat sheet on OCaml – which you need not turn in.

The exam is in three sections comprised of 9 questions. Numbers in brackets like this ***[nnn points]*** *are the points (out of 65 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 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]****.*

Your name:
Your Harvard ID number:

1. Mutability

Problem 1. ***[12 points]*** *For each of the following expressions, give its type if any or specify "doesn't type". If its type is* `int` *give its value. The first problem is done for you as an example.*

Some of the expressions make reference to and depend on earlier ones. Some of the expressions make use of the mutable list type and functions:

Warning: The definition of mutable lists defined and used in this section is different from that in the version of the textbook and labs as of spring 2023.

```
type 'a mlist =
      | Nil
      | Cons of 'a * ('a mlist ref) ;;

let mhead (Cons(hd, _tl)) = hd ;;
let mtail (Cons(_hd, tl)) = !tl ;;
```

(1)
```
let x = 3 in x * 2 ;;
```

Type (if any):	*Value (if integer)*
int	*6*

(2)
```
let a = ref 3 in
let b = ref 5 in
let a = ref b in
!(!a) ;;
```

Type (if any):	*Value (if integer)*

(3)
```
let rec a, b = ref b, ref a in
!a ;;
```

Type (if any):	*Value (if integer)*

(4)
```
let a = ref 1 in
let b = ref a in
let a = ref 2 in
!(!b) ;;
```

Type (if any): *Value (if integer)*

(5)
```
let a = 2 in
let f = (fun b -> a * b) in
let a = 3 in
f (f a) ;;
```

Type (if any): *Value (if integer)*

(6)
```
let a = Cons(2, ref (Cons(3, ref Nil))) ;;
```

Type (if any): *Value (if integer)*

(7)
```
let Cons(h, t) = a in
let b = Cons(1, ref a) in
t := b;
mhead (mtail (mtail b)) ;;
```

Type (if any): *Value (if integer)*

Problem 2. ***[6 points]*** *Which if any of the expressions in Problem 1 above would evaluate to a different value if OCaml were dynamically scoped instead of lexically scoped? Check all that apply.*

Expression	(2)	(3)	(4)	(5)	(6)	(7)
	☐	☐	☐	☐	☐	☐

2. Perfect squares and streams

A perfect square is an integer that is itself the square of an integer, for example, 1, 4, 9, 16, 25. In this section, you'll generate streams of perfect squares in various ways.

Problem 3. ***[5 points]*** *Write a function* `is_square : int -> bool` *that returns* `true` *if and only if its integer argument is a perfect square. You may find one or more of the Pervasives module functions, as listed on the cheatsheet, to be useful.*

Now recall the definition of the `'a stream` type and associated functions from the implementation of streams using OCaml's native Lazy module. For your reference, we've reproduced a version as Figure 1. It can be used, for instance, to generate a stream of the natural numbers starting with a given integer:

```
let rec natsfrom (n : int) : int stream =
  lazy (Cons(n, natsfrom (n + 1))) ;;
```

Using this function, the `nats` stream from lecture and lab can be implemented simply as

```
let nats : int stream = natsfrom 0 ;;
```

```
type 'a str = Cons of 'a * 'a stream
 and 'a stream = 'a str 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

Problem 4. ***[4 points]*** *Write a recursive function* `squaresfrom : int -> int stream` *that takes an integer argument and returns a stream of all of the perfect squares starting with the square of its integer argument. For this part,* do not *use (or directly reimplement) any of the stream functions from Figure 1. The function should have the following behavior:*

```
# first 10 (squaresfrom 1) ;;
- : int list = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
# first 10 (squaresfrom 10) ;;
- : int list = [100; 121; 144; 169; 196; 225; 256; 289; 324; 361]
```

Problem 5. ***[4 points]*** *Write another version of* `squaresfrom` *as in Problem 4 to construct a stream of perfect squares, but this time a* nonrecursive *version that uses* `sfilter` *to filter out non-squares from an appropriate stream.*

Problem 6. ***[6 points]*** *Is one of your two implementations of* `squaresfrom` *more time-efficient than the other in generating the first n perfect squares? If so, which? Why? Couch your argument*

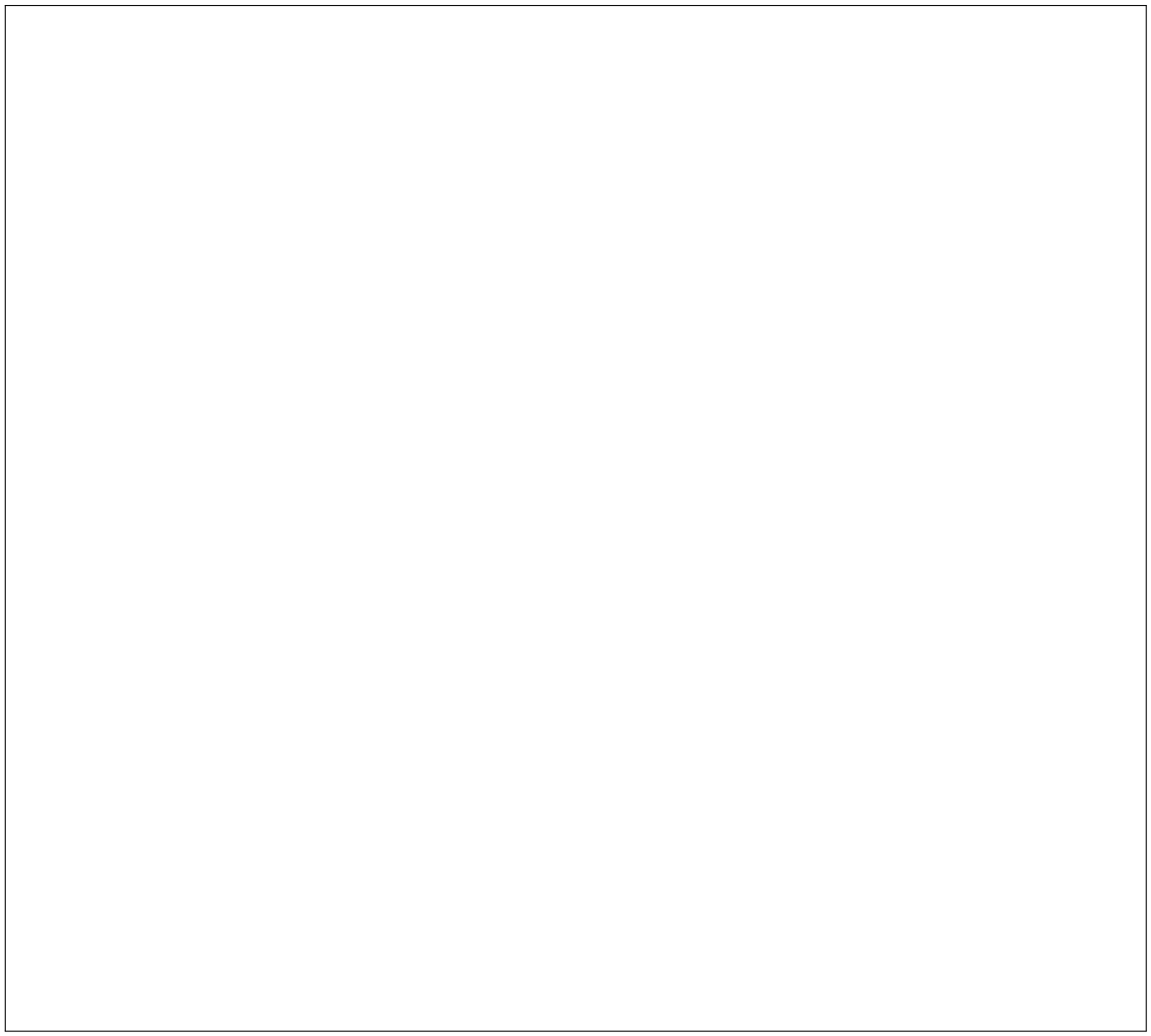
in terms of the time complexity of the functions. (You can assume that built-in OCaml operations over ints and floats take constant time.)

3. Complexity and recurrences

Problem 7. ***[5 points]*** *Consider the following recurrence equations:*

$$T(1) = c$$
$$T(n) = T(n/2) + d$$

where c and d are constants. Derive the "big-O" complexity of $T(n)$ by unfolding the equations and simplifying. Show your work.



4. Radio buttons and checkboxes

Graphical user interfaces (GUIs) often make use of graphical controls or "widgets" to indicate the state of a binary variable, a variable that can be on or off, represented by `true` or `false` values, respectively. Among these are so-called "checkboxes" and "radio buttons". (You can see some examples in Figure 2.)

A checkbox (like those at the bottom of the figure) is used for controlling a binary variable by toggling its state from `false` to `true` or `true` to `false` when clicked.

Radio buttons come in "groups". A radio button is used for controlling a binary variable by setting it to `true` when clicked; all the other radio buttons in its associated group are then set to `false`. For instance, within the group of three "Position on screen" radio buttons, clicking on the "Left" radio button would set it to true, and set the other two radio buttons to false. (For purposes of this exam, we'll say that clicking on a radio button that is already set to `true` leaves its state and all others in its group unchanged.)

Since both kinds of binary state controls share some similar behavior, we might want to implement them in an object-oriented manner by classes with inheritance.

We start with a class type for binary widgets with methods for getting the state of the widget, setting it to true or false, clicking on the widget, and rendering the widget graphically.

```
class type binary_widget_type =
object
  method get_state : bool (* returns the state of the widget *)
  method set : unit       (* set the state to true *)
  method clear : unit     (* set the state to false *)
  method on_click : unit  (* perform action when clicked *)
  method render : unit    (* show the widget state *)
end
```

Here's a class `binary_widget` implementing the `binary_widget_type` class type, suitable for both checkboxes and radio buttons to inherit from:

```
class binary_widget (initial : bool) : binary_widget_type =
object
  val mutable state = initial
  method get_state = state
  method set = state <- true
  method clear = state <- false
  method render = if state then print_string "[x]"
    else print_string "[ ]"
  method on_click = ()
end
```

(For purposes of this exam, the rendering just involves printing out a textual indicator of the value rather than the sort of graphical indication that a real widget would use.)

Finally, here's a `checkbox` class for checkboxes that inherits from `binary_widget`.

```
class checkbox (initial : bool) =
object (this)
  inherit binary_widget initial
```

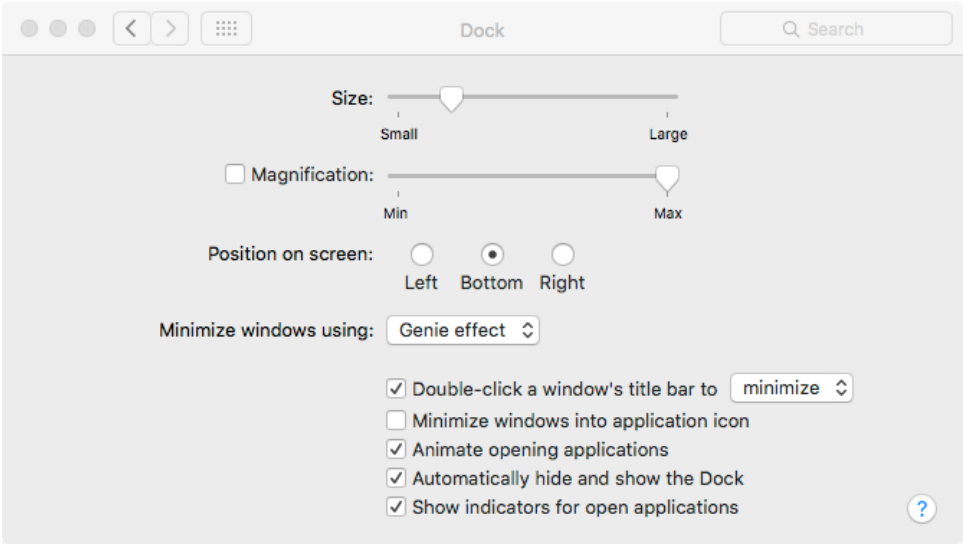
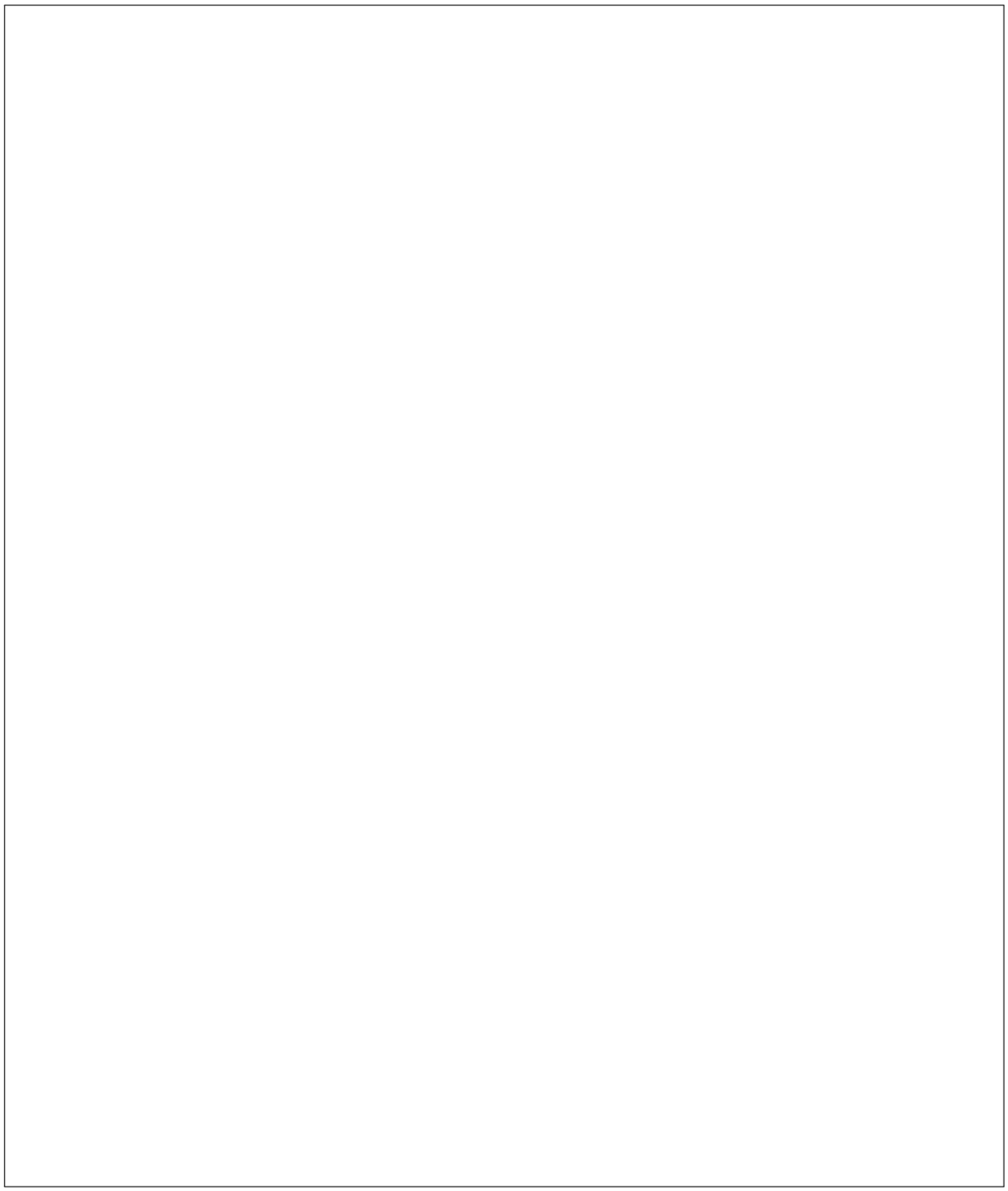


FIGURE 2. The MacOS "Dock" preferences pane showing various widgets including some radio buttons and checkboxes.

```
  method toggle =
    if this#get_state then this#clear else this#set
  method! on_click = this#toggle
end ;;
```

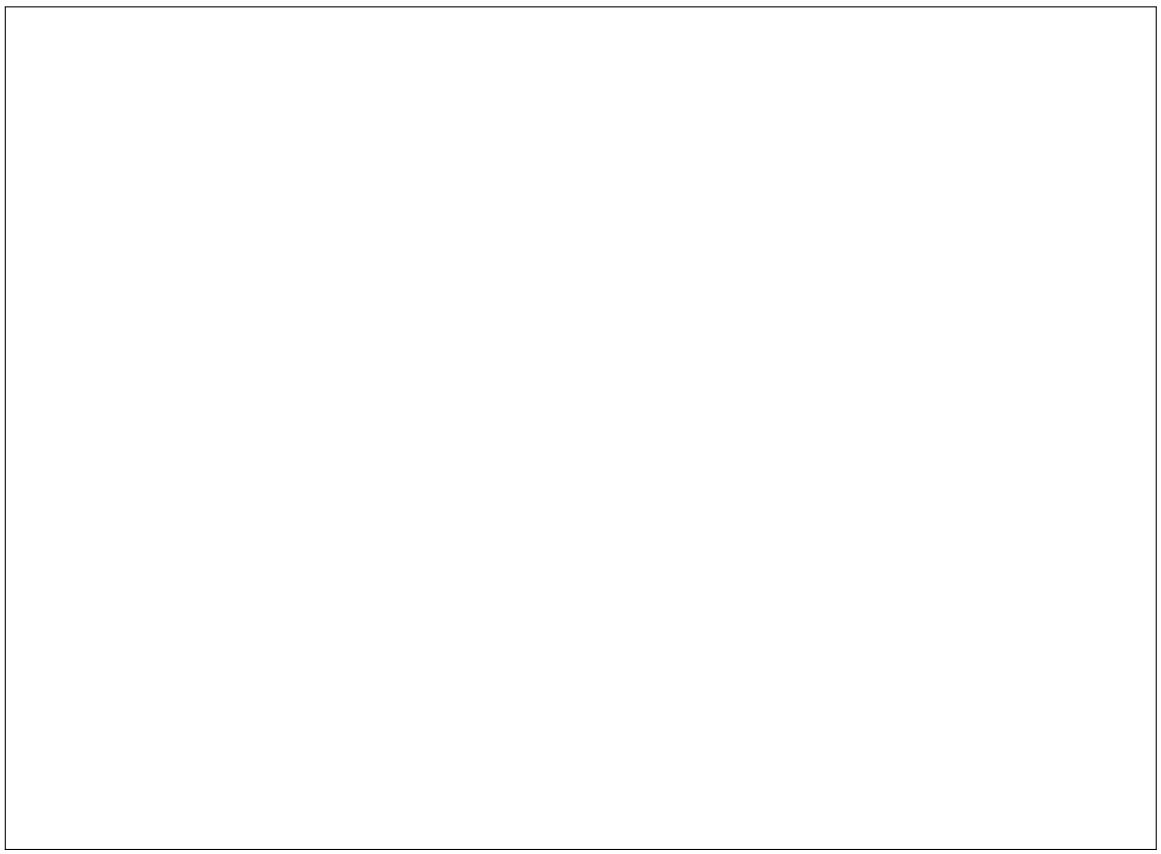
Problem 8. ***[16 points]*** *Implement a* `radio_button` *class that inherits from* `binary_widget`*. (We've gotten things started for you on the next page.) Objects in the class should have a mutable field* `others` *to store a list of the other radio buttons in this radio button's group, initially the empty list, as well as a method* `set_others` *that takes as argument a list of radio buttons to store in the* `others` *field. The class should also provide for appropriate methods of the* `binary_widget_type`*, noting especially that the* `set` *method should clear all of the other widgets in this widget's group.*

```
class radio_button (initial : bool) =
  object (this)
```



```
end
```

Problem 9. ***[6 points]*** *So far, there is no way to combine a group of radio buttons together to form a group. Write a function* `group : radio_button list -> unit` *that takes a list of radio buttons and sets the* `others` *field for each to be the others in the list.*



End of exam.

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”