CS 51: Introduction to Computer Programming II
Final Examination
Spring, 2016

We recommend that you read all instructions carefully and look over the exam in its entirety before beginning the 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 and the lab code for `NativeLazyStreams`*.*

The exam is in five sections comprising 26 problems. Numbers in brackets are the points (out of 100 total) allocated to the problem and may provide an approximate recommendation for allocating time.

Write the answers to all problems in the boxes provided. 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 extra boxes at the back of the examination paper.

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.).*

Please write your name and ID number clearly in the boxes below, but nowhere else on the exam to maintain anonymity of grading.

YOUR NAME:
YOUR HARVARD ID NUMBER:

1. Circuits and Boolean streams

A Boolean circuit is a device with one or more inputs and a single output that receives over time a sequence of Boolean values on its inputs and converts them to a corresponding sequence of Boolean values on its output. The building blocks of circuits are called *gates*. For instance, the *and* gate is a Boolean device with two inputs; its output is `true` when its two inputs are both `true`, and `false` if either output is false. The *not* gate is a Boolean device with a single input; its output is `true` when its input is `false` and vice versa.

In this problem, you'll generate code for modeling Boolean circuits. The inputs and outputs will be modeled as *lazy Boolean streams*. For your reference, we've provided the pertinent code for the `NativeLazyStreams` module from lab in the separate reference material handout. You can (and should) make use of that module as if it had been opened (so no module name prefix is required).

Let's start with an infinite stream of false values.

Problem 1. *[4 points] Define a value* `falses` *to be an infinite stream of the Boolean value* `false`*.*

Problem 2. *[1 points] What is the type of* `falses`*?*

A useful function is the `trueat` function. The expression `trueat n` generates a stream of values that are all `false` except for a single `true` at index `n`:

```
# first 5 (trueat 1) ;;
- : bool list = [false; true; false; false; false]
```

You need not write this function; you should just assume you have access to this function, as it is used in demonstrating the functions described below.

Problem 3. *[4 points] Define a function* `circnot : bool stream -> bool stream` *to represent the* not *gate. It should have the following behavior:*

```
# first 5 (circnot falses) ;;
- : bool list = [true; true; true; true; true]
```

Problem 4. *[4 points] Define a function* `circand` *to represent the* and *gate. It should have the following behavior:*

```
# first 5 (circand (circnot (trueat 1)) (circnot (trueat 3))) ;;
- : bool list = [true; false; true; false; true]
```

A *nand* gate is a gate that computes the negation of an *and* gate. That is, it negates the *and* of its two inputs, so that its output is `false` only if both of its inputs are `true`.

Problem 5. ***[4 points]*** *Succinctly define a function* circnand *using the functions above to represent the* nand *gate. It should have the following behavior:*

```
# first 5 (circnand falses (trueat 3)) ;;
- : bool list = [true; true; true; true; true]
# first 5 (circnand (trueat 3) (trueat 3)) ;;
- : bool list = [true; true; true; false; true]
```

2. Credit card numbers and the Luhn algorithm

CREDIT CARD
4275 3156 0372 5493
4524
VALID THRU 01/19
FIRSTNAME LASTNAME

Here's an interesting bit of trivia: Not all credit card numbers are well-formed. The final digit in a 16-digit credit card number is in fact a *checksum*, a digit that is computed from the previous 15 by a simple algorithm.

The algorithm used to generate the checksum is called the *Luhn check*. To calculate the correct final checksum digit used in a 16-digit credit card number, you perform the following computation on the first 15 digits of the credit card number:

(1) Take all of the digits in an odd-numbered position (the leftmost digit being the first, not the zero-th digit, hence an odd-numbered one) and double them, subtracting nine if the doubling is greater than nine (called "casting out nines").

As an example, we'll use the (partial) credit card number in the sample above:

$$\underline{4}2\underline{7}5 \quad \underline{3}1\underline{5}6 \quad \underline{0}3\underline{7}2 \quad \underline{5}4\underline{9}x$$

Here, the odd-numbered digits (4, 7, 3, 5, 0, 7, 5, and 9) have been underlined. We double them and cast out nines to get 8, 5, 6, 1, 0, 5, 1, and 9.

(2) Add all of the even position digits and the doubled odd position digits together. For the example above, the sum would be

$$(2+5+1+6+3+2+4)+(8+5+6+1+0+5+1+9) = 23+35 = 58 \qquad .$$

(3) The checksum is then the digit that when added to this sum makes it a multiple of ten. In the example above the checksum would be 2, since adding 2 to 58 generates 60, which is a multiple of 10. Thus, the sequence 4275 3156 0372 5492 is a valid credit card number, but changing the last digit to any other makes it invalid. (In particular, the final 3 in the image above is not the correct checksum!)

Problem 6. *[5 points] Define an explicitly recursive polymorphic function* `odds` *to extract the elements at odd-numbered indices in a list, where the indices are counted starting with 1, so that*

```
# let cc = [4;2;7;5;3;1;5;6;0;3;7;2;5;4;9];;
val cc : int list = ...
# odds cc ;;
- : int list = [4; 7; 3; 5; 0; 7; 5; 9]
```

Problem 7. *[2 points] What is the type of the* `odds` *function that you defined?*

Problem 8. *[3 points] What is an appropriate recurrence equation for defining the time complexity of* `odds` *in terms of the length of its list argument?*

Problem 9. ***[3 points]*** *What is the time complexity of* **odds** *(in big-O notation)?*

Problem 10. ***[7 points]*** *If the function $f(n)$ is the time complexity of* **odds** *on a list of n elements, which of the following is true? Check all that apply:*

- ❑ $f \in O(1)$
- ❑ $f \in O(\log n)$
- ❑ $f \in O(\log n/c)$ *for all* $c > 0$
- ❑ $f \in O(c \cdot \log n)$ *for all* $c > 0$
- ❑ $f \in O(n)$
- ❑ $f \in O(n/c)$ *for all* $c > 0$
- ❑ $f \in O(c \cdot n)$ *for all* $c > 0$
- ❑ $f \in O(n^2)$
- ❑ $f \in O(n^2/c)$ *for all* $c > 0$
- ❑ $f \in O(c \cdot n^2)$ *for all* $c > 0$
- ❑ $f \in O(2^n)$
- ❑ $f \in O(2^n/c)$ *for all* $c > 0$
- ❑ $f \in O(c \cdot 2^n)$ *for all* $c > 0$

In addition to odds, it will be useful to have a function evens that extracts the elements at even-numbered indices in a list. Rather than have you write that, we'll just assume that it's available.

The process of doubling a number and "casting out nines" is easy to implement as well. Here is some code to do that:

```
let double9 n =
  (x * 2 - 1) mod 9 + 1;;
```

Finally, it will be useful to have a function to sum a list of integers.

Problem 11. *[3 points] Implement the function* sum *using the tail-recursive* List *module function* *fold_left*.

All the parts are now in place to implement the Luhn check algorithm.

Problem 12. *[8 points] Implement a function* `luhn` *that takes a list of integers and returns the check digit for that digit sequence. (You can assume that it is called with a list of 15 integers.) For instance, for the example above*

```
# luhn cc ;;
- : int = 2
```

You should feel free to use the functions `evens`*,* `odds`*,* `double9`*,* `sum`*, and any other OCaml library functions that you find useful and idiomatic.*

Problem 13. *[5 points] What is the time complexity of the* `luhn` *function in terms of the length n of its list argument? Use big-O notation. Explain why your implementation has that complexity.*

Problem 14. ***[1 points]*** *Now that you know how to generate valid credit card numbers not your own, would it be legal for you to use these numbers on an e-commerce web site to purchase goods? Would it be ethical for you to do so?*

3. Efficient algorithms for Euclid Corporation

Problem 15. ***[9 points]*** *Your friend, who works at EuclidCo, tells you that he's looking for a fast algorithm to solve a problem he's working on. So far, he's come across two algorithms, algorithm A, which has time complexity $O(n^3)$, and algorithm B, which is $O(2^n)$. He prefers algorithm A, and uses three different arguments to convince you of his decision. For each, evaluate the truth of the bolded statement, checking the appropriate box, and justify your answer in the provided space.*

(1) *"We're all about speed at EuclidCo, and **A will always be faster than B.**"*

❑*True* ❑*False*

(2) *"In a high stakes industry like ours, we can't afford to have more than a finite number of inputs that run slower than polynomial time, and **we can avoid this if we go with A.**"*

❑*True* ❑*False*

(3) *"We work with big data at EuclidCo. **For suitably large inputs, A will be faster on average than B.**"*

❑*True* ❑*False*

4. Mutable list miscellany

Recall the definition of the mutable list type from Lab 5:

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

The following functions extract the head and the (dereferenced) tail from a mutable list.

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

The `first` function returns a list (immutable) of the first `n` elements of a mutable list `mlst`:

```
let rec first (n: int) (mlst: 'a mlist) : 'a list =
  if n = 0 then []
  else match mlst with
       | Nil -> []
       | Cons(hd, tl) -> hd :: first (n-1) !tl ;;
```

Problem 16. *[5 points] Write code to define a mutable integer list* `alternating` *such that for all integers* `n`*, the expression* `first n alternating` *returns a list of alternating 1s and 2s, for example,*

```
# first 5 alternating ;;
- : int list = [1; 2; 1; 2; 1]
# first 10 alternating ;;
- : int list = [1; 2; 1; 2; 1; 2; 1; 2; 1; 2]
```

Answer the questions about the following interaction with the OCaml REPL, which makes use of the mutable list type and functions previously defined. (We've redacted portions of the OCaml responses.)

```
# let a = Cons(1, ref Nil);;
val a : ... redaction 1 ...
# let b = Cons(a, ref (Cons(a, ref Nil)));;
val b : ... redaction 2 ...
# let q1 = (mhead b == mhead (mtail b));;
val q1 : ... redaction 3 ...
# let q2 = first 5 a;;
val q2 : ... redaction 4 ...
# let _ =
  match mtail b with
    Cons(hd, tl) -> tl := hd;;
... redaction 5 ...
# let _ =
  match mhead (mtail b) with
    Cons(hd, tl) -> tl := a;;
... redaction 6 ...
# let q3 = first 5 a;;
val q3 : ... redaction 7 ...
```

Problem 17. *[2 **points**] What is the type of* `a` *at redaction 1?*

Problem 18. *[2 **points**] What is the type of* `b` *at redaction 2?*

Problem 19. *[2 **points**] What is the value of* `q1` *at redaction 3?*

Problem 20. *[2* ***points****] What is the value of* `q2` *at redaction 4?*

Problem 21. *[2* ***points****] Briefly explain what happens at redaction 5.*

Problem 22. *[2* ***points****] Briefly explain what happens at redaction 6.*

Problem 23. *[2* ***points****] What is the value of* `q3` *at redaction 7?*

5. Object-oriented counters

Here is a class type and class definition for counter objects. Each object maintains an integer state that can be "bumped" by adding an integer. The interface guarantees that only the two methods are revealed.

```
class type counter_interface =
  object
    method bump : int -> unit
    method get_state : int
  end

class counter : counter_interface =
  object
    val mutable state = 0
    method bump n = state <- state + n
    method get_state = state
  end
```

Problem 24. *[6 points] Write a class definition for a class* `loud_counter` *obeying the same interface that works identically, except that it also prints the resulting state of the counter each time the counter is bumped.*

Problem 25. ***[6 points]*** *Write a class type definition for an interface* `reset_counter_-interface`*, which is just like* `counter_interface` *except that it has an additional method of no arguments intended to reset the state back to zero.*

Problem 26. ***[6 points]*** *Write a class definition for a class* `loud_reset_counter` *satisfying the* `reset_counter_interface` *that implements a counter that both allows for resetting and is "loud" (printing the state whenever a bump or reset occurs).*

End of the 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”

Appendix A. CS51 cheat sheet

```
# (* ..................................... BASICS *)
# () ;;
 - : unit = ()

# 3 ;;
 - : int = 3

# 3.0 ;;
 - : float = 3.

# 'A' ;;
 - : char = A

# "xyz" ;;
 - : string = "xyz"

# false ;;
 - : bool = false

# 3 < 5 && true ;;
 - : bool = true

# Some 3 ;;
 - : int option = Some 3

# None ;;
 - : 'a option = None

# [3; 4] ;;
 - : int list = [3; 4]

# [] ;;
 - : 'a list = []

# (2, "xyz", 3.0) ;;
 - : int * string * float = (2, "xyz", 3.)

# fst (2,"abc") ;;
 - : int = 2

# snd (2, "abc") ;;
 - : string = "abc"

# "x" ^ "y" ^ "z" ;;
 - : string = "xyz"

# let x = 3 in
# if x<0 || x>0 then "nonzero" else "zero" ;;
 - : string = "nonzero"

# let e = exp 1. in
# let pi = 2.*.asin 1. in
# (e, pi) ;;
 - : float * float = (2.71828182846, 3.14159265359)

# (* .................. FUNCTIONS AND APPLICATION *)
# fun x -> x + 1 ;;
 - : int -> int = <fun>

# (fun x -> x + 1) 3 ;;
 - : int = 4

# fun x y -> x + y ;;
 - : int -> int -> int = <fun>

# fun (x, y) -> x + y ;;
 - : int * int -> int = <fun>

# fun () -> 4 ;;
 - : unit -> int = <fun>

# let uncurriedplus (x, y) = x + y in
# let curriedplus x y = x + y in
#   (uncurriedplus (1, 2), curriedplus 1 2) ;;
 - : int * int = (3, 3)

# (* .................................. MATCHING *)
# let x = 3 in
#   match x with
#   | 0 -> "zero"
#   | 1 -> "one"
#   | _ -> "more than one" ;;
 - : string = "more than one"

# let (x, y) = (Some 111, 2999) in
#   match (x, y) with
#   | Some z, _ -> z + y
#   | None, _ -> y ;;
 - : int = 3110

# let rec sum (x : int list) : int =
#   match x with
#   | [] -> 0
#   | u :: t -> u + sum t ;;
 val sum : int list -> int = <fun>

# (* ................................... RECORDS *)
# type rcrd = {foo : int; bar : string} ;;
 type rcrd = { foo : int; bar : string; }

# {foo = 3; bar = "xyz"} ;;
 - : rcrd = {foo = 3; bar = "xyz"}

# (* ..................................... LISTS *)
# 3 :: [4; 5] ;;
 - : int list = [3; 4; 5]

# [1;2;3] @ [4;5;6] ;;
 - : int list = [1; 2; 3; 4; 5; 6]

# List.length [8; 9; 10] ;;
 - : int = 3

# List.hd [3; 4] ;;
 - : int = 3

# List.tl [3 ;4] ;;
 - : int list = [4]

# List.tl [4] ;;
 - : int list = []

# List.map ;;
 - : ('a -> 'b) -> 'a list -> 'b list = <fun>

# List.map (fun x -> x + 100) [2;3;4] ;;
 - : int list = [102; 103; 104]

# List.map (fun x -> x = 3) [2;3;4] ;;
 - : bool list = [false; true; false]

# List.filter ;;
 - : ('a -> bool) -> 'a list -> 'a list = <fun>

# List.filter (fun x -> x < 4) [4;3;9;6;1;0;5] ;;
 - : int list = [3; 1; 0]

# List.fold_right ;;
 - : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

# List.fold_right (^) ["a";"b";"c"] "x" ;;
 - : string = "abcx"

# List.fold_left ;;
 - : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

```
#List.fold_left (^) "x" ["a";"b";"c"] ;;
 - : string = "xabc"

#List.find ;;
 - : ('a -> bool) -> 'a list -> 'a = <fun>

#List.find (fun x -> x > 10) [1;5;10;13;19] ;;
 - : int = 13

#List.find (fun x -> x > 10) [1;5;10] ;;
 Exception: Not_found.

#List.rev [8; 9; 10] ;;
 - : int list = [10; 9; 8]

#List.nth [8; 9; 10] 2 ;;
 - : int = 10

#List.nth [8; 9; 10] 3 ;;
 Exception: (Failure nth).

#(* ......................... VARIANT DATA TYPES *)
#type 'a option =
#  | Some of 'a
#  | None  ;;
 type 'a option = Some of 'a | None

#type 'a stack =
#  | Empty
#  | Top of ('a * 'a stack) ;;
 type 'a stack = Empty | Top of ('a * 'a stack)

#Top (3, Empty) ;;
 - : int stack = Top (3, Empty)

#(* ................................. EXCEPTIONS *)
#Not_found ;;
 - : exn = Not_found

#raise Not_found ;;
 Exception: Not_found.

#raise (Failure "error") ;;
 Exception: (Failure error).

#failwith "error" ;;
 Exception: (Failure error).

#try
#  Some (List.find (fun x -> x > 10) [1;5;10])
#with
#  | Not_found -> None
#  | e -> raise e ;;
 - : int option = None

#(* ..................... MODULES AND SIGNATURES *)
#module type STACK =
#  sig
#    type elt
#    type stack
#    exception Empty of string
#    val empty : unit -> stack
#    val push : stack * elt -> stack
#    val pop : stack -> elt * stack
#    val isEmpty : stack -> bool
#  end ;;
 module type STACK =
   sig
     type elt
     type stack
     exception Empty of string
     val empty : unit -> stack
     val push : stack * elt -> stack
     val pop : stack -> elt * stack
     val isEmpty : stack -> bool
   end

#module MakeStack(Arg: sig type t end)
#       : (STACK with type elt = Arg.t) =
#  struct
#    type elt = Arg.t
#    type stack = elt list
#    exception Empty of string
#    let empty () = []
#    let push (s,x) = x::s
#    let pop s = match s with
#    | x :: t -> (x, t)
#    | [] -> raise (Empty "empty")
#    let isEmpty = fun x -> x = []
#  end ;;
 module MakeStack :
   functor (Arg : sig type t end) ->
     sig
       type elt = Arg.t
       type stack
       exception Empty of string
       val empty : unit -> stack
       val push : stack * elt -> stack
       val pop : stack -> elt * stack
       val isEmpty : stack -> bool
     end

#module IntStack = MakeStack(struct type t = int end) ;;
 module IntStack :
   sig
     type elt = int
     type stack
     exception Empty of string
     val empty : unit -> stack
     val push : stack * elt -> stack
     val pop : stack -> elt * stack
     val isEmpty : stack -> bool
   end

#IntStack.push (IntStack.empty () , 3) ;;
 - : IntStack.stack = <abstr>

#(* .......................... REFS AND MUTABLES *)
#let rint = ref 3 ;;
 val rint : int ref = {contents = 3}

#rint := !rint + 5 ;;
 - : unit = ()

#rint ;;
 - : int ref = {contents = 8}

#!rint ;;
 - : int = 8

#type mut_demo = { mutable mut: int ; nonmut: int } ;;
 type mut_demo = { mutable mut : int; nonmut : int; }

#let m = { mut = 3; nonmut = 4 } ;;
 val m : mut_demo = {mut = 3; nonmut = 4}

#m.mut <- 5 ;;
 - : unit = ()

#m ;;
 - : mut_demo = {mut = 5; nonmut = 4}

#
```

```
#(* .................................... LAZINESS *)
#let a = lazy (!rint * 2) ;;
 val a : int lazy_t = <lazy>

#rint := 42 ;;
 - : unit = ()

#Lazy.force a ;;
 - : int = 84

#(* ......................... OBJECTS AND CLASSES *)
#class type demo_parent =
#  object
#    val v1 : int
#    val mutable v2 : bool
#    method getv1 : int
#    method getv2 : bool
#    method setv2 : bool -> unit
#  end ;;
 class type demo_parent =
   object
     val v1 : int
     val mutable v2 : bool
     method getv1 : int
     method getv2 : bool
     method setv2 : bool -> unit
   end

#class type demo_child =
#  object
#    inherit demo_parent
#    method getv3 : float (* no exposed inst var *)
#    method setv3 : float -> unit
#  end ;;
 class type demo_child =
   object
     val v1 : int
     val mutable v2 : bool
     method getv1 : int
     method getv2 : bool
     method getv3 : float
     method setv2 : bool -> unit
     method setv3 : float -> unit
   end

#class demo : demo_child =
#  object
#    val v1 = 0
#    val mutable v2 = true
#    val mutable v3 = 0.0
#    method getv1 = v1
#    method getv2 = v2
#    method setv2 b = v2 <- b
#    method getv3 = v3
#    method setv3 f = v3 <- f
#  end ;;
 class demo : demo_child

#let ob = new demo ;;
 val ob : demo = <obj>

#ob#getv1, ob#getv2, ob#getv3 ;;
 - : int * bool * float = (0, true, 0.)

#ob#setv2 false; ob#setv3 3.14 ;;
 - : unit = ()

#ob#getv1, ob#getv2, ob#getv3 ;;
 - : int * bool * float = (0, false, 3.14)

#(* .................................... PRINTING *)
#print_int 42; print_newline () ;;
 42
 - : unit = ()

#print_float 3.14159; print_newline () ;;
 3.14159
 - : unit = ()

#print_string "a string\n" ;;
 a string
 - : unit = ()

#Printf.printf "int: %d; float: %f; bool: %b; string: %s\n"
#  42 3.14159 true "a string" ;;
 int: 42; float: 3.141590; bool: true; string: a string
 - : unit = ()
```

Appendix B. The Native Lazy Streams module

```
module NativeLazyStreams =
  struct
    type 'a str = Cons of 'a * 'a stream
     and 'a stream = 'a str Lazy.t ;;

    (* [head s] returns the head of the stream [s] *)
    let head (s : 'a stream) : 'a =
      match Lazy.force s with
      | Cons(h, _) -> h ;;

    (* [tail s] returns the tail (stream) of the stream [s] *)
    let tail (s : 'a stream) : 'a stream =
      match Lazy.force s with
      | Cons(_,t) -> t ;;

    (* [first n s] returns a list (eager, not lazy) containing the
    first [n] elements of the stream [s] *)
    let rec first (n : int) (s : 'a stream) : 'a list =
      if n = 0 then []
      else head s :: first (n-1) (tail s) ;;

    (* [smap f s] applies [f] to all of the elements of the
       stream [s] *)
    let rec smap (f : 'a -> 'b) (s : 'a stream) : ('b stream) =
      lazy (Cons(f (head s), smap f (tail s)));;

    (* [smap2 f s1 s2] applies [f] to the corresponding elements
       of the streams [s1] and [s2] *)
    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))) ;;
  end
```