Final Exam
CS 1101 Computer Science I
Spring 2016

Section 03

KEY

Tuesday May 10, 2016

Instructor Muller
Boston College

Before reading further, please arrange to have an empty seat on either side of you. Now that you are seated, please write your name **on the back** of this exam.

This is a closed-notes and closed-book exam. Computers, calculators, and books are prohibited.

This is a 29 point exam.

- Partial credit will be given so be sure to show your work.
- Feel free to write helper functions if you need them.
- **Please write neatly.**

Problem	Points	Out Of
1		**11**
2		**12**
3		**6**
Total		**29**

Section 1 (11 Points Total)

1. (1 Point) In a sentence or two, what does the word *scope* mean with respect to a piece of code?

   Answer: Scope refers to the region of text where a bound symbol can be meaningfully referenced.

2. (1 Point) In a sentence or two, what does the phrase *abstract data type* mean?

   Answer: An abstract data type is a methodology for introducing new types. It emphasizes the separation of the *specification* of the new type from the *implementation* of the new type.

3. (1 Point) Is the following well-defined? If so, what is its value?

```
let f = List.map (fun x -> "BC")
in
f [false; true]
```

Answer: Yes it is well typed with value `["BC"; "BC"]`.

4. (1 Point) $567_8 = X_2$. Solve for X.

Answer: X = 101110111.

5. (1 Point) $16_{16} + 16_8 = X_{10}$. Solve for X.

Answer: $X = 36$**.**

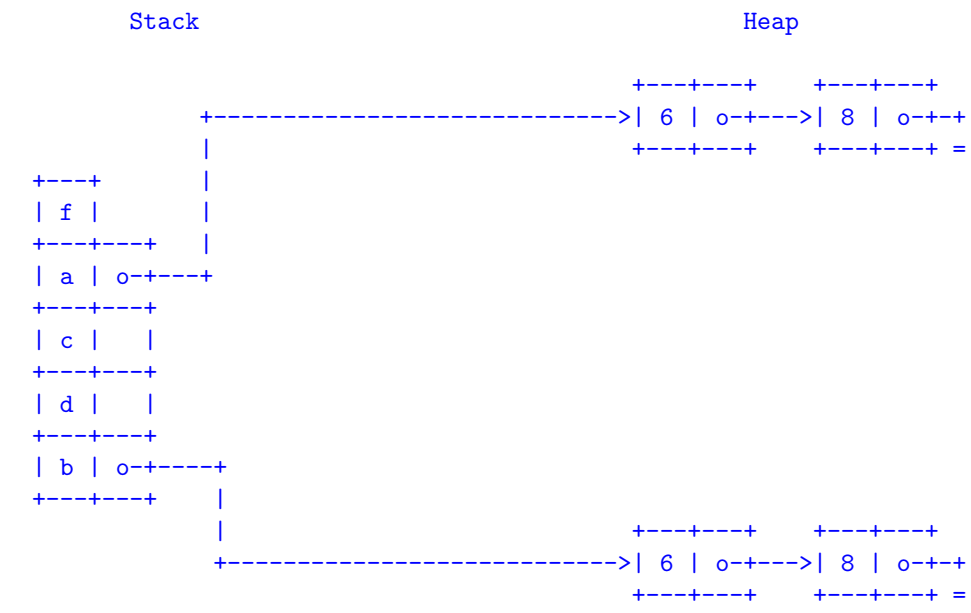
6. (2 Points) Consider the following definition.

```
let f a =
  let b = [6; 8] in                                (1)
  let c = a in                                     (2)
  let d = [(1, c); (3, c); (3, c)]                 (3)
  in
  d                                                (4)

f [6; 8]
```

Show the state of the stack and the heap after (1) has executed but before (2) has executed.

Answer:



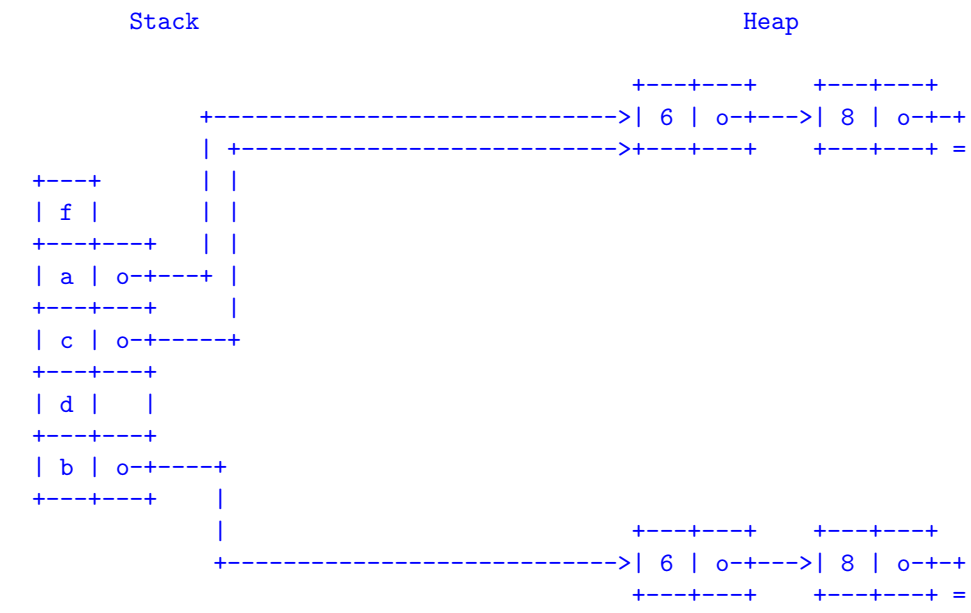
7. (2 Points) Consider the same definition.

```
let f a =
  let b = [6; 8] in                                (1)
  let c = a in                                     (2)
  let d = [(1, c); (3, c); (3, c)]                 (3)
  in
  d                                                (4)

f [6; 8]
```

Show the state of the stack and the heap after (2) has executed but before (3) has executed.

Answer:



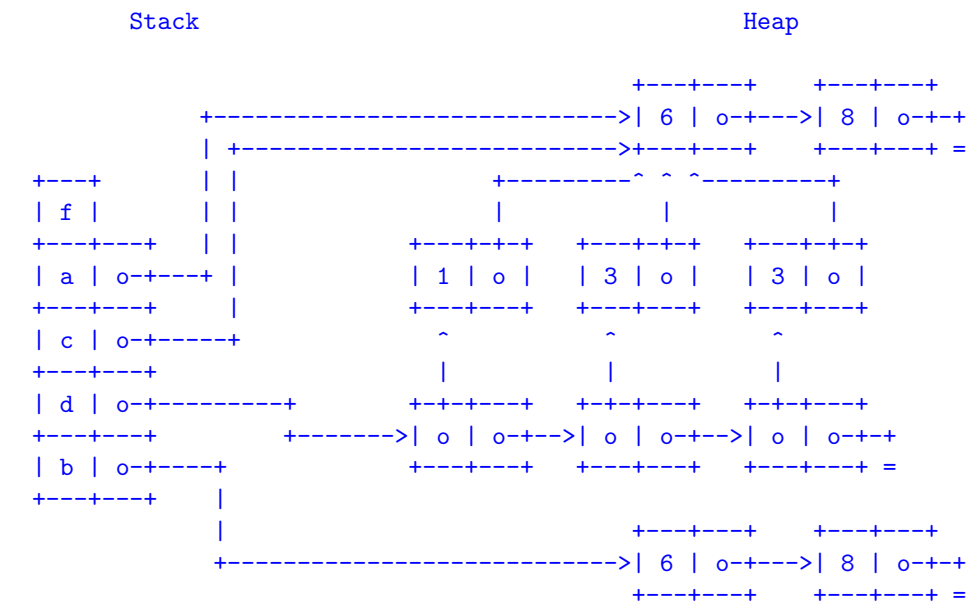
8. (2 Points) Consider the same definition.

```
let f a =
  let b = [6; 8] in                                (1)
  let c = a in                                     (2)
  let d = [(1, c); (3, c); (3, c)]                 (3)
  in
  d                                                (4)

f [6; 8]
```

Show the state of the stack and the heap after (3) has executed but before (4) has executed.

Answer:



Section 2 (12 Points Total)

1. (2 Points) Python has a handy feature for grabbing a slice of a list. For example, if `xs` is the list `[2, 4, 6, 8]` the Python notation `xs[1:3]` denotes the sublist `[4, 6]`. I.e., the elements from position `1` up to but not including position `3`.

   Write an OCaml function `slice : 'a list -> int -> int -> 'a list` such that a call of the function `(slice xs lo hi)` behaves like the Python slicer. You may assume that both `lo` and `hi` are non-negative, that `lo < hi` and that `hi <= List.length xs`.

   Answer:

```
let rec slice xs lo hi =
  match lo = hi with
  | true  -> []
  | false -> (List.nth xs lo) :: slice xs (lo + 1) hi
```

2. (2 Points) Write the function `slice : 'a array -> int -> int -> 'a array` which behaves like the above described function but with an array input rather than a list. In solving this you can make the same assumptions as above and you may use `Array.make : int -> 'a -> 'a array` but you may not use the `Array.to_list` function.

   Answer:

```
let slice a lo hi =
  let b = Array.make (hi - lo) a.(lo)
  in
  for i = 0 to (hi - lo - 1) do
    b.(i) <- a.(lo + i)
  done;
  b
```

3. (4 Points) Write a function `median : int list -> int` such that a given call `(median ns)` returns the *median* of `ns`. That is, the number `n` from `ns` such that there are an equal number of numbers in `ns` that are smaller than `n` as larger than `n`. For example, the call `(median [8; 5; 2; 6; 3])` should return `5` because there are 2 numbers smaller than 5 and 2 numbers larger than 5. You may assume that there will be a unique median. If you solve this problem by sorting, write a complete implementation of a `sort` function.

Answer:

```
let rec count n ns =
  match ns with
  | [] -> (0, n, 0)
  | m :: ms -> let (smaller, _, larger) = count n ms
               in
               match compare m n with
               | -1 -> (1 + smaller, n, larger)
               |  0 -> (smaller, n, larger)
               |  1 -> (smaller, n, 1 + larger)

let rec findMiddle triples =
  match triples with
  | [] -> failwith "not going to happen"
  | (smaller, n, larger)::rest -> (match smaller = larger with
                                   | true  -> n
                                   | false -> findMiddle rest)

let median ns =
  let counts = List.map (fun n -> count n ns) ns
  in
  findMiddle counts
```

4. (4 Points) Lets say a list has unwanted consecutive duplicates. For example, `[1; 2; 2; 2; 3; 3]` rather than `[1; 2; 3]`. Write the function `remDups : 'a list -> 'a list` which repairs such a list by removing consecutive duplicates.

Answer:

```
let rec remDups xs =
  match xs with
  | [] -> []
  | _ :: [] -> xs
  | x :: y :: ys -> let almost = remDups (y :: ys)
                    in
                    (match x = y with
                     | true  -> remDups almost
                     | false -> x :: remDups almost)
```

Section 3 (6 Points Total)

(6 Points) Consider a function `counts : 'a list -> ('a * int) list`. A call `(counts xs)` should evaluate to a list containing the elements of `xs` paired up with the number of occurrences in `xs`. For example, `(counts ['A'; 'B'; 'A'; 'A'])` should return the list `[('A', 3); ('B', 1); ('A', 3); ('A', 3)]`. **Write two versions of the `counts` function. The two versions should use different repetition idioms.**

Answer:

```
let rec count x xs =
  match xs with
  | [] -> 0
  | y :: ys -> (match x = y with
                | true  -> 1 + (count x ys)
                | false -> count x ys)

let counts xs =
  let ns = List.map (fun x -> count x xs) xs
  in
  List.combine xs ns

let counts xs =
  let rec repeat ys =
    match ys with
    | [] -> []
    | y :: ys  -> (y, count y xs) :: repeat ys
  in
  repeat xs
```