Final Exam
CSCI 1101 Computer Science I

Section **03**

KEY

Saturday May 9, 2015
Instructor Muller
Boston College
Spring 2015

Please do not write your name on the top of this test. Before reading further, please arrange to have an empty seat on either side of you. Now that you are seated, please note the number on top of your test and write it together with your name on the sheet that is circulating.

This is a closed-book and closed-notes exam. Computers, calculators and books are prohibited. Feel free to use a solution to one problem in solving subsequent problems. And unless otherwise specified, feel free to use any repetition idiom that you would like.

Do only one of sections 4 or 5.

Partial credit will be given so be sure to show your work. **Please try to write neatly.**

Problem	Points	Out Of
1 Snippets		8
2 Vignettes		6
3 Repetition		24
4 SVM		6
5 Dictionaries		6
Total		44

1 Snippets (8 Points Total)

For each of the following code snippets, indicate what would happen when attempting to compile and then run them. If they have a problem that would be detected when compiled indicate the problem. If they have a problem that would be detected when run, indicate the problem. If they have no problems, indicate what value and type they would produce when executed.

1. (2 Points)

```
let f x y = (x, (y, x))

# f 'A' 8;;
```

Answer:

Code is OK, returns `('A', (8, 'A')) : char * (int * char)`

2. (2 Points)

```
let f x y =
  let z = x / y
  in
    z

# f (1 / 2) 3;;
```

Answer:

Code is OK, returns `0`

3. (2 Points)

```
type pair_t = {first:int; second:float}

let swap pair = {first=pair.second; second=pair.first}

# swap {first=2; second=3.0};;
```

Answer:

Type error, cannot swap `int` and `float` fields.

4. (2 Points)

```
let rec map f xs =
  match xs with
  | [] -> []
  | x::xs -> (f x)::(map f xs)

# map (fun x -> "Hello World!");;
```

Answer:

Code is OK, returns a function from lists to strings.

2 Vignettes (6 Points Total)

1. (3 Points) OCaml's `Random.int : int -> int` function returns random integers. In particular, the call `(Random.int n)` returns an integer between `0` (inclusive) and `n` (exclusive). Write a function `fives : unit -> int` such that a call `(fives ())` returns a number among `5`, `10`, ..., `100`, i.e., a non-zero multiple of `5` less than or equal to `100`.

   Answer:

   ```
   let fives () = 5 + (Random.int 20 * 5)
   ```

2. (3 Points) OCaml's `Char` module has a pair of functions for navigating back and forth between characters and their ASCII codes. The `Char.chr : int -> char` function accepts an integer ASCII code for a character and returns the character. For example, the call `(Char.chr 65)` evaluates to the character `'A'`. OCaml's `Char.code : char -> int` function goes the other way — it returns the ASCII code of its argument. For example, the quadruple

   ```
   (Char.code 'A', Char.code 'Z', Char.code 'a', Char.code 'z')
   ```

   evaluates to `(65, 90, 97, 122)`. Note that the codes for uppercase letters are the 26 consecutive integers `65` through `90` and likewise for the lowercase ones. Write a function `randomLetter : () -> char` that returns a random upper or lowercase letter.

   Answer:

   ```
   let randomLetter () =
     let n = Random.int 26
     in
     match Random.int 2 with
     | 0 -> Char.chr(65 + n)
     | 1 -> Char.chr(97 + n)
   ```

3 Repetition (24 of 30 Points Total)

This section has 8 problems. There are six easier 3 point problems and two more challenging 6 point problems. Do any of the 8 problems totalling 24 points.

1. (3 Points) OCaml's `Char` module contains a function `Char.escaped : char -> string` that converts a character into a string. For example, `(Char.escaped 'A')` returns the string `"A"`. Write a function `randomString : int -> string` such that a call `(randomString n)` returns a string of `n` randomly chosen upper or lower case letters.

   Answer:

```
let rec randomString n =
  match n = 0 with
  | true ->  ""
  | false -> (Char.escaped (randomLetter())) ^ (randomString (n - 1))
```

2. (3 Points) Write a function `removeNth : int -> 'a list -> 'a list` such that a call `(removeNth n xs)` returns the list of elements of `xs` with the `n`th element (starting from 0) removed. For example, the call `(removeNth 3 [2; 4; 6; 8; 10; 12])` should return the list `[2; 4; 6; 10; 12]`. Raise a `failwith` exception if `n` is greater than or equal to the length of `xs`.

   Answer:

```
let rec removeNth n xs =
  match (n, xs) with
  | (0, []) -> failwith "not enough elements"
  | (0, _::xs) -> xs
  | (n, x::xs) -> x::(removeNth (n - 1) xs)
```

3. (3 Points) Write a function `zip : 'a list * 'b list -> ('a * 'b) list` such that a call `(zip xs ys)` returns a list with the ith element of `xs` paired with the ith element of `ys`. For example, the call `zip(['A'; 'B'], [0; 1])` should return the list `[('A', 0); ('B', 1)]`. You may assume that the lists `xs` and `ys` are of the same length. NB: the input type is a **pair** of lists.

Answer:

```
let rec zip (xs, ys) =
  match (xs, ys) with
  | ([], []) -> []
  | (x::xs, y::ys) -> (x, y)::zip(xs, ys)
```

4. (3 Points) Write a function `product : 'a list -> 'b list -> ('a * 'b) list` such that a call `(product xs ys)` returns the list of pairs of each element of `xs` paired up with each element of `ys`. For example, the call `(product ['A'; 'B'; 'C'] [0; 1])` should return the list

`[('A', 0); ('A', 1); ('B', 0); ('B', 1); ('C'; 0); ('C'; 1)]`.

Answer:

```
let product xs ys =
  List.fold_left (@) [] (List.map (fun x -> List.map (fun y -> (x, y)) ys) xs)
```

5. (3 Points) OCaml's `Array` module is specified on the page at the back of the exam. Write a function `arrayFilter : ('a -> bool) -> 'a array -> 'a array` such that a call `(arrayFilter test a)` returns a new array containing only those elements of `a` that pass the `test`. For example, the call `arrayFilter (fun n -> n mod 2 = 0) [| 2; 3; 4|]` should return the array `[|2; 4|]`.

Answer:

```
let arrayFilter a test = Array.of_list (List.filter test (Array.to_list a))
```

6. (3 Points) Write three versions of the function `addArray : int array -> int` that adds the elements of an integer array. You can use recursion, `while`-loops, `for`-loops, `maps`, `folds`, whatever you want, but provide 3 different definitions of `addArray`.

Answer:

```
let addArray a = Array.fold_left (+) 0 a

let addArray a = Array.fold_right (+) a 0

let addArray a =
  let rec repeat i sum =
    match i = Array.length a with
    | true  -> sum
    | false -> repeat (i + 1) (sum + a.(i))
  in
  repeat 0 0

let addArray a =
  let sum = ref 0
  in
  for i = 0 to Array.length a - 1 do
    sum := !sum + a.(i)
  done;
  !sum
```

7. (6 Points) Write a function `shuffle : int array -> unit` which accepts an array of 52 integers and which returns with the elements of the array having been placed in random order.

   NB: Let `a` be an `int array` and consider the call `(shuffle a)`. Since arrays are mutable, the `shuffle` function can rearrange the elements of `a` without returning any result. In particular, in a piece of code `...a...; (shuffle a); ...a...`, the contents of `a` on the right (after the call of `shuffle`) might differ from those of `a` on the left (before the call of `shuffle`) even though `shuffle` returned only `()`. Remember that an array can be mutated using the `<-` operator as in `a.(0) <- 24`.

   Answer:

```
let swap a i j =
  let temp = a.(i) in
  a.(i) <- a.(j);
  a.(j) <- temp

let shuffle deck =
  let back = ref (Array.length deck - 1) in
  while !back > 0 do
    let j = Random.int !back in
    swap deck j !back;
    back := !back - 1
    done
```

8. (6 Points) A list of pairs is *closed* if whenever (a, b) and (b, c) are in the list, so is (a, c). Write a function `closure : ('a * 'a) list -> ('a * 'a) list` such that a call `(closure pairs)` returns the closure of `pairs`. For example, the call `(closure [('A', 'B'); ('B', 'C'); ('C', 'D')])` should evaluate to the list

```
[('A', 'B'); ('B', 'C'); ('C', 'D'); ('A', 'C'); ('B', 'D'); ('A', 'D')]
```

Answer:

```
let rec findMatch (a, b) pairs allPairs =
  match pairs with
  | [] -> None
  | (b', c)::pairs ->
     (match b = b' && (not (List.mem (a, c) allPairs)) with
      | true  -> Some c
      | false -> findMatch (a, b) pairs allPairs)

let rec findNewPair pairs allPairs =
  match pairs with
  | [] -> None
  | (a, b)::pairs ->
     (match findMatch (a, b) allPairs allPairs with
      | Some c -> Some (a, c)
      | None   -> findNewPair pairs allPairs)

let rec closure pairs =
  match (findNewPair pairs pairs) with
  | None -> pairs
  | Some pair -> closure (pair::pairs)
```

4 The Simple Virtual Machine (6 Points)

(6 Points) In March we learned about storage and about the basic machine model that underlies all of our computers. We saw the assembly language for a simple virtual computer (SVM) with a RAM and a CPU with an ALU and 6 registers: **pc**, **psw**, **R0**, ...,**R3**. The SVM instruction set is specified on the attached sheet. Write an SVM program that computes the integer average of the numbers in `R1`, `R2` and `R3` leaving the average in `R0`.

Answer:

```
% Assume RAM[0] == 3
%
  ADD  R1, R1, R2
  ADD  R1, R1, R3
  LOD  R2, 0
  DIV  R0, R1, R2
```

5 Dictionaries (6 Points Total)

Dictionaries (or *maps*) associating *keys* with *values* are ubiquitous in coding. OCaml's Standard Library has a handy `Map` module — we'll set that module aside for this question and review a different representation that we covered in class. The main operations on a dictionary are:

```
insert : key -> value -> dictionary -> dictionary

and

find : key -> dictionary -> value option
```

Finding the value of a key in a dictionary can be efficient when there is an ordering of the keys that allows us to ask if one key is less than or greater than another. If this is the case, the dictionary can be represented as a *binary search tree*. We'll focus on dictionaries with integer keys and character values. In OCaml it's natural to represent such a dictionary with a bst defined as follows:

```
type bst = Empty | Node of bst * int * char * bst;;
```

The empty bst is represented by the constant `Empty` and a non-empty bst is represented by a `Node(left, key, value, right)` where `left` and `right` are bsts and the keys are organized in such a way that every key in `left` is smaller than `key` and every key in `right` is greater than `key`. For example,

```
Node(Node(Empty, 3, 'Z', Empty), 4, 'A', Empty)
```

is a bst representing a dictionary relating key `3` to value `'Z'` and key `4` to value `'A'`.

Given the above type definition, the `insert` function can be written as follows:

```
let rec insert key value dictionary =
  match dictionary with
  | Empty -> Node(Empty, key, value, Empty)
  | Node(left, key', value', right) ->
      (match key == key' with
       | true  -> Node(left, key, value, right)
       | false ->
           (match key < key' with
            | true  -> let newLeft = (insert key value left) in
                       Node(newLeft, key', value', right)
            | false -> let newRight = (insert key value right) in
                           Node(left, key', value', newRight))
```

1. **(3 Points) Write the function** `keys : dictionary -> key list` **such that a call** `(keys dictionary)` **returns a list of all of the keys in** `dictionary`**.**

   Answer:

```
let rec keys dictionary =
  match dictionary with
  | Empty -> []
  | Node(left, key, value, right) -> key::(keys left) @ (keys right)
```

2. **(3 Points) Write the function** `find : key -> dictionary -> value option` **such that a call**

   `(find key dictionary)`

   returns `None` **if the** `key` **is not in** `dictionary`**. Otherwise, if** `key` **is related to** `value` **in the** `dictionary` **the** `find` **function should return** `Some value`**. For example,** `(find 2 Empty)` **should evaluate to** `None` **while** `(find 2 (Node(Node(Empty, 2, 'B', Empty), 3, 'A', Empty)))` **should return** `Some 'B'`**.**

   Answer:

```
let rec find key bst =
  match bst with
  | Empty -> None
  | Node(left, key', value, right) ->
      (match key' == key with
       | true  -> Some value
       | false ->
          (match key < key' with
           | true  -> find key left
           | false -> find key right))
```

6 OCaml's Array Module

- `val length : 'a array -> int`

  return the length of the array.

- `val get : 'a array -> int -> 'a`

  `(Array.get a n)` **returns the element number** `n` **of array** `a`**. The first element has number 0. The last element has number** `Array.length a - 1`**. You can also write** `a.(n)` **instead of** `Array.get a n`**.**

- `val set : 'a array -> int -> 'a -> unit`

  `(Array.set a n x)` **modifies array** `a` **in place, replacing element number** `n` **with** `x`**. You can also write** `a.(n) <- x` **instead of** `Array.set a n x`**.**

- `val make : int -> 'a -> 'a array`

  `(Array.make n x)` **returns a fresh array of length** `n`**, initialized with** `x`**. All the elements of this new array are initially physically equal to** `x` **(in the sense of the == predicate). Consequently, if** `x` **is mutable, it is shared among all elements of the array, and modifying** `x` **through one of the array entries will modify all other entries at the same time.**

- `val init : int -> (int -> 'a) -> 'a array`

  `(Array.init n f)` **returns a fresh array of length** `n`**, with element number** `i` **initialized to the result of** `f i`**. In other words,** `Array.init n f` **tabulates the results of** `f` **applied to the integers** `0` **to** `n-1`**.**

- `val make_matrix : int -> int -> 'a -> 'a array array`

  `(Array.make_matrix dimx dimy e)` **returns a two-dimensional array (an array of arrays) with first dimension** `dimx` **and second dimension** `dimy`**. All the elements of this new matrix are initially physically equal to** `e`**. The element (x,y) of a matrix** `m` **is accessed with the notation** `m.(x).(y)`**.**

- `val append : 'a array -> 'a array -> 'a array`

  `(Array.append v1 v2)` **returns a fresh array containing the concatenation of the arrays** `v1` **and** `v2`**.**

- `val concat : 'a array list -> 'a array`

  Same as `Array.append`**, but concatenates a list of arrays.**

- `val sub : 'a array -> int -> int -> 'a array`

  `(Array.sub a start len)` **returns a fresh array of length** `len`**, containing the elements number** `start` **to** `start + len - 1` **of array** `a`**.**

- `val copy : 'a array -> 'a array`

  `(Array.copy a)` **returns a copy of** `a`**, that is, a fresh array containing the same elements as** `a`**.**

- `val fill : 'a array -> int -> int -> 'a -> unit`

  `(Array.fill a ofs len x)` **modifies the array** `a` **in place, storing** `x` **in elements number** `ofs` **to** `ofs + len - 1`**.**

- `val to_list : 'a array -> 'a list`

  `(Array.to_list a)` **returns the list of all the elements of** `a`**.**

- `val of_list : 'a list -> 'a array`

  `(Array.of_list l)` **returns a fresh array containing the elements of** `l`**.**

- `val iter : ('a -> unit) -> 'a array -> unit`

  `(Array.iter f a)` **applies function** `f` **in turn to all the elements of** `a`**. It is equivalent to** `f a.(0); f a.(1); ...; f a.(Array.length a - 1); ()`**.**

- `val map : ('a -> 'b) -> 'a array -> 'b array`

  `(Array.map f a)` **applies function** `f` **to all the elements of** `a`**, and builds an array with the results returned by** `f`**:** `[| f a.(0); f a.(1);...; f a.(Array.length a - 1) |]`**.**

- `val iteri : (int -> 'a -> unit) -> 'a array -> unit`

  Same as `Array.iter`**, but the function is applied to the index of the element as first argument, and the element itself as second argument.**

- `val mapi : (int -> 'a -> 'b) -> 'a array -> 'b array`

  Same as `Array.map`**, but the function is applied to the index of the element as first argument, and the element itself as second argument.**

- `val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a`

  `(Array.fold_left f x a)` **computes** `f (... (f (f x a.(0)) a.(1)) ...) a.(n-1)`**, where** `n` **is the length of the array** `a`**.**

- `val fold_right : ('b -> 'a -> 'a) -> 'b array -> 'a -> 'a`

  `(Array.fold_right f a x)` **computes** `f a.(0) (f a.(1) ( ... (f a.(n-1) x) ...))`**, where** `n` **is the length of the array** `a`**.**

7 The Simple Virtual Machine

The instruction set of SVM is as follows.

- `LOD Rd, addr`: **loads RAM[addr] into register Rd.**
- `STO Rs, addr`: **stores the contents of register Rs into location addr in the memory.**
- `MOV Rd, Rs`: **copies the contents of register Rs into register Rd.**
- `ADD Rd, Rs, Rt`: **adds the contents of registers Rs and Rt and stores the sum in register Rd.**
- `SUB Rd, Rs, Rt`: **subtracts the contents of register Rt from Rs and stores the difference in register Rd.**
- `MUL Rd, Rs, Rt`: **multiplies the contents of register Rt by Rs and stores the product in register Rd.**
- `DIV Rd, Rs, Rt`: **divides the contents of register Rs by Rt and stores the integer quotient in register Rd.**
- `CMP Rs, Rt`: **sets PSW = Rs - Rt. Note that if Rs > Rt, then PSW will be positive, if Rs == Rt, then PSW will be 0 and if Rs < Rt, then PSW will be negative.**
- `CPZ Rs`: **sets PSW = Rs. If Rs == 0, then PSW will be 0, otherwise it will be non-zero.**
- `BLT disp`: **if PSW is negative, causes the new value of PC to be the sum PC + disp. Note that if disp is negative, this will cause the program to jump backward in the sequence of instructions. If PSW $\geq$ 0, this instruction does nothing.**
- `BEQ disp`: **if PSW == 0, causes the new value of PC to be the sum PC + disp. Note that if disp is negative, this will cause the program to jump backward in the sequence of instructions. If PSW $\neq$ 0, this instruction does nothing.**
- `BGT disp`: **if PSW, is positive, causes the new value of PC to be the sum PC + disp. Note that if disp is negative, this will cause the program to jump backward in the sequence of instructions. If PSW $\leq$ 0, this instruction does nothing.**
- `JMP disp`: **causes the new value of PC to be the sum PC + disp.**
- `HLT`: **causes the svm machine to print the contents of registers PC, PSW, R0, R1, R2 and R3. It then stops, returning ().**