Final Exam
CS 1103 Computer Science I Honors
Fall 2017

KEY

Thursday December 14, 2017

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 25 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 Snippets		**5**
2 Storage		**4**
3 Repetition		**12**
3 SVM		**4**
Total		**25**

Section 1: Snippets (5 Points Total)

1. (1 Point) Does every well-formed expression have a value? If not, give an example.

   Answer: No, e.g., `1 / 0` doesn't have a value.

2. (1 Point) What is the type of `compose`?

   ```
   let compose f g x = f (g x)
   ```

   Answer:

   ```
   compose : ('b -> 'c) -> ('a -> 'b) -> 'a -> 'c
   ```

3. (1 Point) Is the following expression well formed? If so, simplify the expression, one step at a time.

   ```
   type t = A | B
   match (match (1 + 3) = 4 with | true -> A | false -> B) with | A -> 2 | B -> 3
   ```

   Answer:

   ```
   match (match (1 + 3) = 4 with | true -> A | false -> B) with | A -> 2 | B -> 3 ->
     match (match 4 = 4 with | true -> A | false -> B) with | A -> 2 | B -> 3 ->
     match (match true with | true -> A | false -> B) with | A -> 2 | B -> 3 ->
     match A with | A -> 2 | B -> 3 ->
     2
   ```

4. (1 Point) Is the following function well-defined? If so, what is its type?

```
let what x y = 5 + y
```

Answer: Yes it is well typed with type `what : 'a -> int -> int`.

5. (1 Point) $122_3 = X_5$. Solve for X.

Answer: X = 32

Section 2: Storage Diagrams (4 Points)

1. (2 Points) Consider the following code:

```
let rec append xs ys =
  match xs with
  | [] -> ys
  | z :: zs -> z :: append zs ys

let f us vs =
  let ws = append us vs         (1)
  in
  ws                            (2)

f [1; 2] [3; 4; 5]
```

Show the state of the stack and the heap after (1) has executed but before (2) has executed. Note that after (1), the call to `append` is finished so your answer should not have any activation records for `append`.

Answer:

```
         Stack                                        Heap

+----+                                        +---+---+     +---+---+
| f |             +-------------------------->| 1 | o-+---->| 2 | o-+-+
+----+---+        |                           +---+---+     +---+---+ |
| us | o-+--------+                                                   =
+----+---+                                    +---+---+     +---+---+     +---+---+
| vs | o-+----------------------------------->| 3 | o-+---->| 4 | o-+---->| 5 | o-+-+
+----+---+                                    +---+---+     +---+---+     +---+---+ |
| ws | o-+--------+                             ^                                   =
+----+---+        |                             |
                  |         +---+---+     +---+-+-+
                  +-------->| 1 | o-+---->| 2 | o |
                            +---+---+     +---+---+
```

2. (2 Points) Consider the following code:

```
let rec f a =
  let b = Array.map List.length a            (1)
  in
  b                                          (2)

f [| [5]; [4; 8]; [] |]
```

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

Answer:

```
         Stack                                          Heap

+---+                                              +---+---+
| f |                                       +----->| 5 | o-+-+
+---+---+                         +-----+   |      +---+---+ |
| a | o-+------------------------>|  o--+---+                =
+---+---+                         +-----+          +---+---+     +---+---+
| b | o-+----------+              |  o--+--------->| 4 | o-+---->| 8 | o-+-+
+---+---+          |              +-----+          +---+---+     +---+---+ |
                   |              |  o--+-+                                =
                   |              +-----+ |
                   |                      =
                   |
                   |              +-----+
                   +------------->|  1  |
                                  +-----+
                                  |  2  |
                                  +-----+
                                  |  0  |
                                  +-----+
```

Section 3: Repetition (12 Points Total)

1. (3 Points) Write the function `downFrom : int -> int list` such that a call `(downFrom n)`, where `n` is a positive integer, returns the list `[n - 1; n - 2; ...; 0]`.

   Answer:

```
let rec downFrom n =
  match n = 0 with
  | true  -> []
  | false -> (n - 1) :: downFrom (n - 1)
```

2. (3 Points) An array is *palindromic* if it reads the same way left-to-right and right-to-left. For example, all of `[||]`, `[| 3 |]`, `[| 2; 3; 2 |]` and `[| 2; 3; 3; 2 |]` are palindromic. Write the function `isPalindromic : int array -> bool`. (No, there is no `Array.rev` function and yes,

```
[| 1; 2 |] = [| 1; 2 |]
```

   is `true`.)

   Answer:

```
let isPalindromic a = a = Array.of_list (List.rev (Array.to_list a))

let isPalindromic a =
  let n = Array.length a in
  let rec check i = (i > n / 2) || (a.(i) = a.(n - i - 1) && check (i + 1))
  in
  check 0
```

3. (3 Points) We covered Eratosthenes' famous *sieve algorithm* which finds all of the prime numbers in a list of integers ascending from the prime number two `[2; 3; 4; ...]`. The algorithm works by filtering out multiples of primes. Write any version of the function `sieve : int list -> int list` such that a call `(sieve [2; 3; ...; N])` returns a list `[2; 3; ...]` with only the primes.

Answer:

```
let sieve ns =
  let rec repeat ns answer =
    match ns with
    | [] -> answer
    | m :: ns ->
      let ms = List.filter (fun n -> not (isFactor m n)) ns
      in
      repeat ms (m :: answer)
  in
  List.rev (repeat ns [])
```

4. (3 Points) A square 2D array is *symmetrical* if the values are the same across the diagonal running from upper-left to lower-right. For example,

```
[| [| 1; 2; 3 |];           [| [| 6; 2; 3; 4 |];
   [| 2; 1; 4 |];     and      [| 2; 6; 5; 8 |];
   [| 3; 4; 1 |]; |]           [| 3; 5; 6; 7 |];
                               [| 4; 8; 7; 6 |] |]
```

are both symmetrical. The following function `isSymmetrical : int array array -> bool` returns `true` if the input is symmetrical.

```
let isSymmetrical a =
  let n = Array.length a in
  let answer = ref true
  in
  for i = 0 to n - 1 do
    for j = 0 to i do
      answer := !answer && a.(i).(j) = a.(j).(i)
    done
  done;
  !answer
```

Rewrite the function without using for-loops.

Answer:

```
let isSymmetrical a =
  let n = Array.length a in
  let answer = ref true in
  let i = ref 0
  in
  while !i < n do
    let j = ref 0
    in
    while !j <= !i do
      answer := !answer && a.(!i).(!j) = a.(!j).(!i);
      j := !j + 1
    done;
    i := !i + 1
  done;
  !answer
```

Section 4: SVM (4 Points Total)

Assume that the data segment contains a list of non-zero numbers ending with a sentinal zero, something like [3; 1; 2; 4; 5; 0]. Write an SVM program that halts after applying the function $f(x) = 3x^2 + 4$ to each non-zero element in the data segment. For example, given the data above, the code would leave [31; 7; 16; 52; 79; 0].

Answer:

```
Li  R1, 3
Mov R3, Zero
Lod R0, 0(R3)
Cmp R0, Zero
Beq 8
Mul R0, R0, R0
Mul R0, R0, R1
Li  R2, 4
Add R0, R0, R2
Sto R0, 0(R3)
Li  R2, 1
Add R3, R3, R2
Jmp -11
Hlt
```

1 The Simple Virtual Machine

The instruction set of SVM is as follows.

- `Lod Rd, offset(Rs)`: Let `base` be the contents of register `Rs`. Then this instruction loads the contents of data segment location `offset + base` into register `Rd`.
- `Sto Rs, offset(Rd)`: Let `base` be the contents of register `Rd`. Then this instruction stores the contents of register `Rs` into data segment location `offset + base`.
- `Li Rd, number`: loads `number` into register Rd.
- `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.
- `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.
- `Jsr disp`: Jump subroutine: `RA := PC` then `PC := PC + disp`.
- `R`: Return from subroutine: `PC := RA`.
- `Hlt`: causes the svm machine to print the contents of registers PC, PSW, R0, R1, R2 and R3. It then halts.