Exam 2
CSCI 1103 Computer Science I Honors

KEY

Friday November 8, 2019
Instructor Muller
Boston College

Fall 2019

Please do not write your name on the top of this quiz. 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 quiz. 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.

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

Problem	Points	Out Of
1 Snippets		3
2 Storage Diagrams		3
3 Coding		9
4 SVM		5
Total		20

1 Snippets (3 Points Total)

For problems 1. and 2. in this section, use the following definition of type `person`.

```
type person = { name : string; age : int}
```

1. (1 Point) Is the following definition of `voter` well-formed? If so, what is its type? If it's not well-formed, what's wrong with it?

   ```
   let voter person = person.age >= 18
   ```

   Answer:

   Yes, `voter : person -> bool`

2. (1 Point) Is the following definition of `voters` well formed? If so, what is its type? If it's not well-formed, what's wrong with it?

   ```
   let voters = List.map voter
   ```

   Answer:

   Yes, `voters : person list -> bool list`

3. (1 Point) Is the following well-formed? If so, what is its type? If it's not well-formed, what's wrong with it?

   ```
   let f x y = y * 2
   ```

   Answer:

   Yes, `f : 'a -> int -> int`

2 Storage Diagrams (3 Points)

Show the state of the Stack and the Heap after (1) has executed but before (2) has executed.

```
let rec copy xs =
  match xs with
  | [] -> []
  | x :: xs -> x :: copy xs

let what ys =
  let zs = (0, 1) :: (copy ys)                 (1)
  in
  zs                                           (2)

let go () =
  let xs = [(2, 3); (4, 5)]
  in
  what xs

go ()
        Stack                                                     Heap
```

Answer:

```
+------+
| what |
+------+-----+             +---+---+      +---+---+      +---+---+
|   zs |  o--+------------>| o | o-+----->| o | o-+----->| o | o-+-+
|      +-----+             +-+-+---+      +-+-+---+      +-+-+---+ =
|   ys |  o--+---+           v              v              v
+------------+   |         +-------+      +-------+      +-------+
                 |         |   0   |      |   2   |      |   4   |
                 |         +-------+      +-------+      +-------+
                 |         |   1   |      |   3   |      |   5   |
                 |         +-------+      +-------+      +-------+
+----+           |                          ^                ^
| go |           |                          |                |
+----+-+-----+   +----------------------->+-+-+---+      +-+-+---+
|   xs |  o--+--------------------------->| o | o-+----->| o | o-+-+
+------+-----+                            +---+---+      +---+---+ =
```

3 Coding (9 Points)

The following three definitions are for problems 1. and 2.

```
type t = A | B | C

let toS t = match t with | A -> "A" | B -> "B" | C -> "C"

type tree = Empty
          | Node of { info  : t
                    ; left  : tree
                    ; right : tree
                    }
```

1. (3 Points) Remember that `Code.fmt` is a string building function that accepts a format string with holes in it, and values to fill the holes; it returns a string. For example, `(Code.fmt "%s:%s" "AB" "CD")` returns the string `"AB:CD"`.

   Write a function `format : tree -> string` such that a call `(format tree)` returns a string representation of `tree`. You may assume that `tree` is not `Empty`. Here are a few examples.

```
let t1 = Node { info = C; left = Empty; right = Empty }
let t2 = Node { info = B; left = t1; right = Empty }
let t3 = Node { info = B; left = Empty; right = t1 }
let t4 = Node { info = A; left = t1; right = t2 }          A
                                                          / \
(format t1) => "C"                                       C   B
(format t2) => "B(C)"   these two the same!                 /
(format t3) => "B(C)"                                      C
(format t4) => "A(C, B(C))"
```

   Answer:

```
type t = A | B | C
let toS t = match t with | A -> "A" | B -> "B" | C -> "C"
type tree = Empty | Node of { info  : t; left  : tree; right : tree}

let rec format tree =
  match tree with
  | Empty -> failwith "can't happen"
  | Node {info; left = Empty; right = Empty} -> toS info
  | Node {info; left = Empty; right = only}
  | Node {info; left = only; right = Empty} ->
    let root = toS info in
    let child = format only
    in
    Code.fmt "%s(%s)" root child
  | Node {info; left; right} ->
    let root = toS info
    in
    Code.fmt "%s(%s, %s)" root (format left) (format right)
```

2. (3 Points) The *depth* of a node in a tree is the number of hops from the root. Write a function `hasDepth : tree -> int -> bool` such that a call `(hasDepth tree n)` returns `true` if `tree` has at least depth `n`. Otherwise it should return `false`. For example, `(hasDepth t4 2)` should return `true` but `(hasDepth t3 2)` should return `false`. You may assume that n is non-negative.

Answer:

```
type t = A | B | C
type tree = Empty | Node of { info  : t; left  : tree; right : tree}

let rec hasDepth tree m =
  match tree with
  | Empty | Node { left = Empty; right = Empty } ->
    m = 0
  | Node {left; right} ->
    let n = m - 1
    in
    (m = 0) || (hasDepth left n) || (hasDepth right n)
```

3. (3 Points) Hoare's *quicksort* algorithm partitions a list of keys around a *pivot*. For example, the call `(partition 3 [5; 2; 1; 4])` would return the pair of lists `([2; 1], [5; 4])`. Write the function `partition : int -> int list -> (int list * int list)`. You may assume that all of the integers are unique.

Answer:

```
let rec partition pivot xs =
  match xs with
  | [] -> ([], [])
  | y :: ys ->
    let (smaller, larger) = partition pivot ys
    in
    match compare y pivot < 0 with
    | true  -> (y :: smaller, larger)
    | false -> (smaller, y :: larger)
```

4 The Simple Virtual Machine (5 Points)

The SVM instruction set is specified on the attached sheet. The data segment contains some positive integers (with no repeats) and is trailed by a single 0. E.g., `data = [2, 5, 4, 8, 6, 0]`. Write an SVM program that halts with the largest number in `R0`. For the data in this example, that would be `8`.

Answer:

```
Li   R1, 1       # for incrementing
Mov  R2, Zero    #
Lod  R0  0(R2)   # assume first number is largest
Add  R2, R2, R1  # increment the pointer
Lod  R3  0(R2)   # load next number into R3
Cmp  R3, Zero
Beq  4           # all done, max is in R0
Cmp  R0, R3
Bgt  -6          # no new max
Mov  R0, R3      # new max
Jmp  -8
Hlt
```

5 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.