Final Exam (2PM)
CSCI 1103 Computer Science 1 Honors

KEY

Wednesday December 18, 2019
Instructor Muller
Boston College

Fall 2019

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 index card.

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.

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 Storage Diagrams		8
3 Coding		18
4 SVM		6
Total		40

1 Snippets (8 Points Total)

1. (1 Point) Given definitions

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

   Is

   ```
   let mia = { ident = { name = "Mia"
                       ; age = 20
                       }
             ; class = 2022
             }
   in
   mia.ident.name
   ```

   well-formed? If so, what is its type?

   Answer:

   Yes, `mia.ident.name : string`**.**

2. (1 Point) Is `let x : int = 2.0 *. 3.0` well-formed? If so, what is the type of `x`? If it's not well-formed, what's wrong with it?

   Answer:

   The expression is ill-formed because the programmer specified that `x` is of type `int` but the definition is of type `float`.

3. (1 Point) Is `2 :: 3 :: [] = [2; 3]` well-formed? If so, what is its value? If it's not well-formed, what's wrong with it?

Answer:

Yes, the value is `true`.

4. (1 Point) True or false, the code below is well-formed and after (1) but before (2), the storage diagram is as shown.

```
let f x =
  let y = (2, x)          (1)
  in
  y                       (2)

f 3

  Stack           Heap
+--------+
| f      |
|   x: 3 |<-----------------+
|        |       +-----+    |
|   y: o-+------->|  2  |    |
|        |       +-----+    |
+--------+       |  o--+---+
                 +-----+
```

Answer:

The code is well-formed but the diagram is wrong.

5. (2 Points) Is the following definition of `mystery` well-formed? If so, what is its type? If it's not well-formed, what's wrong with it?

```
let mystery x y z = (x y, y z)
```

Answer:

Yes, `mystery : (('a -> 'b) -> 'c) -> ('a -> 'b) -> 'a -> ('c * 'b)`

6. (1 Point) Is `let f x = (x *. 2.0, x * 2)` well-formed? If so, what is its type? If it's not well-formed, what's wrong with it?

Answer:

It's ill-formed because `x` can't be both an `int` and a `float`.

7. (1 Point) Consider binary trees of integers.

```
type bt = Empty
        | Node { info  : int
               ; left  : bt
               ; right : bt
               }
```

Maria's application will require a very large binary tree `a` with a large left child `b` and right child `c`.

```
let a = Node { info  = ...                  ...
             ; left  = b                    / \
             ; right = c                   b   c
             }
```

She'd like to save space in the following way: when `b` and `c` are identical trees, she plans to use the same tree for both left and right.

```
let a = Node { info  = ...                  ...
             ; left  = b                    ( )
             ; right = b                     b
             }
```

Will this work? If not, what is the problem with it?

Answer:

This will work fine if `b` and `c` are immutable. However, if something in one of those trees is mutable it will not work.

2 Storage Diagrams (8 Points)

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

```
let go a =
  let x = (a, 10, [2; 4; 6])                (1)
  in
  x                                         (2)

go [| 2; 4; 6; 8 |]
```

```
        Stack                                               Heap
```

Answer:

```
  Stack                          Heap
+----------+
|go        |
|     a o-+---------------------------------> +--------+
|          |         +--------+       +--------> |   2    |
|     x o-+--------> |    o---+-------+          +--------+
|          |         +--------+                  |   4    |
+----------+         |   10   |                  +--------+
                     +--------+                  |   6    |
                     |    o   |                  +--------+
                     +---+----+                  |   8    |
                         |                       +--------+
                         v
                       +---+---+   +---+---+   +---+---+
                       | 2 | o-+-> | 4 | o-+-> | 6 | o-+-+
                       +---+---+   +---+---+   +---+---+ =
```

2. (3 Points) Let `filter` be defined in the usual way:

```
let rec filter test xs =
  match xs with
  | [] -> []
  | x :: xs ->
    let ys = filter test xs
    in
    if (test x) then x :: ys else ys
```

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

```
let go ms =
  let ns = filter (fun (a, b) -> a = b) ms          (1)
  in
  ns                                                  (2)

go [(2, 2); (3, 4); (5, 5)]

        Stack                                                      Heap
```

Answer:

```
+---------+
|go       |      +---+---+   +---+---+   +---+---+
|    ms o-+----> | o | o-+-> | o | o-+-> | o | o-+-+
|         |      +-|-+---+   +-|-+---+   +-|-+---+ =
|         |        v           v           v
|         |      +-------+   +-------+   +-------+
|    ns o-+-+    |   2   |   |   3   |   |   5   |
|         | |    +-------+   +-------+   +-------+
+---------+ |    |   2   |   |   4   |   |   5   |
            |    +-------+   +-------+   +-------+
            |        ^                       ^
            +----> +-|-+---+               +-|-+---+
                   | o | o-+-------------> | o | o-+-+
                   +---+---+               +---+---+ =
```

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

3. (3 Points) Let the "remove first member" function `rember` be defined as:

```
let rec rember x xs =
  match xs with
  | [] -> []
  | y :: ys -> if x = y then ys else y :: rember x ys

let go xs =
  let ys = rember 3 xs                          (1)
  in
  ys                                            (2)

go [2; 3; 3]
```

```
        Stack                                                   Heap
```

Answer:

```
+----------+
|go        |      +---+---+   +---+---+   +---+---+
|    xs o-+----> | 2 | o-+-> | 3 | o-+-> | 3 | o-+-+
|          |      +---+---+   +---+---+   +---+---+ =
|          |                                ^
|          |      +---+---+                 |
|    ys o-+----> | 2 | o-+-----------------+
|          |      +---+---+
+----------+
```

3 Coding (18 Points)

1. (3 Points) Write a function `truncate : 'a list -> int -> 'a list` such that when called as in `(truncate xs n)`, `truncate` returns a list like `xs` but has no more than the last `n` elements. For example, the call `(truncate [2; 4; 6; 8] 2)` should return `[6; 8]`, `(truncate [2; 4; 6; 8] 10)` should return `[2; 4; 6; 8]`.

   Answer:

```
let rec truncate xs n =
  match List.length xs <= n with
  | true  -> xs
  | false -> truncate (List.tl xs) (n - 1)
```

2. (3 Points) Write a function `tripsToPairs : int * int * int list -> int * int list` such that a call `(tripsToPairs [(m1, m2, m3); (n1, n2, n3); ...]`, returns the list of pairs

```
[(m1 + m2, m3 + m2); (n1 + n2, n3 + n2); ...]
```

Answer:

```
let rec tripsToPairs trips =
  match trips with
  | [] -> []
  | (n1, n2, n3) :: pairs ->
    (n1 + n2, n2 + n3) :: tripsToPairs pairs
```

3. (3 Points) In Python, lists are arrays. Python lists have a handy *slice* operation that creates a new array from a slice of an existing array. If `a` is the 4-element array `[2, 4, 6, 8]` then `a[1:3]` is a fresh 2-element array `[4, 6]`. Write the function `slice : 'a array -> int -> int -> 'a array` such that a call `(slice a lo hi)` returns a fresh array with values `a.(lo)` up to `a.(hi - 1)`. For `(slice a lo hi)` where `lo = hi`, return the 0-length array `[||]`.

Answer:

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

4. (3 Points) Write a function `arrayAppend : 'a array -> 'a array -> 'a array` such that for arrays `a` and `b`, the call `(arrayAppend a b)` returns a fresh array containing all of the elements of `a` followed by all of the elements of `b`.

Answer:

```
let appendArray a b =
  let (m, n) = (Array.length a, Array.length b) in
  let c = Array.make (m + n) a.(0)
  in
  for i = 0 to m - 1 do
    c.(i) <- a.(i)
  done;
  for i = 0 to n - 1 do
    c.(m + i) <- b.(i)
  done;
  c
```

5. (4 Points or 6 Points) There's a relatively easy 4 point version and a somewhat more challenging 6 point version. Consider binary trees of integers.

```
type bt = Empty                                      t0 = 2     t1 = 0          t2 = 0
        | Node of { info  : int                                  / \              /
                  ; left  : bt                                  2   2            0
                  ; right : bt                                                  / \
                  }                                                            2   2

let t0 = Node { info = 2; left = Empty; right = Empty }  t3 = 0          t4 = 0
let t1 = Node { info = 0; left = t0; right = t0 }          /   \           /   \
let t2 = Node { info = 0; left = t1; right = Empty }      0     0         1     1
let t3 = Node { info = 0; left = t1; right = t1 }        / \   / \       / \   / \
                                                        2   2 2   2     2   2 2   2
let t4 = Node { info = 0
              ; left  = Node { info = 1; left = t0; right = t0 }
              ; right = Node { info = 1; left = t0; right = t0 }}
```

A binary tree is *perfect* if every level is full. Of the above, `t0`, `t1`, `t3` and `t4` are perfect. `t2` is not. This problem involves writing a function `makePerfectBT : int -> bt` such that a call `(makePerfectBT n)` makes a perfect binary tree of height `n`. For the 4-point version, all of the `info` fields can be `0`. For the full 6 points, the `info` field of a given node should record the depth of the node, as in `t4`.

Answer:

```
let rec makePerfectBT n =                                   (* Easy version *)
  match n = 0 with
  | true  -> Node { info = 0; left = Empty; right = Empty }
  | false ->
    let subtree = makePerfectBT (n - 1)
    in
    Node { info  = 0
         ; left  = subtree
         ; right = subtree
         }

let makePerfectBT n =                                       (* Less easy version *)
  let rec makePerfectBT i =
  match i = n with
  | true  -> Node {info = i; left = Empty; right = Empty}
  | false ->
    let subtree = makePerfectBT (i + 1)
    in
    Node { info  = i
         ; left  = subtree
         ; right = subtree
         }
  in
  makePerfectBT 0
```

4 The Simple Virtual Machine (6 Points)

The SVM instruction set is specified on the attached sheet. The data segment has a sequence of integers with a 0 sentinel. Register `R0` has an integer `n`. Write an SVM procedure that replaces all occurrences of the integer `3` in the data segment by `n`.

Answer:

```
# Data segment has integers with a 0 sentinel. R0 has n.
# Replace all occurrences of 3 in the data segment with n.
#
Mov R2, Zero
Lod R3, 0(R2) <-+
Cmp R3, R0      |
Beq 8           |  o----------+
Li  R1, 3       |             |
Cmp R3, R1      |             |
Beq 3           |  o---+      |
Li  R1, 1       |      | <-+  |
Add R2, R2, R1  |      |   |  |
Jmp -9     o----+      |   |  |
Sto R0, 0(R2)      <---+   |  |
Jmp -5             o-------+  |
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.