Final Exam (A)
CSCI 1103 Computer Science 1 Honors

KEY

Friday December 13, 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) Is `((*) 2)` well-formed? If so, what is the type?

   Answer:

   `Yes, int -> int`

2. (1 Point) How does OCaml parenthesize the expression `a b c`? Is it `(a b) c` or is it `a (b c)`?

   Answer:

   `(a b) c`

3. (1 Point) Let the type `t = { a : int -> int; b : int }` and let the variable `r` be defined as `r = { a = fun x -> x * 2; b = 12}`. Is `r.a` well-formed? If so, what is its type? If it's not well-formed, what's wrong with it?

   Answer:

   Yes, `r.a : int -> int`

4. (1 Point) Let the type `t = { a : int -> int; b : int }` and let the variable `r` be defined as `r = { a = fun x -> x * 2; b = 12}`. Is `(r.a r.b)` well-formed? If so, what is its type? If it's not well-formed, what's wrong with it?

   Answer:

   Yes, `r.a r.b : int`

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 z)
   ```

   Answer:

   Yes, `mystery : ('a -> 'b) -> ('c -> '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) 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) Let `map` be defined in the usual way:

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

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

```
let go ms =
  let ns = map (fun n -> n * 2) ms             (1)
  in
  ns                                           (2)

go [2; 3; 4]
```

```
        Stack                                                Heap
```

Answer:

```
+----------+
|go        |      +---+---+   +---+---+   +---+---+
|    ms o--+----> | 2 | o-+-> | 3 | o-+-> | 4 | o-+-+
|          |      +---+---+   +---+---+   +---+---+ =
|          |
|          |      +---+---+   +---+---+   +---+---+
|    ns o--+----> | 4 | o-+-> | 6 | o-+-> | 8 | 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-+-+
                               +---+---+ =
```

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 `copy : 'a -> int -> 'a list` such that a call `(copy item n)` returns a list of `n` copies of `item`. For example, `(copy 2 3)` should return the list `[2; 2; 2]`. The call `(copy 2 0)` should return the list `[]`. You may assume that `n` is non-negative.

   Answer:

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

let copy item n =
  let rec loop n answer =
    match n = 0 with
    | true  -> answer
    | false -> loop (n - 1) (item :: answer)
  in
  loop n []
```

2. (3 Points) Write a function `cutPrefix : 'a list -> int * 'a list` such that when `cutPrefix` is called as in `(cutPrefix [a; a; a; b; ...])`, it returns a pair `(n, ys)`. The `cutPrefix` function counts the length of the run of identical values in the front of the input list and returns that length together with the list values after the run. For example, the call `(cutPrefix [5; 5; 3; 5; 5])` should return the pair `(2, [3; 5; 5])` – this says there was a run of 2 consecutive `5`s followed by the list items `[3; 5; 5]`. The call `(cutPrefix [])` should return the pair `(0, [])` and the call `(cutPrefix [5])` should return the pair `(1, [])`.

Answer:

```
let rec cutPrefix xs =
  match xs with
  | [] -> (0, [])
  | [_] -> (1, [])
  | x1 :: x2 :: xs ->
    match x1 = x2 with
    | true  ->
      let (n, ys) = cutPrefix (x2 :: xs)
      in
      (n + 1, ys)
    | false -> (1, x2 :: xs)
```

3. (4 Points) Many data-centric applications have long sequences of common values. For example, silence in auditory processing is often represented by a sequence of zero amplitudes

```
[0; 0; 0; 2; 0; 0; 0; 0; 1; 1; ...]
```

A simple way to compress data of this kind is to use *run length encoding* (RLE). In RLE, the above data would be represented as a list of pairs `(value, count)`, where `count` indicates the length of the run. The example above would be encoded as `[(0, 3); (2, 1); (0, 4); (1, 2); ...]`.

Write a function `rlDecode : ('a * int) list -> 'a list` such that a call decodes the list of pairs.

Answer:

```
let rec rlDecode pairs =
  match pairs with
  | [] -> []
  | (n, x) :: pairs ->
    (copy x n) @ rlDecode pairs
```

4. (4 Points) Write the run length encoding function `rlEncode : 'a list -> ('a * int) list`. A call `(rlEncode xs)` should return a list of pairs of the form `(value, count)` as described in the previous problem.

Answer:

```
let rec rlEncode xs =
  match xs with
  | [] -> []
  | y :: _ ->
    let (n, zs) = cutPrefix xs
    in
    (y, n) :: rlEncode zs
```

5. (4 Points) Write a function `insert : 'a -> 'a array -> 'a array`. In a given call `(insert item a)`, the array `a` is full of items that are known to be in ascending order as determined by the pervasive `compare` function. The `insert` function should return a new array that is exactly like `a` but which contains `item` inserted in its proper spot. For example, the call `(insert 5 [| 2; 4; 8 |])` should return an array `[| 2; 4; 5; 8 |]`.

Answer:

```
let insert item a =
  let b = Array.make (Array.length a + 1) a.(0) in
  let i = ref 0
  in
  while !i < Array.length a && compare a.(!i) item < 0 do
    b.(!i) <- a.(!i);
    i := !i + 1
  done;
  b.(!i) <- item;
  while (!i < Array.length a) do
    b.(!i + 1) <- a.(!i);
    i := !i + 1
  done;
  b
```

4 The Simple Virtual Machine (6 Points)

The SVM instruction set is specified on the attached sheet. The function call `(Code.range n)` generates a list of integers running from 0 to `n - 1`. Write this function in SVM leaving the numbers in consecutive words in the data segment. The input `n` can be found in register `R0`.

Answer:

```
# This is the range function. Register R0 contains n.
#
Li  R1, 1
Mov R2, Zero
Cmp R2, R0
Beq 3
Sto R2, 0(R2)
Add R2, R2, R1
Jmp -5
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.