Second Midterm Exam
CSCI 1103 Computer Science 1 Honors

KEY

Thursday November 17, 2022
Instructor Muller
Boston College

Fall 2022

Before reading further, please arrange to have an empty seat on either side of you if possible. Now that you're seated, please write your name on the **back** of this exam.

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		3
2 Working with Lists		6
2 Binary Trees		6
4 SVM		5
Total		20

1 Snippets (3 Points Total)

1. (1 Point) Let's say we have `type person = {age : int; resident : boolean}`. Is the following function definition well-typed? If so, what is its type? If not, what is wrong with it?

```
let f test a =
  match (test a) with
  | true  -> a.age
  | false -> a.resident
```

Answer:

This is ill-typed, type because `a.age` is an `int` while `a.resident` is a `boolean`.

2. (1 Point) With `type t = {m : int; n : int}` and `let f a = a.m > a.n`, is the following well-formed? If so, what is its type? Show the step-by-step evaluation if there is one.

```
match (f {m=3; n=2+3}) with | true -> "A" | false -> "B"
```

Answer:

```
Yes it is of type string.

match f {m=3; n=2+3} with | true -> "A" | false -> "B" ->
match f {m=3; n=5} with | true -> "A" | false -> "B" ->
match {m=3; n=5}.m > {m=3; n=5}.n with | true -> "A" | false -> "B" ->
match 3 > {m=3; n=5}.n with | true -> "A" | false -> "B" ->
match 3 > 5 with | true -> "A" | false -> "B" ->
match false with | true -> "A" | false -> "B" ->
"B"
```

3. (1 Point) With `type t = {m : int; n : int}` is `(fun i -> {m = i * 2; n = 3 * 2})` well-typed? If so, what is its type? Is it a value?

Answer:

Yes, it's a function value of type `int -> t`**.**

2 Working with Lists (6 Points)

1. (2 Point) The function `isAscending : int list -> boolean` returns `true` if the input list is in strictly ascending order. Otherwise it returns `false`. For example, the call `(isAscending [1; 2; 4])` should return `true`. In general, `isAscending` should return `true` for any list of length less than 2. The call `(isAscending [1; 2; 2; 4])` would return `false`. Write the function `isAscending`.

   Answer:

   ```
   let rec isAscending ns =
     match ns with
     | [] | [n] -> true
     | n1 :: n2 :: ns -> (n1 < n2) && isAscending (n2 :: ns)
   ```

2. (4 Points) Write a function `sublists : a list -> (a list) list`. A call `(sublists xs)` should return a list containing all sub-lists of `xs`. For example, the call `(sublists [1; 2])` should return a list like `[[]; [1]; [2]; [1; 2]]` while the call `(sublists [1; 2; 3])` should return a list like

   ```
   [[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]]
   ```

   Note that `(sublists [])` should return `[[]]`.

   Answer:

   ```
   let rec sublists xs =
     match xs with
     | [] -> [[]]
     | x :: xs ->
       let ans = sublists xs
       in
       ans @ (List.map (fun xs -> x :: xs) ans)
   ```

3 Binary Trees (6 Points)

The problems in this section relate to binary trees where the nodes contain integers.

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

1. (2 Points) Write a function `evenTree : tree -> boolean` such that a call `(evenTree tree)` returns `true` if all of the integers in the tree are even. `evenTree` should return `false` if any integer in the tree is odd. (Hint: the `mod` operator will come in handy.)

   Answer:

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

   let rec evenTree tree =
     match tree with
     | Empty -> true
     | Node {info; left; right} ->
       (info mod 2 = 0) && (evenTree left) && (evenTree right)
   ```

2. (1 Point) What is the invariant that must hold for a binary tree to be a *binary search tree*?

   Answer:

   The info field of each node should be greater than the info fields of all nodes to the left and less than the info fields of all nodes to the right.

3. (1 Point) The following `inorder : tree -> int list` function gathers the integers in a binary tree into a list by visiting the nodes "in order". That is, it first visits all of the nodes to the left, then visits the node, it then visits all of the nodes to the right. (NB: `@` is the built-in list append operator.)

```
let rec inorder tree =
  match tree with
  | Empty -> []
  | Node {info; left; right} -> (inorder left) @ [info] @ (inorder right)
```

True or False: `tree` is a binary search tree if and only if `(isAscending (inorder tree))`.

Answer:

True

4. (2 Points) Assume `tree` is a binary search tree. Write the function `min : tree -> int` which returns the smallest integer in the tree. If the tree is empty, the `min` function should `failwith`.

Answer:

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

let rec min tree =
  match tree with
  | Empty -> failwith "min: empty tree"
  | Node {info; left=Empty} -> info
  | Node {left} -> min left
```

4 The Simple Virtual Machine (5 Points)

1. (3 Points) The SVM instruction set is specified on the attached sheet. Register R0 contains a positive integer `n`. Write an SVM program that halts with the memory containing the range `0,.., n-1`.

   Answer:

   ```
   Li  R0, 5;
   Mov R2, Zero;
   Li  R1, 1;
   Cmp R2, R0;
   Beq 3;
   Sto R2, 0(R2);
   Add R2, R2, R1;
   Jmp (-5);
   Hlt;
   ```

2. (2 Points) SVM instructions are designed to occupy 2-bytes or 16 bits of memory. There are 16 operations (`Mov`, `Lod`, ...) and 8 registers. Consider the instruction `Lod Rd, offset(Rs)`. The `offset` is a two's complement integer, it can be positive, zero or negative. What is the range of `offset`? (Hint: how many bits are available for the `offset` field?)

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.