Final Exam
CSCI 1101 Computer Science I

KEY

Wednesday December 16, 2015
Instructor Muller
Boston College

Fall 2015

Please do not write your name on the top of this test. 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 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		6
2 Storage Diagrams		4
3 Repetition		14
4 SVM		5
Total		29

1 Snippets (6 Points Total)

1. (2 Points) OCaml's `Char` module has a pair of functions for navigating back and forth between characters and their ASCII codes. The `Char.chr : int -> char` function accepts an integer ASCII code for a character and returns the character. For example, the call `(Char.chr 65)` evaluates to the character `'A'`. OCaml's `Char.code : char -> int` function goes the other way — it returns the ASCII code of its argument.

   Recall that the `List.map : ('a -> 'b) -> 'a list -> 'b list` function is defined as in:

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

   Is the expression `(List.map Char.chr)` well-typed? If so, what is its type? And, in a sentence, what does it do?

   Answer:

   Yes, the type is `int list -> char list`. It converts lists of ASCII codes to their characters.

2. (2 Points) True or false? The expressions `1::2::[]` and `[1; 2]` are represented identically as a chain of "cons" nodes as in the storage diagram:

   ```
             +---+---+     +---+---+
      o----->| 1 | o-+---->| 2 | o-+--+
             +---+---+     +---+---+  =
   ```

   Answer:

   True

3. (2 Points) Solve for X. $X_{16} = BC_{16} + 10111100_2$.

   Answer:

   X = 178

2 Storage Diagrams (4 Points)

Show the state of the Stack and the Heap after `(1)` has executed but before `(2)` has executed. Note: between `(1)` and `(2)` the function `append` is called. There is no need to show any of the stack records for the calls of `append` since these records would be popped off the stack at the completion of `(1)`. This question is particularly concerned with the value of `zs`.

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

let go b c =
  let xs = [b; c] in
  let ys = [3; 4; 5] in
  let zs = append xs ys              (1)
  in
  zs                                 (2)

go 1 2
```

```
        Stack                                                Heap
```

Answer:

```
+----+
| go |
+----+---+
|  b | 1 |                   +---+---+    +---+---+
+----+---+             +---->| 1 | o-+--->| 2 | o-+--+
|  c | 2 |             |     +---+---+    +---+---+  =
+----+---+             |
| xs | o-+-------------+
+----+---+                                    +---+---+    +---+---+    +---+---+
| ys | o-+----------------------------------->| 3 | o-+--->| 4 | o-+--->| 5 | o-+--+
+----+---+                                    +---+---+    +---+---+    +---+---+  =
| zs | o-+-------------+                        ^
+----+---+             |     +---+---+    +---+-|-+
                       +---->| 1 | o-+--->| 2 | o |
                             +---+---+    +---+---+
```

3 Repetition (14 Points Total)

1. (4 Points) Write **two versions** of an OCaml function `powersOfTwo : float list -> float list` such that a call `(powersOfTwo ns)` returns a list containing 2.0 raised to the `n` for each `n` in `ns`. For example, the call `(powersOfTwo [8.; 4.; 3.])` should return the list `[256.; 16.; 8.]`.

   The two versions should use different repetition idioms. For example, one version might use recursion while the other might use an imperative loop (`for` or `while`) or `List.map`.

   Answer:

```
let powersOfTwo ns = List.map (fun n -> 2.0 ** n) ns

let rec powersOfTwo ns =
  match ns with
  | [] -> []
  | n::ns -> (2.0 ** n)::powersOfTwo ns

let powersOfTwo ns =
  let answer = ref []
  in
  for i = 0 to (List.length ns - 1) do
    answer := (2.0 ** (List.nth ns i))::(!answer)
  done;
  !answer

let powersOfTwo ns =
  let answer = ref [] in
  let i = ref 0
  in
  while !i < (List.length ns) do
    answer := (2.0 ** (List.nth ns !i))::(!answer);
    i := !i + 1
  done;
  !answer
```

2. (3 Points) Characters, i.e., values of type `char`, can be compared to one another using OCaml's pervasive relational operators `=`, `<`, `<=` etc. For example, the expression `('A' < 'B')` evaluates to `true`.

   Write a function `charsFromTo : char -> char -> char list` such that a call `(charsFromTo lo hi)` returns the list of characters running from `lo` up to `hi`. For example, the call `(charsFromTo 'M' 'P')` should return the list `['M'; 'N'; 'O'; 'P']`. If `lo` isn't less than or equal to `hi`, `charsFromTo` should return the empty list.

   Answer:

```
let rec fromTo lo hi =
  match lo > hi with
  | true  -> []
  | false -> lo::fromTo (lo + 1) hi

let charsFromTo lo hi = List.map Char.chr (fromTo (Char.code lo) (Char.code hi))

let rec charsFromTo lo hi =
  match lo > hi with
  | true  -> []
  | false -> lo::charsFromTo (Char.chr (Char.code lo + 1)) hi
```

3. (3 Points) Write a function `compactList : int list -> int list` such that a call `(compactList ns)` returns a list exactly like `ns` but in which all of the zeros are at the end. For example, the call `(compactList [1; 0; 2; 0; 3])` should evaluate to the list `[1; 2; 3; 0; 0]`. Note that the non-zero values must remain in their original order.

Answer:

```
let compactList ns =
  let xs = List.filter (fun n -> n <> 0) ns in
  let ys = List.filter (fun n -> n =  0) ns
  in
  xs @ ys
```

4. (4 Points) Write an OCaml function `compactArray : int array -> unit` such that a given call `(compactArray ns)` modifies (i.e., mutates) the array `ns` moving all zeros to the end. For example, the call `(compactArray [| 1; 0; 2; 0; 3 |])` should evaluate to `()` but should leave the array `ns` in the following order `[| 1; 2; 3; 0; 0 |]`. As in the previous problem, the non-zero values must remain in their original order.

Answer:

```
let compactArray a =
  let b = Array.from_list (compactList (Array.to_list a))
  in
  for i = 0 to (Array.length a - 1) do
    a.(i) <- b.(i)
  done


let compactArray a =
  let b = Array.make (Array.length a) 0 in
  let j = ref 0
  in
  for i = 0 to (Array.length a - 1) do
    b.(!j) <- a.(i);
    if (a.(i) <> 0) then j := !j + 1 else ()
  done;
  for i = 0 to (Array.length a - 1) do
    a.(i) <- b.(i)
  done
```

4 The Simple Virtual Machine (5 Points)

In October we learned about storage and about the basic machine model that underlies all of our computers. We saw the assembly language for a simple virtual computer (SVM) with a RAM and a CPU with an ALU and 6 registers: `pc`, `psw`, `R0`, ..., `R3`. The SVM instruction set is specified on the attached sheet. Note that for the purposes of this problem, we have extended the SVM instruction set with a `MOD` instruction.

Write an SVM program implementing Euclid's greatest common divisor algorithm:

```
let rec gcd (m, n) =
  match n = 0 with
  | true  -> m
  | false -> gcd(n, m mod n)
```

With the understanding that the inputs `m` and `n` are in `RAM[0]` and `RAM[1]` (resp.), the code should halt with the greatest common divisor in register `R0`.

Answer:

```
Euclid's gcd algorithm:

LOD R0, 0       # R0 <- m
LOD R1, 1       # R1 <- n
CPZ R1          # n =? 0
BEQ 4
MOD R2, R0 R1   # tmp <= m mod n
MOV R0, R1      # m <= n
MOV R1, R2      # n <= the mod
JMP -6
HLT
```

This page is nearly blank.

5 The Simple Virtual Machine

The instruction set of SVM is as follows.

- `LOD Rd, addr`: loads RAM[addr] into register Rd.
- `STO Rs, addr`: stores the contents of register Rs into location addr in the memory.
- `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.
- `MOD Rd, Rs, Rt`: divides the contents of register Rs by Rt and stores the integer remainder 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.
- `CPZ Rs`: sets PSW = Rs. If Rs == 0, then PSW will be 0, otherwise it will be non-zero.
- `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.
- `HLT`: causes the svm machine to print the contents of registers PC, PSW, R0, R1, R2 and R3. It then stops, returning ().