CMSC330 Spring 2013 Midterm #2 Solutions

1. (16 pts) OCaml Types and Type Inference

   Give the type of the following OCaml expressions:

   a. (3 pts) (fun c a t -> 5) 6 7 **Type = 'a -> int**
   b. (3 pts) fun d o g -> [g o] **Type = 'a -> 'b -> ('b -> 'c) -> 'c list**

   Write an OCaml expression with the following type:

   c. (3 pts) (int -> int -> 'a) -> 'a **Code =** fun x -> x 2 3
   d. (3 pts) 'a -> (int -> 'b) -> 'b list **Code =** fun x y -> [y 3]

   Give the value of the following OCaml expressions. If an error exists, describe it

   e. (2 pts) (fun x -> fun y -> x y) (fun z -> z + 2) 3 **Value = 5**
   f. (2 pts) (fun x y -> x + y) (2 3) **Error = 2 is not a function**

2. (8 pts) OCaml programming

   Write a function *nth* which given a number *n* and a list *lst*, returns the *n*th element of *lst*. You may assume the list has at least n members.

   You may not use any library functions, with the exception of the List.rev function, which reverses a list in linear time. Your function must run in linear time (i.e., not use append/reverse for every element of the list). You may use helper functions.

   Examples:
   nth 1 [4;5;6;7] = 4
   nth 2 [4;5;6;7] = 5
   nth 3 [4;5;6;7] = 6
   nth 2 [[1;2] ; [3;4] ; [5]] = [3;4]

   ```
   let rec nth n lst = match lst with
           (h::t) -> if n = 1 then h else nth (n-1) t
   ```

   OR

   ```
   let rec nth n (h::t) = if n = 1 then h else nth (n-1) t
   ```

3. (14 pts) OCaml higher-order & anonymous functions

Using either map or fold and an anonymous function, write a curried function *partition* which when given a predicate function f and a list x, returns a tuple of two lists y, z, where y is a list of all the members of x for which f is true, and z is a list of all the members of x for which f is false. The members of y and z must be in the same relative ordering as in x.

Your function must run in linear time. You may not use any library functions, with the exception of the List.rev function, which reverses a list in linear time. You may not use imperative OCaml (i.e., no ref variables).

Example:
partition (fun x -> true) [1;2;3] = ([1;2;3],[ ])
partition (fun x -> x = 0) [1;2;3] = ([ ],[1;2;3])
partition (fun x -> x < 2) [1;2;3] = ([1],[2;3])
partition (fun x -> x > 2) [1;2;3] = ([3],[1;2])

```
let rec map f l = match l with
        [ ] -> [ ]
      | (h::t) -> (f h)::(map f t)
```

```
let rec fold f a l = match l with
        [ ] -> a
      | (h::t) -> fold f (f a h) t
```

let partition f lst =
let (a,b) = fold (fun (x,y) h -> if (f h) then ((h::x),y) else (x,(h::y))) ([],[]) lst in
(List.rev a, List.rev b)

OR

let partition f lst =
fold (fun (x,y) h -> if (f h) then ((h::x),y) else (x,(h::y))) ([],[]) (List.rev lst)

4. (14 pts) OCaml polymorphic types

Consider the OCaml type *boolExp* implementing boolean expressions:

```
type boolExp =
      True
    | False
    | And of boolExp * boolExp
    | Not of boolExp
```

a. (2 pts) Write an OCaml expression with type boolExp that is equivalent to the expression "true && (not false)"

And (True, Not False)

b. (12 pts) Write a function *evalExpr* of type (boolExp -> bool) that takes an Boolean expression tree and calculates its value. Your code must work in linear time (i.e., avoid multiple passes over the tree). You are not allowed to use any OCaml library functions except &&, ||, and *not*. You may use helper functions.

Examples:

```
let x = True;;                (* (evalExpr x) returns true  *)
let y = False;;               (* (evalExpr y) returns false *)
let z = Not x;;               (* (evalExpr z) returns false *)
let p = And (x,x);;           (* (evalExpr p) returns true  *)
let q = And (p,z);;           (* (evalExpr q) returns false *)
let r = Not q;;               (* (evalExpr r) returns true  *)
```

```
let rec evalExp b = match b with
      True -> true
    | False -> false
    | And (b1, b2) -> (evalExp b1) && (evalExp b2)
    | Not b1 -> not (evalExp b1)
```

5. (10 pts) Context free grammars.

Consider the following grammar (S = start symbol and terminals = **0 1 ^ # !**):

S → S^S | S#S | S! | E

E → 0 | 1

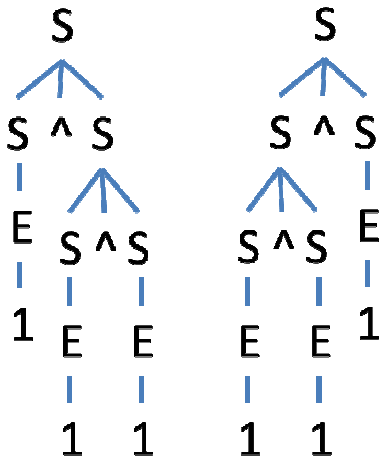
a. (1 pt each) Indicate whether the following strings are generated by this grammar

i.	**^1**	Yes	**No**	(circle one)
ii.	**0#1!^1**	**Yes**	No	(circle one)
iii.	**0!^1!**	**Yes**	No	(circle one)

b. (3 pts) Draw a parse tree for the string "1#0!"

c. (4 pts) Is the grammar is ambiguous? Provide proof if you believe it is ambiguous.

Yes, the grammar is ambiguous. Can come up with 2 parse trees for strings such as 1^1^1.



6. (14 pts) Using context free grammars.

   a. (6 pts) Given the following grammar for generating OCaml *bools*, create a grammar that generates OCaml expressions of type *bool list.* Your grammar should be able to generate strings such as "[]", "[true]", "[false]", "[true;true]", etc...

      B → *true* | *false*

      S → [ ] | [A]
      A → B ; A | B

   b. (8 pts) Consider the grammar from Problem 5 again:

      S → S^S | S#S | S! | E
      E → 0 | 1

      Modify the grammar to make the # operator *left associative* and the ^ operator *right associative*. Make ! have the highest precedence, # the lowest precedence, and ^ medium precedence.

      S → S#A | A
      A → B^A | B
      B → E! | E
      E → 0 | 1

7. (14 pts) Parsing

Consider the following grammar, where S, A, B are nonterminals, and a, b, c are terminals.

S →	AB	|	Ac
A →	aA	|	epsilon
B →	b	|	epsilon

a. (8 pts) Calculate FIRST sets for S, A, and B

FIRST(S) = { a,b,c,epsilon }
FIRST(A) = { a, epsilon }
FIRST(B) = { b, epsilon }

b. (6 pts) Using pseudocode, write *only* the parse_A function found in a recursive descent parser for the grammar. You may assume the functions parse_S , parse_B already exist.

Use the following utilities:

lookahead	Variable holding next terminal
match (x)	Function to match next terminal to x
error ( )	Reports parse error for input

```
parse_A( ) {    // your code starts here

        if (lookahead == "a") {        // A → aA
                match("a");
                parse_A( );
        } else                         // A → epsilon
                ;
}

                OR

parse_A( ) {    // your code starts here

        if (lookahead == "a") {        // A → aA
                match("a");
                parse_A( );
        }                              // else A → epsilon
}
```

8. (10 pts) Multithreading

Consider the following multithreaded Java 1.4 code. Assume there are multiple producer and consumer threads being executed in the program.

```
class Buffer {
  Buffer ( ) {
    Object buf = null;
    boolean empty = true;
  }
```

```
  void produce(o) {
    synchronize (buf) {
1.    while (!empty) wait( );
2.    empty = false;
3.    notifyAll( );
4.    buf = o;
    }
  }
```

```
  Object consume( ) {
    synchronize (buf) {
5.    while (empty) wait( );
6.    empty = true;
7.    notifyAll( );
8.    return buf;
    }
  }
}
```

a. (3 pts) Is it possible given two threads x and y for the last statement executed by both threads to be statement 2 in the code above? Explain your answer.

No, since any thread x reaching statement 2 must be holding the lock for buf. So no other thread can execute statement 2 until thread x finishes executing statements 3 & 4 and releases the lock for buf.

b. (3 pts) Is it possible given two threads x and y for the last statement executed by thread x to be statement 2, and the last statement executed by thread y to be statement 6? Explain your answer.

No, since any thread x reaching statement 2 must be holding the lock for buf. The code in consume( ) is synchronizing on the same lock for buf, even though it is a different function. So no other thread can execute statement 6 until thread x finishes executing statements 3 & 4 and releases the lock for buf.

c. (4 pts) Is it possible given two threads x and y for the last statement executed by thread x to be statement 1, and the last statement executed by thread y to be statement 2? Explain your answer.

Yes, since thread x reaching statement 1 may go to sleep after calling wait( ) if the buffer is full, which releases the lock for buf. A consumer thread may consume the value and change empty to true. Thread y may then acquire the lock (ahead of x) and execute statements 2. It is also possible that both threads x & y execute statement 1 and go to sleep. When notifyAll() is called by a consumer, y wakes up first and reaches statement 2. (Note that it is not possible for y to execute statement 2 first, followed by x executing statement 1, since y will be holding the lock at statement 2).

Note: The preceding answers assume that threads x, y are executing the produce() and/or consume() methods for the same Buffer object, so that *buf* is referring to the same object (and thus synchronize(buf) is acquiring the same lock). If one assumes that threads x, y are executing code for two different Buffer objects, then they are synchronizing on different locks. In this case the answer for all three questions is Yes since any statement interleaving is possible.