CMSC330 Spring 2014 Midterm 2 Solutions

1. (16 pts) OCaml Types and Type Inference

   Give the type of the following OCaml expressions:

   a. (2 pts) fun b -> b + 8 **Type = int -> int**
   b. (3 pts) fun b -> b 8 **Type = (int -> 'a) -> 'a**

   Write an OCaml expression with the following type:

   c. (2 pts) 'a -> 'a -> 'a list list **Code = fun x y -> [[x ; y]]**
   d. (3 pts) (int -> int) -> (int -> int) **Code = fun x y -> (x 1) + y**

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

   e. (3 pts) (fun k t m -> [m; t]) 2 4 6 **Value / Error = [6 ; 4]**
   f. (3 pts) (fun (k::t::m) -> (m, t)) [2;4;6;8] **Value / Error = ([6;8] , 4)**

2. (10 pts) OCaml higher-order & anonymous functions

   Using either map or fold and an anonymous function, write a curried function *process* which when given a predicate function **p**, a compute function **f**, and a list **lst**, returns a list of the results of applying **f** to the elements of lst, but only where **p** is true for that element of the list. If **p** is false for an element of the list, it does not contribute to the final result. The relative order of the elements in the new list must the same as in **lst**.

   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). For example, the results of calling *process* will be as follows:

   Example:
   let p x = x > 2;;
   let f x = x+1;;
   process p f [ ] = [ ]
   process p f [1; 2] = [ ]
   process p f [1; 3; 2; 4] = [4; 5]

```
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 process p f lst = List.rev (fold (fun a h -> if (p h) then (f h)::a else a) [ ] lst)

3. (18 pts) OCaml programming

Write a function ***allPairs*** with type (‘a list -> ‘a list list) which given a list *lst* with n elements, returns a list of all n^2 pairs of elements in *lst* (in any order), where each pair of elements is in a list of length 2.

You may not use any library functions, with the exception of the @ function (which concatenates two lists). You may use helper functions. You may use map or fold, but it is not required. Your function must execute in polynomial time. You may not use imperative OCaml constructs.

Examples:

```
allPairs [ ] ;;        (* = [ ] *)
allPairs [1] ;;        (* = [ [1;1] ] *)
allPairs [1;2] ;;      (* = [ [1;1]; [1;2]; [2;1]; [2;2] ] *)
allPairs [1;2;5] ;;  (* = [ [1;1]; [1;2]; [1;5]; [2;1]; [2;2]; [2;5]; [5;1]; [5;2]; [5;5] ] *)
```

Sample Solutions

```
let rec help1 x y = match y with
   [] -> []
  | h::t -> [x;h] :: (help1 x t) ;;
let rec help2 all z = match z with
   [] -> []
  | h::t -> (help1 h all) @ (help2 all t) ;;
let allPairs lst = help2 lst lst ;;
```

```
let allPairs lst =
        let rec help1 x y = match y with
            [] -> []
          | h::t -> [x;h] :: (help1 x t) in
        let rec help2 z = match z with
            [] -> []
          | h::t -> (help1 h lst) @ (help2 t) in
        help2 lst ;;
```

```
let help1 x y = fold (fun a h -> [x;h]::a) [] y ;;
let allPairs lst = fold (fun a h -> (help1 h lst) @ a) [] lst ;;
```

```
let help1 x y = map (fun h -> [x;h]) y ;;
let allPairs lst = fold (fun a h -> (help1 h lst) @ a) [] lst ;;
```

```
let allPairs lst =
   let help1 x = fold (fun a h -> [x;h]::a) [] lst in
        fold (fun a h -> (help1 h) @ a) [] lst ;;
```

```
let allPairs lst =
   let help1 x = map (fun h -> [x;h]) lst in
        fold (fun a h -> (help1 h) @ a) [] lst ;;
```

```
let allPairs lst = fold (fun a h -> (map (fun z -> [h;z]) lst) @ a) [] lst ;;
```

4. (8 pts) OCaml polymorphic types

Consider the OCaml type *bitwiseExp* implementing bitwise expressions (think of it as a binary tree where all leaves are 0's or 1's). For all answers, 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 +. You may use helper functions.

```
type bitwiseExp =
   Zero                                (* 0 *)
  | One                                (* 1 *)
  | Pair of bitwiseExp * bitwiseExp    (* b1 • b2 *)
```

bitwiseExp	onesExp
let x = Zero;;	0
let y = One;;	1
let p = Pair (x,y);;	1
let q = Pair (y,y);;	2

a. (2 pts) Write an OCaml expression with type bitwiseExp that is equivalent to the expression "0 • (1 • 0)"

Pair (Zero, Pair (One, Zero))

b. (6 pts) Write a function o*nesExp* of type (bitwiseExp -> int) that takes a bitwise expression and returns the number of 1's in the expression.

```
let rec onesExp b = match b with
        Zero -> 0
        | One -> 1
        | Pair (x,y) -> (onesExp x) + (onesExp y)
```

5. (10 pts) Context free grammars.

Consider the following grammar (S = start symbol and terminals = **a b % &**):

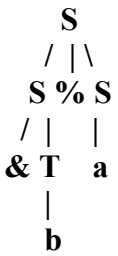
$S \rightarrow a \mid S\%S \mid \&T$

$T \rightarrow a \mid b$

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

i.	**%a**	Yes	**<u>No</u>**	(circle one)
ii.	**a%&b**	**<u>Yes</u>**	No	(circle one)
iii.	**&a&b**	Yes	**<u>No</u>**	(circle one)

b. (3 pts) Draw a parse tree for the string "&b%a"



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

Grammar is ambiguous. Multiple parse trees for strings such as a%a%a.

6. (14 pts) Using context free grammars.
   a. (6 pts) Given the following production for B, create a grammar that generates OCaml tuples of strings (for the 3 strings Joe, Laura, and Mike). Your grammar should be able to generate strings such as ("Joe"), ("Joe", "Mike"), ("Joe", "Laura", "Laura"), etc...

      B → *"Mike"* | *"Laura"* | *"Joe"*
      S → (A)
      A → B | B , A

   b. (8 pts) Consider the following grammar:

      S → c | d | e | S%S | S@S | &S

      Modify the grammar above to make the % operator left associative, the @ operator right associative, and make @ have the lowest precedence and & have the highest precedence.

      S → A@S | A
      A → A%B | B
      B → &B | C **(*B → &C | C is also acceptable *)**
      C → c | d | e

7. (15 pts) Parsing

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

$$S \rightarrow a \mid bB \mid Ac$$
$$A \rightarrow SAd \mid epsilon$$
$$B \rightarrow c \mid b$$

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

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

b. (7 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') || (lookahead  == 'b') || (lookahead  == 'c')) {
            parse_S( );
            parse_A( );
            match('d');
    }
    else
            ;

}
```

8. (9 pts) Multithreading

Consider the following multithreaded Java 1.4 code. Assume there are multiple producer and consumer threads being executed in the program, but only a single Buffer object. Questions about the "last statement executed" by two threads x & y refer to the most recently executed statement by those threads at some arbitrary time during the program execution. It does not mean the last statement executed by a thread before the thread exits. If a situation is possible, you need to give an example of how it is possible (e.g., thread x gets to statement a, then thread y gets to statement b). If a situation is not possible, you need to explain why.

```
class Buffer {
  void produce(o) {
    synchronized (this) {
1.    while (...) wait( );
2.    ...
3.    notifyAll( );
4.    ...
    }
  }
```

```
  Object consume( ) {
    synchronized (this) {
5.    while (...) wait( );
6.    ...
7.    notifyAll( );
8.    ...
    }
  }
}
```

a. (3 pts) Is it possible given two threads x and y for the last statement executed by thread x to be statement 5, and the last statement executed by thread y to be statement 7? Explain your answer.

Yes. Thread x reaches statement 5 and *releases the lock* by calling wait, thread y then acquires lock and reaches statement 7.

b. (3 pts) Is it possible given two threads x and y for the last statement executed by thread x to be statement 7, and the last statement executed by thread y to be statement 6? Explain your answer.

No. Whichever thread enters synchronized region first (and reaches statement 6 or 7) must still be holding the lock. The other thread *cannot acquire the lock* to enter the synchronized region.

c. (3 pts) Is it possible given two threads x and y for the last statement executed by thread x to be statement 6, and the last statement executed by thread y to be statement 1? Explain your answer.

Yes. Thread y reaches statement 1 and *releases the lock* by calling wait, thread x then acquires lock and reaches statement 6.