CMSC330 Spring 2012 Midterm #2 Solutions

1. (16 pts) OCaml Types and Type Inference

   Give the type of the following OCaml expressions:
   a. (2 pts) fun (x,y) -> [y+x] **Type = (int * int) -> int list**
   b. (3 pts) fun x y -> [y x] **Type = ‘a -> (‘a->‘b) -> ‘b list**

   Write an OCaml expression with the following type:
   c. (3 pts) (int -> int -> int) -> int **Code = fun x -> 1+(x 1 2)**
   d. (3 pts) ‘a -> (‘a list -> ‘b) -> ‘b **Code = fun x y -> y [x]**

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

   e. (2 pts) let x = 3 in let x = x+2 in x+4 **Value / Error = 9**
   f. (3 pts) (fun (h::t) -> t) [1;2] **Value / Error = [2]**

2. (20 pts) OCaml programming

Write a function *splitList* which given a number *n* and a list *lst*, returns a list of lists consisting of *lst* split into lists of length *n*. Concatenating the resulting list of lists should result in the original list. Only the last list is allowed to be shorter than length *n*. The function *splitList* ( ) has type int -> 'a list -> 'a list list.

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:

splitList 3 [] = []
splitList 3 [1;2] = [[1;2]]
splitList 3 [1;2;3;4;5;6;7;8] = [[1;2;3];[4;5;6];[7;8]]
splitList 2 ["b";"a";"e";"d";"c"] = [["b";"a"];["e";"d"];["c"]]

```
let rec splitHelp n x y =
    match y with
        [] -> (x,y)
        | h::t -> if (n = 0) then (x,y) else
            splitHelp (n-1) (h::x) t
;;
let rec splitList n lst =
    let (x,y) = (splitHelp n [] lst) in
    match y with
        [] -> [List.rev x]
        | _ -> (List.rev x)::(splitList n y)
;;

let splitList n lst = match (fold_left (fun (i,a) h -> match a with
    [] -> (i+1,[[h]])
    | (x::t) -> if i<n then (i+1, (h::x)::t)
        else (1, [h]::x::t)) (0,[]) lst) with
    (_,a) -> (List.rev (map List.rev a))
;;
```

3. (12 pts) OCaml higher-order & anonymous functions

Using fold and an anonymous function, write a function *getSeconds* which given a list of pairs returns a list of the 2nd value in each pair, in the original order. getSeconds( ) has type (‘a * ‘b) list -> ‘b list.

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). Solutions using recursion and/or helper functions will only receive partial credit.

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

Examples:

getSeconds [] = []
getSeconds [(1,2)] = [2]
getSeconds [(“foo”,”bar”)] = [“bar”]
getSeconds [(“f”,2);(“b”,3)] = [2;3]

let getSeconds l = List.rev (fold (fun a (x,y) -> y::a) [] l)

4. (14 pts) OCaml polymorphic types

Consider the OCaml type *arith* implementing arithmetic expressions:

```
type arith =
      Number of int
    | Sum of arith * arith
    | Prod of arith * arith
```

a. (4 pts) Write an OCaml expression with type arith that is equivalent to 3+(4*5)

(Sum ((Number 3), Prod (Number 4, Number 5))

b. (10 pts) Write a function *evalExpr* of type (arith -> int) that takes an 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 *. You may use helper functions.

Examples:

```
let x = (Number 1);;                  (* (evalExpr x) returns 1 *)
let y = (Number 2);;                  (* (evalExpr y) returns 2 *)
let z = (Sum (x,y));;                 (* (evalExpr z) returns 3 *)
evalExpr (Prod (y,y));;               (* returns 4 *)
evalExpr (Sum (y,(Sum (y,x))));;      (* returns 5 *)
```

```
let rec evalExpr e = match e with
      Number n -> n
      | Sum (e1,e2) -> ((evalExpr e1)+(evalExpr e2))
      | Prod (e1,e2) -> ((evalExpr e1)*(evalExpr e2))
```

5. (20 pts) Context free grammars.

Consider the following grammar (S = start symbol and terminals = **0 1 & @**):

S → S&S | S@S | E
E → 0 | 1

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

i. 0@1	**Yes**	No	(circle one)
ii. 0&1&1	**Yes**	No	(circle one)
iii. 0@@1	Yes	**No**	(circle one)

b. (4 pts) Does the following prove the grammar is ambiguous? Briefly explain.

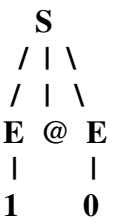
Two derivations of the same string "1@1@0"

S ⇒S@S ⇒E@S ⇒1@S ⇒1@S@S ⇒1@E@S ⇒1@1@S ⇒1@1@E ⇒1@1@0

S ⇒S@S ⇒S@S@S⇒ E@S@S ⇒1@S@S ⇒1@E@S ⇒1@1@S ⇒1@1@E ⇒1@1@0

Yes, since both derivations are left-most derivations.

c. (3 pts) Draw a parse tree for the string "1@0"



d. (10 pts) Applying context free grammars.

Consider the following grammar (S = start symbol and terminals = **0 1 &** @):

S → S&S | S@S | E
E → 0 | 1

Modify the grammar above to make the & operator *left associative*, the @ operator right associative, and make @ have higher precedence than &.

S → S&A | A
A → B@A | B
B → 0 | 1

6. (18 pts) Parsing

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

S → ASa | cb
A → aAc | Bda
B → bBa | ε (* epsilon *)

a. (10 pts) Compute First sets for S, A, and B

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

b. (8 pts) Using pseudocode, write *only* the parse_S function found in a recursive descent parser for the grammar. You may assume the functions parse_A , 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_S( ) {     // your code starts here

    if ((lookahead == "a") || (lookahead == "b") || (lookahead == "d")) { // S → ASa
            parse_A( ); parse_S( ); match("a");
    } else if ((lookahead == "c") {                                       // S → cb
            match("c" ); match("b");
    } else error();                                                       // error
```