CMSC330 Fall 2010 Midterm #2 Solutions

1. (22 pts) OCaml Types and Type Inference

   Give the type of the following OCaml expressions

   a. (2 pts) fun x y -> [x + y]   **Type = int -> int -> int list**
   b. (3 pts) fun x y -> [x ; y]   **Type = ‘a -> ‘a –> ‘a list**
   c. (3 pts) fun x y -> [x y]   **Type = (‘a -> ‘b) -> ‘a -> ‘b list**

   Write an OCaml expression with the following type

   d. (2 pts) bool -> int   **Code = fun x -> if x then 1 else 2**
   e. (3 pts) bool -> int -> int   **Code = fun x y -> if x then y+1 else 2**
   f. (3 pts) (bool -> int) -> int   **Code = fun x -> (x true)+1**

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

   g. (2 pts) let x = 2 in let x = 4 in x+8   **Value / Error = 12**
   h. (2 pts) let x = 2 in let y = x+4 in x+y   **Value / Error = 8**
   i. (2 pts) let x = 2 in let x = x+4 in x+8   **Value / Error = 14**

2. (12 pts) OCaml programming

   a. (8 pts) Implement a function *generateFinder* which when passed a list of strings *wanted* returns a function that takes a string *name* as argument, and returns true if *name* is in *wanted*.

   You are not allowed to use any OCaml library functions.

   Example:
   ```
   let findJedi = generateFinder [“Yoda”; “Luke”; “Obi-Wan”] ;;
   let findSith = generateFinder [“Palpatine”; “Vader”] ;;
   findJedi “Luke”;;            (* returns true *)
   findJedi “Vader”;;           (* returns false *)
   findSith “Vader”;;           (* returns true *)
   ```

   let rec generateFinder lst name = match lst with
   [] -> false
   | (h::t) -> if (h=name) then true else (generateFinder t name)

   b. (4 pts) What feature of closures allows a simple solution to the problem above? Explain.

The closure’s environment stores the value of variables, in this case the list of strings to be searched, so that it may be referenced even after generateFinder has exited (with the closure as its return value).
In comparison, currying is an OCaml feature that makes it simple to write functions that return functions. Currying relies on closures, but is not a feature of closures.

(12 pts) OCaml higher-order & anonymous functions

Recall that association lists are lists of tuples, with the 1st element of each tuple used as a key and the 2nd element of each tuple used as its associated value.

Using fold and an anonymous function, implement a function *countWords* which when passed a **sorted** list of words returns a **sorted** association list with the number of times each words occurs. The type *countWords* is string list -> (string * int) list.

Note you do not need to sort the list of words or associations. Simply build the association list based on the order of the words in the original list.

Your code must work in linear time (i.e., avoid using list append). You are not allowed to use any OCaml library functions except for *List.rev*, which reverses a list in linear time.

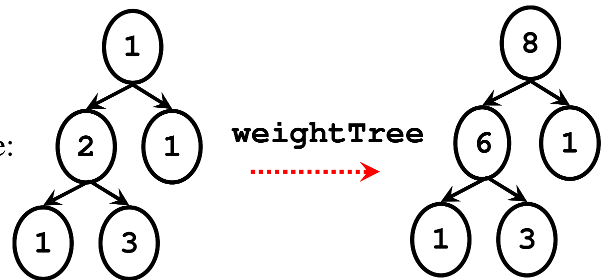
Partial credit given for solutions which do not use fold and/or an anonymous function.

```
Example:    countWords [ ] ;;                  (* returns [ ] *)
            countWords [“a”;“b”] ;;            (* returns [(“a”,1);(“b”,1)] *)
            countWords [“a”;“b”;“b”;“c”] ;;    (* returns [(“a”,1);(“b”,2);(“c”,1)] *)
```

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

```
let countWords lst = List.rev (fold
  (fun x h -> match x with
     [] -> [(h,1)]
     | (a,b)::c -> if (h=a) then ((a,b+1)::c) else ((h,1)::(a,b)::c))
  [] lst)
```

3. (14 pts) OCaml polymorphic types



Consider the OCaml type *tree* implementing a binary tree:

```
type tree =
  Empty
 | Node of int * tree * tree
```

We define the *weight* of a node to the sum of the values of the node and all the nodes of its children. Write a function *weightTree* of type (tree -> tree) that takes a tree and returns a new tree of the same shape but where the value at each node is replaced with its weight.

Your code must work in linear time (i.e., avoid recalculating the weight of the same part of the tree). You are not allowed to use any OCaml library functions. You may use helper functions.

Examples:

```
weightTree (Empty);;                        (* returns Empty *)
weightTree (Node (5, Empty, Empty));;       (* returns Node (5, Empty, Empty)  *)
weightTree (Node (2, Node (1, Empty, Empty), Node (3, Empty, Empty))));;
        (* returns Node (6, Node (1, Empty, Empty), Node (3, Empty, Empty)) *)
```

```
let rootVal t = match t with
  Empty -> 0
 | Node (w, _, _) -> w

let rec weightTree t = match t with
  Empty -> Empty
 | Node (w, a, b) ->
    let x = weightTree a in
    let y = weightTree b in
    Node (w + (rootVal x) + (rootVal y), x, y)
```

4. (22 pts) Context free grammars
   a. (8 pts) Let *nested int lists* be int lists with arbitrary number of levels of nesting. For instance, types for nested int lists include int list, int list list, int list list list, etc.

      Give a grammar for the type of OCaml functions that have a single nested int list argument, and returns a single nested int list, given the three terminals **int, list, ->.** I.e., your grammar should be able to generate int list -> int list, int list -> int list list, int list list list list -> int list list, etc…

      Make your grammar unambiguous and left recursive.

Some possible solutions			
S → L -> L **L → int M** **M → M list | list**	**S → L -> L** **L → int list M** **M → M list | ε**	**S → int L -> int L** **L → L list | list**	**S → int list L -> int list L** **L → L list | ε**

Consider the following grammar: S → S x S | S y S | 0 | 1

   b. (2 pts) Provide a derivation for the string "1y0".

      S => SyS => 1yS => 1y0

   c. (4 pts) Prove that the grammar is ambiguous

      Provide either multiple left-most derivations or parse trees for the same string.
      E.g., for 0x0x0
      S => SxS => SxSxS => 0xSxS => 0x0xS => 0x0x0
      S => SxS => 0xS => 0xSxS => 0x0xS => 0x0x0

   d. (4 pts) What could occur when parsing ambiguous grammars, and why is it a problem?

      Generating multiple parse trees for the same input means the same input could produce different results.

   e. (4 pts) Explain why the example grammar cannot be parsed using a predictive recursive descent parser.

      Two possible reasons. First, the parser cannot choose between the productions SxS and SyS using a lookahead token. Second, recursive descent parsers cannot handle left recursive productions such as S → SxS or S → SyS.

5. (18 pts) Parsing
   Consider the following grammar

   $S \rightarrow A+S \mid aAc$
   $A \rightarrow bA \mid \varepsilon$ (* epsilon *)

   a. (6 pts) Compute First sets for S and A

      First(S) = { a, b, + }
      First(A) = { b, ε }

   b. (12 pts) Using pseudocode, write a recursive descent parser for the grammar.

      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( ) {
        if (lookahead == “b”) || (lookahead == “+”) {       // S → A+S
                parse_A( ); match( “+” ); parse_S( );
        } else if (lookahead == “a”) {                      // S → aAc
                match( “a” ); parse_A( ); match( “c” );
        } else {
                error( );
        }
}

parse_A( ) {
        if (lookahead == “b”) {                             // A → bA
                match( “b” ); parse_A( );
        } else {                                            // A → ε (* epsilon *)
                ;
        }
}
```