CMSC330 Fall 2011 Midterm #2 Solutions

1. (14 pts) OCaml Types and Type Inference

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

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

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

   e. (2 pts) let x = 3 in let x = 5 in x+1 **Value** / Error **= 6**
   f. (2 pts) let x = 4 in let x = x+6 in x+3 **Value** / Error **= 13**

2. (15 pts) OCaml higher-order & anonymous functions

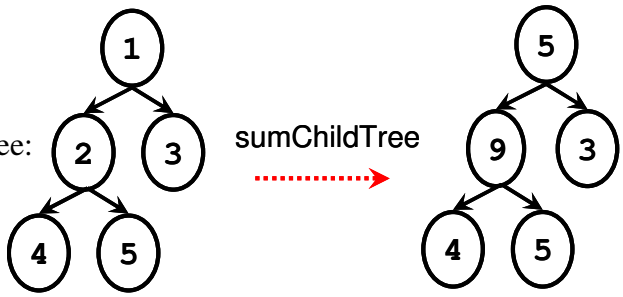
   Using fold and an anonymous function, write a function *pairUp* which given an int list returns a int list list, where every pair of elements in the original list is now paired up in a list. The strings should be in the same order as the in the original list. You may assume the original list has an even number of elements (including 0). 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. 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:
   pairUp [] = []
   pairUp [1;2] = [[1;2]]
   pairUp [1;2;3;4] = [[1;2];[3;4]]
   pairUp [1;2;3;4;5;6] = [[1;2];[3;4];[5;6]]

```
let pairUp l = List.rev (fold (fun a h -> match a with
    [] -> [[h]]                  // empty a = 1st elem
    | [x]::t -> [x;h]::t         // a begins w/ 1 elem list, add to list
    | [x;y]::t -> [h]::a         // a begins w/ 2 elem list, make new elem
    ) [] l) ;;                   // initial a = []
```

3. (10 pts) OCaml polymorphic types



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

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

Write a function *sumChildTree* 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 by the weights of its children in the original tree.

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. You may use helper functions.

Examples:

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

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

let rec sumChildTree t = match t with
      Empty -> Empty
      | Node (w, Empty, Empty) -> t
      | Node (w, a, b) ->
            Node (((nodeVal a) + (nodeVal b)),
                  (sumChildTree a), (sumChildTree b))
```

4. (15 pts) Context free grammars.

Consider the following grammar (where S = start symbol and terminals = * **0** **(** **)**):

S → * S S | A

A → 0 | (S) | epsilon

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

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

b. (2 pts) Does the following prove the grammar is ambiguous? Briefly explain.

Two derivations of the same string "*00"

S ⇒ * S S ⇒ * A S ⇒ * 0 S ⇒ * 0 A ⇒ * 0 0

S ⇒ * S S ⇒ * S A ⇒ * S 0 ⇒ * A 0 ⇒ * 0 0

No, proof requires multiple parse trees, multiple left-most derivations, or multiple right-most derivations.

c. (10 pts) Give a context free grammar for boolean expressions involving the values *t* and *f*, parentheses, and the binary operators & and |. Make both operators *left associative*, and make & have higher precedence than |. For instance, your grammar should be able to generate the boolean expressions "t", "t|f", "t|(t&t|f)|t".

S → S | A | A

A → A & B | B

B → t | f | (S)

5. (16 pts) Parsing

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

S → a | b+c | [A] | +B
A → SA | ε (* epsilon *)
B → Ac

a. (8 pts) Compute First sets for S, A, and B

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

b. (8 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 == "[") || (lookahead == "+")) {     // A → SA
                parse_S( ); parse_A );
        } else ;                                            // A → epsilon
    }
```

6. (10 pts) Lambda calculus

(2 pts each) Evaluate the following λ-expressions as much as possible.

a. (λx.λy.x y) a b c → **(λy.a y) b c → a b c**

b. (λx.λy.x y z) (λz.a z) b → **(λy.(λz.a z) y z) b → (λz.a z) b z → a b z**

c. (λx.λy.x y z) (λm.λn.n m o) → **λy.(λm.λn.n m o) y z → λy.(λn.n y o) z**
→ λy.z y o

(2 pts each) Determine whether alpha-renaming is necessary for each of the following lambda expressions. If it is, list the variable that must be renamed.

d. (λy.λz.z a y) z x No **Yes = z** (circle No, or give renamed var)

e. (λy.λz.y z a) a x **No** Yes = (circle No, or give renamed var)

7. (8 pts) Lambda calculus encodings

Consider the standard encodings for the booleans *if*, *true*, *false,* and *and*. Using these encodings, show that the following expression when simplified yields y:

if (and false true) then x else y

If you understand how they work, you do not need to expand true or false.

if a then b else c = a b c
true = λx.λy.x
false = λx.λy.y
and = λx.λy.(xy) false

if (and false true) then x else y → **(and false true) x y**
→ **(λx.λy.(xy) false) false true x y**
→ **(λy.(false y) false) true x y**
→ **(false true) false x y**
→ **false x y**
→ **y**

8. (12 pts) Operational semantics

a. (2 pts) In plain English, describe what the following means:

• , x :1 ; x + 2 → 3

In the environment created by binding x to 1, evaluating x+2 yields 3.

b. (10 pts) In an empty environment, what does the following expression

(fun x = (fun y = y x)) 2

evaluate to? In other words, find a v such that you can prove the following:

• ; (fun x = (fun y = y x)) 2 → v

Use the operational semantics rules given in class, included here for your reference. Show the complete proof that stacks uses of these rules.

• ; (fun x = (fun y = y x)) → (•, λx.(fun y = y x)) **// value of closure**
• ; 2 → 2 **// value of argument**
•, x:2 ; (fun y = y x) → (x :2, (λy.y x)) **// value of closure body**
// evaluated in new env

• ; (fun x = (fun y = y x)) 2 → (x:2, λy.y x))