

CMSC330 Spring 2010 Midterm #2 Solutions

1. (16 pts) OCaml Types and Type Inference

Give the type of the following OCaml expression

- a. (2 pts) `[[1 ; 2]]` **Type = `int list list`**
- b. (3 pts) `fun x -> 2::x` **Type = `(int list) -> (int list)`**

Write an OCaml expression with the following type

- c. (2 pts) `int list -> int` **Code = `fun (x::_) -> x+2`**
- d. (4 pts) `(int -> bool) -> int` **Code = `fun x -> if (x 1) then 2 else 3`**

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

- e. (2 pts) `if (1 < 2) then 3` **Value = error since 3 must be of type `()`**
- f. (3 pts) `let f x = f 2 in 1` **Value = error since f is undefined**

2. (14 pts) Higher order & anonymous functions

A *prefix sum* is an operation on lists in which the n^{th} element in the result list is obtained from the sum of the first n elements in the operand list. Using the following code for fold and an anonymous function, write a function `prefixSum` which given a list of ints, returns the prefix sum for the list.

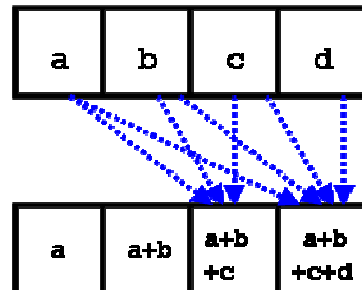
You are not allowed to use any helper functions or OCaml library functions, with the exception of `List.rev` (which reverses a list).

Partial credit given for solutions which do not use fold.

Example: `prefixSum [] = []`
`prefixSum [1;1;1;1;1] = [1;2;3;4;5]`
`prefixSum [1;2;3;4] = [1;3;6;10]`

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

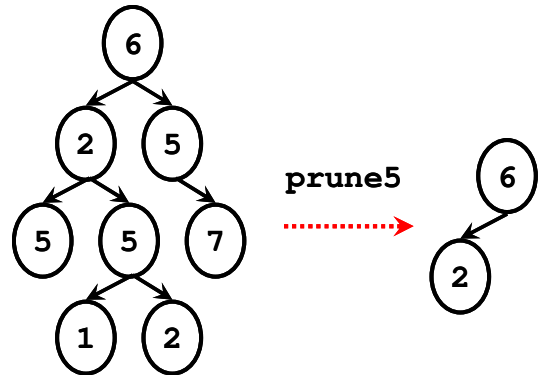
```
let prefixSum lst = List.rev (fold
  (fun a y -> match a with
    [] -> [y]
    | (h::_) -> ((h+y)::a))
  [] lst) ;;
```



3. (16 pts) OCaml polymorphic datatypes

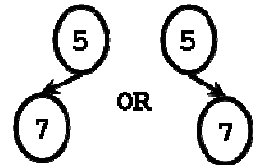
Consider the OCaml type *tree* implementing a binary tree of ints:

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



- a. (4 pts) Write an OCaml expression creating the data structure for a binary tree where the root node has value 5 and has one child node with value 7.

Node (5, Empty, Node (7, Empty, Empty)) **OR**
Node (5, Node (7, Empty, Empty), Empty)



- b. (5 pts) Implement a function *count5* that takes a tree and returns the number of nodes with the value 5. You may use helper functions (though they are not needed).

```
let rec count5 = function
  Empty -> 0                                // 0 if empty
  | Node (n, lt, rt) ->                     // examine node
    (if (n=5) then 1 else 0) + (count5 lt) + (count5 rt) // recurse on subtrees
```

- c. (7 pts) Implement a function *prune5* that takes a tree and returns a tree where all nodes with the value 5 (and their subtrees) are removed. You may use helper functions (though they are not needed).

```
let rec prune5 = function
  Empty -> Empty                            // no change if empty
  | Node (n, lt, rt) ->                     // examine node
    if (n=5) then Empty                    // if 5 then prune
    else (Node (n, prune5 lt, prune5 rt)) // recurse on subtrees
```

4. (16 pts) Context free grammars

Consider the following grammar: $S \rightarrow S \times T \mid T$ $T \rightarrow a \mid b$

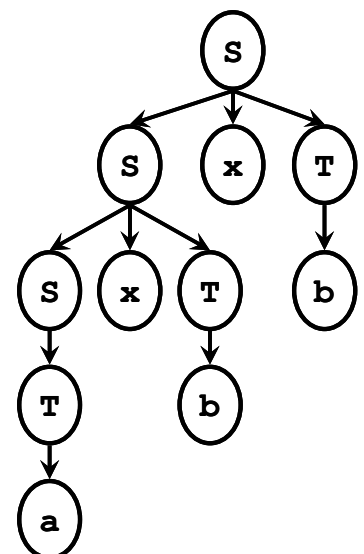
- a. (3 pts) Describe the set of strings generated by the grammar
 $((alb)x)^*(alb)$ **OR** $(alb)(x(alb))^*$

- b. (3 pts) Provide a left-most derivation for the string “axbxb”.

$S \Rightarrow SxT \Rightarrow SxTxT \Rightarrow TxTxT \Rightarrow axTxT \Rightarrow axbxT \Rightarrow axbxb$

- c. (2 pts) Provide a parse tree for the string “axbxb”.

See right



- d. (2 pts) What is the associativity of the x operator for the grammar?

Left associative

- e. (6 pts) Apply the algorithm discussed in class to transform the grammar so that it can be parsed using a recursive descent parser.

$S \rightarrow T L$

$L \rightarrow x T L \mid \epsilon$

$T \rightarrow a \mid b$

5. (22 pts) Parsing

Consider the following grammar

$S \rightarrow Abc \mid dS \mid \epsilon$ (* epsilon *)

$A \rightarrow aSA \mid f$

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

First(S) = { a, d, f, ϵ }

First(A) = { a, f }

- b. (14 pts) Using pseudocode, write a recursive descent parser for the grammar. Use the following utilities:

lookahead	Variable holding next terminal Lookahead == "\$" when at end of input
match (x)	Function to match next terminal to x
error ()	Reports parse error for input

```

parse_S() {
    if (lookahead == "a") || (lookahead == "f") {           // S → Abc
        parse_A(); match("b"); match("c");
    }
    else if (lookahead == "d") {                             // S → dS
        match("d"); parse_S();
    }
    else                                                       // S → ε
        ;
}

```

```

parse_A() {
    if (lookahead == "a") {                                   // A → aSA
        match("a"); parse_S(); parse_A();
    }
    else if (lookahead == "f") {                             // A → f
        match("f");
    }
    else
        error();
}

```

6. (16 pts) Operational semantics

- a. (4 pts) Consider the following operational semantics judgement. State in English what this statement is expressing:

$$\bullet, x:1 ; (+ x 2) \rightarrow 3$$

The expression $(+ x 2)$ evaluates to the value 3 in the environment resulting from the empty environment adding the binding $x=1$.

- b. (12 pts) In an empty environment, to what value v will the expression

$$(\text{fun } z = z) (+ 1 2)$$

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

$$\bullet ; (\text{fun } z = z) (+ 1 2) \rightarrow v$$

Use the operational semantics rules given in class. Show the complete proof that stacks uses of these rules.

$$\frac{\bullet ; (\text{fun } z = z) \rightarrow (\bullet, \lambda z. z) \quad \frac{\bullet ; 1 \rightarrow 1 \quad \bullet ; 2 \rightarrow 2}{\bullet ; (+ 1 2) \rightarrow 3} \quad (z:3 ; z) \rightarrow 3}{\bullet ; (\text{fun } z = z) (+ 1 2) \rightarrow 3}$$