

CMSC 330, Fall 2009, Practice Problems 4

1. OCaml and Functional Programming
 - a. Define functional programming
 - b. Define imperative programming
 - c. Define higher-order functions
 - d. Describe the relationship between type inference and static types
 - e. Describe the properties of OCaml lists
 - f. Describe the properties of OCaml tuples
 - g. Define pattern variables in OCaml
 - h. Describe the usage of “_” in OCaml
 - i. Describe polymorphism
 - j. Write a polymorphic OCaml function
 - k. Describe variable binding
 - l. Describe scope
 - m. Describe lexical scoping
 - n. Describe dynamic scoping
 - o. Describe environment
 - p. Describe closure
 - q. Describe currying
2. OCaml Types & Type Inference

Give the type of the following OCaml expressions:

 - a. []
 - b. 1::[]
 - c. 1::2::[]
 - d. [1;2;3]
 - e. [[1];[1]]
 - f. (1)
 - g. (1, "bar")
 - h. ([1,2], ["foo", "bar"])
 - i. [(1,2, "foo"); (3,4, "bar")]
 - j. let f x = 1
 - k. let f (x) = x *. 3.14
 - l. let f (x,y) = x
 - m. let f (x,y) = x+y
 - n. let f (x,y) = (x,y)
 - o. let f (x,y) = [x,y]
 - p. let f x y = 1
 - q. let f x y = x*y
 - r. let f x y = x::y
 - s. let f x = match x with [] -> 1
 - t. let f x = match x with (y,z) -> y+z
 - u. let f (x::_) = x
 - v. let f (_::y) = y
 - w. let f (x::y::_) = x+y

x. `let f = fun x -> x + 1`
 y. `let rec x = fun y -> x y`
 z. `let rec f x = if (x = 0) then 1 else 1+f (x-1)`
 aa. `let f x y z = x+y+z in f 1 2 3`
 bb. `let f x y z = x+y+z in f 1 2`
 cc. `let f x y z = x+y+z in f`
 dd. `let rec f x = match x with`
 `[] -> 0`
 `| (_::t) -> 1 + f t`
 ee. `let rec f x = match x with`
 `[] -> 0`
 `| (h::t) -> h + f t`
 ff. `let rec f = function`
 `[] -> 0`
 `| (h::t) -> h + (2*(f t))`
 gg. `let rec func (f, l1, l2) = match l1 with`
 `[] -> []`
 `| (h1::t1) -> match l2 with`
 `[] -> [f h1]`
 `| (h2::t2) -> [f h1; f h2]`

3. OCaml Types & Type Inference

Write an OCaml expression with the following types:

- a. `int list`
- b. `int * int`
- c. `int -> int`
- d. `int * int -> int`
- e. `int -> int -> int`
- f. `int -> int list -> int list`
- g. `int list list -> int list`
- h. `'a -> 'a`
- i. `'a * 'b -> 'a`
- j. `'a -> 'b -> 'a`
- k. `'a -> 'b -> 'b`
- l. `'a list * 'b list -> ('a * 'b) list`
- m. `int -> (int -> int)`
- n. `(int -> int) -> int`
- o. `(int -> int) -> (int -> int) -> int`
- p. `('a -> 'b) * ('c * 'c -> 'a) * 'c -> 'b`

4. OCaml Programs

What is the value of the following OCaml expressions? If an error exists, describe the error.

- a. `2 ; 3`
- b. `2 ; 3 + 4`
- c. `(2 ; 3) + 4`
- d. `if 1 < 2 then 3 else 4`
- e. `let x = 1 in 2`
- f. `let x = 1 in x+1`
- g. `let x = 1 in x ; x+1`
- h. `let x = (1, 2) in x ; x+1`
- i. `(let x = (1, 2) in x) ; x+1`
- j. `let x = 1 in let y = x in y`
- k. `let x = 1 let y = 2 in x+y`
- l. `let x = 1 in let x = x+1 in let x = x+1 in x`
- m. `let x = x in let x = x+1 in let x = x+1 in x`
- n. `let rec x y = x in 1`
- o. `let rec x y = y in 1`
- p. `let rec x y = y in x 1`
- q. `let x y = fun z -> z+1 in x`
- r. `let x y = fun z -> z+1 in x 1`
- s. `let x y = fun z -> z+1 in x 1 1`
- t. `let x y = fun z -> x+1 in x 1`
- u. `let rec x y = fun z -> x+1 in x 1`
- v. `let rec x y = fun z -> x+y in x 1`
- w. `let rec x y = fun z -> x y in x 1`
- x. `let rec x y = fun z -> x z in x 1`
- y. `let x y = y 1 in 1`
- z. `let x y = y 1 in x`
- aa. `let x y = y 1 in x 1`
- bb. `let x y = y 1 in x fun z -> z + 1`
- cc. `let x y = y 1 in x (fun z -> z + 1)`
- dd. `let a = 1 in let f x y z = x+y+z+a in f 1 2 3`
- ee. `let a = 1 in let f x y z = x+y+z+a in f 1 2 -3`

5. OCaml Programming

- a. Write an OCaml function named *fib* that takes an int *x*, and returns the Fibonacci number for *x*. Recall that $\text{fib}(0) = 0$, $\text{fib}(1) = 1$, $\text{fib}(2) = 1$, $\text{fib}(3) = 2$.
- b. Write a function *find_suffixes* which applied to a list *lst* returns a list of all the suffixes of *lst*. For instance, $\text{suffixes } [1;2;5] = [[1;2;5] ; [2;5] ; [5]]$
- c. Write an OCaml function named *map_odd* which takes a function *f* and a list *lst*, applies the function to every other element of the list, starting with the first element, and returns the result in a new list.
- d. Use *map_odd* and *fib* applied to the list $[1;2;3;4;5;6;7]$ to calculate the Fibonacci numbers for 1, 3, 5, and 7.
- e. Using *map*, write a function *triple* which applied to a list of ints *lst* returns a list with all elements of *lst* tripled in value.
- f. Using *fold*, write a function *all_true* which applied to a list of booleans *lst* returns true only if all elements of *lst* are true.
- g. Using *fold* and anonymous helper functions, write a function *product* which applied to a list of ints *lst* returns the product of all the elements in *lst*.
- h. Using *fold* and anonymous helper functions, write a function *find_min* which applied to a list of ints *lst* returns the smallest element in *lst*.
- i. Using the *fold* function and anonymous helper functions, write a function *count_vote* which applied to a list of booleans *lst* returns a tuple (x,y) where x is the number of true elements and y is the number of false elements.
- j. Using the function *count_vote*, write a function *majority* which applied to a list of booleans *lst* returns true if 1/2 or more elements of *lst* are true.

6. OCaml Polymorphic Types

Consider a OCaml module Bst that implements a binary search tree:

```
module Bst = struct
  type bst =
    | Empty
    | Node of int * bst * bst

  let empty = Empty          (* empty binary search tree      *)

  let is_empty = function    (* return true for empty bst  *)
    | Empty -> true
    | Node (_, _, _) -> false

  let rec insert n = function (* insert n into binary search tree *)
    | Empty -> Node (n, Empty, Empty)
    | Node (m, left, right) ->
      if m = n then Node (m, left, right)
      else if n < m then Node(m, (insert n left), right)
      else Node(m, left, (insert n right))

  (* Implement the following functions
     val min : bst -> int
     val remove : int -> bst -> bst
     val fold : ('a -> int -> 'a) -> 'a -> bst -> 'a
     val size : bst -> int
  *)

  let rec min =              (* return smallest value in bst *)
  let rec remove n t =       (* tree with n removed          *)
  let rec fold f a t =       (* apply f to nodes of t in inorder *)
  let size t =               (* # of non-empty nodes in t    *)

end
```

- Is insert tail recursive? Explain why or why not.
- Implement min as a tail-recursive function. Raise an exception for an empty bst. Any reasonable exception is fine.
- Implement remove. The result should still be a binary search tree.
- Implement fold as an inorder traversal of the tree so that the code
List.rev (fold (fun a m -> m::a) [] t)
will produce an (ordered) list of values in the binary search tree.
- Implement size using fold.

7. Recursive Descent Parser in OCaml

The example OCaml recursive descent parser 15-parseArith_fact.ml employs a number of shortcuts. For instance, the function parseS handles the grammar rules for

$$S \rightarrow T + S \mid T$$

directly instead of first applying left factoring:

$$S \rightarrow T A \quad A \rightarrow + S \mid \text{epsilon}$$

However, we can still identify where code corresponding to parseA was inserted directly in the code for parseS, in the comments below:

```
let rec parseS lr =                                (* parseS *)
  let x = parseT lr in                              (* S → T A *)
  match !lr with                                    (* parseA *)
  | ('+':t) ->                                     (* if lookahead = First( + S ) *)
    lr := t;                                       (* A → + S *)
    Sum (x,parseS lr)
  | _ -> x                                         (* A → epsilon *)
```

Similarly, the function parseF handles the grammar rules for

$$F \rightarrow U ! \mid U$$

directly instead of rewriting the grammar, creating the following productions:

$$F \rightarrow ? \quad B \rightarrow ?$$

You must identify where code corresponding to parseB was inserted directly in the code for parseF in the comments below:

```
let rec parseF lr =                                (* parseF *)
  let rec fHelper lr tmp =
    match !lr with
    | ('!':t) ->
      lr := t;
      Fact (fHelper lr tmp)
    | _ -> tmp
  in let x = parseU lr in (fHelper lr x)           (* parseB *)
                                                    (* 1: if lookahead = First( ? ) *)
                                                    (* 2: ? → ? *)
                                                    (* 3: ? → ? *)
                                                    (* 4: ? → ? *)
```

- What rule should have been applied to the productions for F?
- What productions for F & B would be created by applying the rule?
- What sentential form should appear in place of ? in comment 1?
- What production should appear in place of ? in comment 2?
- What production should appear in place of ? in comment 3?
- What production should appear in place of ? in comment 4?