

## CMSC 330, Spring 2009, Quiz 2 Practice Problem Solutions

1. OCaml and Functional Programming
  - a. Define functional programming  
**Programs are expression evaluations**
  - b. Define imperative programming  
**Programs change the value of variables**
  - c. Define iterative programming.  
**Programs that use loop constructs (e.g., while, for)**
  - d. Define higher-order functions  
**Functions can be passed as arguments and returned as results**
  - e. Describe the relationship between type inference and static types  
**Variable has a fixed type that can be inferred by looking at how variable is used in the code**
  - f. Describe the properties of OCaml lists  
**Entity containing 0 or more elements of the same type. Type of list is determined by type of element.**
  - g. Describe the properties of OCaml tuples  
**Entity containing 2 or more elements of possibly different types. Type of tuple is determined by type and number of elements.**
  - h. Define pattern variables in OCaml  
**Variables making up patterns used by “match”**
  - i. Describe the usage of “\_” in OCaml  
**Pattern variable that can match anything but does not add binding**
  - j. Describe polymorphism  
**Function that can take different types for same formal parameter**
  - k. Write a polymorphic OCaml function  
**let f x = x // ‘a -> ‘a, x can be of any type**
  - l. Describe variable binding  
**A variable (symbol) is associated with a value in an expression (or environment)**
  - m. Describe scope  
**Portion of program where variable binding is visible**
  - n. Describe lexical scoping  
**Variable binding determined by nearest scope in text of program**
  - o. Describe dynamic scoping  
**Variable binding determined by nearest runtime function invocation**
  - p. Describe environment  
**Collection of variable bindings**
  - q. Describe closure  
**Function code + environment pair, may be invoked as function**
  - r. Describe currying  
**Functions consume one argument at a time, returning closures until all arguments are consumed**

## 2. OCaml Types & Type Inference

Give the type of the following OCaml expressions:

- a. [] // 'a list
- b. 1::[] // int list
- c. 1::2::[] // int list
- d. [1;2;3] // int list
- e. [[1];[1]] // int list list
- f. (1) // int
- g. (1,"bar") // int \* string
- h. ([1,2], ["foo","bar"]) // (int \* int) list \* (string \* string) list
- i. [(1,2,"foo");(3,4,"bar")] // (int \* int \* string) list
- j. let f x = 1 // 'a -> int
- k. let f (x) = x \*. 3.14 // float -> float
- l. let f (x,y) = x // 'a \* 'b -> 'a
- m. let f (x,y) = x+y // int \* int -> int
- n. let f (x,y) = (x,y) // 'a \* 'b -> 'a \* 'b
- o. let f (x,y) = [x,y] // 'a \* 'b -> ('a \* 'b) list
- p. let f x y = 1 // 'a -> 'b -> int
- q. let f x y = x\*y // int -> int -> int
- r. let f x y = x::y // 'a -> 'a list -> 'a list
- s. let f x = match x with [] -> 1 // 'a list -> int
- t. let f x = match x with (y,z) -> y+z // int \* int -> int
- u. let f (x::\_) -> x // 'a list -> 'a
- v. let f (\_::y) = y // 'a list -> 'a list
- w. let f (x::y::\_) = x+y // int list -> int
- x. let f = fun x -> x + 1 // int -> int
- y. let rec x = fun y -> x y // 'a -> 'b
- z. let rec f x = if (x = 0) then 1 else 1+f (x-1) // int -> int
- aa. let f x y z = x+y+z in f 1 2 3 // int
- bb. let f x y z = x+y+z in f 1 2 // int -> int
- cc. let f x y z = x+y+z in f // int -> int -> int -> int
- dd. let rec f x = match x with  
[] -> 0  
| (\_::t) -> 1 + f t // 'a list -> int
- ee. let rec f x = match x with  
[] -> 0  
| (h::t) -> h + f t // int list -> int
- ff. let rec f = function  
[] -> 0  
| (h::t) -> h + (2\*(f t)) // int list -> int
- gg. let rec func (f, l1, l2) = match l1 with // ('a -> 'b) \* 'a list \* 'a list -> 'b list  
[] -> []  
| (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` // `[1]`
- b. `int * int` // `(1,1)`
- c. `int -> int` // `let f x = x+1`
- d. `int * int -> int` // `let f (x,y) = x+y`
- e. `int -> int -> int` // `let f x y = x+y`
- f. `int -> int list -> int list` // `let f x y = (x+1)::y`
- g. `int list list -> int list` // `let f (x::_) = 1::x`
- h. `'a -> 'a` // `let f x = x`
- i. `'a * 'b -> 'a` // `let f (x,y) = x`
- j. `'a -> 'b -> 'a` // `let f x y = x`
- k. `'a -> 'b -> 'b` // `let f x y = y`
- l. `'a list * 'b list -> ('a * 'b) list` // `let f (x::_,y::_) = [(x,y)]`
- m. `int -> (int -> int)` // `let f x y = x+y`
- n. `(int -> int) -> int` // `let f x = 1+(x 1)`
- o. `(int -> int) -> (int -> int) -> int` // `let f x y = 1+(x 1)+(y 1)`
- p. `('a -> 'b) * ('c * 'c -> 'a) * 'c -> 'b` // `let f (x, y, z) = (x (y (z,z)))`

### 4. OCaml Programs

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

- a. `2 ; 3` // `3`
- b. `2 ; 3 + 4` // `7`
- c. `(2 ; 3) + 4` // `7`
- d. `if 1<2 then 3 else 4` // `3`
- e. `let x = 1 in 2` // `2`
- f. `let x = 1 in x+1` // `2`
- g. `let x = 1 in x ; x+1` // `2`
- h. `let x = (1, 2) in x ; x+1`  
// **error: x has type int\*int but used with int**
- i. `(let x = (1, 2) in x) ; x+1` // **error: unbound value x**
- j. `let x = 1 in let y = x in y` // `1`
- k. `let x = 1 let y = 2 in x+y` // **syntax error: missing "in"**
- l. `let x = 1 in let x = x+1 in let x = x+1 in x` // `3`
- m. `let x = x in let x = x+1 in let x = x+1 in x` // **error: unbound value x**
- n. `let rec x y = x in 1` // **error: x has type 'a -> 'b but used with 'b**
- o. `let rec x y = y in 1` // `1`
- p. `let rec x y = y in x 1` // `1`
- q. `let x y = fun z -> z+1 in x` // `fun y -> (fun z -> z+1)`
- r. `let x y = fun z -> z+1 in x 1` // `fun z -> z+1`
- s. `let x y = fun z -> z+1 in x 1 1` // `2`
- t. `let x y = fun z -> x+1 in x 1` // **error: unbound value x**
- u. `let rec x y = fun z -> x+1 in x 1`  
// **error: x has type 'a -> 'b -> 'c but used with int**

- v. let rec x y = fun z -> x+y in x 1  
// error: x has type 'a -> 'b -> 'c but used with int
- w. let rec x y = fun z -> x y in x 1  
// error: x has type 'a -> 'b but used with 'b
- x. let rec x y = fun z -> x z in x 1  
// error: x has type 'a -> 'b but used with 'b
- y. let x y = y 1 in 1 // 1
- z. let x y = y 1 in x // fun y -> (y 1)
- aa. let x y = y 1 in x 1 // error: 1 has type int but used with int -> 'a
- bb. let x y = y 1 in x fun z -> z + 1 // syntax error at "x fun"
- cc. let x y = y 1 in x (fun z -> z + 1) // 2
- dd. let a = 1 in let f x y z = x+y+z+a in f 1 2 3 // 7
- ee. let a = 1 in let f x y z = x+y+z+a in f 1 2 -3  
// error: (f 1 2) has type int -> int but used with int

## 5. OCaml Programming

- ```

let rec map f l = match l with
  [] -> []
  | (h::t) -> (f h)::(map f t)
;;
let rec fold f a l = match l with
  [] -> a
  | (h::t) -> fold f (f a h) t
;;

```
- a. Write an OCaml function named *fib* that takes an int *x*, and returns the Fibonacci number for *x*. Recall that  $\text{fib}(0) = 0$ ,  $\text{fib}(1) = 1$ ,  $\text{fib}(2) = 1$ ,  $\text{fib}(3) = 2$ .
 

```

let rec fib x =
  if (x = 0) then 0
  else if (x = 1) then 1
  else (fib (x-1) + fib (x-2))
;;

```
  - b. Write a function *find\_suffixes* which applied to a list *lst* returns a list of all the suffixes of *lst*. For instance,  $\text{suffixes } [1;2;5] = [ [1;2;5] ; [2;5] ; [5] ]$ 

```

let rec suffix_helper (x, r) =
  match x with
  [] -> r
  | (h::t) -> (suffix_helper (t, (h::t)::r))
;;
let suffixes x = List.rev (suffix_helper (x, []))
;;

```

- 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.

```
let rec map_odd f l = match l with
    [] -> []
   | (x1::[]) -> [f x1]
   | (x1::x2::t) -> (f x1)::(map_odd f t)
;;
```

- 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.

```
map_odd fib [1;2;3;4;5;6;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.

```
let triple x = map (fun x -> 3*x) x ;;
```

- 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.

```
let all_true lst = fold (fun a x -> (x = true) && (a = true)) true lst ;;
```

- 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*.

```
let product x = fold (fun a y -> a*y) 1 x ;;
```

- 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*.

```
let find_min x = fold (fun a y -> min a y) max_int x ;;
```

- 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.

```
let count_vote x = fold (fun (y,n) v ->
    if (v) then (y+1,n) else (y,n+1)) (0,0) x
```

```
;;
```

- 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.

```
let majority x = match (count_vote x) with (y,n) -> (y >= n) ;;
```