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 -> int
m. let f (x,y) = x+y // 'a * 'b -> int

n. let f (x,y) = (x,y) // 'a * 'b

o. let f x y = 1 // int -> int

p. let f x y = x+y // int list

q. let f x y = x::y // 'a list

r. let f x y = x::y // '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 // 'a list -> int

x. let f = fun x -> x + 1 // int

y. let rec f 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 y z = x+y+z in f 1 2 3 // int

bb. let f y z = x+y+z in f 1 2 // int

cc. let f x y z = x+y+z in f // int -> int -> int

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 // (*a -> '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::_) = 1::x
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 -> z+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
   \[\text{// error: x has type `a -> `b -> `c but used with int}\]

w. let rec x y = fun z -> x y in x 1
   \[\text{// error: x has type `a -> `b but used with `b}\]

x. let rec x y = fun z -> x z in x 1
   \[\text{// error: x has type `a -> `b but used with `b}\]

y. let x y = y 1 in 1
   \[\text{// 1}\]

z. let x y = y 1 in x
   \[\text{// fun y -> (y 1)}\]

aa. let x y = y 1 in x
    \[\text{// error: 1 has type int but used with int -> `a}\]

bb. let x y = y 1 in x
    \[\text{// syntax error at “x fun”}\]

cc. let x y = y 1 in x
    \[\text{// 2}\]

dd. let a = 1 in let f x y z = x + y + z + a in f 1 2 3
    \[\text{// 7}\]

ee. let a = 1 in let f x y z = x + y + z + a
    \[\text{// error: (f 1 2) has type int -> int but used with int}\]

5. OCaml Programming
   let rec map f l = match l with
   \[\text{[| ] -> [| ]}\]
   \[\text{| (h::t) -> (f h)::(map f t)}\]
   \[
   \]

   let rec fold f a l = match l with
   \[\text{[| ] -> a}\]
   \[\text{| (h::t) -> fold f (f a h) t}\]
   \[
   \]

   a. Write an OCaml function named fib that takes an int x, and returns the
   Fibonacci number for x. Recall that fib(0) = 0, fib(1) = 1, fib(2) = 1, fib(3) = 2.
   let rec fib x =
   \[\text{if (x = 0) then 0}\]
   \[\text{else if (x = 1) then 1}\]
   \[\text{else (fib (x-1) + fib (x-2))}\]
   \[
   \]

   b. Write a function find_suffixes which applied to a list lst returns a list of all the
   suffixes of lst. For instance, suffixes [1;2;5] = [ [1;2;5] ; [2;5] ; [5] ]
   let rec suffix_helper (x, r) =
   match x with
   \[\text{[| ] -> r}\]
   \[\text{| (h::t) -> (suffix_helper (t, (h::t)::r))}\]
   \[
   \]
   \[\text{let suffixes x = List.rev (suffix_helper (x, [|]))}\]
   \[
   \]
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.

```ocaml
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.

```ocaml
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.

```ocaml
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.

```ocaml
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`.

```ocaml
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`.

```ocaml
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.

```ocaml
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.

```ocaml
let majority x = match (count_vote x) with (y,n) -> (y >= n)
```
6. OCaml Polymorphic Types
Consider a OCaml module Bst that implements a binary search tree:

```ocaml
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 = function
    (* return smallest value in bst  *)
    Empty -> (raise (Failure "min"))
  | Node (m, left, right) ->
    if (is_empty left) then m
    else min left

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

  a. Is insert tail recursive? Explain why or why not.
     No, since the return value for recursive call to insert cannot be used as
     the return value of the original call to insert. The return value is used
     to create a Node data type first, and the Node value is returned.

  b. Implement min as a tail-recursive function. Raise an exception for an empty bst.
     Any reasonable exception is fine.
     let rec min = function
     Empty -> (raise (Failure "min"))
     | Node (m, left, right) ->
       if (is_empty left) then m
       else min left
```
c. Implement remove. The result should still be a binary search tree.

```ocaml
let rec remove n = function
  | Empty -> Empty
  | Node (m, left, right) ->
    if m = n then (
      if is_empty left then right
      else if is_empty right then left
      else let x = min right in
        Node(x, left, remove x right)
        // OR
        // else let x = max left in
        // Node(x, remove x left, right)
    )
    else if n < m then Node(m, (remove n left), right)
    else Node(m, left, (remove n right))
```

d. Implement fold as an inorder traversal of the tree so that the code

```ocaml
List.rev (fold (fun a m -> m::a) [] t)
```

will produce an (ordered) list of values in the binary search tree.

```ocaml
let rec fold f a n = match n with
  | Empty -> a
  | Node (m, left, right) -> fold f (fold f a left) m right
```

e. Implement size using fold.

```ocaml
let size t = fold (fun a m -> a+1) 0 t
```
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:

\[ S \rightarrow T A \]
\[ A \rightarrow + S \mid \epsilon \]

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

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

\[ F \rightarrow ? \mid U \]
\[ B \rightarrow ! B \mid \epsilon \]

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

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

a. What rule should have been applied to the productions for F?

Eliminate left recursion
(e.g., change \( A \rightarrow A B \mid C \) to \( A \rightarrow C N \)
\( N \rightarrow B N \mid \epsilon \))

b. What productions for F & B would be created by applying the rule?

\[ F \rightarrow U B \]
\[ B \rightarrow ! B \mid \epsilon \]

c. What sentential form should appear in place of ? in comment 1?

\( ! B \)

d. What production should appear in place of ? in comment 2?

\( B \rightarrow ! B \)

e. What production should appear in place of ? in comment 3?

\( B \rightarrow \epsilon \)

f. What production should appear in place of ? in comment 4?

\( F \rightarrow U B \)