1. (5 pts) General. For the following multiple choice questions, circle the letter(s) on the right corresponding to the best answer(s) to each question.

   a. In the expression “let \( x_1 = E_1 \) in \( E_2 \)”, where is \( x \) bound to \( x_1 \)?
      A) only \( E_1 \)  B) only \( E_2 \)  C) both \( E_1 \) and \( E_2 \)  D) neither \( E_1 \) nor \( E_2 \)

   b. In the expression “let rec \( x_1 = E_1 \) in \( E_2 \)”, where is \( x \) bound to \( x_1 \)?
      A) only \( E_1 \)  B) only \( E_2 \)  C) both \( E_1 \) and \( E_2 \)  D) neither \( E_1 \) nor \( E_2 \)

   c. In the expression “(fun \( x_1 \rightarrow E_1 \), \( E_2 \)”, where is \( x \) bound to \( x_1 \)?
      A) only \( E_1 \)  B) only \( E_2 \)  C) both \( E_1 \) and \( E_2 \)  D) neither \( E_1 \) nor \( E_2 \)

   d. OCaml modules are used to group together data, types, and functions. They are similar to which of the following?
      A) Java classes  B) Java interfaces  C) .h files in C  D) .c files in C

   e. Context free grammars do not consist of which of the following?
      A) Productions  B) Terminals  C) Nonterminals  D) Final symbol

2. (12 pts) OCaml Types and Type Inference

   a. (4 pts) Give the type of the following OCaml expressions

      i. (2 pts) let \( a \ b = b \) in \( a \ 2 \)  Type = \text{int}

      ii. (2 pts) fun \( r \ u \ n \rightarrow (r \ u, \ n) \)  Type = \( 'a \rightarrow 'b \rightarrow 'a \rightarrow 'c \rightarrow ('b \ast 'c) \)

   b. (4 pts) Write an OCaml expression with the following type

      i. (2 pts) int \rightarrow (int \rightarrow 'a \rightarrow \text{int}) \ \ Code = \text{fun } x \ y \ z \rightarrow x+y

      ii. (2 pts) ('a list \rightarrow 'b) \rightarrow 'a \rightarrow 'b \ \ Code = \text{fun } x \ y \rightarrow x \ [y]

   c. (4 pts) Give the value of the following OCaml expressions. If an error exists, describe the error.

      i. (2 pts) (fun \( r \ u \ n \rightarrow u::[r; n \]) \ 1 \ 2 \ 3 \ \ Value / Error = [2;1;3]

      ii. (2 pts) let \( x = (\text{fun } x \rightarrow 3 - x) \) in \( x \ 2 \) \ Value / Error = 1
3. (12 pts) OCaml higher-order & anonymous functions

Using either map or fold and an anonymous function, write a curried function \texttt{count} which when given an element \texttt{x} of type `a and an associative \texttt{lst} of type \texttt{('a * int) list}, returns a new associative list where the int value associated with \texttt{x} is incremented by 1. The relative order of pairs in the new list must remain the same. You may assume \texttt{lst} includes a pair for \texttt{x}.

Your function must run in linear time. You may not use any library functions, with the exception of functions in Pervasives and the List.rev function, which reverses a list in linear time. You may not use imperative OCaml (i.e., no ref variables).

Example:
\begin{verbatim}
let count x lst =
  map (fun (v,num) -> if (x=v) then (v,num+1) else (v,num)) lst

let count x lst = List.rev (fold (fun a (v,num) ->
  if (v=x) then ((v,num+1)::a) else ((v,num)::a)) [ ] lst)
\end{verbatim}

4. (10 pts) OCaml polymorphic types

Consider the following OCaml user-variant type \texttt{oList} implementing an ordered linked list of ints. Assume ints are stored in \textit{descending} order (i.e., largest value at head of list).

\begin{verbatim}
type oList =
  Empty
| Node of int * oList
\end{verbatim}

Write a function \texttt{insert} of type (int -> oList -> oList) that takes an integer \texttt{n} and an oList \texttt{p}, and returns the the oList resulting from inserting \texttt{n} in \texttt{p}.

Your function must run in linear time. You may not use any library functions, with the exception of functions in Pervasives and the List.rev function, which reverses a list in linear time. You may not use imperative OCaml (i.e., no ref variables).

The following examples are given using int lists. \textbf{Your code must produce the equivalent output for oList. Functions that work only on int list will receive zero partial credit.}

Example:
\begin{verbatim}
let rec insert x lst = match lst with
  | Empty -> Node (x, Empty)
  | Node (y, z) -> if (x > y) then (Node (x, lst))
    else (Node (y, (insert x z)))
\end{verbatim}

\begin{verbatim}
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
\end{verbatim}
5. (20 pts) OCaml programming

Write a function `pack` with type (`'a list -> 'a list list`) which given a list `lst`, returns a new list composed of the elements of `lst` in lists, where consecutive identical elements from `lst` are placed in the same list. The relative order of elements in `lst` must be preserved in the new list.

Your function must run in linear time. You may not use any library functions, with the exception of functions in Pervasives and the List.rev function, which reverses a list in linear time. You may not use imperative OCaml (i.e., no ref variables). You may use map & fold.

Examples:

```
pack [] → []
pack [1;1;2;2;2] → [[1;1], [2;2;2]]
pack [1] → [[1]]
pack [1;1;2;2;1;3;3;3] → [[1;1], [2;2;1], [3;3;3]]
```

```
let rec pack lst =
    let rec helper x lst = match (x, lst) with
    | (_, []) -> (x, lst)
    | ([], h::t) -> helper [h] t2
    | (h::t1, h2::t2) -> if h1 = h2 then helper (h2::x) t2 else (x, lst)
    in match lst with
    | [] -> []
    | h::t -> let (x, y) = (helper [h] t) in x :: (pack y)

let pack lst =
    let rec helper current acc lst = match lst with
    | [] -> []
    | [x] -> (x :: current) :: acc
    | a :: b :: c ->
        if a = b then helper (a :: current) acc (b::c)
        else helper [] ((a :: current) :: acc) (b::c)
    in List.rev (helper [] [] lst)

let pack lst = List.rev (fold (fun a h -> match a with
    | [] -> [h])
    | (x::y)::t -> if x=h then ((h::x::y)::t) else ([h::a] :: [] lst)
```
6. (8 pts) Context free grammars.

Consider the following grammar for OCaml types involving int, int lists, and functions:

\[ T \rightarrow \text{int} | T \rightarrow T | T \text{ list} | (T) \]

a. (1 pt each) Indicate whether the following strings are generated by this grammar
   i. \text{int list -> int} \quad \text{Yes} \quad \text{No} \quad \text{(circle one)}
   ii. \text{int list int} \quad \text{Yes} \quad \text{No} \quad \text{(circle one)}
   iii. \text{(int -> int) list list} \quad \text{Yes} \quad \text{No} \quad \text{(circle one)}

b. (2 pts) Draw a parse tree for the string “int -> int list”

\[
\begin{array}{c}
\text{T} \\
\text{T} \rightarrow \text{T} \\
\text{int} \quad \text{T list} \\
\text{int} \quad \text{int}
\end{array}
\]

c. (3 pts) Is the grammar is ambiguous? Provide proof if you believe it is ambiguous.

Yes, multiple parse trees exist for the same string (alternatively, can show multiple left-most derivations for the same string).

Examples of strings with multiple parse trees include trees for “int -> int list” above, or trees for “int -> int -> int” below:

\[
\begin{array}{c}
\text{T} \\
\text{T} \rightarrow \text{T} \\
\text{int} \quad \text{T} \rightarrow \text{T} \\
\text{int} \quad \text{int} \quad \text{int}
\end{array}
\]
7. (12 pts) Using context free grammars.

Consider the following grammar for OCaml types involving int, int lists, and functions:

\[
T \rightarrow \text{int} \mid \text{T} \rightarrow \text{T} \mid \text{T list} \mid ( \text{T} )
\]

a. (2 pts) Can the grammar be parsed using a recursive descent parser? Explain your answer.

**No. Left recursion (e.g., \( T \rightarrow T \text{ list} \)) & conflicting FIRST sets (e.g., \( T \rightarrow T, T \text{ list} \))**

b. (4 pts) Modify the grammar to make the operator \( \rightarrow \) right associative, \( ( ) \) to have the highest precedence, and \( \text{list} \) to have the lowest precedence. Your grammar should be able to derive strings such as “int list list”.

\[
T \rightarrow \text{T list} \mid \text{A} \\
\text{A} \rightarrow \text{B} \rightarrow \text{A} \mid \text{B} \\
\text{B} \rightarrow \text{int} \mid ( \text{T} )
\]

c. (6 pts) Given the following productions for \( X \) and \( Y \), create a grammar that generates Ruby array references for the integer arrays \( a \) & \( b \), where arrays may be multidimensional. Your grammar should be able to generate strings such as \( a[0], a[1][0], b[1][0][1], a[b[1]] \), but not \( a, 1, [1], [a], \) or \( a[b] \).

\[
\begin{align*}
\text{S} & \rightarrow \text{X A} \\
\text{A} & \rightarrow \text{A}[B] \mid [B] \\
\text{B} & \rightarrow \text{S} \mid \text{Y}
\end{align*}
\]

8. (12 pts) Parsing

a. (6 pts) Calculate FIRST sets for the grammar below right, where \( S, A \) are nonterminals, and \( a, b, c \) are terminals.

\[
\begin{align*}
\text{FIRST}(S) & = \{ \text{b}, \text{c} \} \\
\text{FIRST}(A) & = \{ \epsilon, \text{b}, \text{c} \}
\end{align*}
\]

b. (6 pts) Using pseudocode, write only the parse_C function found in a recursive descent parser for a grammar that includes the productions \( C \rightarrow D a C \mid \epsilon \), and where FIRST(D) = \{ c, d \}. You may assume the function parse_D already exists. Use the utilities on the right.

\[
\begin{align*}
\text{S} & \rightarrow \text{Ab} \mid \text{cAb} \\
\text{A} & \rightarrow \text{Sac} \mid \epsilon
\end{align*}
\]

\[
\begin{align*}
\text{X} & \rightarrow \text{a} \mid \text{b} \\
\text{Y} & \rightarrow 0 \mid 1
\end{align*}
\]

\[
\begin{array}{|c|c|}
\hline
\text{lookahead} & \text{Variable holding next terminal} \\
\hline\hline
\text{match (x)} & \text{Function to match next terminal to x} \\
\hline
\text{error (x)} & \text{Reports parse error for input} \\
\hline
\end{array}
\]

```cpp
parse_C( ) { // your code starts here
    if ((lookahead == “c”) || (lookahead == “d”)) { // C → DaC
        parse_D( ) ; match(“a”) ; parse_C( ) ;
    }
    else ; // C → epsilon
}
```
9. (9 pts) Operational semantics

Prove what value the expression \( \text{let } x = (\text{fun } x \rightarrow 3 - x) \text{ in } x^2 \) evaluates to in an empty environment when using static scoping. In other words, find a \( v \) such that you can prove the following:

\[
\bullet \; \text{let } x = (\text{fun } x \rightarrow 3 - x) \text{ in } x^2 \Rightarrow v
\]

The operational semantics rules given in class for static scoping are below. Show the complete proof that evaluates the expression using these rules.

\[
\begin{array}{c}
\text{let } x = (\text{fun } x \rightarrow 3 - x) \text{ in } x^2 \\
\text{let } x = \text{fun } x \rightarrow 3 - x \text{ in } x^2
\end{array}
\]

\[
\begin{array}{ccc}
\text{A}(x) = v & - \\
A; x \Rightarrow v & A; n \Rightarrow n \\
A; E_1 \Rightarrow v_1 & A; E_2 \Rightarrow v_2 \\
A; E_1 \circ E_2 \Rightarrow v_1 \circ v_2 \\
A; E \Rightarrow (A, \lambda x. E) \\
A; E_1 \Rightarrow v_1 & A; x; v_1; E_2 \Rightarrow v_2 \\
A; \text{let } x = E_1 \text{ in } E_2 \Rightarrow v_2 \\
A; E_1 \Rightarrow (A', \lambda x. E) & A; E_2 \Rightarrow v_2 \\
A, x; v_2; E \Rightarrow v \\
A; E_1, E_2 \Rightarrow v
\end{array}
\]