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:  
   
   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:

```ocaml
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).

```ocaml
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).

```ocaml
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: \( S \rightarrow S \times T \mid T \rightarrow a \mid b \)

a. (3 pts) Describe the set of strings generated by the grammar \( ((alb)x)^* (alb) \text{ OR } (alb)(x(alb))^* \)

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

```
S \Rightarrow SxT \Rightarrow SxTxT \Rightarrow TxTxT \Rightarrow axTxT \Rightarrow axbTxT \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.

\[
S \rightarrow TL \\
L \rightarrow xTL | \varepsilon \\
T \rightarrow a | b
\]

5. (22 pts) Parsing
Consider the following grammar

\[
S \rightarrow Abc | dS | \varepsilon \text{ (* epsilon *)} \\
A \rightarrow aSA | f
\]

a. (8 pts) Compute First sets for S and A
   
   \[
   \text{First}(S) = \{a, d, f, \varepsilon\} \\
   \text{First}(A) = \{a, f\}
   \]

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

<table>
<thead>
<tr>
<th>lookahead</th>
<th>Variable holding next terminal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Lookahead == &quot;$&quot; when at end of input</td>
</tr>
</tbody>
</table>

\[
\text{match ( x )} \quad \text{Function to match next terminal to } x
\]

\[
\text{error ( )} \quad \text{Reports parse error for input}
\]

\[
\text{parse}_S() \\
\text{if (lookahead == "a") || (lookahead == "f")} \\
\quad \text{parse}_A(); \text{match("b")}; \text{match("e")}; \\
\text{else if (lookahead == "d")} \\
\quad \text{match("d")}; \text{parse}_S(); \\
\text{else} \\
\quad \text{parse}_S();
\]

\[
\text{parse}_A() \\
\text{if (lookahead == "a")} \\
\quad \text{match("a")}; \text{parse}_S(); \text{parse}_A(); \\
\text{else if (lookahead == "f")} \\
\quad \text{match("f")}; \\
\text{else} \\
\quad \text{error();}
\]
6. (16 pts) Operational semantics

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

\[ \cdot, x:1 \; ; (+ \; x \; 2) \rightarrow 3 \]

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:

\[ \cdot \; ; (\text{fun } z = z) \; (+ \; 1 \; 2) \rightarrow v \]

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

\[
\begin{align*}
\cdot \; ; 1 & \rightarrow 1 \\
\cdot \; ; 2 & \rightarrow 2 \\
\cdot \; ; (\text{fun } z = z) & \rightarrow (\cdot, \lambda z.z) \\
\cdot \; ; (+ \; 1 \; 2) & \rightarrow 3 \\
(z; 3 \; ; z) & \rightarrow 3 \\
\cdot \; ; (\text{fun } z = z) \; (+ \; 1 \; 2) & \rightarrow 3
\end{align*}
\]