1. (14 pts) Context Free Grammars & Automata
   a. (2 pts) Explain how context free grammars are used for programming languages.
      **CGFs are used to precisely specify syntax of programming languages**
   b. (2 pts) Describe the relationship between derivations and sentential forms.
      **Sentential forms are the strings produced by derivations**
   c. (2 pts) Describe the language accepted by the grammar: $S \rightarrow aaSb \mid aSb \mid \epsilon$
      $a^y b^x$, where $2y \geq x \geq y$, and $x, y \geq 0$
   d. (4 pts) Write a grammar for $a^x b^y a^z$, where $z = 2x - y$, for $x, y, z \geq 0$
      $S \rightarrow aSaa \mid aLba \mid L \mid aLbb \mid \epsilon$
   e. (2 pts) Name features needed by automata to recognize all binary numbers with more 1’s than 0’s.
      **DFA and stack (since language recognizable with simple CFG)**
   f. (2 pts) Explain why a finite automaton with 2 stacks can recognize many more languages than a finite automaton with 1 stack.
      **2 stacks can simulate a tape, yielding a Turing machine**

2. (14 pts) Derivations, Parse Trees, Precedence and Associativity
   For the following grammar: $S \rightarrow S \mid S \mid not S \mid true \mid false$
   a. (4 pts) List all left-most derivations for the string “not true and true”
      **D1: $S \Rightarrow S$ and $S \Rightarrow not S$ and $S \Rightarrow not true$ and $S \Rightarrow not true$ and $true$$D2: S \Rightarrow not S$ \Rightarrow not S and $S \Rightarrow not true$ and $S \Rightarrow not true$ and $true$$D2: S \Rightarrow not S$ \Rightarrow not S and $S \Rightarrow not true$ and $true$$D2: S \Rightarrow not S$ \Rightarrow not S and $S \Rightarrow not true$ and $true$$D2: S \Rightarrow not S$ \Rightarrow not S and $S \Rightarrow not true$ and $true**
   b. (2 pts) Draw the parse tree for one of the left-most derivations above.

   ![Parse Trees](image)
   **where D1 \rightarrow Tree 1, D2 \rightarrow Tree 2**
   c. (6 pts) Rewrite the grammar so that “and” is left associative and has lower precedence than “not”.
      $S \rightarrow S \mid and \mid L \mid L \rightarrow L \mid true \mid false$
   d. (2 pts) Is your rewritten grammar ambiguous?
      **No**
3. (16 pts) Parsing

For the problem, assume the term “predictive parser” refers to a top-down,
non-backtracking, recursive descent parser.

a. (10 pts) Consider the following grammar: \( S \rightarrow A c \mid b \quad A \rightarrow aS \mid \varepsilon \)

i. (4 pts) Compute First sets for each production and nonterminal

\[ \text{First}(aS) = \{ a \} \]
\[ \text{First}(\varepsilon) = \{ \varepsilon \} \]
\[ \text{First}(A) = \text{First}(aS) \cup \text{First}(\varepsilon) = \{ a, \varepsilon \} \]

\[ \text{First}(Ac) = \{ \text{First}(A) - \varepsilon \} \cup \text{First}(c) = \{ a \} \cup \{ c \} = \{ a,c \} \]
\[ \text{First}(b) = \{ b \} \]
\[ \text{First}(S) = \text{First}(Ac) \cup \text{First}(b) = \{ a,b,c \} \]

ii. (4 pts) Write a predictive parser for the grammar

\[
\text{parse}_S() \{
\text{if } ((\text{lookahead} == \text{“a”}) \lor \text{lookahead} == \text{“c”}) \text{ \{ }
\text{parse}_A();
\text{match(“c”);}\}
\text{else if (lookahead == “b”) } \text{\{ } \text{\} }
\text{match(“b”);}
\text{else error();}
\text{\} }
\text{parse}_A() \{ 
\text{If (lookahead == “a”) \{ } \text{\} }
\text{match(“a”);} \text{parse}_S();\}
\text{else ; }
\text{\} }
\text{\} }

iii. (2 pts) Use your parser to parse the string “abc”. Show the sequence of
calls in the parse, and what symbols remain at each point.

<table>
<thead>
<tr>
<th>Parse</th>
<th>Remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse_S()</td>
<td>“abc”</td>
</tr>
<tr>
<td>parse_A()</td>
<td>“abc”</td>
</tr>
<tr>
<td>match(“a”)</td>
<td>“abc”</td>
</tr>
<tr>
<td>parse_S()</td>
<td>“bc”</td>
</tr>
<tr>
<td>match(“b”)</td>
<td>“bc”</td>
</tr>
<tr>
<td>match(“c”)</td>
<td>“c”</td>
</tr>
</tbody>
</table>


b. (6 pts) Consider the following grammar: \( S \rightarrow aSc \mid ab \mid a \)
   i. (2 pts) Show why the grammar cannot be parsed by a predictive parser.

   First(aSc) \( \cap \) First(ab) = \{ a \} \( \cap \) \{ a \} = \{ a \} \( \neq \) \( \emptyset \)

   First sets of productions for S overlap → grammar not predictive

   ii. (4 pts) Rewrite the grammar so it can be parsed by a predictive parser, using the rules presented in class for left factoring & eliminating left recursion.

4. (8 pts) OCaml and Functional Programming
   a. (2 pts) Describe one advantage of functional programming

   Programs easier to analyze than imperative programs. No aliasing

   b. (2 pts) Describe the difference between the usage of “;” and “,” in OCaml

   Semicolon separates expressions, comma separates elements of a tuple

   c. (2 pts) Describe the relationship between type inference and polymorphic types

   Type inference assigns polymorphic types to variables that have multiple possible types based on how they’re used in the code

   d. (2 pts) Describe the difference between function pointers and closures

   Closures include both a function pointer and an environment

5. (10 pts) OCaml Types & Type Inference 1

   Give the type of the following OCaml expressions:

   a. (2 pts) [1,"bar"] // (int * string) list

   b. (2 pts) let rec f x = match x with // int list -> int list
      [ ] -> []
      | (h::t) -> (h+1)::(f t)

   c. (2 pts) let f (x::y) = [y;[x]] // 'a list -> 'a list list

   d. (4 pts) let f x y z = y x // 'a -> ('a -> 'b) -> 'c -> 'b

6. (12 pts) OCaml Types & Type Inference 2

   Write an OCaml expression with the following types:

   a. (2 pts) string list list // ["foo"]

   b. (4 pts) 'a * ('b list) -> ('a * 'b) list // let f (x,y::_) = [x,y]

   c. (6 pts) (int -> 'a) -> (int -> 'a) // let f x y = x (y+1)

7. (12 pts) OCaml Programs

   What are the values of the following OCaml expressions? If an error exists, describe the error.

   a. (2 pts) 1 + 2 ; 3 + 4 // 7

   b. (2 pts) [1;"foo"] // mixed types in list, "foo" has type string but used with int

   c. (2 pts) let x = 1 in let y = x+2 in let x = y+3 in x+4 // 10

   d. (3 pts) let x y = fun z -> z+y in x 1 2 // 3

   e. (3 pts) let x y = fun z -> y z in x (fun x -> x+3) 4 // 7
8. (26 pts) OCaml Programming
   For the following problems, you may use helper functions, but no library functions.
   You are given the curried version of the fold function:
   
   let rec fold f a l = match l with
   | [] -> a
   | (h::t) -> fold f (f a h) t

   a. (4 pts) Using the curried version of the \textit{fold} function, write an OCaml function
      named \textit{reverse} that when applied to a list \textit{lst} returns the list in reverse order.
      Example: reverse \([1;3;5;2;4]\) = \([4;2;5;3;1]\)
      
      \[
      \text{let reverse lst = fold (fun a h -> h::a) [] lst}
      \]

   b. (10 pts) Using the curried version of the \textit{fold} function, write an OCaml function
      named \textit{filter} with type \((\text{('a -> bool)} \rightarrow \text{'a list -> 'a list})\)
      that takes two arguments: a predicate function \textit{pred} with type \((\text{'a -> bool})\), and list \textit{lst} with type \((\text{'a list})\).
      \textit{filter} returns only the elements of \textit{lst} that return true when evaluated by \textit{pred}.
      The filtered elements must be returned in their order in \textit{lst}. You may use the reverse function above.
      Example: filter \((\text{x -> (x > 2)})\) \([1;3;5;2;4]\) = \([3;5;4]\)
      
      \[
      \text{let filter pred lst = reverse (fold (fun a h -> if (pred h) then h::a else a) [] lst)}
      \]

   c. (12 pts) Write an OCaml function named \textit{rev_map} which takes a function \textit{f} and a
      list \textit{lst}, applies \textit{f} to every element \textit{lst}, and returns the results in a new list in reverse order.
      You must implement \textit{rev_map} as a single pass over the input list (i.e., you
      cannot first apply map, then reverse the result).
      Example: rev_map \((\text{x -> x+1})\) \([1;3;5;2;4]\) = \([5;3;6;4;2]\)
      
      \[
      \text{let rev_map f lst = fold (fun a h -> (f h)::a) [] lst}
      \]