1. (Programming Languages, 6 pts) List and (briefly) describe the main steps of the compilation process (including input and resulting output). How does interpreted execution differ?

**Compilation:**
**INPUT:** source code (your program)

1) **Lexical Analysis (Lexing/scanning)**—break the input program into individual tokens such as numbers, operators, identifiers, etc. (use RE/FA)

2) **Syntax Analysis (Parsing)**—group these tokens into higher-level language constructs (uses CFGs, parse trees)

3) **Intermediate Code Generation**—translate the high-level language syntax into an equivalent low-level language/intermediate representation (ASTs)

4) **Optimization (optimal)**—optimize the executable code to make it more efficient (i.e., remove redundancies, improve algorithms, etc.)

**OUTPUT:** executable file translated to a language that can be run directly on the machine

In an interpreted language, the source code is translated and executed one instruction at a time; no separate executable is produced.

2. (Regular Expressions, 4 pts)

(a) (2pts) Write a regular expression that accepts the language:

\[ L = \{ s \mid s \text{ contains at least one } 0 \text{ and at least one } 1 \} \]. \( \Sigma = \{0, 1\} \).

Some possible solutions:

- \( 0(0|1)^*1(0|1)^*1(0|1)^*0(0|1)^* \)
- \( (0|1)^*(01|10)(0|1)^* \)
- \( (1^*0(0|1)^*10^*0^*1(0|1)^*01^* \)

(b) (2pts) Describe the language accepted by the regular expression \( (0|1)(0|1)0(0|1)^* \).

\[ L = \{ s \mid s \text{ contains at least 3 characters and the third character is } 0 \} \]
3. (Finite Automata, 10 pts)

(a) (4 pts) Give a DFA that accepts the language all strings containing at least one 0 and at least one 1 from question 2a.

Consider the following NFA:

(b) (2 pts) Give a regular expression for the language accepted by this NFA. \( \Sigma = \{0, 1\} \).

\[ 0^*|(0^*10^*(0|1)0)^* \]

(c) (4 pts) For each of the strings below, indicate whether it is accepted by the NFA. Write **YES** if the string matches, **NO** if not.

i. 0  **YES**
ii. 01011  **NO**
iii. \( \epsilon \)  **YES**
iv. 1010  **YES**
4. **(NFA to DFA, 15pts)**

Apply the subset construction algorithm to reduce the following NFA to a DFA. Indicate the NFA states associated with each state in your DFA.

```
DFA: R = \{\{S0,S2,S5\}, \{S1,S4,S5\}, \{S1, S3, S5\}, \{S1, S3\}\}
```

5. **(DFA Minimization, 12pts)**

Consider the following DFA:

(a) (4 pts) Using the Hopcroft algorithm for DFA minimization, split the states into initial partitions.

**Partitions:**
\[ P_0 = \{S0, S3\} \text{ (final states)} \]
\[ P_1 = \{S1, S2, S4\} \text{ (non-final states)} \]
(b) (8 pts) What criteria is used to determine when to further split partitions? Do the partitions from part (a) need to be split? List the new partitions from the next iteration of the Hopcroft algorithm, if any.

A partition needs to be split if members have transitions to different partitions for the same input.

\( P_1 \) from part a needs to be split. The resulting partitions will depend on the choice of reference state \( r \). Any member that transitions to a different partition than \( r \) for the same input is added to a new partition. Possible solutions are:

<table>
<thead>
<tr>
<th>Reference State</th>
<th>Partitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>( P_1 = {S1} )  \ ( P_2 = {S2, S4} )</td>
</tr>
<tr>
<td>S2</td>
<td>( P_1 = {S2} )  \ ( P_2 = {S1, S4} )</td>
</tr>
<tr>
<td>S4</td>
<td>( P_1 = {S4} )  \ ( P_2 = {S1, S4} )</td>
</tr>
</tbody>
</table>

6. (Context Free Grammars, 15 points)
Consider the following grammar where \( \Sigma = \{a,b\} \), \( N = \{S,T\} \), and \( S \) is the start symbol:
\[
S \rightarrow S+T | T \\
T \rightarrow a | b
\]

(a) (2 pts) Give the language accepted by this grammar.

Possible REs:
\[
((a|b)+)*(a|b) \\
(a|b)(+(a|b))^*
\]

(b) (3 pts) Provide a leftmost derivation of the string \( b+a+b \).

\[
S \Rightarrow S+T \Rightarrow S+T+T \Rightarrow T+T+T \Rightarrow b+T+T \Rightarrow b+a+T \Rightarrow b+a+b
\]

(c) (2 pts) Draw a parse tree for the string \( b+a+b \).
(d) (8 pts) What would happen if we tried to use a recursive descent parser (RDP) to parse this language? Use the algorithms described in class to transform the grammar so it can be parsed with an RDP.

Because of the left recursion, this grammar would cause an infinite loop while parsing S with a recursive descent parser.

New grammar:

\[
\begin{align*}
S & \rightarrow TL \\
L & \rightarrow +TL | \epsilon \\
T & \rightarrow a | b
\end{align*}
\]

7. (Ruby Features, 8 points)

What is the output (if any) of the following Ruby programs? Write FAIL if code does not execute.

(a) \(x= \{\} \)
\[
x[\text{\textquotesingle\textquotesingle}foo\text{\textquotesingle\textquotesingle}] = 4 \\
x[\text{\textquotesingle\textquotesingle}bar\text{\textquotesingle\textquotesingle}] = \text{\textquotesingle\textquotesingle}hello\text{\textquotesingle\textquotesingle} \\
\text{puts} x[\text{\textquotesingle\textquotesingle}foo\text{\textquotesingle\textquotesingle}] \\
\text{puts} x[\text{\textquotesingle\textquotesingle}foo\text{\textquotesingle\textquotesingle]].\text{class} \\
x[\text{\textquotesingle\textquotesingle}hello\text{\textquotesingle\textquotesingle}]
\]

(b) \(\text{strings=} \left[\text{\textquotesingle\textquotesingle}cmsc330\text{\textquotesingle\textquotesingle}, \text{\textquotesingle\textquotesingle}cmps250\text{\textquotesingle\textquotesingle}, \text{\textquotesingle\textquotesingle}cmsc umd\text{\textquotesingle\textquotesingle}\right] \)
\[
\text{strings}.\text{each} \{ |s| \\
\quad s = \sim /cmsc(([\backslash d])*)/ \\
\quad \text{puts} $1 \\
\quad \text{puts} $2 \\
\}
\]
(c) x = "hello\n"
    puts x.chomp  # OUTPUT = hello
    puts x
    y = x.chomp!
    puts '#{x} #{y}',

8. (Ruby Programming, 30 points)
As mentioned in class, scripting languages are often used for text processing applications. In this problem, you will write a short Ruby program to analyze sensor data measuring the water quality of the Chesapeake Bay. The first line of a sensor data file will consist of a tab-delimited header describing the attributes in the rest of the file:

<table>
<thead>
<tr>
<th>ID</th>
<th>DATE</th>
<th>TIME</th>
<th>PARAMETER</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>177531</td>
<td>01/30/2009</td>
<td>09:12</td>
<td>PH</td>
<td>8.07</td>
</tr>
<tr>
<td>177531</td>
<td>01/30/2009</td>
<td>09:12</td>
<td>SALINITY</td>
<td>15.89</td>
</tr>
<tr>
<td>177532</td>
<td>01/30/2009</td>
<td>09:14</td>
<td>TN</td>
<td>0.4497</td>
</tr>
</tbody>
</table>

Implement a class BaySensor that includes the following methods:

(a) (18 pts) parse(filename) reads the contents of a file “filename” and stores the records in the BaySensor class. Return true if filename is a valid sensor file with the format above; otherwise, return false.

(b) (6 pts) find_record(id) takes a record id and returns the rest of the record information (i.e., date, time, parameter, value). If the id is not found, return nil.

(c) (6 pts) avg(p) takes a parameter name p and will output its average value over the entire sensor file. If the parameter does not occur in the file, return nil.

You can assume that in all properly formatted files, dates and times will be within valid ranges. In addition the find_record and avg methods will only be called for sensor data where parse returned true. The parse function can also store values in class instance variables that can be used by the find_record and avg methods.
Helpful functions:

- `File.new(filename, mode)` // opens filename in mode, returns File
- `file.eof?` // is File object file at end?
- `file.readline` // read & return single line from file
- `file.readlines` // read entire file, return array of lines

```ruby
class BaySensor
  def initialize()
    @data = {}
  end
  # parse method
  def parse(filename)
    line_num = 1
    file = File.new(filename, "r") # open file
    until file.eof? do               # repeat until EOF
      line = file.readline           # read lines from file
      if line =~ /^ID\tDATE\tTIME\tPARAMETER\tVALUE$/ #match header
        if line_num != 1
          puts "Syntax error line #{line_num}"
          return false
        end
        elsif line =~ /^\d+\t\d\d/\d\d/\d\d\d\d\\d\d\d\d\t\w+\t\d\d\?\d\d\d$/
          # create a record with the line of data
          record = {"date" => $2, "time" => $3, "parameter" => $4, "value" => $5}
          # add the record to our data hash, keyed by id
          id = $1.to_i
          if(@data[id] == nil) then @data[id] = Array.new end
          @data[id] << record
        else
          puts "Syntax error line #{line_num}"
          return false
        end
    line_num += 1
    end
    return true
  end
  # find_record method
  def find_record(id)
    return @data[id]
  end
  # avg method
  def avg(p)
    sum = 0.0;
    count = 0;
    @data.keys.each{ |k|
      @data[k].each{ |r|
        if r["parameter"] == p
          sum += r["value"].to_f
          count += 1
        end
      }
    }
    return sum / count
  end
end
```