1 Part I: Short Answer Problems (30 points)

1. Describe the difference between a high-level programming language and a low-level programming language.

2. Give a regular expression that describes words that start with the letters 'a' or 'z'.

3. Given the grammar:

   ```
   exp : prim ('@' prim | '#' prim)*;
   prim : '<' W '>' ;
   W : ('a'..'z') ;
   ```

   Please answer the following:

   (a) Compute the lookahead set for rule 'exp'.

   (b) Does the string 'a@b#c' belong to the language of the grammar? Why? Why not?

   (c) If the '@' operator has a higher precedence than the '#' operator, how would you rewrite the grammar to reflect that?

4. Given this grammar,

   ```
   sen : rambling followed by one_ending
       | rambling followed by another_ending
   ```
Is the above grammar a LL(1) grammar? Why? Why not? If not, how would you rewrite it to make it a LL(1) grammar?
2 Part II: Programming Problems (70 points)

MiniBasic is a programming language available to Windows users to perform system level programming. For the full language definition look in the reference section here:

http://www.malcolmmclean.site11.com/www/Minibasic/MiniBasicHome.html

Here we define a subset of that language, call it Mini\textsuperscript{2}Basic, with the following features: \(^1\)

The full grammar of the language is available on the course website as mini2basic.g.

- **Program** – A program is a list of statements
- **Assignment Statement** – An assignment statement has the form \texttt{ID = exp} and sets the value of the variable \texttt{ID} to the value of the expression \texttt{exp}.
- **Input Statement** – An input statement has the form \texttt{input STRING, ID} and prints out the optional prompt string \texttt{STRING}, waits for the user input, and set the value of the variable \texttt{ID} to the user value. The user is only allowed to enter integer values. Example:

  \[
  \text{input "Enter a value for variable x: ", x}
  \]

- **Print Statement** – The print statement is the keyword \texttt{print} followed by a comma separated list of one or more values. Example:

  \[
  \text{input "Enter a value for variable x: ", x}
  \text{print "The value of variable x is ", x}
  \]

- **End Statement** – The end statement terminates the execution of the program.
- **If Statement** – In the If-statement the conditional expression is followed by one or more then-statements which is optionally followed by one or more else-statements. Example:

\(^1\)Mini\textsuperscript{2}Basic stands for MiniMiniBasic.
input "enter a value: ", a
if a==5 then
    print "a is equal to 5"
else
    print "a is not equal to 5"
endif

• While Loop – In the while loop the conditional expression is followed by one or more statements in the body of the loop. Example:

    a = 1
    while a <= 100
        print a
        a = a + 1
    endwhile

• For Loop – The for loop initializes the loop variable with a start value and then at each loop iteration increases the loop variable by the amount given in the step value. The loop terminates once the loop variable reaches the end value. If the step value is missing then a step value of 1 is assumed. The step value can also be negative. Note, in the following example the keyword rem starts a comment line, which is the usual way to write a comment in Basic. Example:

    rem print a list of integers from 1 through 10
    for x = 1 to 10
        print x
    next x

    Another example:

    rem print the even numbers between 1 and 10
    for x = 2 to 10 step 2
        print x
    next x

    A final example:
rem print the odd numbers in reverse order between 1 and 10
for x = 9 to 1 step -2
    print x
next x

• Expressions – Expressions are best summarized by the grammar productions:

\[\text{exp} : \text{logexp};\]
\[\text{logexp} : \text{relexp} ('\&' \text{relexp} | '\|' \text{relexp})*;\]
\[\text{relexp} : \text{addexp} ('==' \text{addexp} | '<=' \text{addexp})*;\]
\[\text{addexp} : \text{mulexp} ('+' \text{mulexp} | '-' \text{mulexp})*;\]
\[\text{mulexp} : \text{atom} ('\*' \text{atom} | '\/' \text{atom})*;\]

The grammar encodes the standard precedences for these operators. We have the logical operators \& (and) and \| (or); the relational operators == (equal) and <= (less equal); the additive operators + (addition) and - (subtraction); and finally the multiplicative operators * (multiplication) and / (division).

The logical operators work on integer values as follows, anything that is not equal to zero is interpreted as true and a value zero is interpreted as false. For example

\[0 \& 20 \Rightarrow 0\]

but

\[-50 \& 20 \Rightarrow 1\]

The atomic expressions are best summarized by the following grammar production

\[\text{atom} : '\(' \text{exp} '\)',\]
\[| '!\text{atom}\]
\[| \text{ID}\]
\[| \text{num}\]

This should be self-explanatory with the exception of the ! operator. This operator is the logical not operator and sets any value not equal to 0 to the value 0. Conversely it sets a value 0 to the value of 1.
Here is another example program pulling this all together:

```plaintext
rem compute the factorial of a number
input "Enter a value: ", x
if x <= 0 then
    print "illegal input value"
    end
endif

rem input ok - continue computation
y = 1
for i = 1 to x
    y = y * i
next i
print "The factorial of ",x," is ",y
```

**Tasks**

- Given this language definition together with the grammar write an interpreter for this language as a standalone application.
- If variables are used before they were defined assume a value 0 assigned to them.
- Demonstrate that your interpreter works by showing that it correctly interprets the example programs given above.
- You should **not** use the ANTLR builtin AST and pattern matching, IT DOES NOT WORK. Instead you will need to define your own AST and write the interpreter as an external visitor. (Hint: we developed an interpreter for simple1 with an external visitor in slide set csc402-ln010.pdf).

**Deliverables**

On Sakai: Hand in

- a copy of the answers to the short answer questions
• a copy of your code for the long answer question, together with proof that your interpreter works, i.e., screen shots of your interpreter working on examples. Don’t forget to hand in the source code for your examples!

NOTE: We will test your interpreter on an additional test suite, so it is in your best interest to make your interpreter as robust as you can.