1. Arithmetic operators in a programming language are typically left associative with the notable exception of exponentiation (\(^\)\) which is usually right associative.

Associativity can be captured in a grammar. For a left associative dyadic operator \(\text{lop}\) we define a production of the form:

\[
<\text{expr}> \rightarrow <\text{term}>
| <\text{expr}> \text{ lop } <\text{term}>
\]

For example, \(a+b+c\) is evaluated from the left to the right by summing \(a\) and \(b\) first. The parse tree of \(a+b+c\) (assuming that \(<\text{term}>\) represents identifiers) is:

```
<expr>
  / | |
 <expr> + <term>
  / | |
 <expr> + <term> c
 | | |
 <term> b
 | a
```

As you can see, the left subtree represents \(a+b\) which is a subexpression of \(a+b+c\), because \(a+b+c\) is parsed as \((a+b)+c\).

Note that the production for a left associative operator is left recursive. To eliminate left recursion, we can rewrite the grammar into:

\[
<\text{expr}> \rightarrow <\text{term}> <\text{term}_\text{tail}>
| <\text{term}_\text{tail}> \text{ lop } <\text{term}> <\text{term}_\text{tail}>
| \text{ empty}
\]

This (part of the) grammar is LL(1) and therefore suitable for recursive descent parsing. However, the parse tree structure does not capture the left-associativity of the \(\text{lop}\) operator anymore.

Draw the parse tree of \(a+b+c\) using the LL(1) grammar shown above. You may assume that \(<\text{term}>\) represents identifiers. Hint: draw the tree from the top down by simulating a top-down predictive parser.

2. For a right associative operator \(\text{rop}\) we can create a grammar production of the form:

\[
<\text{expr}> \rightarrow <\text{term}>
| <\text{term}> \text{ rop } <\text{expr}>
\]

An example right associative operator is exponentiation \(^\) and \(a^b^c\) is evaluated from the right to the left such that \(b^c\) is evaluated first.

Draw the parse tree of \(a^b^c\) (you may assume that \(<\text{term}>\) represents identifiers).

3. The precedence of an operator indicates the priority of applying the operator relative to other operators. For example, multiplication has a higher precedence than addition, so \(a+b*c\) is evaluated by multiplying \(b\) and \(c\) first. In other words, multiplication groups more tightly

COP4020 Programming Assignment 4
compared to addition. The rules of operator precedence vary from one programming language to another.

The relative precedences between operators can be captured in a grammar as follows:

- A nonterminal is introduced for every group of operators with identical precedence. The nonterminal of the group of operators with lowest precedence is the nonterminal for the expression as a whole.
- Productions for operators with the lowest precedence are defined first.
- Productions for left associative binary operators are written in a form suitable for recursive descent parsing, which means that the left recursion must be eliminated:

  
  |<e1>     -> <e1> + <e2>
  |<e2>

is rewritten into

  |<e1>     -> <e2> <e1_tail>
  |<e1_tail> -> <e2> <e1> <e1_tail>
  |empty

Here is a complete outline, where <expr> is the start symbol:

  |<expr>     -> <e1> <e1_tail>
  |<e1>      -> <e2> <e2_tail>
  |<e1_tail>  -> <lowest_op> <e1> <e1_tail>
  |empty
  |<e2>      -> <e3> <e3_tail>
  |<e2_tail>  -> <one_but_lowest_op> <e2> <e2_tail>
  |empty
  |...
  |<eN>      -> (' <expr> ')'
  |'-', <eN>
  |identifier
  |number
  |<eN_tail>  -> <highest_op> <eN> <eN_tail>
  |empty

where <lowest_op> is a nonterminal denoting all operators with the same lowest precedence, etc.

The following Java program uses these concepts to implement a recursive descent parser for a calculator language:

```java
/* Parser.java
   Implements a parser for a calculator language
   Uses java.io.StreamTokenizer and recursive descent parsing

   Compile:
   javac Parser.java
   */
import java.io.*;
/* Calculator language grammar:

  |<expr>     -> <term> <term_tail>
  |<term>     -> <factor> <factor_tail>
  |<term_tail> -> <add_op> <term> <term_tail>
  |empty
```
public class Parser {
    private static StreamTokenizer tokens;
    private static int token;
    public static void main(String argv[]) throws IOException {
        InputStreamReader reader;
        if (argv.length > 0)
            reader = new InputStreamReader(new FileInputStream(argv[0]));
        else
            reader = new InputStreamReader(System.in);
        // create the tokenizer:
        tokens = new StreamTokenizer(reader);
        tokens.ordinaryChar('.');
        tokens.ordinaryChar('-');
        tokens.ordinaryChar('/');
        // advance to the first token on the input:
        getToken();
        // check if expression:
        expr();
        // check if expression ends with ‘;’
        if (token == (int)';')
            System.out.println("Syntax ok");
        else
            System.out.println("Syntax error");
    }
    // getToken - advance to the next token on the input
    private static void getToken() throws IOException {
        token = tokens.nextToken();
    }
    // expr - parse <expr> -> <term> <term_tail>
    private static void expr() throws IOException {
        term();
        term_tail();
    }
    // term - parse <term> -> <factor> <factor_tail>
    private static void term() throws IOException {
        factor();
        factor_tail();
    }
    // term_tail - parse <term_tail> -> <add_op> <term> <term_tail> | empty
    private static void term_tail() throws IOException {
        if (token == (int)+'+' || token == (int)-')'
            add_op();
        term();
        term_tail();
    }
    // factor - parse <factor> -> '(' <expr> ')' | '-' <expr> | identifier | number
    private static void factor() throws IOException {
        if (token == (int)'(')
            getToken();
        expr();
        if (token == (int)')')
            getToken();
        else System.out.println("closing ')' expected");
    }
}

Download this example parser Java program from:
http://www.cs.fsu.edu/~engelen/courses/COP4020/Parser.java

Compile it as indicated and execute as follows:
```java
java Parser
```
- Give the output of the program when you type `2*(1+3)/x`; and explain why this expression is accepted by the parser by drawing the parse tree.
- Give the output of the program when you type `2x+1`; and explain why it is not accepted. At what point in the program does the parser fail? Why?

4. Extend the parser program to include syntax checking of function calls with one argument with a new production for `<factor>`:

```plaintext
<factor> -> '(' <expr> ')' 
        | '-' <factor> 
        | number 
        | <varfun>
<varfun> -> identifier '(' <expr> ')' 
        | identifier
```

- Is this grammar still LL(1)? To determine this, implement the new productions. How far does your recursive descent parser have to look ahead to select a production for `<varfun>`?
- Test your implementation on the valid expression `2*f(1+a)`; Draw the parse tree of `2*f(1+a)`.
5. Extend the parser to include the exponentiation \(^\) operator, such that expressions like \(-a^2\) and \(-(a^b)^{(c\cdot d)}^{(e+f)}\) can be parsed. Note that exponentiation is right associative and has the highest precedence (even higher than unary minus, so \(-a^2\) is evaluated by evaluating \(a^2\) first. To implemented this, you must add a \(<\text{power}>\) nonterminal and also change the production of \(<\text{factor}>\) so that the parse tree of \(-a^{(-3)}\) is:

```
<factor>
   / \
  -   <power>
     /   \
 <varfun> ^   <power>
       /     |
      a      <varfun>
           |
           ( <expr> )
        /   \
       <term>   <term_tail>
         /   \   
        <factor> <factor_tail> empty
          /   \   |
         -   <power> empty
           |
           <varfun>
           |
            3
```