1. **(a)** Show a stratified LL(1) grammar for arithmetic expressions over the three terminals \(\{\text{num, +, -}\}\) so that “-” can be used as a binary operator in subtraction and as a unary operator in arithmetic negation. The grammar should yield a concrete syntax tree reflecting that unary negation is right-associative and binds more tightly than addition and subtraction. Addition and subtraction have the same precedence and are left-associative. For example, the expression \(2 - 3 + -4 - -5\) should have a concrete syntax tree that reflects the ordering \(((2-3)+(-4))-(-(-5))\).

**Soln (a):** Write an augmented grammar reflecting the desired precedence and associativity

\[
\begin{align*}
S & ::= E \ \$ \quad \text{augmented grammar} \\
E & ::= E \ ( + \ | \ - \ ) \ U \ | \ U \quad \text{binary opns: left associative, lowest precedence} \\
U & ::= - \ U \ | \ T \quad \text{unary opn: right associative, higher precedence} \\
T & ::= \text{num} \quad \text{leaves of the syntax trees: numbers}
\end{align*}
\]

Place the grammar into LL(1) form by removing the left recursion in the second rule.

\[
\begin{align*}
S & ::= E \ \$ \quad \text{(1)} \\
E & ::= U \ ( ( + \ | \ - \ ) \ U )^* \quad \text{(2)} \\
U & ::= - \ U \ | \ T \quad \text{(3)} \\
T & ::= \text{num} \quad \text{(4)}
\end{align*}
\]

Rules (2) and (3) are the only rules with choices. Rule (3) trivially meets the LL(1) condition. In rule (2) the Kleene star choice point can be resolved with a single terminal lookahead. Since

\[
\text{Predict}( ( + \ | \ - \ ) \ U ) = \{+,-\}
\]

is disjoint from

\[
\text{Predict}(\epsilon) = \text{Follow}(E) = \{$\}
\]

So our grammar satisfies the LL(1) condition and is suitable for recursive descent parsing.
1. (b) Describe how you would modify the simpleAST example on the class website to build Expr ASTs using BinExpr, UnaryExpr and NumExpr nodes? Describe your extensions to the scanner, parser, AbstractSyntaxTrees, and visitors. It’s not necessary to write out complete code, but if you are interested you can extend the simpleAST example to try it out.

Soln 1(b): (refer to simpleAST example (#2 on web page) to understand the additions)

To scan the “-“ terminal we need to:
- Add a MINUS token to the TokenKind enum in the scanner.
- Extend the scanner to return a MINUS token when encountered.

To build an AST with unary expressions add to the AbstractSyntaxTrees package

```java
public class UnaryExpr extends Expr {
    public Token oper;
    public Expr right;
}
```
Also add a constructor in this class.

To traverse ASTs we need to add to interface Visitor<Inh, Syn> the method
- public Syn visitUnaryExpr(UnaryExpr expr, Inh arg);
and add the corresponding visit method in the UnaryExpr class.

To build the AST we need to modify the Parser class
- Extend the parseE() method to use whichever token (+ or -) is encountered in constructing a BinExpr
- Add a parseU() method

```java
Expr parseU() {
    if (currentToken.kind == TokenKind.MINUS) {
        Token negation = currentToken;
        acceptIt();
        return new UnaryExpr(negation, parseU());
    } else
        return parseT();
}
```

To evaluate the AST
- Extend the DisplayAST visitor in the TraverseAST package to include.

```java
public String visitUnaryExpr(UnaryExpr e, Object arg){
    return "(" + "-" + e.right.visit(this, null) + ")";
}
```
(display of "-" in a BinaryExpr is automatically correct as it uses the operator spelling)
- Extend the Eval visitor in the TraverseAST package to include unary negation

```java
public Integer visitUnaryExpr(UnaryExpr e, Object arg){
    // evaluate right subtree
    Integer rval = e.right.visit(this, null);
    return -rval;
}
```
- Also add the subtraction case in evaluation of a BinaryExpr
1(c) Draw the AST for \( 2 - 3 + -4 - - -5 \) using \textit{BinExpr}, \textit{UnaryExpr} and \textit{NumExpr} nodes and show the spelling of the tokens in the AST.

\textbf{Soln 1(c).} The AST is shown below. It corresponds to \(((2 - 3) + (-4)) - (-(-5)))\)
2. Using the class constructors from the miniJava AbstractSyntaxTrees package, show how to build the AssignStmt AST for the miniJava statement \( x = 1 - -a[3] \);

**Soln (2):** We use the AST constructors. The leaves are constructed from tokens. All steps are spelled out for clarity. The constructed AST as shown by ASTDisplay and as a tree are on the next page.

```java
// IdRef "x" on left hand side
Token id_x_token = new Token(TokenKind.IDENTIFIER, "x", null);
Identifier id_x = new Identifier(id_x_token);
IdRef idr_x = new IdRef(id_x, null);

// IntLitExpr 1 on right hand side
Token intlit_1_token = new Token(TokenKind.INTLITERAL, "1", null);
IntLiteral intlit_1 = new IntLiteral(intlit_1_token);
LiteralExpr litexp_1 = new LiteralExpr(intlit_1, null);

// RefExpr "a[3]" on right hand side
Token id_a_token = new Token(TokenKind.IDENTIFIER, "a", null);
Identifier id_a = new Identifier(id_a_token);
Token intlit_3_token = new Token(TokenKind.INTLITERAL, "3", null);
IntLiteral intlit_3 = new IntLiteral(intlit_3_token);
LiteralExpr le_3 = new LiteralExpr(intlit_3, null);
IxIdRef ixidref_a3 = new IxIdRef(id_a, le_3, null);
RefExpr refexp_a3 = new RefExpr(ixidref_a3, null);

// Expr "1 - -a[3]" on right hand side
Token minus_token = new Token(TokenKind.MINUS, ",-\'', null);
Operator minus_op = new Operator(minus_token);
UnaryExpr un_exp = new UnaryExpr(minus_op, refexp_a3, null);
BinaryExpr bin_exp = new BinaryExpr(minus_op, litexp_1, un_exp, null);

// AssignStmt
AssignStmt as = new AssignStmt(idr_x, bin_exp, null);

// show tree (not part of the answer)
ASTDisplay astd = new ASTDisplay();
astd.showTree(as);
```
====== AST Display ================
AssignStmt
  IdRef
    "x" Identifier
  BinaryExpr
    "-" Operator
    LiteralExpr
      "1" IntLiteral
  UnaryExpr
    "-" Operator
    RefExpr
      IxIdRef
        "a" Identifier
        LiteralExpr
          "3" IntLiteral

========================================