Computer Science Department

Senior Team Portfolio

Compilers

CSCI 468

Spring Semester of 2019

by

Spencer Debuf, Gregor Smirnow, & Kenneth Taylor

Montana State University

Bozeman, Montana

Section 1: Program

```
grammar Little;
program: 'PROGRAM' id 'BEGIN' pgm_body 'END';
id: IDENTIFIER;
pgm_body: decl func_declarations | decl ;
decl: string_decl decl | var_decl decl |;
string_decl: 'STRING' id ':=' str ';';
str: STRINGLITERAL;
var_decl: var_type id_list ';';
var_type: 'FLOAT' | 'INT';
any_type: var_type | 'VOID';
id_list: id id_tail;
id_tail: ',' id id_tail |;
param_decl_list: param_decl param_decl_tail |;
param_decl: var_type id;
param_decl_tail: ',' param_decl param_decl_tail |;
func_declarations: func_decl func_declarations | func_decl;
func_decl: 'FUNCTION' any_type id '(' param_decl_list ')' 'BEGIN' func_body 'END';
func_body: decl stmt_list;
stmt_list: stmt stmt_list_tail |;
stmt_list_tail: stmt stmt_list_tail |;
stmt: base_stmt | if_stmt | while_stmt;
base_stmt: assign_stmt | read_stmt | write_stmt | return_stmt;
assign_stmt: assign_expr ';';
assign_expr: id ':=' expr;
read_stmt: 'READ' '(' id_list ')' ';';
write_stmt: 'WRITE' '(' id_list ')' ';';
return_stmt: 'RETURN' expr ';';
expr: add_minus_expression;
add_minus_expression: add_minus_expression addop add_minus_expression |
multiply_divide_expression;
multiply_divide_expression: expression_component mulop multiply_divide_expression |
expression_component;
expression_component: primary | call_expr;
call_expr: id '(' expr_list ')';
expr_list: expr expr_list_tail |;
expr_list_tail: ',' expr expr_list_tail |;
primary: '(' expr ')' | id | INTLITERAL | FLOATLITERAL;
addop: '+' | '-';
mulop: '*' | '/';
if_stmt: 'IF' '(' cond ')' decl stmt_list else_part 'ENDIF' | 'IF' '(' cond ')' decl stmt_list 'ENDIF' ;
else_part: 'ELSE' decl stmt_list;
```

```
cond: expr compop expr;
compop: '<' | '>' | '=' | '!=' | '<=' | '>=';
while_stmt: 'WHILE' '(' cond ')' decl stmt_list 'ENDWHILE';
IDENTIFIER: LETTER ALPHANUMERIC*;
INTLITERAL: DIGIT+;
FLOATLITERAL: DIGIT* '.' DIGIT+;
STRINGLITERAL: '"' ~["]* '"';
DIGIT: [0-9];
ALPHANUMERIC: [a-zA-Z0-9];
LETTER: [a-zA-Z];
COMMENT: '--' ~['\n']* '\n' -> skip ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

```
/*
*/
package littlecompiler;

import littlecompiler.GeneratedGrammarFiles.LittleBaseListener;
import littlecompiler.GeneratedGrammarFiles.LittleLexer;
import littlecompiler.GeneratedGrammarFiles.LittleParser;
import org.antlr.v4.runtime.tree.ParseTree;
import org.antlr.v4.runtime.tree.ParseTreeWalker;
import symboltables.SymbolTableVisualizer;

/**
 *  The entity that performs scanning / parsing / code generation.
 */
public class Compiler
{
   /* The dependencies of the Compiler */
   private final LittleBaseListener listener;
   private final LittleLexer lexer;
   private final LittleParser parser;
   private ParseTree parseTree;

   public Compiler(
      LittleBaseListener listener,
      LittleLexer lexer,
      LittleParser parser)
   {
      this.listener = listener;
      this.lexer = lexer;
      this.parser = parser;
```

```
    }

    public void printSymbolTable()
    {
        new SymbolTableVisualizer().printSymbolTable(
            listener.symbolTables.peek());
    }

    /**
     *  STEP 1 + 2:
     *  Parsing the generated tokens into a parse tree.
     */
    public void generateAndParseTokens()
    {
        try
        {
            parseTree = parser.program();

            ParseTreeWalker walker = new ParseTreeWalker();
            walker.walk(
                listener,
                parseTree);
        }
        catch (Exception e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

```
/*
*/
package littlecompiler;

import java.io.IOException;

public class LittleCompiler
{
    /**
     * @param arguments
     *  The command line arguments
     * @throws
     *  java.io.IOException
```

```
     */
    public static void main(String[] arguments) throws IOException
    {
        for (String compiledFilePath : arguments)
        {
            CompilerFactory compilerFactory = new CompilerFactory();
            Compiler littleCompiler = compilerFactory
                .createCompiler(compiledFilePath);

            littleCompiler.generateAndParseTokens();
        }
    }
}
```

```
/*
*/
package littlecompiler;

import java.io.IOException;
import littlecompiler.GeneratedGrammarFiles.LittleBaseListener;
import littlecompiler.GeneratedGrammarFiles.LittleLexer;
import littlecompiler.GeneratedGrammarFiles.LittleParser;
import org.antlr.v4.runtime.CharStream;
import org.antlr.v4.runtime.CharStreams;
import org.antlr.v4.runtime.CommonTokenStream;

/**
 *  A factory class that handles creating Compilers and injecting their
 *  dependencies. The dependencies are created here to allow for unit testing
 *  of the Compiler class.
 */
public class CompilerFactory
{
    /**
     * @param compiledFilePath
     *  The file path of the file being compiled.
     * @return
     *  A Compiler to compile the provided file.
     * @throws IOException
     */
    public Compiler createCompiler(String compiledFilePath) throws IOException
    {
        return createCompiler(
```

```
            generateCharStreamFromFilePath(
                compiledFilePath));
    }

    /**
     * @param compiledFilePath
     *  The file path of the file being compiled.
     * @return
     *  A CharStream of all the characters in the compiled file.
     * @throws IOException
     */
    private CharStream generateCharStreamFromFilePath(
        String compiledFilePath) throws IOException
    {
        return CharStreams.fromFileName(compiledFilePath);
    }

    /**
     * @param compiledFileCharStream
     *  A CharStream of all the characters in the compiled file.
     * @return
     *  A Compiler for compiling the provided CharStream.
     */
    private Compiler createCompiler(CharStream compiledFileCharStream)
    {
        LittleBaseListener littleListener = new LittleBaseListener();

        LittleLexer littleLexer =
            new LittleLexer(compiledFileCharStream);

        LittleParser littleParser = new LittleParser(
            new CommonTokenStream(littleLexer));
        littleParser.setErrorHandler(new LittleErrorStrategy());

        return new Compiler(
            littleListener,
            littleLexer,
            littleParser);
    }
}
```

```
/*
*/
```

```java
package littlecompiler;

import java.util.List;
import littlecompiler.GeneratedGrammarFiles.LittleLexer;
import org.antlr.v4.runtime.Token;

/**
 *  A utility class that prints the generated tokens in the format:
 *     Token Type: {type}
 *     Value: {value}
 *     Token Type: {type}
 *     Value: {value}
 */
public class TokenVisualizer
{
   /* A List of generated Tokens */
   private final List<Token> tokens;

   public TokenVisualizer(List<Token> tokens)
   {
      this.tokens = tokens;
   }

   /**
    *  Generates a String which displays info on all the generate tokens.
    * @return
    *  A String with info on all the generated tokens in your compiled program.
    */
   public String getTokenInfoString()
   {
      StringBuilder stringBuilder = new StringBuilder();

      tokens.forEach(token ->
      {
         String nextTokenInfo = generateTokenDataString(token);
         stringBuilder.append(nextTokenInfo);
      });

      return stringBuilder.toString();
   }

   /**
    *  Generates a String which describes a token.
```

```
    * </p>
    * @param token
    *  The token whose info is contained in this String.
    * @return
    *  A String of the format:
    *     Token Type: {type}
    *     Value: {value}
    */
   private String generateTokenDataString(Token token)
   {
      int tokenTypeID = token.getType();
      String tokenType = LittleLexer.ruleNames[tokenTypeID - 1];

      return
         "Token Type: " + tokenType + "\nValue: " + token.getText() + "\n";
   }
}
```

```
/*
*/
package littlecompiler;

import org.antlr.v4.runtime.DefaultErrorStrategy;
import org.antlr.v4.runtime.InputMismatchException;
import org.antlr.v4.runtime.Parser;
import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.RecognitionException;
import org.antlr.v4.runtime.Token;
import org.antlr.v4.runtime.misc.ParseCancellationException;

/**
 * @summary
 *  This class defined protocols for handling errors that occur during scanning
 *  and parsing.
 */
public class LittleErrorStrategy extends DefaultErrorStrategy
{
   @Override
   public Token recoverInline(Parser recognizer)
      throws RecognitionException
   {
      InputMismatchException e = new InputMismatchException(recognizer);
```

```
        for (ParserRuleContext context = recognizer.getContext(); context != null; context =
context.getParent()) {
            context.exception = e;
        }

        throw new ParseCancellationException(e);
    }

    @Override
    public void recover(
        Parser recognizer,
        RecognitionException e)
    {
        for (ParserRuleContext context = recognizer.getContext(); context != null; context =
context.getParent())
        {
            context.exception = e;
        }

        throw new ParseCancellationException(e);
    }
}
```

```
/*
*/
package littlecompiler;

import symboltables.SymbolTable;

public class SymbolTableContainer
{
    private static SymbolTableContainer instance;
    public SymbolTable globalSymbolTable;

    public static SymbolTableContainer getInstance()
    {
        return instance == null
            ? instance = new SymbolTableContainer()
            : instance;
    }

    private SymbolTableContainer()
    {
```

```
    }

    public SymbolTable getGlobalTable()
    {
        return globalSymbolTable;
    }

    public void setGlobalTable(SymbolTable globalSymbolTable)
    {
        this.globalSymbolTable = globalSymbolTable;
    }
}
```

```
/*
*/
package littlecompiler.GeneratedGrammarFiles;

// Generated from Little.g4 by ANTLR 4.7.1

import AbstractSyntaxTree.AST;
import AbstractSyntaxTree.Nodes.AssignNode;
import AbstractSyntaxTree.Nodes.BeginFunctionNode;
import AbstractSyntaxTree.Nodes.ConditionNode;
import AbstractSyntaxTree.Nodes.DeclarationNode;
import AbstractSyntaxTree.Nodes.ElseNode;
import AbstractSyntaxTree.Nodes.EndFunctionNode;
import AbstractSyntaxTree.Nodes.FloatLiteralNode;
import AbstractSyntaxTree.Nodes.FunctionCallNode;
import AbstractSyntaxTree.Nodes.Operators.DivideNode;
import AbstractSyntaxTree.Nodes.FunctionNode;
import AbstractSyntaxTree.Nodes.IfNode;
import AbstractSyntaxTree.Nodes.InputParameterListNode;
import AbstractSyntaxTree.Nodes.InputParameterNode;
import AbstractSyntaxTree.Nodes.IntLiteralNode;
import AbstractSyntaxTree.Nodes.Operators.EqualNode;
import AbstractSyntaxTree.Nodes.Operators.GreaterThanNode;
import AbstractSyntaxTree.Nodes.Operators.GreaterThanOrEqualToNode;
import AbstractSyntaxTree.Nodes.Operators.LessThanNode;
import AbstractSyntaxTree.Nodes.Operators.LessThanOrEqualToNode;
import AbstractSyntaxTree.Nodes.Operators.MinusNode;
import AbstractSyntaxTree.Nodes.Operators.MultiplyNode;
import AbstractSyntaxTree.Nodes.Operators.NotEqualNode;
import AbstractSyntaxTree.Nodes.Operators.PlusNode;
```

```
import AbstractSyntaxTree.Nodes.ParameterListNode;
import AbstractSyntaxTree.Nodes.ProgramNode;
import AbstractSyntaxTree.Nodes.ReadNode;
import AbstractSyntaxTree.Nodes.ReturnNode;
import AbstractSyntaxTree.Nodes.StatementListNode;
import AbstractSyntaxTree.Nodes.VariableNode;
import AbstractSyntaxTree.Nodes.WhileNode;
import AbstractSyntaxTree.Nodes.WriteNode;
import AbstractSyntaxTree.TACLine;
import AbstractSyntaxTree.TinyAssemblyGenerator;
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import littlecompiler.SymbolTableContainer;
import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.tree.ErrorNode;
import org.antlr.v4.runtime.tree.ParseTree;
import org.antlr.v4.runtime.tree.TerminalNode;
import symboltables.Symbol;
import symboltables.SymbolTable;
import symboltables.enums.ESymbolAttribute;
import symboltables.enums.ESymbolType;

/**
*/
public class LittleBaseListener implements LittleListener
{
  public final Stack<SymbolTable> symbolTables = new Stack<>();
  private int currentBlockCount = 1;
  private AST ast;

  /**
   */
  @Override public void enterProgram(LittleParser.ProgramContext ctx)
  {
    SymbolTable globalSymbolTable = new SymbolTable("GLOBAL");
    this.symbolTables.push(globalSymbolTable);
    SymbolTableContainer.getInstance().setGlobalTable(globalSymbolTable);
    this.ast = new AST(new ProgramNode());
  }
```

```
/**
 */
@Override public void exitProgram(LittleParser.ProgramContext ctx)
{
   this.ast.pop();
   List<TACLine> linesOfCode = this.ast.generate3AC();
   TinyAssemblyGenerator generator = new TinyAssemblyGenerator();
   generator.assemble((ArrayList<TACLine>)linesOfCode).forEach(line ->
   {
      System.out.println(line);
   });

}

/**
 */
@Override public void enterId(LittleParser.IdContext ctx)
{
}

/**
 */
@Override public void exitId(LittleParser.IdContext ctx)
{
}

/**
 */
@Override public void enterPgm_body(LittleParser.Pgm_bodyContext ctx)
{
}

/**
 */
@Override public void exitPgm_body(LittleParser.Pgm_bodyContext ctx)
{
}

/**
 */
@Override public void enterDecl(LittleParser.DeclContext ctx)
{
}
```

```
/**
 */
@Override public void exitDecl(LittleParser.DeclContext ctx)
{
}

/**
 */
@Override public void enterString_decl(LittleParser.String_declContext ctx)
{
   this.ast.push(new DeclarationNode());

   SymbolTable currentScopeTable = this.symbolTables.peek();

   ParseTree stringNameNode = ctx.children.get(1);
   String stringName = stringNameNode.getText();
   Symbol stringSymbol = new Symbol(
      stringName,
      ESymbolType.VAR,
      ESymbolAttribute.STRING);

   ParseTree stringValueNode = ctx.children.get(3);
   String stringValue = stringValueNode.getText().replace("\"", "");
   stringSymbol.setValue(stringValue);

   currentScopeTable.addSymbol(stringSymbol);
}

/**
 */
@Override public void exitString_decl(LittleParser.String_declContext ctx)
{
   this.ast.pop();
}

/**
 */
@Override public void enterStr(LittleParser.StrContext ctx)
{
}

/**
```

```
 */
@Override public void exitStr(LittleParser.StrContext ctx)
{
}

/**
 */
@Override public void enterVar_decl(LittleParser.Var_declContext ctx)
{
   this.ast.push(new DeclarationNode());

   SymbolTable currentScope = this.symbolTables.peek();

   ParseTree varTypeNode = ctx.children.get(0);
   String varType = varTypeNode.getText();
   ParseTree varNamesNode = ctx.children.get(1);
   String[] varNames = varNamesNode.getText().split(",");
   ESymbolAttribute symbolAttribute = varType
      .equals(ESymbolAttribute.INT.toString())
         ? ESymbolAttribute.INT
         : ESymbolAttribute.FLOAT;

   for (String varName : varNames)
   {

      if (currentScope.getSymbolByName(varName) != null)
      {
         System.out.println("DECLARATION ERROR " + varName);
         System.exit(0);
      }

      currentScope.addSymbol(
         new Symbol(
            varName,
            ESymbolType.VAR,
            symbolAttribute));
   }
}

/**
 */
@Override public void exitVar_decl(LittleParser.Var_declContext ctx)
{
```

```
    this.ast.pop();
}

/**
 */
@Override public void enterVar_type(LittleParser.Var_typeContext ctx)
{
}

 */
@Override public void exitVar_type(LittleParser.Var_typeContext ctx)
{
}

/**
 */
@Override public void enterAny_type(LittleParser.Any_typeContext ctx)
{
}

/**
 */
@Override public void exitAny_type(LittleParser.Any_typeContext ctx)
{
}

/**
 */
@Override public void enterId_list(LittleParser.Id_listContext ctx)
{
    ParseTree firstVariableTree = ctx.children.get(0);
    String firstVariableName = firstVariableTree.getText();
    this.ast.push(new VariableNode(firstVariableName));
    this.ast.pop();

    ParseTree followingVariablesTree = ctx.children.get(1);
    int childCount = followingVariablesTree.getChildCount();
    while (childCount > 1)
    {
        String nextVariableName = followingVariablesTree
            .getChild(1)
            .getText();
        this.ast.push(new VariableNode(nextVariableName));
```

```
            this.ast.pop();
            followingVariablesTree = followingVariablesTree.getChild(2);
            childCount = followingVariablesTree.getChildCount();
        }
    }

    /**
     */
    @Override public void exitId_list(LittleParser.Id_listContext ctx)
    {
    }

    /**
     */
    @Override public void enterId_tail(LittleParser.Id_tailContext ctx)
    {
    }

    /**
     */
    @Override public void exitId_tail(LittleParser.Id_tailContext ctx)
    {
    }

    /**
     */
    @Override public void enterParam_decl_list(LittleParser.Param_decl_listContext ctx)
    {
        this.ast.push(new InputParameterListNode());
    }

    /**
     */
    @Override public void exitParam_decl_list(LittleParser.Param_decl_listContext ctx)
    {
        this.ast.pop();
    }

    /**
     */
    @Override public void enterParam_decl(LittleParser.Param_declContext ctx)
    {
        String parameterType = ctx.children.get(0).getText();
```

```
    String parameterName = ctx.children.get(1).getText();
    this.ast.push(new InputParameterNode(parameterName));

    SymbolTable currentScope = this.symbolTables.peek();


    currentScope.addSymbol(
        new Symbol(
            parameterName,
            ESymbolType.VAR,
            ESymbolAttribute.valueOf(parameterType)));
}

/**
 */
@Override public void exitParam_decl(LittleParser.Param_declContext ctx)
{
    this.ast.pop();
}

/**
 */
@Override public void enterParam_decl_tail(LittleParser.Param_decl_tailContext ctx)
{
}

/**
 */
@Override public void exitParam_decl_tail(LittleParser.Param_decl_tailContext ctx)
{
}

/**
 */
@Override public void enterFunc_declarations(LittleParser.Func_declarationsContext ctx)
{
}

/**
 */
@Override public void exitFunc_declarations(LittleParser.Func_declarationsContext ctx)
{
}
```

```
/**
 */
@Override public void enterFunc_decl(LittleParser.Func_declContext ctx)
{
   String functionName = ctx.getChild(2).getText();
   this.ast.push(new FunctionNode(functionName));

   String returnTypeString = ctx.getChild(1).getText();
   ESymbolAttribute returnType = ESymbolAttribute
      .valueOf(returnTypeString);

   SymbolTable currentScopeTable = this.symbolTables.peek();

   currentScopeTable.addSymbol(
      new Symbol(
         functionName,
         ESymbolType.PROCEDURE,
         returnType));

   SymbolTable childTable = currentScopeTable
      .getChildTableByName(functionName);
   this.symbolTables.push(childTable);
}

/**
 */
@Override public void exitFunc_decl(LittleParser.Func_declContext ctx)
{
   this.ast.pop();

   this.symbolTables.pop();
}

/**
 */
@Override public void enterFunc_body(LittleParser.Func_bodyContext ctx)
{
   this.ast.push(new BeginFunctionNode());
   this.ast.pop();
}

/**
```

```
 */
@Override public void exitFunc_body(LittleParser.Func_bodyContext ctx)
{
   this.ast.push(new EndFunctionNode());
   this.ast.pop();
}

/**
 */
@Override public void enterStmt_list(LittleParser.Stmt_listContext ctx)
{
   this.ast.push(new StatementListNode());
}

/**
 */
@Override public void exitStmt_list(LittleParser.Stmt_listContext ctx)
{
   this.ast.pop();
}

/**
 */
@Override public void enterStmt(LittleParser.StmtContext ctx)
{
}

/**
 */
@Override public void exitStmt(LittleParser.StmtContext ctx)
{
}

/**
 */
@Override public void enterBase_stmt(LittleParser.Base_stmtContext ctx)
{
}

/**
 */
@Override public void exitBase_stmt(LittleParser.Base_stmtContext ctx)
{
```

```
}

/**
 */
@Override public void enterAssign_stmt(LittleParser.Assign_stmtContext ctx)
{
}

/**
 */
@Override public void exitAssign_stmt(LittleParser.Assign_stmtContext ctx)
{
}

/**
 */
@Override public void enterAssign_expr(LittleParser.Assign_exprContext ctx)
{
   this.ast.push(new AssignNode(this.symbolTables.peek()));
   this.ast.push(new VariableNode(ctx.getChild(0).getText()));
   this.ast.pop();
}

/**
 */
@Override public void exitAssign_expr(LittleParser.Assign_exprContext ctx)
{
   this.ast.pop();
}

/**
 */
@Override public void enterRead_stmt(LittleParser.Read_stmtContext ctx)
{
   this.ast.push(new ReadNode(this.symbolTables.peek()));
}

/**
 */
@Override public void exitRead_stmt(LittleParser.Read_stmtContext ctx)
{
   this.ast.pop();
}
```

```
/**
 */
@Override public void enterWrite_stmt(LittleParser.Write_stmtContext ctx)
{
   this.ast.push(new WriteNode(this.symbolTables.peek()));
}

/**
 */
@Override public void exitWrite_stmt(LittleParser.Write_stmtContext ctx)
{
   this.ast.pop();
}

/**
 */
@Override public void enterReturn_stmt(LittleParser.Return_stmtContext ctx)
{
   String containingMethodName = this.symbolTables.peek().getName();
   this.ast.push(new ReturnNode(containingMethodName));
}

/**
 */
@Override public void exitReturn_stmt(LittleParser.Return_stmtContext ctx)
{
   this.ast.pop();
}

/**
 */
@Override public void enterExpr(LittleParser.ExprContext ctx)
{
   boolean expressionIsAnOperation = ctx.getChildCount() == 3;
   if (expressionIsAnOperation)
   {
      String operator = ctx.children.get(1).getText();
      switch (operator)
      {
         case "+":
            this.ast.push(new PlusNode());
            break;
```

```
                case "-":
                    this.ast.push(new MinusNode());
                    break;
                case "*":
                    this.ast.push(new MultiplyNode());
                    break;
                case "/":
                    this.ast.push(new DivideNode());
                    break;
            }
        }
    }

    /**
     */
    @Override public void exitExpr(LittleParser.ExprContext ctx)
    {
        boolean expressionIsAnOperation = ctx.getChildCount() == 3;
        if (expressionIsAnOperation)
        {
            this.ast.pop();
        }
    }

    /**
     */
    @Override public void enterCall_expr(LittleParser.Call_exprContext ctx)
    {
        this.ast.push(new FunctionCallNode(ctx.getText().replace("()", "")));
    }

    /**
     */
    @Override public void exitCall_expr(LittleParser.Call_exprContext ctx)
    {
        this.ast.pop();
    }

    /**
     */
    @Override public void enterExpr_list(LittleParser.Expr_listContext ctx)
    {
        this.ast.push(new ParameterListNode());
```

```
}

/**
 */
@Override public void exitExpr_list(LittleParser.Expr_listContext ctx)
{
   this.ast.pop();
}

/**
 */
@Override public void enterExpr_list_tail(LittleParser.Expr_list_tailContext ctx)
{
}

/**
 */
@Override public void exitExpr_list_tail(LittleParser.Expr_list_tailContext ctx)
{
}

/**
 */
@Override public void enterPrimary(LittleParser.PrimaryContext ctx)
{
   Pattern pattern = Pattern.compile("[0-9]+");
   String primaryText = ctx.getText();
   Matcher matcher = pattern.matcher(primaryText);
   if (matcher.matches())
   {
      this.ast.push(new IntLiteralNode(Integer.parseInt(ctx.getText())));
      return;
   }

   pattern = Pattern.compile("[(].*[)]");
   matcher = pattern.matcher(primaryText);
   if (matcher.matches())
   {
      return;
   }

   pattern = Pattern.compile("[0-9]+[.][0-9]+");
   matcher = pattern.matcher(ctx.getText());
```

```
    if (matcher.matches())
      this.ast.push(
        new FloatLiteralNode(
          Float.parseFloat(ctx.getText())));
    else
      this.ast.push(new VariableNode(ctx.getText()));
  }

  /**
   */
  @Override public void exitPrimary(LittleParser.PrimaryContext ctx)
  {
    String primaryText = ctx.getText();
    Pattern pattern = Pattern.compile("[(].*[)]");
    Matcher matcher = pattern.matcher(primaryText);
    if (matcher.matches())
    {
      return;
    }

    this.ast.pop();
  }

  /**
   */
  @Override public void enterAddop(LittleParser.AddopContext ctx)
  {
  }

  /**
   */
  @Override public void exitAddop(LittleParser.AddopContext ctx)
  {
  }

  /**
   */
  @Override public void enterMulop(LittleParser.MulopContext ctx)
  {
  }

  /**
   */
```

```
@Override public void exitMulop(LittleParser.MulopContext ctx)
{
}

/**
 */
@Override public void enterIf_stmt(LittleParser.If_stmtContext ctx)
{
   this.ast.push(new IfNode());

   String blockScopeName = "BLOCK " + currentBlockCount++;

   SymbolTable currentScope = this.symbolTables.peek();
   SymbolTable newScope = new SymbolTable(blockScopeName);
   currentScope.addChildTable(newScope);
   this.symbolTables.push(newScope);
}

/**
 */
@Override public void exitIf_stmt(LittleParser.If_stmtContext ctx)
{
   this.ast.pop();
   this.symbolTables.pop();
}

/**
 */
@Override public void enterElse_part(LittleParser.Else_partContext ctx)
{
   this.ast.push(new ElseNode());

   String blockScopeName = "BLOCK " + currentBlockCount++;

   SymbolTable currentScope = this.symbolTables.peek();
   SymbolTable newScope = new SymbolTable(blockScopeName);
   currentScope.addChildTable(newScope);
   this.symbolTables.push(newScope);
}

/**
 */
@Override public void exitElse_part(LittleParser.Else_partContext ctx)
```

```
{
   this.ast.pop();
   this.symbolTables.pop();
}

/**
 */
@Override public void enterCond(LittleParser.CondContext ctx)
{
   this.ast.push(new ConditionNode());
   String operator = ctx.children.get(1).getText();

   switch (operator)
   {
      case ">":
         this.ast.push(new GreaterThanNode(this.symbolTables.peek()));
         break;
      case ">=":
         this.ast.push(
            new GreaterThanOrEqualToNode(this.symbolTables.peek()));
         break;
      case "<":
         this.ast.push(new LessThanNode(this.symbolTables.peek()));
         break;
      case "<=":
         this.ast.push(
            new LessThanOrEqualToNode(this.symbolTables.peek()));
         break;
      case "=":
         this.ast.push(new EqualNode(this.symbolTables.peek()));
         break;
      case "!=":
         this.ast.push(new NotEqualNode(this.symbolTables.peek()));
         break;
      default:
         break;
   }
}

/**
 */
@Override public void exitCond(LittleParser.CondContext ctx)
{
```

```
    this.ast.pop();
    this.ast.pop();
}

/**
 */
@Override public void enterCompop(LittleParser.CompopContext ctx)
{
}

/**
 */
@Override public void exitCompop(LittleParser.CompopContext ctx)
{
}

/**
 */
@Override public void enterWhile_stmt(LittleParser.While_stmtContext ctx)
{
    this.ast.push(new WhileNode());

    String blockScopeName = "BLOCK " + currentBlockCount++;

    SymbolTable currentScope = this.symbolTables.peek();
    SymbolTable newScope = new SymbolTable(blockScopeName);
    currentScope.addChildTable(newScope);
    this.symbolTables.push(newScope);
}

/**
 */
@Override public void exitWhile_stmt(LittleParser.While_stmtContext ctx)
{
    this.ast.pop();
}

/**
 */
@Override public void enterEveryRule(ParserRuleContext ctx)
{
}
```

```
/**
 */
@Override public void exitEveryRule(ParserRuleContext ctx)
{
}

/**
 */
@Override public void visitTerminal(TerminalNode node)
{
}

/**
 */
@Override public void visitErrorNode(ErrorNode node)
{
}

@Override
public void enterAdd_minus_expression(LittleParser.Add_minus_expressionContext ctx)
{
   boolean nodeIsAddop = ctx.getChildCount() == 3;
   if (nodeIsAddop)
   {
      String operator = ctx.children.get(1).getText();
      this.ast.push(
         operator.equals("+")
            ? new PlusNode()
            : new MinusNode());
   }
}

@Override
public void exitAdd_minus_expression(LittleParser.Add_minus_expressionContext ctx)
{
   boolean nodeIsAddop = ctx.getChildCount() == 3;
   if (nodeIsAddop)
      this.ast.pop();
}

@Override
public void
enterMultiply_divide_expression(LittleParser.Multiply_divide_expressionContext ctx)
```

```
    {
        boolean nodeIsMulop = ctx.getChildCount() == 3;
        if (nodeIsMulop)
        {
            String operator = ctx.children.get(1).getText();
            this.ast.push(
                operator.equals("*")
                    ? new MultiplyNode()
                    : new DivideNode());
        }
    }

    @Override
    public void
exitMultiply_divide_expression(LittleParser.Multiply_divide_expressionContext ctx)
    {
        boolean nodeIsMulop = ctx.getChildCount() == 3;
        if (nodeIsMulop)
            this.ast.pop();
    }

    @Override
    public void enterExpression_component(LittleParser.Expression_componentContext ctx)
    {
    }

    @Override
    public void exitExpression_component(LittleParser.Expression_componentContext ctx)
    {
    }

    @Override
    public void enterStmt_list_tail(LittleParser.Stmt_list_tailContext ctx)
    {
    }

    @Override
    public void exitStmt_list_tail(LittleParser.Stmt_list_tailContext ctx)
    {
    }
}
```

```
/*
```

```
*/
package AbstractSyntaxTree;

import AbstractSyntaxTree.Nodes.ASTNode;
import java.util.List;
import java.util.Stack;
import symboltables.enums.ESymbolAttribute;

public class AST
{
  private final ASTNode root;
  private final Stack<ASTNode> currentAstBranch;

  public AST(ASTNode root)
  {
    this.root = root;
    this.currentAstBranch = new Stack<>();
    this.currentAstBranch.push(root);
  }

  public ASTNode getCurrentNode()
  {
    return this.currentAstBranch.peek();
  }

  public void push(ASTNode newCurrentNode)
  {
    this.currentAstBranch.peek().addChild(newCurrentNode);
    this.currentAstBranch.push(newCurrentNode);
  }

  public ASTNode pop()
  {
    return this.currentAstBranch.pop();
  }

  public List<TACLine> generate3AC()
  {
    return root.generate3AC();
  }
}
```

```
/*
```

```
*/
package AbstractSyntaxTree;

import java.util.ArrayList;

public class TACLine
{
   private final ArrayList<String> elements;

   public TACLine()
   {
      this.elements = new ArrayList<>();
   }

   public ArrayList<String> getElements()
   {
      return elements;
   }

   public void addElement(String element)
   {
      this.elements.add(element);
   }

   public String getLineText()
   {
      String lineText = "";

      for (String element : elements)
      {
         lineText = lineText.concat(element.concat(" "));
      }

      return lineText;
   }
}
```

```
/*
*/
package AbstractSyntaxTree;

import java.util.HashMap;
import java.util.List;
```

```
public class FunctionCodeContainer
{
    private static FunctionCodeContainer instance;
    private HashMap<String, List<TACLine>> functionCodes;

    public static FunctionCodeContainer getInstance()
    {
        return instance == null
            ? instance = new FunctionCodeContainer()
            : instance;
    }

    private FunctionCodeContainer()
    {
        this.functionCodes = new HashMap<>();
    }

    public void addFunctionCode(
        String functionName,
        List<TACLine> linesOfCode)
    {
        this.functionCodes.put(
            functionName,
            linesOfCode);
    }

    public List<TACLine> getFunctionCode(String functionName)
    {
        return this.functionCodes.get(functionName);
    }
}
```

```
/*
*/
package AbstractSyntaxTree;

/**
 * Singleton for keeping track of labels for If and While nodes.
 */
public class Labels {
    private static Labels instance;
    private int a = 0;
```

```
    static {
        instance = new Labels();
    }

    public String getLabel() {
        a++;
        return "label".concat(Integer.toString(a));
    }

    public static Labels getInstance() {
        return instance;
    }
}
```

```
/*
*/
package AbstractSyntaxTree;

public class ParameterRegisterHandler
{
    private static final int RETURN_VALUE_REGISTER_INDEX = 900;
    private static final int FIRST_PARAMETER_REGISTER_INDEX = 901;

    private static ParameterRegisterHandler instance;
    private int nextIndex;

    public static ParameterRegisterHandler getInstance()
    {
        return instance == null
            ? instance = new ParameterRegisterHandler()
            : instance;
    }

    private ParameterRegisterHandler()
    {
        this.nextIndex = FIRST_PARAMETER_REGISTER_INDEX;
    }

    public String getNextRegister()
    {
        return String.format(
            "$T%d",
```

```
        nextIndex++);
    }

    public void resetParameterIndex()
    {
        this.nextIndex = FIRST_PARAMETER_REGISTER_INDEX;
    }

    public String getReturnRegister()
    {
        return "$T" + RETURN_VALUE_REGISTER_INDEX;
    }
}
```

```
/*
*/
package AbstractSyntaxTree;

import java.util.LinkedHashMap;

/*
 * Singleton for keeping track of temporary register numbers
 */
public class TempararyRegisters {
    private static TempararyRegisters instance;
    private int a = 0;
    private LinkedHashMap<String, String> lHM = new LinkedHashMap<>();

    static {
        instance = new TempararyRegisters();
    }

    public String getTempReg(String key) {
        a++;
        String value = "$T".concat(Integer.toString(a));
        lHM.put(key, value);
        return value;
    }

    public String getTempReg() {
        a++;
        String value = "$T".concat(Integer.toString(a));
        return value;
```

```
    }

    public String checkTempReg(String varName) {
        if (lHM.containsKey(varName)) {
            return lHM.get(varName);
        }
        return getTempReg(varName);
    }

    public static TempararyRegisters getInstance() { return instance; }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;
import java.util.List;
import symboltables.SymbolTable;
import symboltables.enums.ESymbolAttribute;

public class AssignNode extends ASTNode
{
    protected final static int VARIABLE_INDEX = 0;
    protected final static int EXPRESSION_INDEX = 1;

    private final SymbolTable scopeTable;

    public AssignNode(SymbolTable table)
    {
        this.scopeTable = table;
    }

    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> completeAssignTAC = new ArrayList<>();
        TACLine tac1 = new TACLine();
        TACLine tac2 = new TACLine();
```

```
ASTNode rhs = this.children.get(EXPRESSION_INDEX);
String variableName = ((VariableNode)this.children.get(VARIABLE_INDEX))
   .getVariableName();

if (rhs instanceof IntLiteralNode)
{
   String tempRegister = TempararyRegisters
      .getInstance()
      .checkTempReg(variableName);
   Integer value = ((IntLiteralNode) rhs).getLiteralValue();
   tac1.addElement("STOREI");
   tac1.addElement(value.toString());
   tac1.addElement(tempRegister);
   tac2.addElement("STOREI");
   tac2.addElement(tempRegister);
   tac2.addElement(variableName);
   completeAssignTAC.add(tac1);
   completeAssignTAC.add(tac2);
}
else if (rhs instanceof FloatLiteralNode)
{
   String tempRegister = TempararyRegisters
      .getInstance()
      .checkTempReg(variableName);
   Float value = ((FloatLiteralNode) rhs).getLiteralValue();
   tac1.addElement("STOREF");
   tac1.addElement(value.toString());
   tac1.addElement(tempRegister);
   tac2.addElement("STOREF");
   tac2.addElement(tempRegister);
   tac2.addElement(variableName);
   completeAssignTAC.add(tac1);
   completeAssignTAC.add(tac2);
}
else if (rhs instanceof VariableNode)
{
   VariableNode node = (VariableNode) rhs;
   String variable = node.getVariableName();
   ESymbolAttribute type = this.scopeTable
      .getSymbolByName(variable)
      .getAttribute();
   String variableType = type == ESymbolAttribute.INT
      ? "I"
```

```
          : "F";
        TACLine tac = new TACLine();
        tac.addElement("STORE".concat(variableType));
        tac.addElement(variable);
        tac.addElement(variableName);
        completeAssignTAC.add(tac);
      }
      else
      {
        List<TACLine> expressionCode = rhs.generate3AC();
        ESymbolAttribute expressionType = this.getChildResultType(
          expressionCode);
        String expressionResultRegister = this.getChildResultRegister(
          expressionCode);

        String storeType = expressionType == ESymbolAttribute.INT
          ? "STOREI"
          : "STOREF";

        tac2.addElement(storeType);
        tac2.addElement(expressionResultRegister);
        tac2.addElement(variableName);

        completeAssignTAC.addAll(expressionCode);
        completeAssignTAC.add(tac2);
      }


      return completeAssignTAC;
    }
  }
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;
import symboltables.enums.ESymbolAttribute;


public abstract class ASTNode
```

```
{
    protected List<ASTNode> children;

    public ASTNode()
    {
        this.children = new ArrayList<>();
    }

    public void addChild(ASTNode childNode)
    {
        this.children.add(childNode);
    }

    protected String getChildResultRegister(List<TACLine> childCode)
    {
        List<String> lastLineElements = childCode
            .get(childCode.size() - 1)
            .getElements();
        return lastLineElements.get(lastLineElements.size() - 1);
    }

    protected ESymbolAttribute getChildResultType(List<TACLine> childCode)
    {
        String finalStatement = childCode
            .get(childCode.size() - 1).getElements().get(0);

        return finalStatement.charAt(finalStatement.length() - 1) == 'I'
            ? ESymbolAttribute.INT
            : ESymbolAttribute.FLOAT;
    }

    public abstract List<TACLine> generate3AC();
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;

public class BeginFunctionNode extends ASTNode
```

```
{
   @Override
   public List<TACLine> generate3AC()
   {
      List<TACLine> code = new ArrayList<>();

      TACLine linkLine = new TACLine();
      linkLine.addElement("LINK");
      code.add(linkLine);

      return code;
   }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.List;

public class ConditionNode extends ASTNode
{
   private static final int BOOLEAN_OPERATOR_INDEX = 0;

   @Override
   public List<TACLine> generate3AC()
   {
      return this.children.get(BOOLEAN_OPERATOR_INDEX).generate3AC();
   }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;

public class DeclarationNode extends ASTNode
{
   protected final static int INT_VALUE_INDEX = 0;
```

```java
    protected final static int VAR_VALUE_INDEX = 1;

    // This shouldn't require code generation (I don't think).
    @Override
    public List<TACLine> generate3AC()
    {
        return new ArrayList<>();
    }
}
```

```java
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.List;

public class ElseNode extends ASTNode
{
    //protected final static int CONDITION_INDEX = 0;
    protected final static int STATEMENT_LIST_INDEX = 0;

    @Override
    public List<TACLine> generate3AC()
    {
        return this.children.get(STATEMENT_LIST_INDEX).generate3AC();
    }
}
```

```java
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;

public class EndFunctionNode extends ASTNode
{
    @Override
    public List<TACLine> generate3AC()
    {
```

```
        List<TACLine> code = new ArrayList<>();

        TACLine linkLine = new TACLine();
        linkLine.addElement("RET");
        code.add(linkLine);

        return code;
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.List;

public class FloatLiteralNode extends ASTNode
{
    private final float literalValue;

    public FloatLiteralNode(float value)
    {
        this.literalValue = value;
    }

    public float getLiteralValue()
    {
        return literalValue;
    }

    @Override
    public List<TACLine> generate3AC()
    {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
```

```
import java.util.ArrayList;
import java.util.List;

public class FunctionCallNode extends ASTNode
{
   private final String functionName;

   public FunctionCallNode(String functionName)
   {
      this.functionName = functionName;
   }

   public String getFunctionName()
   {
      return this.functionName;
   }

   @Override
   public List<TACLine> generate3AC()
   {
      List<TACLine> linesEndingWithFunctionResultStore = new ArrayList<>();

      this.children.forEach(child ->
         linesEndingWithFunctionResultStore
            .addAll(child.generate3AC()));

      return linesEndingWithFunctionResultStore;
   }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.FunctionCodeContainer;
import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;

public class FunctionNode extends ASTNode
{
   private static final int INPUT_NODE_INDEX = 0;
   private static final int STATEMENT_LIST_INDEX = 2;
```

```
    private final String functionName;

    public FunctionNode(String functionName)
    {
        this.functionName = functionName;
    }

    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> methodLines = new ArrayList<>();
        TACLine methodLabel = new TACLine();
        methodLabel.addElement("LABEL");
        methodLabel.addElement(this.functionName);
        methodLines.add(methodLabel);

        this.children.forEach(child -> methodLines.addAll(child.generate3AC()));

        return methodLines;
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.Labels;
import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;
import java.util.List;

public class IfNode extends ASTNode
{
    protected final static int CONDITION_INDEX = 0;
    protected final static int STATEMENT_LIST_INDEX = 1;
    protected final static int ELSE_NODE_INDEX = 2;

    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> completeIfTAC = new ArrayList<>();
```

```
        /* Add jump line */
        Labels label = Labels.getInstance();
        String elseLabel = label.getLabel();
        List<TACLine> conditionCode = this.children
            .get(CONDITION_INDEX)
            .generate3AC();
        conditionCode.get(conditionCode.size() - 1).addElement(elseLabel);
        completeIfTAC.addAll(conditionCode);

        /* Add statement list */
        ASTNode statementListNode = this.children.get(STATEMENT_LIST_INDEX);
        completeIfTAC.addAll(statementListNode.generate3AC());

        /* Add jump-out to end of statement list */
        TACLine jumpOutLine = new TACLine();
        jumpOutLine.addElement("JUMP");
        String outLabel = label.getLabel();
        jumpOutLine.addElement(outLabel);
        completeIfTAC.add(jumpOutLine);

        /* Add else block */
        TACLine elseLabelLine = new TACLine();
        elseLabelLine.addElement("LABEL");
        elseLabelLine.addElement(elseLabel);
        completeIfTAC.add(elseLabelLine);

        if (this.children.size() > 2)
        {
            ASTNode elseNode = this.children.get(ELSE_NODE_INDEX);
            completeIfTAC.addAll(elseNode.generate3AC());
        }

        /* Add jump-out label */
        TACLine jumpOutLabel = new TACLine();
        jumpOutLabel.addElement("LABEL");
        jumpOutLabel.addElement(outLabel);
        completeIfTAC.add(jumpOutLabel);

        return completeIfTAC;
    }
}
```

```java
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;

public class InputParameterListNode extends ASTNode
{
    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> parameterLoadingCode = new ArrayList<>();
        this.children.forEach(node ->
            parameterLoadingCode
                .addAll(node.generate3AC()));

        return parameterLoadingCode;
    }
}
```

```java
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.ParameterRegisterHandler;
import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;

public class InputParameterNode extends ASTNode
{
    private final String parameterName;

    public InputParameterNode(String parameterName)
    {
        this.parameterName = parameterName;
    }

    public String getParameterName()
    {
        return parameterName;
```

```
    }

    @Override
    public List<TACLine> generate3AC()
    {
        // Use singleton class to load in variables.
        List<TACLine> variableLoadingCode = new ArrayList<>();

        TACLine newLine = new TACLine();
        newLine.addElement("LOAD");
        newLine.addElement(
            ParameterRegisterHandler.getInstance().getNextRegister());
        newLine.addElement(this.parameterName);

        variableLoadingCode.add(newLine);

        return variableLoadingCode;
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.List;

public class IntLiteralNode extends ASTNode
{
    private final int literalValue;

    public IntLiteralNode(int literalValue)
    {
        this.literalValue = literalValue;
    }

    public int getLiteralValue()
    {
        return literalValue;
    }

    @Override
    public List<TACLine> generate3AC()
```

```
    {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.ParameterRegisterHandler;
import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;

public class ParameterListNode extends ASTNode
{
    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> parameterStoringCode = new ArrayList<>();

        this.children.forEach(child ->
        {
            TACLine line = new TACLine();
            parameterStoringCode.add(line);
            line.addElement("STORE");
            if (child instanceof IntLiteralNode)
            {
                Integer value = ((IntLiteralNode) child).getLiteralValue();
                line.addElement(value.toString());
            }
            else if (child instanceof FloatLiteralNode)
            {
                Float value = ((FloatLiteralNode) child).getLiteralValue();
                line.addElement(value.toString());
            }
            line.addElement(
                ParameterRegisterHandler
                    .getInstance()
                    .getNextRegister());
        });
        ParameterRegisterHandler.getInstance().resetParameterIndex();
```

```
        return parameterStoringCode;
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;

public class ProgramNode extends ASTNode
{
    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> programCode = new ArrayList<>();

        for (ASTNode child : children)
        {
            programCode.addAll(child.generate3AC());
        }

        return programCode;
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;
import symboltables.SymbolTable;
import symboltables.enums.ESymbolAttribute;

public class ReadNode extends ASTNode
{
    protected final static int READ_INDEX = 0;

    private final SymbolTable scopeTable;
```

```
    public ReadNode(SymbolTable table)
    {
        this.scopeTable = table;
    }

    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> completeReadTAC = new ArrayList<>();
        TACLine tac = new TACLine();
        ASTNode read = this.children.get(READ_INDEX);

        String variableName = ((VariableNode) read).getVariableName();
        ESymbolAttribute type = this.scopeTable
            .getSymbolByName(variableName)
            .getAttribute();
        String variableType = type == ESymbolAttribute.INT
            ? "I"
            : "F";

        tac.addElement("READ".concat(variableType));
        tac.addElement(variableName);

        completeReadTAC.add(tac);
        return completeReadTAC;
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.ParameterRegisterHandler;
import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;
import littlecompiler.SymbolTableContainer;
import symboltables.enums.ESymbolAttribute;

public class ReturnNode extends ASTNode
{
    protected final static int RETURN_EXPRESSION_INDEX = 0;
```

```
private final String methodName;

public ReturnNode(String methodName)
{
   this.methodName = methodName;
}

@Override
public List<TACLine> generate3AC()
{
   List<TACLine> lines = new ArrayList<>();

   TACLine returnRegisterStoreLine = new TACLine();
   lines.add(returnRegisterStoreLine);

   ASTNode node = this.children.get(RETURN_EXPRESSION_INDEX);
   if (node instanceof VariableNode)
   {

   }
   else
   {
      List<TACLine> tac = node.generate3AC();

      ESymbolAttribute type = this.getChildResultType(tac);
      if (type == ESymbolAttribute.INT)
         returnRegisterStoreLine.addElement("STOREI");
      else
         returnRegisterStoreLine.addElement("STOREF");

      String registerToReturn = this.getChildResultRegister(tac);
      returnRegisterStoreLine.addElement(registerToReturn);
   }

   returnRegisterStoreLine
      .addElement(
         ParameterRegisterHandler
            .getInstance()
            .getReturnRegister());

   return lines;
}
```

```
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;

public class StatementListNode extends ASTNode
{
    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> statements = new ArrayList<>();

        children.forEach(child ->
        {
            statements.addAll(child.generate3AC());
        });

        return statements;
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.List;

public class StringLiteralNode extends ASTNode
{
    private final String literalValue;

    public StringLiteralNode(String literalValue)
    {
        this.literalValue = literalValue;
    }

    public String getLiteralValue()
```

```
    {
        return literalValue;
    }

    @Override
    public List<TACLine> generate3AC()
    {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.List;

public class VariableNode extends ASTNode
{
    private final String variableName;

    public VariableNode(String variableName)
    {
        this.variableName = variableName;
    }

    public String getVariableName()
    {
        return this.variableName;
    }

    @Override
    public List<TACLine> generate3AC()
    {
        throw new UnsupportedOperationException("Not supported yet.");
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;
```

```
import AbstractSyntaxTree.Labels;
import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;

public class WhileNode extends ASTNode
{
  protected final static int CONDITION_INDEX = 0;
  protected final static int STATEMENT_LIST_INDEX = 1;

  @Override
  public List<TACLine> generate3AC()
  {
    List<TACLine> completeWhileTAC = new ArrayList<>();

    Labels label = Labels.getInstance();
    String whileLabel = label.getLabel();
    String exitLabel = label.getLabel();

    TACLine whileLabelLine = new TACLine();
    whileLabelLine.addElement("LABEL");
    whileLabelLine.addElement(whileLabel);
    completeWhileTAC.add(whileLabelLine);

    List<TACLine> conditionCode = this.children
      .get(CONDITION_INDEX)
      .generate3AC();
    conditionCode.get(conditionCode.size() - 1).addElement(exitLabel);
    completeWhileTAC.addAll(conditionCode);

    /* Statement list */
    completeWhileTAC.addAll(
      this.children
        .get(STATEMENT_LIST_INDEX)
        .generate3AC());

    TACLine jumpUpLine = new TACLine();
    jumpUpLine.addElement("JUMP");
    jumpUpLine.addElement(whileLabel);
    completeWhileTAC.add(jumpUpLine);

    TACLine jumpOutLabel = new TACLine();
    jumpOutLabel.addElement("LABEL");
```

```
        jumpOutLabel.addElement(exitLabel);
        completeWhileTAC.add(jumpOutLabel);

        return completeWhileTAC;
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;
import java.util.ArrayList;
import java.util.List;
import symboltables.SymbolTable;
import symboltables.enums.ESymbolAttribute;

public class WriteNode extends ASTNode
{
    private final SymbolTable scopeTable;

    public WriteNode(SymbolTable table)
    {
        this.scopeTable = table;
    }

    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> completeWriteTAC = new ArrayList<>();

        this.children.forEach(child ->
        {
            VariableNode node = (VariableNode) child;
            String variableName = node.getVariableName();
            ESymbolAttribute type = this.scopeTable
                .getSymbolByName(variableName)
                .getAttribute();
            String variableType = type == ESymbolAttribute.INT
                ? "I"
                : "F";
            TACLine tac = new TACLine();
            tac.addElement("WRITE".concat(variableType));
```

```java
            tac.addElement(variableName);
            completeWriteTAC.add(tac);
        });

        return completeWriteTAC;
    }
}
```

```java
/*
*/
package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;
import AbstractSyntaxTree.Nodes.FloatLiteralNode;
import AbstractSyntaxTree.Nodes.IntLiteralNode;
import static AbstractSyntaxTree.Nodes.Operators.PlusNode.LEFT_OPERAND_INDEX;
import AbstractSyntaxTree.Nodes.VariableNode;
import AbstractSyntaxTree.TACLine;
import AbstractSyntaxTree.TempararyRegisters;
import java.util.ArrayList;
import java.util.List;

public class DivideNode extends ASTNode
{
    protected final static int LEFT_OPERAND_INDEX = 0;
    protected final static int RIGHT_OPERAND_INDEX = 1;

    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> completeAddTAC = new ArrayList<>();
        TACLine tac = new TACLine();
        ASTNode left = this.children.get(LEFT_OPERAND_INDEX);
        ASTNode right = this.children.get(RIGHT_OPERAND_INDEX);
        if (left instanceof IntLiteralNode || right instanceof IntLiteralNode){
            tac.addElement("DIVI");
        }
        else {
            tac.addElement("DIVF");
        }

        String leftValue = null;
```

```
if (left instanceof IntLiteralNode)
{
   leftValue = String.valueOf(
      ((FloatLiteralNode) left).getLiteralValue());
   TACLine storeRightValue = new TACLine();
   storeRightValue.addElement("STOREI");
   storeRightValue.addElement(leftValue);
   leftValue = TempararyRegisters.getInstance().getTempReg();
   storeRightValue.addElement(leftValue);
   completeAddTAC.add(storeRightValue);
}
else if (left instanceof FloatLiteralNode)
{
   leftValue = String.valueOf(
      ((FloatLiteralNode) left).getLiteralValue());
   TACLine storeRightValue = new TACLine();
   storeRightValue.addElement("STOREF");
   storeRightValue.addElement(leftValue);
   leftValue = TempararyRegisters.getInstance().getTempReg();
   storeRightValue.addElement(leftValue);
   completeAddTAC.add(storeRightValue);
}
else if (left instanceof VariableNode)
{
   leftValue = ((VariableNode) left).getVariableName();
}
else
{
   List<TACLine> leftExpressionCode = left.generate3AC();
   leftValue = this.getChildResultRegister(leftExpressionCode);
   completeAddTAC.addAll(leftExpressionCode);
}

String rightValue = null;

if (right instanceof IntLiteralNode)
{
   rightValue = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeRightValue = new TACLine();
   storeRightValue.addElement("STOREI");
   storeRightValue.addElement(rightValue);
   rightValue = TempararyRegisters.getInstance().getTempReg();
```

```
            storeRightValue.addElement(rightValue);
            completeAddTAC.add(storeRightValue);
        }
        else if (right instanceof FloatLiteralNode)
        {
            rightValue = String.valueOf(
                ((FloatLiteralNode) right).getLiteralValue());
            TACLine storeRightValue = new TACLine();
            storeRightValue.addElement("STOREF");
            storeRightValue.addElement(rightValue);
            rightValue = TempararyRegisters.getInstance().getTempReg();
            storeRightValue.addElement(rightValue);
            completeAddTAC.add(storeRightValue);
        }
        else if (right instanceof VariableNode)
        {
            rightValue = ((VariableNode) right).getVariableName();
        }
        else
        {
            List<TACLine> rightExpressionCode = right.generate3AC();
            rightValue = this.getChildResultRegister(rightExpressionCode);
            completeAddTAC.addAll(rightExpressionCode);
        }

        tac.addElement(leftValue);
        tac.addElement(rightValue);
        tac.addElement(TempararyRegisters.getInstance().getTempReg());

        completeAddTAC.add(tac);
        return completeAddTAC;
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;
import AbstractSyntaxTree.Nodes.FloatLiteralNode;
import AbstractSyntaxTree.Nodes.IntLiteralNode;
import AbstractSyntaxTree.Nodes.VariableNode;
import AbstractSyntaxTree.TACLine;
```

```
import AbstractSyntaxTree.TempararyRegisters;
import java.util.ArrayList;
import java.util.List;
import symboltables.SymbolTable;
import symboltables.enums.ESymbolAttribute;

public class EqualNode extends ASTNode
{
   protected final static int LEFT_OPERAND_INDEX = 0;
   protected final static int RIGHT_OPERAND_INDEX = 1;

   private final SymbolTable scopeTable;

   public EqualNode(SymbolTable table)
   {
      this.scopeTable = table;
   }

   @Override
   public List<TACLine> generate3AC()
   {
      List<TACLine> completeAddTAC = new ArrayList<>();
      ASTNode left = this.children.get(LEFT_OPERAND_INDEX);
      ASTNode right = this.children.get(RIGHT_OPERAND_INDEX);

      String variableType = "F";
      String leftValue = null;

      if (left instanceof IntLiteralNode)
      {
         String literal = String.valueOf(
            ((IntLiteralNode) right).getLiteralValue());
         TACLine storeLiteralLine = new TACLine();
         storeLiteralLine.addElement("STOREI");
         storeLiteralLine.addElement(literal);
         leftValue = TempararyRegisters
            .getInstance()
            .getTempReg();
         storeLiteralLine.addElement(leftValue);
         completeAddTAC.add(storeLiteralLine);
      }
      else if (left instanceof FloatLiteralNode)
      {
```

```
    String literal = String.valueOf(
        ((IntLiteralNode) right).getLiteralValue());
    TACLine storeLiteralLine = new TACLine();
    storeLiteralLine.addElement("STOREF");
    storeLiteralLine.addElement(literal);
    leftValue = TempararyRegisters
        .getInstance()
        .getTempReg();
    storeLiteralLine.addElement(leftValue);
    completeAddTAC.add(storeLiteralLine);
}
else if (left instanceof VariableNode)
{
    leftValue = ((VariableNode) left).getVariableName();
}
else
{
    List<TACLine> leftExpressionCode = left.generate3AC();
    leftValue = this.getChildResultRegister(leftExpressionCode);
    completeAddTAC.addAll(leftExpressionCode);
}

String rightValue = null;

if (right instanceof IntLiteralNode)
{
    String literal = String.valueOf(
        ((IntLiteralNode) right).getLiteralValue());
    TACLine storeLiteralLine = new TACLine();
    storeLiteralLine.addElement("STOREI");
    storeLiteralLine.addElement(literal);
    rightValue = TempararyRegisters
        .getInstance()
        .getTempReg();
    storeLiteralLine.addElement(rightValue);
    completeAddTAC.add(storeLiteralLine);
    variableType = "I";
}
else if (right instanceof FloatLiteralNode)
{
    String rightLiteral = String.valueOf(
        ((IntLiteralNode) right).getLiteralValue());
    TACLine storeLiteralLine = new TACLine();
```

```
        storeLiteralLine.addElement("STOREF");
        storeLiteralLine.addElement(rightLiteral);
        rightValue = TempararyRegisters
          .getInstance()
          .getTempReg();
        storeLiteralLine.addElement(rightValue);
        completeAddTAC.add(storeLiteralLine);
      }
      else if (right instanceof VariableNode)
      {
        rightValue = ((VariableNode) right).getVariableName();
        ESymbolAttribute type = this.scopeTable
          .getSymbolByName(rightValue)
          .getAttribute();
        variableType = type == ESymbolAttribute.INT
          ? "I"
          : "F";
      }
      else
      {
        List<TACLine> rightExpressionCode = right.generate3AC();
        rightValue = this.getChildResultRegister(rightExpressionCode);
        completeAddTAC.addAll(rightExpressionCode);
        ESymbolAttribute expressionType = this.getChildResultType(
          rightExpressionCode);
        variableType = expressionType == ESymbolAttribute.INT
          ? "I"
          : "F";
      }

      TACLine jumpLine = new TACLine();
      jumpLine.addElement("NE".concat(variableType));
      jumpLine.addElement(leftValue);
      jumpLine.addElement(rightValue);
      completeAddTAC.add(jumpLine);
      /* Add label in caller */

      return completeAddTAC;
    }
}
```

```
/*
*/
```

```
package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;
import AbstractSyntaxTree.Nodes.FloatLiteralNode;
import AbstractSyntaxTree.Nodes.IntLiteralNode;
import AbstractSyntaxTree.Nodes.VariableNode;
import AbstractSyntaxTree.TACLine;
import AbstractSyntaxTree.TempararyRegisters;
import java.util.ArrayList;
import java.util.List;
import symboltables.SymbolTable;
import symboltables.enums.ESymbolAttribute;

public class GreaterThanNode extends ASTNode
{
   protected final static int LEFT_OPERAND_INDEX = 0;
   protected final static int RIGHT_OPERAND_INDEX = 1;

   private final SymbolTable scopeTable;

   public GreaterThanNode(SymbolTable table)
   {
      this.scopeTable = table;
   }

   @Override
   public List<TACLine> generate3AC()
   {
      List<TACLine> completeAddTAC = new ArrayList<>();
      ASTNode left = this.children.get(LEFT_OPERAND_INDEX);
      ASTNode right = this.children.get(RIGHT_OPERAND_INDEX);

      String variableType = "F";
      String leftValue = null;

      if (left instanceof IntLiteralNode)
      {
         String literal = String.valueOf(
            ((IntLiteralNode) right).getLiteralValue());
         TACLine storeLiteralLine = new TACLine();
         storeLiteralLine.addElement("STOREI");
         storeLiteralLine.addElement(literal);
         leftValue = TempararyRegisters
```

```
        .getInstance()
        .getTempReg();
    storeLiteralLine.addElement(leftValue);
    completeAddTAC.add(storeLiteralLine);
}
else if (left instanceof FloatLiteralNode)
{
    String literal = String.valueOf(
        ((IntLiteralNode) right).getLiteralValue());
    TACLine storeLiteralLine = new TACLine();
    storeLiteralLine.addElement("STOREF");
    storeLiteralLine.addElement(literal);
    leftValue = TempararyRegisters
        .getInstance()
        .getTempReg();
    storeLiteralLine.addElement(leftValue);
    completeAddTAC.add(storeLiteralLine);
}
else if (left instanceof VariableNode)
{
    leftValue = ((VariableNode) left).getVariableName();
}
else
{
    List<TACLine> leftExpressionCode = left.generate3AC();
    leftValue = this.getChildResultRegister(leftExpressionCode);
    completeAddTAC.addAll(leftExpressionCode);
}

String rightValue = null;

if (right instanceof IntLiteralNode)
{
    String literal = String.valueOf(
        ((IntLiteralNode) right).getLiteralValue());
    TACLine storeLiteralLine = new TACLine();
    storeLiteralLine.addElement("STOREI");
    storeLiteralLine.addElement(literal);
    rightValue = TempararyRegisters
        .getInstance()
        .getTempReg();
    storeLiteralLine.addElement(rightValue);
    completeAddTAC.add(storeLiteralLine);
```

```
    variableType = "I";
}
else if (right instanceof FloatLiteralNode)
{
    String rightLiteral = String.valueOf(
        ((IntLiteralNode) right).getLiteralValue());
    TACLine storeLiteralLine = new TACLine();
    storeLiteralLine.addElement("STOREF");
    storeLiteralLine.addElement(rightLiteral);
    rightValue = TempararyRegisters
        .getInstance()
        .getTempReg();
    storeLiteralLine.addElement(rightValue);
    completeAddTAC.add(storeLiteralLine);
}
else if (right instanceof VariableNode)
{
    rightValue = ((VariableNode) right).getVariableName();
    ESymbolAttribute type = this.scopeTable
        .getSymbolByName(rightValue)
        .getAttribute();
    variableType = type == ESymbolAttribute.INT
        ? "I"
        : "F";
}
else
{
    List<TACLine> rightExpressionCode = right.generate3AC();
    rightValue = this.getChildResultRegister(rightExpressionCode);
    completeAddTAC.addAll(rightExpressionCode);
    ESymbolAttribute expressionType = this.getChildResultType(
        rightExpressionCode);
    variableType = expressionType == ESymbolAttribute.INT
        ? "I"
        : "F";
}

TACLine jumpLine = new TACLine();
jumpLine.addElement("LE".concat(variableType));
jumpLine.addElement(leftValue);
jumpLine.addElement(rightValue);
completeAddTAC.add(jumpLine);
/* Add label in caller */
```

```java
        return completeAddTAC;
    }
}
```

```java
/*
*/
package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;
import AbstractSyntaxTree.Nodes.FloatLiteralNode;
import AbstractSyntaxTree.Nodes.IntLiteralNode;
import AbstractSyntaxTree.Nodes.VariableNode;
import AbstractSyntaxTree.TACLine;
import AbstractSyntaxTree.TempararyRegisters;
import java.util.ArrayList;
import java.util.List;
import symboltables.SymbolTable;
import symboltables.enums.ESymbolAttribute;

public class GreaterThanOrEqualToNode extends ASTNode
{
    protected final static int LEFT_OPERAND_INDEX = 0;
    protected final static int RIGHT_OPERAND_INDEX = 1;

    private final SymbolTable scopeTable;

    public GreaterThanOrEqualToNode(SymbolTable table)
    {
        this.scopeTable = table;
    }

    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> completeAddTAC = new ArrayList<>();
        ASTNode left = this.children.get(LEFT_OPERAND_INDEX);
        ASTNode right = this.children.get(RIGHT_OPERAND_INDEX);

        String variableType = "F";
        String leftValue = null;

        if (left instanceof IntLiteralNode)
```

```
{
   String literal = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeLiteralLine = new TACLine();
   storeLiteralLine.addElement("STOREI");
   storeLiteralLine.addElement(literal);
   leftValue = TempararyRegisters
      .getInstance()
      .getTempReg();
   storeLiteralLine.addElement(leftValue);
   completeAddTAC.add(storeLiteralLine);
}
else if (left instanceof FloatLiteralNode)
{
   String literal = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeLiteralLine = new TACLine();
   storeLiteralLine.addElement("STOREF");
   storeLiteralLine.addElement(literal);
   leftValue = TempararyRegisters
      .getInstance()
      .getTempReg();
   storeLiteralLine.addElement(leftValue);
   completeAddTAC.add(storeLiteralLine);
}
else if (left instanceof VariableNode)
{
   leftValue = ((VariableNode) left).getVariableName();
}
else
{
   List<TACLine> leftExpressionCode = left.generate3AC();
   leftValue = this.getChildResultRegister(leftExpressionCode);
   completeAddTAC.addAll(leftExpressionCode);
}

String rightValue = null;

if (right instanceof IntLiteralNode)
{
   String literal = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeLiteralLine = new TACLine();
```

```
    storeLiteralLine.addElement("STOREI");
    storeLiteralLine.addElement(literal);
    rightValue = TempararyRegisters
        .getInstance()
        .getTempReg();
    storeLiteralLine.addElement(rightValue);
    completeAddTAC.add(storeLiteralLine);
    variableType = "I";
}
else if (right instanceof FloatLiteralNode)
{
    String rightLiteral = String.valueOf(
        ((IntLiteralNode) right).getLiteralValue());
    TACLine storeLiteralLine = new TACLine();
    storeLiteralLine.addElement("STOREF");
    storeLiteralLine.addElement(rightLiteral);
    rightValue = TempararyRegisters
        .getInstance()
        .getTempReg();
    storeLiteralLine.addElement(rightValue);
    completeAddTAC.add(storeLiteralLine);
}
else if (right instanceof VariableNode)
{
    rightValue = ((VariableNode) right).getVariableName();
    ESymbolAttribute type = this.scopeTable
        .getSymbolByName(rightValue)
        .getAttribute();
    variableType = type == ESymbolAttribute.INT
        ? "I"
        : "F";
}
else
{
    List<TACLine> rightExpressionCode = right.generate3AC();
    rightValue = this.getChildResultRegister(rightExpressionCode);
    completeAddTAC.addAll(rightExpressionCode);
    ESymbolAttribute expressionType = this.getChildResultType(
        rightExpressionCode);
    variableType = expressionType == ESymbolAttribute.INT
        ? "I"
        : "F";
}
```

```java
        TACLine jumpLine = new TACLine();
        jumpLine.addElement("LT".concat(variableType));
        jumpLine.addElement(leftValue);
        jumpLine.addElement(rightValue);
        completeAddTAC.add(jumpLine);
        /* Add label in caller */

        return completeAddTAC;
    }
}
```

```java
/*
*/
package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;
import AbstractSyntaxTree.Nodes.FloatLiteralNode;
import AbstractSyntaxTree.Nodes.IntLiteralNode;
import AbstractSyntaxTree.Nodes.VariableNode;
import AbstractSyntaxTree.TACLine;
import AbstractSyntaxTree.TempararyRegisters;
import java.util.ArrayList;
import java.util.List;
import symboltables.SymbolTable;
import symboltables.enums.ESymbolAttribute;

public class LessThanNode extends ASTNode
{
    protected final static int LEFT_OPERAND_INDEX = 0;
    protected final static int RIGHT_OPERAND_INDEX = 1;

    private final SymbolTable scopeTable;

    public LessThanNode(SymbolTable table)
    {
        this.scopeTable = table;
    }

    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> completeAddTAC = new ArrayList<>();
```

```
ASTNode left = this.children.get(LEFT_OPERAND_INDEX);
ASTNode right = this.children.get(RIGHT_OPERAND_INDEX);

String variableType = "F";
String leftValue = null;

if (left instanceof IntLiteralNode)
{
   String literal = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeLiteralLine = new TACLine();
   storeLiteralLine.addElement("STOREI");
   storeLiteralLine.addElement(literal);
   leftValue = TempararyRegisters
      .getInstance()
      .getTempReg();
   storeLiteralLine.addElement(leftValue);
   completeAddTAC.add(storeLiteralLine);
}
else if (left instanceof FloatLiteralNode)
{
   String literal = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeLiteralLine = new TACLine();
   storeLiteralLine.addElement("STOREF");
   storeLiteralLine.addElement(literal);
   leftValue = TempararyRegisters
      .getInstance()
      .getTempReg();
   storeLiteralLine.addElement(leftValue);
   completeAddTAC.add(storeLiteralLine);
}
else if (left instanceof VariableNode)
{
   leftValue = ((VariableNode) left).getVariableName();
}
else
{
   List<TACLine> leftExpressionCode = left.generate3AC();
   leftValue = this.getChildResultRegister(leftExpressionCode);
   completeAddTAC.addAll(leftExpressionCode);
}
```

```
String rightValue = null;

if (right instanceof IntLiteralNode)
{
   String literal = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeLiteralLine = new TACLine();
   storeLiteralLine.addElement("STOREI");
   storeLiteralLine.addElement(literal);
   rightValue = TempararyRegisters
      .getInstance()
      .getTempReg();
   storeLiteralLine.addElement(rightValue);
   completeAddTAC.add(storeLiteralLine);
   variableType = "I";
}
else if (right instanceof FloatLiteralNode)
{
   String rightLiteral = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeLiteralLine = new TACLine();
   storeLiteralLine.addElement("STOREF");
   storeLiteralLine.addElement(rightLiteral);
   rightValue = TempararyRegisters
      .getInstance()
      .getTempReg();
   storeLiteralLine.addElement(rightValue);
   completeAddTAC.add(storeLiteralLine);
}
else if (right instanceof VariableNode)
{
   rightValue = ((VariableNode) right).getVariableName();
   ESymbolAttribute type = this.scopeTable
      .getSymbolByName(rightValue)
      .getAttribute();
   variableType = type == ESymbolAttribute.INT
      ? "I"
      : "F";
}
else
{
   List<TACLine> rightExpressionCode = right.generate3AC();
   rightValue = this.getChildResultRegister(rightExpressionCode);
```

```
            completeAddTAC.addAll(rightExpressionCode);
            ESymbolAttribute expressionType = this.getChildResultType(
                rightExpressionCode);
            variableType = expressionType == ESymbolAttribute.INT
                ? "I"
                : "F";
        }

        TACLine jumpLine = new TACLine();
        jumpLine.addElement("GE".concat(variableType));
        jumpLine.addElement(leftValue);
        jumpLine.addElement(rightValue);
        completeAddTAC.add(jumpLine);
        /* Add label in caller */

        return completeAddTAC;
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;
import AbstractSyntaxTree.Nodes.FloatLiteralNode;
import AbstractSyntaxTree.Nodes.IntLiteralNode;
import AbstractSyntaxTree.Nodes.VariableNode;
import AbstractSyntaxTree.TACLine;
import AbstractSyntaxTree.TempararyRegisters;
import java.util.ArrayList;
import java.util.List;
import symboltables.SymbolTable;
import symboltables.enums.ESymbolAttribute;

public class LessThanOrEqualToNode extends ASTNode
{
    protected final static int LEFT_OPERAND_INDEX = 0;
    protected final static int RIGHT_OPERAND_INDEX = 1;

    private final SymbolTable scopeTable;

    public LessThanOrEqualToNode(SymbolTable table)
    {
```

```
    this.scopeTable = table;
}

@Override
public List<TACLine> generate3AC()
{
    List<TACLine> completeAddTAC = new ArrayList<>();
    ASTNode left = this.children.get(LEFT_OPERAND_INDEX);
    ASTNode right = this.children.get(RIGHT_OPERAND_INDEX);

    String variableType = "F";
    String leftValue = null;

    if (left instanceof IntLiteralNode)
    {
        String literal = String.valueOf(
            ((IntLiteralNode) right).getLiteralValue());
        TACLine storeLiteralLine = new TACLine();
        storeLiteralLine.addElement("STOREI");
        storeLiteralLine.addElement(literal);
        leftValue = TempararyRegisters
            .getInstance()
            .getTempReg();
        storeLiteralLine.addElement(leftValue);
        completeAddTAC.add(storeLiteralLine);
    }
    else if (left instanceof FloatLiteralNode)
    {
        String literal = String.valueOf(
            ((IntLiteralNode) right).getLiteralValue());
        TACLine storeLiteralLine = new TACLine();
        storeLiteralLine.addElement("STOREF");
        storeLiteralLine.addElement(literal);
        leftValue = TempararyRegisters
            .getInstance()
            .getTempReg();
        storeLiteralLine.addElement(leftValue);
        completeAddTAC.add(storeLiteralLine);
    }
    else if (left instanceof VariableNode)
    {
        leftValue = ((VariableNode) left).getVariableName();
    }
```

```
else
{
   List<TACLine> leftExpressionCode = left.generate3AC();
   leftValue = this.getChildResultRegister(leftExpressionCode);
   completeAddTAC.addAll(leftExpressionCode);
}

String rightValue = null;

if (right instanceof IntLiteralNode)
{
   String literal = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeLiteralLine = new TACLine();
   storeLiteralLine.addElement("STOREI");
   storeLiteralLine.addElement(literal);
   rightValue = TempararyRegisters
      .getInstance()
      .getTempReg();
   storeLiteralLine.addElement(rightValue);
   completeAddTAC.add(storeLiteralLine);
   variableType = "I";
}
else if (right instanceof FloatLiteralNode)
{
   String rightLiteral = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeLiteralLine = new TACLine();
   storeLiteralLine.addElement("STOREF");
   storeLiteralLine.addElement(rightLiteral);
   rightValue = TempararyRegisters
      .getInstance()
      .getTempReg();
   storeLiteralLine.addElement(rightValue);
   completeAddTAC.add(storeLiteralLine);
}
else if (right instanceof VariableNode)
{
   rightValue = ((VariableNode) right).getVariableName();
   ESymbolAttribute type = this.scopeTable
      .getSymbolByName(rightValue)
      .getAttribute();
   variableType = type == ESymbolAttribute.INT
```

```
            ? "I"
            : "F";
      }
      else
      {
         List<TACLine> rightExpressionCode = right.generate3AC();
         rightValue = this.getChildResultRegister(rightExpressionCode);
         completeAddTAC.addAll(rightExpressionCode);
         ESymbolAttribute expressionType = this.getChildResultType(
            rightExpressionCode);
         variableType = expressionType == ESymbolAttribute.INT
            ? "I"
            : "F";
      }

      TACLine jumpLine = new TACLine();
      jumpLine.addElement("GT".concat(variableType));
      jumpLine.addElement(leftValue);
      jumpLine.addElement(rightValue);
      completeAddTAC.add(jumpLine);
      /* Add label in caller */

      return completeAddTAC;
   }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;
import AbstractSyntaxTree.Nodes.FloatLiteralNode;
import AbstractSyntaxTree.Nodes.IntLiteralNode;
import AbstractSyntaxTree.Nodes.VariableNode;
import AbstractSyntaxTree.TACLine;
import AbstractSyntaxTree.TempararyRegisters;
import java.util.ArrayList;
import java.util.List;

public class MinusNode extends ASTNode
{
   protected final static int LEFT_OPERAND_INDEX = 0;
   protected final static int RIGHT_OPERAND_INDEX = 1;
```

```
@Override
public List<TACLine> generate3AC()
{
   List<TACLine> completeAddTAC = new ArrayList<>();
   TACLine tac = new TACLine();
   ASTNode left = this.children.get(LEFT_OPERAND_INDEX);
   ASTNode right = this.children.get(RIGHT_OPERAND_INDEX);
   if (left instanceof IntLiteralNode || right instanceof IntLiteralNode){
      tac.addElement("SUBI");
   }
   else {
      tac.addElement("SUBF");
   }

   String leftValue = null;

   if (left instanceof IntLiteralNode)
   {
      leftValue = String.valueOf(
         ((FloatLiteralNode) left).getLiteralValue());
      TACLine storeRightValue = new TACLine();
      storeRightValue.addElement("STOREI");
      storeRightValue.addElement(leftValue);
      leftValue = TempararyRegisters.getInstance().getTempReg();
      storeRightValue.addElement(leftValue);
      completeAddTAC.add(storeRightValue);
   }
   else if (left instanceof FloatLiteralNode)
   {
      leftValue = String.valueOf(
         ((FloatLiteralNode) left).getLiteralValue());
      TACLine storeRightValue = new TACLine();
      storeRightValue.addElement("STOREF");
      storeRightValue.addElement(leftValue);
      leftValue = TempararyRegisters.getInstance().getTempReg();
      storeRightValue.addElement(leftValue);
      completeAddTAC.add(storeRightValue);
   }
   else if (left instanceof VariableNode)
   {
      leftValue = ((VariableNode) left).getVariableName();
   }
```

```
else
{
   List<TACLine> leftExpressionCode = left.generate3AC();
   leftValue = this.getChildResultRegister(leftExpressionCode);
   completeAddTAC.addAll(leftExpressionCode);
}

String rightValue = null;

if (right instanceof IntLiteralNode)
{
   rightValue = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeRightValue = new TACLine();
   storeRightValue.addElement("STOREI");
   storeRightValue.addElement(rightValue);
   rightValue = TempararyRegisters.getInstance().getTempReg();
   storeRightValue.addElement(rightValue);
   completeAddTAC.add(storeRightValue);
}
else if (right instanceof FloatLiteralNode)
{
   rightValue = String.valueOf(
      ((FloatLiteralNode) right).getLiteralValue());
   TACLine storeRightValue = new TACLine();
   storeRightValue.addElement("STOREF");
   storeRightValue.addElement(rightValue);
   rightValue = TempararyRegisters.getInstance().getTempReg();
   storeRightValue.addElement(rightValue);
   completeAddTAC.add(storeRightValue);
}
else if (right instanceof VariableNode)
{
   rightValue = ((VariableNode) right).getVariableName();
}
else
{
   List<TACLine> rightExpressionCode = right.generate3AC();
   rightValue = this.getChildResultRegister(rightExpressionCode);
   completeAddTAC.addAll(rightExpressionCode);
}

tac.addElement(leftValue);
```

```
        tac.addElement(rightValue);
        tac.addElement(TempararyRegisters.getInstance().getTempReg());

        completeAddTAC.add(tac);
        return completeAddTAC;
    }
}
```

```
/*
 */
package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;
import AbstractSyntaxTree.Nodes.FloatLiteralNode;
import AbstractSyntaxTree.Nodes.IntLiteralNode;
import static AbstractSyntaxTree.Nodes.Operators.DivideNode.LEFT_OPERAND_INDEX;
import AbstractSyntaxTree.Nodes.VariableNode;
import AbstractSyntaxTree.TACLine;
import AbstractSyntaxTree.TempararyRegisters;
import java.util.ArrayList;
import java.util.List;

public class MultiplyNode extends ASTNode
{
    protected final static int LEFT_OPERAND_INDEX = 0;
    protected final static int RIGHT_OPERAND_INDEX = 1;

    @Override
    public List<TACLine> generate3AC()
    {
        List<TACLine> completeAddTAC = new ArrayList<>();
        TACLine tac = new TACLine();
        ASTNode left = this.children.get(LEFT_OPERAND_INDEX);
        ASTNode right = this.children.get(RIGHT_OPERAND_INDEX);
        if (left instanceof IntLiteralNode || right instanceof IntLiteralNode){
            tac.addElement("MULTI");
        }
        else {
            tac.addElement("MULTF");
        }

        String leftValue = null;
```

```
if (left instanceof IntLiteralNode)
{
   leftValue = String.valueOf(
      ((FloatLiteralNode) left).getLiteralValue());
   TACLine storeRightValue = new TACLine();
   storeRightValue.addElement("STOREI");
   storeRightValue.addElement(leftValue);
   leftValue = TempararyRegisters.getInstance().getTempReg();
   storeRightValue.addElement(leftValue);
   completeAddTAC.add(storeRightValue);
}
else if (left instanceof FloatLiteralNode)
{
   leftValue = String.valueOf(
      ((FloatLiteralNode) left).getLiteralValue());
   TACLine storeRightValue = new TACLine();
   storeRightValue.addElement("STOREF");
   storeRightValue.addElement(leftValue);
   leftValue = TempararyRegisters.getInstance().getTempReg();
   storeRightValue.addElement(leftValue);
   completeAddTAC.add(storeRightValue);
}
else if (left instanceof VariableNode)
{
   leftValue = ((VariableNode) left).getVariableName();
}
else
{
   List<TACLine> leftExpressionCode = left.generate3AC();
   leftValue = this.getChildResultRegister(leftExpressionCode);
   completeAddTAC.addAll(leftExpressionCode);
}

String rightValue = null;

if (right instanceof IntLiteralNode)
{
   rightValue = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeRightValue = new TACLine();
   storeRightValue.addElement("STOREI");
   storeRightValue.addElement(rightValue);
   rightValue = TempararyRegisters.getInstance().getTempReg();
```

```
            storeRightValue.addElement(rightValue);
            completeAddTAC.add(storeRightValue);
        }
        else if (right instanceof FloatLiteralNode)
        {
            rightValue = String.valueOf(
                ((FloatLiteralNode) right).getLiteralValue());
            TACLine storeRightValue = new TACLine();
            storeRightValue.addElement("STOREF");
            storeRightValue.addElement(rightValue);
            rightValue = TempararyRegisters.getInstance().getTempReg();
            storeRightValue.addElement(rightValue);
            completeAddTAC.add(storeRightValue);
        }
        else if (right instanceof VariableNode)
        {
            rightValue = ((VariableNode) right).getVariableName();
        }
        else
        {
            List<TACLine> rightExpressionCode = right.generate3AC();
            rightValue = this.getChildResultRegister(rightExpressionCode);
            completeAddTAC.addAll(rightExpressionCode);
        }

        tac.addElement(leftValue);
        tac.addElement(rightValue);
        tac.addElement(TempararyRegisters.getInstance().getTempReg());

        completeAddTAC.add(tac);
        return completeAddTAC;
    }
}
```

```
/*
*/
package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;
import AbstractSyntaxTree.Nodes.FloatLiteralNode;
import AbstractSyntaxTree.Nodes.IntLiteralNode;
import AbstractSyntaxTree.Nodes.VariableNode;
import AbstractSyntaxTree.TACLine;
```

```
import AbstractSyntaxTree.TempararyRegisters;
import java.util.ArrayList;
import java.util.List;
import symboltables.SymbolTable;
import symboltables.enums.ESymbolAttribute;

public class NotEqualNode extends ASTNode
{
   protected final static int LEFT_OPERAND_INDEX = 0;
   protected final static int RIGHT_OPERAND_INDEX = 1;

   private final SymbolTable scopeTable;

   public NotEqualNode(SymbolTable table)
   {
      this.scopeTable = table;
   }

   @Override
   public List<TACLine> generate3AC()
   {
      List<TACLine> completeAddTAC = new ArrayList<>();
      ASTNode left = this.children.get(LEFT_OPERAND_INDEX);
      ASTNode right = this.children.get(RIGHT_OPERAND_INDEX);

      String variableType = "F";
      String leftValue = null;

      if (left instanceof IntLiteralNode)
      {
         String literal = String.valueOf(
            ((IntLiteralNode) right).getLiteralValue());
         TACLine storeLiteralLine = new TACLine();
         storeLiteralLine.addElement("STOREI");
         storeLiteralLine.addElement(literal);
         leftValue = TempararyRegisters
            .getInstance()
            .getTempReg();
         storeLiteralLine.addElement(leftValue);
         completeAddTAC.add(storeLiteralLine);
      }
      else if (left instanceof FloatLiteralNode)
      {
```

```
    String literal = String.valueOf(
        ((IntLiteralNode) right).getLiteralValue());
    TACLine storeLiteralLine = new TACLine();
    storeLiteralLine.addElement("STOREF");
    storeLiteralLine.addElement(literal);
    leftValue = TempararyRegisters
        .getInstance()
        .getTempReg();
    storeLiteralLine.addElement(leftValue);
    completeAddTAC.add(storeLiteralLine);
}
else if (left instanceof VariableNode)
{
    leftValue = ((VariableNode) left).getVariableName();
}
else
{
    List<TACLine> leftExpressionCode = left.generate3AC();
    leftValue = this.getChildResultRegister(leftExpressionCode);
    completeAddTAC.addAll(leftExpressionCode);
}

String rightValue = null;

if (right instanceof IntLiteralNode)
{
    String literal = String.valueOf(
        ((IntLiteralNode) right).getLiteralValue());
    TACLine storeLiteralLine = new TACLine();
    storeLiteralLine.addElement("STOREI");
    storeLiteralLine.addElement(literal);
    rightValue = TempararyRegisters
        .getInstance()
        .getTempReg();
    storeLiteralLine.addElement(rightValue);
    completeAddTAC.add(storeLiteralLine);
    variableType = "I";
}
else if (right instanceof FloatLiteralNode)
{
    String rightLiteral = String.valueOf(
        ((IntLiteralNode) right).getLiteralValue());
    TACLine storeLiteralLine = new TACLine();
```

```
            storeLiteralLine.addElement("STOREF");
            storeLiteralLine.addElement(rightLiteral);
            rightValue = TempararyRegisters
                .getInstance()
                .getTempReg();
            storeLiteralLine.addElement(rightValue);
            completeAddTAC.add(storeLiteralLine);
        }
        else if (right instanceof VariableNode)
        {
            rightValue = ((VariableNode) right).getVariableName();
            ESymbolAttribute type = this.scopeTable
                .getSymbolByName(rightValue)
                .getAttribute();
            variableType = type == ESymbolAttribute.INT
                ? "I"
                : "F";
        }
        else
        {
            List<TACLine> rightExpressionCode = right.generate3AC();
            rightValue = this.getChildResultRegister(rightExpressionCode);
            completeAddTAC.addAll(rightExpressionCode);
            ESymbolAttribute expressionType = this.getChildResultType(
                rightExpressionCode);
            variableType = expressionType == ESymbolAttribute.INT
                ? "I"
                : "F";
        }

        TACLine jumpLine = new TACLine();
        jumpLine.addElement("EQ".concat(variableType));
        jumpLine.addElement(leftValue);
        jumpLine.addElement(rightValue);
        completeAddTAC.add(jumpLine);
        /* Add label in caller */

        return completeAddTAC;
    }
}
```

```
/*
*/
```

```
package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;
import AbstractSyntaxTree.Nodes.FloatLiteralNode;
import AbstractSyntaxTree.Nodes.IntLiteralNode;
import AbstractSyntaxTree.Nodes.VariableNode;
import AbstractSyntaxTree.TACLine;
import AbstractSyntaxTree.TempararyRegisters;
import java.util.ArrayList;
import java.util.List;

public class PlusNode extends ASTNode
{
  protected final static int LEFT_OPERAND_INDEX = 0;
  protected final static int RIGHT_OPERAND_INDEX = 1;

  @Override
  public List<TACLine> generate3AC()
  {
    List<TACLine> completeAddTAC = new ArrayList<>();
    TACLine tac = new TACLine();
    ASTNode left = this.children.get(LEFT_OPERAND_INDEX);
    ASTNode right = this.children.get(RIGHT_OPERAND_INDEX);
    if (left instanceof IntLiteralNode || right instanceof IntLiteralNode){
      tac.addElement("ADDI");
    }
    else {
      tac.addElement("ADDF");
    }

    String leftValue = null;

    if (left instanceof IntLiteralNode)
    {
      leftValue = String.valueOf(
        ((FloatLiteralNode) left).getLiteralValue());
      TACLine storeRightValue = new TACLine();
      storeRightValue.addElement("STOREI");
      storeRightValue.addElement(leftValue);
      leftValue = TempararyRegisters.getInstance().getTempReg();
      storeRightValue.addElement(leftValue);
      completeAddTAC.add(storeRightValue);
    }
```

```
else if (left instanceof FloatLiteralNode)
{
   leftValue = String.valueOf(
      ((FloatLiteralNode) left).getLiteralValue());
   TACLine storeRightValue = new TACLine();
   storeRightValue.addElement("STOREF");
   storeRightValue.addElement(leftValue);
   leftValue = TempararyRegisters.getInstance().getTempReg();
   storeRightValue.addElement(leftValue);
   completeAddTAC.add(storeRightValue);
}
else if (left instanceof VariableNode)
{
   leftValue = ((VariableNode) left).getVariableName();
}
else
{
   List<TACLine> leftExpressionCode = left.generate3AC();
   leftValue = this.getChildResultRegister(leftExpressionCode);
   completeAddTAC.addAll(leftExpressionCode);
}

String rightValue = null;

if (right instanceof IntLiteralNode)
{
   rightValue = String.valueOf(
      ((IntLiteralNode) right).getLiteralValue());
   TACLine storeRightValue = new TACLine();
   storeRightValue.addElement("STOREI");
   storeRightValue.addElement(rightValue);
   rightValue = TempararyRegisters.getInstance().getTempReg();
   storeRightValue.addElement(rightValue);
   completeAddTAC.add(storeRightValue);
}
else if (right instanceof FloatLiteralNode)
{
   rightValue = String.valueOf(
      ((FloatLiteralNode) right).getLiteralValue());
   TACLine storeRightValue = new TACLine();
   storeRightValue.addElement("STOREF");
   storeRightValue.addElement(rightValue);
   rightValue = TempararyRegisters.getInstance().getTempReg();
```

```
            storeRightValue.addElement(rightValue);
            completeAddTAC.add(storeRightValue);
        }
        else if (right instanceof VariableNode)
        {
            rightValue = ((VariableNode) right).getVariableName();
        }
        else
        {
            List<TACLine> rightExpressionCode = right.generate3AC();
            rightValue = this.getChildResultRegister(rightExpressionCode);
            completeAddTAC.addAll(rightExpressionCode);
        }

        tac.addElement(leftValue);
        tac.addElement(rightValue);
        tac.addElement(TempararyRegisters.getInstance().getTempReg());

        completeAddTAC.add(tac);
        return completeAddTAC;
    }
}
```

```
/*
*/
package AbstractSyntaxTree;

import java.util.ArrayList;
import java.util.Hashtable;

public class TinyAssemblyGenerator {
    private int registerCounter;
    private Hashtable<String, String> variableTable;
    private ArrayList<String> variableDeclarations;
    private ArrayList<String> assemblyCode;
    private boolean jumpGenerated;
    private boolean newlineUsed;
    private ArrayList<String> freeRegisters;

    public TinyAssemblyGenerator() {
        registerCounter = 0;
        variableTable = new Hashtable<>();
        assemblyCode = new ArrayList<>();
```

```
    variableDeclarations = new ArrayList<>();
    jumpGenerated = false;
    newlineUsed = false;
    freeRegisters = new ArrayList<>();
}

/**
 *  must be called before generating new Tiny code from a new Basic Block
 */
public void reset() {
    registerCounter = 0;
    variableTable = new Hashtable<>();
    assemblyCode = new ArrayList<>();
    variableDeclarations = new ArrayList<>();
    jumpGenerated = false;
    newlineUsed = false;
    freeRegisters = new ArrayList<>();
}

/**
 *  generate Tiny code from the TAC
 * @param tacList the TAC from which to generate Tiny code
 * @return the ArrayList of Strings representing the generated Tiny code
 */
public ArrayList<String> assemble(ArrayList<TACLine> tacList) {
    for (TACLine tacLine: tacList) {
        ArrayList<String> newCode;

        // switch based on the TAC instruction type
        switch (tacLine.getElements().get(0)) {
            case "STOREI":
            case "STOREF":
                newCode = generateSTORE(tacLine);
                break;

            case "MULTI":
            case "MULTF":
            case "ADDI":
            case "ADDF":
            case "SUBI":
            case "SUBF":
            case "DIVI":
            case "DIVF":
```

```
        newCode = generateAddSubMultDiv(tacLine);
        break;

    case "WRITEI":
    case "WRITEF":
    case "WRITES":
        newCode = generateWRITE(tacLine);
        break;

    case "RET":
        newCode = generateRET(tacLine);
        break;

    case "LABEL":
        newCode = generateLABEL(tacLine);
        break;

    case "LEI":
    case "LEF":
    case "EQI":
    case "EQF":
    case "NEI":
    case "NEF":
    case "LTI":
    case "LTF":
    case "GTI":
    case "GTF":
    case "GEI":
    case "GEF":
        newCode = generateConditionalBranch(tacLine);
        break;

    case "READI":
    case "READF":
        newCode = generateREAD(tacLine);
        break;

    case "JUMP":
        newCode = generateJUMP(tacLine);
        break;

    // this should only be entered if an invalid TAC instruction is encountered
    // in the case of an invalid TAC instruction, do not produce any Tiny code
```

```
            // ###
            // Note: maybe this should be changed? Maybe we should throw an error?
            // ##

            default:
               newCode = new ArrayList<>();
         }

         assemblyCode.addAll(newCode);
      }

      // handles the way that the given test code only declares 'str newline' when the newline
is actually used
      if (newlineUsed) {
         assemblyCode.add(0, "str newline \"\\n\"");
      }

      // handles the stipulation that all variable declarations must appear at the beginning of
the tiny code,
      // as opposed to when they are first needed
      for (int i = variableDeclarations.size() - 1; i >= 0; i--) {
         assemblyCode.add(0, variableDeclarations.get(i));
      }

      // handles the way that the given test code only generates 'label' statements if the Tiny
code contains a jump
      // statement
      if (!jumpGenerated) {
         ArrayList<String> toRemove = new ArrayList<>();
         for (String s: assemblyCode) {
            if (s.startsWith("label ")) {
               toRemove.add(s);
            }
         }

         // remove label statements if there isn't a jump statement in the tiny code
         for (String s: toRemove) {
            assemblyCode.remove(s);
         }
      }

      return assemblyCode;
   }
```

```
/**
 *  gets the next free register
 *
 *  note: in accordance to the given test code, a register is only considered free if it has not
been assigned yet,
 *  or if it has only been assigned temporarily in a Store instruction in which both
arguments are variables
 *  (neither argument is already in a register)
 *
 *  this seems like a random arbitrary rule, but I implemented it this way to match the test
code
 *
 * @return the string representation of the next available register
 */
private String getNextFreeRegister() {
   if (!freeRegisters.isEmpty()) {
      return freeRegisters.remove(0);
   }

   String r = "r" + registerCounter++;
   freeRegisters.add(r);
   return r;
}

/**
 * @return the lowest unassigned register number
 */
private String getNextRegister() {
   return "r" + registerCounter++;
}

/**
 * @param tacName   the TAC variable name
 * @return the tiny variable name or register name associated with tacName
 */
private String getVarNameFromVariableTable(String tacName) {
   if (!variableTable.containsKey(tacName))
   {
      // assign register
      if (tacName.startsWith("$")) {
         variableTable.put(tacName, "r" + registerCounter++);
      }
```

```
            // the var name is not a register, and not 'newLine'
            else if (!tacName.equals("newline"))
            {
                variableTable.put(tacName, tacName);
                String varInitialization = "var " + variableTable.get(tacName);
                variableDeclarations.add(varInitialization);
            }

            // the var name is specifically 'newLine'
            else
            {
                newlineUsed = true;
                return "newline";
            }
        }

        return variableTable.get(tacName);
    }

    /**
     * STORE I/F
     *
     * @param tacLine
     * @return
     */
    private ArrayList<String> generateSTORE(TACLine tacLine) {
        ArrayList<String> code =  new ArrayList<>();

        ArrayList<String> tac = tacLine.getElements();

        String arg1 = tac.get(1);
        String arg2 = tac.get(2);

        // we need to address the special scenario in which we declare this temporary register
'free'
        if (!arg1.startsWith("$") && !arg2.startsWith("$"))
        {
            String arg3 = getNextFreeRegister();

            String line = "move " + arg1 + " " + arg3;
            code.add(line);
            line = "move " + arg3 + " " + arg2;
```

```
            code.add(line);

            return code;
        }

        if (!Character.isDigit(arg1.toCharArray()[0])) {
            arg1 = getVarNameFromVariableTable(arg1);
        }

        if (!Character.isDigit(arg2.toCharArray()[0])) {
            arg2 = getVarNameFromVariableTable(arg2);
        }

        String line = "move " + arg1 + " " + arg2;
        code.add(line);

        return code;
    }

    /**
     * WRITE S/F/I
     *
     * @param tacLine
     * @return
     */
    private ArrayList<String> generateWRITE(TACLine tacLine) {
        ArrayList<String> code =  new ArrayList<>();

        ArrayList<String> tac = tacLine.getElements();

        String arg0 = tac.get(0);
        String arg1 = tac.get(1);

        arg1 = getVarNameFromVariableTable(arg1);

        String operator = "";

        switch (arg0) {
            case "WRITES":
                operator = "writes ";
                break;

            case "WRITEF":
```

```
            operator = "writer ";
            break;

        case "WRITEI":
            operator = "writei ";
            break;
    }

    String line = "sys " + operator + arg1;
    code.add(line);

    return code;
}

/**
 * RET
 *
 * @param tacLine
 * @return
 */
private ArrayList<String> generateRET(TACLine tacLine) {
    ArrayList<String> code =  new ArrayList<>();

    String line = "sys halt";
    code.add(line);

    return code;
}

/**
 * LABEL
 *
 * @param tacLine
 * @return
 */
private ArrayList<String> generateLABEL(TACLine tacLine) {
    ArrayList<String> code =  new ArrayList<>();

    String arg1 = tacLine.getElements().get(1);

    String line = "label " + arg1;
    code.add(line);
```

```
    return code;
}

/**
 * JUMP
 *
 * @param tacLine
 * @return
 */
private ArrayList<String> generateJUMP(TACLine tacLine) {
    ArrayList<String> code =  new ArrayList<>();

    jumpGenerated = true;

    String arg1 = tacLine.getElements().get(1);

    String line = "jmp " + arg1;
    code.add(line);

    return code;
}

/**
 * Conditional Branch: LE EQ GT NE GE LT
 *
 * @param tacLine
 * @return
 */
private ArrayList<String> generateConditionalBranch(TACLine tacLine) {
    ArrayList<String> code =  new ArrayList<>();

    jumpGenerated = true;

    ArrayList<String> tac = tacLine.getElements();

    String arg0 = tac.get(0);
    String arg1 = tac.get(1);
    String arg2 = tac.get(2);
    String arg3 = tac.get(3);

    if (!arg1.startsWith("$") && !arg2.startsWith("$"))
    {
        String temp = getNextRegister();
```

```
    String line = "move " + arg2 + " " + temp;
    arg2 = temp;
    code.add(line);

    arg1 = getVarNameFromVariableTable(arg1);
}

else
{
    arg1 = getVarNameFromVariableTable(arg1);
    arg2 = getVarNameFromVariableTable(arg2);
}

String operator = "";

switch (arg0.charAt(2)) {
    case 'I':
        operator = "cmpi ";
        break;

    case 'F':
        operator = "cmpr ";
        break;
}

String line = operator + arg1 + " " + arg2;
code.add(line);

String operator2 = "";

switch (arg0.substring(0,2)) {
    case "LE":
        operator2 = "jle ";
        break;

    case "EQ":
        operator2 = "jeq ";
        break;

    case "NE":
        operator2 = "jne ";
        break;
```

```
        case "GT":
            operator2 = "jgt ";
            break;

        case "GE":
            operator2 = "jge ";
            break;

        case "LT":
            operator2 = "jlt ";
            break;
    }

    String line2 = operator2 + arg3;
    code.add(line2);

    return code;
}

/**
 * READ
 *
 * @param tacLine
 * @return
 */
private ArrayList<String> generateREAD(TACLine tacLine) {
    ArrayList<String> code =  new ArrayList<>();

    ArrayList<String> tac = tacLine.getElements();

    String arg0 = tac.get(0);
    String arg1 = tac.get(1);

    arg1 = getVarNameFromVariableTable(arg1);

    String operator = "";

    switch (arg0) {
        case "READI":
            operator = "readi ";
            break;

        case "READF":
```

```
            operator = "readr ";
            break;

        case "READS":
            operator = "reads ";
            break;
    }
    String line = "sys " + operator + arg1;
    code.add(line);

    return code;
}

/**
 * ADD/MUL/DIV/SUB
 *
 * @param tacLine
 * @return
 */
private ArrayList<String> generateAddSubMultDiv(TACLine tacLine) {
    ArrayList<String> code =  new ArrayList<>();

    ArrayList<String> tac = tacLine.getElements();

    String arg0 = tac.get(0);
    String arg1 = tac.get(1);
    String arg2 = tac.get(2);
    String arg3 = tac.get(3);

    arg1 = getVarNameFromVariableTable(arg1);
    arg2 = getVarNameFromVariableTable(arg2);
    arg3 = getVarNameFromVariableTable(arg3);

    String line = "move " + arg1 + " " + arg3;
    code.add(line);

    String operator = "";

    switch (arg0) {
        case "MULTI":
            operator = "muli ";
            break;
```

```
            case "MULTF":
                operator = "mulr ";
                break;

            case "ADDI":
                operator = "addi ";
                break;

            case "ADDF":
                operator = "addr ";
                break;

            case "SUBI":
                operator = "subi ";
                break;

            case "SUBF":
                operator = "subr ";
                break;

            case "DIVI":
                operator = "divi ";
                break;

            case "DIVF":
                operator = "divr ";
                break;
        }

        String line2 = operator + arg2 + " " + arg3;
        code.add(line2);

        return code;
    }
}
```

Section 2: Teamwork

The team for this capstone project consisted of three people. While group projects are notoriously frustrating things that make some swear to never take classes that include this type of class work, it was quite the opposite for this project. All members started each of the assignments early on and met, if only briefly, at least once a week to see where each of the other members were with their part and was needed before the next meeting. One of the group members was more experienced with documentation writing and therefore took over the vast majority of writing each of the reports and this portfolio. This same member was a little slower with the coding aspect of the project but helped out design ideas, trouble-shooting, and debugging. The other two members of the group then pretty much cut the coding portion of the project in half and worked together to complete each section. As a group we used GitHub to manage the contributions from each of the members and merge the project into a single working venture. The member that did most of the documentation was also more experienced with using git and was able to help out in managing branching and merging bits and pieces of the project and helped teach this tool to the other members. Overall, the feeling was that the project was pretty much split into thirds and shared equally.

Section 3: Design Pattern

The factory pattern was used to handle the character stream and generate tokens from the source code. This method pattern uses factory methods that deal with creating objects without the need to specify the exact class of the object that is being created. This pattern was used to take the hard work of writing all the code that would need to know the class it was creating out of the project at this point.

Section 4: Technical Writing

Compiler Project

1. Introduction

The purpose of the project was to learn about the concepts of a compiler in an abbreviated and practical manner. Briefly, a compiler reads in a high level code written by a human and makes sure that all the symbols are valid. It will then pass along those symbols and check on the validity of groupings of symbols. The groupings are then passes along and checked against predetermined statement lists that define the language. Lastly, these groupings are then translated into code that a machine can execute at the register level.

This project was part of the Compilers class, which was a requirement in the completion of the degree of Bachelor of Science in Computer Science.

The goal of this project, to create a compiler for the Little programming language, was achieved in several steps. A brief introduction to each of these steps will be made in the next section. However, the purpose of the rest of this paper is to explain the details of this achievement in more detail.

2. Background

A compiler is a program that takes a text file written in a given high level language (i.e., Java or Python) and creates byte-code for the written program that can be executed on some target machine.

The purpose of a compiler is to allow programmers to develop coded projects using higher-level, more powerful languages that are easier to read and write than lower-level languages (i.e.: C or assembly).

Compilers are partitioned into four general components to reduce coupling and simplify the compilation process. These four components are:

* Scanner or Lexer
* Parser (syntax and semantic analysis)
* Symbol Table and Intermediate Code Generation
* Code Generation and Optimization

Examples of compilers include gcc (C and C++), clang (for languages C, C++, Objective C/C++, OpenCL, CUDA, and RenderScript [https://clang.llvm.org/]), javac (for Java), go (for the Go language), and Microsoft Visual Studio (for C#) (https://stackoverflow.com/questions /31180056/example-of-compiler-interpreter-and-both).

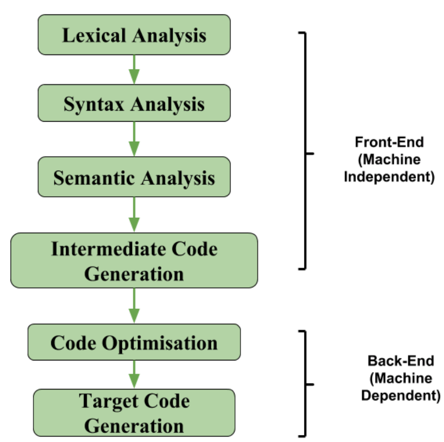

Figure 1: Parts of a compiler
https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/

3. **Methods and Discussion**

A. ANTLR Setup

ANTLR is a program which generates a scanner and parser for the user given a configuration file (of the file format .g4). G4-files contain 2 sections: the lexer and the parser. This program is provided by ANTLR in the form of a jar named 'antlr-4.7.1-complete.jar'. This jar contains multiple programs, including one that we used to generate a lexer and a parser.

As part of the process of setting up ANTLR, we followed a tutorial on using ANTLR provided by ANTLR at www.antlr.org. The tutorial provided the following .g4-file named Hello.g4:

```
grammar Hello;
r  : 'hello' ID ;
ID : [a-z]+ ;
WS : [ \t\r\n]+ -> skip ;
```

After downloading and placing ANTLR's 'antlr-4.7.1-complete.jar' file-path into our environment variables, we were able to invoke the 'org.antlr.v4.Tool' program in this jar with the Hello.g4 file and an arbitrary string as input. This program scanned the string using the scanner defined in Hello.g4, and it determined if it was syntactically correct according to Hello.g4's grammar.

B. Scanner

A scanner (also known as a tokenizer or lexer) is a program that reads in a text file that is a series of characters representing a program of a specific language. The scanner then produces a series of tokens that are passed on to the parser to analyze. A token is the most basic of components of a programming language. Characters are separated into different types of tokens based on their functions (keywords, operators, identifiers, separators, and reserved words). These tokens are dependant on the rules of the specific programming language that is being used.

After installing and testing ANTLR, a partial grammar for the language called Little was created from the text file describing the structure of the language. The grammar file included rules for identifiers, keywords, and operators. Based on the grammar, ANTLR was used to generate the basics of the scanner in Java (see figure below). Java was decided upon based upon the common experiences of the group members.

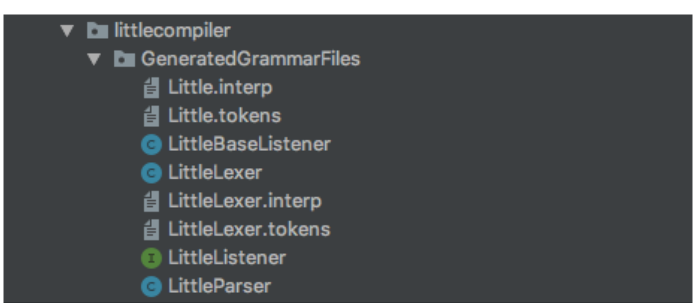

Figure 2: List of automatically generated files.

There were four files created in addition to those that were auto generated. The driver class was used as a start point. A factory class was written to help with dependency injection and to make unit testing of written files easier. The factory pattern was used to keep class coupling to a minimum. This type of coupling made the project more flexible and less fragile. The compiler class was written to aggregate all the tools used for the entire Compiler class project. The compiler class was created with two dependences, the lexer (or scanner) and the parser. The implemented method is for the scanner and was used to generate a file for holding the token output to be used later by the parser. The method for the parser was left as a stub since it was not needed for this part of the project. The last hand written class was token visualizer. This class was used to generate a string of all the tokens from a given program and then compared to a known output. There were five different programs used to test the operation of the scanner output and each difference was return with a value of zero. The two auto generated classes used were LittleLexer.java and LittleParser.java. These names were automatically given from the name of the grammar used and the function of the class. Lastly, script files were made to aid in running the appropriate files to recreate the ANTLR files and then use the hand written driver file to automatically scan test files for grading.

There were a few difficulties in accomplishing this first part of the compiler project. The first came from learning the nuances of the regular expressions that ANTLR will accept. For example, in the figure below the regular expressions for a string literal and use the tilde symbol (~) as NOT. This was not listed at either http://www.rexegg.com/regex-quickstart.html or https://www.shortcutfoo.com/app/dojos/regex/cheatsheet and was eventually arrived at through trial and error. So, ~[‘ ‘] means ‘NOT nothing’ or ‘anything’ is acceptable as a string literal and a comment.

```
D:          [0-9];              // digit
AN:         [a-zA-Z0-9];        // alphanumeric
ID:         [a-zA-Z]AN*;        // identifier
INTLIT:     D+;                 // integer literal
FLOATLIT:   D*.D+;              // float literal
STRINGLIT:  '"' ~['']* '"';     // string literal
COMMENT:    '--'~['']* '\n';    // comment
WS:         [ \t\r\n]+ -> skip; // skip spaces, tabs, carrage returns, and newlines
```

Figure 3: Identifiers from the grammar Little.g4

Another hurdle to get past was learning the ANTLR API. This took time to learn the new tool but was made a little easier from experience using other development environments and APIs. The automatically generated files were not exactly human readable and took a while to wade through along with searching for and reading through ANTLR documentation. This in turn helped to solve another challenge, how to pass files to the parser and then, in turn, pass tokens to the parser. The ANTLR libraries provided options and methods to help keep the hand written code simpler and easier to manage. The last item to work through was how ANTLR handled scanning and parsing. In class this was handled as two different but related steps in building a compiler. ANTLR, however, generates both of these components at the same time, which made it more difficult to distinguish what exactly need to be done for this part of the project.

C. Parser

According to Merriam-Webster the definition of parsing a grammar is “to divide (a sentence) into grammatical parts and identify the parts and their relations to each other” (https://merriam-webster.com). A parser is a program that acts as the interpreter part of a compiler that analyses a list of tokens by breaking them into smaller pieces and finds meaning in them that conforms to a set of rules defined by the grammar. Meaning can be made of these tokens by creating a parse tree of all of the tokens and their groupings. There are two stages involved in the parsing process: syntactical analysis, and semantic parsing. Syntactic analysis looks for a meaningful expression from the tokens generated by the lexer. The definition of the algorithmic procedures to create components comes from the usage of a context-free grammar. The placement of tokens must happen in a meticulous order and each of the rules in the grammar work to create an expression that has the proper syntax. This does not mean that the accepted string will have any meaning. The semantic stage of parsing determines the meaning and implications of a valid expression. There are two ways in which data from the input string of tokens can be derived from the start symbol: top-down parsing, and bottom-up parsing. Bottom-up parsing takes the entire list of tokens and tries to assemble a parse tree in reverse, working from the leaves to the start symbol. Top-down parsing, which is the way ANTLR works (see figure below), begins with the start symbol and works from left to right to find all the expressions that fit the given grammar. (https://www. techopedia.com)

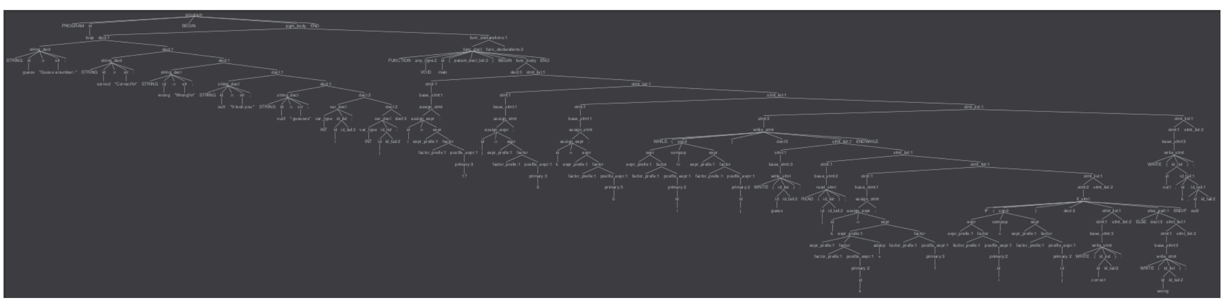

Figure 4: Generated parse tree.

There were a few changes to our code base for this stage of the project. First of all we needed to create a method generate a string of valid tokens that wasn’t printed out this time. This token stream was then to be fed into the parser as input. Before we could actually feed this stream to the parser, the LittleGrammar.g4 class needed to have all the productions added to it. This was a rather involved process since order and naming of non-terminals matters. A small sample of these productions can be seen below.

```
program: 'PROGRAM' id 'BEGIN' pgm_body 'END';
id: IDENTIFIER;
pgm_body: decl func_declarations;
decl: string_decl decl | var_decl decl |;

string_decl: 'STRING' id ':=' str ';';
str: STRINGLITERAL;

var_decl: var_type id_list ';';
var_type: 'FLOAT' | 'INT';
any_type: var_type | 'VOID';
id_list: id id_tail;
id_tail: ',' id id_tail |;
```

Figure 5: Sample of grammar productions.

We could then make the necessary addition to the driver to call the parser with the output string of tokens from the lexer. The sole output was to print that the input was either accepted or rejected. In order to accomplish this task we added a new class for error identification and handling. The factory class was also updated to accommodate this new dependency on the error strategy we selected.

Again, there were several roadblocks to overcome in finishing this part of the compiler project. First off, with the addition of productions to our grammar file, we found that there were needed changes to the original identifier list (seen in the figure below on the left). One identifier was added to the file and the names were changed to become more human readable. This also made it easier to add these to the different productions without needing to lookup what abbreviation belonged to each of the identifiers. The last change was in how to identify a comment. Originally a comment could contain anything in it and ended with a new line identifier (\n). This became problematic since the first new line encountered at the end of a line of code was turned into a comment. Simply excluding the new line symbol from the body of the comment identifier changed this undesirable behavior.

```
D:          [0-9];              // digit
AN:         [a-zA-Z0-9];        // alphanumeric
ID:         [a-zA-Z]AN*;        // identifier
INTLIT:     D+;                 // integer literal
FLOATLIT:   D*.D+;              // float literal
STRINGLIT:  '"' ~['']* '"';     // string literal
COMMENT:    '--'~['']* '\n';    // comment
WS:         [ \t\r\n]+ -> skip; // skip spaces, tabs, carrage returns, and newlines
```

```
IDENTIFIER: LETTER ALPHANUMERIC*;
INTLITERAL: DIGIT+;
FLOATLITERAL: DIGIT* '.' DIGIT+;
STRINGLITERAL: '"' ~["]* '"';

DIGIT: [0-9];
ALPHANUMERIC: [a-zA-Z0-9];
LETTER: [a-zA-Z];

COMMENT: '--' ~['\n']* '\n' -> skip ;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Figure 6: Changes to the identifiers in the grammar.

The next challenge came in trying to change the non-terminal names of some of the productions to make them more readable. However, this quickly cascaded into chaos and the only option was to go back to the naming convention given in the reference text file for the grammar. Even though it had been mentioned in class, we found that disordering productions caused havoc when trying to some of the test files. The challenge was trying to determine a proper order. This was mostly done by carefully reading through each of the productions and a little bit of trial and error. The last obstacle was perhaps the largest. In order to decide whether or not the input would be accepted or rejected we had to find some way to handle errors in parsing. There didn't seem to be any way of catching errors since the lexer output a string of all tokens, valid or not. The parser would also work its way through the entire string fed to it even though there were marked errors in the output. After trying to catch these errors for both the lexer and parser and failing, it was decided to look for some hints in outside resources. This proved fruitful. The solution was to override some methods in one of the ANTLR libraries to do the error handling (https://stackoverflow .com/questions/39533809/antlr4-how-to-detect-unrecognized-token-and-given-sentence-is-invalid). Time to explore this solution was necessary to understand exactly what errors were being caught and how. The solution was finally modified to catch only the error we needed to either accept or reject the given input.

D.Symbol Table

A compiler creates and maintains a data structure called a symbol table so that information can be stored about occurrences of various items in the language "such as variable names, function names, objects, classes, interfaces" and the like (https://www. tutorialspoint.com/compiler_design/compiler_design_symbol_table.htm). A compiler will use this data structure in both synthesis and analysis. Depending on the language used, this table may serve several different purposes. These include variable declaration verification, type checking (semantic correctness of expressions and assignments), scope resolution of a name, and centralized and structured names of all entities. This symbol table can take the form of a linear list, linked list, binary search tree or a hash table. For small languages a linear or linked list is adequate since traversing this list wouldn't take too much time or computing power. For larger languages a hash table or binary search tree is necessary for reducing time (O(1) and O($\log_2$ n) respectively) and cutting down on the usage of the CPU (https://www. geeksforgeeks.org/symbol-table-compiler/). The information stored in the table about each symbol is its name, type, and attribute and can be formatted as follows:

<symbol name, type, attribute>

The first decision to be made was to use either the default setting (listener pattern) for ANTLR or choose the visitor pattern. The decision was made based on the pros and cons of each pattern. The visitor pattern is good for using the parser output directly for interpretation. Tree traversal is in full control of the user, meaning only one branch is visited in conditionals and ***n*** number of visits can be made in loops. The visitor pattern also seems to be more flexible. Both pattern have their virtues when the input is translated to a lower level such as instructions on a virtual machine. In his book, *The Definitive ANTLR 4 Reference* (Terence Parr 2012, https://media.pragprog .com/titles/tpantlr2/listener.pdf), Terence Parr states:

> *"The biggest difference between the listener and visitor mechanisms is that listener methods are called by the ANTLR-provided walker object, whereas visitor methods mush walk their children with explicit visit calls. Forgetting to invoke visit() on a node's children means those subtrees don't get visited."*

Another important deciding factor between these two patterns is that the visitor uses the *call stack* and the listener used an *explicit stack*. The explicit stack is "allocated on the heap" and managed by the walker provided by ANTLR (https://stackoverflow.com/questions/20714492/ antlr4-listeners-and-visitors-which-to-implement#30762056). This means that for large inputs the visitor pattern could run into stack overflow issues but the listener pattern wouldn't have a problem. The main deciding factor came with the amount of management it would take to use the visitor pattern. With the listener pattern a programmer is only interacting with the provided tree walker. This seemed like the least amount of added headache for the most gain even though the language is small. The second decision was to come to a consensus on what type of data structure to use in making the symbol table. Since this is a little language a stack was decided upon for this purpose. Three changes to the grammar file became necessary at this stage in the project. As seen below in Figure 1, *func_declarations:* recursively refers to itself or an empty string. The caused a new function declaration to be created after an existing declaration that was empty. The second line in the figure shows the change to fix this issue.

```
func_declarations: func_decl func_declarations |;
func_declarations: func_decl func_declarations | func_decl;
```

Figure 7: Change to func_declarations.

The second change to the grammar file was for the *if_stmt*. An *if* statement doesn't necessarily need to have an *else* block to follow it. The change made was so that the *else* block would be optional, as seen in Figure 2.

```
if_stmt: 'IF' '(' cond ')' decl stmt_list else_part 'ENDIF';
else_part: 'ELSE' decl stmt_list |;
if_stmt: 'IF' '(' cond ')' decl stmt_list else_part 'ENDIF' | 'IF' '(' cond ')' decl stmt_list 'ENDIF' ;
else_part: 'ELSE' decl stmt_list;
```

Figure 8: Change to if_stmt and else_part.

The third change was in the *pgm_body* declaration. Since one of the test files contained a body declaration but no function declarations, an adjustment to the grammar file was necessary here (see Figure 3).

```
pgm_body: decl func_declarations;
```

```
pgm_body: decl func_declarations | decl ;
```

Figure 9: Change to pgm_body.

There were six total new classes used for part three of the project. The first was Symbol.java to represent a symbol held in a table. The next was SymbolTable.java; this holds all the symbols with a single scope. The class SymbolTables.java links all the tables in their relative scopes. The last of the table classes was SymbolTableVisualizer.java. This class recursively traversed all of the symbol tables in a tree structure while printing each of the table names and contents. Two enumerator classes were also added to support the symbol classes. The first held the possible attributes of a symbol (int, float, string, or void). The second was for the two possible types of a symbol (var or procedure).

The difficulties for this portion of the project came in the form of a few smaller hurdles to get past. One of the example output files listed the attribute for a procedure to be an *int* but it didn't seem to be an integer. The issue ended up being reading the return type from the procedure. For the Little language a procedure can either return an integer value (*int*) or return nothing (*void*). So either will show up depending on the whether the procedure returns anything or not. Another small issue that needed to be hunted down was that all child tables added to the *global* table and beyond were added twice. This turned out to be a minor oversight. It was thought that children needed to be added manually but when the SymbolTable.java class was called with a new symbol it automatically added it as well. Once found, this was a simple matter to correct. Another thing that was realized at this point in the project was that not all the methods in the LittleBaseListener.java class needed to be filled out. Only the methods that dealt with variables, procedures, integers, floats, strings, and voids.

D.Code Generation

Although optimizations can still be applied, the generation of assembly language or machine code "can be considered as the final phase of compilation" (https://www.tutorials point .com/compiler_design/compiler_design_code_generation.htm). Optimizing the generated code wasn't a part of the absolute requirement for this project it is often seen as part of the code generation step and process. From the first step to the last, the entire process should minimally have the following properties:

- The value of the source code should be exactly represented.
- The management of memory and usage of the target CPU ought to be efficient.

Keeping the two points above in mind, one type of optimization is called peephole optimization. This technique is easiest to apply locally within a basic block of code. Several statements will be analyzed and checked for several possible optimizations. One is the elimination of redundant instructions. There may be multiple instructions for loading and storing values that end up having the same meaning even when some are removed. A second way to optimize is the elimination of code that's unreachable. If, for example, a method is to print a certain line but breaks or returns before that statement is reached then it can be removed from the generated code. Optimizing flow control is another way to improve performance. For example, the following figure has unnecessary jumps that accomplish nothing as far as the compiler is concerned.

```
...
MOV R1, R2
GOTO L1
...
L1 :   GOTO L2
L2 :   INC R1
```

Figure 10: Example of code with an unnecessary jump.

It is easy to see here that the first *GOTO* statement can easily be changed to go directly to *L2* and completely remove the intermediate jump. The simplification of algebraic expressions is another place to make simple changes to streamline the code generation process. Adding zero to a value is an unnecessary process altogether and adding or subtracting by one can be replace by a simple *increment* or *decrement* statement. Strength reduction involves replacing operations with ones that will manufacture the same result but burn up less space and time to process. Also, the instructions held on the target machine can deploy instructions that are more sophisticated and be much more efficient in performing specific operations. This will not only produce results in a more proficient manner but also improve code quality. Additionally, the registers in the processor that variable must be allocated to should be taken into consideration. If the compiler is to truly be useful to a developer then the generation of debug data will also be required (https://en.wikipedia.org/wiki/Code_generation_%28compiler%29).

An optional yet extremely useful part of code generation is the development and use of a syntax tree. This is actually just a parse tree but in a condensed form. The syntax tree takes out all the names and types of variables and declarations and creates an easy to read tree with just the essentials in it (see Figure 11).

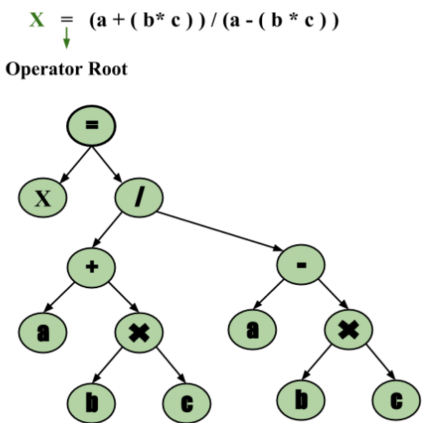

Figure 11: Syntax tree representing the assignment of an algebraic expression to *X*.

From the syntax tree it is much easier to generate three-address code working from the leaves back up to the root node. Three-address code is a linear representation of the code received from the semantic analyzer. This involves making a statement that contains three or fewer references, one for the result and two for the operands. Occasionally a statement will have less than three references in it but will still fall under the name of a three-address statement. There are two forms that a three-address code can be represented as: quadruples and triples. For example, take a statement such as *a = b + c * d;* and turn it into the following three-address statements.

```
r1 = c * d;
r2 = b + r1;
a = r2
```

Figure 12: Three-address statement for *a = b + c * d;*

The quadruple representation of this is separated into four different fields: the operator, two arguments, and the result (see Figure below).

Op	arg_1	arg_2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		a

Figure 13: Quadruple representation of three-address statements.

Triples representation, as the name implies, only has three fields: the operator and two arguments. Triples, however, "face the problem of code immovability while optimizing, as the results are positional and changing the order for position of an expression may cause problems" (https://www.tutorialspoint.com/compiler_design/compiler_design_intermediate_code_generations.htm).

Changes made include minor edits to the Little.g4 grammar file to make the three-address code a little easier to generate from the nodes in the abstract syntax tree. Thirty-two node classes were made to actually generate the appropriate intermediate three-address code. All of these classes inherited from one ASTNode.java class. Two Singleton classes were written to keep track of the labels for *if*, *else*, and *while* blocks of code and the temporary registers (*$T4*) for storing values to. The third Singleton class to be created was ParameterRegisterHandler.java to keep track of value registers that would be passed to and from functions. A fourth Singleton class, FunctionCodeGenerator.java, was written to make sure functions each had their own identifier and could be kept track of. The AST.java class more or less sits at the top of this part of the project, keeping track of nodes and children in the abstract syntax tree. The class TACLine.java takes all of the individual pieces of the three-address code generated in each of the node classes and concatenates them into strings that will then be translated into Tiny code. One working class and one test class were also created to take the three-address code and convert it to Tiny code as the last step.

Difficulties overcome in this part of the project are as follows. Nested and recursive statements needed to be changed and debugged so that the three-address code would generate properly and keep the correct register number throughout the life of the variable. Mapping from the given *.micro files to three-address code was a bit different from the examples in class and ended up taking a little while to see how the loops were labeled and taken care of. Going from three-address code to Tiny code also had similar difficulties because there was not a one to one mapping between the two codes. Another challenge was matching out put generator values to given outputs and freeing the proper registers for reuse. Also, learning all the syntax differences between conditional branching instructions was not trivial.

F.Full-fledged Compiler

All of the parts of this project built upon one another in a fairly seamless manner. Overall project planning was able to be accomplished since the over view of the entire project was explained early on in the class and was reiterated for each of the different parts.

This allowed for forethought to be made for the next portion of the project while designing and writing the current portion. The factory pattern was used for the first two parts of the project to create the token stream. A logic and code was built onto this base for the second two parts. Before any real programming began, it was decided that GitHub would be used to bring all the parts written by each of the group members together and as excellent version control tool. As far as day-to-day communication between members, a group text message was started in the first week of class and maintained throughout the semester.

4. Conclusion and Future Work

While each of the sections of the project was completed by each deadline and in working order, time was not a commodity in surplus. Had this not been a constraint better coding practices would have been employed. This would include common code reuse, more opportunities for inheritance and better use of design patterns. Optimization was discussed in class but was optional for the project. This was a welcome relief as the project was done just on time with the requirements as they were. However, it would have been a constructive challenge to use something like peephole optimization and see how much the code could have been reduced and what the clock cycles may possibly have been brought down to. Along with this would be to see what minimum number of registers could be used and still meet the requirements of three-address code generation. An exploration of many of the coding possibilities with the Tiny language and what that might have done to the compiler would be well worth the time to investigate along with what actual programs could be written in this language.

Section 5: UML

The size of this project didn't really lend itself to creating a UML diagram. While diagramming can be an important part of a project that will span several classes and be an undertaking by many developers, we were able to discuss our project and stick to a design without the use of this tool.

Section 6: Design Trade-offs

Due to time constraints, a few of the classes started to follow the God Class Anti-Pattern. This wasn't exactly a conscious decision to write code this way but happened as quick fixes and trial runs of testing. These coding snippets were slotted for cleanup at a later date that never seemed to come along while the snippets grew in numbers. Suddenly, it seemed, this technical debt was too large to deal with before the end of the project.

Section 7: Software Development Life Cycle Model

We decided to use an agile approach to the project since we consisted of a small working group. This scrum approach worked well for us. Since class met three times weekly we would take just a few minutes after class to hold brief sprints to see what each of the others had accomplished and where we needed to go as a group for lab each week. This approach kept the lines of communication open and a sense of progress from week to week. No large contributions ever needed to be made since several small one were accomplished each sprint cycle.