CSCI 468 Compilers
Spring 2017

Ryan Darnell
Stephanie McLaren

Teamwork

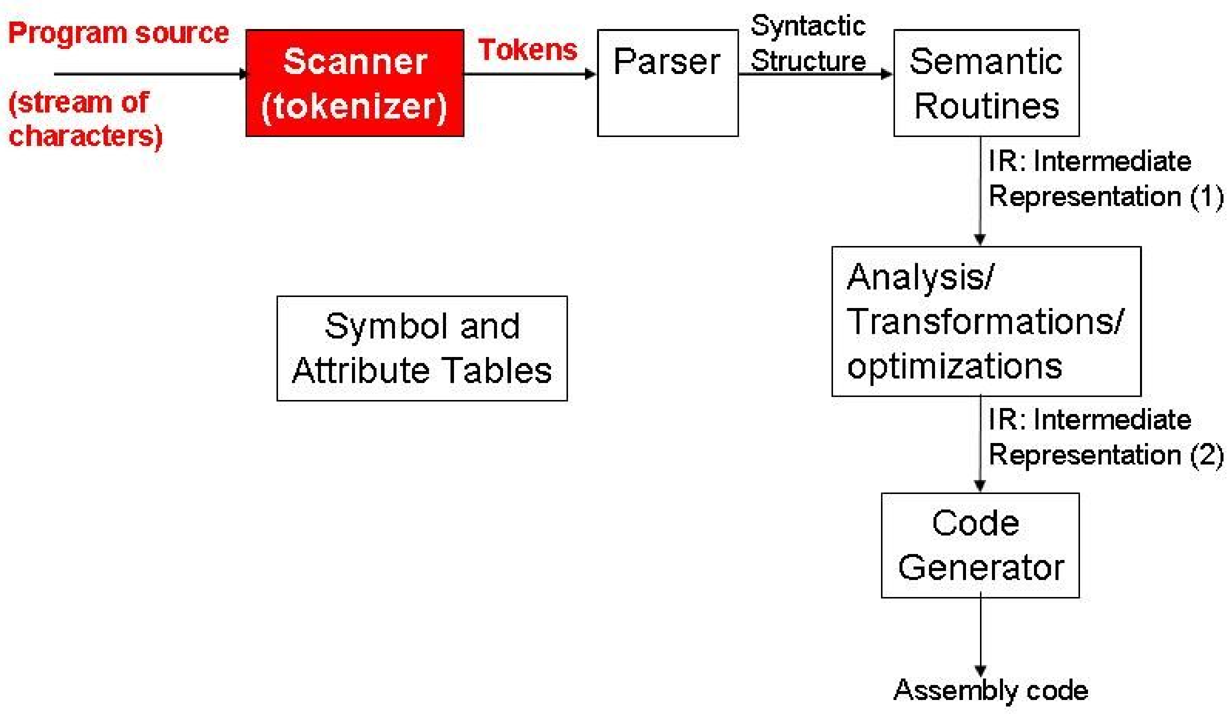
Our team started out with three people and we were working on everything together. Just before the first step was due the third team member ended up dropping the class leaving us with two team members. Working on everything together and meeting our deadlines became difficult as the steps also became more difficult. By the end of step 2 we knew that a different approach was needed.

To accomplish the tasks efficiently we then decided to use a divide and conquer method. Team member 1 became primarily responsible for writing the code and team member 2 became primarily responsible for writing the report and completing the portfolio.

During each lab we spent 5 - 10 minutes covering what we had finished, what still needed to be done, our time frame and any issues we were having. We spent the remainder of the lab working on our tasks. Once we had our tasks finished, each team member double checked the code and the report before the final version was submitted.

Once we split the workload meeting our deadlines for the programming and staying on target with the report became much easier to accomplish. As we worked through each step, if team member 1 needed assistance figuring out how to implement a piece of code, team member 2 provided research and suggestions. If team member 2 was unsure what should be put in the report, team member one provided suggestions. In essence, we functioned as a working unit to complete our project.

Program


Program source
(stream of characters)
Scanner (tokenizer)
Tokens
Parser
Syntactic Structure
Semantic Routines
IR: Intermediate Representation (1)
Analysis/ Transformations/ optimizations
IR: Intermediate Representation (2)
Code Generator
Assembly code
Symbol and Attribute Tables

Step 1: Scanner

You will develop a scanner (also known as a tokenizer) for the given grammar in this step. A scanner will take a sequence of characters (i.e. the source files of the LITTLE language) as input and produce a sequence of tokens which will be the input to the next step (Parser) as shown in the figure.

The scanner's source code is normally generated by a scanner generation tool such as ANTLR. These tools normally work by taking the token definitions expressed by regular expressions and generates the source code for the scanner automatically (for this project, the tokens are specified in the language's grammar).

The programmer has to add the code to handle the scanner's output. For example, at this step you will print all the tokens in the standard output. In the step 2 of this project you will modify your scanner to feed the parser replacing the print routines by calls to the parser and passing the tokens as parameters. In order to learn how to merge the code of the scanner generator with the rest of your source code, please, read the user's manual of the tool you decide to use.

At the end of this step, the program developed (the very first part of our compiler) should be able to open a program's source file written in the LITTLE language and recognize its tokens. At this step, the output of your program should be the prints of each token's type and value (see below). In the next step the same output will be used to feed the parser.

Your scanner program should be able to open and read a LITTLE source file and print the all the valid tokens within the source file and their respective type in the standard output. You might want to redirect the output to a file and compare your results with the output files provided for the test cases.

Step 2: Parser

Your goal in this step is to generate the parser for the project's grammar. By the end of this step your compiler should be able to take a source file as the input and parse the content of that file returning "Accepted" if the file's content is correct according to the grammar or "Not Accepted" if it is not. Now the scanner created in the first step will be modified to feed the parser. Instead of printing the tokens, the scanner has to return what token is recognized in each step.

Similar to the scanner case, there are tools that automatically generate parsers based on context-free grammar descriptions. More precisely, there are parser generators for two main classes of context-free grammars: LR(1) and LL(k). Please, refer to the textbook or one of the references at the end of this handout for the theory of LL(k) and LR(k) parsing.

At this step you have to create the code for the parser of your compiler. We recommend you to do that using one of the tools described above, although the whole parser could be written manually without the help of any tool.
The following basic steps should be followed:

1. Write the parser using one of the tools suggested above;
2. Modify your lexer in the first step to feed tokens to parser instead of printing them;
3. Your parser should print "Accepted" to the standard output if the input file is accepted by the language's grammar or "Not accepted" otherwise.

Step 3: Symbol Table Generation

Your goal in this step is to process variable declarations and create Symbol Tables. A symbol table is a data structure that keeps information about non-keyword symbols that appear in source programs. Variables and String Variables are examples of such symbols. Other example of symbols kept by the symbol table are the names of functions or procedures. The symbols added to the symbol table will be used in many of the further phases of the compilation.

In Step 2 you didn't need token values since only token types are used by parser generator tools to guide parsing. But in this step your parser needs to get token values such as identifier names and string literals from your scanner. You also need to add semantic actions to create symbol table entries and add those to the symbol table.

There are multiple scopes where variables can be declared:

- Variables can be declared before any functions. These are "global" variables, and can be accessed from any function.
- Variables can be declared as part of a function's parameter list. These are "local" to the function, and cannot be accessed by any other function.
- Variables can be declared at the beginning of a function body. These are "local" to the function as well.
- Variables can be declared at the beginning of a then block, an else block, or a repeat statement. These are "local" to the block itself. Other blocks, even in the same function, cannot access these variables.

Your task in this step of the project is to construct symbol tables for each scope in your program. For each scope, construct a symbol table, then add entries to that symbol table as you see declarations. The declarations you have to handle are integer/float declarations, which should record the name and type of the variable, and string declarations, which should additionally record the value of the string. Note that typically function declarations/definitions would result in entries in the symbol table, too, but you do not have to record them for this step.

Note that the scopes in the program are nested (function scopes are inside global scopes, and block scopes are nested inside function scopes, or each other). You will have to keep track of this nesting so that when a piece of code uses a variable named "x" you know which scope that variable is from.

Your compiler should output the string "DECLARATION ERROR <var_name>" if there are two declarations with the same name in the same scope.

For each symbol table in your program, use the following format:

```
Symbol table <scope_name>
name <var_name> type <type_name>
name <var_name> type <type_name> value <string_value>
```

The global scope should be named "GLOBAL", function scopes should be given the same name as the function name, and block scopes should be called "BLOCK X" where X is a counter that increments every time you see a new block scope. Function parameters should be included as part of the function scope.

It is expected that the entries in your symbol table appear in the same order that they appear in the original program. Keep this in mind as you design the data structures to store your symbol tables.

Step 4 IRCode and Assembly Code Generation

Stage 1: Design the IRNode
IRNode have a basic format like this:

Opcode First operand Second operand Result

The IRs that you will be using in this step will look like:

ADDI OP1 OP2 RESULT (Integer Add)
ADDF OP1OP2RESULT (Floating Point Add)
SUBIOP1OP2RESULT (Integer Subtract)
SUBFOP1OP2RESULT (Floating Point Subtract: RESULT = OP1/OP2)
MULTI OP1OP2RESULT (Integer Multiplication)
MULTF OP1OP2RESULT (Floating Point Multiplication)
DIVIOP1OP2RESULT (Integer Division)
DIVFOP1OP2RESULT (Floating Point Division: RESULT = OP1/OP2)
STOREI OP1RESULT (Integer Store, Store OP1 to RESULT)
STOREFOP1RESULT (FP Store)
GTOP1OP2LABEL (If OP1 > OP2 Goto LABEL)
GEOP1OP2LABEL (If OP1 >= OP2 Goto LABEL)
LTOP1OP2LABEL (If OP1 < OP2 Goto LABEL)
LEOP1OP2LABEL (If OP1 <= OP2 Goto LABEL)
NEOP1OP2LABEL (If OP1 != OP2 Goto LABEL)
EQOP1OP2LABEL (If OP1 = OP2 Goto LABEL)
JUMPLABEL (Direct Jump)
LABEL STRING (set a STRING Label)
READIRESULT
READFRESULT
WRITEIRESULT
WRITEFRESULT

Stage 2: Create IRNode and place it in "list" and "graph" In this step you will implement semantic routines to generate IRNodes and then you will put the IRNodes in two different data structures the "list" is what you will need for code (tiny code) generation and "graph" is what you need in later steps to do data flow analysis. While writing semantic routines you would have to introduce temporary variables to store temporary results from an arithmetic expression.

ADDI A $T1 $T2 In many languages when you have a “float” and “int” operand mixed together in an expression, the result type becomes a “float”. If you need to get a new Temp variable to store the result, it will be of type “float” in that case. HOWEVER, as a simplification we don’t mix types in our micro language.

Driver.java

```java
import java.io.*;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

/**
 *
 * @author Ryan Darnell
 * && Stephanie McLaren
 *
 */

public class Driver {
/**
* @param args the command line arguments
*/
      public static void main(String[] args) {
         try {
            FileReader file = new FileReader(args[0]);
            CharStream stream = new ANTLRInputStream(file);
            junkLexer lexer = new junkLexer(stream);
            Vocabulary vocab = lexer.getVocabulary();
            Token tok = null;
            CommonTokenStream tokens = new CommonTokenStream(lexer);
            junkParser parser = new junkParser(tokens);
            Listener listener = new Listener();
            new ParseTreeWalker().walk(listener, parser.program());
            parser.getNumberOfSyntaxErrors());
        } catch (FileNotFoundException ex) {
            System.out.println("Failed to do stuff");
        } catch (IOException ex) {
            System.out.println("Failed to do stuff");
        }
    }
}
```

Symbol.java

```
public class Symbol {

   private String type;
   private String name;
   private String value;

   public Symbol(String type, String name) {
      this.type = type;
      this.name = name;
  }

   public Symbol(String type, String name, String value) {
      this.type = type;
      this.name = name;
      this.value = value;
  }

   public String getType() { return type; }
   public String getName() { return name; }
   public String getValue() { return value; }

public void print() {
   System.out.println(toString());
  }

@Override
public String toString() {
   String print = "name " + name + " type " + type;
   if (value != null) {
      print += " value " + value;
     }
   return print;
  }
```

SymbolTable.java

```
import java.util.*;

public class SymbolTable {
   private String name;
   private HashMap<String, Symbol> symbols;
   private ArrayList<Symbol> inOrder;
   private HashMap<String, Symbol> ancestorSymbols;
   private SymbolTable parent;
   private ArrayList<SymbolTable> children;

public SymbolTable(SymbolTable parent, HashMap<String, Symbol> ancestorSymbols){
   this.parent = parent;
   this.ancestorSymbols = (ancestorSymbols == null) ? new HashMap<>() :
      ancestorSymbols;
   symbols = new HashMap<>();
   inOrder = new ArrayList<>();
   children = new ArrayList<>();
  }

public void setName(String name) {
   this.name = name;
  }

public void addSymbol(Symbol symbol) {
   if (!exists(symbol)) {
      symbols.put(symbol.getName(), symbol);
      inOrder.add(symbol);
   } else {
      System.out.println("DECLARATION ERROR " +
                  symbol.getName());
      System.exit(1);
     }
  }

public void addChild(SymbolTable table) {
   children.add(table);
  }

public SymbolTable createChild() {
   SymbolTable child = new SymbolTable(this, packageSymbols());
   addChild(child);
   return child;
  }

public SymbolTable getParent() {
   return parent;
  }
public void printTable() {
   System.out.println("Symbol table " + name);
   for (Symbol symbol : inOrder) {
      symbol.print();
     }
  }

public void printAll() {
   printTable();
   for (SymbolTable table : children) {
      System.out.println(); //line space
      table.printAll();
     }
  }

private boolean exists(Symbol symbol) {
   if (symbols.containsKey(symbol.getName()))
            return true;
   else
      return false;
  }

private HashMap<String, Symbol> packageSymbols() {
   HashMap<String, Symbol> temp = new HashMap<>();
   temp.putAll(symbols);
   temp.putAll(ancestorSymbols);
   return temp;
  }

public Symbol searchSymbol(String name) {
   HashMap<String, Symbol> allSymbols = packageSymbols();
   return allSymbols.get(name);
  }
}
```

IRBuilder.java

```
import java.util.*;

public class IRBuilder {

   private ArrayList<String> ir_list;
   private String condition; //either 'while' or 'if' to state
   private int labelNum;
   private ArrayList<String> labelStack;
   private int regNum;
   private String compIR;
   private char dataType;
   private SymbolTable currentTable;

public IRBuilder(SymbolTable currentTable) {
   ir_list = new ArrayList<>();
   labelNum = 1;
   regNum = 1;
   this.currentTable = currentTable;
   labelStack = new ArrayList<>();
  }

public void updateTable(SymbolTable s) {
   currentTable = s;
  }

private void clean() {
   //clean all token values to prevent spilling
   condition = null;
   compIR = null;
   dataType = ' ';
  }

public void enterMain() {
   ir_list.add("LABEL main");
   ir_list.add("LINK");
  }

public void endProgram() {
   ir_list.add("RET");
  }

public void buildWrite(String[] params) {
   for (String param : params) {
      Type = currentTable.searchSymbol(param).getType().toUpperCase().
      toCharArray()[0];
      ir_list.add("WRITE" + type + " " + param);
      System.out.println("WRITE" + type + " " + param);
    }}
```

```
public void buildRead(String[] params) {
   char type;
   for (String param : params) {
      type = currentTable.searchSymbol(param).getType().toUpperCase().
      toCharArray()[0];
      ir_list.add("READ" + type + " " + param);
      System.out.println("WRITE" + type + " " + param);
     }
  }

public void parseComparison(String[] set) {
      if (isNumber(set[2])) {
         dataType = set[2].contains(".") ? 'F' : 'I';
      } else {
         dataType = currentTable.searchSymbol(set[0]).getType().toUpperCase().
         toCharArray()[0];
     }
      setCompop(set[1]);
      routeCondition(set[0], set[2]);
}

private boolean isNumber(String numberMaybe) {
   return numberMaybe.matches("-?\\d*\\.?\\d+");
  }

// Set the condition to be routed to for routeCondition
public void setCondition(String cond) {
   this.condition = cond;
  }

// call appropriate condition statement
public void routeCondition(String op1, String op2)throws NullPointerException {
      //Determining which complex statement is executed
   if (condition == null)throw new NullPointerException("Condition value is
      null");
   else if (condition == "if")
      buildIfHeader(op1, op2);
   else
      buildWhileHeader(op1, op2);
      clean();
  }

private void setCompop(String compop) {
//set the proper IR instruction type for the given comparison operator
//Note: this does not set the int vs float call
   switch(compop) {
      case "<":
         compIR = "GE";
         break;
      case "<=":
         compIR = "GT";
         break;
      case ">":
         compIR = "LE";
         break;
      case ">=":
         compIR = "LT";
         break;
      case "=":
         compIR = "NE";
         break;
      case "!=":
         compIR = "EQ";
         break;
      default:
         //shits broke yo
         compIR = null;
     }
   compIR += dataType; //append the dataType flag?
  }

private void buildWhileHeader(String op1, String op2) {
   ir_list.add("LABEL label" + labelNum++);
   if (isNumber(op1)) {
      System.out.println("STORE" + dataType + " " + op1 + " " + (op1 = "$T" +
      regNum));
      ir_list.add("STORE" + dataType + " " + op1 + " " + (op1 = "$T" +
      regNum++));
     }
   if (isNumber(op2)) {
      System.out.println("STORE" + dataType + " " + op2 + " " + (op2 = "$T" +
      regNum));
      ir_list.add("STORE" + dataType + " " + op2 + " " + (op2 = "$T" +
      regNum++));
     }
   labelStack.add(0,"label" + (labelNum - 1));
   System.out.println(compIR + " " + op1 + " " + op2 + " label" + (labelNum));
   ir_list.add(compIR + " " + op1 + " " + op2 + " label" + labelNum++);
}

public void endWhile() {
   String op1 = "JUMP " + labelStack.remove(0);
   String op2 = "LABEL " + labelStack.remove(0);
   System.out.println(op1 + "\n" + op2);
   ir_list.add(op1);
   ir_list.add(op2);
  }

private void buildIfHeader(String op1, String op2) {
   if (isNumber(op1)) {
```

```
            System.out.println("STORE" + dataType + " " + op1 + " " + (op1 = "$T" +
            regNum));
            ir_list.add("STORE" + dataType + " " + op1 + " " + (op1 = "$T" +
            regNum++));
          }
      if (isNumber(op2)) {
            System.out.println("STORE" + dataType + " " + op2 + " " + (op2 = "$T" +
            regNum));
            ir_list.add("STORE" + dataType + " " + op2 + " " + (op2 = "$T" +
            regNum++));
          }
      System.out.println(compIR + " " + op1 + " " + op2 + " label" + (labelNum));
      ir_list.add(compIR + " " + op1 + " " + op2 + " label" +labelNum++);
  }

  public void enterElsePart() {
      String elseLabel = labelStack.remove(0);
      labelStack.add(0, "label" + labelNum);
      System.out.println("JUMP label" + labelNum);
      ir_list.add("JUMP label" + labelNum++);
      System.out.println("LABEL " + elseLabel);
      ir_list.add("LABEL " + elseLabel);
  }

  public void exitIf() {
      String elseLabel = labelStack.remove(0);
      System.out.println("LABEL " + elseLabel);
      ir_list.add("LABEL " + elseLabel);
    }
  }
```

Listener.java

```java
import java.util.*;

public class Listener extends junkBaseListener {
    SymbolTable s;
    IRBuilder ir;
    Symbol symbol;
    boolean newTable, newTableHeader, programHeader = false;
    String variableType, variableValue;
    ArrayList<String> variableName = new ArrayList<>();
    int block = 1;

@Override
public void enterProgram(junkParser.ProgramContext ctx) {
    s = new SymbolTable(null, null);
    ir = new IRBuilder(s);
    s.setName("GLOBAL");
    programHeader = true;
}

@Override
public void exitProgram(junkParser.ProgramContext ctx) {
    ir.endProgram();
}

@Override
public void enterFunc_decl(junkParser.Func_declContext ctx) {
    pushSymbolTable();
    newTable = true;
    newTableHeader = true;
    ir.enterMain();
}

@Override
public void enterFunc_body(junkParser.Func_bodyContext ctx) {
    newTableHeader = false;
}

@Override
public void exitFunc_decl(junkParser.Func_declContext ctx) {
    popSymbolTable();
}

@Override
public void enterVar_decl(junkParser.Var_declContext ctx) {
    variableName.clear();
}

@Override
```

```
public void exitVar_decl(junkParser.Var_declContext ctx) {
    for (String name : variableName) {
        s.addSymbol(new Symbol(variableType, name));
   }
    variableType = null;
    variableName.clear();
}

@Override
public void enterVar_type(junkParser.Var_typeContext ctx){
    variableType = ctx.getText();
}

@Override
public void enterId(junkParser.IdContext ctx) {
    if (newTable) {
        s.setName(ctx.getText());
        variableName.clear();
        newTable = false;

    } else if (newTableHeader) {
        s.addSymbol(new Symbol(variableType, ctx.getText()));
        variableType = null;

    } else if (programHeader) {
        programHeader = false;

    } else {
        variableName.add(ctx.getText());
    }
}

@Override
public void exitId(junkParser.IdContext ctx) { }

@Override
public void enterString_decl(junkParser.String_declContext ctx) {
    variableType = "STRING";
}

@Override
public void exitString_decl(junkParser.String_declContext ctx) {
    s.addSymbol(new Symbol(variableType,variableName.get(variableName.size() - 1),
    variableValue));
    variableName.clear();
    variableType = null;
    variableValue = null;
}

@Override
```

```
public void enterStr(junkParser.StrContext ctx) {
    variableValue = ctx.getText();
}

@Override
public void enterAssign_expr(junkParser.Assign_exprContext ctx) {
    System.out.println("Assignment Expression: " + ctx.getText());
    int count = ctx.getChildCount();
}

@Override
public void enterExpr(junkParser.ExprContext ctx) {
    System.out.println("Expression: " + ctx.getText());
}

@Override
public void exitExpr(junkParser.ExprContext ctx) {
    System.out.println("Exit Expression");
}

@Override
public void enterExpr_prefix(junkParser.Expr_prefixContext ctx){ }

@Override
public void enterFactor(junkParser.FactorContext ctx) {
    int count = ctx.getChildCount();
}

@Override
public void enterPostfix_expr(junkParser.Postfix_exprContext ctx){ }

@Override
public void enterPrimary(junkParser.PrimaryContext ctx)
{
    System.out.println("Primary: " + ctx.getText());
}

@Override
public void enterAddop(junkParser.AddopContext ctx) {
    System.out.println("Addop: " + ctx.getText());
}

@Override
public void enterMulop(junkParser.MulopContext ctx) {
    System.out.println("Mulop: " + ctx.getText());
}

@Override
public void exitAssign_expr(junkParser.Assign_exprContext ctx){
    System.out.println("----------------------------------\n");
}

@Override
public void enterCond(junkParser.CondContext ctx) {
    int count = ctx.getChildCount();
    String[] set = new String[count];
    for (int i = 0; i < count; i++) {
        set[i] = ctx.getChild(i).getText();
   }
    ir.parseComparison(set);
}

@Override
public void enterIf_stmt(junkParser.If_stmtContext ctx) {
    pushSymbolTable();
    s.setName("BLOCK " + block);
    block++;
    ir.setCondition("if");
}

@Override
public void exitIf_stmt(junkParser.If_stmtContext ctx) {
    popSymbolTable();
    ir.exitIf();
    ir.setCondition(null);
}

@Override
public void enterElse_part(junkParser.Else_partContext ctx) {
    if (ctx.getChildCount() > 0) {
        popSymbolTable();
        pushSymbolTable();
        s.setName("BLOCK " + block);
        block++;
        ir.enterElsePart();
    }
}

@Override
public void exitElse_part(junkParser.Else_partContext ctx){ }

@Override
public void enterWhile_stmt(junkParser.While_stmtContext ctx) {
    pushSymbolTable();
    s.setName("BLOCK " + block);
    block++;
    ir.setCondition("while");
}

@Override
```

```
public void exitWhile_stmt(junkParser.While_stmtContext ctx) {
   popSymbolTable();
   ir.endWhile();
   ir.setCondition(null);
}

@Override
public void enterWrite_stmt(junkParser.Write_stmtContext ctx) {
   String[] params = ctx.getChild(2).getText().split(",");
   ir.buildWrite(params);
}

@Override
public void enterRead_stmt(junkParser.Read_stmtContext ctx) {
   String[] params = ctx.getChild(2).getText().split(",");
   ir.buildRead(params);
}

public SymbolTable getSymbolTable() {
   return s;
}

public void popSymbolTable() {
   s = s.getParent();
   ir.updateTable(s);
}

public void pushSymbolTable() {
   s = s.createChild();
   ir.updateTable(s);
  }
}
```

IRBuilder.java

```
import java.util.*;

public class IRBuilder {

private ArrayList<String> ir_list;
private String condition; //either 'while' or 'if' to state
private int labelNum;
private ArrayList<String> labelStack;
private int regNum;
private String compIR;
private char dataType;
private SymbolTable currentTable;

// Expression Builder
private ArrayList<String> stack;
private ArrayList<String> postfixOutput;
private boolean inExpression = false;

public IRBuilder(SymbolTable currentTable) {
   ir_list = new ArrayList<>();
   labelNum = 1;
   regNum = 1;
   this.currentTable = currentTable;
   labelStack = new ArrayList<>();
   stack = new ArrayList<>();
   postfixOutput = new ArrayList<>();
  }

public void updateTable(SymbolTable s) {
   currentTable = s;
  }

private void clean() {
   //clean all token values to prevent spilling
   condition = null;
   compIR = null;
   dataType = ' ';
   stack.clear();
   postfixOutput.clear();
  }

//remove '(' and ')' from stack along with variables
private void trimStack() {

ArrayList<String> to_trim = new
            ArrayList<>(Arrays.asList("(",")"));
   stack.removeAll(to_trim);
  }
```

```
//This is officially my new favorite method
private void printList(ArrayList<String> list, boolean vertical) {
   String orient = vertical ? "\n" : " ";
   for (String el : list) {
      System.out.print(el + orient);
     }
   System.out.println();
  }

private boolean isNumber(String numberMaybe) {
   return numberMaybe.matches("-?\\d*\\.?\\d+");
  }

private boolean isRegister(String numberMaybe) {
   return numberMaybe.matches("r\\d+");
  }

public void enterMain() {
   ir_list.add("LABEL main");
   ir_list.add("LINK");
  }

public void endProgram() {
   ir_list.add("RET");
   System.out.println("Printing IR Code: <size> " + ir_list.size());
   printList(ir_list, true);
  }

public void buildWrite(String[] params) {
   //Structure: list1,list2,list3,etc
   char type;
   for (String param : params) {
      type = currentTable.searchSymbol(param).getType().toUpperCase().
      toCharArray()[0];
      ir_list.add("WRITE" + type + " " + param);
     }
  }

public void buildRead(String[] params) {
   //Structure: list1,list2,list3,etc
   char type;
   for (String param : params) {
      type = currentTable.searchSymbol(param).getType().toUpperCase().
      toCharArray()[0];
      ir_list.add("READ" + type + " " + param);
  }
}

public void parseComparison(String[] set) {
```

```
// Scanning op2 as it is more likely to be a literal &
// therefore faster to determine
   if (isNumber(set[2])) {
      //check if it floats
      dataType = set[2].contains(".") ? 'F' : 'I';
   } else {
      //get the data type from the variable
      dataType = currentTable.searchSymbol(set[0]).getType().toUpperCase().
      toCharArray()[0];
     }

   // convert ops if expression evident
   String op1 = elementToIR(set[0]);
   String op2 = elementToIR(set[2]);
   //checking if op1 or op2 was an expr ? change ops
   //to what's left on stack : do nothing
   trimStack();
   if (!isNumber(op1) && !isRegister(op1) && currentTable.searchSymbol(op1) ==
   null) {
      op1 = stack.get(0);
   }
   if (!isNumber(op2) && !isRegister(op2) && currentTable.searchSymbol(op2) ==
   null) {
      op2 = stack.get(1 % stack.size());
   }
   //set compop after dataType is found
   setCompop(set[1]);
   routeCondition(op1, op2);
  }

// Set the condition to be routed to for routeCondition
public void setCondition(String cond) {
   this.condition = cond;
}

// call appropriate condition statement
public void routeCondition(String op1, String op2)throws NullPointerException {
//Determining which complex statement is executed in order to route the call
   if (condition == null) throw new NullPointerException("Condition value is
   null");
   else if (condition == "if")
      buildIfHeader(op1, op2);
   else
      buildWhileHeader(op1, op2);
   clean();
  }

private void setCompop(String compop) {
//set the proper IR instruction type for the given comparison operator
   //Note: this does not set the int vs float call
```

```
      switch(compop) {
         case "<":
            compIR = "GE";
            break;
         case "<=":
            compIR = "GT";
            break;
         case ">":
            compIR = "LE";
            break;
         case ">=":
            compIR = "LT";
            break;
         case "=":
            compIR = "NE";
            break;
         case "!=":
            compIR = "EQ";
            break;
         default:
            //shits broke yo
            compIR = null;
        }
      compIR += dataType; //append the dataType flag?
     }

   private void buildWhileHeader(String op1, String op2) {
      //Assumption: buildWhile is called when exitCond() listener method is called
      // IRCode to be built:
      // LABEL label#
      // [store immediate to register] (regNum++)
      // [evaluate condition with jump to label#++]
      // Note: may be issue with IR order if registers weren't ifninite.
      // Address if there's time

      ir_list.add("LABEL label" + labelNum++);

      //push new labelNum onto labelStack
      labelStack.add(0,"label" + labelNum);
      labelStack.add(0,"label" + (labelNum - 1));
      ir_list.add(compIR + " " + op1 + " " + op2 + " label" + labelNum++);
     }

   public void endWhile() {
      // IRCode to be built:
      // JUMP label#
      // LABEL label#++
      String op1 = "JUMP " + labelStack.remove(0);
      String op2 = "LABEL " + labelStack.remove(0);
      ir_list.add(op1);
```

```
    ir_list.add(op2);
   }

private void buildIfHeader(String op1, String op2) {
    //Assumption: buildIf is called when exitCond() listener method is called
    // IRCode to be built:
    // [store immediate to register] (regNum++)
    // [evaluate condition with jump to label#]
    // Add comparison instruction and push labelNum to stack

    labelStack.add(0,"label" + labelNum);
    ir_list.add(compIR + " " + op1 + " " + op2 + " label" + labelNum++);
}

public void enterElsePart() {
    // IRCode to be built(if needed):
    // JUMP label#
    // LABEL label#

    String elseLabel = labelStack.remove(0);
    labelStack.add(0, "label" + labelNum);
    ir_list.add("JUMP label" + labelNum++);
    ir_list.add("LABEL " + elseLabel);
   }

public void exitIf() {
    // IRCode to be built:
    // LABEL label#

    String elseLabel = labelStack.remove(0);
    ir_list.add("LABEL " + elseLabel);
}

/* Expression Builder Funcitonality
 * Conversion process:
 * expression --> postfix notation --> IR code
 *  ie. a + b -->     a  b  +      -->  addi a b r1
 */

public void enterExpression() {
    inExpression = true;
    stack.add(0,"(");
}

public void exitExpression(boolean end) {
    stack.add(0,")");
    popPostFixStack(end);
}

public void addElement(String element) {
   if (!inExpression) return;
   postfixOutput.add(element);
}

public void addOperator(String op) {
   if (!inExpression) return;
   String postfixOp;
   switch(op) {
      case "+":
      case "-":

if (stack.isEmpty() || stack.get(0) == "(")
      stack.add(0,op);
else {
// nothing is lower precedence than the two pop, push, and build
   postfixOp = stack.remove(0);
   postfixOutput.add(postfixOp);
   addOperator(op); //oh boy...
}
break;
   case "*":
   case "/":
   //check the top of the stack if same level of hierarchy
      if (stack.get(0) == "*" || stack.get(0) == "/") {
         postfixOp = stack.remove(0);
         postfixOutput.add(postfixOp);
         addOperator(op); //oh boy...
      } else { // for "+","-", or "("
            stack.add(0,op);
     }
     break;
     default:
     System.out.println("Shits broke yo...");
     }
}

public void printPostfix() {
   String stacky = "";
   //reverse the appearance of the stack for readability
   for (String el : stack) {
      stacky = el + " " + stacky;
   }
   for (String el : postfixOutput) {

   }
}

private void popPostFixStack(boolean completely) {
   if (completely) {
      if (postfixOutput.size() == 1) {
         stack.add(0,postfixOutput.get(0));
      }
      else {
         inExpression = false;
         while (!stack.isEmpty()) {
         // don't push parenthesis
         if (stack.get(0) == "(" || stack.get(0) == ")") {
            stack.remove(0);
            continue;
         }
         // pop and push
         postfixOutput.add(stack.remove(0));
         }
         exprToIR();
      }
         postfixOutput.clear();
      } else {
         stack.remove(0); //remove ")" tag
         while (stack.get(0) != "(") {
            postfixOutput.add(stack.remove(0));
        }
         stack.remove(0); //remove "(" tag
     }
  }

//if element is an immediate, then store to register
private String elementToIR(String el) {
   if (isNumber(el)) {
      dataType = el.contains(".") ? 'F' : 'I';
      ir_list.add(String.format("STORE%c %s $T%d",dataType,el,regNum++));
      el = "$T" + (regNum-1);
     }
   return el;
  }

// Converts postfixOutput array values into IR code
private void exprToIR() {
   // repurposing stack to work with IR building
   stack.clear(); //superfluous action. should already be empty

   // set dataType
   if (isNumber(postfixOutput.get(0))) {
            //check if it floats
      dataType = postfixOutput.get(0).contains(".") ? 'F' : 'I';
   } else { //get the data type from the variable
      dataType = currentTable.searchSymbol(postfixOutput.get(0)).getType().
      toUpperCase().toCharArray()[0];
  }

   //Build IR code
   String a;
   String b;
   for (String el : postfixOutput) {
      if (el.equals("+") || el.equals("-") || el.equals("*") || el.equals("/")) {
         b = elementToIR(stack.remove(0));
         a = elementToIR(stack.remove(0));
         switch (el) {
            case "+":
               stack.add(0,"$T"+regNum);
               ir_list.add(String.format("ADD%c %s $T%d",dataType,a,b,regNum++));
               break;
            case "-":
               stack.add(0,"$T"+regNum);
               ir_list.add(String.format("SUB%c %s $T%d",dataType,a,b,regNum++));
               break;
            case "*":
               stack.add(0,"$T"+regNum);
               ir_list.add(String.format("MUL%c %s %s $T%d",dataType,a,b,
               regNum++));
               break;
            case "/":
               stack.add(0,"$T"+regNum);
               ir_list.add(String.format("DIV%c %s %s $T%d",dataType,a,b,
               regNum++));
               break;
            default:
               System.out.println("Shits broke yo...");
           }
      }
      else {
         stack.add(0, el);
     }
  }
}

public void assignmentStatement(String variable) {
   //convert value if immediate
   dataType = currentTable.searchSymbol(variable).getType().toUpperCase()
   .toCharArray()[0];
   String value = elementToIR(stack.get(0));
   ir_list.add(String.format("STORE%c %s %s",dataType,value,variable));
   clean();
  }
}
```

tinyBuilder.java

```java
import java.util.*;

public class TinyBuilder {

private ArrayList<String> ir_list;
private SymbolTable table;
private ArrayList<String> tiny;
private ArrayList<String> vars;
private int regNum;
private char dataType;

public TinyBuilder(ArrayList<String> ir_list, SymbolTable table) {
   this.ir_list = ir_list;
   this.table = table;
   tiny = new ArrayList<>();
   vars = new ArrayList<>();
   regNum = 0; //for storing the temps
  }

public void print(boolean vertical) {
   printList(tiny,vertical);
  }

private void printList(ArrayList<String> list, boolean vertical) {
   String orient = vertical ? "\n" : " ";
   for (String el : list) {
      System.out.print(el + orient);
     }
   System.out.println();
  }

public void parseIrList() {
   //let the parsing games begin...
   for (String opcode : ir_list) {
      checkForVariable(opcode);
      opcode = opcode.replace("$T","r");
      String[] ops = opcode.split(" ");
      dataType = ops[0].toLowerCase().charAt(ops[0].length() - 1);
      dataType = dataType == 'f' ? 'r' : dataType;

      //route instruction
      if (opcode.contains("LABEL")) parseLabel(opcode);
      else if (opcode.contains("STORE")) parseStore(opcode);
      else if (opcode.contains("ADD")) parseAdd(opcode);
      else if (opcode.contains("SUB")) parseSub(opcode);
      else if (opcode.contains("MUL")) parseMul(opcode);
      else if (opcode.contains("DIV")) parseDiv(opcode);
      else if (opcode.contains("JUMP")) parseJump(opcode);
      else if (opcode.contains("LINK")) {}
      else if (opcode.contains("READ")) parseRead(opcode);
      else if (opcode.contains("WRITE")) parseWrite(opcode);
      else if (opcode.contains("RET")) parseRet(opcode);
      else parseComp(opcode); // I think that's all...
     }
   }

private void checkForVariable(String opcode) {
   String[] ops = opcode.split(" ");
   for (String potential : ops) {
      Symbol sym = table.searchSymbol(potential);
      if (sym != null && !vars.contains(potential)) {
         foundVariable(sym);
        }
     }
   }

private void foundVariable(Symbol var) {
   String variable = var.getName();
   String type = "var";
   vars.add(variable);
   if (var.getType().equals("STRING")) {
      variable += String.format(" %s",var.getValue());
      type = "str";
     }
   tiny.add(0,String.format("%s %s",type,variable));
   }

// LABEL [label]
private void parseLabel(String code) {
   String[] ops = code.split(" ");
   tiny.add(String.format("label %s",ops[1]));
   }

// STORE[T] op1 op2
private void parseStore(String code) {
   code = code.replace("$T","r");
   String[] ops = code.split(" ");
   //we'll see if we need to deal with variable checking...
   //tiny.add(String.format("%s matches register?
%b",ops[1],ops[1].matches("r\\d+")));
   //tiny.add(String.format("%s matches register?
%b",ops[2],ops[2].matches("r\\d+")));
   if (!ops[1].matches("r\\d+") && !ops[2].matches("r\\d+")) {
      tiny.add(String.format("move %s r0",ops[1])); //use temp register r0
      tiny.add(String.format("move r0 %s",ops[2]));
   } else {
        tiny.add(String.format("move %s %s",ops[1],ops[2]));
       }
    }

// COMP[T] op1 op2 label#
private void parseComp(String code) {
    code = code.replace("$T","r");
    String[] ops = code.split(" ");
    String compop = "j" + ((String)ops[0].subSequence(0,ops[0].
    length() - 1)).toLowerCase();
    if (!ops[1].matches("r\\d+") && !ops[2].matches("r\\d+")) {
        tiny.add(String.format("move %s r0",ops[2])); //use temp register r0
        tiny.add(String.format("cmp%c %s r0",dataType,ops[1]));
    } else {
        tiny.add(String.format("cmp%c %s %s",dataType,ops[1],ops[2]));
       }
    tiny.add(String.format("%s %s",compop,ops[3]));
   }

// ADD[T] r1 r2 r3
private void parseAdd(String code) {
    code = code.replace("$T","r");
    String[] ops = code.split(" ");
    tiny.add(String.format("move %s %s",ops[1],ops[3]));
    tiny.add(String.format("add%c %s %s",dataType,ops[2],ops[3]));
   }

// SUB[T] r1 r2 r3
private void parseSub(String code) {
    code = code.replace("$T","r");
    String[] ops = code.split(" ");
    tiny.add(String.format("move %s %s",ops[1],ops[3]));
    tiny.add(String.format("sub%c %s %s",dataType,ops[2],ops[3]));
   }

// MULT[T] r1 r2 r3
private void parseMul(String code) {
    code = code.replace("$T","r");
    String[] ops = code.split(" ");
    tiny.add(String.format("move %s %s",ops[1],ops[3]));
    tiny.add(String.format("mul%c %s %s",dataType,ops[2],ops[3]));
   }

// DIV[T] r1 r2 r3
private void parseDiv(String code) {
    code = code.replace("$T","r");
    String[] ops = code.split(" ");
    tiny.add(String.format("move %s %s",ops[1],ops[3]));
    tiny.add(String.format("div%c %s %s",dataType,ops[2],ops[3]));
   }
```

```
// JUMP label#
private void parseJump(String code) {
   String[] ops = code.split(" ");
   tiny.add(String.format("jmp %s",ops[1]));
  }

// READ[T] op1
private void parseRead(String code) {
   String[] ops = code.split(" ");
   tiny.add(String.format("sys read%c %s",dataType,ops[1]));
  }

// WRITE[T] op1
private void parseWrite(String code) {
   String[] ops = code.split(" ");
   tiny.add(String.format("sys write%c %s",dataType,ops[1]));
  }

// RET
private void parseRet(String code) {
   String[] ops = code.split(" ");
   tiny.add("sys halt");
  }
}
```

Design Pattern

We used the Abstract Design Pattern when creating the Listener.java class. This class extends junkBaseListener.java which implements junkListener.java. Both of these are generated by the org.antlr.v4 library that is being implemented in our program. We decided to utilize this pattern instead of coding this from scratch because Listener.java mainly overrides the methods created by org.antlr.v4 in junkBaseListener.java. In short, using the auto-generated methods as overrides allowed us to focus on the code for the SymbolTable.java and Symbol.java classes, which we felt was a better use of our time.

Technical Paper

Introduction

The purpose of this project is to demonstrate practical knowledge of the java programming language, regular expressions, finite automata, software engineering and design, as well as general problem solving skills in a group dynamic. Throughout this project we have applied knowledge acquired during our undergraduate studies to design, build, test and deploy a working compiler for a LITTLE language.

A working compiler, for even a LITTLE language, is a positive for future endeavors and job searches. It serves to show that we not only understand how to program but that we also understand the foundations of the computer science field. By completing this project, we have gained insight into each of the elements that make up computer science and how they fit together to create a finished product.

In the course of this report we will touch on the background and purpose of a compiler, each of the components that make up a compiler and a summary of what we learned, what we could have done better, and where we will go from here.

Background

In general terms, a compiler takes instructions, written in a specific programming language, and translates them into a machine language also known as an assembly language. This translation is necessary because a computer processor cannot ‘read’ instructions written in a programming language. The processor needs instructions written in a very basic language. This is why a compiler is necessary for any, non-assembly, programming language.

A compiler is made up of several different sub-routines or steps. The first of those steps is scanning, the second is parsing, the third is creating a symbol table, the fourth is token conversion using a semantic routine and the fifth is optimization which leads to a functional compiler.

A scanner reads character input, in this case a computer program, and converts the incoming characters into tokens that are then read by the parser using a grammar made up of regular expressions.

> The scanner represents an interface between the source program and the syntactic analyzer or parser. The scanner, through a character-by-character examination of the input text, separates the source program into pieces called tokens which

represent the variable names, operators, labels, and so on that comprise the source program[1].

This process is known as a lexical analysis. Should the scanner not recognize the character sequence being read, it will return ‘invalid’ and not convert that sequence into a token. In this case, the input or the grammar will need to be corrected so that each character sequence is converted into a valid token. Breaking the language up into smaller pieces speeds up the parsing step.

The second step is parsing the tokens using the defined grammar for the language. During this step a data structure, usually a parse tree, is created using the input. The parser looks at the token and compares it to the language grammar. If the token is a part of the grammar it is accepted. If it is not a part of the grammar it is rejected. There are two common parsing implementations, top-down and bottom-up parsing.

Top-Down Parsing: Involves searching a parse tree to find the left most derivations of an input stream by using a top-down expansion. Examples include LL parsers and recursive-descent parsers.
Bottom-Up Parsing: Involves rewriting the input back to the start symbol. This type of parsing is also known as shift-reduce parsing. One example is a LR parser.[2]

Regardless of the parser type, a parse tree is built to represent the variables and functions being tokenized by the parser. The parse tree is used in the next step.

The third step is creating a symbol table. The parse tree built in step two is traversed and symbol tables are built based on the tokens encountered at each of the nodes. An updated symbol table is built for each new node.

A symbol table may serve the following purposes depending upon the language in hand:

- To store the names of all entities in a structured form at one place.
- To verify if a variable has been declared.
- To implement type checking, by verifying assignments and expressions in the source code are semantically correct.
- To determine the scope of a name (scope resolution).

[1] http://what-when-how.com/compilier-writing/scanners-compiler-writing-part-1/
[2] https://www.techopedia.com/definition/3854/parser

> A symbol table is simply a table which can be either linear or a hash table.[3]

As each new node is encountered an updated symbol table is created and stored. The outer table is known as the global symbol table. The inner tables are scope symbol tables. The global table represents the structure of the program. In other words, it contains global methods and global variables that can be used by any part of the program. The scope symbol tables represent the commands that make up the global methods and local variables. The scope tables can be made up of many sub-tables representing the layers of a method.

The fourth step in building a compiler is analyzing the semantics of the language. The parser created a parse tree by turning blocks of words into tokens. The parser did not check that the blocks were properly structured or ordered. The semantic routines make sure the structure of the program is valid.

In English, verbs, nouns and adjectives are the building blocks of a sentence. Grammar is the order those blocks should have. For example, "Green run fox." is a sentence made up of a noun, a verb, and an adjective but it is not a grammatically correct sentence. The order is wrong and therefore the sentence makes no sense. In the same way semantic routines make sure that the tokens created by the parser are in the proper syntax.

> Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.[4]

If the tokens are not in the proper syntax an error will occur and the token will be rejected. With most compilers a syntax error will result in a failed build and the code will need to be corrected before the program can be successfully compiled.

The final step creates an intermediate code, IR for short, from the original source code. The intermediate code can be either high level or low level. A high level IR code is semantically close to the original code. A low level IR code is closer to the assembly code the compiler should generate. Some compilers process the IR code multiple times until the high level IR is turned into a low level IR.

A three address code is created from the IR code. The three address code uses no more than three memory locations to calculate each function. Should all memory locations be full

[3] https://www.tutorialspoint.com/compiler_design/compiler_design_symbol_table.html

[4] https://www.tutorialspoint.com/compiler_design/compiler_design_semantic_analysis.html

variables that are not currently being used will be stored and their memory location will be reassigned.

Once the three address code is completed an optimization, if desired, can be performed. Optimizing the code is typically done to remove redundant computations and reduce memory slots needed. This results in faster run times with less memory required. The three address code can be read by the computer and the computations written in the original programming language will be carried out. The compiler is functional and complete.

Methods and Discussion

Scanner

To begin, we looked at many options for implementing the given grammar and reading, we decided on a strategy that would simplify development. We determined that Java was the programming language we are the most comfortable with, so we decided to use it. While researching ANTLR, we saw that setting it up in an IDE, Integrated Development Environment, was complex and IDE specific.

Given our familiarity with Linux and OSX we decided to do all of our development in the terminal using a text editor. This made running the code simple with no extra IDE configuration. Given the differences between Windows and Linux shell commands, we decided to use Linux as our default environment for creating, running and testing our bash scripts. So far, no issues have been found when using OSX to run and test our bash scripts. However, if issues arise later in development Linux will become the only operating system used.

After researching how to use Antlr and run the scanner in general, we ran into few difficulties. Some minor grammatical errors and difficulties with regular expressions were the worst issues we ran into getting our grammar set up. At first, getting everything to communicate with Antlr properly caused some minor trouble. Running demonstration files, when they were available, helped us work through those difficulties. It was the demonstration files and what we learned from them that led us to use Linux and OSX rather than Windows for our development.

Parser

Implementing the parser provided in the ANTLR4 library was a very straight forward process. The Driver class was updated to invoke the ANTLR parser method, add buffer stream to read the input grammar, and add a method to store the parse tree. Having smoothed out the ANTLR4 integration during the first step, the only difficulty we encountered was finding the appropriate method call for the parser.

Using the Linux and OSX environment has posed no difficulties, so we will continue to develop our application using them.

Symbol Table

Several different options were discussed and debated for storing the symbol table information. Early in the design process, stacks and lists were discarded as being difficult to keep track of, especially as they grew. The structure options were narrowed down to either a tree structure or a 'routing table' structure. Each option was easy to build and access allowing us to visualize what the tree walker function was actually doing. In theory, this should have made testing and debugging the code an easier process.

In reality, creating the table after walking the parse tree was laborious. Debugging and testing required a lot of thought in order to see which part of the method was failing. To debug the symbol table methods a system out print call was included to print the output of the method. This let us see what each method was sending to the symbol table.

Once we knew what each method was sending to the table we were able to adjust the code to achieve the correct output. This required patience to get the correct output for each method. Once each method worked correctly on it's own the task of getting the methods to work together began. Starting with two methods the code was adjusted until, as a unit, they presented the correct output, then another method was added and adjusted. Eventually, all of the methods were fine tuned and the output going to the table was correct.

Semantic Routines

This step was the most challenging. Generating the IR code from the parse tree was difficult. Some of the difficulty arose from the design choices we made within the Listener Class.

To reduce time complexity and space needs we opted to use hashmaps to store symbols inside of the symbol tables. This design choice allowed us to keep much of the key functionality for generating the symbol tables inside the Listener class. The drawback to this design choice was building the IR code. The programming code to generate the IR code in our compiler became too complex to leave inside the Listener class.

To account for that we moved created a separate IRBuilder class and altered the methods in the Listener class to call the IRBuilder class. This also caused some of the functionality of the Listener class to become redundant and unnecessary. Rather than having the methods in the Listener class keep track of the method and context of the data that was being sent to the IRBuilder class, we were able to send chunks of data to the IRBuilder class where it could be processed by the appropriate method and returned to the caller.

Both use id_list non-terminal method which also refers to ids containing the functionality of handling SymbolTable behavior along with other things. Trying to add behavior at the enterId() listener would lead to confusing code, be redundant, and be time consuming.

Full-fledged Compiler

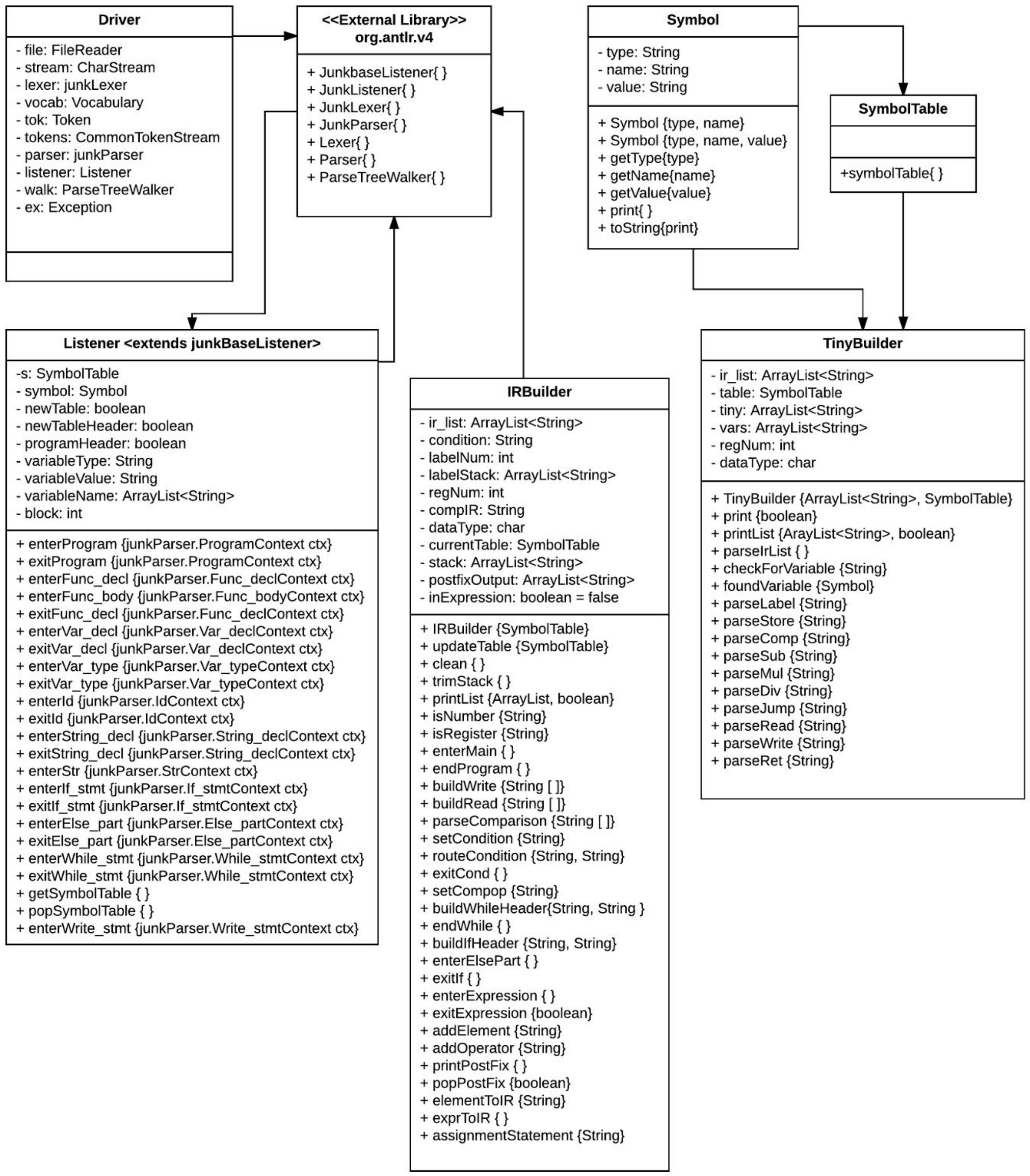
After working through generating the IR code we were able to focus on converting it to the assembly language we wanted. We decided to use a stack and an array to hold the IR code that was being passed in. This proved to be difficult to implement.

To account for order of precedence when dealing with add, subtract and multiply commands we needed to be able to pop items off of the stack and into the arraylist in the proper order. The stack design does not allow for pulling items out of the middle of the stack, so another way to deal with that had to be found. Instead of pre-sorting items going into the stack, we opted to use arraylist data structures to hold characters being read.

By using if/else statements and switches we were able to sort the incoming data into elements and operators and assign order of precedence for the operators. Once the functionality for sorting the characters was figured out completing the compiler came down to testing and debugging. This took more time than expected. When the output error was universal tracking down the issue was pretty straight forward. When the error was more random tracking down the issue was laborious.

Throughout the project we used Github for version control and sharing. This allowed us to make changes while debugging at each step without fear of making the issue worse. That was especially true while completing final step to achieve a full-fledged compiler. The hard work and testing paid off when we finally worked out the bugs and completed our compiler.

Working UML

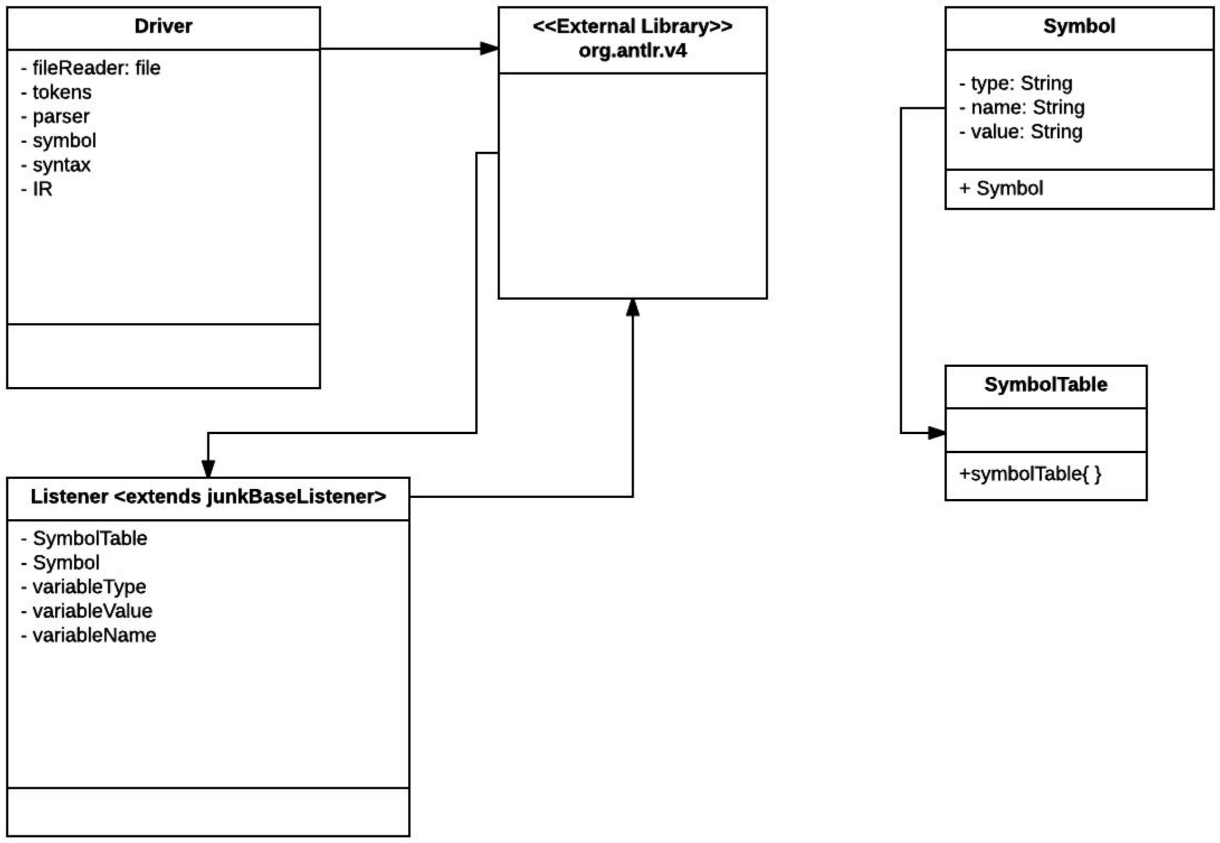

Driver
- file: FileReader
- stream: CharStream
- lexer: junkLexer
- vocab: Vocabulary
- tok: Token
- tokens: CommonTokenStream
- parser: junkParser
- listener: Listener
- walk: ParseTreeWalker
- ex: Exception
<<External Library>> org.antlr.v4
+ JunkbaseListener{ }
+ JunkListener{ }
+ JunkLexer{ }
+ JunkParser{ }
+ Lexer{ }
+ Parser{ }
+ ParseTreeWalker{ }
Symbol
- type: String
- name: String
- value: String
+ Symbol {type, name}
+ Symbol {type, name, value}
+ getType{type}
+ getName{name}
+ getValue{value}
+ print{ }
+ toString{print}
SymbolTable
+symbolTable{ }
Listener <extends junkBaseListener>
-s: SymbolTable
- symbol: Symbol
- newTable: boolean
- newTableHeader: boolean
- programHeader: boolean
- variableType: String
- variableValue: String
- variableName: ArrayList<String>
- block: int
+ enterProgram {junkParser.ProgramContext ctx}
+ exitProgram {junkParser.ProgramContext ctx}
+ enterFunc_decl {junkParser.Func_declContext ctx}
+ enterFunc_body {junkParser.Func_bodyContext ctx}
+ exitFunc_decl {junkParser.Func_declContext ctx}
+ enterVar_decl {junkParser.Var_declContext ctx}
+ exitVar_decl {junkParser.Var_declContext ctx}
+ enterVar_type {junkParser.Var_typeContext ctx}
+ exitVar_type {junkParser.Var_typeContext ctx}
+ enterId {junkParser.IdContext ctx}
+ exitId {junkParser.IdContext ctx}
+ enterString_decl {junkParser.String_declContext ctx}
+ exitString_decl {junkParser.String_declContext ctx}
+ enterStr {junkParser.StrContext ctx}
+ enterIf_stmt {junkParser.If_stmtContext ctx}
+ exitIf_stmt {junkParser.If_stmtContext ctx}
+ enterElse_part {junkParser.Else_partContext ctx}
+ exitElse_part {junkParser.Else_partContext ctx}
+ enterWhile_stmt {junkParser.While_stmtContext ctx}
+ exitWhile_stmt {junkParser.While_stmtContext ctx}
+ getSymbolTable { }
+ popSymbolTable { }
+ enterWrite_stmt {junkParser.Write_stmtContext ctx}
IRBuilder
- ir_list: ArrayList<String>
- condition: String
- labelNum: int
- labelStack: ArrayList<String>
- regNum: int
- compIR: String
- dataType: char
- currentTable: SymbolTable
- stack: ArrayList<String>
- postfixOutput: ArrayList<String>
- inExpression: boolean = false
+ IRBuilder {SymbolTable}
+ updateTable {SymbolTable}
+ clean { }
+ trimStack { }
+ printList {ArrayList, boolean}
+ isNumber {String}
+ isRegister {String}
+ enterMain { }
+ endProgram { }
+ buildWrite {String [ ]}
+ buildRead {String [ ]}
+ parseComparison {String [ ]}
+ setCondition {String}
+ routeCondition {String, String}
+ exitCond { }
+ setCompop {String}
+ buildWhileHeader{String, String }
+ endWhile { }
+ buildIfHeader {String, String}
+ enterElsePart { }
+ exitIf { }
+ enterExpression { }
+ exitExpression {boolean}
+ addElement {String}
+ addOperator {String}
+ printPostFix { }
+ popPostFix {boolean}
+ elementToIR {String}
+ exprToIR { }
+ assignmentStatement {String}
TinyBuilder
- ir_list: ArrayList<String>
- table: SymbolTable
- tiny: ArrayList<String>
- vars: ArrayList<String>
- regNum: int
- dataType: char
+ TinyBuilder {ArrayList<String>, SymbolTable}
+ print {boolean}
+ printList {AraryList<String>, boolean}
+ parseIrList { }
+ checkForVariable {String}
+ foundVariable {Symbol}
+ parseLabel {String}
+ parseStore {String}
+ parseComp {String}
+ parseSub {String}
+ parseMul {String}
+ parseDiv {String}
+ parseJump {String}
+ parseRead {String}
+ parseWrite {String}
+ parseRet {String}

Conclusion and Future Work

While we were able to keep our code to a minimum and conserve space, we did not have enough time to optimize the IR code. A future endeavor will be to write methods for reducing redundant operations when converting the IR code to the three address code. By reducing redundant operations we can reduce the number of memory registers needed to store variables and results. This also reduces needing to spill registers when creating the final assembly code.

In conclusion, this project brought together many facets of the computer science discipline. Working in incremental steps allowed us to focus on the task at hand without being overwhelmed by the the final goal. Understanding how a compiler is built has uses outside of building another compiler. For example, reading, parsing and placing data it into tables is a useful tool.

UML


Driver
- fileReader: file
- tokens
- parser
- symbol
- syntax
- IR
<<External Library>>
org.antlr.v4
Symbol
- type: String
- name: String
- value: String
+ Symbol
SymbolTable
+symbolTable{ }
Listener <extends junkBaseListener>
- SymbolTable
- Symbol
- variableType
- variableValue
- variableName

Design Trade-offs

To reduce time complexity and space needs we opted to use hashmaps to store symbols inside of the symbol tables. This design choice allowed us to keep much of the key functionality for generating the symbol tables inside the Listener class. The drawback to this design choice was building the IR code. The programming code to generate the IR code in our compiler became too complex to leave inside the Listener class.

To account for that we moved created a separate IRBuilder class and altered the methods in the Listener class to call IRBuilder. This also caused some of the functionality of the Listener class to become redundant and unnecessary. Rather than having the methods in the Listener class keep track of the method and context of the data that was being sent to the IRBuilder class, we were able to send chunks of data to IRBuilder where it could be processed by the appropriate method and returned to the caller.

For example, the design trade-off is apparent when building the IR code for READ(arg1, arg2, … , argN) and WRITE(arg1, arg2, …, argN).

```
write_stmt → ‘WRITE’ ‘(‘ id_list ‘)’ ‘;’
read_stmt → ‘READ’  ‘(‘ id_list ‘)’ ‘;’
```

Both use id_list non-terminal method which also refers to ids containing the functionality of handling SymbolTable behavior along with other things. Trying to add behavior at the enterId() listener would lead to confusing code, be redundant, and be time consuming.

The id_list listener method suffers from the same, albeit slightly less confusing and time consuming, issue as id_list non-terminal. The additional logic and method state code that would need to be included to verify the context id_list is being used in would add to memory requirements. Instead the enterRead_stmt() and enterWrite_stmt() listeners in IRBuilder pass in the context for which id_list is being called.

Using the parse tree structure, the ids can be extracted from the id_list listener method and passed to the appropriate methods in IRBuilder to build the read and write statements in IR code. This makes the code easier to read and interpret because the functionality is located in one method. Less code and less memory needed seem like a good trade-off but pulling the ids from the id_list listener is not the natural flow for reading the grammar and did require more thought and testing to fully implement.

Software Development Life-Cycle Model

We used an agile approach to development in this project. Initially we determined who would be in charge of each portion of the project. Team member 1 was primarily responsible for writing the actual code and team member 2 was primarily responsible for writing the project report and portfolio. Both team members proof-read the code, the report and the portfolio.

At the beginning of each step the team met and determined a plan of attack for writing the code. Invariably, the data structures and tree-walking method were our primary focus for each step. Then at each lab we sat down and updated each other on what was accomplished, what still needed to be done and what difficulties were encountered. We then adjusted our strategy to address the difficulties and keep ourselves on track.

Splitting the workload and having each team member be primarily responsible for a portion of the project was highly advantageous. We were able to address issues quickly, review each other’s progress, and adjust our strategy quickly. As a result the team met each deadline and the set parameters for each step of the project.