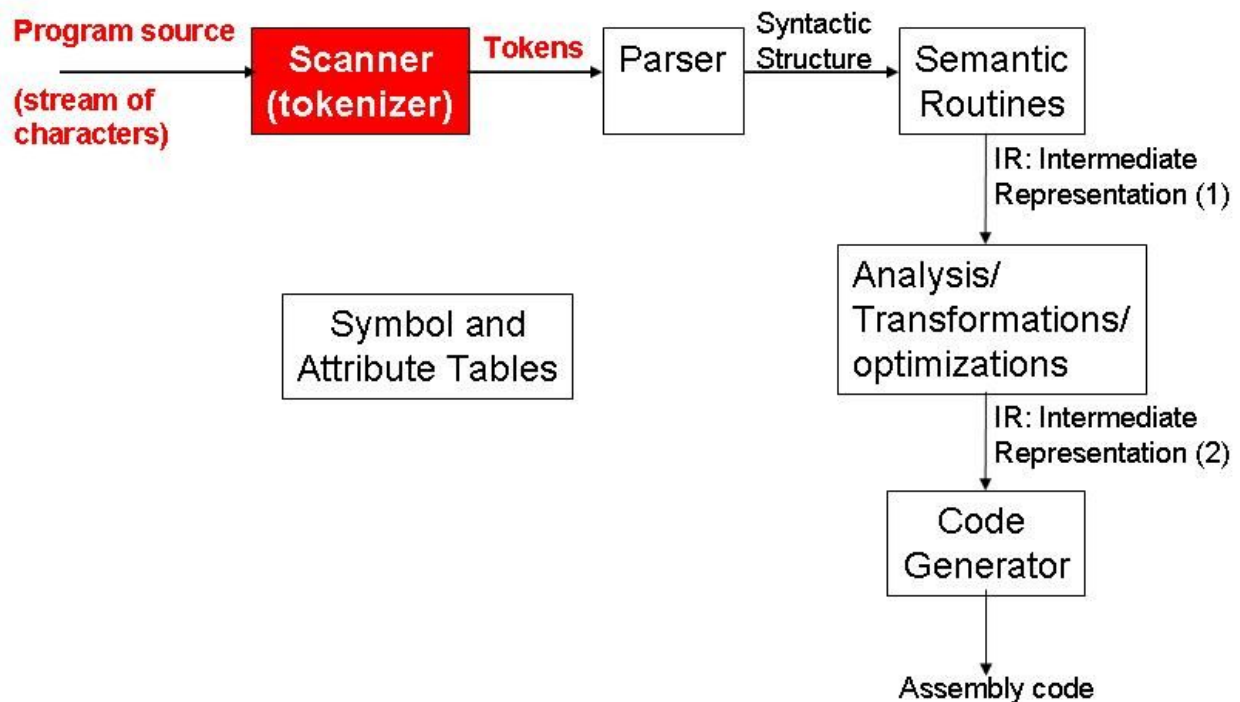


CSCI 468 Compilers
Spring 2017

Ryan Darnell
Stephanie McLaren

Program



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 $OP1 \geq OP2$ Goto LABEL)

LTOP1OP2LABEL (If $OP1 < OP2$ Goto LABEL)

LEOP1OP2LABEL (If $OP1 \leq OP2$ Goto LABEL)

NEOP1OP2LABEL (If $OP1 \neq OP2$ Goto LABEL)

EQOP1OP2LABEL (If $OP1 = OP2$ Goto LABEL)

JUMPLABEL (Direct Jump)

LABEL STRING (set a STRING Label)

READIRESLT

READFRESULT

WRITEIRESLT

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

```
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

```
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 infinite.
    // 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 Functionality
 * 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();
}
}

```

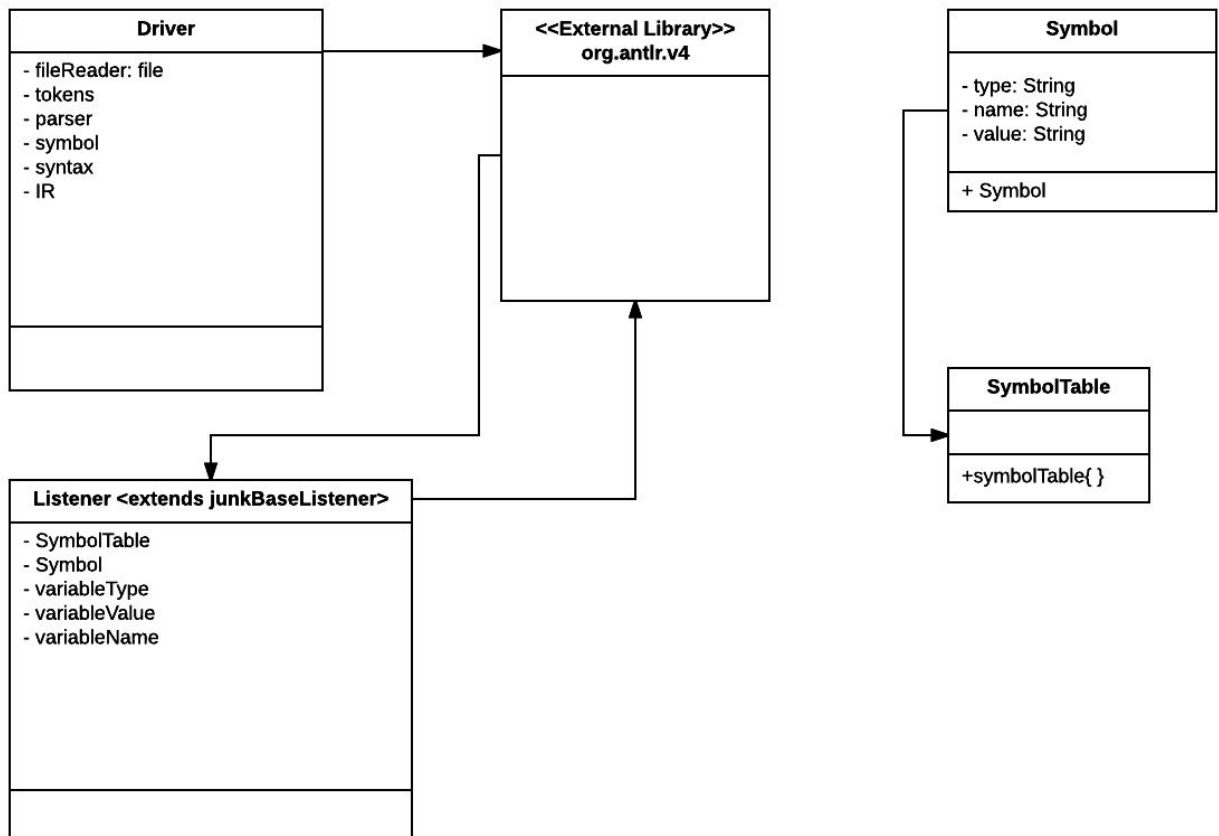
Design Pattern

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

This section is blank because our project was specified for us and did not require a technical paper.

UML



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 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.

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.