CS 454/554 Project 2

Team No Compile Errors
Nathan Patrizi
Del Jones
Tanner Evans

Introduction

This project tasked us with taking our program from project 1 and running it through various optimization procedures.

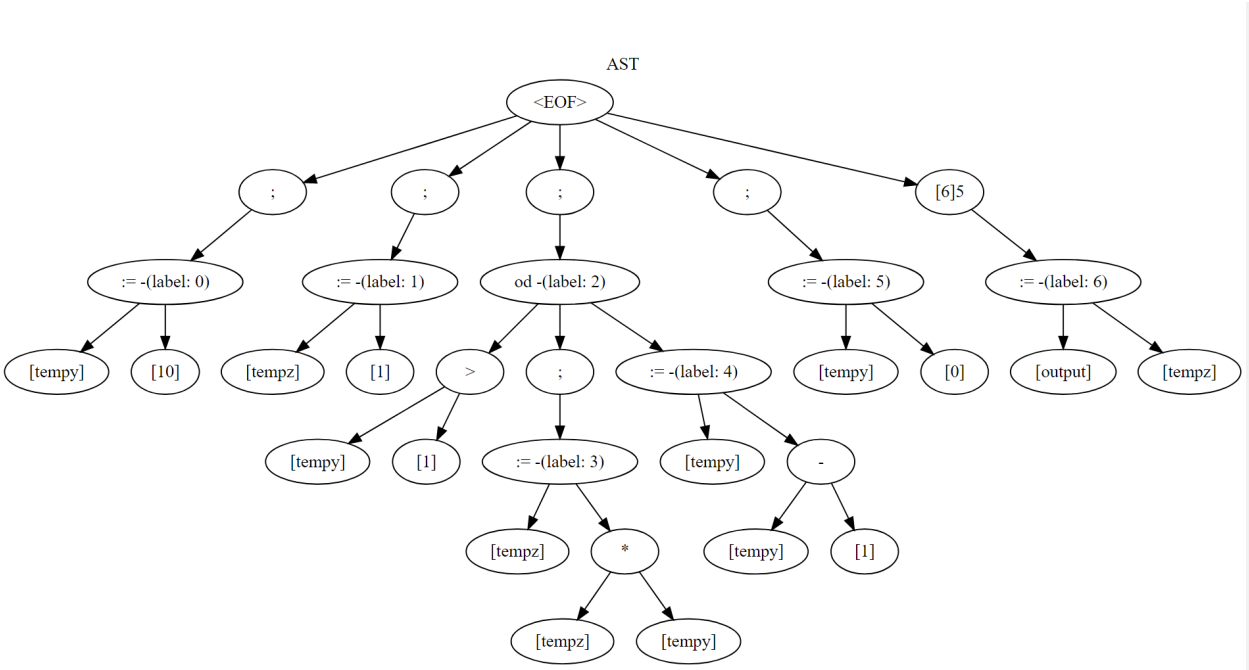
These tasks were split between the three members of the team. Labelled syntax and control flow graphs were handled by Nathan. Tanner handled reaching definitions, simple constant folding, liveness analysis, and dead code elimination. Register allocation and code generation was done by Del.

Labelled Syntax and Control Flow Graphs

Nathan Patrizi

Labeled AST and Source File Generation

The first two steps of project 2 after finishing project 1(which we had already completed) were to create a labeled AST, labeled file, and a Control Flow Graph(CFG). The first of these two tasks, the labeled AST creation was handled very similarly to project 1. This was easy to implement because in project 1 we had done all of our parsing using an actual parse tree, we already had created ways to make an AST from the parse tree we already utilized. Thus to accomplish the first goal of the labeled AST we simply added a check when a node was created in the dot file for if the expression was a boolean or an assignment operation as those were the statements that required labels. This allowed us to create a dot file that produced outputs in the format seen below.



With this labeled AST created we were also tasked with producing a labeled source file as well. This was simple to accomplish as we had already done the parsing required and would then just reconstruct the file based off of our parsed lists of expressions. This meant we had to adjust for the semantic requirements of the language when outputting but this was reasonable to do with some if statements. The annotated file for the AST above can be seen below along with the source files we used to accomplish both tasks as well.

```
Starting annotated file
tempy := 10;  --label: 0
tempz := 1;  --label: 1
while tempy>1 do --label: 2
tempz := tempz *tempy;  --label: 3
tempy := tempy -1  --label: 4
od
tempy := 0;  --label: 5
output := tempz  --label: 6
end annotated file
```

ASTNode.java

```
package DataStructureGeneration;

public class ASTNode<type> {
   public type getData() {
       return data;
   }

   public type[] getChildren() {
       return children;
   }

   public type getParent() {
       return parent;
   }

   type data;
   type [] children;
   type parent;

   public ASTNode(type data, type [] children, type parent){
       this.data = data;
       this.children = children;
       this.parent = parent;
   }


}
```

ASTBuilder.java

```
package DataStructureGeneration;

import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import static java.nio.file.StandardOpenOption.*;
import java.io.*;
import java.nio.file.*;

public class ASTBuilder {
   public static void  buildDotAST(Map<String, ASTNode<String>> nodeMap,
Map<Integer,String>visitOrder){
       String filename = "AST.txt";
       File file = new File(filename);
       List<String> open = new ArrayList<>();
       String [] nameIdentifiers = {"]1","]2","]3","]4","]5","]6","]7","]8","]9",};
       List<String> boolIdent = new ArrayList<>();

       boolIdent.add("od");
       boolIdent.add("fi");

       // Delete the existing dotfile
       try{
           boolean result = Files.deleteIfExists(file.toPath());
       }catch (IOException x){
           System.err.println(x);
       }
       // create the header for the dot file
       String s = """
           digraph "DirectedGraph" {
           graph [label = "AST", labelloc=t, concentrate = true];
           """;
       byte[] data = s.getBytes();
       Path p = Paths.get(filename);

       try (OutputStream out = new BufferedOutputStream(
               Files.newOutputStream(p, CREATE, APPEND))) {
           out.write(data, 0, data.length);
       } catch (IOException x) {
           System.err.println(x);
       }

       boolean found;
       //Correct Labels of node to show operation or operand
       Integer increment = 0;
       for (int i = 0; i < visitOrder.size(); i ++) {
           ASTNode<String> entry = nodeMap.get(visitOrder.get(i));
           String[] activeChildren = entry.getChildren();
           for (String temp: activeChildren){
               found = false;
               for (String nameIdentifier : nameIdentifiers) {
```

```
                    if (temp.trim().contains(nameIdentifier)) {
                        found = true;
                    }
                }
                if(!found) {
                    boolean isLabelled = false;
                    //System.out.println("entry test: " +temp);
                    if(temp.trim().contains(":=") || boolIdent.contains(temp.trim())){
                        isLabelled = true;
                    }

                    if(isLabelled) {//check for assignment or booloean and add the label
field to those
                        s = "\"" + entry.getData() + "\"" + " [ label=\"" + temp + " -(label:
" + increment++ + ")" + "\" ]" + "\n";
                        data = s.getBytes();
                        try (OutputStream out = new BufferedOutputStream(
                                Files.newOutputStream(p, CREATE, APPEND))) {
                            out.write(data, 0, data.length);
                        } catch (IOException x) {
                            System.err.println(x);
                        }
                        //default labeling with no specified label
                    }else{
                        s = "\"" + entry.getData() + "\"" + " [ label=\"" + temp + "\" ]" +
"\n";
                        data = s.getBytes();
                        try (OutputStream out = new BufferedOutputStream(
                                Files.newOutputStream(p, CREATE, APPEND))) {
                            out.write(data, 0, data.length);
                        } catch (IOException x) {
                            System.err.println(x);
                        }
                    }
                }
            }
        }
        //For each node in the tree print the relation it has to its children
        for (int i = 0; i < visitOrder.size(); i ++) {
            ASTNode<String> entry = nodeMap.get(visitOrder.get(i));
            String[] activeChildren = entry.getChildren();

            for (String temp: activeChildren){
                found = false;
                for (String nameIdentifier : nameIdentifiers) {
                    if (temp.trim().contains(nameIdentifier)) {
                        found = true;
                    }
                }
                if(found) {
                    s = "\"" + entry.getData() + "\"" + " -> " + "\"" + temp + "\"" + "\n";
                    data = s.getBytes();
```

```
                    try (OutputStream out = new BufferedOutputStream(
                            Files.newOutputStream(p, CREATE, APPEND))) {
                        out.write(data, 0, data.length);
                    } catch (IOException x) {
                        System.err.println(x);
                    }
                }
            }
        }


        //create the end of the dotfile
        s = "\n}";
        data = s.getBytes();
        try (OutputStream out = new BufferedOutputStream(
                Files.newOutputStream(p, CREATE, APPEND))) {
            out.write(data, 0, data.length);
        } catch (IOException x) {
            System.err.println(x);
        }


    }

    public static void createAnnotatedFile(Map<String, operandNode<String>> operandMap,
Map<String, ASTNode<String>> nodeMap, List<String> order){
    System.out.println("Starting annotated file");
    String [] branches = {"w=", "w>=", "w<=", "w<", "w>" , "i=",  "i<=", "i<", "i>" , "i>="};
    String [] nameIdentifiers = {"]1","]2","]3","]4","]5","]6","]7","]8","]9",};
    String [] semicolonChecks = {"od", "fi", "else"};
    String [] operators = {"+", "-", "*", "or", "and"};
    List<String> line = new ArrayList<>();
    Integer label = 0;
    int count = 0;
    for(String entry: order){

        if(entry.trim().equals("else")){
            System.out.println("else");
        }else if (entry.trim().contains("skip")) {
            System.out.println("skip" + " --label " + label++);
        }else{
            String expr = operandMap.get(entry).data;
            Boolean containsBranch = false;
            for (String testBranch : branches) {
                if (expr.contains(testBranch)) {
                    containsBranch = true;
                    if (expr.contains("]w")) { //while printing
                        expr = expr.replaceAll(testBranch, testBranch.substring(1));
                        expr = expr.replaceAll("]", "");
                        expr = expr.replaceAll("\\[", "");
                        System.out.println("while " + expr + " do" + " --label: " + label++);
                    }
```

```
                    if (expr.contains("]i")) {
                        expr = expr.replaceAll(testBranch, testBranch.substring(1));
                        expr = expr.replaceAll("]", "");
                        expr = expr.replaceAll("\\[", "");
                        System.out.println("if " + expr + " then" + " --label: " + label++);
                    }
                    line.clear();
                    break;

                }
            }

            if (expr.contains(":=")) {
                String[] split = expr.split(":=");
                if (line.isEmpty()) {
                    line.add(split[1].trim());
                }
                line.add(0, split[0].trim());
                line.add(1, ":=");
                if(count < order.size()-1) {
                    Boolean needSemicolon = true;
                    //System.out.println("count: " + count + " next expr: " +
operandMap.get(order.get(count+1)).data);
                    for(int i = 0; i < semicolonChecks.length; i++) {
                        if(!order.get(count+1).contains("skip") &&
!order.get(count+1).trim().equals("else")) {
                            if (operandMap.get(order.get(count +
1)).data.contains(semicolonChecks[i])) {
                                needSemicolon = false;
                                break;
                            }
                        }
                    }
                    if(needSemicolon) {
                        String testing = line + ";" + "  --label: " + label++;
                        testing = testing.replaceAll("]","");
                        testing = testing.replaceAll("\\[","");
                        testing = testing.replaceAll(",","");
                        System.out.println(testing);
                    }else{
                        String testing = line + "  --label: " + label++;
                        testing = testing.replaceAll("]","");
                        testing = testing.replaceAll("\\[","");
                        testing = testing.replaceAll(",","");
                        System.out.println(testing);
                    }
                }else{
                    String testing = line + "  --label: " + label++;
                    testing = testing.replaceAll("]","");
                    testing = testing.replaceAll("\\[","");
                    testing = testing.replaceAll(",","");
                    System.out.println(testing);
```

```
                }
                line.clear();
            }
            if (expr.contains("od")) {
                containsBranch = true;
                System.out.println("od");
            }
            if (expr.contains("fi")) {
                containsBranch = true;
                System.out.println("fi");
            }
            if (!containsBranch && !expr.contains(":=")) {
                String [] split = {};
                String tempOp = null;
                for(String operator: operators){

                    if(expr.contains(operator)){
                        String tempoperator = operator;
                        if(tempoperator.equals("+")){
                            tempoperator = "\\+";
                        }
                        if(tempoperator.equals("*")){
                            tempoperator = "\\*";
                        }
                        tempOp = operator;
                        split = expr.split(tempoperator);
                    }
                }
                Boolean containsNode = false;
                for(String name: nameIdentifiers){
                    //System.out.println("split 0 : " + split[0]+ " split 1: " + split[1]+ "
name: " + name);
                    if(split[0].contains(name)){
                        containsNode = true;
                        if(!split[1].contains(name)) {
                            String templine = tempOp + split[1];
                            line.add(templine);
                        }
                        break;
                    }
                    if(split[1].contains(name)){
                        containsNode = true;
                        if(split[0].contains(name)) {
                            String templine = split[0] + tempOp;
                            line.add(templine);
                        }
                        break;
                    }
                }
                if(!containsNode) {
                    //System.out.println("1: " + expr);
                    line.add(expr);
```

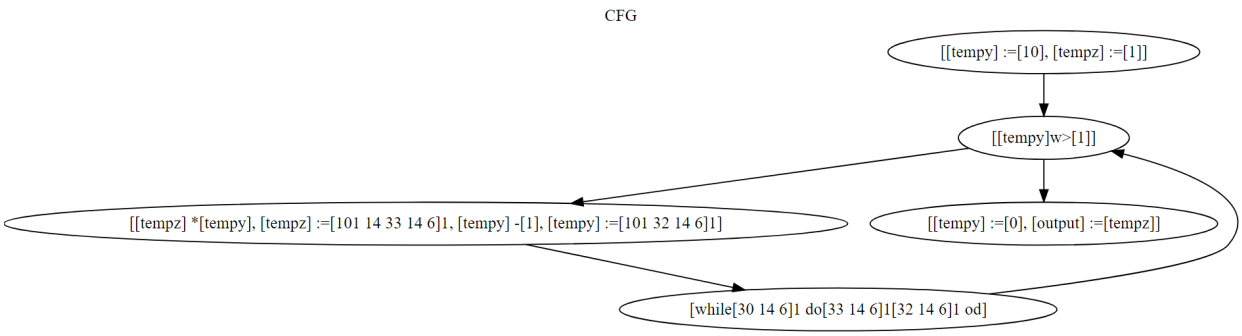
```
                }
            }
        }
        count++;
    }
        System.out.println("end annotated file");
        System.out.println("");
    }

}
```

CFG Creation

The second part of project 2 that I handled was the creation of the CFG. The CFG took in the list of expressions I had previously passed to the assembly generation and then created a CFG based off of those expressions. To accomplish this I created a CFG data structure which consisted of an integer identifier; a basic block which was a list of the expressions in that block; a set of edges to other blocks; and a list of expression identifiers relating to each expression in the basic block. To create the overall CFG we would have to add expressions to a basic block until we reach a branch and then handle the branch correctly. To understand how this was done I'll describe the desired behaviour of the CFG in the three basic cases: continuous execution; If then else branch; and a while do branch.

For the continuous execution, the task was fairly straight-forward as previously mentioned. Simply continuously add each expression to the current basic block along with their associated identifier, until a branch is seen. For an if then else branch, we would want to update the edges from the branch start node to point to the first block that appears after the start block along with the node that follows the else statement. This ensures the correct edges. Lastly, for a while do conditional, we should point to the next block as well but also the block that immediately follows the od block. This allows for the correct paths to be followed when the conditional is evaluated as either true or false.

We chose to create the CFG off of expressions rather than statements because this would allow us to maintain much of the same assembly generation code previously used. We had to make some adjustments to the optimizations sections which will be presented further in the report but having the CFG based on expressions allowed for the optimizations to be incorporated into our old code much easier than if we had chosen to use full statements. Below an example CFG can be seen. The basic block list of expressions are separated with commas and can contain other expression's identifiers when appropriate. These identifiers are unique to each expression and are used to identify and build the full statements later. Additionally, the relevant source code for CFG creation is included below as well.



CFG.java

```
package DataStructureGeneration;

import java.util.ArrayList;

public class CFG {
	Integer identifier;
	public ArrayList<String> basicBlock;
	public ArrayList<String> nodeNames;
	public ArrayList<Integer> edges;

	public CFG(Integer identifier, ArrayList<String> basicBlock, ArrayList<Integer>
edges, ArrayList<String> nodeNames){
	    this.identifier = identifier;
	    this.basicBlock = basicBlock;
	    this.nodeNames = nodeNames;
	    this.edges = edges;

	}

}
```

CFGBuilder.java

```
package DataStructureGeneration;

import java.io.BufferedOutputStream;
import java.io.File;
```

```
import java.io.IOException;
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.*;

import static java.nio.file.StandardOpenOption.APPEND;
import static java.nio.file.StandardOpenOption.CREATE;

public class CFGBuilder {

   private static Boolean checkBranches(String expr){
       String [] branchIdentifiers = {"w=", "w>=", "w<=", "w<", "w>" , "i=",  "i<=",
"i<","i>=", "i>" , "i>=", "od","fi", "else"};
       for(String identifier : branchIdentifiers) {
           if (expr.contains(identifier)) {
               return true;
           }
       }
       return false;
   }
   private static Boolean checkWhileStart(String expr){
       String [] branchIdentifiers = {"w=", "w>=", "w<=", "w<", "w>" };
       for(String identifier : branchIdentifiers) {
           if (expr.contains(identifier)) {
               return true;
           }
       }
       return false;
   }
   private static Boolean checkIfStart(String expr){
       String [] branchIdentifiers = { "i=",  "i<=",  "i>=", "i<", "i>"};
       for(String identifier : branchIdentifiers) {
           if (expr.contains(identifier)) {
               return true;
           }
       }
       return false;
   }

   public static Map<Integer, CFG> createCFG(Map<String, operandNode<String>> operandMap,
Map<String, ASTNode<String>> nodeMap, List<String> order){
       Map<Integer, CFG> controlGraph = new HashMap<>();
       Integer identifier = 0;
       ArrayList<String> temp = new ArrayList<>();
       ArrayList<String> tempName = new ArrayList<>();
       ArrayList<String> branches = new ArrayList<>();
       ArrayList<Integer> elseBlock = new ArrayList<>();
       //find basic blocks and add to graph
       //basic block is a continuous section of code without branches out of it
       //so the first block would be from the start until the first branch
```

```
        controlGraph.put(identifier,new CFG(identifier,new ArrayList<>(), new ArrayList<>(),
new ArrayList<>()));

        for (String s : order) {
            String expr = "";
            if ((!s.contains("else") || s.contains("fi")) && !s.contains("skip")) {
                expr = operandMap.get(s).data;
            } else {
                if (s.contains("skip")){
                    expr = "skip";
                }else{
                    expr = "else";
                }
            }
            if (!checkBranches(expr)) { //checks if expression is a branch
                // if expr isn't a branch we add it to the current basic block
                temp = controlGraph.get(identifier).basicBlock;
                temp.add(expr);
                tempName = controlGraph.get(identifier).nodeNames;
                tempName.add(s);
                controlGraph.put(identifier, new CFG(controlGraph.get(identifier).identifier,
temp, controlGraph.get(identifier).edges, tempName));
            } else {
                //add the branch statement to the DataStructureGeneration.CFG
                if (!expr.contains("else") || expr.contains("fi")) { //for an else we don't
actually want to create a new block
                    if (controlGraph.get(identifier).basicBlock.isEmpty()) { //check for back
to back end of while/if statements
                        branches.add(expr);
                        ArrayList<String> branchTest = new ArrayList<>();
                        branchTest.add(expr);
                        ArrayList<String> branchName = new ArrayList<>();
                        branchName.add(s);
                        controlGraph.put(identifier, new CFG(identifier, branchTest, new
ArrayList<>(), branchName));
                    } else {
                        identifier++;
                        branches.add(expr);
                        ArrayList<String> branchTest = new ArrayList<>();
                        branchTest.add(expr);
                        ArrayList<String> branchName = new ArrayList<>();
                        branchName.add(s);
                        controlGraph.put(identifier, new CFG(identifier, branchTest, new
ArrayList<>(), branchName));
                    }
                } else {
                    elseBlock.add(identifier + 1);
                }
                // move to next basic block
                identifier++;
                controlGraph.put(identifier, new CFG(identifier, new ArrayList<>(), new
ArrayList<>(), new ArrayList<>()));
```

```
            }
            //System.out.println("expression: " + expr + " node id: " + s);//+  " parent
node: " + nodeMap.get(order.get(i)).parent);

        }

        //System.out.println();
        //add edges for flow like with while and if statements
        for(String branch: branches){
            //System.out.println("Branch: " + branch);
        }
        //System.out.println();

        Stack<Integer> branchIdent = new Stack<>();
        Stack<Integer> ifIdent = new Stack<>();
        for(int i = 0; i < controlGraph.size(); i ++){
            ArrayList<Integer> edges = new ArrayList<>();
            //any branch will include the next node
            //System.out.println(controlGraph.get(i).basicBlock.size());
            if(controlGraph.get(i).basicBlock.size() > 0) {
                if (!controlGraph.get(i).basicBlock.get(0).contains("od")) { //end of a while
loop goes back to conditional

                    edges = controlGraph.get(i).edges;
                    if (i != controlGraph.size() - 1) {
                        edges.add(i + 1);
                    }
                } else {// if this is the end of a while loop we should pop from the stack
and add that to the edge set
                    Integer whileLoc = branchIdent.pop();
                    ///System.out.println("popped while: " + whileLoc);
                    edges.add(whileLoc);
                    // we should update the while start to include an edge to the next node
if it exists
                    ArrayList<Integer> tempEdges = controlGraph.get(whileLoc).edges;
                    tempEdges.add(i + 1);
                    controlGraph.put(whileLoc, new CFG(controlGraph.get(whileLoc).identifier,
controlGraph.get(whileLoc).basicBlock, tempEdges, controlGraph.get(whileLoc).nodeNames));
                }


                //System.out.println("testing: " + controlGraph.get(i).basicBlock.get(0));
                //System.out.println(checkIfStart(controlGraph.get(i).basicBlock.get(0)));
                //a branch can also go to the block after the branch occurs, this happens if
the initial condition isn't met.
                if (checkWhileStart(controlGraph.get(i).basicBlock.get(0))) { // Check if
this is the start of a branch
                    branchIdent.push(i); //push current identifier onto stack
                    //System.out.println("Check While: " + branchIdent);
                }
                if (checkIfStart(controlGraph.get(i).basicBlock.get(0))) { // Check if this
is the start of a branch
```

```
                    ifIdent.push(i); //push current identifier onto stack
                    //System.out.println("Check If: " + ifIdent);
                }

                if (controlGraph.get(i).basicBlock.get(0).contains("fi")) {
                    //the end of an if should go back and point to the first else block
                    //System.out.println("end of if");
                    Integer ifLoc = ifIdent.pop();
                    ArrayList<Integer> tempEdges = controlGraph.get(ifLoc).edges;
                    tempEdges.add(elseBlock.get(0));
                    //update if to include the else statement in its flow
                    controlGraph.put(ifLoc, new CFG(controlGraph.get(ifLoc).identifier,
controlGraph.get(ifLoc).basicBlock, tempEdges, controlGraph.get(ifLoc).nodeNames));
                    //update first block after if to include the current end block in its
edges
                    ArrayList<Integer> tempEdgestwo = new ArrayList<>();
                    tempEdgestwo.add(i);
                    controlGraph.put(elseBlock.get(0) - 1, new
CFG(controlGraph.get(elseBlock.get(0) - 1).identifier, controlGraph.get(elseBlock.get(0) -
1).basicBlock, tempEdgestwo,  controlGraph.get(elseBlock.get(0) - 1).nodeNames));
                    elseBlock.remove(0);
                }


                //put the new edges into the data structure
                controlGraph.put(i, new CFG(controlGraph.get(i).identifier,
controlGraph.get(i).basicBlock, edges, controlGraph.get(i).nodeNames));
            }
        }

/*
        for(int i  = 0; i < controlGraph.size(); i ++){
            System.out.println("Identifier: " + i);
            for(int j = 0; j < controlGraph.get(i).basicBlock.size(); j ++){
                System.out.println("Basic Block Expr #" + j + " : " +
controlGraph.get(i).basicBlock.get(j));
            }
            System.out.println("Edges: " + controlGraph.get(i).edges);
            System.out.println();
        }


*/
```

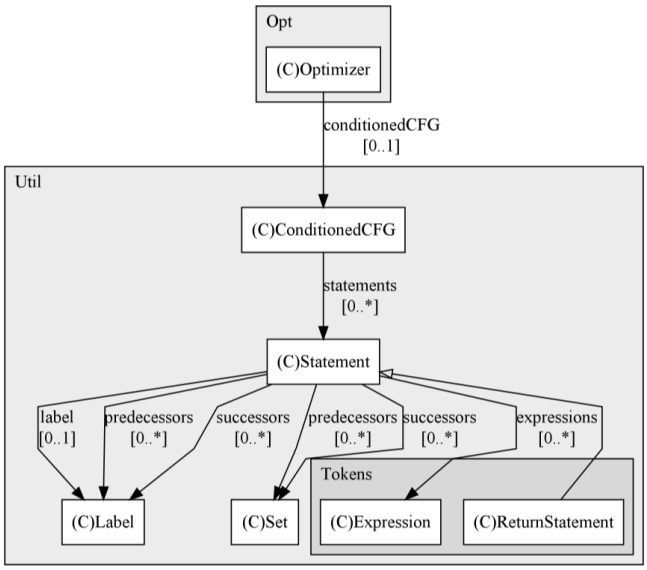
```
    return controlGraph;
}



public static void  buildDotCFG(Map<Integer, CFG> cfgMap){
    String filename = "CFG.txt";
    File file = new File(filename);

    // Delete the existing dotfile
    try{
        boolean result = Files.deleteIfExists(file.toPath());
    }catch (IOException x){
        System.err.println(x);
    }
    // create the header for the dot file
    String s = """
        digraph "DirectedGraph" {
        graph [label = "CFG", labelloc=t, concentrate = true];
        """;
    byte[] data = s.getBytes();
    Path p = Paths.get("CFG.txt");

    try (OutputStream out = new BufferedOutputStream(
            Files.newOutputStream(p, CREATE, APPEND))) {
        out.write(data, 0, data.length);
    } catch (IOException x) {
        System.err.println(x);
    }

    boolean found;
    //Correct Labels of node to show operation or operand
    for (int i = 0; i < cfgMap.size(); i ++) {
        CFG entry = cfgMap.get(i);
        s = "\"" + i + "\"" +" [ label=\""+ entry.basicBlock + "\" ]"+"\n";
        data = s.getBytes();
        try (OutputStream out = new BufferedOutputStream(
                Files.newOutputStream(p, CREATE, APPEND))) {
            out.write(data, 0, data.length);
        } catch (IOException x) {
            System.err.println(x);
        }
    }
    //For each node in the tree print the relation it has to its children
    for (int i = 0; i < cfgMap.size(); i ++) {
        CFG entry = cfgMap.get(i);

        for (int j = 0; j < entry.edges.size(); j++){
            s = "\"" + i + "\"" + " -> " + "\"" + entry.edges.get(j) + "\"" + "\n";
            data = s.getBytes();
            try (OutputStream out = new BufferedOutputStream(
```

```
                        Files.newOutputStream(p, CREATE, APPEND))) {
                        out.write(data, 0, data.length);
                } catch (IOException x) {
                        System.err.println(x);
                }

            }
        }

        //create the end of the dotfile
        s = "\n}";
        data = s.getBytes();
        try (OutputStream out = new BufferedOutputStream(
                Files.newOutputStream(p, CREATE, APPEND))) {
            out.write(data, 0, data.length);
        } catch (IOException x) {
            System.err.println(x);
        }


    }




}
```

Optimization

Tanner J. Evans

I spear-headed the optimization steps in this project, in part because Del and Nathan needed to do a lot more coding for the first project than me, and in part because I had some ideas about using Java's inheritance and polymorphism to separate the concern of how things are implemented from that which implements them. In particular, I built data structures to represent the elements in play during optimization, leaning heavily on abstract classes, subclassing, and default and overridden methods. In doing so, I was able to ensure that subclasses implemented a small set of low-level operations that both Java collection methods and my own encapsulating data structures could use to faithfully perform their duties. This ended up paying off quite well in moments when I discovered I had failed to include some support for crucial language features, enabling me to make some very minor changes to base-level data structures to factor in new functionality.



A simple class diagram for the Optimizer.

Functionally, I worked hard in this project to make my portions of the code as distinct as possible, and to ensure that implementing it within the larger structure of the project would require minimal change and addition to other portions of code. As such, I centralized my portions of the code into the Opt package. To optimize code, then, once a CFG was obtained, other portions of the code needed only to create an Optimizer object, feed it a CFG and a list of variables, request optimizations, and request the optimized CFG be returned to it.

Optimizer.java

```
package Opt;

import Opt.Util.ConditionedCFG;
import DataStructureGeneration.*;
import Opt.Util.Tokens.Variable;
import Opt.Util.Set;

import java.util.*;

public class Optimizer {
    private ConditionedCFG conditionedCFG;
    private final List<String> VARS;
    private String[] vars;

    public Optimizer(Map<Integer, CFG> cfg, List<String> vars) {
        vars.remove("true");
        vars.remove( "false");
        this.VARS = vars;
        Conditioner conditioner = new Conditioner(cfg);
        conditionedCFG = conditioner.conditionedCFG;
        conditionedCFG.printOut("Conditioned CFG:");
        this.vars = vars.toArray(new String[0]);
    }

    public void optimizeDefault() {
        foldConstants();
        conditionedCFG.printOut("After Constant Folding:");
        eliminateDeadCode();
    }

    public void foldConstants() {
        RDAnalyzer rd = new RDAnalyzer(conditionedCFG, vars);
        conditionedCFG = rd.fold();
    }

    public void eliminateDeadCode() {
        Set<Variable> liveVarsAtEnd = new Set<>();
        String outputVar = Data.outputVariable;
        if (!VARS.contains(outputVar)) {
            System.out.println("The provided output variable does not exist in code. Live Variable Analysis not " +
                               "allowed. Continuing without dead code elimination. Fix and try again for full " +
                               "optimization.");
        } else {
            liveVarsAtEnd.add(new Variable(outputVar));
            LivenessAnalyzer livenessAnalyzer = new LivenessAnalyzer(conditionedCFG, liveVarsAtEnd);
            conditionedCFG = livenessAnalyzer.getConditionedCFG();
            conditionedCFG.printOut("After Dead Code Elimination:");
        }
    }

    public Map<Integer, CFG> getOptimizedCFG() {
        return conditionedCFG.toOutputForm();
    }
}
```

We found that the original parsing failed to mark true and false as constants instead of variables, and the simplest solution seemed to be to let it do this and remove these as variables if they appear. Nathan's code featured a far more imperative than object-oriented style, so I quickly realized when trying to work with it that if I left it in the form given to me, manipulating it would require a huge amount of analysis and calculation to even redetermine where I had found an element, so my first decision was to convert the CFG into what I called a ConditionedCFG. Whereas the original CFG was a set of nodes with a series of strings for each line within that node, my ConditionedCFG is a Java object with a collection of Statement objects and a variety of methods for manipulating the CFG and extracting information. In general, this allowed me to mostly focus on the algorithms for analyses and code modification, while the ConditionedCFG handled the specifics of how to accomplish that on its contents.

Converting the original CFG format to my ConditionedCFG format required a fairly tricky algorithm. The original CFG's nodes were permitted to have any number of statements, and ended in the case of branching or annotation. Annotations were also placed in their own nodes, and expressions with subexpressions were split into multiple lines such that each line featured a Value Operator Value structure, with later lines referencing previous subexpressions through what I called Indirect values. In order to perform code analysis or modification with any sort of ease, I needed to extract a set of statements that contained an entire source statement's subexpressions. I accomplished this with a Conditioner object, which initializes an empty ConditionedCFG, and then iterates over the nodes and lines of the original CFG. As it goes, it parses each subexpression into its left value, operator, and right value, builds an Expression object with these values and various information needed for later manipulation, and then pushes that subexpression to a stack. When an expression encounters an operator that can be the root of an expression stack (assignment or conditional), a new Statement is created and the stack of subexpressions is popped and passed into it in the proper order. The data culled from the original CFG and given to Statements and Expressions is later used to build the ConditionedCFG's edges between statements. This was fairly complicated, because my ConditionedCFG needed to at times both split nodes and at others eliminate them entirely to build its own set of Statements, whose label numbers would not be equal to or map reasonably to the originals, so the edge transitions in the original nodes were not simply copyable. I ended up including original positional data with Statements and Expressions, and delaying edge production until the entire Statement list was populated.

Along the way I built many small object classes, most of which extend superclasses that larger objects hold. These small objects define their own basic methods for determining their equality to other objects, answers to questions about what their attributes and qualities are, and so forth. I will not explain each one in detail, but a very important one is Set. Set is a class with generic type parameters which extends LinkedList, and ensures that duplicate entries can not be added. Any type of objects that I place into a Set is forced to implement methods that allow me to call some very useful Java library functions, such as .equals(), and .contains() on collections. The most important part of this is that I defined a set of operations for Set that allows for direct translation from traditional set operations: .union(), .add(), .remove(), and .difference(). Getting these functioning properly here was small and simple, but it removed a huge amount of work from the analysis and optimization algorithms.

I will discuss analysis and optimization in the next sections, but I think it will be useful to include the source files for my various objects here. These are arranged roughly in order of importance, with some of the final classes being wrappers for other classes that greatly simplified some code in other sections.

Tokens and Utility Objects / Classes

Conditioner.java

```
package Opt;

import DataStructureGeneration.CFG;
import Opt.Util.*;
import Opt.Util.Pairs.*;
import Opt.Util.Tokens.*;
import Opt.Util.Operator.*;
import Opt.Util.Tokens.Expression.*;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.Map;
import java.util.Stack;

public class Conditioner {
    ConditionedCFG conditionedCFG;
    Stack<Expression> expressionStack = new Stack<>();
    Stack<Expression> annotationStack = new Stack<>();

    public Conditioner(Map<Integer, CFG> cfg) {
        conditionedCFG = new ConditionedCFG();
        convertToConditionedCFG(cfg);
        conditionedCFG.generateEdges(annotationStack);
    }

    public ConditionedCFG getConditionedCFG() { return conditionedCFG; }

    private void convertToConditionedCFG(Map<Integer, CFG> cfg) {
        // Loop over nodes i and lines j
        LinkedList<Expression> indirects = new LinkedList<>();
        for (int i = 0; i < cfg.size(); i++) {

            ArrayList<String> block = cfg.get(i).basicBlock;
            for (int j = 0; j < block.size(); j++) {

                String line = block.get(j);
                String name = cfg.get(i).nodeNames.get(j);
                String[] splitStr;
                OpEnum op = null;
                Expression expression;
                ExprType exprType = null;

                if (null != (op = splitIfAny(line, AssignOp.values(), CompareOp.values(), ArithOp.values(),
BoolOp.values()))) {
                    if (op == ArithOp.ADD) {
                        splitStr = line.split("\\+");
                    } else if (op == ArithOp.MUL) {
                        splitStr = line.split("\\*");
                    } else if (op.equals(BoolOp.OR)) {
                        splitStr =  line.split(" or\\[");
                        splitStr[1] = "[" + splitStr[1];
                    }
```

```
                    else if (op.equals(BoolOp.AND)) {
                        splitStr = line.split(" and\\[");
                        splitStr[1] = "[" + splitStr[1];
                    }
                    else if (op.equals(BoolOp.NOT)) {
                        splitStr = line.split("not\\(");
                        splitStr[1] = "[" + splitStr[1];
                    } else {
                        splitStr = line.split(op.getOperator());
                    }

                    if (op instanceof AssignOp) exprType = ExprType.ASSIGNMENT;
                    else if (op instanceof CompareOp) exprType = ExprType.COMPARISON;
                    else if (op instanceof BoolOp) exprType = ExprType.BOOLEAN;
                    else exprType = ExprType.ALGEBRA;

                    Value firstValue = Value.parseValue(splitStr[0].trim());
                    Value secondValue = Value.parseValue(splitStr[1].trim());

                    expression = new Expression(exprType, firstValue, op, secondValue);
                } else {
                    exprType = ExprType.ANNOTATION;
                    expression = new Expression(exprType, null, null, null);
                }

                expression.nodeName = name;
                expression.originalPosition = new Position(i, j);

                if (j < block.size() - 1) expression.outgoingEdges.add(new Position(i, j+1));
                else if (j == block.size() - 1)
                    for (int edge : cfg.get(i).edges) expression.outgoingEdges.add(new Position(edge, 0));

                if (exprType == ExprType.ANNOTATION) {
                    annotationStack.push(expression);
                    continue;
                }
                expressionStack.push(expression);

                if (exprType == ExprType.ASSIGNMENT || exprType == ExprType.COMPARISON) {
                    Statement statement = new Statement();
                    while (!expressionStack.isEmpty()) statement.addExpression(expressionStack.pop());
                    statement.label = new Label(conditionedCFG.statements.size() + 1);
                    conditionedCFG.addStatement(statement);
                }
            }
        }
    }

    private OpEnum splitIfAny(String line, OpEnum[] ... arguments) {
        String[] splitString;
        for (OpEnum[] opEnums : arguments) {
            for (OpEnum opEnum : opEnums) {
                String operator = opEnum.getOperator();
                if (operator.equals("+")) splitString = line.split("\\+");
                else if (operator.equals("*")) splitString = line.split("\\*");
                else if (operator.equals("or")) splitString = line.split(" or\\[");
                else if (operator.equals("and")) splitString = line.split(" and\\[");
                else if (operator.equals("not")) splitString = line.split("not\\(");
                else splitString = line.split(operator);
                if (splitString.length > 1) return opEnum;
            }
        }
        return null;
    }
}
```

My enum methods allowed me to avoid hardcoding most operator split expressions, but regular expression escape character requirements and string subexpression possibilities for boolean operators required me to handle these edge cases. Had I time, I would include enum methods for obtaining strings suitable for regex splitting.

ConditionedCFG.java

```
package Opt.Util;

import DataStructureGeneration.CFG;
import Opt.Util.Tokens.*;
import Opt.Util.Pairs.*;

import java.util.*;

public class ConditionedCFG {
    public LinkedList<Statement> statements = new LinkedList<>();

    public ConditionedCFG() { }

    public void addStatement(Statement statement) { statements.add(statement); }
    public int getSize() { return statements.size(); }

    public Label[] getLabels() {
        Label[] labels = new Label[statements.size()];
        for (int i = 0; i < statements.size(); i++) labels[i] = statements.get(i).label;
        return labels;
    }

    public void generateEdges(Stack<Expression> annotationStack) {
        for (Expression annotation : annotationStack) {
            for (Statement source : statements) {
                for (Expression expression : source.expressions) {
                    if (expression.outgoingEdges.contains(annotation.originalPosition)) {
                        expression.outgoingEdges.remove(annotation.originalPosition);
                        expression.outgoingEdges.union(annotation.outgoingEdges);
                    }
                }
            }
        }
        for (Statement source : statements) {
            for (Expression expression : source.expressions) {
                for (Position position : expression.outgoingEdges) {
                    Label destination = locationOf(position);
                    if (destination == null) {
                        source.successors.add(new Label(0));
                    } else if (destination != source.label) {
                        source.successors.add(destination);
                        statements.get(destination.value - 1).predecessors.add(source.label);
                    }
                }
            }
        }
        for (Statement statement : statements) if (statement.successors.isEmpty())
            statement.successors.add(new Label(0));
    }

    public Label locationOf(Position position) {
```

```
        for (Statement statement : statements) {
            for (Expression expression : statement.expressions) {
                Position tempPos = expression.originalPosition;
                if (tempPos.line == position.line && tempPos.node == position.node) return
statement.label;
            }
        }
        return null;
    }

    // Printing well was a high priority for me, so the function for it dynamically determines column
widths.
    public void printOut(String title) {
        String[][] statementPrints = new String[statements.size()][4];
        for (int i = 0; i < statements.size(); i++) {
            String[] response = statements.get(i).printOut();
            for (int j = 0; j < 4; j++) {
                statementPrints[i][j] = response[j];
                if (i == 0 && j == 0) {
                    if (statementPrints[i][j].equals(" ->")) statementPrints[i][j] = "Start" +
statementPrints[i][j];
                    else if (statementPrints[i][j].equals("Start ->")) continue;
                    else statementPrints[i][j] = "Start, " + statementPrints[i][j];
                }
            }
        }

        int[] maxLengths = { 0, 0, 0, 0 };

        for (int i = 0; i < statements.size(); i++) {
            for (int j = 0; j < 4; j++) maxLengths[j] = max(statementPrints[i][j].length(),
maxLengths[j]);
        }

        String tableLabelZero = "Predecessors:";
        String tableLabelOne = "Label:";
        String tableLabelTwo = "Statement:";
        String tableLabelThree = "Successors:";

        maxLengths[0] = max(maxLengths[0], tableLabelZero.length());
        maxLengths[1] = max(maxLengths[1], tableLabelOne.length());
        maxLengths[2] = max(maxLengths[2], tableLabelTwo.length());
        maxLengths[3] = max(maxLengths[3], tableLabelThree.length());

        title = " " + title + " ";
        int totalLength = Arrays.stream(maxLengths).sum() + 10;
        if (totalLength > title.length()) {
            int padding = (totalLength - title.length())/2;
            StringBuilder paddingString = new StringBuilder();
            for (int i = 0; i < padding; i++) paddingString.append("=");
            title = paddingString.toString() + title + paddingString.toString();
        }

        System.out.println();
        System.out.println(title);
        System.out.printf("%-" + (maxLengths[0]) + "s   ", tableLabelZero);
        System.out.printf("%" + (maxLengths[1]) + "s   ", tableLabelOne);
        System.out.printf("%-" + (maxLengths[2]) + "s   ", tableLabelTwo);
        System.out.printf("%-" + (maxLengths[3]) + "s\n", tableLabelThree);
```

```
        for (int i = 0; i < statementPrints.length; i++) {
            System.out.printf("%" + (maxLengths[0]) + "s   ", statementPrints[i][0]);
            System.out.printf("%" + (maxLengths[1]) + "s   ", statementPrints[i][1]);
            System.out.printf("%-" + (maxLengths[2]) + "s   ", statementPrints[i][2]);
            System.out.printf("%-" + (maxLengths[3]) + "s\n", statementPrints[i][3]);
        }
        System.out.println();
    }

    private int max(int first, int second) {
        if (first > second) return first;
        else return second;
    }

    // Conversion back into the original form is far simpler.
    public Map<Integer, CFG> toOutputForm() {
        Map<Integer, CFG> outputForm = new HashMap<>();
        for (Statement statement : statements) {
            int identifier = statement.label.value - 1;
            ArrayList<String> basicBlock = new ArrayList<>();
            ArrayList<Integer> edges = new ArrayList<>();
            ArrayList<String> nodeNames = new ArrayList<>();

            for (Expression expression : statement.expressions) {
                if (!expression.exprType.equals(Expression.ExprType.ANNOTATION)) {
                    String left = "[" + expression.getLeftString() + "]";
                    if (expression.leftVal.isInd()) {
                        left = ((Indirect)expression.leftVal).indExprName;
                    }
                    String right = "[" + expression.getRightString() + "]";
                    if (expression.rightVal.isInd()) {
                        right = ((Indirect)expression.rightVal).indExprName;
                    }
                    String operator = " " + expression.operator.getOperator() + " ";
                    String fullString = left + operator + right;
                    basicBlock.add(fullString);
                    nodeNames.add(expression.nodeName);
                }
            }
            for (Label label : statement.successors) edges.add(label.value - 1);
            outputForm.put(identifier, new CFG(identifier, basicBlock, edges, nodeNames));
        }
        return outputForm;
    }

    public Statement getStatWithLbl(Label label) {
        for (Statement statement : statements) if (statement.label.equals(label)) return statement;
        return null;
    }

    public boolean reduce() {
        boolean result = false;
        for (Statement statement : statements) result = statement.reduce() || result;
        return result;
    }

    public VarSet[] getGen() {
        VarSet[] gen = new VarSet[statements.size()];
        for (int i = 0; i < statements.size(); i++) gen[i] = statements.get(i).getGen();
```

```
        return gen;
    }

    public VarSet[] getKill() {
        VarSet[] kill = new VarSet[statements.size()];
        for (int i = 0; i < statements.size(); i++) kill[i] = statements.get(i).getKill();
        return kill;
    }

    public LabSet[] getSucc() {
        LabSet[] succ = new LabSet[statements.size()];
        for (int i = 0; i < statements.size(); i++) succ[i] = statements.get(i).getSucc();
        return succ;
    }

    public void addReturn(VarSet returnVars) {
        Statement returnStatement = new ReturnStatement(returnVars.get(0));
        returnStatement.label = new Label(statements.size() + 1);
        returnStatement.addExpression(new ReturnExpression(Expression.ExprType.RETURN));

        Label label = new Label(0);
        for (Statement statement : statements) {
            if (statement.successors.contains(label)) {
                statement.successors.remove(label);
                statement.successors.add(returnStatement.label);
                returnStatement.predecessors.add(statement.label);
            }
        }
        this.addStatement(returnStatement);
    }

    public void removeReturn() {
        Label returnLabel = statements.getLast().label;
        for (Statement statement : statements) {
            if ( statement.successors.contains(returnLabel) ) {
                statement.successors.remove(returnLabel);
                statement.successors.add(new Label(0));
            }
        }
        statements.removeLast();
    }

    public void removeStatements(Set<Label> labels) {
        for (Label label : labels) {
            Statement dying = getStatWithLbl(label);
            for (Statement statement : statements) {
                if (statement.successors.contains(label)) {
                    statement.successors.union(dying.successors);
                    statement.successors.remove(label);
                }
                if (statement.predecessors.contains(label)) {
                    statement.predecessors.union(dying.predecessors);
                    statement.predecessors.remove(label);
                }
            }
        }
        LinkedList<Statement> tempStatements = new LinkedList<>();
        while (!statements.isEmpty()) {
            Statement tempStat = statements.removeFirst();
```

```
            if (!labels.contains(tempStat.label)) tempStatements.add(tempStat);
        }
        statements = tempStatements;
        for (Statement statement : statements) {
            statement.successors.difference(labels);
            statement.predecessors.difference(labels);
        }
        condense();
    }

    public void condense() {
        LinkedList<Statement> tempStatements = new LinkedList<>();
        while (!statements.isEmpty()) {
            Statement tempStat = statements.removeFirst();
            tempStatements.add(tempStat);
            if (tempStat.label.value != tempStatements.size()) {
                Label tempLabel = new Label(tempStatements.size());
                for (Statement statement : tempStatements) {
                    if (statement.predecessors.contains(tempStat.label)) {
                        statement.predecessors.remove(tempStat.label);
                        statement.predecessors.add(tempLabel);
                    }
                    if (statement.successors.contains(tempStat.label)) {
                        statement.successors.remove(tempStat.label);
                        statement.successors.add(tempLabel);
                    }
                }
                for (Statement statement : statements) {
                    if (statement.predecessors.contains(tempStat.label)) {
                        statement.predecessors.remove(tempStat.label);
                        statement.predecessors.add(tempLabel);
                    }
                    if (statement.successors.contains(tempStat.label)) {
                        statement.successors.remove(tempStat.label);
                        statement.successors.add(tempLabel);
                    }
                }
                tempStat.label = tempLabel;
            }
        }
        statements = tempStatements;
        for (Statement statement : statements) {
            Set<Label> tempPred = new Set<>();
            Set<Label> tempSucc = new Set<>();
            for (Label label : statement.predecessors) if (label.value <= statements.size())
tempPred.add(label);
            for (Label label : statement.successors) if (label.value <= statements.size())
tempSucc.add(label);
            statement.predecessors = tempPred;
            statement.successors = tempSucc;
        }
    }
}
```

During dead code elimination, I build a list of statements to be removed. The ConditionedCFG takes this list and removes them. This was very tricky, and eventually required splitting iteration into several passes, in order to eliminate concurrent modifications that interfered

with other parts of the process. I originally called condense after each removal, removing the statements in reverse order, thinking that this would recursively ensure that the final statements would have labels from 1 to the number of statements, but the concurrent changes made this very problematic. Calling it at the end was better, and also enabled me to resolve a deeper complexity where in certain cases original edge values were not being deleted. Thanks to edge redirection, after statement removal and prior to condensing, any edge still matching a removed statement's label was a dead edge, erroneously not removed, and I could utilize my Set.difference() function to pass through and eliminate all of these. I regard this as a quick and dirty solution, as opposed to handling why the problem occurs. Given more time, I would optimize this. Condense then "rearranges" the list of statements, though this is largely just bringing the statement labels into alignment with their current actual ordering, since LinkedList.get(index) grabs the index element in every case.

In general, when I started running into this issue, I started trying to treat collections in a more functional way, building new collections from elements of an old collection, and switching them at the end. This proved far less prone to errors.

Set.java

```
package Opt.Util;

import java.util.LinkedList;

public class Set<T> extends LinkedList<T> {
    Class<T> type;

    @Override
    public boolean add(T t) {
        if (!super.contains(t)) {
            super.add(t);
            return true;
        }
        return false;
    }

    public boolean union(Set<T> ts) {
        boolean changes = false;
        for (T t : ts) {
            changes = changes | this.add(t);
        }
        return changes;
    }

    public boolean difference(Set<T> ts) {
        boolean changes = false;
        for (T t : ts) {
            changes = changes | this.remove(t);
        }
        return changes;
    }

    @Override
    public boolean remove(Object t) {
        if (super.contains(t)) {
            super.remove(t);
            return true;
```

```
        }
        return false;
    }
}
```

I am rather proud of this class, despite its simplicity. By extending LinkedList, using a type parameter, overriding LinkedList functions, and using non-short-circuit boolean operators, I created a set of operations that is easy to call, and returns all information needed to determine if iteration must continue. All object classes that I build Sets with override functions that allows the super.contains() and super.remove() to function exactly as they should, without the need for complicated testing duplication.

Label.java

```
package Opt.Util;

public class Label {
    public int value;
    public int index;
    public boolean endStatement = false;

    public Label(int value) {
        this.value = value;
        this.index = value - 1;
        if (value == 0) endStatement = true;
    }

    @Override
    public boolean equals(Object object) {
        boolean result = false;
        if (object instanceof Label) result = this.value == ((Label) object).value;
        return result;
    }

    @Override
    public int hashCode() {
        return 0;
    }

    public String printOut() {
        if (endStatement) return "End";
        return Integer.toString(value);
    }
}
```

Strictly speaking, this class is not necessary. It mostly exists to make it easier to avoid accidentally using a label value to access a Statement in 0-indexed collections, and provide a .printOut() function like all tokens.

Operator.java

```
package Opt.Util;
```

```
public abstract class Operator {

    public static OpEnum getOpEnum(String string) {
        OpEnum result;
        result = ArithOp.get(string);
        if (result == null) result = AssignOp.get(string);
        if (result == null) result = CompareOp.get(string);
        return result;
    }

    public interface OpEnum { String getOperator(); }

    public enum ArithOp implements OpEnum {
        ADD("+"),
        SUB("-"),
        MUL("*"),
        DIV("/");
        final String operator;

        ArithOp(String operator) { this.operator = operator; }
        public String getOperator() { return this.operator; }

        public static ArithOp get(String string) {
            for (ArithOp op : ArithOp.values()) if (op.operator.equals(string)) return op;
            return null;
        }
    }

    public enum AssignOp implements OpEnum {
        SET_EQ(":=");
        final String operator;

        AssignOp(String operator) { this.operator = operator; }
        public String getOperator() { return this.operator; }

        public static AssignOp get(String string) {
            for (AssignOp op : AssignOp.values()) if (op.operator.equals(string)) return op;
            return null;
        }
    }

    public enum CompareOp implements OpEnum {
        W_EQ("w="),
        W_LEQ("w<="),
        W_GEQ("w>="),
        W_LT("w<"),
        W_GT("w>"),
        I_EQ("i="),
        I_LEQ("i<="),
        I_GEQ("i>="),
        I_LT("i<"),
        I_GT("i>");
        final String operator;

        CompareOp(String operator) { this.operator = operator; }
        public String getOperator() { return this.operator; }

        public static CompareOp get(String string) {
            for (CompareOp op : CompareOp.values()) if (op.operator.equals(string)) return op;
```

```
            return null;
        }
    }

    public enum BoolOp implements OpEnum {
        NOT("not"),
        AND("and"),
        OR("or");

        final String operator;

        BoolOp(String operator) { this.operator = operator; }
        public String getOperator() { return this.operator; }

        public static BoolOp get(String string) {
            for (BoolOp op : BoolOp.values()) if (op.operator.equals(string)) return op;
            return null;
        }
    }

    public enum AnnotOp implements OpEnum {
        ANNOT_OP;

        @Override
        public String getOperator() { return ""; }
    }
}
```

This class provides great functionality for pulling String equivalents and very quick testing of operator categories by means of instanceof.

Value.java

```
package Opt.Util.Tokens;

public abstract class Value {

    public static Value parseValue(String string) {
        string = string.replace("(", "").replace(")", "").trim();
        if (string.matches("\\[.*][0-9]+")) return new Indirect(string);
        else if (string.matches("\\[[0-9]*]")) {
            String newString = string.replace("[", "").replace("]", "");
            return new Constant(Integer.parseInt(newString));
        } else if (string.matches("\\[true]")) {
            String newString = string.replace("[", "").replace("]", "");
            return new Constant(true);
        } else if (string.matches("\\[false]")) {
            return new Constant(false);
        } else {
            return new Variable(string.replace("[", "").replace("]", ""));
        }
    }

    public static Variable castVar(Value value) { return (Variable) value; }
    public static Constant castConst(Value value) { return (Constant) value; }

    public abstract String printOut();
```

```
    public boolean isVar() { return false; }
    public boolean isConst() { return false; }
    public boolean isBool() { return false; }
    public abstract boolean isEq(Value value);
    public boolean isInd() { return false; }
    public Constant toConst() {return null;}
    public Variable toVar() {return null;}
    public Indirect toInd() {return null;}
}
```

A superclass to allow storage of a generic Value, with default methods to ensure that a subclass need only override methods if the default is insufficient. Also provides the parseValue() function.

Value subclass Variable.java

```
package Opt.Util.Tokens;

public class Variable extends Value {
    public final String name;

    public Variable(String name) { this.name = name; }

    @Override
    public String printOut() { return this.name; }

    @Override
    public boolean equals(Object object) {
        boolean result = false;
        if (object instanceof Variable) result = this.name.equals(((Variable) object).name);
        return result;
    }

    @Override
    public int hashCode() { return name.hashCode(); }

    @Override
    public boolean isEq(Value value) {
        if (value.isConst() || value.isInd()) return false;
        return (((Variable) value).name.equals(this.name));
    }

    @Override
    public Variable toVar() { return this; }

    @Override
    public boolean isVar() { return true; }
}
```

Value subclass Constant.java

```
package Opt.Util.Tokens;

public class Constant extends Value {
    public int value;
```

```
    public boolean boolVal;
    public boolean isBoolVal;

    public Constant(int value) {
        this.value = value;
        this.isBoolVal = false;
    }
    public Constant(boolean boolVal) {
        this.boolVal = boolVal;
        this.isBoolVal = true;
    }

    @Override
    public String printOut() {
        if (this.isBoolVal) {
            if (boolVal) return "true";
            else return "false";
        } else return Integer.toString(this.value);
    }

    @Override
    public boolean isConst() { return true; }

    @Override
    public boolean isEq(Value value) {
        if (value.isVar() || value.isInd()) return false;
        Constant constant = (Constant) value;
        if (constant.isBoolVal && this.isBoolVal) {
            return constant.boolVal = this.boolVal;
        } else if (!constant.isBoolVal && !this.isBoolVal) {
            return constant.value == this.value;
        }
        return false;
    }

    @Override
    public Constant toConst() { return this; }

    @Override
    public boolean isBool() { return isBoolVal; }
}
```

Value subclass Indirect.java

```
package Opt.Util.Tokens;

public class Indirect extends Value {
    public int index;
    public String indExprName;

    public Indirect(String string) { indExprName = string; }

    @Override
    public String printOut() { return ""; }

    @Override
    public boolean isEq(Value value) {
```

```
        if (value instanceof Indirect) return ((Indirect) value).indExprName.equals(this.indExprName);
        return false;
    }

    @Override
    public boolean isInd() { return true; }

    @Override
    public Indirect toInd() { return this; }
}
```

Expression.java

```
package Opt.Util.Tokens;

import Opt.Util.Operator;
import Opt.Util.Pairs.*;
import Opt.Util.Set;

import static Opt.Util.Tokens.Value.castVar;

public class Expression {

    public enum ExprType {
        ASSIGNMENT,
        COMPARISON,
        ALGEBRA,
        BOOLEAN,
        ANNOTATION,
        RETURN;
    }

    public ExprType exprType;
    public Set<Position> outgoingEdges;
    public Position originalPosition;
    public String nodeName = "";
    public Operator.OpEnum operator;
    public Value leftVal;
    public Value rightVal;

    public Expression() {}

    public Expression(ExprType exprType, Value leftVal, Operator.OpEnum operator, Value rightVal) {
        this.exprType = exprType;
        this.operator = operator;
        this.leftVal = leftVal;
        this.rightVal = rightVal;
        this.outgoingEdges = new Set<>();
    }

    public boolean isArith() { return this.operator instanceof Operator.ArithOp; }
    public boolean isAssign() { return this.operator instanceof Operator.AssignOp; }
    public boolean isComp() { return this.operator instanceof Operator.CompareOp; }
    public boolean isBool() { return this.operator instanceof Operator.BoolOp; }

    public String getLeftString() {
        if (leftVal == null) return "Indirect";
```

```
        else return (leftVal.printOut());
    }

    public String getRightString() {
        if (rightVal == null) return "Indirect";
        else return (rightVal.printOut());
    }

    public Set<Variable> getFreeVariables() {
        Set<Variable> freeVars = new Set<>();
        if (rightVal.isVar()) freeVars.add(castVar(rightVal));
        if (!this.isAssign() && leftVal.isVar()) freeVars.add(castVar(leftVal));
        return freeVars;
    }

    public void foldVariable(Variable var, Constant constant) {
        if (rightVal.isEq(var)) rightVal = constant;
        if (!this.isAssign() && leftVal.isEq(var)) leftVal = constant;
    }

    public boolean reduceable() {
        if (this.isBool() || this.isArith())
            if (leftVal.isConst() && rightVal.isConst()) return leftVal.isBool() == rightVal.isBool();
        return false;
    }

    public Constant reduce() {
        int intResult = 0;
        boolean boolResult = false;
        if (this.isArith() && leftVal.isConst() && rightVal.isConst()) {
            int left = leftVal.toConst().value;
            int right = rightVal.toConst().value;
            switch ((Operator.ArithOp) this.operator) {
                case ADD -> intResult = left + right;
                case MUL -> intResult = left * right;
                case SUB -> intResult = left - right;
                case DIV -> intResult = left / right;
            }
            return new Constant(intResult);
        } else if (this.isBool() && leftVal.isBool() && rightVal.isBool()) {
            boolean left = leftVal.toConst().boolVal;
            boolean right = rightVal.toConst().boolVal;
            switch ((Operator.BoolOp) this.operator) {
                case AND -> boolResult = left && right;
                case OR -> boolResult = left || right;
                case NOT -> { }
            }
        }
        return new Constant(boolResult);
    }

    public void replaceIndirect(Constant constant, String side) {
        if (side.equals("left")) leftVal = constant;
        else if (side.equals("right")) rightVal = constant;
    }

    public Set<Variable> getGen() {
        Set<Variable> gen = new Set<>();
        if (rightVal.isVar()) gen.add(rightVal.toVar());
```

```
        if ((!this.isAssign() || this.isComp() || this.isBool()) && leftVal.isVar())
gen.add(leftVal.toVar());
        return gen;
    }

    public Set<Variable> getKill() {
        Set<Variable> kill = new Set<>();
        if (this instanceof ReturnExpression) return kill;
        if (this.isAssign()) kill.add(leftVal.toVar());
        return kill;
    }
}
```

Most objects build their return values from the return values obtained by calling their internal objects' equivalent methods. Here we see where getGen() and getKill() bottom out, since each expression knows what type it is and what its variables are. A Statement only needs to ask each of its expressions what its gen and kill set is, union them, and pass them back up. Reduce works similarly, though it probably should have been called evaluate().

Expression subclass ReturnExpression.java

```
package Opt.Util.Tokens;

public class ReturnExpression extends Expression {
    public ReturnExpression(ExprType exprType) { this.exprType = exprType; }
}
```

Statement.java

```
package Opt.Util;

import Opt.Util.Tokens.*;
import Opt.Util.Tokens.Expression.*;

import java.util.LinkedList;

public class Statement {
    public LinkedList<Expression> expressions;
    public Set<Label> predecessors;
    public Set<Label> successors;
    public Label label;

    public Statement() {
        label = new Label(0);
        expressions = new LinkedList<>();
        predecessors = new Set<>();
        successors = new Set<>();
    }

    public void addExpression(Expression expression) { expressions.add(0, expression); }

    public String[] printOut() {
        String[] response = new String[4];
        LinkedList<String> redirects = new LinkedList<>();
        LinkedList<String> redirectNames = new LinkedList<>();
```

```
        response[1] = "" + label.printOut() + ":";
        for (Expression expression : expressions) {
            String first = "";
            String operator;
            if (expression.exprType == ExprType.ANNOTATION) operator = "";
            else operator = " " + expression.operator.getOperator() + " ";
            String second = "";
            String fullString = "";
            boolean firstIndirect = (expression.leftVal instanceof Indirect);
            boolean secondIndirect = (expression.rightVal instanceof Indirect);
            first = expression.getLeftString();
            second = expression.getRightString();
            if (firstIndirect && !redirects.isEmpty()) {
                for (int i = 0; i < redirectNames.size(); i++) {
                    if (redirectNames.get(i).equals(((Indirect) expression.leftVal).indExprName)) {
                        first = "(" + redirects.remove(i) + ")";
                        redirectNames.remove(i);
                    }
                }
            }
            if (secondIndirect && !redirects.isEmpty()) {
                for (int i = 0; i < redirectNames.size(); i++) {
                    if (redirectNames.get(i).equals(((Indirect) expression.rightVal).indExprName)) {
                        second = "(" + redirects.remove(i) + ")";
                        redirectNames.remove(i);
                    }
                }
            }
            fullString = first + operator + second;
            switch (expression.exprType) {
                case ANNOTATION:
                    break;
                case BOOLEAN:
                case ALGEBRA:
                    redirects.add(fullString);
                    redirectNames.add(expression.nodeName);
                    break;
                case ASSIGNMENT:
                case COMPARISON:
                    response[2] = fullString;
                    break;
            }
        }
        String temp = "";
        if (label.value == 1) {
            temp += "Start";
            if (predecessors.size() != 0) temp += ", ";
        }
        for (int i = 0; i < predecessors.size(); i++) {
            temp += predecessors.get(i).printOut();
            if (i != predecessors.size() - 1) temp += ", ";
        }
        temp += " ->";
        response[0] = temp;
        temp = "-> ";
        for (int i = 0; i < successors.size(); i++) {
            temp += successors.get(i).printOut();
            if (i != successors.size() - 1) temp += ", ";
        }
```

```
        response[3] = temp;

        return response;
    }

    public Set<Variable> getFreeVariables() {
        Set<Variable> freeVars = new Set<>();
        for (Expression expression : expressions) freeVars.addAll(expression.getFreeVariables());
        return freeVars;
    }

    public boolean isAssignmentStatement() { return
expressions.getLast().operator.equals(Operator.AssignOp.SET_EQ); }
    public Variable getAssignment() { return expressions.getLast().leftVal.toVar(); }
    public boolean assignsToConstant() { return expressions.getLast().rightVal.isConst(); }
    public Constant getConstantAssigned() { return expressions.getLast().rightVal.toConst(); }
    public void foldConstant(Variable variable, Constant constant) {
        for (Expression expression : expressions) expression.foldVariable(variable, constant);
    }

    public boolean reduce() {
        LinkedList<Pairs.Pair<String, Constant>> constants = new LinkedList<>();
        LinkedList<Expression> newExpressionList = new LinkedList<>();
        boolean result = false;

        for (Expression expression : this.expressions) {
            if (!constants.isEmpty()) {
                if (expression.leftVal.isInd()) {
                    String test = expression.leftVal.toInd().indExprName;
                    for (Pairs.Pair pair : constants)
                        if (test.equals(pair.first))
                            expression.replaceIndirect((Constant) pair.second, "left");
                }
            }
            if (!constants.isEmpty()) {
                if (expression.rightVal.isInd()) {
                    String test = expression.rightVal.toInd().indExprName;
                    for (Pairs.Pair pair : constants)
                        if (test.equals(pair.first))
                            expression.replaceIndirect((Constant) pair.second, "right");
                }
            }
            if (expression.reduceable()) {
                result = true;
                constants.add(new Pairs.Pair<>(expression.nodeName, expression.reduce()));
            } else newExpressionList.add(expression);
        }

        this.expressions = newExpressionList;
        return result;
    }


    public VarSet getGen() {
        VarSet gen = new VarSet();
        for (Expression expression : expressions) gen.union(expression.getGen());
        return gen;
    }
```

```
    public VarSet getKill() {
        VarSet kill = new VarSet();
        for (Expression expression : expressions) kill.union(expression.getKill());
        return kill;
    }

    public LabSet getSucc() {
        LabSet succ = new LabSet();
        for (Label label : successors) succ.add(label);
        return succ;
    }
}
```

Statement subclass ReturnStatement.java

```
package Opt.Util.Tokens;

import Opt.Util.Statement;
import Opt.Util.VarSet;

public class ReturnStatement extends Statement {
    public Variable returnVar;

    public ReturnStatement(Variable returnVar) {
        super();
        this.returnVar = returnVar;
    }

    @Override
    public VarSet getGen() {
        VarSet gen = new VarSet();
        gen.add(returnVar);
        return gen;
    }

    @Override
    public VarSet getKill() { return new VarSet(); }
    @Override
    public String[] printOut() { return new String[] { "RETURN", "", returnVar.name, "" }; }
}
```

Data.java

```
package Opt;
public abstract class Data { public static String outputVariable; }
```

This class exists for the sole purpose of passing the command-line argument of the output variable to the liveness analyzer without modifying a whole bunch of parameter lists.

Set subclass LabSet.java

```
package Opt.Util;
public class LabSet extends Set<Label> { }
```

Set subclass VarSet.java

```
package Opt.Util;
import Opt.Util.Tokens.Variable;
public class VarSet extends Set<Variable> { }
```

Pairs.java

```
package Opt.Util;

public abstract class Pairs {

    public static class Pair<T, U> {
        final T first;
        final U second;

        private Pair() {
            first = null;
            second = null;
        }

        public Pair(T first, U second) {
            this.first = first;
            this.second = second;
        }
    }

    public static class Position extends Pair<Integer, Integer> {
        final int node;
        final int line;

        public Position(int node, int line) {
            this.node = node;
            this.line = line;
        }

        @Override
        public boolean equals(Object o) {
            Position position = (Position) o;
            return position.line == this.line && position.node == this.node;
        }

        @Override
        public int hashCode() { return 0; }
    }
}
```

Pairs mostly exists to make .contains() and .equals() work in other places in the code.

Reaching Definitions and Simple Constant Folding

RDAnalyzer.java

```
package Opt;
```

```
import Opt.Util.*;
import Opt.Util.Tokens.Constant;
import Opt.Util.Tokens.Expression;
import Opt.Util.Tokens.Variable;

public class RDAnalyzer {
    private ConditionedCFG conditionedCFG;
    private final String[] VARS;

    private int numberOfStatements;
    private EntryRD[] entryRDs; // The entry RD structures.
    private ExitRD[]  exitRDs;  // The exit RD structures.

    public RDAnalyzer(ConditionedCFG conditionedCFG, String[] vars) {
        this.conditionedCFG = conditionedCFG;
        this.VARS = vars;
        initialize();
        calculateRDs();
    }

    private void initialize() {
        numberOfStatements = conditionedCFG.statements.size();
        entryRDs = new EntryRD[numberOfStatements];
        exitRDs = new ExitRD[numberOfStatements];
        for (int i = 0; i < numberOfStatements; i++) {
            entryRDs[i] = new EntryRD(i+1);
            exitRDs[i] = new ExitRD(i+1);
        }
        for (int i = 0; i < numberOfStatements; i++) {
            Statement statement = conditionedCFG.statements.get(i);
            for (Label label : statement.predecessors)
                entryRDs[i].addPred(label.value);
            if (statement.expressions.getLast().operator == Operator.AssignOp.SET_EQ) {
                String assignment = statement.expressions.getLast().getLeftString();
                exitRDs[i].setAssignment(assignment);
            }
        }
    }

    public void calculateRDs() {
        do {
            for (int i = 0; i < numberOfStatements; i++) {
                entryRDs[i].transfer();
                exitRDs[i].transfer();
            }
            for (int i = 0; i < numberOfStatements; i++) {
                entryRDs[i].calculateNextIter();
                exitRDs[i].calculateNextIter();
            }
        } while (setsChanged());
        printRDEquations();
        printRDSets();
    }

    public ConditionedCFG fold() {
        conditionedCFG.reduce();

        boolean changes = true;
```

```
        boolean foldChange;
        boolean reduceChange;

        while (changes) {
            foldChange = false;
            for (Statement statement : conditionedCFG.statements) {
                Set<Variable> freeVariables = statement.getFreeVariables();
                int sourceLabel = statement.label.value;
                EntryRD entryRD = entryRDs[sourceLabel - 1];
                Set<Definition> entryDefs = new Set<>();
                for (Variable var : freeVariables) {
                    boolean doNotAdd = false;
                    int count = 0;
                    for (Definition def : entryRD.getCurr()) {
                        if (def.VAR.equals(var.name)) {
                            count++;
                            if (def.LBL == 0) doNotAdd = true;
                        }
                    }
                    if (count == 1 && !doNotAdd) {
                        for (Definition def : entryRD.getCurr()) {
                            if (def.VAR.equals(var.name)) entryDefs.add(def);
                        }
                    }
                }

                for (Definition def : entryDefs) {
                    if (conditionedCFG.statements.get(def.LBL - 1).assignsToConstant()) {
                        Constant constant = conditionedCFG.statements.get(def.LBL -
1).getConstantAssigned();
                        statement.foldConstant(new Variable(def.VAR), constant);
                        foldChange = true;
                    }
                }
            }
            reduceChange = conditionedCFG.reduce();
            changes = foldChange || reduceChange;
        }
        return conditionedCFG;
    }

    public void printRDEquations() {
        String labels = "";
        for (int i = 1; i <= numberOfStatements; i++) {
            labels += i;
            labels += ", ";
        }
        labels = "Labels    = { " + labels + "? }";
        String vars = "";
        for (int i = 0; i < VARS.length; i++) {
            vars += VARS[i];
            if (i != VARS.length - 1) vars += ", ";
        }
        vars = "Variables = { " + vars + " }";
        System.out.println("=============== Reaching Definition Equations ===============");
        System.out.println(labels);
        System.out.println(vars);
        System.out.println();
        for (int i = 0; i < numberOfStatements; i++) {
```

```
            String definitionEquation = "RD_in(" + (i + 1) + ") = ";
            System.out.printf("%16s", definitionEquation);
            definitionEquation = "";
            if (i == 0) {
                definitionEquation += "{ ";
                for (int j = 0; j < VARS.length; j++) {
                    definitionEquation += "(" + VARS[j] + ", ?)";
                    if (j != VARS.length - 1) definitionEquation += ", ";
                }
                definitionEquation += " }";
            } else {
                Statement statement = conditionedCFG.statements.get(i);
                Set<Label> preds = statement.predecessors;
                for (int j = 0; j < preds.size(); j++) {
                    definitionEquation += "RD_out(" + preds.get(j).value + ")";
                    if (j != preds.size() - 1) definitionEquation += " U ";
                }
            }
            System.out.println(definitionEquation);
        }
        System.out.println();

        for (int i = 0; i < numberOfStatements; i++) {
            String definitionEquation = "RD_out(" + (i + 1) + ") = RD_in(" + (i + 1) + ")";
            System.out.printf("%24s", definitionEquation);
            definitionEquation = "";
            Statement statement = conditionedCFG.statements.get(i);
            Expression expression = statement.expressions.getLast();
            if (expression.isAssign()) {
                String var = expression.getLeftString();
                definitionEquation += " \\ { (" + var + ", L) | L is an element of Labels } U { ("
                                      + var + ", " + (i + 1) + ") }";
            }
            System.out.println(definitionEquation);
        }
        System.out.println();
    }

    public void printRDSets() {
        System.out.println("=============== Reaching Definition Sets ==============");
        for (int i = 0; i < numberOfStatements; i++) {
            System.out.printf("RD_in(%3s):   { ", i + 1);
            for (Definition definition : entryRDs[i].getCurr()) {
                String label = "";
                if (definition.LBL == 0) label = "?";
                else label += definition.LBL;
                System.out.printf("(%s, %s) ", definition.VAR, label);
            }
            System.out.println(" }");
        }
        System.out.println();
        for (int i = 0; i < numberOfStatements; i++) {
            System.out.printf("RD_out(%3s):  { ", i + 1);
            for (Definition definition : exitRDs[i].getCurr()) {
                String label = "";
                if (definition.LBL == 0) label = "?";
                else label += definition.LBL;
                System.out.printf("(%s, %s) ", definition.VAR, label);
            }
```

```
            System.out.println(" }");
        }
    }

    // Short-circuit testing for any set having changed.
    private boolean setsChanged() {
        for (int i = 0; i < numberOfStatements; i++)
            if (entryRDs[i].isChanged() || exitRDs[i].isChanged()) return true;
        return false;
    }

    // A class to define a Definition as a tuple of a variable and an expression label.
    // Also provides functionality for comparison to enable easy sorting.
    public class Definition implements Comparable<Definition> {
        final String VAR;
        final int LBL;

        public Definition(String var, int lbl) {
            this.VAR = var;
            this.LBL = lbl;
        }

        public int compareTo(Definition definition) {
            if (!this.VAR.equals(definition.VAR)) return this.VAR.compareTo(definition.VAR);
            else return this.LBL - definition.LBL;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null) return false;
            if (this.getClass() != o.getClass()) return false;
            Definition definition = (Definition) o;
            return (this.VAR.equals(definition.VAR) && this.LBL == definition.LBL);
        }
    }

    // The superclass for Entry and Exit RDs, to reduce repetition and enable polymorphism.
    private abstract class RD {
        public abstract void calculateNextIter();

        private final Set<Definition> prev; // The RD set obtained on the previous iteration.
        private final Set<Definition> curr; // The working RD set on the current iteration.

        public RD() {
            curr = new Set<>();
            prev = new Set<>();
        }
        public boolean isChanged() {
            prev.sort(Definition::compareTo);
            curr.sort(Definition::compareTo);
            boolean changes = false;
            if (prev.size() != curr.size()) changes = true;
            else {
                for (int i = 0; i < prev.size(); i++) {
                    if (!prev.get(i).equals(curr.get(i))) changes = true;
                }
            }
            return changes;
```

```
        }
        // For use after determination that a fixed point was not reached.
        // Moves curr to prev in preparation for next iteration.
        public void transfer() {
            prev.clear();
            prev.addAll(curr);
            curr.clear();
        }

        // Get previous iteration's set.
        public Set<Definition> getPrev() { return prev; }
        public Set<Definition> getCurr() { return curr; }
        // Add a provided definition to current iteration's set.
        public void add(Definition definition) { if (!curr.contains(definition)) curr.add(definition); }
        // Add a provided set's definitions to current iteration's set.
        public void union(Set<Definition> newSet) { for (Definition definition : newSet)
add(definition); }
        // Remove all definitions for a provided variable from current iteration's set.
        public void clearVar(String var) { curr.removeIf(def -> def.VAR.equals(var)); }
    }

    // The subclass of RD that is used for Entry RD sets and equations.
    private class EntryRD extends RD {
        // A flag for the first Entry RD set, since it behaves uniquely.
        private boolean firstEntryRD = false;
        // The set of initial definitions for use by the first Entry RD set.
        private Set<Definition> init;
        // A list of the labels which are predecessors to this Entry RD.
        private Set<Integer> preds;

        public EntryRD(int label) {
            preds = new Set<>();
            init = new Set<>();
            if (label == 1) {
                firstEntryRD = true;
                for (String var : VARS) init.add(new Definition(var, 0));
            }
        }

        // Add a label to the list of predecessors.
        public void addPred(int pred) { preds.add(pred); }
        // Behavior for Entry RD equations.
        @Override
        public void calculateNextIter() {
            if (firstEntryRD) {
                this.union(init);
            } else {
                for (Integer pred : preds) {
                    this.union(exitRDs[pred-1].getPrev());
                }
            }
        }
    }

    // The subclass of RD that is used for Exit RD sets and equations.
    private class ExitRD extends RD {
        // The Exit RD label, since Exit RD equations always depend on their label's Entry RD.
        private final int LABEL;
        // If relevant, the definition given by the assignment in the labelled expression.
```

```
        private String assignment = null;

        public ExitRD(int label) { LABEL = label; }

        // Set the assignment if one exists.
        public void setAssignment(String var) { assignment = var; }
        // Behavior for Exit RD equations.
        @Override
        public void calculateNextIter() {
            this.union(entryRDs[LABEL-1].getPrev());
            if (assignment != null) {
                this.clearVar(assignment);
                this.add(new Definition(assignment, LABEL));
            }
        }
    }
}
```

RDAnalyzer was mostly written before CFG generation was complete, and therefore before the creation of the ConditionedCFG, and large portions predate the creation of the robust Set class and other quality of life developments in my code. Given more time, I would like to refactor it to match the style of the LivenessAnalyzer and utilize polymorphism, separation of concerns, and such. As it is, it progresses in a more imperative way than the rest of my code.

To accomplish constant folding, I developed an RD object which Entry and Exit RDs would extend, each knowing what to do when called to iterate. A Definition inner class provides methods for comparison of Definitions, which I used to sort sets. I sorted them in order to compare the contents of the sets after each iteration. We will see that my LivenessAnalyzer, written after the development of my various quality if life changes, uses a much less contrived method for tracking changes.

Upon creation of the RDAnalyzer, an EntryRD and ExitRD is made for each statement, each receiving the values relevant to its own calculations. The first EntryRD receives the set of Definitions with the question mark, in accordance with how we represented values with no in-function assignments yet. All other EntryRDs receive a list of their predecessors, and all ExitRDs are informed of if they correspond to an assignment, and if so, which variable they assign.

Calculation of the reaching definition sets then iterates while a separate method checks for changes by brute force. In order to ensure that each iteration is using the last iteration's values, RDs have a current and a previous set of definitions. setsChanged() compares these (correctly, but badly). Iteration occurs largely within the RDs themselves, when I call the RD.calculateNextIter() method. Each EntryRD (except the first) uses the list of its predecessors to request the previous iteration's set from the corresponding ExitRD, unioning this with its current iteration's set. Each ExitRD gets the previous iteration's set from its corresponding entry equation, and unions this set with its own current set. If it corresponds to an assignment, is clears all definitions of that variable, and adds the definition of that variable corresponding to its own label. The driving method calls for each RD to move its previous iteration's set into its current set once it has determined that a fixed point has not been reached, and the process starts again.

Once a fixed point is reached, the algorithm calls fold(), which uses the answer to the RD analysis to propagate constants through the statements of the ConditionedCFG. It does this

manually, obtaining each statement in the CFG, asking it for its free variables. It iterates over these freeVariables, and then iterates over the definitions in that Statement's entryRD set, checking to see if that definition's variable is the current free variable. At the end of iteration over the definitions, if the statement had a free variable in the entry set, and the definition in the entry set was not undefined, it adds that variable to a Set of definitions. Once iteration over the statement's free variables is complete, it uses this Set of definitions to step over all statements (while within the uppermost iteration over statements) in ConditionedCFG with labels in the definition set, and if they assign to a constant, it gets that constant and tells the current statement from the outermost loop to fold its constant with a Variable and a constant value. It then asks ConditionedCFG to reduce (simplify statements), and if anything was folded or reduced, it starts again. Frankly, this algorithm for folding is *terrible*. It works, but barely.

The writing of this class predated the writing of the LivenessAnalyzer by only a week or two, but it is very different. It was adapted to use the CondtionedCFG even before I wrote the LivenessAnalyzer, so the ConditionedCFG and supplementary objects were not even in their refined forms yet. It uses a prototype of the set operation methods, and its quality of life methods are far from optimized. It features a great deal less separation of concerns, and I wrote the prints before any of the others, so while finishing up prettifying them was a desire, but was lower priority than fixing bugs. Overall, I regard it as illustrative of correct but mediocre code.

In order to improve it, I would make quite a few changes. I would probably hoist the actual definition sets out of the RD objects. I have contemplated extending Set into a ReachingDefinitionSet, which would inherit its superclass's set operation methods, and could implement some of the previous and current iteration logic simply, using the set operation method boolean return values to determine if changes occurred. At present, the RD classes are a poor representation of a collection, and I would like to be able to operate on them with simple collection operations, that they would then know how to implement themselves.

The fold algorithm would probably be redeveloped entirely. I would most likely refactor Definition, including adding a field for a constant, to be filled if that Definition corresponds to an assignment to a constant. I could accomplish this in a single pass over statements and the much cleaner use of .contains(). I could then simply iterate through each statement, and pass it a list of its unique Definitions that are not undefined, which it could use to very simply fold its constants *and* reduce accordingly, returning a boolean indicating whether any changes were made. It would loop over this while something changes, in order to ensure that new foldings and reductions made possible by previous iterations can occur. This could be further improved by removing the Definition from the working set once it has been found to be equal to a constant and all statements with that Definition have used it to fold.

Liveness Analysis and Dead Code Elimination

LivenessAnalyzer.java

```
package Opt;

import Opt.Util.*;
import Opt.Util.Tokens.Variable;
```

```
import java.util.Arrays;
import java.util.LinkedList;

public class LivenessAnalyzer {
    private ConditionedCFG conditionedCFG;
    private VarSet liveVarsAtEnd;
    private int numEquations;
    private VarSet[] gen;
    private VarSet[] kill;
    private LabSet[] succ;
    private VarSet[] in;
    private VarSet[] out;

    public LivenessAnalyzer(ConditionedCFG conditionedCFG, Set<Variable> liveVarsAtEnd) {
        this.conditionedCFG = conditionedCFG;
        this.liveVarsAtEnd = new VarSet();
        this.liveVarsAtEnd.union(liveVarsAtEnd);
        conditionedCFG.addReturn(this.liveVarsAtEnd);
        this.numEquations = conditionedCFG.statements.size();

        this.gen = conditionedCFG.getGen();
        this.gen[numEquations - 1].union(this.liveVarsAtEnd);
        this.kill = conditionedCFG.getKill();
        this.succ = conditionedCFG.getSucc();

        this.in = new VarSet[numEquations];
        this.out = new VarSet[numEquations];

        for (int i = 0; i < numEquations; i++) {
            this.in[i] = new VarSet();
            this.out[i] = new VarSet();
        }

        iterate();
        printOut();
        eliminateDeadCode();
    }

    private void eliminateDeadCode() {
        conditionedCFG.removeReturn();
        LinkedList<Statement> tempStatements = conditionedCFG.statements;
        Set<Label> deadStatements = new Set<>();

        for (int i = 0; i < tempStatements.size(); i++) {
            Statement statement = conditionedCFG.statements.get(i);
            if (statement.isAssignmentStatement() && !out[i].contains(statement.getAssignment())) {
                deadStatements.add(statement.label);
            }
        }
        conditionedCFG.removeStatements(deadStatements);
    }

    private void iterate() {
        boolean changes = true;
        while (changes) {
            changes = false;

            for (int i = numEquations - 1; i >= 0; i--) {
```

```
                for (Label label : succ[i]) {
                    changes = changes | out[i].union(in[label.index]);
                }
                VarSet temp = new VarSet();
                for (Variable variable : out[i]) temp.add(variable);
                temp.difference(kill[i]);
                VarSet temp2 = new VarSet();
                for (Variable variable : gen[i]) temp2.add(variable);
                temp2.union(temp);
                changes = changes | in[i].union(temp2);
            }
        }
    }

    public void printOut() {
        String title       = " Liveness Analysis ";
        String tableLab    = "i:";
        String tableSucc   = "succ[i]:";
        String tableGen    = "gen[i]:";
        String tableKill   = "kill[i]:";
        String tableInEq   = "in[i] equation:";
        String tableOutEq  = "out[i] equation:";
        String tableInRes  = "in[i] result:";
        String tableOutRes = "out[i] result:";

        int[] maxLengths = new int[8];

        maxLengths[0] = tableLab.length();
        maxLengths[1] = tableSucc.length();
        maxLengths[2] = tableGen.length();
        maxLengths[3] = tableKill.length();
        maxLengths[4] = tableInEq.length();
        maxLengths[5] = tableOutEq.length();
        maxLengths[6] = tableInRes.length();
        maxLengths[7] = tableOutRes.length();

        String[] labelStrings  = new String[numEquations];
        String[] succStrings   = getStringSet(succ);
        String[] genStrings    = getStringSet(gen);
        String[] killStrings   = getStringSet(kill);
        String[] inEqStrings   = new String[numEquations];
        String[] outEqStrings  = new String[numEquations];
        String[] inResStrings  = getStringSet(in);
        String[] outResStrings = getStringSet(out);

        Label[] labels = conditionedCFG.getLabels();
        for (int i = 0; i < labels.length; i++) {
            labelStrings[i] = Integer.toString(labels[i].value);
        }

        for (int i = 0; i < numEquations; i++) {
            int label = i + 1;

            inEqStrings[i] = "gen[" + label + "] U ( out[" + label + "] \\ kill[" + label + "] )";
            outEqStrings[i] = "";
            if (succ[i].size() == 0) outEqStrings[i] += "{ }";
            else {
                for (int j = 0; j < succ[i].size(); j++) {
                    Label temp = succ[i].get(j);
```

```
                outEqStrings[i] += "in[" + temp.value + "]";
                if (j < succ[i].size() - 1) {
                    outEqStrings[i] += " U ";
                }
            }
        }
    }

    int totalLength = 0;

    maxLengths[0] = max(maxLength(labelStrings),  maxLengths[0]);
    maxLengths[1] = max(maxLength(succStrings),   maxLengths[1]);
    maxLengths[2] = max(maxLength(genStrings),    maxLengths[2]);
    maxLengths[3] = max(maxLength(killStrings),   maxLengths[3]);
    maxLengths[4] = max(maxLength(inEqStrings),   maxLengths[4]);
    maxLengths[5] = max(maxLength(outEqStrings),  maxLengths[5]);
    maxLengths[6] = max(maxLength(inResStrings),  maxLengths[6]);
    maxLengths[7] = max(maxLength(outResStrings), maxLengths[7]);

    totalLength = Arrays.stream(maxLengths).sum() + (3 * 7);
    if (totalLength > title.length()) {
        int padding = (totalLength - title.length())/2;
        StringBuilder paddingString = new StringBuilder();
        for (int i = 0; i < padding; i++) paddingString.append("=");
        title = paddingString.toString() + title + paddingString.toString();
    }

    System.out.println();
    System.out.println(title);

    System.out.printf("%-" + maxLengths[0] + "s   ", tableLab);
    System.out.printf("%-" + maxLengths[1] + "s   ", tableSucc);
    System.out.printf("%-" + maxLengths[2] + "s   ", tableGen);
    System.out.printf("%-" + maxLengths[3] + "s   ", tableKill);
    System.out.printf("%-" + maxLengths[4] + "s   ", tableInEq);
    System.out.printf("%-" + maxLengths[5] + "s   ", tableOutEq);
    System.out.printf("%-" + maxLengths[6] + "s   ", tableInRes);
    System.out.printf("%-" + maxLengths[7] + "s   ", tableOutRes);
    System.out.println();

    for (int i = 0; i < numEquations; i++) {
        System.out.printf("%-" + maxLengths[0] + "s   ", labelStrings[i]);
        System.out.printf("%-" + maxLengths[1] + "s   ", succStrings[i]);
        System.out.printf("%-" + maxLengths[2] + "s   ", genStrings[i]);
        System.out.printf("%-" + maxLengths[3] + "s   ", killStrings[i]);
        System.out.printf("%-" + maxLengths[4] + "s   ", inEqStrings[i]);
        System.out.printf("%-" + maxLengths[5] + "s   ", outEqStrings[i]);
        System.out.printf("%-" + maxLengths[6] + "s   ", inResStrings[i]);
        System.out.printf("%-" + maxLengths[7] + "s   ", outResStrings[i]);
        System.out.println();
    }

    System.out.println();
}

public int maxLength(String[] stringSet) {
    int max = 0;

    for (String string : stringSet) {
```

```
            if (string.length() > max) max = string.length();
        }

        return max;
    }

    public String[] getStringSet(VarSet[] varSets) {
        String[] stringSet = new String[varSets.length];

        String setStart = "{ ";
        String setEnd = " }";
        for (int i = 0; i < varSets.length; i++) {
            String temp = setStart;
            for (int j = 0; j < varSets[i].size(); j++) {
                temp += varSets[i].get(j).name;
                if (j != varSets[i].size() - 1) temp += ", ";
            }
            stringSet[i] = temp + setEnd;
        }
        return stringSet;
    }

    public String[] getStringSet(LabSet[] labSets) {
        String[] stringSet = new String[labSets.length];

        String setStart = "{ ";
        String setEnd = " }";
        for (int i = 0; i < labSets.length; i++) {
            String temp = setStart;
            for (int j = 0; j < labSets[i].size(); j++) {
                temp += labSets[i].get(j).value;
                if (j != labSets[i].size() - 1) temp += ", ";
            }
            stringSet[i] = temp + setEnd;
        }
        return stringSet;
    }

    private int max(int first, int second) {
        if (first > second) return first;
        else return second;
    }

    public ConditionedCFG getConditionedCFG() {
        return conditionedCFG;
    }
}
```

I consider the LivenessAnayzer to be far more refined than the RDAnalyzer. I learned lessons in writing the RDAnalyzer, and even moreso while developing the ConditionedCFG to be used by it, that fundamentally changed my approach to the fixed-point iteration problem. Instead of going back and modifying code just enough to use the CondtionedCFG, I was able to refine the ConditionedCFG and the LivenessAnalyzer in tandem as I worked towards the final algorithm.

During the creation of the ConditionedCFG, most of the information needed for Liveness Analysis was already produced. LivenessAnalyzer asks ConditionedCFG to add a return statement,

and then for the gen, succ, and kill sets for all its statements. (ConditionedCFG simply asks its Statements, which either calculates or asks its Expressions, which in turn calculate or return the already calculated answer.). It adds the variable that will be live at the end to the gen set for the final statement, and initializes new sets for in and out.

To iterate, the algorithm uses the fact that the set operation methods return a boolean is changes occurred. Combined with a non-short-circuit or, changes tracks if any set operation returns true at any point. Each iteration proceeds thus:

For each statement, in reverse order:

Union the statement's out set with all it's predecessors' in sets.

Build a set from the statement's gen set, unioned with the difference between its out set and its kill set.

Union the statement's in set with the set from the last operation.

Afterwards, dead code can be eliminated. To do so, it tells ConditionedCFG to remove the injected return statement, and then it iterates, over each statement in the ConditionedCFG, and if that statement assigns to a variable that is not alive at its exit, it marks that statement as dead by adding its label to a Set of dead Statements. When complete, it tells the ConditionedCFG to remove all those Statements.

This is the bulk of the work of the algorithm, and it fits into the first 80 lines of the class. The remainder is pretty-print and pretty-print utility methods. (The print method makes up nearly half of the entire class.) ConditionedCFG has some subtle work ahead of it in removing the statements, as we have already discussed, but since it is passed a complete list of statements to be removed, it can do this in relatively few passes.

This is far simpler than the current incarnation of the RDAnalyzer, and far, far faster.

Once the optimization steps are complete, the Optimizer converts the ConditionedCFG back into the original CFG format, which is far easier than the other way, and passes this back to the main thread of the compiler.

Some Important Considerations

Variable Elimination

In general, we discussed how to resolve the fact that some variables can be eliminated during optimization, but the source code writer expects a certain set of variables, and the C code is supposed to print each of these. In the end, the prevailing argument was that, while on the machine side dead variables are irrelevant, a programmer should be able to trust that the parameters of a function do not change, even if those parameters are pointless. As such, our generated C code takes as arguments all variables in the original code, regardless of whether they will be used or not, and the assembly only loads in the ones that were not eliminated during optimization.

Upgradeability

I chose to try and make the optimization as expandable as possible. The schema for CFG representation and various token and set manipulations was determined with this in mind. If this code were being prepared for public use, the important abstract classes would even more strongly enforce that subclasses implement a variety of functions. I believe that the model developed in the Liveness Analysis has strong potential to be generalizable to any fixed point iteration. Java has some minimal support for functions as parameters, and I have wondered if there is a way to build a general Fixed Point Iterator over a ConditionedCFG which can be configured to cover most situations if configured properly.

Object Orientation and Class Explosion

My code featured a relatively huge number of classes for the relatively small scope. Java has a tendency to shake out this way, since its object orientation encourages the creation of classes, and some of its rules for accessing subclasses can be arcane, or incredibly verbose. But although this increases the number of classes visible in a project, if used properly, it is the source of the things that Java tends to be really good at. The derivation of attributes and methods in a long chain from Object to your own smallest subclass means that knowing how a few very important native functions perform their job enables you to build classes that can be operated on by any number of very stable and well-engineered methods. The built-in interfaces increase this ability ever more. Since Java was our source code, I decided to work with Java's strengths.

Known Issues

There are a couple of areas where the optimization falls short. I have detailed them below in various categories.

Garbage In, Garbage Out

Issue:	Code with conditional blocks that can be eliminated entirely will often result in incorrect edges, due to the way edges are changed when code is removed.
Possible Solution:	This could be fixed by some additional logic checking for the presence of a conditional annotation in the original CFG and carrying this information forward.

Issue:	Conditional statements with entirely dead blocks, whether due to elimination, or due to constant folding revealing that their condition is always true or always false, are not eliminated.
Possible Solution:	The fix for the previous problem would solve half of this, and the information it uses could be used to redirect edges properly when removing conditionals. This would open up an opportunity to perform another pass of constant folding and liveness analysis, if we wished it.

Issue:	If a variable is not defined as the output variable in the command line, only constant folding is performed.
Possible Solution:	I thought about defaulting to some variable, whether "output" or the last one assigned in the code, but I would rather output correct but under-optimized code than code that is very good at being very incorrect, or none at all.

Incomplete Support

Issue:	Certain arrangements of arithmetic expressions block partial expression evaluation, as the reduction algorithm only looks for reduceability on a per-expression basis. Non-constant-foldable variables will not be rearranged when associativity would normally determine that this is perfectly acceptable.
Possible Solution:	A process for analyzing subexpression chains was considered desirable, though time proved too scarce.

Issue:	The optimizer does not provide support for the unary operator not. This is due to the initial decision to define expressions as being comprised of two values and an operator, and realizing too late that both the lexing and parsing and the optimization steps would need to add functionality to include not.
Possible Solution:	In the optimization modules, Expression could be subclassed into a NotExpression, of different form and appropriately different behavior. Though the assembly code generation can handle this behavior, the code which parses a CFG into assembly code can not, and time ran out to fix it.

Efficiency

Issue:	The Reaching Definitions Algorithm is terribly inefficient.
Possible Solution:	It should be rewritten to be more like the Liveness Analyzer.

Register Allocation and Code Generation

Del Jones

Due to the design philosophy used in project 1, editing the assembly generation to allocate variables into the s-registers and memory was not complicated. Before code generation occurs the function initializeMemory is called. This function takes a list as its parameter, called mList. This list holds all variables that will be used during the program. In project 1 this function simply placed all of these variables in a list called memStorage, which is used to keep track of things in memory for both saving and loading. In project 2 the function was expanded. It begins by iterating through mList, placing up to eleven variables into sRegisters, which is an ArrayList that manages variables that are to be stored in the s-registers. These first 11 variables are also put into memory, and opcodes to load those values into the relevant s-registers are called. If there are more than 11 variables, the rest are simply put into memory and called upon as needed in a similar manner as in project 1.

InitializeMemory()

```
public void initializeMemory(List<String> mList)
{
   for(int i = 0; (i < 11 && i < mList.size()); i++)
   {
       memStorage.add(mList.get(i));
       sRegisters.add(mList.get(i));
       addToFileString("LW" + " s" + (i + 1) + ", " + i*8 +"(a0)");
   }
   if(mList.size() > 11)
   {
       for(int i = 11; i < mList.size(); i++)
       {
           memStorage.add(mList.get(i));
       }
   }
}
```

Due to this change in register allocation, correctly referencing registers and memory were slightly modified. Often when building the assembly codes, either memLookup or memLookupOrAdd will be called. These functions take a string representing a variable name, and they look through the register and memory ArrayLists to return the correct string that should be added to the opcode. For example, if the variable “x” is found in the tempRegisters ArrayList at index 2, then memLookup will return “t2” to the calling function. These two functions were modified so that the s-registers will also be searched. S-register lookup takes precedence over memStorage lookup.

The main issue that was encountered in this part of development was mostly misunderstanding the instructions. When I began coding, I put the first 11 variables only in

the s-registers, not storing them in memory. I also forgot to save the data in those registers back to memory at the end. These errors were noted and fixed in testing.

AssemblyFunctions.java

```
package CodeGeneration.Intermediate;

import java.io.BufferedOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.OutputStream;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

import static java.nio.file.StandardOpenOption.APPEND;
import static java.nio.file.StandardOpenOption.CREATE;

public class AssemblyFunctions {

   public List<String> registers;
   public List<String> tempRegisters;
   public List<String> memStorage;
   public List<String> sRegisters;
   public Stack<String> whileStack;
   public Stack<String> elseStack;
   public Stack<String> exitStack;
   public int elseExitCounter;
   public int notCounter;
   public int mulCounter;
   private int whileExitCounter;
   public boolean ISDEBUGGING;
   private String fileString;

   public AssemblyFunctions()
   {
       registers = new ArrayList<>();
       tempRegisters = new ArrayList<>();
       memStorage = new ArrayList<>();
       sRegisters = new ArrayList<>();
       whileStack = new Stack<>();
       elseStack = new Stack<>();
       exitStack = new Stack<>();
       elseExitCounter = 0;
       notCounter = 0;
       mulCounter = 0;
       whileExitCounter = 0;
       ISDEBUGGING = false;
       fileString = "";
   }

   private void addToFileString(String toAdd)
   {
       fileString += ("\t\t" + toAdd + "\n");
   }
```

```
private void addLabelToFile(String toAdd)
{
    fileString += ("\t" + toAdd + "\n");
}

public void addPrefix(String prefix){
    fileString = prefix + fileString;
}

public void addSuffix(String suffix){
    fileString = fileString + suffix;
}

public void addHeader()
{
    fileString += ".extern main\n";
    fileString += "main:\n";
}

public void printFileString()
{
    System.out.println(fileString);
}

public void initializeMemory(List<String> mList)
{
    for(int i = 0; (i < 11 && i < mList.size()); i++)
    {
        memStorage.add(mList.get(i));
        sRegisters.add(mList.get(i));
        addToFileString("LW" + " s" + (i + 1) + ", " + i*8 +"(a0)");
    }
    if(mList.size() > 11)
    {
        for(int i = 11; i < mList.size(); i++)
        {
            memStorage.add(mList.get(i));
        }
    }

    //System.out.println("sRegs, size "+ sRegisters.size() + ":");
    for (String sRegister : sRegisters) {
        //System.out.println(sRegister);
    }
    //System.out.println("memory, size " + memStorage.size() + ":");
    for (String s : memStorage) {
        //System.out.println(s);
    }
}

public void saveMemory()
{
    for(int i = 0; i < sRegisters.size(); i++)
    {
        addToFileString("SW s" + (i + 1) + ", " + (8 * i) + "(a0)");
    }
}
```

```
public void printToFile(String fileName)
{
    String filename = ".\\output.s";

    File file = new File(fileName);

    try{
        boolean result = Files.deleteIfExists(file.toPath());
    }catch (IOException x){
        System.err.println(x);
    }

    byte[] data = fileString.getBytes();
    Path p = Paths.get(fileName);

    try (OutputStream out = new BufferedOutputStream(
            Files.newOutputStream(p, CREATE, APPEND))) {
        out.write(data, 0, data.length);
    } catch (IOException x) {
        System.err.println(x);
    }
}

private void addVariable(String variableName)
{
    if(registers.size() < 8)
    {
        registers.add(variableName);
    }
    else
    {
        //Handle changing full registers here
    }
}

private void addTempVariable(String variableName)
{
    tempRegisters.add(variableName);
}

private String memLookupStore(String variableName)
{
    if(sRegisters.indexOf(variableName) == -1) {
        int memIndex = memStorage.indexOf(variableName);
        if (memIndex > -1) {
            String tempName = tempLookupOrAdd(variableName);
            //System.out.println("LW" + " " + tempName + ", " + memIndex*8 +"(a0)");
            return ("LW" + " " + tempName + ", " + memIndex * 8 + "(a0)\n");
            //return "t" + tempRegisters.indexOf(tempName);
        }
    }
    return "";
}

private String regLookupNoFile(String variableName)
{
    int index = sRegisters.indexOf(variableName);
    if(index > -1)
        return "s" + (index + 1); index = registers.indexOf(variableName);
```

```
    if(index > -1)
        return "a" + (index + 1);
    else
    {
        index = tempRegisters.indexOf(variableName);
        if(index > -1)
            return "t" + index;
        else
            return "VARIABLE NOT FOUND";
    }
}

private String regLookup(String variableName)
{

    int index = sRegisters.indexOf(variableName);
    if(index > -1)
        return "s" + (index + 1); index = registers.indexOf(variableName);
    int memIndex = memStorage.indexOf(variableName);
    if( memIndex > -1)
    {
        String tempName = tempLookupOrAdd(variableName);
        //System.out.println("LW" + " " + tempName + ", " + memIndex*8 +"(a0)");
        addToFileString("LW" + " " + tempName + ", " + memIndex*8 +"(a0)");
        //return "t" + tempRegisters.indexOf(tempName);
    }
    if(index > -1)
        return "a" + (index + 1);
    else
    {
        index = tempRegisters.indexOf(variableName);
        if(index > -1)
            return "t" + index;
        else
            return "VARIABLE NOT FOUND";
    }
}

private int memIndexLookup(String variableName)
{
    int index = memStorage.indexOf(variableName);
    if(index > -1)
        return index * 8;
    else
    {
        return -1;
    }
}

private String tempLookupOrAdd(String variableName)
{
    int index = tempRegisters.indexOf(variableName);
    if(index > -1)
        return "t" + index;
    else
    {
        for(int i = 0; i < tempRegisters.size(); i++)
        {
            if(tempRegisters.get(i).equals("REP"))
```

```
                {
                    tempRegisters.set(i, variableName);
                    return ("t" + i);
                }
            }
            addTempVariable(variableName);
            return "t" + (tempRegisters.size() - 1);
        }
    }

    //Does not actually remove the register, but replaces it with a dummy REP value
    private void removeTemp(String variableName)
    {
        int index = tempRegisters.indexOf(variableName);
        if(index > -1)
        {
            tempRegisters.set(index, "REP");
        }
    }

    private String regLookupOrAdd(String variableName)
    {
        int index = sRegisters.indexOf(variableName);
        if(index > -1)
            return "s" + (index + 1);
        index = registers.indexOf(variableName);
        if(index > -1)
            return "a" + (index + 1);
        else
        {
            addVariable(variableName);
            return "a" + (registers.indexOf(variableName) + 1);
        }
    }

    private void clearARegs()
    {
        registers.clear();
    }

    private String getFirstEmptyTempIndex(int offset)
    {
        int index = tempRegisters.size() + offset;
        return ("t" + index);
    }

    private void CheckIfNeedToSave(String destination)
    {
        if(sRegisters.contains(destination)) {
        }
        else
        {
            addToFileString("SW " + regLookupOrAdd(destination) + ", " + memIndexLookup(destination)  +
"(a0)");
        }
    }

    public void ASSIGN(String destination, String operand1)
    {
```

```
        if(ISDEBUGGING)
            System.out.println("ADD " + regLookupOrAdd(destination) + ", x0" + ", " +
regLookup(operand1));
        addToFileString("ADD " + regLookupOrAdd(destination) + ", x0" + ", " + regLookup(operand1));
        CheckIfNeedToSave(destination);
        tempRegisters.clear();
        clearARegs();
    }

    public void ASSIGN(String destination, int value)
    {
        if(ISDEBUGGING)
            System.out.println("ADDI " + regLookupOrAdd(destination) + ", x0" + ", " + value);
        addToFileString("ADDI " + regLookupOrAdd(destination) + ", x0" + ", " + value);
        CheckIfNeedToSave(destination);
        tempRegisters.clear();
        clearARegs();
    }

    public void ADD(String destination, String operand1, String operand2)
    {
        if(ISDEBUGGING)
            System.out.println("ADD " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", "
+ regLookup(operand2));


        addToFileString("ADD " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", " +
regLookup(operand2));
        removeTemp(operand1);
        removeTemp(operand2);
    }

    public void ADDI(String destination, String operand1, int immediateValue)
    {
        if(ISDEBUGGING)
            System.out.println("ADDI " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", "
+ immediateValue);
        addToFileString("ADDI " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", " +
immediateValue);
        removeTemp(operand1);
    }

    public void ADDI(String destination, int immediateValue, String operand2)
    {
        if(ISDEBUGGING)
            System.out.println("ADDI " + tempLookupOrAdd(destination) + ", " + regLookup(operand2) + ", "
+ immediateValue);
        addToFileString("ADDI " + tempLookupOrAdd(destination) + ", " + regLookup(operand2) + ", " +
immediateValue);
        removeTemp(operand2);
    }

    public void ADDI(String destination, int value1, int value2)
    {
        String temp0 = getFirstEmptyTempIndex(0);
        if(ISDEBUGGING) {
            System.out.println("ADDI " + temp0 + ", x0, " + value1);
            System.out.println("ADDI " + tempLookupOrAdd(destination) + ", " + temp0 + ", " + value2);
        }
```

```
        addToFileString("ADDI " + temp0 + ", x0, " + value1);
        addToFileString("ADDI " + tempLookupOrAdd(destination) + ", " + temp0 + ", " + value2);
    }

    public void SUB(String destination, String operand1, String operand2)
    {
        if(ISDEBUGGING)
            System.out.println("SUB " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", "
+ regLookup(operand2));
        addToFileString("SUB " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", " +
regLookup(operand2));
        removeTemp(operand1);
        removeTemp(operand2);
    }

    public void SUB(String destination, String operand1, int immediateValue)
    {
        String temp0 = getFirstEmptyTempIndex(0);
        if(ISDEBUGGING)
        {
            System.out.println("ADDI " + temp0 + ", x0, " + immediateValue);
            System.out.println("SUB " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", "
+ temp0);
        }
        addToFileString("ADDI " + temp0 + ", x0, " + immediateValue);
        addToFileString("SUB " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", " +
temp0);
        removeTemp(operand1);
    }
    public void SUB(String destination, int immediateValue, String operand2)
    {
        String temp0 = getFirstEmptyTempIndex(0);
        if(ISDEBUGGING) {
            System.out.println("ADDI " + temp0 + ", x0, " + immediateValue);
            System.out.println("SUB " + tempLookupOrAdd(destination) + ", " + temp0 + ", " +
regLookup(operand2));
        }
        addToFileString("ADDI " + temp0 + ", x0, " + immediateValue);
        addToFileString("SUB " + tempLookupOrAdd(destination) + ", " + temp0 + ", " +
regLookup(operand2));
        removeTemp(operand2);
    }
    public void SUB(String destination, int operand1, int operand2)
    {
        String temp0 = getFirstEmptyTempIndex(0);
        if(ISDEBUGGING)
            System.out.println("ADDI " + temp0 + ", x0, " + operand1);
        addToFileString("ADDI " + temp0 + ", x0, " + operand1);
        String temp1 = getFirstEmptyTempIndex(1);
        if(ISDEBUGGING)
        {
            System.out.println("ADDI " + temp0 + ", x0, " + operand2);
            System.out.println("SUB " + tempLookupOrAdd(destination) + ", " + temp0 + ", " + temp1);
        }
        addToFileString("ADDI " + temp0 + ", x0, " + operand2);
        addToFileString("SUB " + tempLookupOrAdd(destination) + ", " + temp0 + ", " + temp1);
    }

    public void MUL(String destination, String operand1, String operand2)
```

```
   {
       if(ISDEBUGGING)
           System.out.println("MUL " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", "
+ regLookup(operand2));
       addToFileString("MUL " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", " +
regLookup(operand2));
       removeTemp(operand1);
       removeTemp(operand2);
   }

   public void MUL(String destination, String operand1, int immediateValue)
   {
       String temp0 = getFirstEmptyTempIndex(0);
       if(ISDEBUGGING) {
           System.out.println("ADDI " + temp0 + ", x0, " + immediateValue);
           System.out.println("MUL " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", "
+ temp0);
       }
       addToFileString("ADDI " + temp0 + ", x0, " + immediateValue);
       addToFileString("MUL " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", " +
temp0);
       removeTemp(operand1);
   }
   public void MUL(String destination, int operand1, String operand2)
   {
       String temp0 = getFirstEmptyTempIndex(0);
       if(ISDEBUGGING) {
           System.out.println("ADDI " + temp0 + ", x0, " + operand1);
           System.out.println("MUL " + tempLookupOrAdd(destination) + ", " + regLookup(operand2) + ", "
+ temp0);
       }
       addToFileString("ADDI " + temp0 + ", x0, " + operand1);
       addToFileString("MUL " + tempLookupOrAdd(destination) + ", " + regLookup(operand2) + ", " +
temp0);
       removeTemp(operand2);
   }
   public void MUL(String destination, int operand1, int immediateValue)
   {
       String temp0 = getFirstEmptyTempIndex(0);
       if(ISDEBUGGING)
           System.out.println("ADDI " + temp0 + ", x0, " + operand1);
       addToFileString("ADDI " + temp0 + ", x0, " + operand1);
       String temp1 = getFirstEmptyTempIndex(1);
       if(ISDEBUGGING) {
           System.out.println("ADDI " + temp0 + ", x0, " + immediateValue);
           System.out.println("MUL " + tempLookupOrAdd(destination) + ", " + temp0 + ", " + temp1);
       }
       addToFileString("ADDI " + temp1 + ", x0, " + immediateValue);
       addToFileString("MUL " + tempLookupOrAdd(destination) + ", " + temp0 + ", " + temp1);
   }

   public void AND(String destination, String operand1, String operand2)
   {
       if(ISDEBUGGING)
           System.out.println("AND " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", "
+ regLookup(operand2));
       addToFileString("AND " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", " +
regLookup(operand2));
       removeTemp(operand1);
```

```
        removeTemp(operand2);
    }

    public void ANDI(String destination, String operand1, int immediateValue)
    {
        if(ISDEBUGGING)
            System.out.println("ANDI " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", "
+ immediateValue);
        addToFileString("ANDI " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", " +
immediateValue);
        removeTemp(operand1);
    }

    public void ANDI(String destination, int operand1, String operand2)
    {
        if(ISDEBUGGING)
            System.out.println("ANDI " + tempLookupOrAdd(destination) + ", " + regLookup(operand2) + ", "
+ operand1);
        addToFileString("ANDI " + tempLookupOrAdd(destination) + ", " + regLookup(operand2) + ", " +
operand1);
        removeTemp(operand2);
    }

    public void ANDI(String destination, int operand1, int operand2)
    {
        String temp0 = getFirstEmptyTempIndex(0);
        if(ISDEBUGGING) {
            System.out.println("ADDI " + temp0 + ", x0, " + operand1);
            System.out.println("ANDI " + tempLookupOrAdd(destination) + ", " + temp0 + ", " + operand2);
        }
            addToFileString("ADDI " + temp0 + ", x0, " + operand1);
            addToFileString("ANDI " + tempLookupOrAdd(destination) + ", " + temp0 + ", " + operand2);

    }
    public void OR(String destination, String operand1, String operand2)
    {
        if(ISDEBUGGING)
            System.out.println("OR " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", " +
regLookup(operand2));
        addToFileString("OR " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", " +
regLookup(operand2));
        removeTemp(operand1);
        removeTemp(operand2);
    }

    public void ORI(String destination, String operand1, int immediateValue)
    {
        if(ISDEBUGGING)
            System.out.println("ORI " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", "
+ immediateValue);
        addToFileString("ORI " + tempLookupOrAdd(destination) + ", " + regLookup(operand1) + ", " +
immediateValue);
        removeTemp(operand1);
    }

    public void ORI(String destination, int operand1, String operand2)
    {
        if(ISDEBUGGING)
            System.out.println("ORI " + tempLookupOrAdd(destination) + ", " + regLookup(operand2) + ", "
```

```
+ operand1);
        addToFileString("ORI " + tempLookupOrAdd(destination) + ", " + regLookup(operand2) + ", " +
operand1);
        removeTemp(operand2);
    }

    public void ORI(String destination, int operand1, int operand2)
    {
        String temp0 = getFirstEmptyTempIndex(0);
        if(ISDEBUGGING) {
            System.out.println("ADDI " + temp0 + ", x0, " + operand1);
            System.out.println("ORI " + tempLookupOrAdd(destination) + ", " + temp0 + ", " + operand2);
        }
        addToFileString("ADDI " + temp0 + ", x0, " + operand1);
        addToFileString("ORI " + tempLookupOrAdd(destination) + ", " + temp0 + ", " + operand2);
    }

    public void WHILE(String label, String checkType, String reg1, String reg2)
    {
        if(ISDEBUGGING)
            System.out.println(label + ":");
        addInitialWhileCheck(checkType,reg1,reg2);
        addLabelToFile(label + ":");
        String endLabel = "\n\tOD" + whileExitCounter +":";
        whileExitCounter++;
        switch (checkType) {
            case "=" -> whileStack.push(BEQ(reg1, reg2, label) + endLabel);
            case ">" -> whileStack.push(BGT(reg1, reg2, label) + endLabel);
            case ">=" -> whileStack.push(BGE(reg1, reg2, label) + endLabel);
            case "<" -> whileStack.push(BLT(reg1, reg2, label) + endLabel);
            case "<=" -> whileStack.push(BLE(reg1, reg2, label) + endLabel);
            default -> System.out.println("ERROR: Invalid while argument");
        }
    }

    public void WHILE(String label, String checkType, int reg1, String reg2)
    {
        if(ISDEBUGGING)
            System.out.println(label + ":");
        addInitialWhileCheck(checkType,reg1,reg2);
        addLabelToFile(label + ":");
        String endLabel = "\n\tOD" + whileExitCounter +":";
        whileExitCounter++;
        switch (checkType) {
            case "=" -> whileStack.push(BEQ(reg1, reg2, label) + endLabel);
            case ">" -> whileStack.push(BGT(reg1, reg2, label) + endLabel);
            case ">=" -> whileStack.push(BGE(reg1, reg2, label) + endLabel);
            case "<" -> whileStack.push(BLT(reg1, reg2, label) + endLabel);
            case "<=" -> whileStack.push(BLE(reg1, reg2, label) + endLabel);
            default -> System.out.println("ERROR: Invalid while argument");
        }
    }

    public void WHILE(String label, String checkType, String reg1, int reg2)
    {
        if(ISDEBUGGING)
            System.out.println(label + ":");
        addInitialWhileCheck(checkType,reg1,reg2);
        addLabelToFile(label + ":");
```

```
    String endLabel = "\n\tOD" + whileExitCounter +":";
    whileExitCounter++;
    switch (checkType) {
        case "=" -> whileStack.push(BEQ(reg1, reg2, label) + endLabel);
        case ">" -> whileStack.push(BGT(reg1, reg2, label) + endLabel);
        case ">=" -> whileStack.push(BGE(reg1, reg2, label) + endLabel);
        case "<" -> whileStack.push(BLT(reg1, reg2, label) + endLabel);
        case "<=" -> whileStack.push(BLE(reg1, reg2, label) + endLabel);
        default -> System.out.println("ERROR: Invalid while argument");
    }
}

public void WHILE(String label, String checkType, int reg1, int reg2)
{
    if(ISDEBUGGING)
        System.out.println(label + ":");
    addInitialWhileCheck(checkType,reg1,reg2);
    addLabelToFile(label + ":");
    String endLabel = "\n\tOD" + whileExitCounter +":";
    whileExitCounter++;
    switch (checkType) {
        case "=" -> whileStack.push(BEQ(reg1, reg2, label) + endLabel);
        case ">" -> whileStack.push(BGT(reg1, reg2, label) + endLabel);
        case ">=" -> whileStack.push(BGE(reg1, reg2, label) + endLabel);
        case "<" -> whileStack.push(BLT(reg1, reg2, label) + endLabel);
        case "<=" -> whileStack.push(BLE(reg1, reg2, label) + endLabel);
        default -> System.out.println("ERROR: Invalid while argument");
    }
}

public void IF(String checkType, String reg1, String reg2)
{
    elseStack.push("ELSE" + elseExitCounter);
    exitStack.push("EXIT" + elseExitCounter);
    elseExitCounter++;
    String elseLabel = elseStack.peek();

    switch (checkType) {
        case "=" ->
            //System.out.println(BNE(reg1, reg2, elseLabel));
            addToFileString(BNE(reg1, reg2, elseLabel));
        case ">" ->
            //System.out.println(BLE(reg1, reg2, elseLabel));
            addToFileString(BLE(reg1, reg2, elseLabel));
        case ">=" ->
            //System.out.println(BLT(reg1, reg2, elseLabel));
            addToFileString(BLT(reg1, reg2, elseLabel));
        case "<" ->
            //System.out.println(BGE(reg1, reg2, elseLabel));
            addToFileString(BGE(reg1, reg2, elseLabel));
        case "<=" ->
            //System.out.println(BGT(reg1, reg2, elseLabel));
            addToFileString(BGT(reg1, reg2, elseLabel));
        default -> System.out.println("ERROR: Invalid if argument");
    }
}
public void IF(String checkType, int reg1, String reg2)
{
    elseStack.push("ELSE" + elseExitCounter);
```

```
        exitStack.push("EXIT" + elseExitCounter);
        elseExitCounter++;
        String elseLabel = elseStack.peek();

        switch (checkType) {
            case "=" ->
                //System.out.println(BNE(reg1, reg2, elseLabel));
                addToFileString(BNE(reg1, reg2, elseLabel));
            case ">" ->
                // System.out.println(BLE(reg1, reg2, elseLabel));
                addToFileString(BLE(reg1, reg2, elseLabel));
            case ">=" ->
                //System.out.println(BLT(reg1, reg2, elseLabel));
                addToFileString(BLT(reg1, reg2, elseLabel));
            case "<" ->
                //System.out.println(BGE(reg1, reg2, elseLabel));
                addToFileString(BGE(reg1, reg2, elseLabel));
            case "<=" ->
                //System.out.println(BGT(reg1, reg2, elseLabel));
                addToFileString(BGT(reg1, reg2, elseLabel));
            default -> System.out.println("ERROR: Invalid if argument");
        }
    }
    public void IF(String checkType, String reg1, int reg2)
    {
        elseStack.push("ELSE" + elseExitCounter);
        exitStack.push("EXIT" + elseExitCounter);
        elseExitCounter++;
        String elseLabel = elseStack.peek();

        switch (checkType) {
            case "=" ->
                //System.out.println(BNE(reg1, reg2, elseLabel));
                addToFileString(BNE(reg1, reg2, elseLabel));
            case ">" ->
                //System.out.println(BLE(reg1, reg2, elseLabel));
                addToFileString(BLE(reg1, reg2, elseLabel));
            case ">=" ->
                // System.out.println(BLT(reg1, reg2, elseLabel));
                addToFileString(BLT(reg1, reg2, elseLabel));
            case "<" ->
                //System.out.println(BGE(reg1, reg2, elseLabel));
                addToFileString(BGE(reg1, reg2, elseLabel));
            case "<=" ->
                //System.out.println(BGT(reg1, reg2, elseLabel));
                addToFileString(BGT(reg1, reg2, elseLabel));
            default -> System.out.println("ERROR: Invalid if argument");
        }
    }
    public void IF(String checkType, int reg1, int reg2)
    {
        elseStack.push("ELSE" + elseExitCounter);
        exitStack.push("EXIT" + elseExitCounter);
        elseExitCounter++;
        String elseLabel = elseStack.peek();

        switch (checkType) {
            case "=" ->
                // System.out.println(BNE(reg1, reg2, elseLabel));
```

```
            addToFileString(BNE(reg1, reg2, elseLabel));
        case ">" ->
            // System.out.println(BLE(reg1, reg2, elseLabel));
            addToFileString(BLE(reg1, reg2, elseLabel));
        case ">=" ->
            // System.out.println(BLT(reg1, reg2, elseLabel));
            addToFileString(BLT(reg1, reg2, elseLabel));
        case "<" ->
            //System.out.println(BGE(reg1, reg2, elseLabel));
            addToFileString(BGE(reg1, reg2, elseLabel));
        case "<=" ->
            //System.out.println(BGT(reg1, reg2, elseLabel));
            addToFileString(BGT(reg1, reg2, elseLabel));
        default -> {
        }
        // System.out.println("ERROR: Invalid if argument");
    }
}

public void ELSE()
{
    UNCONDITIONALJUMP(exitStack.peek());
    String elseLabel = elseStack.pop();
    if(ISDEBUGGING)
        System.out.println(elseLabel + ":");
    addLabelToFile(elseLabel + ":");
}

public void FI()
{
    String fiLabel = exitStack.pop();
    if(ISDEBUGGING)
        System.out.println(fiLabel + ":");
    addLabelToFile(fiLabel + ":");
}

public void OD()
{
    String odLabel = whileStack.pop();
    if(ISDEBUGGING)
        System.out.println(odLabel);
    addToFileString(odLabel);
}

private String BEQ(String reg1, String reg2, String label)
{
    String buffer = memLookupStore(reg1);
    buffer += "\t\t" + memLookupStore(reg2);
    buffer += ("\t\tBEQ " + regLookupNoFile(reg1) + ", " + regLookupNoFile(reg2) + ", " + label);
    return buffer;
}

private String BEQ(int reg1, String reg2, String label)
{
    String buffer = ("ADDI t6, x0, " + reg1 + "\n");
    buffer += "\t\t" +memLookupStore(reg2);
    buffer += ("\t\tBEQ t6, " + regLookupNoFile(reg2) + ", " + label);
    return buffer;
}
```

```
private String BEQ(String reg1, int reg2, String label)
{
    String buffer = ("ADDI t6, x0, " + reg2 + "\n");
    buffer += "\t\t" + memLookupStore(reg1);
    buffer += ("\t\tBEQ " + regLookupNoFile(reg1) + ", t6, " + label);
    return buffer;
}

private String BEQ(int reg1, int reg2, String label)
{
    String buffer = ("ADDI t6, x0, " + reg1 + "\n");
    buffer += ("\t\tADDI t5, x0, " + reg2 + "\n");
    buffer += ("\t\tBEQ t6, t5, " + label);
    return buffer;
}


private String BNE(String reg1, String reg2, String label)
{
    String buffer = memLookupStore(reg1);
    buffer += "\t\t" + memLookupStore(reg2);
    buffer += ("\t\tBNE " + regLookupNoFile(reg1) + ", " + regLookupNoFile(reg2) + ", " + label);
    return buffer;
}

private String BNE(int reg1, String reg2, String label)
{
    String buffer = ("ADDI t6, x0, " + reg1 + "\n");
    buffer += "\t\t" + memLookupStore(reg2);
    buffer += ("\t\tBNE t6, " + regLookupNoFile(reg2) + ", " + label);
    return buffer;
}

private String BNE(String reg1, int reg2, String label)
{
    String buffer = ("ADDI t6, x0, " + reg2 + "\n");
    buffer += "\t\t" + memLookupStore(reg1);
    buffer += ("\t\tBNE " + regLookupNoFile(reg1) + ", t6, " + label);
    return buffer;
}

private String BNE(int reg1, int reg2, String label)
{
    String buffer = ("ADDI t6, x0, " + reg1 + "\n");
    buffer += ("\t\tADDI t5, x0, " + reg2 + "\n");
    buffer += ("\t\tBNE t6, t5, " + label);
    return buffer;
}

private String BLT(String reg1, String reg2, String label)
{
    String buffer = memLookupStore(reg1);
    buffer += "\t\t" +memLookupStore(reg2);
    buffer += ("\t\tBLT " + regLookupNoFile(reg1) + ", " + regLookupNoFile(reg2) + ", " + label);
    return buffer;
}

private String BLT(int reg1, String reg2, String label)
```

```
{
    String buffer = ("ADDI t6, x0, " + reg1 + "\n");
    buffer += "\t\t" + memLookupStore(reg2);
    buffer += ("\t\tBLT t6, " + regLookupNoFile(reg2) + ", " + label);
    return buffer;
}

private String BLT(String reg1, int reg2, String label)
{
    String buffer = ("ADDI t6, x0, " + reg2 + "\n");
    buffer += "\t" +memLookupStore(reg1);
    buffer += ("\tBLT " + regLookupNoFile(reg1) + ", t6, " + label);
    return buffer;
}

private String BLT(int reg1, int reg2, String label)
{
    String buffer = ("ADDI t6, x0, " + reg1 + "\n");
    buffer += ("\t\tADDI t5, x0, " + reg2 + "\n");
    buffer += ("\t\tBLT t6, t5, " + label);
    return buffer;
}

private String BGE(String reg1, String reg2, String label)
{
    String buffer = memLookupStore(reg1);
    buffer += "\t\t" + memLookupStore(reg2);
    buffer +=("\t\tBGE " + regLookupNoFile(reg1) + ", " + regLookupNoFile(reg2) + ", " + label);
    return buffer;
}

private String BGE(int reg1, String reg2, String label)
{
    String buffer = ("ADDI t6, x0, " + reg1 + "\n");
    buffer += "\t\t" + memLookupStore(reg2);
    buffer += ("\t\tBGE t6, " + regLookupNoFile(reg2) + ", " + label);
    return buffer;
}

private String BGE(String reg1, int reg2, String label)
{
    String buffer = ("ADDI t6, x0, " + reg2 + "\n");
    buffer += "\t\t" + memLookupStore(reg1);
    buffer += ("\t\tBGE " + regLookupNoFile(reg1) + ", t6, " + label);
    return buffer;
}

private String BGE(int reg1, int reg2, String label)
{
    String buffer = ("ADDI t6, x0, " + reg1 + "\n");
    buffer += ("\t\tADDI t5, x0, " + reg2 + "\n");
    buffer += ("\t\tBGE t6, t5, " + label);
    return buffer;
}

private String BLE(String reg1, String reg2, String label)
{
    String buffer = memLookupStore(reg1);
    buffer += "\t\t" + memLookupStore(reg2);
```

```
    //Add 1 to reg2 and store it in t0
    buffer += ("\t\tADDI " + "t6, " + regLookupNoFile(reg2) + " 1\n");
    //Use BLT comparison on reg1 and t0
    buffer += ("\t\tBLT " + regLookupNoFile(reg1) + ", t6, " + label);
    return buffer;
}

private String BLE(int reg1, String reg2, String label)
{
    String buffer = memLookupStore(reg2);
    buffer += ("\t\tADDI " + "t5, " + regLookupNoFile(reg2) + " 1\n");
    buffer += ("\t\tADDI t6, x0, " + reg1 + "\n");
    buffer += ("\t\tBLT t6, t5, " + label);
    return buffer;
}

private String BLE(String reg1, int reg2, String label)
{
    //Store value 1 in t0
    String buffer = ("ADDI t6, x0, 1\n");
    //Add that 1 to reg2 and store in t0
    buffer += ("\t\tADDI t6, t6, " + reg2 + "\n");
    buffer += "\t\t" + memLookupStore(reg1);
    buffer += ("\t\tBLT " + regLookupNoFile(reg1) + ", t6, " + label);
    return buffer;
}

private String BLE(int reg1, int reg2, String label)
{
    //Store value 1 in t0
    String buffer = ("ADDI t6, x0, 1\n");
    //Add that 1 to reg2 and store in t0
    buffer += ("\t\tADDI t6, t6, " + reg2 + "\n");
    //Add reg1 into t1
    buffer += ("\t\tADDI t5, x0, " + reg1 + "\n");
    //Use BLT comparison on reg1 and t0
    buffer += ("\t\tBLT t5, t6, " + label);
    return buffer;
}


private String BGT(String reg1, String reg2, String label)
{
    String buffer = memLookupStore(reg1);
    buffer += "\t\t" +memLookupStore(reg2);
    buffer += ("\t\tADDI t6, " + regLookupNoFile(reg2) + ", 1\n");
    buffer += ("\t\tBGE " + regLookupNoFile(reg1) + ", t6, " + label);
    return buffer;
}

private String BGT(int reg1, String reg2, String label)
{
    //Add 1 to reg2 and store in t0
    String buffer = memLookupStore(reg2);
    buffer += ("\t\tADDI t6, " + regLookupNoFile(reg2) + ", 1\n");
    //Store value of reg1 in t1
    buffer += ("\t\tADDI t5, x0, " + reg1 + "\n");
    buffer += ("\t\tBGE t5, t6, " + label);
    return buffer;
```

```
}

private String BGT(String reg1, int reg2, String label)
{
    //store value in reg2 in t0
    String buffer = ("ADDI t6, x0, " + reg2 + "\n");
    //add 1 to value in t0
    buffer += ("\t\tADDI t6, t6, 1\n");
    buffer += "\t\t" + memLookupStore(reg1);
    buffer += ("\t\tBGE " + regLookupNoFile(reg1) + ", t6, " + label);
    return buffer;
}

private String BGT(int reg1, int reg2, String label)
{
    //store value in reg1 in t0
    String buffer = ("ADDI t0, x0, " + reg1 + "\n");
    //store value in reg2 in t1
    buffer += ("\t\tADDI t1, x0, " + reg2 + "\n");
    //add 1 to value in t1
    buffer += ("\t\tADDI t1, t1, 1\n");
    buffer += ("\t\tBGE t0, t1, " + label);
    return buffer;
}

private void UNCONDITIONALJUMP(String label)
{
    if(ISDEBUGGING)
        System.out.println("BEQ x0, x0, " + label);
    addToFileString("BEQ x0, x0, " + label);
}

private void addInitialWhileCheck(String checkType, String reg1, String reg2)
{
    String loopEndLabel = "OD" + whileExitCounter;
    switch (checkType) {
        case "=" ->
            //System.out.println(BNE(reg1, reg2, loopEndLabel));
            addToFileString(BNE(reg1, reg2, loopEndLabel));
        case ">" ->
            //System.out.println(BLE(reg1, reg2, loopEndLabel));
            addToFileString(BLE(reg1, reg2, loopEndLabel));
        case ">=" ->
            //System.out.println(BLT(reg1, reg2, loopEndLabel));
            addToFileString(BLT(reg1, reg2, loopEndLabel));
        case "<" ->
            //System.out.println(BGE(reg1, reg2, loopEndLabel));
            addToFileString(BGE(reg1, reg2, loopEndLabel));
        case "<=" ->
            //System.out.println(BGT(reg1, reg2, loopEndLabel));
            addToFileString(BGT(reg1, reg2, loopEndLabel));
        default -> System.out.println("ERROR: Invalid if argument");
    }
}
private void addInitialWhileCheck(String checkType, String reg1, int reg2)
{
    String loopEndLabel = "OD" + whileExitCounter;
    switch (checkType) {
        case "=" ->
```

```
            //System.out.println(BNE(reg1, reg2, loopEndLabel));
            addToFileString(BNE(reg1, reg2, loopEndLabel));
        case ">" ->
            //System.out.println(BLE(reg1, reg2, loopEndLabel));
            addToFileString(BLE(reg1, reg2, loopEndLabel));
        case ">=" ->
            //System.out.println(BLT(reg1, reg2, loopEndLabel));
            addToFileString(BLT(reg1, reg2, loopEndLabel));
        case "<" ->
            //System.out.println(BGE(reg1, reg2, loopEndLabel));
            addToFileString(BGE(reg1, reg2, loopEndLabel));
        case "<=" ->
            //System.out.println(BGT(reg1, reg2, loopEndLabel));
            addToFileString(BGT(reg1, reg2, loopEndLabel));
        default -> System.out.println("ERROR: Invalid if argument");
    }
}
private void addInitialWhileCheck(String checkType, int reg1, String reg2)
{
    String loopEndLabel = "OD" + whileExitCounter;
    switch (checkType) {
        case "=" ->
            //System.out.println(BNE(reg1, reg2, loopEndLabel));
            addToFileString(BNE(reg1, reg2, loopEndLabel));
        case ">" ->
            //System.out.println(BLE(reg1, reg2, loopEndLabel));
            addToFileString(BLE(reg1, reg2, loopEndLabel));
        case ">=" ->
            //System.out.println(BLT(reg1, reg2, loopEndLabel));
            addToFileString(BLT(reg1, reg2, loopEndLabel));
        case "<" ->
            // System.out.println(BGE(reg1, reg2, loopEndLabel));
            addToFileString(BGE(reg1, reg2, loopEndLabel));
        case "<=" ->
            //System.out.println(BGT(reg1, reg2, loopEndLabel));
            addToFileString(BGT(reg1, reg2, loopEndLabel));
        default -> {
        }
        //System.out.println("ERROR: Invalid if argument");
    }
}
private void addInitialWhileCheck(String checkType, int reg1, int reg2)
{
    String loopEndLabel = "OD" + whileExitCounter;
    switch (checkType) {
        case "=" ->
            //System.out.println(BNE(reg1, reg2, loopEndLabel));
            addToFileString(BNE(reg1, reg2, loopEndLabel));
        case ">" ->
            //System.out.println(BLE(reg1, reg2, loopEndLabel));
            addToFileString(BLE(reg1, reg2, loopEndLabel));
        case ">=" ->
            //System.out.println(BLT(reg1, reg2, loopEndLabel));
            addToFileString(BLT(reg1, reg2, loopEndLabel));
        case "<" ->
            //System.out.println(BGE(reg1, reg2, loopEndLabel));
            addToFileString(BGE(reg1, reg2, loopEndLabel));
        case "<=" ->
            //System.out.println(BGT(reg1, reg2, loopEndLabel));
```

```
            addToFileString(BGT(reg1, reg2, loopEndLabel));
        default -> System.out.println("ERROR: Invalid if argument");
    }
}
public void NOT(String variableName)
{
    if(ISDEBUGGING) {
        System.out.println("BNE " + regLookup(variableName) + ", x0, ELSENOT" + notCounter);
        System.out.println("ADDI t0, x0, 1");
    }
    addToFileString("BNE " + regLookup(variableName) + ", x0, ELSENOT" + notCounter);
    addToFileString("ADDI t0, x0, 1");
    UNCONDITIONALJUMP("EXITNOT" + notCounter);
    if(ISDEBUGGING) {
        System.out.println("ELSENOT" + notCounter + ":");
        System.out.println("ADD t0, x0, x0");
        System.out.println("EXITNOT" + notCounter + ":");
    }
    addToFileString("ELSENOT" + notCounter + ":");
    addToFileString("ADD t0, x0, x0");
    addToFileString("EXITNOT" + notCounter + ":");

    notCounter++;
}

public void SKIP()
{
    addToFileString("ADDI x0, x0, 0");
}

public void MULV2(String destination, String operand1, String operand2)
{
    //Jump to end if either is 0
    String returnValue = getFirstEmptyTempIndex(2);
    System.out.println("ADD " + returnValue + ", x0, x0");
    System.out.println("BEQ x0, " + regLookup(operand1) + ", MULEXIT" + mulCounter);
    System.out.println("BEQ x0, " + regLookup(operand2) + ", MULEXIT" + mulCounter);

    //Store both initial values in temp vars
    String val1Store = getFirstEmptyTempIndex(1);
    String val2Store = getFirstEmptyTempIndex(0);
    String absolute1Store = getFirstEmptyTempIndex(3);
    System.out.println("ADD " + val1Store + ", x0, " + regLookup(operand1));
    System.out.println("ADD " + val2Store + ", x0, " + regLookup(operand2));
    System.out.println("ADDI " + absolute1Store + ", x0, 1");

    //Main body. Add val1 to returnValue. decrement val2. check if val2 equals 0. loop if not
    System.out.println("MULBODY" + mulCounter + ":");
    System.out.println("ADD " + returnValue + ", " + returnValue + ", " + val1Store);
    System.out.println("SUB " + val2Store + ", " + val2Store + ", " + absolute1Store);
    System.out.println("BNE " + val2Store + ", x0, MULBODY" + mulCounter);

    //Set the destination value to the result value. Increment the mulcounter
    System.out.println("MULEXIT" + mulCounter + ":");
    System.out.println("ADD " + tempLookupOrAdd(destination) + ", x0, " + returnValue);
    mulCounter++;
}

public void MULV2(String destination, int operand1, String operand2)
```

```
{
    //if operand2 is zeros, just assign zero to the destination
    if(operand1 == 0)
    {
        System.out.println("ADD " + destination + ", x0, x0");
    }
    else
    {
        //Jump to end if either is 0
        String returnValue = getFirstEmptyTempIndex(2);
        System.out.println("ADD " + returnValue + ", x0, x0");
        System.out.println("BEQ x0, " + regLookup(operand2) + ", MULEXIT" + mulCounter);

        //Store both initial values in temp vars
        String val1Store = getFirstEmptyTempIndex(1);
        String val2Store = getFirstEmptyTempIndex(0);
        String absolute1Store = getFirstEmptyTempIndex(3);
        System.out.println("ADDI " + val1Store + ", x0, " + operand1);
        System.out.println("ADD " + val2Store + ", x0, " + regLookup(operand2));
        System.out.println("ADDI " + absolute1Store + ", x0, 1");

        //Main body. Add val1 to returnValue. decrement val2. check if val2 equals 0. loop if not
        System.out.println("MULBODY" + mulCounter + ":");
        System.out.println("ADD " + returnValue + ", " + returnValue + ", " + val1Store);
        System.out.println("SUB " + val2Store + ", " + val2Store + ", " + absolute1Store);
        System.out.println("BNE " + val2Store + ", x0, MULBODY" + mulCounter);

        //Set the destination value to the result value. Increment the mulcounter
        System.out.println("MULEXIT" + mulCounter + ":");
        System.out.println("ADD " + tempLookupOrAdd(destination) + ", x0, " + returnValue);
        mulCounter++;
    }
}

public void MULV2(String destination, String operand1, int operand2)
{
    //if operand2 is zeros, just assign zero to the destination
    if(operand2 == 0)
    {
        System.out.println("ADD " + destination + ", x0, x0");
    }
    else
    {
        //Jump to end if either is 0
        String returnValue = getFirstEmptyTempIndex(2);
        System.out.println("ADD " + returnValue + ", x0, x0");
        System.out.println("BEQ x0, " + regLookup(operand1) + ", MULEXIT" + mulCounter);

        //Store both initial values in temp vars
        String val1Store = getFirstEmptyTempIndex(1);
        String val2Store = getFirstEmptyTempIndex(0);
        String absolute1Store = getFirstEmptyTempIndex(3);
        System.out.println("ADD " + val1Store + ", x0, " + regLookup(operand1));
        System.out.println("ADDI " + val2Store + ", x0, " + operand2);
        System.out.println("ADDI " + absolute1Store + ", x0, 1");

        //Main body. Add val1 to returnValue. decrement val2. check if val2 equals 0. loop if not
        System.out.println("MULBODY" + mulCounter + ":");
        System.out.println("ADD " + returnValue + ", " + returnValue + ", " + val1Store);
```

```
            System.out.println("SUB " + val2Store + ", " + val2Store + ", " + absolute1Store);
            System.out.println("BNE " + val2Store + ", x0, MULBODY" + mulCounter);

            //Set the destination value to the result value. Increment the mulcounter
            System.out.println("MULEXIT" + mulCounter + ":");
            System.out.println("ADD " + tempLookupOrAdd(destination) + ", x0, " + returnValue);
            mulCounter++;
        }
    }

    public void MULV2(String destination, int operand1, int operand2) {
        //if operand2 is zeros, just assign zero to the destination
        if (operand2 == 0 || operand1 == 0) {
            System.out.println("ADD " + destination + ", x0, x0");
        } else {
            //Jump to end if either is 0
            String returnValue = getFirstEmptyTempIndex(2);
            System.out.println("ADD " + returnValue + ", x0, x0");

            //Store both initial values in temp vars
            String val1Store = getFirstEmptyTempIndex(1);
            String val2Store = getFirstEmptyTempIndex(0);
            String absolute1Store = getFirstEmptyTempIndex(3);
            System.out.println("ADDI " + val1Store + ", x0, " + operand1);
            System.out.println("ADDI " + val2Store + ", x0, " + operand2);
            System.out.println("ADDI " + absolute1Store + ", x0, 1");

            //Main body. Add val1 to returnValue. decrement val2. check if val2 equals 0. loop if not
            System.out.println("MULBODY" + mulCounter + ":");
            System.out.println("ADD " + returnValue + ", " + returnValue + ", " + val1Store);
            System.out.println("SUB " + val2Store + ", " + val2Store + ", " + absolute1Store);
            System.out.println("BNE " + val2Store + ", x0, MULBODY" + mulCounter);

            //Set the destination value to the result value. Increment the mulcounter
            System.out.println("ADD " + tempLookupOrAdd(destination) + ", x0, " + returnValue);
            mulCounter++;
        }
    }
}
```

Testing, Performance Evaluation, and Analysis

Multiple unit tests were built to test various aspects of the code generation and optimization. While we did run tests for runtime, the numbers feel unusual enough that we

believe that our method was incorrect. The numbers have still been recorded under each specific test, though not given an overall table.

boolTest1.while

```
n := input;
if n > 2 then
 output := true
else
 output := false
fi;
```

boolTest1 is used to show that the boolean keywords true and false are working. It also shows the if-then-else block working correctly and that the user can give input upon program execution.
Unoptimized compile time: 0.180 seconds
Optimized compile time: 0.266 seconds
GCC compile time: 0.309 seconds
Unoptimized run time: - Could not run
Optimized run time: 0.001223
GCC run time: 0.001258

andTest.while

```
x := true;
y := false;

output := x and y
```

andTest is used to determine that the and keyword is working correctly. It also indirectly shows that true and false are working. Output will always return false.
Unoptimized compile time: 0.189 seconds
Optimized compile time: 0.249 seconds
GCC compile time: 0.305 seconds
Unoptimized run time: 0.001344
Optimized run time: 0.001262
GCC run time: 0.001363

orTest.while

```
x := true;
y := false;

output := x or y
```

orTest is much like andTest, but demonstrates that the or keyword is working. Output always returns true.

Unoptimized compile time: 0.192 seconds
Optimized compile time: 0.233 seconds
GCC compile time: 0.299 seconds
Unoptimized run time: 0.001346
Optimized run time: 0.001265
GCC run time: 0.001294

factorial.while

```
y := input;
z := 1;
while y > 1 do
z := z * y;
y := y - 1
od;
y := 0;
output := z
```

One of the tests given to us in project 1, factorial demonstrates that the while code block works, as well as several arithmetic operators. Output will return the factorial of the input.
Unoptimized compile time: 0.201 seconds
Optimized compile time: 0.264 seconds
GCC compile time: 0.308 seconds
Unoptimized run time: 0.001374
Optimized run time: 0.001198
GCC run time: 0.001223

collatz.while

```
n := input;
steps := 0;
while n > 1 do
   rem := n; --Here we divide n by 2:
   quot := 0;
   while rem > 1 do
       rem := rem - 2;
       quot := quot + 1
   od;
   if rem = 0 then
       n := quot
   else
       n := 3*n+1
   fi;
   steps := steps + 1
od;
output := steps
```

collatz is the other test given to us in project 1. It shows off the ability to have next while and if statements, all arithmetic operators, and arithmetic statements with more than 3 operands.
Unoptimized compile time: 0.278 seconds
Optimized compile time: 0.361 seconds
GCC compile time: 0.308 seconds
Unoptimized run time: 0.001387
Optimized run time: 0.001195
GCC run time: 0.001304

longWhile.while

```
bigNum := 999999999;
y := 1;
while bigNum > 0 do
bigNum := bigNum - 1;
constantFold := bigNum + 9 + 7 * y

od;

output := constantFold
```

longWhile is used as an easy test for benchmarking, as it simply does a single long while loop with the same arithmetic operation performed each time. Output will always equal 1.
Unoptimized compile time: 0.211 seconds
Optimized compile time: 0.251 seconds
GCC compile time: 0.302 seconds
Unoptimized run time: 3.45
Optimized run time: 3.63
GCC run time: 3.16

longFile.while

```
x := 0;
steps := 10;
--nested while loop
while steps > 0 do
   while steps > 2 do
       y := (steps - 2) + ((2 - 1) * (3 + 1))
   od;
   steps := steps - 1
   if y > 9 then
       output := y
   else
       output := 0 --output should always be 0
```

```
   fi
od;

--skip testing for if
if y > 0 then
       skip
   else
       skip
fi;
temp := 0;
if y = 2 then
   temp := 1
else
   skip
fi;

z := temp + 1 * x + temp ;


--factorial of 10
tempy := 10;
tempz := 1;
while tempy > 1 do
tempz := tempz * tempy;
tempy := tempy - 1
od;
tempy := 0;
factorialoutput := tempz
```

LongFile is a combination of some previous files, as well as some new code. It is used as a way to benchmark compiling for longer files. It also shows off some aspects that the other unit tests did not, such as using parentheses to modify the order of arithmetic operations and the skip keyword.
Unoptimized compile time: 0.211 seconds
Optimized compile time: 0.375 seconds
GCC compile time: 0.305 seconds
Unoptimized run time: 0.001364
Optimized run time: - Could not run
GCC run time: - Could not run

Compilation Times Table

	Unoptimized	Optimized	GCC
AndTest	0.189	0.249	0.305
BoolTest	0.18	0.266	0.309
Collatz	0.278	0.361	0.308
Factorial	0.201	0.264	0.308
LongFile	0.264	0.375	0.305
LongWhile	0.211	0.251	0.302
OrTest	0.192	0.233	0.299
Mean	0.216	0.286	0.305

Script to compare execution time
(from https://www.techiedelight.com/find-execution-time-c-program/)

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>       // for clock_t, clock(), CLOCKS_PER_SEC
int main() {
 double time_spent = 0.0;

 clock_t begin = clock();

 system("./a.out 0 0 0 0 ");

 clock_t end = clock();

 time_spent += (double)(end - begin) / CLOCKS_PER_SEC;

 printf("The elapsed time is %f seconds", time_spent);

 return 0;
```

```
}
```

Script to compare compile time
(from
https://stackoverflow.com/questions/1557571/how-do-i-get-time-of-a-python-programs-execution)

```
import os
import subprocess
import time
start_time = time.time()

list_files = subprocess.run(["java", "SimpleCompiler", "longFile.while", "output"])

print("--- %s seconds ---" % (time.time() - start_time))
```

Conclusion

This project required us to integrate a large variety of skills and knowledge bases, many of which we had previously only dealt with in theory or alone. This made for a pretty steep learning curve at various points. At times this was frustrating, and at most times it demanded a lot more time from us than we initially expected. There were multiple instances of realizing that code design from project 1 could help or hinder us. However, when we began bringing our various code portions together, we were pleasantly surprised by the relative seamlessness of the process. This is probably thanks to the relatively isolated work required for the development of our individual project portions, and thanks to frequent communication throughout.