**Computer Science Department**

**Senior Team Portfolio**

Compilers

CSCI 468

Spring Semester of 2019

by

Spencer Debuf, Gregor Smirnow, & Kenneth Taylor

Montana State University

Bozeman, Montana

**Section 1: Program**

grammar Little;

program: 'PROGRAM' id 'BEGIN' pgm\_body 'END';

id: IDENTIFIER;

pgm\_body: decl func\_declarations | decl ;

decl: string\_decl decl | var\_decl decl |;

string\_decl: 'STRING' id ':=' str ';';

str: STRINGLITERAL;

var\_decl: var\_type id\_list ';';

var\_type: 'FLOAT' | 'INT';

any\_type: var\_type | 'VOID';

id\_list: id id\_tail;

id\_tail: ',' id id\_tail |;

param\_decl\_list: param\_decl param\_decl\_tail |;

param\_decl: var\_type id;

param\_decl\_tail: ',' param\_decl param\_decl\_tail |;

func\_declarations: func\_decl func\_declarations | func\_decl;

func\_decl: 'FUNCTION' any\_type id '(' param\_decl\_list ')' 'BEGIN' func\_body 'END';

func\_body: decl stmt\_list;

stmt\_list: stmt stmt\_list\_tail |;

stmt\_list\_tail: stmt stmt\_list\_tail |;

stmt: base\_stmt | if\_stmt | while\_stmt;

base\_stmt: assign\_stmt | read\_stmt | write\_stmt | return\_stmt;

assign\_stmt: assign\_expr ';';

assign\_expr: id ':=' expr;

read\_stmt: 'READ' '(' id\_list ')' ';';

write\_stmt: 'WRITE' '(' id\_list ')' ';';

return\_stmt: 'RETURN' expr ';';

expr: add\_minus\_expression;

add\_minus\_expression: add\_minus\_expression addop add\_minus\_expression | multiply\_divide\_expression;

multiply\_divide\_expression: expression\_component mulop multiply\_divide\_expression | expression\_component;

expression\_component: primary | call\_expr;

call\_expr: id '(' expr\_list ')';

expr\_list: expr expr\_list\_tail |;

expr\_list\_tail: ',' expr expr\_list\_tail |;

primary: '(' expr ')' | id | INTLITERAL | FLOATLITERAL;

addop: '+' | '-';

mulop: '\*' | '/';

if\_stmt: 'IF' '(' cond ')' decl stmt\_list else\_part 'ENDIF' | 'IF' '(' cond ')' decl stmt\_list 'ENDIF' ;

else\_part: 'ELSE' decl stmt\_list;

cond: expr compop expr;

compop: '<' | '>' | '=' | '!=' | '<=' | '>=';

while\_stmt: 'WHILE' '(' cond ')' decl stmt\_list 'ENDWHILE';

IDENTIFIER: LETTER ALPHANUMERIC\*;

INTLITERAL: DIGIT+;

FLOATLITERAL: DIGIT\* '.' DIGIT+;

STRINGLITERAL: '"' ~["]\* '"';

DIGIT: [0-9];

ALPHANUMERIC: [a-zA-Z0-9];

LETTER: [a-zA-Z];

COMMENT: '--' ~['\n']\* '\n' -> skip ;

WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines

/\*

\*/

package littlecompiler;

import littlecompiler.GeneratedGrammarFiles.LittleBaseListener;

import littlecompiler.GeneratedGrammarFiles.LittleLexer;

import littlecompiler.GeneratedGrammarFiles.LittleParser;

import org.antlr.v4.runtime.tree.ParseTree;

import org.antlr.v4.runtime.tree.ParseTreeWalker;

import symboltables.SymbolTableVisualizer;

/\*\*

\* The entity that performs scanning / parsing / code generation.

\*/

public class Compiler

{

/\* The dependencies of the Compiler \*/

private final LittleBaseListener listener;

private final LittleLexer lexer;

private final LittleParser parser;

private ParseTree parseTree;

public Compiler(

LittleBaseListener listener,

LittleLexer lexer,

LittleParser parser)

{

this.listener = listener;

this.lexer = lexer;

this.parser = parser;

}

public void printSymbolTable()

{

new SymbolTableVisualizer().printSymbolTable(

listener.symbolTables.peek());

}

/\*\*

\* STEP 1 + 2:

\* Parsing the generated tokens into a parse tree.

\*/

public void generateAndParseTokens()

{

try

{

parseTree = parser.program();

ParseTreeWalker walker = new ParseTreeWalker();

walker.walk(

listener,

parseTree);

}

catch (Exception e)

{

System.out.println(e.getMessage());

}

}

}

/\*

\*/

package littlecompiler;

import java.io.IOException;

public class LittleCompiler

{

/\*\*

\* @param arguments

\* The command line arguments

\* @throws

\* java.io.IOException

\*/

public static void main(String[] arguments) throws IOException

{

for (String compiledFilePath : arguments)

{

CompilerFactory compilerFactory = new CompilerFactory();

Compiler littleCompiler = compilerFactory

.createCompiler(compiledFilePath);

littleCompiler.generateAndParseTokens();

}

}

}

/\*

\*/

package littlecompiler;

import java.io.IOException;

import littlecompiler.GeneratedGrammarFiles.LittleBaseListener;

import littlecompiler.GeneratedGrammarFiles.LittleLexer;

import littlecompiler.GeneratedGrammarFiles.LittleParser;

import org.antlr.v4.runtime.CharStream;

import org.antlr.v4.runtime.CharStreams;

import org.antlr.v4.runtime.CommonTokenStream;

/\*\*

\* A factory class that handles creating Compilers and injecting their

\* dependencies. The dependencies are created here to allow for unit testing

\* of the Compiler class.

\*/

public class CompilerFactory

{

/\*\*

\* @param compiledFilePath

\* The file path of the file being compiled.

\* @return

\* A Compiler to compile the provided file.

\* @throws IOException

\*/

public Compiler createCompiler(String compiledFilePath) throws IOException

{

return createCompiler(

generateCharStreamFromFilePath(

compiledFilePath));

}

/\*\*

\* @param compiledFilePath

\* The file path of the file being compiled.

\* @return

\* A CharStream of all the characters in the compiled file.

\* @throws IOException

\*/

private CharStream generateCharStreamFromFilePath(

String compiledFilePath) throws IOException

{

return CharStreams.fromFileName(compiledFilePath);

}

/\*\*

\* @param compiledFileCharStream

\* A CharStream of all the characters in the compiled file.

\* @return

\* A Compiler for compiling the provided CharStream.

\*/

private Compiler createCompiler(CharStream compiledFileCharStream)

{

LittleBaseListener littleListener = new LittleBaseListener();

LittleLexer littleLexer =

new LittleLexer(compiledFileCharStream);

LittleParser littleParser = new LittleParser(

new CommonTokenStream(littleLexer));

littleParser.setErrorHandler(new LittleErrorStrategy());

return new Compiler(

littleListener,

littleLexer,

littleParser);

}

}

/\*

\*/

package littlecompiler;

import java.util.List;

import littlecompiler.GeneratedGrammarFiles.LittleLexer;

import org.antlr.v4.runtime.Token;

/\*\*

\* A utility class that prints the generated tokens in the format:

\* Token Type: {type}

\* Value: {value}

\* Token Type: {type}

\* Value: {value}

\*/

public class TokenVisualizer

{

/\* A List of generated Tokens \*/

private final List<Token> tokens;

public TokenVisualizer(List<Token> tokens)

{

this.tokens = tokens;

}

/\*\*

\* Generates a String which displays info on all the generate tokens.

\* @return

\* A String with info on all the generated tokens in your compiled program.

\*/

public String getTokenInfoString()

{

StringBuilder stringBuilder = new StringBuilder();

tokens.forEach(token ->

{

String nextTokenInfo = generateTokenDataString(token);

stringBuilder.append(nextTokenInfo);

});

return stringBuilder.toString();

}

/\*\*

\* Generates a String which describes a token.

\* </p>

\* @param token

\* The token whose info is contained in this String.

\* @return

\* A String of the format:

\* Token Type: {type}

\* Value: {value}

\*/

private String generateTokenDataString(Token token)

{

int tokenTypeID = token.getType();

String tokenType = LittleLexer.ruleNames[tokenTypeID - 1];

return

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

}

}

/\*

\*/

package littlecompiler;

import org.antlr.v4.runtime.DefaultErrorStrategy;

import org.antlr.v4.runtime.InputMismatchException;

import org.antlr.v4.runtime.Parser;

import org.antlr.v4.runtime.ParserRuleContext;

import org.antlr.v4.runtime.RecognitionException;

import org.antlr.v4.runtime.Token;

import org.antlr.v4.runtime.misc.ParseCancellationException;

/\*\*

\* @summary

\* This class defined protocols for handling errors that occur during scanning

\* and parsing.

\*/

public class LittleErrorStrategy extends DefaultErrorStrategy

{

@Override

public Token recoverInline(Parser recognizer)

throws RecognitionException

{

InputMismatchException e = new InputMismatchException(recognizer);

for (ParserRuleContext context = recognizer.getContext(); context != null; context = context.getParent()) {

context.exception = e;

}

throw new ParseCancellationException(e);

}

@Override

public void recover(

Parser recognizer,

RecognitionException e)

{

for (ParserRuleContext context = recognizer.getContext(); context != null; context = context.getParent())

{

context.exception = e;

}

throw new ParseCancellationException(e);

}

}

/\*

\*/

package littlecompiler;

import symboltables.SymbolTable;

public class SymbolTableContainer

{

private static SymbolTableContainer instance;

public SymbolTable globalSymbolTable;

public static SymbolTableContainer getInstance()

{

return instance == null

? instance = new SymbolTableContainer()

: instance;

}

private SymbolTableContainer()

{

}

public SymbolTable getGlobalTable()

{

return globalSymbolTable;

}

public void setGlobalTable(SymbolTable globalSymbolTable)

{

this.globalSymbolTable = globalSymbolTable;

}

}

/\*

\*/

package littlecompiler.GeneratedGrammarFiles;

// Generated from Little.g4 by ANTLR 4.7.1

import AbstractSyntaxTree.AST;

import AbstractSyntaxTree.Nodes.AssignNode;

import AbstractSyntaxTree.Nodes.BeginFunctionNode;

import AbstractSyntaxTree.Nodes.ConditionNode;

import AbstractSyntaxTree.Nodes.DeclarationNode;

import AbstractSyntaxTree.Nodes.ElseNode;

import AbstractSyntaxTree.Nodes.EndFunctionNode;

import AbstractSyntaxTree.Nodes.FloatLiteralNode;

import AbstractSyntaxTree.Nodes.FunctionCallNode;

import AbstractSyntaxTree.Nodes.Operators.DivideNode;

import AbstractSyntaxTree.Nodes.FunctionNode;

import AbstractSyntaxTree.Nodes.IfNode;

import AbstractSyntaxTree.Nodes.InputParameterListNode;

import AbstractSyntaxTree.Nodes.InputParameterNode;

import AbstractSyntaxTree.Nodes.IntLiteralNode;

import AbstractSyntaxTree.Nodes.Operators.EqualNode;

import AbstractSyntaxTree.Nodes.Operators.GreaterThanNode;

import AbstractSyntaxTree.Nodes.Operators.GreaterThanOrEqualToNode;

import AbstractSyntaxTree.Nodes.Operators.LessThanNode;

import AbstractSyntaxTree.Nodes.Operators.LessThanOrEqualToNode;

import AbstractSyntaxTree.Nodes.Operators.MinusNode;

import AbstractSyntaxTree.Nodes.Operators.MultiplyNode;

import AbstractSyntaxTree.Nodes.Operators.NotEqualNode;

import AbstractSyntaxTree.Nodes.Operators.PlusNode;

import AbstractSyntaxTree.Nodes.ParameterListNode;

import AbstractSyntaxTree.Nodes.ProgramNode;

import AbstractSyntaxTree.Nodes.ReadNode;

import AbstractSyntaxTree.Nodes.ReturnNode;

import AbstractSyntaxTree.Nodes.StatementListNode;

import AbstractSyntaxTree.Nodes.VariableNode;

import AbstractSyntaxTree.Nodes.WhileNode;

import AbstractSyntaxTree.Nodes.WriteNode;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TinyAssemblyGenerator;

import java.util.ArrayList;

import java.util.List;

import java.util.Stack;

import java.util.regex.Matcher;

import java.util.regex.Pattern;

import littlecompiler.SymbolTableContainer;

import org.antlr.v4.runtime.ParserRuleContext;

import org.antlr.v4.runtime.tree.ErrorNode;

import org.antlr.v4.runtime.tree.ParseTree;

import org.antlr.v4.runtime.tree.TerminalNode;

import symboltables.Symbol;

import symboltables.SymbolTable;

import symboltables.enums.ESymbolAttribute;

import symboltables.enums.ESymbolType;

/\*\*

\*/

public class LittleBaseListener implements LittleListener

{

public final Stack<SymbolTable> symbolTables = new Stack<>();

private int currentBlockCount = 1;

private AST ast;

/\*\*

\*/

@Override public void enterProgram(LittleParser.ProgramContext ctx)

{

SymbolTable globalSymbolTable = new SymbolTable("GLOBAL");

this.symbolTables.push(globalSymbolTable);

SymbolTableContainer.getInstance().setGlobalTable(globalSymbolTable);

this.ast = new AST(new ProgramNode());

}

/\*\*

\*/

@Override public void exitProgram(LittleParser.ProgramContext ctx)

{

this.ast.pop();

List<TACLine> linesOfCode = this.ast.generate3AC();

TinyAssemblyGenerator generator = new TinyAssemblyGenerator();

generator.assemble((ArrayList<TACLine>)linesOfCode).forEach(line ->

{

System.out.println(line);

});

}

/\*\*

\*/

@Override public void enterId(LittleParser.IdContext ctx)

{

}

/\*\*

\*/

@Override public void exitId(LittleParser.IdContext ctx)

{

}

/\*\*

\*/

@Override public void enterPgm\_body(LittleParser.Pgm\_bodyContext ctx)

{

}

/\*\*

\*/

@Override public void exitPgm\_body(LittleParser.Pgm\_bodyContext ctx)

{

}

/\*\*

\*/

@Override public void enterDecl(LittleParser.DeclContext ctx)

{

}

/\*\*

\*/

@Override public void exitDecl(LittleParser.DeclContext ctx)

{

}

/\*\*

\*/

@Override public void enterString\_decl(LittleParser.String\_declContext ctx)

{

this.ast.push(new DeclarationNode());

SymbolTable currentScopeTable = this.symbolTables.peek();

ParseTree stringNameNode = ctx.children.get(1);

String stringName = stringNameNode.getText();

Symbol stringSymbol = new Symbol(

stringName,

ESymbolType.VAR,

ESymbolAttribute.STRING);

ParseTree stringValueNode = ctx.children.get(3);

String stringValue = stringValueNode.getText().replace("\"", "");

stringSymbol.setValue(stringValue);

currentScopeTable.addSymbol(stringSymbol);

}

/\*\*

\*/

@Override public void exitString\_decl(LittleParser.String\_declContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterStr(LittleParser.StrContext ctx)

{

}

/\*\*

\*/

@Override public void exitStr(LittleParser.StrContext ctx)

{

}

/\*\*

\*/

@Override public void enterVar\_decl(LittleParser.Var\_declContext ctx)

{

this.ast.push(new DeclarationNode());

SymbolTable currentScope = this.symbolTables.peek();

ParseTree varTypeNode = ctx.children.get(0);

String varType = varTypeNode.getText();

ParseTree varNamesNode = ctx.children.get(1);

String[] varNames = varNamesNode.getText().split(",");

ESymbolAttribute symbolAttribute = varType

.equals(ESymbolAttribute.INT.toString())

? ESymbolAttribute.INT

: ESymbolAttribute.FLOAT;

for (String varName : varNames)

{

if (currentScope.getSymbolByName(varName) != null)

{

System.out.println("DECLARATION ERROR " + varName);

System.exit(0);

}

currentScope.addSymbol(

new Symbol(

varName,

ESymbolType.VAR,

symbolAttribute));

}

}

/\*\*

\*/

@Override public void exitVar\_decl(LittleParser.Var\_declContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterVar\_type(LittleParser.Var\_typeContext ctx)

{

}

\*/

@Override public void exitVar\_type(LittleParser.Var\_typeContext ctx)

{

}

/\*\*

\*/

@Override public void enterAny\_type(LittleParser.Any\_typeContext ctx)

{

}

/\*\*

\*/

@Override public void exitAny\_type(LittleParser.Any\_typeContext ctx)

{

}

/\*\*

\*/

@Override public void enterId\_list(LittleParser.Id\_listContext ctx)

{

ParseTree firstVariableTree = ctx.children.get(0);

String firstVariableName = firstVariableTree.getText();

this.ast.push(new VariableNode(firstVariableName));

this.ast.pop();

ParseTree followingVariablesTree = ctx.children.get(1);

int childCount = followingVariablesTree.getChildCount();

while (childCount > 1)

{

String nextVariableName = followingVariablesTree

.getChild(1)

.getText();

this.ast.push(new VariableNode(nextVariableName));

this.ast.pop();

followingVariablesTree = followingVariablesTree.getChild(2);

childCount = followingVariablesTree.getChildCount();

}

}

/\*\*

\*/

@Override public void exitId\_list(LittleParser.Id\_listContext ctx)

{

}

/\*\*

\*/

@Override public void enterId\_tail(LittleParser.Id\_tailContext ctx)

{

}

/\*\*

\*/

@Override public void exitId\_tail(LittleParser.Id\_tailContext ctx)

{

}

/\*\*

\*/

@Override public void enterParam\_decl\_list(LittleParser.Param\_decl\_listContext ctx)

{

this.ast.push(new InputParameterListNode());

}

/\*\*

\*/

@Override public void exitParam\_decl\_list(LittleParser.Param\_decl\_listContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterParam\_decl(LittleParser.Param\_declContext ctx)

{

String parameterType = ctx.children.get(0).getText();

String parameterName = ctx.children.get(1).getText();

this.ast.push(new InputParameterNode(parameterName));

SymbolTable currentScope = this.symbolTables.peek();

currentScope.addSymbol(

new Symbol(

parameterName,

ESymbolType.VAR,

ESymbolAttribute.valueOf(parameterType)));

}

/\*\*

\*/

@Override public void exitParam\_decl(LittleParser.Param\_declContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterParam\_decl\_tail(LittleParser.Param\_decl\_tailContext ctx)

{

}

/\*\*

\*/

@Override public void exitParam\_decl\_tail(LittleParser.Param\_decl\_tailContext ctx)

{

}

/\*\*

\*/

@Override public void enterFunc\_declarations(LittleParser.Func\_declarationsContext ctx)

{

}

/\*\*

\*/

@Override public void exitFunc\_declarations(LittleParser.Func\_declarationsContext ctx)

{

}

/\*\*

\*/

@Override public void enterFunc\_decl(LittleParser.Func\_declContext ctx)

{

String functionName = ctx.getChild(2).getText();

this.ast.push(new FunctionNode(functionName));

String returnTypeString = ctx.getChild(1).getText();

ESymbolAttribute returnType = ESymbolAttribute

.valueOf(returnTypeString);

SymbolTable currentScopeTable = this.symbolTables.peek();

currentScopeTable.addSymbol(

new Symbol(

functionName,

ESymbolType.PROCEDURE,

returnType));

SymbolTable childTable = currentScopeTable

.getChildTableByName(functionName);

this.symbolTables.push(childTable);

}

/\*\*

\*/

@Override public void exitFunc\_decl(LittleParser.Func\_declContext ctx)

{

this.ast.pop();

this.symbolTables.pop();

}

/\*\*

\*/

@Override public void enterFunc\_body(LittleParser.Func\_bodyContext ctx)

{

this.ast.push(new BeginFunctionNode());

this.ast.pop();

}

/\*\*

\*/

@Override public void exitFunc\_body(LittleParser.Func\_bodyContext ctx)

{

this.ast.push(new EndFunctionNode());

this.ast.pop();

}

/\*\*

\*/

@Override public void enterStmt\_list(LittleParser.Stmt\_listContext ctx)

{

this.ast.push(new StatementListNode());

}

/\*\*

\*/

@Override public void exitStmt\_list(LittleParser.Stmt\_listContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterStmt(LittleParser.StmtContext ctx)

{

}

/\*\*

\*/

@Override public void exitStmt(LittleParser.StmtContext ctx)

{

}

/\*\*

\*/

@Override public void enterBase\_stmt(LittleParser.Base\_stmtContext ctx)

{

}

/\*\*

\*/

@Override public void exitBase\_stmt(LittleParser.Base\_stmtContext ctx)

{

}

/\*\*

\*/

@Override public void enterAssign\_stmt(LittleParser.Assign\_stmtContext ctx)

{

}

/\*\*

\*/

@Override public void exitAssign\_stmt(LittleParser.Assign\_stmtContext ctx)

{

}

/\*\*

\*/

@Override public void enterAssign\_expr(LittleParser.Assign\_exprContext ctx)

{

this.ast.push(new AssignNode(this.symbolTables.peek()));

this.ast.push(new VariableNode(ctx.getChild(0).getText()));

this.ast.pop();

}

/\*\*

\*/

@Override public void exitAssign\_expr(LittleParser.Assign\_exprContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterRead\_stmt(LittleParser.Read\_stmtContext ctx)

{

this.ast.push(new ReadNode(this.symbolTables.peek()));

}

/\*\*

\*/

@Override public void exitRead\_stmt(LittleParser.Read\_stmtContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterWrite\_stmt(LittleParser.Write\_stmtContext ctx)

{

this.ast.push(new WriteNode(this.symbolTables.peek()));

}

/\*\*

\*/

@Override public void exitWrite\_stmt(LittleParser.Write\_stmtContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterReturn\_stmt(LittleParser.Return\_stmtContext ctx)

{

String containingMethodName = this.symbolTables.peek().getName();

this.ast.push(new ReturnNode(containingMethodName));

}

/\*\*

\*/

@Override public void exitReturn\_stmt(LittleParser.Return\_stmtContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterExpr(LittleParser.ExprContext ctx)

{

boolean expressionIsAnOperation = ctx.getChildCount() == 3;

if (expressionIsAnOperation)

{

String operator = ctx.children.get(1).getText();

switch (operator)

{

case "+":

this.ast.push(new PlusNode());

break;

case "-":

this.ast.push(new MinusNode());

break;

case "\*":

this.ast.push(new MultiplyNode());

break;

case "/":

this.ast.push(new DivideNode());

break;

}

}

}

/\*\*

\*/

@Override public void exitExpr(LittleParser.ExprContext ctx)

{

boolean expressionIsAnOperation = ctx.getChildCount() == 3;

if (expressionIsAnOperation)

{

this.ast.pop();

}

}

/\*\*

\*/

@Override public void enterCall\_expr(LittleParser.Call\_exprContext ctx)

{

this.ast.push(new FunctionCallNode(ctx.getText().replace("()", "")));

}

/\*\*

\*/

@Override public void exitCall\_expr(LittleParser.Call\_exprContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterExpr\_list(LittleParser.Expr\_listContext ctx)

{

this.ast.push(new ParameterListNode());

}

/\*\*

\*/

@Override public void exitExpr\_list(LittleParser.Expr\_listContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterExpr\_list\_tail(LittleParser.Expr\_list\_tailContext ctx)

{

}

/\*\*

\*/

@Override public void exitExpr\_list\_tail(LittleParser.Expr\_list\_tailContext ctx)

{

}

/\*\*

\*/

@Override public void enterPrimary(LittleParser.PrimaryContext ctx)

{

Pattern pattern = Pattern.compile("[0-9]+");

String primaryText = ctx.getText();

Matcher matcher = pattern.matcher(primaryText);

if (matcher.matches())

{

this.ast.push(new IntLiteralNode(Integer.parseInt(ctx.getText())));

return;

}

pattern = Pattern.compile("[(].\*[)]");

matcher = pattern.matcher(primaryText);

if (matcher.matches())

{

return;

}

pattern = Pattern.compile("[0-9]+[.][0-9]+");

matcher = pattern.matcher(ctx.getText());

if (matcher.matches())

this.ast.push(

new FloatLiteralNode(

Float.parseFloat(ctx.getText())));

else

this.ast.push(new VariableNode(ctx.getText()));

}

/\*\*

\*/

@Override public void exitPrimary(LittleParser.PrimaryContext ctx)

{

String primaryText = ctx.getText();

Pattern pattern = Pattern.compile("[(].\*[)]");

Matcher matcher = pattern.matcher(primaryText);

if (matcher.matches())

{

return;

}

this.ast.pop();

}

/\*\*

\*/

@Override public void enterAddop(LittleParser.AddopContext ctx)

{

}

/\*\*

\*/

@Override public void exitAddop(LittleParser.AddopContext ctx)

{

}

/\*\*

\*/

@Override public void enterMulop(LittleParser.MulopContext ctx)

{

}

/\*\*

\*/

@Override public void exitMulop(LittleParser.MulopContext ctx)

{

}

/\*\*

\*/

@Override public void enterIf\_stmt(LittleParser.If\_stmtContext ctx)

{

this.ast.push(new IfNode());

String blockScopeName = "BLOCK " + currentBlockCount++;

SymbolTable currentScope = this.symbolTables.peek();

SymbolTable newScope = new SymbolTable(blockScopeName);

currentScope.addChildTable(newScope);

this.symbolTables.push(newScope);

}

/\*\*

\*/

@Override public void exitIf\_stmt(LittleParser.If\_stmtContext ctx)

{

this.ast.pop();

this.symbolTables.pop();

}

/\*\*

\*/

@Override public void enterElse\_part(LittleParser.Else\_partContext ctx)

{

this.ast.push(new ElseNode());

String blockScopeName = "BLOCK " + currentBlockCount++;

SymbolTable currentScope = this.symbolTables.peek();

SymbolTable newScope = new SymbolTable(blockScopeName);

currentScope.addChildTable(newScope);

this.symbolTables.push(newScope);

}

/\*\*

\*/

@Override public void exitElse\_part(LittleParser.Else\_partContext ctx)

{

this.ast.pop();

this.symbolTables.pop();

}

/\*\*

\*/

@Override public void enterCond(LittleParser.CondContext ctx)

{

this.ast.push(new ConditionNode());

String operator = ctx.children.get(1).getText();

switch (operator)

{

case ">":

this.ast.push(new GreaterThanNode(this.symbolTables.peek()));

break;

case ">=":

this.ast.push(

new GreaterThanOrEqualToNode(this.symbolTables.peek()));

break;

case "<":

this.ast.push(new LessThanNode(this.symbolTables.peek()));

break;

case "<=":

this.ast.push(

new LessThanOrEqualToNode(this.symbolTables.peek()));

break;

case "=":

this.ast.push(new EqualNode(this.symbolTables.peek()));

break;

case "!=":

this.ast.push(new NotEqualNode(this.symbolTables.peek()));

break;

default:

break;

}

}

/\*\*

\*/

@Override public void exitCond(LittleParser.CondContext ctx)

{

this.ast.pop();

this.ast.pop();

}

/\*\*

\*/

@Override public void enterCompop(LittleParser.CompopContext ctx)

{

}

/\*\*

\*/

@Override public void exitCompop(LittleParser.CompopContext ctx)

{

}

/\*\*

\*/

@Override public void enterWhile\_stmt(LittleParser.While\_stmtContext ctx)

{

this.ast.push(new WhileNode());

String blockScopeName = "BLOCK " + currentBlockCount++;

SymbolTable currentScope = this.symbolTables.peek();

SymbolTable newScope = new SymbolTable(blockScopeName);

currentScope.addChildTable(newScope);

this.symbolTables.push(newScope);

}

/\*\*

\*/

@Override public void exitWhile\_stmt(LittleParser.While\_stmtContext ctx)

{

this.ast.pop();

}

/\*\*

\*/

@Override public void enterEveryRule(ParserRuleContext ctx)

{

}

/\*\*

\*/

@Override public void exitEveryRule(ParserRuleContext ctx)

{

}

/\*\*

\*/

@Override public void visitTerminal(TerminalNode node)

{

}

/\*\*

\*/

@Override public void visitErrorNode(ErrorNode node)

{

}

@Override

public void enterAdd\_minus\_expression(LittleParser.Add\_minus\_expressionContext ctx)

{

boolean nodeIsAddop = ctx.getChildCount() == 3;

if (nodeIsAddop)

{

String operator = ctx.children.get(1).getText();

this.ast.push(

operator.equals("+")

? new PlusNode()

: new MinusNode());

}

}

@Override

public void exitAdd\_minus\_expression(LittleParser.Add\_minus\_expressionContext ctx)

{

boolean nodeIsAddop = ctx.getChildCount() == 3;

if (nodeIsAddop)

this.ast.pop();

}

@Override

public void enterMultiply\_divide\_expression(LittleParser.Multiply\_divide\_expressionContext ctx)

{

boolean nodeIsMulop = ctx.getChildCount() == 3;

if (nodeIsMulop)

{

String operator = ctx.children.get(1).getText();

this.ast.push(

operator.equals("\*")

? new MultiplyNode()

: new DivideNode());

}

}

@Override

public void exitMultiply\_divide\_expression(LittleParser.Multiply\_divide\_expressionContext ctx)

{

boolean nodeIsMulop = ctx.getChildCount() == 3;

if (nodeIsMulop)

this.ast.pop();

}

@Override

public void enterExpression\_component(LittleParser.Expression\_componentContext ctx)

{

}

@Override

public void exitExpression\_component(LittleParser.Expression\_componentContext ctx)

{

}

@Override

public void enterStmt\_list\_tail(LittleParser.Stmt\_list\_tailContext ctx)

{

}

@Override

public void exitStmt\_list\_tail(LittleParser.Stmt\_list\_tailContext ctx)

{

}

}

/\*

\*/

package AbstractSyntaxTree;

import AbstractSyntaxTree.Nodes.ASTNode;

import java.util.List;

import java.util.Stack;

import symboltables.enums.ESymbolAttribute;

public class AST

{

private final ASTNode root;

private final Stack<ASTNode> currentAstBranch;

public AST(ASTNode root)

{

this.root = root;

this.currentAstBranch = new Stack<>();

this.currentAstBranch.push(root);

}

public ASTNode getCurrentNode()

{

return this.currentAstBranch.peek();

}

public void push(ASTNode newCurrentNode)

{

this.currentAstBranch.peek().addChild(newCurrentNode);

this.currentAstBranch.push(newCurrentNode);

}

public ASTNode pop()

{

return this.currentAstBranch.pop();

}

public List<TACLine> generate3AC()

{

return root.generate3AC();

}

}

/\*

\*/

package AbstractSyntaxTree;

import java.util.ArrayList;

public class TACLine

{

private final ArrayList<String> elements;

public TACLine()

{

this.elements = new ArrayList<>();

}

public ArrayList<String> getElements()

{

return elements;

}

public void addElement(String element)

{

this.elements.add(element);

}

public String getLineText()

{

String lineText = "";

for (String element : elements)

{

lineText = lineText.concat(element.concat(" "));

}

return lineText;

}

}

/\*

\*/

package AbstractSyntaxTree;

import java.util.HashMap;

import java.util.List;

public class FunctionCodeContainer

{

private static FunctionCodeContainer instance;

private HashMap<String, List<TACLine>> functionCodes;

public static FunctionCodeContainer getInstance()

{

return instance == null

? instance = new FunctionCodeContainer()

: instance;

}

private FunctionCodeContainer()

{

this.functionCodes = new HashMap<>();

}

public void addFunctionCode(

String functionName,

List<TACLine> linesOfCode)

{

this.functionCodes.put(

functionName,

linesOfCode);

}

public List<TACLine> getFunctionCode(String functionName)

{

return this.functionCodes.get(functionName);

}

}

/\*

\*/

package AbstractSyntaxTree;

/\*\*

\* Singleton for keeping track of labels for If and While nodes.

\*/

public class Labels {

private static Labels instance;

private int a = 0;

static {

instance = new Labels();

}

public String getLabel() {

a++;

return "label".concat(Integer.toString(a));

}

public static Labels getInstance() {

return instance;

}

}

/\*

\*/

package AbstractSyntaxTree;

public class ParameterRegisterHandler

{

private static final int RETURN\_VALUE\_REGISTER\_INDEX = 900;

private static final int FIRST\_PARAMETER\_REGISTER\_INDEX = 901;

private static ParameterRegisterHandler instance;

private int nextIndex;

public static ParameterRegisterHandler getInstance()

{

return instance == null

? instance = new ParameterRegisterHandler()

: instance;

}

private ParameterRegisterHandler()

{

this.nextIndex = FIRST\_PARAMETER\_REGISTER\_INDEX;

}

public String getNextRegister()

{

return String.format(

"$T%d",

nextIndex++);

}

public void resetParameterIndex()

{

this.nextIndex = FIRST\_PARAMETER\_REGISTER\_INDEX;

}

public String getReturnRegister()

{

return "$T" + RETURN\_VALUE\_REGISTER\_INDEX;

}

}

/\*

\*/

package AbstractSyntaxTree;

import java.util.LinkedHashMap;

/\*

\* Singleton for keeping track of temporary register numbers

\*/

public class TempararyRegisters {

private static TempararyRegisters instance;

private int a = 0;

private LinkedHashMap<String, String> lHM = new LinkedHashMap<>();

static {

instance = new TempararyRegisters();

}

public String getTempReg(String key) {

a++;

String value = "$T".concat(Integer.toString(a));

lHM.put(key, value);

return value;

}

public String getTempReg() {

a++;

String value = "$T".concat(Integer.toString(a));

return value;

}

public String checkTempReg(String varName) {

if (lHM.containsKey(varName)) {

return lHM.get(varName);

}

return getTempReg(varName);

}

public static TempararyRegisters getInstance() { return instance; }

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;

import java.util.List;

import symboltables.SymbolTable;

import symboltables.enums.ESymbolAttribute;

public class AssignNode extends ASTNode

{

protected final static int VARIABLE\_INDEX = 0;

protected final static int EXPRESSION\_INDEX = 1;

private final SymbolTable scopeTable;

public AssignNode(SymbolTable table)

{

this.scopeTable = table;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeAssignTAC = new ArrayList<>();

TACLine tac1 = new TACLine();

TACLine tac2 = new TACLine();

ASTNode rhs = this.children.get(EXPRESSION\_INDEX);

String variableName = ((VariableNode)this.children.get(VARIABLE\_INDEX))

.getVariableName();

if (rhs instanceof IntLiteralNode)

{

String tempRegister = TempararyRegisters

.getInstance()

.checkTempReg(variableName);

Integer value = ((IntLiteralNode) rhs).getLiteralValue();

tac1.addElement("STOREI");

tac1.addElement(value.toString());

tac1.addElement(tempRegister);

tac2.addElement("STOREI");

tac2.addElement(tempRegister);

tac2.addElement(variableName);

completeAssignTAC.add(tac1);

completeAssignTAC.add(tac2);

}

else if (rhs instanceof FloatLiteralNode)

{

String tempRegister = TempararyRegisters

.getInstance()

.checkTempReg(variableName);

Float value = ((FloatLiteralNode) rhs).getLiteralValue();

tac1.addElement("STOREF");

tac1.addElement(value.toString());

tac1.addElement(tempRegister);

tac2.addElement("STOREF");

tac2.addElement(tempRegister);

tac2.addElement(variableName);

completeAssignTAC.add(tac1);

completeAssignTAC.add(tac2);

}

else if (rhs instanceof VariableNode)

{

VariableNode node = (VariableNode) rhs;

String variable = node.getVariableName();

ESymbolAttribute type = this.scopeTable

.getSymbolByName(variable)

.getAttribute();

String variableType = type == ESymbolAttribute.INT

? "I"

: "F";

TACLine tac = new TACLine();

tac.addElement("STORE".concat(variableType));

tac.addElement(variable);

tac.addElement(variableName);

completeAssignTAC.add(tac);

}

else

{

List<TACLine> expressionCode = rhs.generate3AC();

ESymbolAttribute expressionType = this.getChildResultType(

expressionCode);

String expressionResultRegister = this.getChildResultRegister(

expressionCode);

String storeType = expressionType == ESymbolAttribute.INT

? "STOREI"

: "STOREF";

tac2.addElement(storeType);

tac2.addElement(expressionResultRegister);

tac2.addElement(variableName);

completeAssignTAC.addAll(expressionCode);

completeAssignTAC.add(tac2);

}

return completeAssignTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

import symboltables.enums.ESymbolAttribute;

public abstract class ASTNode

{

protected List<ASTNode> children;

public ASTNode()

{

this.children = new ArrayList<>();

}

public void addChild(ASTNode childNode)

{

this.children.add(childNode);

}

protected String getChildResultRegister(List<TACLine> childCode)

{

List<String> lastLineElements = childCode

.get(childCode.size() - 1)

.getElements();

return lastLineElements.get(lastLineElements.size() - 1);

}

protected ESymbolAttribute getChildResultType(List<TACLine> childCode)

{

String finalStatement = childCode

.get(childCode.size() - 1).getElements().get(0);

return finalStatement.charAt(finalStatement.length() - 1) == 'I'

? ESymbolAttribute.INT

: ESymbolAttribute.FLOAT;

}

public abstract List<TACLine> generate3AC();

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class BeginFunctionNode extends ASTNode

{

@Override

public List<TACLine> generate3AC()

{

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

TACLine linkLine = new TACLine();

linkLine.addElement("LINK");

code.add(linkLine);

return code;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.List;

public class ConditionNode extends ASTNode

{

private static final int BOOLEAN\_OPERATOR\_INDEX = 0;

@Override

public List<TACLine> generate3AC()

{

return this.children.get(BOOLEAN\_OPERATOR\_INDEX).generate3AC();

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class DeclarationNode extends ASTNode

{

protected final static int INT\_VALUE\_INDEX = 0;

protected final static int VAR\_VALUE\_INDEX = 1;

// This shouldn't require code generation (I don't think).

@Override

public List<TACLine> generate3AC()

{

return new ArrayList<>();

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.List;

public class ElseNode extends ASTNode

{

//protected final static int CONDITION\_INDEX = 0;

protected final static int STATEMENT\_LIST\_INDEX = 0;

@Override

public List<TACLine> generate3AC()

{

return this.children.get(STATEMENT\_LIST\_INDEX).generate3AC();

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class EndFunctionNode extends ASTNode

{

@Override

public List<TACLine> generate3AC()

{

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

TACLine linkLine = new TACLine();

linkLine.addElement("RET");

code.add(linkLine);

return code;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.List;

public class FloatLiteralNode extends ASTNode

{

private final float literalValue;

public FloatLiteralNode(float value)

{

this.literalValue = value;

}

public float getLiteralValue()

{

return literalValue;

}

@Override

public List<TACLine> generate3AC()

{

throw new UnsupportedOperationException("Not supported yet.");

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class FunctionCallNode extends ASTNode

{

private final String functionName;

public FunctionCallNode(String functionName)

{

this.functionName = functionName;

}

public String getFunctionName()

{

return this.functionName;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> linesEndingWithFunctionResultStore = new ArrayList<>();

this.children.forEach(child ->

linesEndingWithFunctionResultStore

.addAll(child.generate3AC()));

return linesEndingWithFunctionResultStore;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.FunctionCodeContainer;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class FunctionNode extends ASTNode

{

private static final int INPUT\_NODE\_INDEX = 0;

private static final int STATEMENT\_LIST\_INDEX = 2;

private final String functionName;

public FunctionNode(String functionName)

{

this.functionName = functionName;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> methodLines = new ArrayList<>();

TACLine methodLabel = new TACLine();

methodLabel.addElement("LABEL");

methodLabel.addElement(this.functionName);

methodLines.add(methodLabel);

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

return methodLines;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.Labels;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class IfNode extends ASTNode

{

protected final static int CONDITION\_INDEX = 0;

protected final static int STATEMENT\_LIST\_INDEX = 1;

protected final static int ELSE\_NODE\_INDEX = 2;

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeIfTAC = new ArrayList<>();

/\* Add jump line \*/

Labels label = Labels.getInstance();

String elseLabel = label.getLabel();

List<TACLine> conditionCode = this.children

.get(CONDITION\_INDEX)

.generate3AC();

conditionCode.get(conditionCode.size() - 1).addElement(elseLabel);

completeIfTAC.addAll(conditionCode);

/\* Add statement list \*/

ASTNode statementListNode = this.children.get(STATEMENT\_LIST\_INDEX);

completeIfTAC.addAll(statementListNode.generate3AC());

/\* Add jump-out to end of statement list \*/

TACLine jumpOutLine = new TACLine();

jumpOutLine.addElement("JUMP");

String outLabel = label.getLabel();

jumpOutLine.addElement(outLabel);

completeIfTAC.add(jumpOutLine);

/\* Add else block \*/

TACLine elseLabelLine = new TACLine();

elseLabelLine.addElement("LABEL");

elseLabelLine.addElement(elseLabel);

completeIfTAC.add(elseLabelLine);

if (this.children.size() > 2)

{

ASTNode elseNode = this.children.get(ELSE\_NODE\_INDEX);

completeIfTAC.addAll(elseNode.generate3AC());

}

/\* Add jump-out label \*/

TACLine jumpOutLabel = new TACLine();

jumpOutLabel.addElement("LABEL");

jumpOutLabel.addElement(outLabel);

completeIfTAC.add(jumpOutLabel);

return completeIfTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class InputParameterListNode extends ASTNode

{

@Override

public List<TACLine> generate3AC()

{

List<TACLine> parameterLoadingCode = new ArrayList<>();

this.children.forEach(node ->

parameterLoadingCode

.addAll(node.generate3AC()));

return parameterLoadingCode;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.ParameterRegisterHandler;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class InputParameterNode extends ASTNode

{

private final String parameterName;

public InputParameterNode(String parameterName)

{

this.parameterName = parameterName;

}

public String getParameterName()

{

return parameterName;

}

@Override

public List<TACLine> generate3AC()

{

// Use singleton class to load in variables.

List<TACLine> variableLoadingCode = new ArrayList<>();

TACLine newLine = new TACLine();

newLine.addElement("LOAD");

newLine.addElement(

ParameterRegisterHandler.getInstance().getNextRegister());

newLine.addElement(this.parameterName);

variableLoadingCode.add(newLine);

return variableLoadingCode;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.List;

public class IntLiteralNode extends ASTNode

{

private final int literalValue;

public IntLiteralNode(int literalValue)

{

this.literalValue = literalValue;

}

public int getLiteralValue()

{

return literalValue;

}

@Override

public List<TACLine> generate3AC()

{

throw new UnsupportedOperationException("Not supported yet.");

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.ParameterRegisterHandler;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class ParameterListNode extends ASTNode

{

@Override

public List<TACLine> generate3AC()

{

List<TACLine> parameterStoringCode = new ArrayList<>();

this.children.forEach(child ->

{

TACLine line = new TACLine();

parameterStoringCode.add(line);

line.addElement("STORE");

if (child instanceof IntLiteralNode)

{

Integer value = ((IntLiteralNode) child).getLiteralValue();

line.addElement(value.toString());

}

else if (child instanceof FloatLiteralNode)

{

Float value = ((FloatLiteralNode) child).getLiteralValue();

line.addElement(value.toString());

}

line.addElement(

ParameterRegisterHandler

.getInstance()

.getNextRegister());

});

ParameterRegisterHandler.getInstance().resetParameterIndex();

return parameterStoringCode;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class ProgramNode extends ASTNode

{

@Override

public List<TACLine> generate3AC()

{

List<TACLine> programCode = new ArrayList<>();

for (ASTNode child : children)

{

programCode.addAll(child.generate3AC());

}

return programCode;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

import symboltables.SymbolTable;

import symboltables.enums.ESymbolAttribute;

public class ReadNode extends ASTNode

{

protected final static int READ\_INDEX = 0;

private final SymbolTable scopeTable;

public ReadNode(SymbolTable table)

{

this.scopeTable = table;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeReadTAC = new ArrayList<>();

TACLine tac = new TACLine();

ASTNode read = this.children.get(READ\_INDEX);

String variableName = ((VariableNode) read).getVariableName();

ESymbolAttribute type = this.scopeTable

.getSymbolByName(variableName)

.getAttribute();

String variableType = type == ESymbolAttribute.INT

? "I"

: "F";

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

tac.addElement(variableName);

completeReadTAC.add(tac);

return completeReadTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.ParameterRegisterHandler;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

import littlecompiler.SymbolTableContainer;

import symboltables.enums.ESymbolAttribute;

public class ReturnNode extends ASTNode

{

protected final static int RETURN\_EXPRESSION\_INDEX = 0;

private final String methodName;

public ReturnNode(String methodName)

{

this.methodName = methodName;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> lines = new ArrayList<>();

TACLine returnRegisterStoreLine = new TACLine();

lines.add(returnRegisterStoreLine);

ASTNode node = this.children.get(RETURN\_EXPRESSION\_INDEX);

if (node instanceof VariableNode)

{

}

else

{

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

ESymbolAttribute type = this.getChildResultType(tac);

if (type == ESymbolAttribute.INT)

returnRegisterStoreLine.addElement("STOREI");

else

returnRegisterStoreLine.addElement("STOREF");

String registerToReturn = this.getChildResultRegister(tac);

returnRegisterStoreLine.addElement(registerToReturn);

}

returnRegisterStoreLine

.addElement(

ParameterRegisterHandler

.getInstance()

.getReturnRegister());

return lines;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class StatementListNode extends ASTNode

{

@Override

public List<TACLine> generate3AC()

{

List<TACLine> statements = new ArrayList<>();

children.forEach(child ->

{

statements.addAll(child.generate3AC());

});

return statements;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.List;

public class StringLiteralNode extends ASTNode

{

private final String literalValue;

public StringLiteralNode(String literalValue)

{

this.literalValue = literalValue;

}

public String getLiteralValue()

{

return literalValue;

}

@Override

public List<TACLine> generate3AC()

{

throw new UnsupportedOperationException("Not supported yet.");

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.List;

public class VariableNode extends ASTNode

{

private final String variableName;

public VariableNode(String variableName)

{

this.variableName = variableName;

}

public String getVariableName()

{

return this.variableName;

}

@Override

public List<TACLine> generate3AC()

{

throw new UnsupportedOperationException("Not supported yet.");

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.Labels;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

public class WhileNode extends ASTNode

{

protected final static int CONDITION\_INDEX = 0;

protected final static int STATEMENT\_LIST\_INDEX = 1;

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeWhileTAC = new ArrayList<>();

Labels label = Labels.getInstance();

String whileLabel = label.getLabel();

String exitLabel = label.getLabel();

TACLine whileLabelLine = new TACLine();

whileLabelLine.addElement("LABEL");

whileLabelLine.addElement(whileLabel);

completeWhileTAC.add(whileLabelLine);

List<TACLine> conditionCode = this.children

.get(CONDITION\_INDEX)

.generate3AC();

conditionCode.get(conditionCode.size() - 1).addElement(exitLabel);

completeWhileTAC.addAll(conditionCode);

/\* Statement list \*/

completeWhileTAC.addAll(

this.children

.get(STATEMENT\_LIST\_INDEX)

.generate3AC());

TACLine jumpUpLine = new TACLine();

jumpUpLine.addElement("JUMP");

jumpUpLine.addElement(whileLabel);

completeWhileTAC.add(jumpUpLine);

TACLine jumpOutLabel = new TACLine();

jumpOutLabel.addElement("LABEL");

jumpOutLabel.addElement(exitLabel);

completeWhileTAC.add(jumpOutLabel);

return completeWhileTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes;

import AbstractSyntaxTree.TACLine;

import java.util.ArrayList;

import java.util.List;

import symboltables.SymbolTable;

import symboltables.enums.ESymbolAttribute;

public class WriteNode extends ASTNode

{

private final SymbolTable scopeTable;

public WriteNode(SymbolTable table)

{

this.scopeTable = table;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeWriteTAC = new ArrayList<>();

this.children.forEach(child ->

{

VariableNode node = (VariableNode) child;

String variableName = node.getVariableName();

ESymbolAttribute type = this.scopeTable

.getSymbolByName(variableName)

.getAttribute();

String variableType = type == ESymbolAttribute.INT

? "I"

: "F";

TACLine tac = new TACLine();

tac.addElement("WRITE".concat(variableType));

tac.addElement(variableName);

completeWriteTAC.add(tac);

});

return completeWriteTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;

import AbstractSyntaxTree.Nodes.FloatLiteralNode;

import AbstractSyntaxTree.Nodes.IntLiteralNode;

import static AbstractSyntaxTree.Nodes.Operators.PlusNode.LEFT\_OPERAND\_INDEX;

import AbstractSyntaxTree.Nodes.VariableNode;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;

import java.util.List;

public class DivideNode extends ASTNode

{

protected final static int LEFT\_OPERAND\_INDEX = 0;

protected final static int RIGHT\_OPERAND\_INDEX = 1;

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeAddTAC = new ArrayList<>();

TACLine tac = new TACLine();

ASTNode left = this.children.get(LEFT\_OPERAND\_INDEX);

ASTNode right = this.children.get(RIGHT\_OPERAND\_INDEX);

if (left instanceof IntLiteralNode || right instanceof IntLiteralNode){

tac.addElement("DIVI");

}

else {

tac.addElement("DIVF");

}

String leftValue = null;

if (left instanceof IntLiteralNode)

{

leftValue = String.valueOf(

((FloatLiteralNode) left).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREI");

storeRightValue.addElement(leftValue);

leftValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(leftValue);

completeAddTAC.add(storeRightValue);

}

else if (left instanceof FloatLiteralNode)

{

leftValue = String.valueOf(

((FloatLiteralNode) left).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREF");

storeRightValue.addElement(leftValue);

leftValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(leftValue);

completeAddTAC.add(storeRightValue);

}

else if (left instanceof VariableNode)

{

leftValue = ((VariableNode) left).getVariableName();

}

else

{

List<TACLine> leftExpressionCode = left.generate3AC();

leftValue = this.getChildResultRegister(leftExpressionCode);

completeAddTAC.addAll(leftExpressionCode);

}

String rightValue = null;

if (right instanceof IntLiteralNode)

{

rightValue = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREI");

storeRightValue.addElement(rightValue);

rightValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(rightValue);

completeAddTAC.add(storeRightValue);

}

else if (right instanceof FloatLiteralNode)

{

rightValue = String.valueOf(

((FloatLiteralNode) right).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREF");

storeRightValue.addElement(rightValue);

rightValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(rightValue);

completeAddTAC.add(storeRightValue);

}

else if (right instanceof VariableNode)

{

rightValue = ((VariableNode) right).getVariableName();

}

else

{

List<TACLine> rightExpressionCode = right.generate3AC();

rightValue = this.getChildResultRegister(rightExpressionCode);

completeAddTAC.addAll(rightExpressionCode);

}

tac.addElement(leftValue);

tac.addElement(rightValue);

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

completeAddTAC.add(tac);

return completeAddTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;

import AbstractSyntaxTree.Nodes.FloatLiteralNode;

import AbstractSyntaxTree.Nodes.IntLiteralNode;

import AbstractSyntaxTree.Nodes.VariableNode;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;

import java.util.List;

import symboltables.SymbolTable;

import symboltables.enums.ESymbolAttribute;

public class EqualNode extends ASTNode

{

protected final static int LEFT\_OPERAND\_INDEX = 0;

protected final static int RIGHT\_OPERAND\_INDEX = 1;

private final SymbolTable scopeTable;

public EqualNode(SymbolTable table)

{

this.scopeTable = table;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeAddTAC = new ArrayList<>();

ASTNode left = this.children.get(LEFT\_OPERAND\_INDEX);

ASTNode right = this.children.get(RIGHT\_OPERAND\_INDEX);

String variableType = "F";

String leftValue = null;

if (left instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof FloatLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof VariableNode)

{

leftValue = ((VariableNode) left).getVariableName();

}

else

{

List<TACLine> leftExpressionCode = left.generate3AC();

leftValue = this.getChildResultRegister(leftExpressionCode);

completeAddTAC.addAll(leftExpressionCode);

}

String rightValue = null;

if (right instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

variableType = "I";

}

else if (right instanceof FloatLiteralNode)

{

String rightLiteral = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(rightLiteral);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

}

else if (right instanceof VariableNode)

{

rightValue = ((VariableNode) right).getVariableName();

ESymbolAttribute type = this.scopeTable

.getSymbolByName(rightValue)

.getAttribute();

variableType = type == ESymbolAttribute.INT

? "I"

: "F";

}

else

{

List<TACLine> rightExpressionCode = right.generate3AC();

rightValue = this.getChildResultRegister(rightExpressionCode);

completeAddTAC.addAll(rightExpressionCode);

ESymbolAttribute expressionType = this.getChildResultType(

rightExpressionCode);

variableType = expressionType == ESymbolAttribute.INT

? "I"

: "F";

}

TACLine jumpLine = new TACLine();

jumpLine.addElement("NE".concat(variableType));

jumpLine.addElement(leftValue);

jumpLine.addElement(rightValue);

completeAddTAC.add(jumpLine);

/\* Add label in caller \*/

return completeAddTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;

import AbstractSyntaxTree.Nodes.FloatLiteralNode;

import AbstractSyntaxTree.Nodes.IntLiteralNode;

import AbstractSyntaxTree.Nodes.VariableNode;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;

import java.util.List;

import symboltables.SymbolTable;

import symboltables.enums.ESymbolAttribute;

public class GreaterThanNode extends ASTNode

{

protected final static int LEFT\_OPERAND\_INDEX = 0;

protected final static int RIGHT\_OPERAND\_INDEX = 1;

private final SymbolTable scopeTable;

public GreaterThanNode(SymbolTable table)

{

this.scopeTable = table;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeAddTAC = new ArrayList<>();

ASTNode left = this.children.get(LEFT\_OPERAND\_INDEX);

ASTNode right = this.children.get(RIGHT\_OPERAND\_INDEX);

String variableType = "F";

String leftValue = null;

if (left instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof FloatLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof VariableNode)

{

leftValue = ((VariableNode) left).getVariableName();

}

else

{

List<TACLine> leftExpressionCode = left.generate3AC();

leftValue = this.getChildResultRegister(leftExpressionCode);

completeAddTAC.addAll(leftExpressionCode);

}

String rightValue = null;

if (right instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

variableType = "I";

}

else if (right instanceof FloatLiteralNode)

{

String rightLiteral = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(rightLiteral);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

}

else if (right instanceof VariableNode)

{

rightValue = ((VariableNode) right).getVariableName();

ESymbolAttribute type = this.scopeTable

.getSymbolByName(rightValue)

.getAttribute();

variableType = type == ESymbolAttribute.INT

? "I"

: "F";

}

else

{

List<TACLine> rightExpressionCode = right.generate3AC();

rightValue = this.getChildResultRegister(rightExpressionCode);

completeAddTAC.addAll(rightExpressionCode);

ESymbolAttribute expressionType = this.getChildResultType(

rightExpressionCode);

variableType = expressionType == ESymbolAttribute.INT

? "I"

: "F";

}

TACLine jumpLine = new TACLine();

jumpLine.addElement("LE".concat(variableType));

jumpLine.addElement(leftValue);

jumpLine.addElement(rightValue);

completeAddTAC.add(jumpLine);

/\* Add label in caller \*/

return completeAddTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;

import AbstractSyntaxTree.Nodes.FloatLiteralNode;

import AbstractSyntaxTree.Nodes.IntLiteralNode;

import AbstractSyntaxTree.Nodes.VariableNode;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;

import java.util.List;

import symboltables.SymbolTable;

import symboltables.enums.ESymbolAttribute;

public class GreaterThanOrEqualToNode extends ASTNode

{

protected final static int LEFT\_OPERAND\_INDEX = 0;

protected final static int RIGHT\_OPERAND\_INDEX = 1;

private final SymbolTable scopeTable;

public GreaterThanOrEqualToNode(SymbolTable table)

{

this.scopeTable = table;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeAddTAC = new ArrayList<>();

ASTNode left = this.children.get(LEFT\_OPERAND\_INDEX);

ASTNode right = this.children.get(RIGHT\_OPERAND\_INDEX);

String variableType = "F";

String leftValue = null;

if (left instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof FloatLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof VariableNode)

{

leftValue = ((VariableNode) left).getVariableName();

}

else

{

List<TACLine> leftExpressionCode = left.generate3AC();

leftValue = this.getChildResultRegister(leftExpressionCode);

completeAddTAC.addAll(leftExpressionCode);

}

String rightValue = null;

if (right instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

variableType = "I";

}

else if (right instanceof FloatLiteralNode)

{

String rightLiteral = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(rightLiteral);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

}

else if (right instanceof VariableNode)

{

rightValue = ((VariableNode) right).getVariableName();

ESymbolAttribute type = this.scopeTable

.getSymbolByName(rightValue)

.getAttribute();

variableType = type == ESymbolAttribute.INT

? "I"

: "F";

}

else

{

List<TACLine> rightExpressionCode = right.generate3AC();

rightValue = this.getChildResultRegister(rightExpressionCode);

completeAddTAC.addAll(rightExpressionCode);

ESymbolAttribute expressionType = this.getChildResultType(

rightExpressionCode);

variableType = expressionType == ESymbolAttribute.INT

? "I"

: "F";

}

TACLine jumpLine = new TACLine();

jumpLine.addElement("LT".concat(variableType));

jumpLine.addElement(leftValue);

jumpLine.addElement(rightValue);

completeAddTAC.add(jumpLine);

/\* Add label in caller \*/

return completeAddTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;

import AbstractSyntaxTree.Nodes.FloatLiteralNode;

import AbstractSyntaxTree.Nodes.IntLiteralNode;

import AbstractSyntaxTree.Nodes.VariableNode;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;

import java.util.List;

import symboltables.SymbolTable;

import symboltables.enums.ESymbolAttribute;

public class LessThanNode extends ASTNode

{

protected final static int LEFT\_OPERAND\_INDEX = 0;

protected final static int RIGHT\_OPERAND\_INDEX = 1;

private final SymbolTable scopeTable;

public LessThanNode(SymbolTable table)

{

this.scopeTable = table;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeAddTAC = new ArrayList<>();

ASTNode left = this.children.get(LEFT\_OPERAND\_INDEX);

ASTNode right = this.children.get(RIGHT\_OPERAND\_INDEX);

String variableType = "F";

String leftValue = null;

if (left instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof FloatLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof VariableNode)

{

leftValue = ((VariableNode) left).getVariableName();

}

else

{

List<TACLine> leftExpressionCode = left.generate3AC();

leftValue = this.getChildResultRegister(leftExpressionCode);

completeAddTAC.addAll(leftExpressionCode);

}

String rightValue = null;

if (right instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

variableType = "I";

}

else if (right instanceof FloatLiteralNode)

{

String rightLiteral = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(rightLiteral);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

}

else if (right instanceof VariableNode)

{

rightValue = ((VariableNode) right).getVariableName();

ESymbolAttribute type = this.scopeTable

.getSymbolByName(rightValue)

.getAttribute();

variableType = type == ESymbolAttribute.INT

? "I"

: "F";

}

else

{

List<TACLine> rightExpressionCode = right.generate3AC();

rightValue = this.getChildResultRegister(rightExpressionCode);

completeAddTAC.addAll(rightExpressionCode);

ESymbolAttribute expressionType = this.getChildResultType(

rightExpressionCode);

variableType = expressionType == ESymbolAttribute.INT

? "I"

: "F";

}

TACLine jumpLine = new TACLine();

jumpLine.addElement("GE".concat(variableType));

jumpLine.addElement(leftValue);

jumpLine.addElement(rightValue);

completeAddTAC.add(jumpLine);

/\* Add label in caller \*/

return completeAddTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;

import AbstractSyntaxTree.Nodes.FloatLiteralNode;

import AbstractSyntaxTree.Nodes.IntLiteralNode;

import AbstractSyntaxTree.Nodes.VariableNode;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;

import java.util.List;

import symboltables.SymbolTable;

import symboltables.enums.ESymbolAttribute;

public class LessThanOrEqualToNode extends ASTNode

{

protected final static int LEFT\_OPERAND\_INDEX = 0;

protected final static int RIGHT\_OPERAND\_INDEX = 1;

private final SymbolTable scopeTable;

public LessThanOrEqualToNode(SymbolTable table)

{

this.scopeTable = table;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeAddTAC = new ArrayList<>();

ASTNode left = this.children.get(LEFT\_OPERAND\_INDEX);

ASTNode right = this.children.get(RIGHT\_OPERAND\_INDEX);

String variableType = "F";

String leftValue = null;

if (left instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof FloatLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof VariableNode)

{

leftValue = ((VariableNode) left).getVariableName();

}

else

{

List<TACLine> leftExpressionCode = left.generate3AC();

leftValue = this.getChildResultRegister(leftExpressionCode);

completeAddTAC.addAll(leftExpressionCode);

}

String rightValue = null;

if (right instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

variableType = "I";

}

else if (right instanceof FloatLiteralNode)

{

String rightLiteral = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(rightLiteral);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

}

else if (right instanceof VariableNode)

{

rightValue = ((VariableNode) right).getVariableName();

ESymbolAttribute type = this.scopeTable

.getSymbolByName(rightValue)

.getAttribute();

variableType = type == ESymbolAttribute.INT

? "I"

: "F";

}

else

{

List<TACLine> rightExpressionCode = right.generate3AC();

rightValue = this.getChildResultRegister(rightExpressionCode);

completeAddTAC.addAll(rightExpressionCode);

ESymbolAttribute expressionType = this.getChildResultType(

rightExpressionCode);

variableType = expressionType == ESymbolAttribute.INT

? "I"

: "F";

}

TACLine jumpLine = new TACLine();

jumpLine.addElement("GT".concat(variableType));

jumpLine.addElement(leftValue);

jumpLine.addElement(rightValue);

completeAddTAC.add(jumpLine);

/\* Add label in caller \*/

return completeAddTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;

import AbstractSyntaxTree.Nodes.FloatLiteralNode;

import AbstractSyntaxTree.Nodes.IntLiteralNode;

import AbstractSyntaxTree.Nodes.VariableNode;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;

import java.util.List;

public class MinusNode extends ASTNode

{

protected final static int LEFT\_OPERAND\_INDEX = 0;

protected final static int RIGHT\_OPERAND\_INDEX = 1;

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeAddTAC = new ArrayList<>();

TACLine tac = new TACLine();

ASTNode left = this.children.get(LEFT\_OPERAND\_INDEX);

ASTNode right = this.children.get(RIGHT\_OPERAND\_INDEX);

if (left instanceof IntLiteralNode || right instanceof IntLiteralNode){

tac.addElement("SUBI");

}

else {

tac.addElement("SUBF");

}

String leftValue = null;

if (left instanceof IntLiteralNode)

{

leftValue = String.valueOf(

((FloatLiteralNode) left).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREI");

storeRightValue.addElement(leftValue);

leftValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(leftValue);

completeAddTAC.add(storeRightValue);

}

else if (left instanceof FloatLiteralNode)

{

leftValue = String.valueOf(

((FloatLiteralNode) left).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREF");

storeRightValue.addElement(leftValue);

leftValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(leftValue);

completeAddTAC.add(storeRightValue);

}

else if (left instanceof VariableNode)

{

leftValue = ((VariableNode) left).getVariableName();

}

else

{

List<TACLine> leftExpressionCode = left.generate3AC();

leftValue = this.getChildResultRegister(leftExpressionCode);

completeAddTAC.addAll(leftExpressionCode);

}

String rightValue = null;

if (right instanceof IntLiteralNode)

{

rightValue = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREI");

storeRightValue.addElement(rightValue);

rightValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(rightValue);

completeAddTAC.add(storeRightValue);

}

else if (right instanceof FloatLiteralNode)

{

rightValue = String.valueOf(

((FloatLiteralNode) right).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREF");

storeRightValue.addElement(rightValue);

rightValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(rightValue);

completeAddTAC.add(storeRightValue);

}

else if (right instanceof VariableNode)

{

rightValue = ((VariableNode) right).getVariableName();

}

else

{

List<TACLine> rightExpressionCode = right.generate3AC();

rightValue = this.getChildResultRegister(rightExpressionCode);

completeAddTAC.addAll(rightExpressionCode);

}

tac.addElement(leftValue);

tac.addElement(rightValue);

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

completeAddTAC.add(tac);

return completeAddTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;

import AbstractSyntaxTree.Nodes.FloatLiteralNode;

import AbstractSyntaxTree.Nodes.IntLiteralNode;

import static AbstractSyntaxTree.Nodes.Operators.DivideNode.LEFT\_OPERAND\_INDEX;

import AbstractSyntaxTree.Nodes.VariableNode;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;

import java.util.List;

public class MultiplyNode extends ASTNode

{

protected final static int LEFT\_OPERAND\_INDEX = 0;

protected final static int RIGHT\_OPERAND\_INDEX = 1;

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeAddTAC = new ArrayList<>();

TACLine tac = new TACLine();

ASTNode left = this.children.get(LEFT\_OPERAND\_INDEX);

ASTNode right = this.children.get(RIGHT\_OPERAND\_INDEX);

if (left instanceof IntLiteralNode || right instanceof IntLiteralNode){

tac.addElement("MULTI");

}

else {

tac.addElement("MULTF");

}

String leftValue = null;

if (left instanceof IntLiteralNode)

{

leftValue = String.valueOf(

((FloatLiteralNode) left).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREI");

storeRightValue.addElement(leftValue);

leftValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(leftValue);

completeAddTAC.add(storeRightValue);

}

else if (left instanceof FloatLiteralNode)

{

leftValue = String.valueOf(

((FloatLiteralNode) left).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREF");

storeRightValue.addElement(leftValue);

leftValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(leftValue);

completeAddTAC.add(storeRightValue);

}

else if (left instanceof VariableNode)

{

leftValue = ((VariableNode) left).getVariableName();

}

else

{

List<TACLine> leftExpressionCode = left.generate3AC();

leftValue = this.getChildResultRegister(leftExpressionCode);

completeAddTAC.addAll(leftExpressionCode);

}

String rightValue = null;

if (right instanceof IntLiteralNode)

{

rightValue = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREI");

storeRightValue.addElement(rightValue);

rightValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(rightValue);

completeAddTAC.add(storeRightValue);

}

else if (right instanceof FloatLiteralNode)

{

rightValue = String.valueOf(

((FloatLiteralNode) right).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREF");

storeRightValue.addElement(rightValue);

rightValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(rightValue);

completeAddTAC.add(storeRightValue);

}

else if (right instanceof VariableNode)

{

rightValue = ((VariableNode) right).getVariableName();

}

else

{

List<TACLine> rightExpressionCode = right.generate3AC();

rightValue = this.getChildResultRegister(rightExpressionCode);

completeAddTAC.addAll(rightExpressionCode);

}

tac.addElement(leftValue);

tac.addElement(rightValue);

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

completeAddTAC.add(tac);

return completeAddTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;

import AbstractSyntaxTree.Nodes.FloatLiteralNode;

import AbstractSyntaxTree.Nodes.IntLiteralNode;

import AbstractSyntaxTree.Nodes.VariableNode;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;

import java.util.List;

import symboltables.SymbolTable;

import symboltables.enums.ESymbolAttribute;

public class NotEqualNode extends ASTNode

{

protected final static int LEFT\_OPERAND\_INDEX = 0;

protected final static int RIGHT\_OPERAND\_INDEX = 1;

private final SymbolTable scopeTable;

public NotEqualNode(SymbolTable table)

{

this.scopeTable = table;

}

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeAddTAC = new ArrayList<>();

ASTNode left = this.children.get(LEFT\_OPERAND\_INDEX);

ASTNode right = this.children.get(RIGHT\_OPERAND\_INDEX);

String variableType = "F";

String leftValue = null;

if (left instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof FloatLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(literal);

leftValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(leftValue);

completeAddTAC.add(storeLiteralLine);

}

else if (left instanceof VariableNode)

{

leftValue = ((VariableNode) left).getVariableName();

}

else

{

List<TACLine> leftExpressionCode = left.generate3AC();

leftValue = this.getChildResultRegister(leftExpressionCode);

completeAddTAC.addAll(leftExpressionCode);

}

String rightValue = null;

if (right instanceof IntLiteralNode)

{

String literal = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREI");

storeLiteralLine.addElement(literal);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

variableType = "I";

}

else if (right instanceof FloatLiteralNode)

{

String rightLiteral = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeLiteralLine = new TACLine();

storeLiteralLine.addElement("STOREF");

storeLiteralLine.addElement(rightLiteral);

rightValue = TempararyRegisters

.getInstance()

.getTempReg();

storeLiteralLine.addElement(rightValue);

completeAddTAC.add(storeLiteralLine);

}

else if (right instanceof VariableNode)

{

rightValue = ((VariableNode) right).getVariableName();

ESymbolAttribute type = this.scopeTable

.getSymbolByName(rightValue)

.getAttribute();

variableType = type == ESymbolAttribute.INT

? "I"

: "F";

}

else

{

List<TACLine> rightExpressionCode = right.generate3AC();

rightValue = this.getChildResultRegister(rightExpressionCode);

completeAddTAC.addAll(rightExpressionCode);

ESymbolAttribute expressionType = this.getChildResultType(

rightExpressionCode);

variableType = expressionType == ESymbolAttribute.INT

? "I"

: "F";

}

TACLine jumpLine = new TACLine();

jumpLine.addElement("EQ".concat(variableType));

jumpLine.addElement(leftValue);

jumpLine.addElement(rightValue);

completeAddTAC.add(jumpLine);

/\* Add label in caller \*/

return completeAddTAC;

}

}

/\*

\*/

package AbstractSyntaxTree.Nodes.Operators;

import AbstractSyntaxTree.Nodes.ASTNode;

import AbstractSyntaxTree.Nodes.FloatLiteralNode;

import AbstractSyntaxTree.Nodes.IntLiteralNode;

import AbstractSyntaxTree.Nodes.VariableNode;

import AbstractSyntaxTree.TACLine;

import AbstractSyntaxTree.TempararyRegisters;

import java.util.ArrayList;

import java.util.List;

public class PlusNode extends ASTNode

{

protected final static int LEFT\_OPERAND\_INDEX = 0;

protected final static int RIGHT\_OPERAND\_INDEX = 1;

@Override

public List<TACLine> generate3AC()

{

List<TACLine> completeAddTAC = new ArrayList<>();

TACLine tac = new TACLine();

ASTNode left = this.children.get(LEFT\_OPERAND\_INDEX);

ASTNode right = this.children.get(RIGHT\_OPERAND\_INDEX);

if (left instanceof IntLiteralNode || right instanceof IntLiteralNode){

tac.addElement("ADDI");

}

else {

tac.addElement("ADDF");

}

String leftValue = null;

if (left instanceof IntLiteralNode)

{

leftValue = String.valueOf(

((FloatLiteralNode) left).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREI");

storeRightValue.addElement(leftValue);

leftValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(leftValue);

completeAddTAC.add(storeRightValue);

}

else if (left instanceof FloatLiteralNode)

{

leftValue = String.valueOf(

((FloatLiteralNode) left).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREF");

storeRightValue.addElement(leftValue);

leftValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(leftValue);

completeAddTAC.add(storeRightValue);

}

else if (left instanceof VariableNode)

{

leftValue = ((VariableNode) left).getVariableName();

}

else

{

List<TACLine> leftExpressionCode = left.generate3AC();

leftValue = this.getChildResultRegister(leftExpressionCode);

completeAddTAC.addAll(leftExpressionCode);

}

String rightValue = null;

if (right instanceof IntLiteralNode)

{

rightValue = String.valueOf(

((IntLiteralNode) right).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREI");

storeRightValue.addElement(rightValue);

rightValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(rightValue);

completeAddTAC.add(storeRightValue);

}

else if (right instanceof FloatLiteralNode)

{

rightValue = String.valueOf(

((FloatLiteralNode) right).getLiteralValue());

TACLine storeRightValue = new TACLine();

storeRightValue.addElement("STOREF");

storeRightValue.addElement(rightValue);

rightValue = TempararyRegisters.getInstance().getTempReg();

storeRightValue.addElement(rightValue);

completeAddTAC.add(storeRightValue);

}

else if (right instanceof VariableNode)

{

rightValue = ((VariableNode) right).getVariableName();

}

else

{

List<TACLine> rightExpressionCode = right.generate3AC();

rightValue = this.getChildResultRegister(rightExpressionCode);

completeAddTAC.addAll(rightExpressionCode);

}

tac.addElement(leftValue);

tac.addElement(rightValue);

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

completeAddTAC.add(tac);

return completeAddTAC;

}

}

/\*

\*/

package AbstractSyntaxTree;

import java.util.ArrayList;

import java.util.Hashtable;

public class TinyAssemblyGenerator {

private int registerCounter;

private Hashtable<String, String> variableTable;

private ArrayList<String> variableDeclarations;

private ArrayList<String> assemblyCode;

private boolean jumpGenerated;

private boolean newlineUsed;

private ArrayList<String> freeRegisters;

public TinyAssemblyGenerator() {

registerCounter = 0;

variableTable = new Hashtable<>();

assemblyCode = new ArrayList<>();

variableDeclarations = new ArrayList<>();

jumpGenerated = false;

newlineUsed = false;

freeRegisters = new ArrayList<>();

}

/\*\*

\* must be called before generating new Tiny code from a new Basic Block

\*/

public void reset() {

registerCounter = 0;

variableTable = new Hashtable<>();

assemblyCode = new ArrayList<>();

variableDeclarations = new ArrayList<>();

jumpGenerated = false;

newlineUsed = false;

freeRegisters = new ArrayList<>();

}

/\*\*

\* generate Tiny code from the TAC

\* @param tacList the TAC from which to generate Tiny code

\* @return the ArrayList of Strings representing the generated Tiny code

\*/

public ArrayList<String> assemble(ArrayList<TACLine> tacList) {

for (TACLine tacLine: tacList) {

ArrayList<String> newCode;

// switch based on the TAC instruction type

switch (tacLine.getElements().get(0)) {

case "STOREI":

case "STOREF":

newCode = generateSTORE(tacLine);

break;

case "MULTI":

case "MULTF":

case "ADDI":

case "ADDF":

case "SUBI":

case "SUBF":

case "DIVI":

case "DIVF":

newCode = generateAddSubMultDiv(tacLine);

break;

case "WRITEI":

case "WRITEF":

case "WRITES":

newCode = generateWRITE(tacLine);

break;

case "RET":

newCode = generateRET(tacLine);

break;

case "LABEL":

newCode = generateLABEL(tacLine);

break;

case "LEI":

case "LEF":

case "EQI":

case "EQF":

case "NEI":

case "NEF":

case "LTI":

case "LTF":

case "GTI":

case "GTF":

case "GEI":

case "GEF":

newCode = generateConditionalBranch(tacLine);

break;

case "READI":

case "READF":

newCode = generateREAD(tacLine);

break;

case "JUMP":

newCode = generateJUMP(tacLine);

break;

// this should only be entered if an invalid TAC instruction is encountered

// in the case of an invalid TAC instruction, do not produce any Tiny code

// ###

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

// ##

default:

newCode = new ArrayList<>();

}

assemblyCode.addAll(newCode);

}

// handles the way that the given test code only declares 'str newline' when the newline is actually used

if (newlineUsed) {

assemblyCode.add(0, "str newline \"\\n\"");

}

// handles the stipulation that all variable declarations must appear at the beginning of the tiny code,

// as opposed to when they are first needed

for (int i = variableDeclarations.size() - 1; i >= 0; i--) {

assemblyCode.add(0, variableDeclarations.get(i));

}

// handles the way that the given test code only generates 'label' statements if the Tiny code contains a jump

// statement

if (!jumpGenerated) {

ArrayList<String> toRemove = new ArrayList<>();

for (String s: assemblyCode) {

if (s.startsWith("label ")) {

toRemove.add(s);

}

}

// remove label statements if there isn't a jump statement in the tiny code

for (String s: toRemove) {

assemblyCode.remove(s);

}

}

return assemblyCode;

}

/\*\*

\* gets the next free register

\*

\* note: in accordance to the given test code, a register is only considered free if it has not been assigned yet,

\* or if it has only been assigned temporarily in a Store instruction in which both arguments are variables

\* (neither argument is already in a register)

\*

\* this seems like a random arbitrary rule, but I implemented it this way to match the test code

\*

\* @return the string representation of the next available register

\*/

private String getNextFreeRegister() {

if (!freeRegisters.isEmpty()) {

return freeRegisters.remove(0);

}

String r = "r" + registerCounter++;

freeRegisters.add(r);

return r;

}

/\*\*

\* @return the lowest unassigned register number

\*/

private String getNextRegister() {

return "r" + registerCounter++;

}

/\*\*

\* @param tacName the TAC variable name

\* @return the tiny variable name or register name associated with tacName

\*/

private String getVarNameFromVariableTable(String tacName) {

if (!variableTable.containsKey(tacName))

{

// assign register

if (tacName.startsWith("$")) {

variableTable.put(tacName, "r" + registerCounter++);

}

// the var name is not a register, and not 'newLine'

else if (!tacName.equals("newline"))

{

variableTable.put(tacName, tacName);

String varInitialization = "var " + variableTable.get(tacName);

variableDeclarations.add(varInitialization);

}

// the var name is specifically 'newLine'

else

{

newlineUsed = true;

return "newline";

}

}

return variableTable.get(tacName);

}

/\*\*

\* STORE I/F

\*

\* @param tacLine

\* @return

\*/

private ArrayList<String> generateSTORE(TACLine tacLine) {

ArrayList<String> code = new ArrayList<>();

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

String arg1 = tac.get(1);

String arg2 = tac.get(2);

// we need to address the special scenario in which we declare this temporary register 'free'

if (!arg1.startsWith("$") && !arg2.startsWith("$"))

{

String arg3 = getNextFreeRegister();

String line = "move " + arg1 + " " + arg3;

code.add(line);

line = "move " + arg3 + " " + arg2;

code.add(line);

return code;

}

if (!Character.isDigit(arg1.toCharArray()[0])) {

arg1 = getVarNameFromVariableTable(arg1);

}

if (!Character.isDigit(arg2.toCharArray()[0])) {

arg2 = getVarNameFromVariableTable(arg2);

}

String line = "move " + arg1 + " " + arg2;

code.add(line);

return code;

}

/\*\*

\* WRITE S/F/I

\*

\* @param tacLine

\* @return

\*/

private ArrayList<String> generateWRITE(TACLine tacLine) {

ArrayList<String> code = new ArrayList<>();

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

String arg0 = tac.get(0);

String arg1 = tac.get(1);

arg1 = getVarNameFromVariableTable(arg1);

String operator = "";

switch (arg0) {

case "WRITES":

operator = "writes ";

break;

case "WRITEF":

operator = "writer ";

break;

case "WRITEI":

operator = "writei ";

break;

}

String line = "sys " + operator + arg1;

code.add(line);

return code;

}

/\*\*

\* RET

\*

\* @param tacLine

\* @return

\*/

private ArrayList<String> generateRET(TACLine tacLine) {

ArrayList<String> code = new ArrayList<>();

String line = "sys halt";

code.add(line);

return code;

}

/\*\*

\* LABEL

\*

\* @param tacLine

\* @return

\*/

private ArrayList<String> generateLABEL(TACLine tacLine) {

ArrayList<String> code = new ArrayList<>();

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

String line = "label " + arg1;

code.add(line);

return code;

}

/\*\*

\* JUMP

\*

\* @param tacLine

\* @return

\*/

private ArrayList<String> generateJUMP(TACLine tacLine) {

ArrayList<String> code = new ArrayList<>();

jumpGenerated = true;

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

String line = "jmp " + arg1;

code.add(line);

return code;

}

/\*\*

\* Conditional Branch: LE EQ GT NE GE LT

\*

\* @param tacLine

\* @return

\*/

private ArrayList<String> generateConditionalBranch(TACLine tacLine) {

ArrayList<String> code = new ArrayList<>();

jumpGenerated = true;

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

String arg0 = tac.get(0);

String arg1 = tac.get(1);

String arg2 = tac.get(2);

String arg3 = tac.get(3);

if (!arg1.startsWith("$") && !arg2.startsWith("$"))

{

String temp = getNextRegister();

String line = "move " + arg2 + " " + temp;

arg2 = temp;

code.add(line);

arg1 = getVarNameFromVariableTable(arg1);

}

else

{

arg1 = getVarNameFromVariableTable(arg1);

arg2 = getVarNameFromVariableTable(arg2);

}

String operator = "";

switch (arg0.charAt(2)) {

case 'I':

operator = "cmpi ";

break;

case 'F':

operator = "cmpr ";

break;

}

String line = operator + arg1 + " " + arg2;

code.add(line);

String operator2 = "";

switch (arg0.substring(0,2)) {

case "LE":

operator2 = "jle ";

break;

case "EQ":

operator2 = "jeq ";

break;

case "NE":

operator2 = "jne ";

break;

case "GT":

operator2 = "jgt ";

break;

case "GE":

operator2 = "jge ";

break;

case "LT":

operator2 = "jlt ";

break;

}

String line2 = operator2 + arg3;

code.add(line2);

return code;

}

/\*\*

\* READ

\*

\* @param tacLine

\* @return

\*/

private ArrayList<String> generateREAD(TACLine tacLine) {

ArrayList<String> code = new ArrayList<>();

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

String arg0 = tac.get(0);

String arg1 = tac.get(1);

arg1 = getVarNameFromVariableTable(arg1);

String operator = "";

switch (arg0) {

case "READI":

operator = "readi ";

break;

case "READF":

operator = "readr ";

break;

case "READS":

operator = "reads ";

break;

}

String line = "sys " + operator + arg1;

code.add(line);

return code;

}

/\*\*

\* ADD/MUL/DIV/SUB

\*

\* @param tacLine

\* @return

\*/

private ArrayList<String> generateAddSubMultDiv(TACLine tacLine) {

ArrayList<String> code = new ArrayList<>();

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

String arg0 = tac.get(0);

String arg1 = tac.get(1);

String arg2 = tac.get(2);

String arg3 = tac.get(3);

arg1 = getVarNameFromVariableTable(arg1);

arg2 = getVarNameFromVariableTable(arg2);

arg3 = getVarNameFromVariableTable(arg3);

String line = "move " + arg1 + " " + arg3;

code.add(line);

String operator = "";

switch (arg0) {

case "MULTI":

operator = "muli ";

break;

case "MULTF":

operator = "mulr ";

break;

case "ADDI":

operator = "addi ";

break;

case "ADDF":

operator = "addr ";

break;

case "SUBI":

operator = "subi ";

break;

case "SUBF":

operator = "subr ";

break;

case "DIVI":

operator = "divi ";

break;

case "DIVF":

operator = "divr ";

break;

}

String line2 = operator + arg2 + " " + arg3;

code.add(line2);

return code;

}

}

**Section 2: Teamwork**

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

**Section 3: Design Pattern**

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

**Section 4: Technical Writing**

*Include the technical document that accompanied your capstone project.*

**Section 5: UML**

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

**Section 6: Design Trade-offs**

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

**Section 7: Software Development Life Cycle Model**

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