FLOYD COMPILER MANUAL

Feature Level: B

APRIL 26, 2018

CpS 450
Italo Moraes

Contents

Introduction

The purpose of this manual is to document a Floyd compiler written by Italo Moraes. The first section will cover usage, which includes information on how to build and run the compiler. Next is the features with details on the features the compiler supports and their limitations. The features are proceeded by the technical notes, the meat of the manual. The technical notes will cover how the compiler is organized, information about memory management, the runtime stack, I/O, and strings. The last section is a list of tests, their outputs, and whether they pass or fail.

Usage

To use the compiler, the project and its dependencies must first be built. The compiler can be built and executed either manually using the command line or automatically using shell scripts.

Manual Compilation

First, open a command prompt in the floyd directory. Next, run the following commands:

```
make
./gradlew clean install
```

Make will produce an object file from the C standard library. This object file will be necessary in the linking process of Floyd programs. *./gradlew clean install* will remove any old generated files and build the project.

Manual Execution

To run the compiler, run the following command from a command prompt in the floyd directory:

```
build/install/floyd/bin/floyd [-ds] [-S] [-dp] <Floyd Program >
```

The bracketed arguments are optional command line arguments that are supported by the compiler. Refer to the Supported Command Line Options section below for more information. *<Floyd Program>* refers to the path of a Floyd program. Running a Floyd program named "test.floyd" with no command line options is as simple as this:

```
build/install/floyd/bin/floyd test.floyd
```

If there are no errors in the Floyd program, this command will produce an executable. To run the executable, simply execute the following command:

```
./test
```

Compilation and Execution by Shell Scripts

Two shell scripts are provided. compileProg.sh will build the project and its dependencies, then it will run the compiler and provide it the file name listed in the script. To change the file name in the script, open it with a text editor and edit the fileName= line so that the right-hand side of the = has the desired file name. For example, if the file is named test.floyd, it would look like this:

```
fileName="test.floyd"
```

To run the script, make sure it has execution privileges, and type this into the terminal:

```
./ compileProg.sh
```

The second shell script, compileProgs.sh, builds the compiler and its dependencies, but it also compiles all Floyd programs in the current working directory. To run this script, simply use the following command:

```
./compileProgs.sh
```

Furthermore, compileProgs.sh also accepts one command line option called "clean." When run with the "clean" option, the script does not build the compiler. It cleans the floyd folder of assembly files, object files, and any files generated by the gradle build. This is how to run the script with the clean option:

```
./compileProg.sh clean
```

Note: Any other variation of command line arguments (or lack thereof) will simply build the compiler and compile all Floyd programs in the current working directory.

After either one of these scripts is executed, at least one Floyd executable will be in the current working directory. To run an executable generated from a "test.floyd" file, for example, run a command like this:

```
./test
```

Supported Command Line

The following three command line options are supported:

Option	Description
-ds	Produces a list of tokens generated by the lexer and outputs it to standard output.

-S	Stops the compiler at the code generation step. Generates an assembly file of the form fileName.S but does not produce an executable.
-dp	Displays a graphical parse tree of the program. Also displays the stack trace of syntax errors.

To run the compiler with one of these options, simply add it as a command line argument before the Floyd file. For example, to compile a program with all three command line options type this:

```
build/install/floyd/bin/floyd -S -ds -dp test.floyd
```

Features

These are the features supported by the compiler. A red asterisk marks a feature that has a limitation and a blue asterisk marks a feature with an extension. More information on these features can be found in their respective sections.

Feature
Local and instance variable declarations*
Literal, identifier, parenthesized expressions
Assignment statement
If-Then-Else statement
Call statement
Loop While statement
Method declarations with arguments and an optional return type
Method calls (with recursion support)
Multiple class declarations*
Null, me and new support for objects
Run-time null pointer checks
String support
Predefined in/out variables for input and output
Compile time semantic checks

Limitations

Multiple class declarations: Inheritance is not supported at this time.

Extensions

Instance variable declarations: Memory is dynamically allocated for instance variables.

Technical Notes

Tools

These are the necessary tools to build and run the compiler:

Name	Version	Website
ANTLR	4.7.1	http://www.antlr.org/
Gradle	2.2.1	https://gradle.org/install/
GCC	5.4.0	https://gcc.gnu.org/releases.html
(GNU) Make	4.1	https://www.gnu.org/software/make/
Java	8	http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

Compiler Organization

The compiler has three key classes. Each key class has a significant role in the compilation process. The first class is MyFloydLexer. MyFloydLexer turns the characters specified in a Floyd program into tokens that are stored in a parse tree. The parse tree is a representation of the program in tree form, which is vital for the semantic and code generation steps that will be discussed later. MyFloydLexer also performs syntactic checks. With the help of MyFloydErrorListener, error messages provide information like the file name, line number, character position.

The second key class is the SemanticChecker class. It has the task of catching and reporting several semantic errors along with their file name, line number, and character position. Furthermore, the semantic checker has the task of decorating the generated parse tree with different information that may be needed during the semantic and code generation phases. The parse tree is decorated using a listener approach. For example, when processing expressions, the type of the expression must be passed all the way up the tree for the semantic checker to perform type checks. The SemanticChecker class depends on the SymbolTable class to accurately catch semantic errors. The SymbolTable is a singleton class that contains a symbol table, which is a stack of all the declared variables, methods, and classes. The symbol table also keeps track of information like a symbol's scope and type. Without the symbol table, it would not be possible to catch semantic errors.
Here is a list of the errors the semantic checker must catch:

Semantic Errors
Use of undeclared variables
Attempting to declare an already defined variable, method, or class.
Parameter mismatch: type and number

Attempting to use an undeclared feature
Type errors

The third and final key class is the CodeGen class. This class generates assembly instructions as it walks the tree that was decorated by the semantic checker. The CodeGen class uses a visitor approach as it traverses the tree, which means there's an option to choose when to visit what nodes. Every rule defined in the ANTLR grammar must be accounted for in the code generation phase because each visited rule must generate a snippet of assembly.
The CodeGen class relies on the TargetInstruction class. The TargetInstruction class uses a Builder design pattern to facilitate in the generation of code snippets. CodeGen keeps a list of TargetInstructions; this is the whole assembly program in list form. Once CodeGen finishes traversing the tree, the entire TargetInstruction list is dumped into what is called the assembly file.

A separate, but just as important part of the compiler is the ANTLR grammar. The ANTLR grammar is stored in the Floyd.g4 file and it's a collection of regular expressions defining the Floyd grammar. The grammar is vital to the compiler because it generates the scanner that's used by MyFloydLexer.

Memory Management

Memory is dynamically allocated for non int and boolean types when variables are declared. This is done through a C function call to calloc. The size of the chunk of memory allocated is calculated by: (the number of instance variables * 4) + 8. The extra 8 bytes of memory are reserved for future feature implementations. In the future, the first four bytes of the object will contain a reference to the parent of the object's virtual function table. The next four bytes will be used for reference counting to free allocated memory no longer in use. This means that memory is not deallocated at this time. Here is what an object with two instance variables (x and y) would look like:

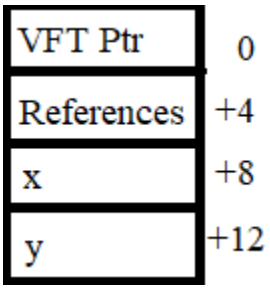

Figure 1: Object Structure

Runtime Stack

References to the stack are BP relative. There are reserved chunks of space above and below the BP, so method parameters begin at +12 and local variables begin at -8. Since it's easier to grasp the concept of a stack graphically, refer to figure 2 for an example. Figure 2 is what the stack would look like during the function call on line 7.

```
1  class Sample is
2    x: int
3    start() is
4    y : int
5    begin
6      y := 5
7      out.writeint(y)
8    end start
9  end Sample
```

Figure 2: Sample Floyd Program

Above BP there are two four-byte chunks of reserved memory. At +4 is the return address that will be used to jump back from the method call. +8 holds a reference to the caller object "out" which is a Writer type. At +12 is the parameter value that was passed into writeint, the number 5.

Below BP at -4 is a chunk of memory reserved for a return value. Finally, local variables begin at -8 relative to BP. The reason behind such odd offsets for parameters and local variables should be much clearer now.

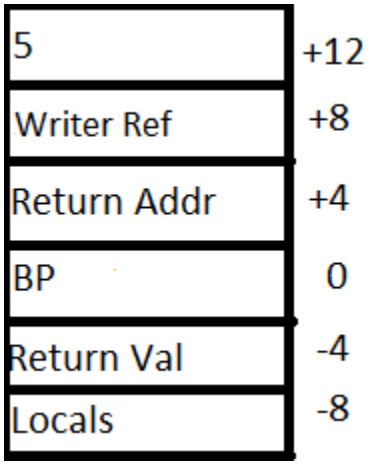

Figure 3: Sample Stack

I/O

Input and output is implemented through the Reader and Writer classes defined in the Floyd standard library. These classes make use of the read and write system calls by invoking them through the C standard library. The standard library function that calls read is Reader_io_read and the function that calls write is Writer_io_write. For example, the implementation of the write function defined in the Floyd standard library is rather simple. It grabs the length of the given string using Floyd's length method and then loops through each character of the string, making syscalls to Writer_io_write. for each character.

Strings

Strings are also implemented through the Floyd standard library. Similar to languages like C# and Java, Floyd's built-in string datatype is an alias of the String class. The similarities end there because behind the hood, Floyd strings are implemented using linked lists. Each character is a CharNode with a reference to the character it holds and a pointer to the next CharNode. String literal code generation is different from regular object code generation. Instead of simply calling calloc directly, a call to string_fromlit is made. String_fromlit is defined in the stdlib.c and it

allocates a CharNode-sized chunk of memory for each character in the string and links them together to form a Floyd string.

Testing and Bug Report

Note: Testing with comma delimited inputs and outputs refer to multiple executions of the same program but with different input data which provides different output data.

Official Test Files

File Name	Result
assign1.floyd	**PASS**
breakit.floyd	**PASS**
loopy.floyd	**PASS**
testnum.floyd	**PASS**
cbasics.floyd	**PASS**
cchange.floyd	**PASS**
citerfact.floyd	**PASS**
cfact.floyd	**PASS**
cgcd.floyd	**PASS**
bchange.floyd	**PASS**
blist.floyd	**PASS**
bnulltest.floyd	**PASS**
bobjbasics.floyd	**PASS**
bstrbasics.floyd	**PASS**
Bstrlits.floyd	**PASS**
aarrlist.floyd	**FAIL**
atest1.floyd	**FAIL**
atest2.floyd	**FAIL**
atestpoly.floyd	**FAIL**

Phase 4

File Name	Input	Output	Result
assign1.floyd	[NONE]	15 2 14 -4 0	**PASS**
breakit.floyd	10	120 -120 120 -1 -2 1 -1 -2 2 -1 -2 3	**PASS**
loopy.floyd	[NONE]	3 2 1	**PASS**
testnum.floyd	0, 123, -55	0 9 9, 1 9, -1 9 9	**PASS**

Phase 5 C

File Name	Input	Output	Result
cbasics.floyd	[NONE]	0 10 20 0 10 -5 0 10 20 -5 25	**PASS**
cchange.floyd	105, 216, 1	4 0 1, 8 1 1 1, 0 0 0 1	**PASS**
cfact.floyd	5, 12, 1	120, 479001600, 1	**PASS**
citerfact.floyd	5, 12, 1	120, 479001600, 1	**PASS**
cgcd.floyd	125 225	25	**PASS**
cgcd.floyd	160 4900	20	**PASS**

Phase 5 B

File Name	Input	Output	Result
bchange.floyd	105	Quarters: 4 Dimes: 0 Nickels: 1 Pennies: 0	**PASS**
bchange.floyd	216	Quarters: 8 Dimes: 1 Nickels: 1 Pennies: 1	**PASS**
bchange.floyd	1	Quarters: 0 Dimes: 0 Nickels: 0 Pennies: 1	**PASS**
blist.floyd (tostring uncommented)	[NONE]	10 20 5 50 [10,20,5,50] []	**PASS**
bnulltest.floyd	[NONE]	5 10 Null pointer exception on line 54	**PASS**
bobjbasics.floyd	[NONE]	5 10 100 200 -5 10 100 200	**PASS**
bstrbasics.floyd	jaja	Enter a string of characters:jaja s has 4 characters. charAt(0) = 'j' s > ' ' s >= ' ' q = wowsers!	**PASS**
bstrbasics.floyd	:thinking:	s has 10 characters. charAt(0) = ':' s > ' ' s >= ' ' q = wowsers!	**PASS**
bstrbasics.floyd	[NONE]	s has 0 characters. charAt(0) = '□' ! s > ' ' ! s >= ' ' q = wowsers!	**PASS**
bstrlits.floyd	[NONE]	q = This is a test. q = This is a tab. carriage return. q = This is a newline.	**PASS**

		q = This is a form feed. q = This is a\backslash. q = This is a "quote" q = This is an octal tab.	

Phase 5 A

File Name	Input	Output	Result
atest1.floyd	[NONE]	atest1.floyd:24,0:Unsupported feature: Inheritance atest1.floyd:29,4:Attempting to call undefined function initP atest1.floyd:42,17:Attempting to call undefined function getX atest1.floyd:42,4:Type mismatch for out: Expected int but got <error>	**FAIL**
atest2.floyd	[NONE]	atest2.floyd:24,0:Unsupported feature: Inheritance atest2.floyd:30,4:Attempting to call undefined function initEmp atest2.floyd:43,0:Unsupported feature: Inheritance atest2.floyd:48,4:Attempting to call undefined function initEmp atest2.floyd:55,26:Attempting to call undefined function getAge atest2.floyd:76,4:Type mismatch in assignment statement. Oyd expected on RHS, got SalariedEmployee atest2.floyd:77,4:Type mismatch in assignment statement. Employee expected on RHS, got Oyd atest2.floyd:80,4:Type mismatch in assignment statement. Oyd expected on RHS, got Point atest2.floyd:81,4:Type mismatch in assignment statement. Employee expected on RHS, got Oyd	**FAIL**
atestpoly.floyd	[NONE]	atestpoly.floyd:36,0:Unsupported feature: Inheritance atestpoly.floyd:41,4:Attempting to call undefined function initP	**FAIL**

		atestpoly.floyd:43,4:Attempting to call undefined function setKind atestpoly.floyd:55,17:Attempting to call undefined function getX atestpoly.floyd:55,4:Type mismatch for out: Expected int but got <error> atestpoly.floyd:68,6:Type mismatch in assignment statement. Child expected on RHS, got Parent atestpoly.floyd:80,4:Type mismatch in assignment statement. Parent expected on RHS, got Child atestpoly.floyd:88,4:Type mismatch in assignment statement. Child expected on RHS, got Parent	
aarlist.floyd	[NONE]	aarrlist.floyd:187,0:Unsupported feature: Inheritance aarrlist.floyd:193,4:Attempting to call undefined function initEmp aarrlist.floyd:211,0:Unsupported feature: Inheritance aarrlist.floyd:216,4:Attempting to call undefined function initEmp aarrlist.floyd:228,26:Attempting to call undefined function getAge aarrlist.floyd:243,4:Type mismatch for list: Expected Oyd but got String aarrlist.floyd:244,4:Type mismatch for list: Expected Oyd but got String aarrlist.floyd:245,4:Type mismatch for list: Expected Oyd but got Employee aarrlist.floyd:246,4:Type mismatch for list: Expected Oyd but got SalariedEmployee aarrlist.floyd:247,4:Type mismatch for list: Expected Oyd but got HourlyEmployee	**FAIL**

		aarrlist.floyd:250,4:Type mismatch for list: Expected Oyd but got String	

Appendix A: Source Code

SemanticChecker.java

Apr 24, 18 0:19 SemanticChecker.java Page 1/22

```
/*
Name: Italo Moraes (IMORA128)
Class: CpS 450
Filename: SemanticChecker.java
Description: Contains the class that checks for and prints semantic checks.
*/
package cps450;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.HashMap;
import java.util.List;
import java.util.Scanner;
import javax.lang.model.element.TypeElement;
import org.antlr.v4.runtime.ParserRuleContext;
import cps450.FloydParser.AddMinus_ExpContext;
import cps450.FloydParser.AddMulti_ExpContext;
import cps450.FloydParser.AddPlus_ExpContext;
import cps450.FloydParser.AndConcat_ExpContext;
import cps450.FloydParser.AndX_ExpContext;
import cps450.FloydParser.And_expContext;
import cps450.FloydParser.Argument_declContext;
import cps450.FloydParser.Argument_decl_listContext;
import cps450.FloydParser.Assignment_stmtContext;
import cps450.FloydParser.Call_stmtContext;
import cps450.FloydParser.Class_Context;
import cps450.FloydParser.ConcatAdd_ExpContext;
import cps450.FloydParser.ConcatX_ExpContext;
import cps450.FloydParser.Concat_expContext;
import cps450.FloydParser.ExprCont_ArrayContext;
import cps450.FloydParser.ExprCont_FalseContext;
import cps450.FloydParser.ExprCont_IDContext;
import cps450.FloydParser.ExprCont_IDExprContext;
import cps450.FloydParser.Method_expContext;
import cps450.FloydParser.MultiDIV_ExpContext;
import cps450.FloydParser.MultiTimes_ExpContext;
import cps450.FloydParser.MultiUnary_ExpContext;
import cps450.FloydParser.OrAnd_ExpContext;
import cps450.FloydParser.OrX_ExpContext;
import cps450.FloydParser.Or_expContext;
import cps450.FloydParser.Relate_switch_expContext;
import cps450.FloydParser.RelationalEQ_ExpContext;
import cps450.FloydParser.RelationalGE_ExpContext;
import cps450.FloydParser.RelationalGT_ExpContext;
import cps450.FloydParser.Relational_expContext;
import cps450.FloydParser.StartContext;
import cps450.FloydParser.TypeBoolContext;
import cps450.FloydParser.TypeIDContext;
import cps450.FloydParser.Var_declContext;
import cps450.FloydParser.TypeIntContext;
import cps450.FloydParser.TypeStringContext;
import cps450.FloydParser.UnaryMethod_ExpContext;
import cps450.FloydParser.UnaryMinus_ExpContext;
import cps450.FloydParser.Unary_expContext;
import cps450.FloydParser.ExprCont_StrLitContext;
import cps450.FloydParser.ExprCont_TrueContext;
import cps450.FloydParser.Expr_ExprContext;
import cps450.FloydParser.If_stmtContext;
import cps450.FloydParser.Loop_stmtContext;
import cps450.FloydParser.MethodDot_ExpContext;
//import cps450.FloydParser.MethodDot_ExpContext;
import cps450.FloydParser.ExprCont_IntLitContext;
import cps450.FloydParser.ExprCont_MEContext;
import cps450.FloydParser.ExprCont_NewContext;
import cps450.FloydParser.ExprCont_NullContext;
import cps450.FloydParser.ExprCont_ParExpContext;
import cps450.FloydParser.UnaryNot_ExpContext;
import cps450.FloydParser.UnaryPlus_ExpContext;
import cps450.FloydParser.MethodExpr_ContContext;
import cps450.FloydParser.Method_declContext;
```


Apr 24, 18 0:19 SemanticChecker.java Page 2/22

```
/*
Function Name: SemanticChecker
Description: Performs semantic checks on the code the user provided.
*/
public class SemanticChecker extends FloydBaseListener {
        //instance scope is always 1, local scope is always 2.
        static int INSTANCE_SCOPE = 1;
        //singleton symbol table
        SymbolTable symTable;
        MyError print = new MyError(false);
        //needed for things like file name info.
        Option opt;
        //needed for code gen
        String currentClass = "";
        //necessary for calculating instance offsets
        int InstanceVarOffset = 8;

        /*
        Function Name: SemanticChecker (Constructor)
        Description: io_write and io_read need to be pushed into the symbol table at
        this time to prevent issues with
        in/out.
        */
        SemanticChecker(Option opt) {
                this.opt = opt;
                print.opt = opt;
                symTable = SymbolTable.getInstance();

                MethodDeclaration io_write = new MethodDeclaration(Type.VOID);
                MethodDeclaration io_read = new MethodDeclaration(Type.INT);
                io_write.appendParameter(Type.INT, "ch");
                symTable.push("io_write", io_write);
                symTable.push("io_read", io_read);

        }

        /*
        Function Name: doesTypeExist
        Description: Checks if the parameter type is either a predefined type or if
        it's an existing class type.
        */
        boolean doesTypeExist(Type t) {
                if (t == Type.INT || t == Type.BOOLEAN || t == Type.STRING || Ty
pe.getTypeForName(t.name) != null) {
                        return true;
                }
                else {
                        return false;
                }
        }

        /*
        Function Name: exprCompareTypes
        Description: Created to make life easier on some expression type comparisons
        .
        */
        Type exprCompareTypes(Type expectedType, Type givenType1, Type givenType
2,ParserRuleContext ctx, List<String> fields) {
                if (givenType1 == Type.ERROR || givenType2 == Type.ERROR) {
                        return Type.ERROR;
                }
                if (expectedType == givenType1) {
                        if (expectedType == givenType2) {
                                return expectedType;
                        } else {
                                print.err(String.format(print.errMsgs.get(fields
.get(0)), fields.get(1), expectedType, givenType2),ctx);
                                return Type.ERROR;
                        }
                } else {
```

```
                        print.err(String.format(print.errMsgs.get(fields.get(0))
, fields.get(1), expectedType, givenTypel),ctx);
                        return Type.ERROR;
                }
        }
    /*
    Function Name: exprCompareTypes
    Description: Created to make life easier on unary type comparisons.
    */
        Type exprCompareTypes(Type expectedType, Type givenType, String errMsg,
ParserRuleContext ctx) {
                if (expectedType == givenType && givenType != Type.ERROR) {
                        return expectedType;
                } else {
                        print.err(errMsg, ctx);
                        return Type.ERROR;
                }
        }
    /*
    Function Name: exitVar_decl
    Description: Checks all the typing related to variable declarations. The var
iable offsets for instance variables
    and local variables are set here.
    */
        @Override
        public void exitVar_decl(Var_declContext ctx) {
                //Assignment is not allowed within variable declaration.
                if (ctx.children.contains(ctx.ASSIGNMENT_OPERATOR())) {
                        print.err(String.format(print.errMsgs.get("Unsupported"),
                                                "Attempting to initialize a variable in the declaration sectio
n."),ctx);
                        return;
                }

                //Making sure a type is beign given and that it exists
                if (ctx.ty != null && doesTypeExist(ctx.type().myType)) {
                        Symbol sym = symTable.lookup(ctx.IDENTIFIER().toString()
);
                        if (sym != null && sym.getScope() == symTable.getScope()
) {
                                print.err(String.format(print.errMsgs.get("Redefi
neVar"),
                                                        ctx.IDENTIFIER().toString()),ctx
);
                                return;
                        }
                        //If it's an instance variable, set its offset, put it i
nto the class declaration and push it into the
                        //symbol table
                        if (symTable.getScope() == INSTANCE_SCOPE) {
                                VarDeclaration classVariable = new VarDeclaratio
n(ctx.type().myType, ctx.IDENTIFIER().toString());
                                //setting offset of instance var
                                classVariable.setOffset(instanceVarOffset);
                                instanceVarOffset += 4;
                                Type foo = Type.getTypeForName(currentClass);
                                foo.getClassDecl().appendVar(classVariable);
                                symTable.push(ctx.IDENTIFIER().toString(), class
Variable);
                                ctx.sym = symTable.lookup(ctx.IDENTIFIER().toStr
ing());
                                return;
                        }
                        //Sets the offset for local variable and pushes them int
o the symbol table
                        VarDeclaration variable = new VarDeclaration(ctx.type().
myType, ctx.IDENTIFIER().toString());
                        variable.setOffset(symTable.getLocalOffset());
                        symTable.setLocalOffset(symTable.getLocalOffset() - 4);
                        symTable.push(ctx.IDENTIFIER().toString(), variable);
```



```
                        ctx.sym = symTable.lookup(ctx.IDENTIFIER().toString());

                }
                else {
                        print.err(print.errMsgs.get("BadVarType"),ctx);
                }


                super.exitVar_decl(ctx);
        }


    /*
    Function Name: exitAssignment_stmt
    Description: Checks that the types match for assignment.
    */
        @Override
        public void exitAssignment_stmt(Assignment_stmtContext ctx) {
                Type lhs = null;
                Type rhs = null;
                //LHS must be a variable that we're assigning to, so it needs to
 grab it from the symbol table.
                Symbol sym = symTable.lookup(ctx.IDENTIFIER().getText());
                if (sym != null) {
                        for (int i = 0; i < ctx.expression().size(); i++) {
                                lhs = ctx.expression(i).myType;
                                rhs = sym.getDecl().type;
                                if (Type.getTypeForName(lhs.name) != null) {
                                        lhs = Type.getTypeForName(lhs.name);
                                }
                                if (Type.getTypeForName(rhs.name) != null) {
                                        rhs = Type.getTypeForName(rhs.name);
                                }
                                if (lhs != Type.ERROR && rhs != lhs) {
                                        print.err(String.format(print.errMsgs.ge
t("AssMismatch"),
                                                                sym.getDecl().type, ctx.
expression(i).myType),ctx);
                                }
                        }
                        ctx.sym = sym;
                }
                else {


                        print.err(String.format(print.errMsgs.get("UndefinedVar"),

                                                ctx.IDENTIFIER().getText()),ctx);
                }
                super.exitAssignment_stmt(ctx);
        }

    /*
    Function Name: exitExprCont_ParExp
    Description: Expression inside parentheses (exp). Sets the type to whatever
its child was
    and moves along. No checks needed here.
    */
        @Override
        public void exitExprCont_ParExp(ExprCont_ParExpContext ctx) {
                ctx.myType = ctx.expression().myType;
                super.exitExprCont_ParExp(ctx);
        }

    /*
    Function Name: exitExprCont_Array
    Description: Arrays are not supported at this time.
    */
        @Override
        public void exitExprCont_Array(ExprCont_ArrayContext ctx) {
```

```
                ctx.myType = Type.ERROR;
                print.err(String.format(print.errMsgs.get("Unsupported"), "Array"),
ctx);
                super.exitExprCont_Array(ctx);
        }


    /*
    Function Name: exitIf_stmt
    Description: If statement type check.. Just checking if the condition is a b
oolean.
    */
        @Override
        public void exitIf_stmt(If_stmtContext ctx) {
                if (ctx.expression().myType == Type.BOOLEAN) {
                }
                else {
                        print.err(String.format(print.errMsgs.get("TypeMismatch")
,
                                        ctx.IF(0).toString(), "boolean", ctx.expr
ession().myType),ctx);
                }

                super.exitIf_stmt(ctx);
        }


    /*
    Function Name: exitLoop_stmt
    Description: Once again, just checking that the conditional is a boolean.
    */
        @Override
        public void exitLoop_stmt(Loop_stmtContext ctx) {
                if (ctx.expression().myType == Type.BOOLEAN) {
                }
                else {
                        print.err(String.format(print.errMsgs.get("TypeMismatch")
,
                                        ctx.WHILE().toString(), "boolean", ctx.ex
pression().myType),ctx);
                }

                super.exitLoop_stmt(ctx);
        }

//    /*
//    Function Name: exitExprRelationalExpr
//    Description: Just pushing up the type up the tree for the functions that n
eed it.
//    */
//      @Override
//      public void exitExprRelational_Expr(ExprRelational_ExprContext ctx) {
//              if (ctx.relational_exp().myType != null) {
//                      ctx.myType = ctx.relational_exp().myType;
//              } else {
//                      ctx.myType = Type.ERROR;
//                      print.err("exitExprRelational_Expr: No type to pass up t
he tree",ctx);
//              }
//              super.exitExprRelational_Expr(ctx);
//      }

    /*
    Function Name: exitExprOr_Expr
    Description: Just pushing up the type up the tree for the functions that nee
d it.
    */
        @Override
        public void exitExprOr_Expr(ExprOr_ExprContext ctx) {
                if (ctx.or_exp().myType != null) {
```



```
                        ctx.myType = ctx.or_exp().myType;
                }
                else {
                        ctx.myType = Type.ERROR;
                        print.err("exitExprOr_Expr: No type to pass up the tree",ctx);
                }
                super.exitExprOr_Expr(ctx);
        }

    /*
    Function Name: exitRelationalGE_Exp
    Description: Type checking the operands used by >=. Only strings and ints ar
e allowed, otherwise
    an error is thrown. NOTE: All relational operators must push up a boolean ty
pe.
    */
        @Override
        public void exitRelationalGE_Exp(RelationalGE_ExpContext ctx) {
                if (ctx.e1.myType == Type.STRING) {
                        List<String> foo = Arrays.asList("TypeMismatch", ctx.GE()
.toString(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                        ctx.myType = exprCompareTypes(Type.STRING, ctx.e1.myType
, ctx.e2.myType, ctx, foo);
                        if (ctx.myType == Type.STRING) {ctx.myType = Type.BOOLEA
N;}
                }
                else if (ctx.e1.myType == Type.INT) {
                        List<String> foo = Arrays.asList("TypeMismatch", ctx.GE()
.toString(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                        ctx.myType = exprCompareTypes(Type.INT, ctx.e1.myType, c
tx.e2.myType, ctx, foo);
                        if (ctx.myType == Type.INT) {ctx.myType = Type.BOOLEAN;}
                }
                else {
                        print.err(String.format(print.errMsgs.get("TypeMismatch")
,
                        ctx.GE().toString(), "int or string ", ctx.e1.myType),ctx);
                        ctx.myType = Type.ERROR;
                }
                super.exitRelationalGE_Exp(ctx);
        }

    /*
    Function Name: exitRelationalGT_Exp
    Description: Type checking the operands used by >. String and ints are the o
nly types allowed.
    */
        @Override
        public void exitRelationalGT_Exp(RelationalGT_ExpContext ctx) {

                if (ctx.e1.myType == Type.STRING) {
                        List<String> foo = Arrays.asList("TypeMismatch", ctx.GT()
.toString(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                        ctx.myType = exprCompareTypes(Type.STRING, ctx.e1.myType
, ctx.e2.myType, ctx, foo);
                        if (ctx.myType == Type.STRING) {ctx.myType = Type.BOOLEA
N;}
                }
                else if (ctx.e1.myType == Type.INT) {
                        List<String> foo = Arrays.asList("TypeMismatch", ctx.GT()
.toString(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                        ctx.myType = exprCompareTypes(Type.INT, ctx.e1.myType, c
tx.e2.myType, ctx, foo);
                        if (ctx.myType == Type.INT) {ctx.myType = Type.BOOLEAN;}
                }
                else {
                        print.err(String.format(print.errMsgs.get("TypeMismatch")
,
                        ctx.GT().toString(), "int or string ", ctx.e1.myType),ctx);
                        ctx.myType = Type.ERROR;
```

```
                }


                super.exitRelationalGT_Exp(ctx);
        }

    /*
    Function Name: exitRelationalEQ_Exp
    Description: Type checking the operands used by =.
    All types are allowed, including class types, but obviously ,they must match
.
    */
        @Override
        public void exitRelationalEQ_Exp(RelationalEQ_ExpContext ctx) {
                //making sure equality tests can be don
                Type classTestType = Type.getTypeForName(ctx.e1.myType.name);
                if (classTestType != null && (classTestType == Type.getTypeForNa
me(ctx.e2.myType.name)) ) {
                        ctx.myType = Type.BOOLEAN;
                        return;
                }

                if (ctx.e1.myType == Type.INT) {
                        List<String> foo = Arrays.asList("TypeMismatch", ctx.EQ()
.toString(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                        ctx.myType = exprCompareTypes(Type.INT, ctx.e1.myType, c
tx.e2.myType, ctx, foo);
                        if (ctx.myType == Type.INT) {ctx.myType = Type.BOOLEAN;}
                }
                else if (ctx.e1.myType == Type.STRING) {
                        List<String> foo = Arrays.asList("TypeMismatch", ctx.EQ()
.toString(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                        ctx.myType = exprCompareTypes(Type.STRING, ctx.e1.myType
, ctx.e2.myType, ctx, foo);
                        if (ctx.myType == Type.STRING) {ctx.myType = Type.BOOLEA
N;}
                }
                else if (ctx.e1.myType == Type.BOOLEAN) {
                        List<String> foo = Arrays.asList("TypeMismatch", ctx.EQ()
.toString(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                        ctx.myType = exprCompareTypes(Type.BOOLEAN, ctx.e1.myTyp
e, ctx.e2.myType, ctx, foo);
                } else {
                        print.err(String.format(print.errMsgs.get("TypeMismatch")
,
                        ctx.EQ().toString(), "int, string, or bool ", ctx.e1.myType),ctx
);
                        ctx.myType = Type.ERROR;
                }
                super.exitRelationalEQ_Exp(ctx);
        }


//    /*
//    Function Name: exitRelationalOr_Exp
//    Description: Pushing up the type for the functions that need it.
//    */
//      @Override
//      public void exitRelationalOr_Exp(RelationalOr_ExpContext ctx) {
//              if (ctx.or_exp().myType != null) {
//                      ctx.myType = ctx.or_exp().myType;
//              }
//              else
//              {
//                      ctx.myType = Type.ERROR;
//                      print.err("exitRelationalOr_Exp: No type to pass up the
tree",ctx);
//              }
//              super.exitRelationalOr_Exp(ctx);
//      }
```



```


    /*
    Function Name: exitOrX_Exp
    Description: Type checking for ors. Both operands need to be booleans.
    */
        @Override
        public void exitOrX_Exp(OrX_ExpContext ctx) {
                List<String> foo = Arrays.asList("TypeMismatch", ctx.OR().toStrin
g(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                ctx.myType = exprCompareTypes(Type.BOOLEAN, ctx.e1.myType, ctx.e
2.myType, ctx, foo);
        }

    /*
    Function Name: exitOrAnd_Exp
    Description: Pushing up the type for the functions that need it.
    */
        @Override
        public void exitOrAnd_Exp(OrAnd_ExpContext ctx) {
                if (ctx.and_exp().myType != null) {
                        ctx.myType = ctx.and_exp().myType;
                }
                else{
                        ctx.myType = Type.ERROR;
                        print.err("exitOrAnd_Exp: No type to pass up the tree",ctx);
                }
                super.exitOrAnd_Exp(ctx);
        }

    @Override
        public void exitRelate_switch_exp(Relate_switch_expContext ctx) {
                if (ctx.concat_exp() != null) {
                        ctx.myType = ctx.concat_exp().myType;
                } else if (ctx.relational_exp() != null) {
                        ctx.myType = ctx.relational_exp().myType;
                } else {
                        ctx.myType = Type.ERROR;
                        print.err("exitRelate_switch_exp: No type to pass up the tree",ctx);
                }
                super.exitRelate_switch_exp(ctx);
        }
        /*
    Function Name: exitAndX_Exp
    Description: Type checking for Ands. Need to be booleans.
    */
        @Override
        public void exitAndX_Exp(AndX_ExpContext ctx) {
                List<String> foo = Arrays.asList("TypeMismatch", ctx.AND().toStri
ng(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                ctx.myType = exprCompareTypes(Type.BOOLEAN, ctx.e1.myType, ctx.e
2.myType, ctx, foo);
                super.exitAndX_Exp(ctx);
        }



    /*
    Function Name: exitAndConcat_Exp
    Description: Pushing up the type for the functions that need it.
    */
        @Override
        public void exitAndConcat_Exp(AndConcat_ExpContext ctx) {
                if (ctx.relate_switch_exp()  != null) {
                        ctx.myType = ctx.relate_switch_exp().myType;
                }
                else
                {
                        ctx.myType = Type.ERROR;
                        print.err("exitAndConcat_Exp: No type to pass up the tree",ctx);
```

```
                }
                super.exitAndConcat_Exp(ctx);
        }

    /*
    Function Name: exitConcatX_Exp
    Description: Type checking for the ampersand (concat) operator. The type is
required to be a string. Not implemented
    into the language, but you could always use the floyd concat function.
    */
        @Override
        public void exitConcatX_Exp(ConcatX_ExpContext ctx) {
                List<String> foo = Arrays.asList("TypeMismatch", ctx.AMPERSAND().
toString(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                ctx.myType = exprCompareTypes(Type.STRING, ctx.e1.myType, ctx.e2
.myType, ctx, foo);
                super.exitConcatX_Exp(ctx);
        }

    /*
    Function Name: exitConcatAdd_Exp
    Description: Pushing up the type for the functions that need it.
    */
        @Override
        public void exitConcatAdd_Exp(ConcatAdd_ExpContext ctx) {
                if (ctx.add_exp().myType != null) {
                        ctx.myType = ctx.add_exp().myType;
                }
                else
                {
                        ctx.myType = Type.ERROR;
                        print.err("exitConcatAdd_Exp: No type to pass up the tree",ctx);
                }
                super.exitConcatAdd_Exp(ctx);
        }

    /*
    Function Name: exitAddPlus_Exp
    Description: Type checking for the plus operator. The operands need to be in
tegers.
    */
        @Override
        public void exitAddPlus_Exp(AddPlus_ExpContext ctx) {
                List<String> foo = Arrays.asList("TypeMismatch", ctx.PLUS().toStr
ing(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                ctx.myType = exprCompareTypes(Type.INT, ctx.e1.myType, ctx.e2.my
Type, ctx, foo);
                super.exitAddPlus_Exp(ctx);
        }

    /*
    Function Name: exitAddMinus_Exp
    Description: Type checking for the minus operator. The operands need to be i
ntegers.
    */
        @Override
        public void exitAddMinus_Exp(AddMinus_ExpContext ctx) {
                List<String> foo = Arrays.asList("TypeMismatch", ctx.MINUS().toSt
ring(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                ctx.myType = exprCompareTypes(Type.INT, ctx.e1.myType, ctx.e2.my
Type, ctx, foo);
                super.exitAddMinus_Exp(ctx);
        }

    /*
    Function Name: exitAddMulti_Exp
    Description: Pushing up the type for the functions that need it.
    */
        @Override
        public void exitAddMulti_Exp(AddMulti_ExpContext ctx) {
```



```
                if (ctx.multi_exp().myType != null) {
                        ctx.myType = ctx.multi_exp().myType;
                }
                else
                {
                        ctx.myType = Type.ERROR;
                        print.err("exitAddMulti_Exp: No type to pass up the tree",ctx);
                }
                super.exitAddMulti_Exp(ctx);
        }
    /*
    Function Name: exitMultiTimes_Exp
    Description: Type checking for the * operator. Both operands must be integer
s.
    */
        @Override
        public void exitMultiTimes_Exp(MultiTimes_ExpContext ctx) {
                List<String> foo = Arrays.asList("TypeMismatch", ctx.TIMES().toSt
ring(), ctx.e1.myType.toString(), ctx.e2.myType.toString());
                ctx.myType = exprCompareTypes(Type.INT, ctx.e1.myType, ctx.e2.my
Type, ctx, foo);
                super.exitMultiTimes_Exp(ctx);
        }

    /*
    Function Name: exitMultiDIV_Exp
    Description: Type checking for the / operator. Both operands must be integer
s.
    */
        @Override
        public void exitMultiDIV_Exp(MultiDIV_ExpContext ctx) {
                List<String> foo = Arrays.asList("TypeMismatch", ctx.DIV().toStri
ng(),ctx.e1.myType.toString(), ctx.e2.myType.toString());
                ctx.myType = exprCompareTypes(Type.INT, ctx.e1.myType, ctx.e2.my
Type, ctx, foo);
                super.exitMultiDIV_Exp(ctx);
        }

    /*
    Function Name: exitMultiUnary_Exp
    Description: Pushing up the type for the functions that need it.
    */
        @Override
        public void exitMultiUnary_Exp(MultiUnary_ExpContext ctx) {
                if (ctx.unary_exp().myType != null) {
                        ctx.myType = ctx.unary_exp().myType;
                }
                else
                {
                        ctx.myType = Type.ERROR;
                        print.err("exitMultiUnary_Exp: No type to pass up the tree",ctx);
                }
                super.exitMultiUnary_Exp(ctx);
        }

    /*
    Function Name: exitUnaryPlus_Exp
    Description: Checking the unary + operand. It needs to be an integer.
    */
        @Override
        public void exitUnaryPlus_Exp(UnaryPlus_ExpContext ctx) {
                String errorMsg = String.format(print.errMsgs.get("TypeMismatch")
,
                                        ctx.PLUS().toString(),"int", ctx.unary_exp().myTy
pe);
                ctx.myType = exprCompareTypes(Type.INT, ctx.unary_exp().myType,
errorMsg, ctx);
                super.exitUnaryPlus_Exp(ctx);
        }
```

```
    /*
    Function Name: exitUnaryMinus_Exp
    Description: Checking the unary - operand. It needs to be an integer.
    */
        @Override
        public void exitUnaryMinus_Exp(UnaryMinus_ExpContext ctx) {
                String errorMsg = String.format(print.errMsgs.get("TypeMismatch")
,
                                ctx.MINUS().toString(), "int", ctx.unary_exp().my
Type);
                ctx.myType = exprCompareTypes(Type.INT, ctx.unary_exp().myType,
errorMsg, ctx);
        }

    /*
    Function Name: exitUnaryNot_Exp
    Description: Checking the unary not operand. It needs to be a bool
    */
        @Override
        public void exitUnaryNot_Exp(UnaryNot_ExpContext ctx) {
                String errorMsg = String.format(print.errMsgs.get("TypeMismatch")
,
                                ctx.NOT().toString(), "boolean", ctx.unary_exp().
myType);
                ctx.myType = exprCompareTypes(Type.BOOLEAN, ctx.unary_exp().myTy
pe,errorMsg, ctx);
                super.enterUnaryNot_Exp(ctx);
        }

    /*
    Function Name: exitUnaryMethod_Exp
    Description: Passing up the type for the functions that need it.
    */
        @Override
        public void exitUnaryMethod_Exp(UnaryMethod_ExpContext ctx) {
                if (ctx.method_exp().myType != null) {
                        ctx.myType = ctx.method_exp().myType;
                }
                else
                {
                        ctx.myType = Type.ERROR;
                        print.err("exitUnaryMethod_Exp: No type to pass up the tree",ctx);
                }
                super.exitUnaryMethod_Exp(ctx);
        }

    /*
    Function Name: exitMethodExpr_Cont
    Description: Passing up the type for the functions that need it.
    */
        @Override
        public void exitMethodExpr_Cont(MethodExpr_ContContext ctx) {
                ctx.myType = ctx.expr_cont().myType;
                if (ctx.expr_cont().myType != null) {
                        ctx.myType = ctx.expr_cont().myType;
                }
                else
                {
                        ctx.myType = Type.ERROR;
                        print.err("exitMethodExpr_Cont: No type to pass up the tree",ctx);
                }

                super.exitMethodExpr_Cont(ctx);
        }

    /*
    Function Name: exitExprCont_New
    Description: Makes sure the LHS type is a user defined class type, otherwise
 it's an error.
    */
```



```
@Override
        public void exitExprCont_New(ExprCont_NewContext ctx) {
        if (Type.getTypeForName(ctx.typ.getText()) != null ) {
                ctx.myType = Type.getTypeForName(ctx.typ.getText());
                ctx.paramNum = ctx.myType.getClassDecl().vars.size();
        } else {
                print.err(String.format("Invalid type %s on RHS of new.", ctx.typ.getTex
t()),ctx);
                ctx.myType = Type.ERROR;
        }

                super.exitExprCont_New(ctx);
        }


        /*
        Function Name: exitExprCont_Null
        Description: A little tricky to do with the listener approach, so I have
 to loop up the to get the LHS and
        check it's a user defined class type.
        */
        @Override
        public void exitExprCont_Null(ExprCont_NullContext ctx) {
           Assignment_stmtContext nullAssign = null;
           RelationalEQ_ExpContext equalityTest = null;
           ParserRuleContext foo = ctx;
                while (foo != null) {
                        if (foo.getParent() instanceof Assignment_stmtContext) {
                                nullAssign = (Assignment_stmtContext)foo.getPare
nt();
                                break;
                        } else if (foo.getParent() instanceof RelationalEQ_ExpCo
ntext) {
                                equalityTest = (RelationalEQ_ExpContext)foo.getP
arent();
                        }

                        foo = foo.getParent();
                }
                //if its not null, check that it's a class type.
                if (nullAssign != null) {
                        Type typerino = symTable.lookup(nullAssign.IDENTIFIER().
getText()).getDecl().type;
                        //if its not a class type, error.
                        if (typerino == Type.INT ||typerino == Type.BOOLEAN ) {
                        ctx.myType = Type.ERROR;
                        print.err(String.format("Attempting to assign null to %s. Null can only
 be assigned to class types.", typerino),ctx);
                        return;

                        } else {
                                //it's not null and it's a class type, so we're
set.
                                ctx.myType = typerino;
                                return;
                        }
                } else if (equalityTest != null) {
                        if (Type.getTypeForName(equalityTest.e1.myType.name) ==
null) {
                                ctx.myType = Type.ERROR;
                                print.err(String.format("Cannot compare type %s to null"
, equalityTest.e1.myType),ctx);
                                return;
                        } else {
                                ctx.myType = Type.getTypeForName(equalityTest.e1
.myType.name);
                                return;
                        }

                }
```

```
                        ctx.myType = Type.ERROR;
                        print.err("Null failed. Could not find LHS", ctx);

        }
        /*
        Function Name: exitExprCont_ME
        Description: Annoying like new because I had to loop up the tree to get
the type and make sure it was
        a user defined class type.
        */
        @Override
        public void exitExprCont_ME(ExprCont_MEContext ctx) {
                Assignment_stmtContext meAssign = null;
                Call_stmtContext meCall = null;

                ParserRuleContext foo = ctx;
                while (foo != null) {
                        if (foo.getParent() instanceof Assignment_stmtContext) {
                                meAssign = (Assignment_stmtContext)foo.getParent
();
                                break;
                        } else if (foo.getParent() instanceof Call_stmtContext)
{
                                meCall = (Call_stmtContext)foo.getParent();
                                break;
                        }

                        foo = foo.getParent();
                }

                //type for assignment statements
                if (meAssign != null) {
                        if (symTable.lookup(meAssign.IDENTIFIER().getText()) !=
null) {
                        ctx.myType = symTable.lookup(meAssign.IDENTIFIER().getTe
xt()).getDecl().type;
                        return;
                        }
                        //me type for call stmts
                } else if (meCall != null) {
                        if (meCall.t1.myType != null) {
                                ctx.myType = meCall.t1.myType;
                                return;
                        } else {
                                ctx.myType = Type.ERROR;

                                print.err("Call stmt obj expression type is null.", ctx);
                                return;
                        }
                }
                //if I make it here, then I never found the LHS type.
                ctx.myType = Type.ERROR;

                print.err("Me failed. No valid type found.", ctx);
                super.exitExprCont_ME(ctx);
        }

        /*
        Function Name: exitExprCont_Strlit
        Description: Just passes up the string type for the functions that need
it up the tree.
        */
        @Override
        public void exitExprCont_Strlit(ExprCont_StrlitContext ctx) {
                ctx.myType = Type.STRING;
                super.enterExprCont_Strlit(ctx);
        }
```



```
        /*
        Function Name: exitCall_stmt
        Description: Decorates the tree with the class name and symbol to be use
d in CodeGen. Checks that the function
        is either in a class declaration or currently in the symbol table.
        */
        @Override
        public void exitCall_stmt(Call_stmtContext ctx) {

                //class name for codegen
                ctx.className = currentClass;
                List<VarDeclaration> info = new ArrayList<VarDeclaration>();
                int paramNum = 0;
                //setting the parameter number if there are any
                if (ctx.expression_list() != null) {
                        paramNum = ctx.expression_list().expression().size();
                }
                //checking an object or type exists. If so, then it's a blank.fu
nction() kind of call.
                //If the type is the same as the current class name, then I need
 to look in the symbol table for the function definition.
                if (ctx.t1 != null && !(symTable.lookup(ctx.t1.getText()).getDec
l().type.getClassDecl().name.equals(currentClass))) {
                        Symbol foo = symTable.lookup(ctx.t1.getText());
                        ctx.sym = foo;
                        if (foo == null) {
                                print.err(String.format(print.errMsgs.get("Undefi
nedFunction"),
                                                ctx.IDENTIFIER().getText()),ctx)
;
                                return;
                        }

                                //checking inside of the class declaration to ma
ke sure the parameter number is correct
                                //and that each parameter type matches
                                if (foo.getDecl().type.getClassDecl().methods.co
ntainsKey(ctx.IDENTIFIER().getText())) {
                                        MethodDeclaration meth = foo.getDecl().t
ype.getClassDecl().methods.get(ctx.IDENTIFIER().getText());
                                        if (meth.parameters.size() != paramNum)
{
                                                print.err(String.format(print.er
rMsgs.get("ParameterNumberMismatch"),
                                                        meth.parameters.
size(), paramNum),ctx);

                                                return;
                                        } else {
                                                for (int i = 0; i < meth.paramet
ers.size(); i++) {
                                                        if (meth.parameters.get(
i).type == ctx.expression_list().expression().get(i).myType) {
                                                        } else {
                                                                print.err(String
.format(print.errMsgs.get("TypeMismatch"),

ctx.t1.getText(), meth.parameters.get(i).type, ctx.expression_list().expression(
).get(i).myType),ctx);
                                                                return;
                                                        }
                                                }
                                                return;
                                        }
                                }
                                else {
                                        print.err(String.format(print.errMsgs.ge
t("UndefinedFunction"),
                                                ctx.IDENTIFIER().getText
()),ctx);
```

```
                                        return;
                                }

                }


                if (symTable.lookup(ctx.IDENTIFIER().getText()) == null) {
                        print.err(String.format(print.errMsgs.get("UndefinedFunction
"),
                                        ctx.IDENTIFIER().getText()),ctx);

                        return;
                }


                //If I make it this far, then it's just a function call, with: f
unctionCall()
                //I do the same type and parameter number checks.
                MethodDeclaration mDecl = (MethodDeclaration) symTable.lookup(ct
x.IDENTIFIER().getText()).getDecl();
                info = mDecl.getParameters();
                if (symTable.lookup(ctx.IDENTIFIER().getText()) != null) {

                        if (info.size() == paramNum) {
                                for (int i = 0; i < info.size(); i++) {
                                        if (info.get(i).type == null) {
                                                VarDeclaration moo = info.get(i)
;
                                                moo.type = symTable.lookup(info.
get(i).name).getDecl().type;
                                                info.set(i, moo);
                                        }
                                        if ( info.get(i).type == ctx.expression_
list().expression().get(i).myType) {
                                        }
                                        else {
                                                print.err(String.format(print.er
rMsgs.get("ParameterMismatch"),
                                                                info.get(i).type
.toString(),
                                                                ctx.expression_l
ist().expression().get(i).myType.toString()),ctx);
                                                return;
                                        }
                                }
                                if (ctx.t1 != null) {
                                ctx.sym = symTable.lookup(ctx.t1.getText());
                                }
                        } else {
                                print.err(String.format(print.errMsgs.get("Parame
terNumberMismatch"),
                                                info.size(), ctx.expression_list
().expression().size()),ctx);
                                return;
                        }

                } else {
                        print.err(String.format(print.errMsgs.get("UndefinedFunction
"),
                                        ctx.IDENTIFIER().getText()),ctx);
                        return;
                }
                super.exitCall_stmt(ctx);
        }


        /*
        Function Name: exitExprCont_IDExpr
        Description: Expression call statement check is done here. Same checks a
s done up above on the
```



```
        call_stmt function. I have to check if it's just a regular function call
, or if theres an obj/type associated with it.
        */
        @Override
        public void exitExprCont_IDExpr(ExprCont_IDExprContext ctx) {
                int paramNum = 0;


                if (ctx.expression_list() != null) {
                        paramNum = ctx.expression_list().expression().size();
                }

                List<VarDeclaration> info = new ArrayList<VarDeclaration>();
                MethodDeclaration mDecl = null;
                //if we are dealing with obj.function() then we need to do that
here
                if (ctx.getParent() instanceof MethodDot_ExpContext) {
                        MethodDot_ExpContext objCheck = (MethodDot_ExpContext)ct
x.getParent();

                        Type objType = objCheck.e1.myType;


                        //checkikng if error so I don't propogate anymore error
msgs
                        if (objType == Type.ERROR) {
                                ctx.myType = Type.ERROR;
                                return;
                        }
                        //if it's null, then we have an issue. error an dleave.
                        if (objType == null){
                                print.err(String.format("The type of object %s is null.",
objCheck.e1.getText()),ctx);
                                ctx.myType = Type.ERROR;
                                return;
                        }
                        //This needs to be checked so it doesn't crash during th
e inheritance tests.
                        //Basically, it's trying to call a function that exists
in a parent class, so it'll
                        //give a null exception if i don't leave here
                        if (objType.getClassDecl() == null) {
                                ctx.myType = Type.ERROR;
                                return;
                        }
                        //CHECKING if this function is in the current class, if
so, then we look in symbol table
                        if (objType.getClassDecl().name.equals(currentClass) &&
symTable.lookup(ctx.IDENTIFIER().getText()) != null) {
                                mDecl = (MethodDeclaration) symTable.lookup(ctx.
IDENTIFIER().getText()).getDecl();
                                ctx.myType = mDecl.type;
                        }
                        //checking if function is in its type.classdecl
                        else if (objType != null) {
                                MethodDeclaration test = objType.getClassDecl().
methods.get(ctx.IDENTIFIER().getText());
                                if (test == null) {
                                        print.err(String.format(print.errMsgs.ge
t("UndefinedFunction"),
                                                        ctx.IDENTIFIER().getText
()),ctx);
                                        ctx.myType = Type.ERROR;
                                } else {
                                        mDecl = test;
                                         ctx.myType = mDecl.type;

                                }
```

```
                                //if it's not null, then check if the method we'
r elooking for exists in the class. if it does,
                                //then put set mDecl to the method in the class
                        }
                } else if (symTable.lookup(ctx.IDENTIFIER().getText()) == null)
{
                                print.err(String.format(print.errMsgs.get("Undefi
nedFunction"),
                                        ctx.IDENTIFIER().getText()),ctx);
                                ctx.myType = Type.ERROR;
                                return;
                } else if (symTable.lookup(ctx.IDENTIFIER().getText()) != null)
{
                        ctx.classType = Type.getTypeForName(currentClass);
                         mDecl = (MethodDeclaration) symTable.lookup(ctx.IDENTIF
IER().getText()).getDecl();
                         ctx.myType = mDecl.type;

                }

                        info = mDecl.getParameters();
                        if (info.size() == paramNum) {
                                for (int i = 0; i < info.size(); i++) {
                                        /*
                                         *
                                         * recursion case
                                         * If I don't do this, then I wont have
the types during a recursion call.
                                         */
                                        if (info.get(i).type == null) {
                                                VarDeclaration moo = info.get(i)
;
                                                moo.type = symTable.lookup(info.
get(i).name).getDecl().type;
                                                info.set(i, moo);
                                        }
                                        if ( info.get(i).type == ctx.expression_
list().expression().get(i).myType) {
                                        }
                                        else {
                                                print.err(String.format(print.er
rMsgs.get("ParameterMismatch"),
                                                                        info.get(i).type
.toString(),
                                                                        ctx.expression_l
ist().expression().get(i).myType.toString()),ctx);
                                                ctx.myType = Type.ERROR;
                                                return;
                                        }
                                }
                        } else {
                                print.err(String.format(print.errMsgs.get("Parame
terNumberMismatch"),
                                        info.size(), paramNum),ctx);
                                ctx.myType = Type.ERROR;
                                return;
                        }

                super.exitExprCont_IDExpr(ctx);
        }

        /*
        Function Name: exitExprCont_Intlit
        Description: Passing the type up the tree.
        */
        @Override
        public void exitExprCont_Intlit(ExprCont_IntlitContext ctx) {
                ctx.myType = Type.INT;
                super.enterExprCont_Intlit(ctx);
        }
```



```
        /*
        Function Name: exitExprCont_True
        Description: Passing the type up the tree.
        */
        @Override
        public void exitExprCont_True(ExprCont_TrueContext ctx) {
                ctx.myType = Type.BOOLEAN;
                super.exitExprCont_True(ctx);
        }
        /*
        Function Name: exitExprCont_False
        Description: Passing the type up the tree.
        */
        @Override
        public void exitExprCont_False(ExprCont_FalseContext ctx) {
                ctx.myType = Type.BOOLEAN;
                super.exitExprCont_False(ctx);
        }

        /*
        Function Name: exitExprCont_ID
        Description: Passing the type up the tree. Need to look up in the symbol
table first, of course.
        */
        @Override
        public void exitExprCont_ID(ExprCont_IDContext ctx) {

                Symbol sym = symTable.lookup(ctx.IDENTIFIER().getText());
                if (sym != null) {
                        ctx.sym = sym;
                        ctx.myType = sym.getDecl().type;
                }
                else {
                        print.err(String.format(print.errMsgs.get("UndefinedVar"),
                                                ctx.IDENTIFIER().getText()),ctx);
                        ctx.myType = Type.ERROR;
                }

                super.exitExprCont_ID(ctx);
        }

        /*
        Function Name: exitTypeInt
        Description: Passing the type up the tree.
        */
        @Override
        public void exitTypeInt(TypeIntContext ctx) {
                ctx.myType = Type.INT;
        }

        /*
        Function Name: exitTypeString
        Description: Passing the type up the tree.
        */
        @Override
        public void exitTypeString(TypeStringContext ctx) {
                ctx.myType = Type.STRING;
                super.enterTypeString(ctx);
        }
        /*
        Function Name: exitTypeBool
        Description: Passing the type up the tree.
        */
        @Override
        public void exitTypeBool(TypeBoolContext ctx) {
                ctx.myType = Type.BOOLEAN;
                super.enterTypeBool(ctx);
```

```
	}

	/*
	Function Name: exitTypeID
	Description: Passing the type up the tree.
	*/
	@Override
	public void exitTypeID(TypeIDContext ctx) {
		Symbol sym = symTable.lookup(ctx.IDENTIFIER().getText());
		if (sym != null) {
			ctx.myType = sym.getDecl().type;
		}
		else {
			print.err(String.format(print.errMsgs.get("UndefinedVar"),

					ctx.IDENTIFIER().getText()),ctx);
			ctx.myType = Type.ERROR;
		}
		super.exitTypeID(ctx);
	}

	/*
	Function Name: exitArgument_decl
	Description: Checking the parameter type the user is passing in.
	*/
	@Override
	public void exitArgument_decl(Argument_declContext ctx) {
		if (doesTypeExist(ctx.type().myType)) {
		}
		else {
			print.err(String.format("Type %s does not exist", ctx.type().m
yType), ctx);
		}
		super.exitArgument_decl(ctx);
	}

	/*
	Function Name: exitArgument_decl_list
	Description: Parameter types get set here. They're also pushed into the
symbol table.
	*/
	@Override
	public void exitArgument_decl_list(Argument_decl_listContext ctx) {
		int offset = 12;
		for (int i = 0; i < ctx.argument_decl().size(); i++) {
			VarDeclaration foo = new VarDeclaration(ctx.argument_dec
l(i).type().myType,ctx.argument_decl(i).IDENTIFIER().getText());
			foo.setOffset(offset);
			offset += 4;
			symTable.push(ctx.argument_decl(i).IDENTIFIER().getText(
), foo);

		}

		super.exitArgument_decl_list(ctx);
	}


	/*
	Function Name: exitMethod_decl
	Description: Decorating this node with the class name for use in code ge
n. Adding the parameters to the
	method and ending the scope to clean the locals off the symbol table.
	*/
	@Override
	public void exitMethod_decl(Method_declContext ctx) {
		ctx.className = currentClass;
		MethodDeclaration mDecl = (MethodDeclaration) symTable.lookup(ct
x.IDENTIFIER(0).getText()).getDecl();
		if (ctx.argument_decl_list() != null) {
```



```
			mDecl.clearParameters();
			for (int i = 0; i < ctx.argument_decl_list().argument_de
cl().size(); i++) {
				mDecl.appendParameter(ctx.argument_decl_list().a
rgument_decl(i).type().myType,ctx.argument_decl_list().argument_decl(i).IDENTIFI
ER().getText());
			}
		}
		ctx.params = ctx.var_decl().size();


		symTable.endScope();
		super.exitMethod_decl(ctx);
	}

	/*
	Function Name: enterMethod_decl
	Description: Checking the type of the function so it can be pushed corre
ctly onto the stack. Set the local offset
	to -8 so that it isn't giving offsets based on the last method declarati
on.
	*/
	@Override
	public void enterMethod_decl(Method_declContext ctx) {
		List<Type> types = Arrays.asList(Type.INT, Type.BOOLEAN, Type.ST
RING, Type.VOID);
		Type t = Type.VOID;

		for (Type i : types ) {
			if (ctx.typ != null && i.name.equals(ctx.typ.getText()))
 {
				t = i;
			}
		}
		//user defined types check
		if (t == Type.VOID && ctx.typ != null &&  Type.getTypeForName(ct
x.typ.getText()) != null) {
			t = Type.getTypeForName(ctx.typ.getText());
		}

		symTable.push(ctx.IDENTIFIER(0).getText(), new MethodDeclaration
(t));
		MethodDeclaration mDecl = (MethodDeclaration) symTable.lookup(ct
x.IDENTIFIER(0).getText()).getDecl();
		//giving offsets to all parameters. (POSITIVE)
		if (ctx.argument_decl_list() != null) {
			for (int i = 0; i < ctx.argument_decl_list().argument_de
cl().size(); i++) {
				mDecl.appendParameter(ctx.argument_decl_list().a
rgument_decl(i).type().myType,ctx.argument_decl_list().argument_decl(i).IDENTIFI
ER().getText());
			}
		}
		symTable.beginScope();
		//setting local offset to -8 because the function is beginning
		symTable.setLocalOffset(-8);
		super.enterMethod_decl(ctx);
	}

	/*
	Function Name: enterClass_
	Description:Need to push in the "in" and "out" variables as soon as the
class shows up
	so that there are no issues with calling those functions. Setting th ein
stance variable offset to 8 so we don't
	give offsets based on the last class.
	*/
	@Override
	public void enterClass_(Class_Context ctx) {
		//Perform semantic processing on instance variable and method de
```

```
clarations (these symbols go in the new scope)
                if (ctx.INHERITS() != null) {
                        print.err(String.format(print.errMsgs.get("Unsupported"),
                                        "Inheritance"),ctx);
                }
                //instance variable offset set to 8;
                instanceVarOffset = 8;
                //keeping track of current class
                currentClass = ctx.IDENTIFIER(0).getText();
                //Create a new type for the class
                ClassDeclaration myClass = new ClassDeclaration(ctx.IDENTIFIER(0
).getText());
                Type classType = Type.createType(myClass);
                myClass.type = classType;
                if (currentClass.equals("Reader")) {
                        symTable.push("in", new VarDeclaration(classType, "in"));

                }
                if (currentClass.equals("Writer")) {
                        symTable.push("out", new VarDeclaration(classType, "out")
);
                }
                //Add the class name as a symbol in the current scope
                symTable.push(ctx.IDENTIFIER(0).getText(), myClass);
                //SymbolTable.BeginScope( )
                symTable.beginScope();
                super.enterClass_(ctx);
        }

        /*
        Function Name: exitClass_
        Description:Ending the scope to clear off all the methods/instance param
eters, because we're done processing the class.
        */
        @Override
        public void exitClass_(Class_Context ctx) {

                ClassDeclaration myClass = (ClassDeclaration)symTable.lookup(ctx
.IDENTIFIER(0).getText()).getDecl();
                //METHODS
                for (FloydParser.Method_declContext func : ctx.method_decl()) {
                        MethodDeclaration  classMethod = (MethodDeclaration)symT
able.lookup(func.IDENTIFIER(0).getText()).getDecl();
                        if (myClass != null) {
                        myClass.appendMethod(func.IDENTIFIER(0).getText(), class
Method);
                        }
                        else {
                                print.err("Class doesn't seem to be in the symbol table",ctx);
                        }
                }

                //SymbolTable.EndScope( ) to remove the instance variable and me
thod declarations from the symbol table
                symTable.endScope();
                //instanceVarOffset set to 8
                instanceVarOffset = 8;
                //Update the class declaration info in the symbol table

                super.exitClass_(ctx);
        }
        /*
        Function Name: exitMethodDot_Exp
        Description:If there's a variable on the LHS of the dot, then the symbol
 gets passed along. Otherwise just the function type.

        */
        @Override
        public void exitMethodDot_Exp(MethodDot_ExpContext ctx) {
```



```
                Symbol sym = symTable.lookup(ctx.e1.getText());
                if (sym != null) {
                        ctx.sym = sym;
                }

                ctx.myType = ctx.e2.myType;

                super.exitMethodDot_Exp(ctx);
        }

}
```

```
/*
Name: Italo Moraes (IMORA128)
Class: CpS 450
Filename: CodeGen.java
Description: Contains CodeGen class that generates all the code for the program
*/
package cps450;
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;
import java.util.Stack;
import org.antlr.v4.runtime.ParserRuleContext;
import java.lang.ProcessBuilder;
import cps450.FloydParser.AddMinus_ExpContext;
import cps450.FloydParser.AddPlus_ExpContext;
import cps450.FloydParser.AndX_ExpContext;
import cps450.FloydParser.Assignment_stmtContext;
import cps450.FloydParser.Call_stmtContext;
import cps450.FloydParser.Class_Context;
import cps450.FloydParser.ExprCont_FalseContext;
import cps450.FloydParser.ExprCont_IDContext;
import cps450.FloydParser.ExprCont_IDExprContext;
import cps450.FloydParser.ExprCont_IntlitContext;
import cps450.FloydParser.ExprCont_MEContext;
import cps450.FloydParser.ExprCont_NewContext;
import cps450.FloydParser.ExprCont_NullContext;
import cps450.FloydParser.ExprCont_StrlitContext;
import cps450.FloydParser.ExprCont_TrueContext;
import cps450.FloydParser.If_stmtContext;
import cps450.FloydParser.Loop_stmtContext;
import cps450.FloydParser.MethodDot_ExpContext;
import cps450.FloydParser.Method_declContext;
import cps450.FloydParser.MultiDIV_ExpContext;
import cps450.FloydParser.MultiTimes_ExpContext;
import cps450.FloydParser.OrX_ExpContext;
import cps450.FloydParser.RelationalEQ_ExpContext;
import cps450.FloydParser.RelationalGE_ExpContext;
import cps450.FloydParser.RelationalGT_ExpContext;
import cps450.FloydParser.StartContext;
import cps450.FloydParser.UnaryMinus_ExpContext;
import cps450.FloydParser.UnaryNot_ExpContext;
import cps450.FloydParser.UnaryPlus_ExpContext;
import cps450.FloydParser.Var_declContext;

public class CodeGen extends FloydBaseVisitor<Void> {
        List<TargetInstruction> instructions = new ArrayList<TargetInstruction>(
);
        Stack<String> registers = new Stack<String>();
        Option opt;
        int labelCounter;
        static int LOCAL_SCOPE = 2;
        MyError PRINT = new MyError(true);
        SymbolTable symTable;

        //pushing the gp registers in case I want to use them
        CodeGen(Option opt, int labelCounterValue) {
                labelCounter = labelCounterValue;
                registers.push("%edx");
                registers.push("%ecx");
                registers.push("%ebx");
                registers.push("%eax");
                symTable = SymbolTable.getInstance();
                this.opt = opt;
        }
    /*
    Function Name: callFunction
    Description: Used to facilitate the function calls needed for all the differ
```



```
ent operator expressions. They usually have
    2 params, so always adding 8 to the stack
    */
        void callFunction(String functionName) {
                emit(new TargetInstruction.Builder().instruction("call").operand1
(functionName).build());
                emit(new TargetInstruction.Builder().instruction("addl").operand1
("$8,").operand2("%esp").build());
                emit(new TargetInstruction.Builder().instruction("pushl").operand
1("%eax").build());
        }
    /*
    Function Name: println
    Parameters:
    Description: Prints a new line. (for comments)
    */
        void println() {
                emit(new TargetInstruction.Builder().directive("\n").build());
        }

    /*
    Function Name: emit
    Parameters: TargetInstruction t
    Description: Adds <t> to the list of instructions
    */
        void emit(TargetInstruction t) {
                instructions.add(t);
        }


    /*
    Function Name: printInstructions
    Parameters:
    Description: For debugging purposes: prints out the instructions
    */
        void printInstructions() {
                for (TargetInstruction i: instructions) {
                        System.out.println(i);
                }
        }

    /*
    Function Name: writeToFile
    Parameters: boolean s
    Description: Creates a file and writes all the instructions to it. If s is t
rue,
    then it stops at creating the .s file. If s is false, it links it with stdli
b and compiles.
    */
        void writeToFile(boolean s) {
                String fileName = opt.fileName.get(0);
                fileName = fileName.substring(0, fileName.lastIndexOf('.'));
                try {
                PrintWriter w = new PrintWriter(fileName + ".s");
                for (TargetInstruction i: instructions) {
                        w.println(i);
                }
                w.close();
                } catch (IOException e) {
                        e.printStackTrace();
                }
                compile(fileName, s);
        }

    /*
    Function Name: compile
    Parameters: String fileName, boolean s
    Description: Creates a processbuilder object and then calls invokeGCC
    */
```

```
        void compile(String fileName, boolean s) {
                //building the object file
                if (!s) {
                ProcessBuilder buildObject = new ProcessBuilder("gcc", "-c", file
Name + ".s");
                invokeGCC(buildObject, fileName + " object file");
                ProcessBuilder buildExecutable = new ProcessBuilder("gcc", fileN
ame + ".o", "stdlib.o", "-o", fileName);
                invokeGCC(buildExecutable, fileName + " executable");
                }
        }

    /*
    Function Name: invokeGCC
    Parameters: ProcessBuilder procBuilder, String jobname
    Description: Invokes GCC using procbuilder and prints out errors if it fails
    */
        void invokeGCC(ProcessBuilder procBuilder, String jobName) {
                int exitCode;
                try {
                        Process proc = procBuilder.start();
                        proc.waitFor();
                        exitCode = proc.exitValue();
                        if (exitCode == 0) {
                        } else {
                                BufferedReader buf = new BufferedReader(new Inp
utStreamReader(proc.getErrorStream()));
                                String out;
                                while ((out = buf.readLine()) != null) {
                                        System.out.println(out);
                                }
                        }
                } catch (Exception e) {
                        System.out.println(String.format("Error while invoking process: %
s", e));
                }

        }
    /*
    Function Name: emitComment
    Parameters: ParserRUleContext ctx
    Description: prints a conmment using the given context, used mostly for debu
gging purposes.
    */
        void emitComment(ParserRuleContext ctx) {
                emit(new TargetInstruction.Builder().comment(String.format("Line
%s: %s",ctx.start.getLine(), ctx.getText())).build());
                //===============DEBUGGING============================
                //.stabn 68,0,%s,.line%s-main
                //.line%s:
                if (opt.g) {
                emit(new TargetInstruction.Builder().directive(String.format(".st
abn 68,0,%s,.line%s-main",
                                ctx.start.getLine(), ctx.start.getLine())).build
());
                emit(new TargetInstruction.Builder().directive(String.format(".li
ne%s:",
                                ctx.start.getLine())).build());
                }
                //===============DEBUGGING============================
        }

    /*
    Function Name: nullPointerCheck
    Description: Calls a C function that checks for null pointers. Prints an err
or message if the pointer is null, then exits.
    */
        void nullPointerCheck(ParserRuleContext ctx) {
                emit(new TargetInstruction.Builder().comment("Checking if the object abo
ve is null").build());
```



```
                emit(new TargetInstruction.Builder().comment("pushing line number").b
uild());
                emit(new TargetInstruction.Builder().instruction(String.format("
pushl $%s", ctx.start.getLine())).build());
                emit(new TargetInstruction.Builder().instruction("call nullpointertest"
).build());
                emit(new TargetInstruction.Builder().comment("Removing the line number
from the top of the stack, leaving the obj reference there").build());
                emit(new TargetInstruction.Builder().instruction("addl $4, %esp").b
uild());
        }


    /*
    Function Name: visitExprCont_Intlit
    Description: Pushes its integer value onto the stack
    */
        @Override
        public Void visitExprCont_Intlit(ExprCont_IntlitContext ctx) {
                TargetInstruction foo = new TargetInstruction.Builder().instruct
ion("pushl").operand1(String.format("$%s", ctx.INTEGER_LITERAL().getText())).buil
d();
                emit(foo);
                return null;
        }

    /*
    Function Name: visitExprCont_Intlit
    Description: Pushes the offset of the location relative to the base pointer
depending on whether it's local or instance.
    In and out are a special case (globals) so it simply pushes their variable n
ame.
    Methods are a special case, since they're used as a return in Floyd. So I pu
sh the offset
    to the return value space inside the stack.
    */
        @Override
        public Void visitExprCont_ID(ExprCont_IDContext ctx) {
                String name = ctx.IDENTIFIER().getText();
                Symbol sym = ctx.sym;
                if (sym.getDecl() instanceof VarDeclaration) {
                        if (sym.getScope() == LOCAL_SCOPE) {
                                VarDeclaration variable = (VarDeclaration)sym.ge
tDecl();
                                int offset = variable.getOffset();
                                //printing comment to explain which variable is
being pushed
                                emit(new TargetInstruction.Builder().comment(Str
ing.format("pushl %s", sym.getName())).build());
                                emit(new TargetInstruction.Builder().instruction
("pushl").operand1(String.format("%s(%%ebp)", offset)).build());
                        } else {
                                VarDeclaration lhs = (VarDeclaration)sym.getDecl
();
                                if (name.equals("in")) {
                                        emit(new TargetInstruction.Builder().ins
truction("pushl _in").build());
                                }
                                else if (name.equals("out")) {
                                        emit(new TargetInstruction.Builder().ins
truction("pushl _out").build());
                                } else {
                                emit(new TargetInstruction.Builder().comment("ge
t reference to me").build());
                                emit(new TargetInstruction.Builder().instruction
("movl 8(%ebp), %ebx").build());
                                emit(new TargetInstruction.Builder().comment("pu
sh value inside of the reference").build());
                                emit(new TargetInstruction.Builder().instruction
(String.format("pushl %s(%%ebx)", lhs.getOffset())).build());
```

```
                        }

                    }
                } else {
                    //All method offsets are linked to the return value area
 of memory in the stack
                    MethodDeclaration variable = (MethodDeclaration)sym.getD
ecl();
                    int offset = variable.getOffset();
                    emit(new TargetInstruction.Builder().comment(String.form
at("pushl %s", sym.getName())).build());
                    emit(new TargetInstruction.Builder().instruction("pushl")
.operand1(String.format("%s(%%ebp)", offset)).build());
                }

                return null;
        }

    /*
    Function Name: visitExprCont_True
    Description: True is 1 in floyd
    */
        @Override
        public Void visitExprCont_True(ExprCont_TrueContext ctx) {

                TargetInstruction instruction = new TargetInstruction.Builder().
instruction("pushl").operand1("$1").build();
                emit(instruction);
                return null;
        }

        /*
    Function Name: visitExprCont_False
    Description: false is 0 in floyd
    */
        @Override
        public Void visitExprCont_False(ExprCont_FalseContext ctx) {
                TargetInstruction instruction = new TargetInstruction.Builder().
instruction("pushl").operand1("$0").build();
                emit(instruction);
                return null;
        }

        /*
    Function Name: visitVar_decl
    Description: Used to set instance variables to 0 at declaration
    */
        @Override
        public Void visitVar_decl(Var_declContext ctx) {
                //checking if it's a class obj, if so, need to go ahead and make
 it null (which is 0)
                VarDeclaration newVar = (VarDeclaration)ctx.sym.getDecl();
                if (Type.getTypeForName(ctx.ty.getText()) != null) {
                        //initializing variable to 0 (null) at declaration
                        emit(new TargetInstruction.Builder().instruction(String.
format("movl $0, %s(%%ebp)", newVar.getOffset())).build());

                }
                return null;
        }

        /*
    Function Name: visitAssignment_stmt
    Description: Pops the value into the memory location offset that's given in
the symbol that was decorated
    onto the node at semantic checking.
    */
        @Override
        public Void visitAssignment_stmt(Assignment_stmtContext ctx) {
                emit(new TargetInstruction.Builder().comment(String.format("Line
```



```
%s: %s",ctx.start.getLine(), ctx.getText())).build());
                visit(ctx.e1);
                Symbol sym = ctx.sym;
                //check if its a variable
                if (sym.getDecl() instanceof VarDeclaration) {
                        VarDeclaration lhs = (VarDeclaration)sym.getDecl();
                        //local variables
                        if (sym.getScope() == LOCAL_SCOPE) {
                                emit(new TargetInstruction.Builder().comment(Str
ing.format("popl %s", sym.getName())).build());
                                emit(new TargetInstruction.Builder().instruction
(String.format("popl %s(%%ebp)", lhs.getOffset())).build());
                        } else {
                                //instance variables
                                emit(new TargetInstruction.Builder().comment("pu
t param value into eax").build());
                                emit(new TargetInstruction.Builder().instruction
("popl %eax").build());

                                emit(new TargetInstruction.Builder().comment("ge
t reference to me").build());
                                emit(new TargetInstruction.Builder().instruction
("movl 8(%ebp), %ebx").build());
                                emit(new TargetInstruction.Builder().comment("sto
re new value in offset inside of me").build());
                                emit(new TargetInstruction.Builder().instruction
(String.format("movl %%eax, %s(%%ebx)", lhs.getOffset())).build());

                        }
                        //check if its a function
                } else if (sym.getDecl() instanceof MethodDeclaration) {
                        //i set the function offset at -4 when it is created, si
nce that's where in the stack the ret value goes
                        MethodDeclaration lhs = (MethodDeclaration) sym.getDecl(
);
                        emit(new TargetInstruction.Builder().comment(String.form
at("popl %s", sym.getName())).build());
                        emit(new TargetInstruction.Builder().instruction(String.
format("popl %s(%%ebp)", lhs.getOffset())).build());
                }


                println();
                return null;
        }


        /*
    Function Name: visitAddPlus_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitAddPlus_Exp(AddPlus_ExpContext ctx) {
                visit(ctx.e1);
                visit(ctx.e2);
                String sourceReg = registers.pop();
                String destReg = registers.pop();
                TargetInstruction pop1 = new TargetInstruction.Builder().
                                instruction(String.format("popl %s", sourceReg)).
build();
                TargetInstruction pop2 = new TargetInstruction.Builder().
                                instruction(String.format("popl %s", destReg)).bu
ild();
                emit(pop1);
                emit(pop2);
                TargetInstruction add = new TargetInstruction.Builder().
                                instruction("addl ").operand1(String.format("%s,",
```

```
 sourceReg)).operand2(destReg).build();
                emit(add);
                TargetInstruction pushResult = new TargetInstruction.Builder().
                                instruction(String.format("pushl %s", destReg)).b
uild();
                emit(pushResult);
                registers.push(destReg);
                registers.push(sourceReg);
                return null;
        }

        /*
    Function Name: visitAddMinus_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitAddMinus_Exp(AddMinus_ExpContext ctx) {
                visit(ctx.e2);
                visit(ctx.e1);
                callFunction("minus");
                return null;
        }


        /*
    Function Name: visitMultiTimes_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitMultiTimes_Exp(MultiTimes_ExpContext ctx) {
                visit(ctx.e1);
                visit(ctx.e2);
                callFunction("times");
                return null;
        }

        /*
    Function Name: visitMultiDIV_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitMultiDIV_Exp(MultiDIV_ExpContext ctx) {
                visit(ctx.e2);
                visit(ctx.e1);
                callFunction("division");
                return null;
        }


        /*
    Function Name: visitRelationalGT_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitRelationalGT_Exp(RelationalGT_ExpContext ctx) {
                visit(ctx.e2);
                visit(ctx.e1);
                callFunction("greaterThan");
```



```
                return null;
        }


        /*
    Function Name: visitUnaryMinus_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitUnaryMinus_Exp(UnaryMinus_ExpContext ctx) {
                visit(ctx.e1);
                //needs only $4 because unary only uses 1 argument
                emit(new TargetInstruction.Builder().instruction("call").operand1
("unaryMinus").build());
                emit(new TargetInstruction.Builder().instruction("addl").operand1
("$4,").operand2("%esp").build());
                emit(new TargetInstruction.Builder().instruction("pushl").operand
1("%eax").build());
                return null;
        }

        /*
    Function Name: visitRelationalGE_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitRelationalGE_Exp(RelationalGE_ExpContext ctx) {
                visit(ctx.e2);
                visit(ctx.e1);
                callFunction("greaterEqual");
                return null;
        }
        /*
    Function Name: visitRelationalEQ_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitRelationalEQ_Exp(RelationalEQ_ExpContext ctx) {
                visit(ctx.e2);
                visit(ctx.e1);
                callFunction("eqTo");
                return null;
        }


        /*
    Function Name: visitAndX_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitAndX_Exp(AndX_ExpContext ctx) {
                visit(ctx.e2);
                visit(ctx.e1);
                callFunction("andOp");
                return null;
        }
```

```
        /*
    Function Name: visitOrX_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitOrX_Exp(OrX_ExpContext ctx) {
                visit(ctx.e2);
                visit(ctx.e1);
                callFunction("orOp");
                return null;
        }

        /*
    Function Name: visitUnaryPlus_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitUnaryPlus_Exp(UnaryPlus_ExpContext ctx) {
                //needs only $4 because unary only uses 1 argument
                visit(ctx.e1);
                emit(new TargetInstruction.Builder().instruction("call").operand1
("unaryPlus").build());
                emit(new TargetInstruction.Builder().instruction("addl").operand1
("$4,").operand2("%esp").build());
                emit(new TargetInstruction.Builder().instruction("pushl").operand
1("%eax").build());
                return null;
        }

        /*
    Function Name: visitUnaryNot_Exp
    Description: Visits the operands so that it pushes them onto the stack, call
s the
    function and cleans up the stack by adding to it the number of parameters *
4
    */
        @Override
        public Void visitUnaryNot_Exp(UnaryNot_ExpContext ctx) {
                visit(ctx.e1);
                //needs only $4 because unary only uses 1 argument
                emit(new TargetInstruction.Builder().instruction("call").operand1
("unaryNot").build());
                emit(new TargetInstruction.Builder().instruction("addl").operand1
("$4,").operand2("%esp").build());
                emit(new TargetInstruction.Builder().instruction("pushl").operand
1("%eax").build());
                return null;
        }

        /*
    Function Name: visitStart
    Description: Very important. This is where I set up the program and visit al
l the classes. This spot also generates
    code from a dummy main function into the start function of the last defined
class.
    */
        @Override
        public Void visitStart(StartContext ctx) {
                //necessary for the program to run
                if (opt.fileName.get(0).equals("stdlib.floyd")) {
                        for (int i = 0; i < ctx.class_().size(); i++) {
                                visit(ctx.class_(i));
                        }
```



```
                        opt.labelCounter = labelCounter;
                        return null;
                }
                emit(new TargetInstruction.Builder().label(String.format(".global
%s", "main")).build());
                emit(new TargetInstruction.Builder().directive(String.format(".fil
e \"%s\"", opt.fileName.get(0))).build());
                /*
                 * need to create reader/writer objects at the beginning of the
program otherwise it'll fail the null pointer
                  * when calling in/out
                 */
                //making them globals
                emit(new TargetInstruction.Builder().comment("Making in & out globals"
).build());
                emit(new TargetInstruction.Builder().directive(".comm\t _in,4,4").bu
ild());
                emit(new TargetInstruction.Builder().directive(".comm\t _out,4,4").b
uild());


                //going to use this to make sure the class defined class has def
ined a start method
                Class_Context lastClass = ctx.class_().get(ctx.class_().size()-1
);
                boolean startDefined = false;
                //checking all the methods of the last defined class. If it has
start, we're good
                for (int i = 0; i < lastClass.method_decl().size(); i++) {
                        if (lastClass.method_decl().get(i).IDENTIFIER(0).getText
().equals("start")) {
                                startDefined = true;
                        }
                }
                if (startDefined) {
                        //Creating a main method to instantiate the main clas ob
ject and then call the start function
                        int instanceVars = lastClass.var_decl().size();
                        emit(new TargetInstruction.Builder().comment("Main method.
Creates obj instance of the last class and calls its start method").build());
                        emit(new TargetInstruction.Builder().label(String.format
("main:")).build());

                        //instantiating in & out objects
                        //creating their objects. 8 bytes b/c they don't have an
y instance vars
                        //in obj
                        emit(new TargetInstruction.Builder().comment("instantiating _i
n object").build());
                        emit(new TargetInstruction.Builder().instruction(String.
format("pushl $%s", 8)).build());
                        emit(new TargetInstruction.Builder().instruction("pushl $1
").build());
                        emit(new TargetInstruction.Builder().instruction("call").
operand1("calloc").build());
                        emit(new TargetInstruction.Builder().instruction("addl").
operand1("$8,").operand2("%esp").build());
                        emit(new TargetInstruction.Builder().instruction("movl %e
ax, _in").build());
                        //out obj
                        emit(new TargetInstruction.Builder().comment("instantiating _
out object").build());
                        emit(new TargetInstruction.Builder().instruction(String.
format("pushl $%s", 8)).build());
                        emit(new TargetInstruction.Builder().instruction("pushl $1
").build());
                        emit(new TargetInstruction.Builder().instruction("call").
operand1("calloc").build());
                        emit(new TargetInstruction.Builder().instruction("addl").
operand1("$8,").operand2("%esp").build());
```

```
                        emit(new TargetInstruction.Builder().instruction("movl %e
ax, _out").build());

                        //Need to create the last class object and call its star
t method
                        emit(new TargetInstruction.Builder().comment("Creating last c
lass object & calling its start method").build());
                        //Number of instance vars * 4 (because each var is 4 byt
es) + 8 (8 extra bytes for every object instanc)
                        emit(new TargetInstruction.Builder().instruction(String.
format("pushl $%s", (instanceVars * 4) + 8)).build());
                        emit(new TargetInstruction.Builder().instruction("pushl $1
").build());
                        emit(new TargetInstruction.Builder().instruction("call").
operand1("calloc").build());
                        emit(new TargetInstruction.Builder().instruction("addl").
operand1("$8,").operand2("%esp").build());
                        //setting instance vars to 0
                        if (instanceVars > 0) {
                                emit(new TargetInstruction.Builder().comment(Str
ing.format("Initializing %s instance vars to 0", instanceVars)).build());
                                //put 0 in each memory offset
                                for (int i = 8; i < (instanceVars * 4) + 8; i +=
 4 ) {
                                        emit(new TargetInstruction.Builder().ins
truction(String.format("movl $0, %s(%%eax)", i)).build());
                                }
                        }

                        //pushing pointer to the class obj onto the stack
                        emit(new TargetInstruction.Builder().instruction("pushl %
eax").build());
                        //calling the start method
                        String startName = String.format("%s_%s", lastClass.IDEN
TIFIER(0).getText(), "start");
                        //calling the start method
                        emit(new TargetInstruction.Builder().instruction(String.
format("call %s", startName)).build());
                        //when we return from the start method, the program shou
ld exit, so that code goes here
                        emit(new TargetInstruction.Builder().comment("Calling exit be
cause the program is finished").build());
                        emit(new TargetInstruction.Builder().instruction("pushl")
.operand1("$0").build());
                        emit(new TargetInstruction.Builder().instruction("call").
operand1("exit").build());
                        //visiting all the classes
                        for (int i = 0; i < ctx.class_().size(); i++) {
                                visit(ctx.class_(i));
                        }

                } else {
                        String ansi_reset = "\u001B[0m";
                        String ansi_red = "\u001B[31m";
                        System.out.println(String.format("%sThe start method was not defi
ned in the last class %s. Exiting.%s",ansi_red, lastClass.IDENTIFIER(0).getText(), ansi_rese
t));
                        System.exit(1);
                }

                return null;
        }

        /*
    Function Name: visitCLass
    Description: Visits all the methods and variable declarations to generate th
eir code.
    */
        @Override
        public Void visitClass_(Class_Context ctx) {
```



```
                println();
                emit(new TargetInstruction.Builder().comment(String.format("****
**** Class Definition: %s ******** ", ctx.IDENTIFIER(0).getText())).build());

                for (int i = 0; i < ctx.var_decl().size(); i++) {
                        visit(ctx.var_decl(i));
                }
                for(int i = 0; i < ctx.method_decl().size(); i++) {
                        visit(ctx.method_decl(i));
                }

                return null;
        }


        /*
    Function Name: visitMethod_decl
    Description: Prints out a function preamble (so we don't mess up the stack),
 makes space for the return value
    and local variables.
    */
        @Override
        public Void visitMethod_decl(Method_declContext ctx) {
                emit(new TargetInstruction.Builder().comment(String.format("Line
%s: %s() %s", ctx.start.getLine(),
                                ctx.IDENTIFIER(0).getText(), ctx.IS().getText())
).build());
                String funcName = String.format("%s_%s", ctx.className, ctx.IDEN
TIFIER(0).getText());
                emit(new TargetInstruction.Builder().label(String.format(("%s:")
, funcName)).build());
                //}
                //FUNCTION PREAMBLE
                emit(new TargetInstruction.Builder().comment("Function preamble").bu
ild());
                emit(new TargetInstruction.Builder().instruction("pushl %ebp").bui
ld());
                emit(new TargetInstruction.Builder().instruction("movl %esp, %ebp"
).build());
                //return value
                emit(new TargetInstruction.Builder().comment("Making space for return v
alue").build());
                emit(new TargetInstruction.Builder().instruction("pushl $0").build
());
                //locals
                for (int i =  0; i < ctx.params; i++) {
                                emit(new TargetInstruction.Builder().comment(String.form
at("making space for %s local", ctx.params)).build());
                                emit(new TargetInstruction.Builder().instruction("pushl $0
").build());
                }
                //folowed by visiting the statement list to print the instructio
ns for the content of the function
                visit(ctx.statement_list());

                //at the end of the function, put the value inside the return va
lue area into eax
                if (ctx.typ != null) {
                emit(new TargetInstruction.Builder().comment("Moving the value inside th
e return value section of the stack into eax").build());
                emit(new TargetInstruction.Builder().instruction(("movl -4(%ebp), %
eax")).build());
                }

                //cleaning up the stack
                emit(new TargetInstruction.Builder().comment("cleaning up the stack and r
eturnig").build());
                emit(new TargetInstruction.Builder().instruction("leave").build()
);
                emit(new TargetInstruction.Builder().instruction(("ret")).build()
```

```
);
                emit(new TargetInstruction.Builder().comment(String.format("Line
%s: %s", ctx.stop.getLine(), "end " + ctx.IDENTIFIER(0).getText())).build());
                return null;
        }

        /*
    Function Name: visitCall_stmt
    Description: Generates code for function call statements. Makes sure to use
right to left calling parameter passing convention.
    */
        @Override
        public Void visitCall_stmt(Call_stmtContext ctx) {
                emit(new TargetInstruction.Builder().comment(String.format("Line
%s: %s",ctx.start.getLine(), ctx.getText())).build());

                //right to left
                int paramNum = 0;
                if (ctx.t2 != null) {
                        paramNum = ctx.expression_list().expression().size();
                        for (int i = ctx.expression_list().expression().size() -
 1; i > -1; i--) {
                                visit(ctx.expression_list().expression().get(i))
;
                        }
                }

                //offset for LHS to pass in "this"
                //only if sym is not null, meaning there's an object there
                if (ctx.t1 != null) {
                        if (ctx.t1.getText().equals("out")) {
                                emit(new TargetInstruction.Builder().instruction
("pushl _out").build());
                                //need to check if its null
                                nullPointerCheck(ctx);
                        } else if (ctx.t1.getText().equals("in")) {
                                emit(new TargetInstruction.Builder().instruction
("pushl _in").build());
                                //need to check if its null
                                nullPointerCheck(ctx);
                        } else {
                                VarDeclaration test = (VarDeclaration)ctx.sym.ge
tDecl();
                                //pushing "this"
                                emit(new TargetInstruction.Builder().comment("ref
erence to the object").build());
                                emit(new TargetInstruction.Builder().comment(Str
ing.format("pushl %s", test.name)).build());
                                emit(new TargetInstruction.Builder().instruction
(String.format("pushl %s(%%ebp)", test.getOffset())).build());
                                //need to check if its null
                                nullPointerCheck(ctx);
                        }
                } else {
                        emit(new TargetInstruction.Builder().comment("reference to th
e object (this)").build());
                        emit(new TargetInstruction.Builder().instruction("pushl 8(
%ebp)").build());
                        //need to check if its null
                        nullPointerCheck(ctx);
                }
                //function name will be appended to the class name
                String functionName = "FunctionNameFailed";
                if (ctx.t1 != null ) {

                        functionName = String.format("%s_%s", ctx.t1.myType.name
, ctx.IDENTIFIER().getText());
                } else {
                        functionName = String.format("%s_%s", ctx.className, ctx
.IDENTIFIER().getText());
```



```
                        //print location
                }
                emit(new TargetInstruction.Builder().instruction("call").operand1
(functionName).build());
                if (paramNum > 0) {
                        emit(new TargetInstruction.Builder().comment(String.form
at("Clean up parameters: %s * 4", paramNum)).build());
                emit(new TargetInstruction.Builder().instruction("addl").operand1
(String.format("$%s,", paramNum * 4)).operand2("%esp").build());
                }
                //if func doesnt have t1, it has no obj that we need to pass "me
" for
                if(ctx.t1 != null) {
                emit(new TargetInstruction.Builder().comment("Clean up this reference pu
shed on last: 4").build());
                emit(new TargetInstruction.Builder().instruction("addl").operand1
(String.format("$%s,",4)).operand2("%esp").build());
                }
                println();

                return null;
        }



        /*
    Function Name: visitExprCont_IDExpr
    Description: Generates code for expression function calls
    */
        @Override
        public Void visitExprCont_IDExpr(ExprCont_IDExprContext ctx) {
                int paramNum = 0;
                //C style parameter passing convention
                if (ctx.expression_list() != null) {
                        paramNum = ctx.expression_list().expression().size();
                        for (int i = ctx.expression_list().expression().size() -
 1; i > -1; i--) {
                                visit(ctx.expression_list().expression().get(i))
;
                        }
                }
                //appends function name to the class name.
                String functionName = String.format("%s_%s", ctx.classType.name,
 ctx.IDENTIFIER().getText());
                emit(new TargetInstruction.Builder().instruction("pushl 8(%ebp)").b
uild());
                //need to check if its null
                nullPointerCheck(ctx);
                emit(new TargetInstruction.Builder().instruction("call").operand1
(functionName).build());
                if (paramNum > 0) {
                emit(new TargetInstruction.Builder().instruction("addl").operand1
(String.format("$%s,", paramNum * 4)).operand2("%esp").build());
                }
                emit(new TargetInstruction.Builder().comment("cleaning up the obj ref").
build());
                emit(new TargetInstruction.Builder().instruction("addl").operand1
(String.format("$%s,", 4)).operand2("%esp").build());
                emit(new TargetInstruction.Builder().instruction("pushl").operand
1("%eax").build());
                println();
                return null;
        }

        /*
    Function Name: visitMethodDot_Exp
    Description: Generates code for expression function calls of the form: type/
object.function
    */
        @Override
```

```
        public Void visitMethodDot_Exp(MethodDot_ExpContext ctx) {
                int paramNum = 0;
                ExprCont_IDExprContext foo = null;
                //it SHOULD be exprcont_IDExprcontext if it ever reaches here...
.
                if (ctx.expr_cont() instanceof ExprCont_IDExprContext) {
                        foo = (ExprCont_IDExprContext) ctx.expr_cont();
                        //parameter pushing right to left
                        if (foo.expression_list() != null) {
                                paramNum = foo.expression_list().expression().si
ze();
                                for (int i = foo.expression_list().expression().
size() - 1; i > -1; i--) {
                                        visit(foo.expression_list().expression()
.get(i));

                                }
                        }

                        visit(ctx.e1);
                        //visit above pushes the obj, i need to check if its nul
l
                        nullPointerCheck(ctx);

                        String functionName = String.format("%s_%s", ctx.e1.myTy
pe, foo.IDENTIFIER().getText());
                        emit(new TargetInstruction.Builder().instruction("call").
operand1(functionName).build());
                        if (paramNum > 0) {
                                emit(new TargetInstruction.Builder().comment(Str
ing.format("Clean up parameters: (%s * 4) + 4 (this ptr)", paramNum)).build());
                                emit(new TargetInstruction.Builder().instruction
("addl").operand1(String.format("$%s,", (paramNum * 4) + 4)).operand2("%esp").buil
d());
                        } else {
                                emit(new TargetInstruction.Builder().comment("Cl
ean up THIS obj reference param").build());
                                emit(new TargetInstruction.Builder().instruction
("addl $4, %esp").build());
                        }

                        emit(new TargetInstruction.Builder().comment("Pushing the re
sult from the called function").build());
                        emit(new TargetInstruction.Builder().instruction("pushl %
eax").build());
                }
                return null;
        }


        /*
    Function Name: visitIf_stmt
    Description: Generates code for if statements.
    */
        @Override
        public Void visitIf_stmt(If_stmtContext ctx) {
                emit(new TargetInstruction.Builder().comment(String.format("Line
%s: %s %s %s",ctx.start.getLine(), ctx.IF().get(0).getText(),ctx.cond_expr.getText
(),
                                ctx.THEN().getText())).build());
                visit(ctx.cond_expr);
                labelCounter = labelCounter + 2;
                int currentIf = labelCounter;
                //the result of the expression should be on top of the stack
                //popping it into eax
                emit(new TargetInstruction.Builder().instruction("popl").operand1
("%eax").build());
                //pushing 1 to EDX so I can compare it to the result of the expr
ession and do the logical jumps
                emit(new TargetInstruction.Builder().instruction("movl").operand
```



```
1("$1,").operand2("%edx").build());
                //comparison
                emit(new TargetInstruction.Builder().instruction("cmpl").operand
1("%eax,").operand2("%edx").build());
                //+1 is false, + 2 is true
                //-2 is false, -1 is true
                //jump to +1 if false
                emit(new TargetInstruction.Builder().instruction("jne").operand1
(String.format(".L%s", (currentIf - 1))).build());
                //emit tru stmt list
                visit(ctx.truestm);
                //emit jmp to +2
                emit(new TargetInstruction.Builder().instruction("jmp").operand1
(String.format(".L%s", currentIf)).build());
                //emit +1 label and then false stmtlist
                emit(new TargetInstruction.Builder().directive(String.format(".L
%s:", (currentIf - 1))).build());
                if (ctx.falsestm != null) {
                        emit(new TargetInstruction.Builder().comment(String.form
at("Line %s: %s", (ctx.falsestm.start.getLine() - 1), "Else")).build());
                        visit(ctx.falsestm);
                }
                //emit +2 label
                emit(new TargetInstruction.Builder().directive(String.format(".L
%s:", currentIf)).build());
                emit(new TargetInstruction.Builder().comment(String.format("Line
%s: %s %s",ctx.stop.getLine(), ctx.END().getText(),
                                ctx.IF(0).getText())).build());


                return null;
        }


        /*
    Function Name: visitLoop_stmt
    Description: Generates code for while statements.
    */
        @Override
        public Void visitLoop_stmt(Loop_stmtContext ctx) {
                labelCounter = labelCounter + 2;
                int currentWhile = labelCounter;
                emit(new TargetInstruction.Builder().instruction("jmp").operand1
(String.format(".L%s", (currentWhile - 1))).build());
                emit(new TargetInstruction.Builder().directive(String.format(".L
%s:", currentWhile)).build());
                visit(ctx.loop_body);
                emit(new TargetInstruction.Builder().directive(String.format(".L
%s:", (currentWhile - 1))).build());
                visit(ctx.exp);
                emit(new TargetInstruction.Builder().instruction("pop").operand1
("%eax").build());
                emit(new TargetInstruction.Builder().instruction("cmpl").operand
1("$0,").operand2("%eax").build());
                emit(new TargetInstruction.Builder().instruction("jne").operand1
(String.format(".L%s", (currentWhile))).build());
                emit(new TargetInstruction.Builder().comment(String.format("Line
%s: %s", ctx.stop.getLine(), "end loop")).build());

                return null;
        }

        /*
    Function Name: visitExprCont_New
    Description: Generates code for new expression. Calls calloc and puts the re
turn value (after initializing
    the instance variables to 0) into the LHS variable.
    */
        @Override
        public Void visitExprCont_New(ExprCont_NewContext ctx) {
```

```
                //Allocate memory from the heap to hold the instance variables f
or Point
                //NumberOfParameters*4 + 8 reserved bytes
                int reserveBytes = (ctx.paramNum * 4) + 8;
                emit(new TargetInstruction.Builder().instruction(String.format("
pushl $%s", reserveBytes)).build());
                emit(new TargetInstruction.Builder().instruction("pushl $1").build
());
                emit(new TargetInstruction.Builder().instruction("call").operand1
("calloc").build());
                emit(new TargetInstruction.Builder().instruction("addl").operand1
("$8,").operand2("%esp").build());


                //Initialize the values of the instance variables to 0
                if (ctx.paramNum > 0) {
                        emit(new TargetInstruction.Builder().comment(String.form
at("Initializing %s instance vars to 0", ctx.paramNum)).build());
                        //put 0 in each memory offset
                        for (int i = 8; i < (ctx.paramNum * 4) + 8; i += 4 ) {
                                emit(new TargetInstruction.Builder().instruction
(String.format("movl $0, %s(%%eax)", i)).build());
                        }
                }


                //Leave a reference to the memory on the top of the stack
                emit(new TargetInstruction.Builder().instruction("pushl").operand
1("%eax").build());

                return null;
        }

        /*
    Function Name: visitExprCont_Null
    Description: Null is 0 in floyd, so pushing onto the stack.
    */
        @Override
        public Void visitExprCont_Null(ExprCont_NullContext ctx) {
                //null is 0
                emit(new TargetInstruction.Builder().instruction("pushl $0").build
());
                return null;
        }
        /*
    Function Name: visitExprCont_ME
    Description: Me will always be the "this" section of the stack, so offset 8
from BP.
    */
        @Override
        public Void visitExprCont_ME(ExprCont_MEContext ctx) {
                emit(new TargetInstruction.Builder().instruction("pushl 8(%ebp)").b
uild());
                return null;
        }

        /*
    Function Name: visitExprCont_Strlit
    Description: String literals will always create a label for the string and c
all string_fromlit
    which is a C function in the stdlib.c file.
    */
        @Override
        public Void visitExprCont_Strlit(ExprCont_StrlitContext ctx) {
                //I stuck the string label counter in the SymbolTable class beca
use it's a singleton
                //and I'd be running through CodeGen twice.
                String stringLitLabel = String.format("stringlit%s", symTable.strin
gLabelCounter);
                emit(new TargetInstruction.Builder().label(".data").build());
```



```
                emit(new TargetInstruction.Builder().label(stringLitLabel + ":")
.build());
                emit(new TargetInstruction.Builder().directive(String.format(".st
ring %s", ctx.getText())).build());
                println();
                emit(new TargetInstruction.Builder().label(".text").build());
                emit(new TargetInstruction.Builder().instruction(String.format("
pushl $%s", stringLitLabel)).build());
                emit(new TargetInstruction.Builder().instruction("call string_fromlit"
).build());
                emit(new TargetInstruction.Builder().instruction("addl $4, %esp").b
uild());
                emit(new TargetInstruction.Builder().instruction("pushl %eax").bui
ld());
                symTable.stringLabelCounter++;
                return null;
        }



}
```

```
/*
Name: Italo Moraes (IMORA128)
Class: CpS 450
Filename: Floyd.g4
Description: Contains the rules used to generate the scanner
*/
grammar Floyd;


start
: cr? class_ (cr class_)* cr?
;

cr
: CR+
;
class_
: CLASS IDENTIFIER (INHERITS FROM IDENTIFIER)? IS cr var_decl* method_decl* END
IDENTIFIER
;

var_decl returns [Symbol sym]
: IDENTIFIER (COLON ty=type)? (ASSIGNMENT_OPERATOR expression)? cr
;


method_decl returns [int params, String className]
: IDENTIFIER R_PAR ((argument_decl_list)?) L_PAR (COLON typ=type)? IS cr var_dec
l* BEGIN cr statement_list END IDENTIFIER cr
;

argument_decl_list
: (argument_decl SEMICOLON)* argument_decl
;

argument_decl
: IDENTIFIER COLON type
;

type returns [Type myType]
: INT   #TypeInt
| STRING        #TypeString
| BOOLEAN       #TypeBool
| IDENTIFIER #TypeID
| type '[' (expression)? ']' #TypeExpr
;


statement_list
: (statement cr)*
;

statement
: assignment_stmt
| if_stmt
| loop_stmt
| call_stmt
;

assignment_stmt returns [Symbol sym]
: IDENTIFIER ('[' expression ']')* ASSIGNMENT_OPERATOR e1=expression
;

if_stmt
: IF cond_expr=expression THEN cr truestm=statement_list (ELSE cr falsestm=state
ment_list)? END IF
;

loop_stmt
: LOOP WHILE exp=expression cr loop_body=statement_list END LOOP
```



```
;

call_stmt returns [String className, Symbol sym]
: (t1=expression PERIOD)? func=IDENTIFIER R_PAR (t2=expression_list)? L_PAR
;

expression_list
: (expression COMMA)* expression
;

expression returns [Type myType]
: or_exp        #ExprOr_Expr
;

or_exp returns [Type myType]
: e1=or_exp OR e2=and_exp #OrX_Exp
| and_exp       #OrAnd_Exp
;

and_exp returns [Type myType]
: e1=and_exp AND e2=relate_switch_exp #AndX_Exp
| relate_switch_exp #AndConcat_Exp
;

relate_switch_exp returns [Type myType]
: relational_exp
| concat_exp
;

relational_exp returns [Type myType]
: e1=concat_exp GE e2=concat_exp         #RelationalGE_Exp
| e1=concat_exp GT e2=concat_exp         #RelationalGT_Exp
| e1=concat_exp EQ e2=concat_exp         #RelationalEQ_Exp
;

concat_exp returns [Type myType]
: e1=concat_exp AMPERSAND e2=add_exp    #ConcatX_Exp
| add_exp       #ConcatAdd_Exp
;

add_exp returns [Type myType]
: e1=add_exp PLUS e2=multi_exp  #AddPlus_Exp
| e1=add_exp MINUS e2=multi_exp #AddMinus_Exp
| multi_exp     #AddMulti_Exp
;

multi_exp returns [Type myType]
: e1=multi_exp TIMES e2=unary_exp #MultiTimes_Exp
| e1=multi_exp DIV e2=unary_exp #MultiDIV_Exp
| unary_exp     #MultiUnary_Exp
;

unary_exp returns [Type myType]
: PLUS e1=unary_exp     #UnaryPlus_Exp
| MINUS e1=unary_exp    #UnaryMinus_Exp
| NOT e1=unary_exp              #UnaryNot_Exp
| method_exp            #UnaryMethod_Exp
;

method_exp returns [Type myType, Symbol sym]
: e1=method_exp PERIOD e2=expr_cont     #MethodDot_Exp
| expr_cont                                                     #MethodExpr_Cont
;

expr_cont returns [Type myType, Symbol sym, int paramNum, Type classType]
: IDENTIFIER    #ExprCont_ID
| STRING_LITERAL        #ExprCont_Strlit
| INTEGER_LITERAL       #ExprCont_Intlit
| TRUE  #ExprCont_True
| FALSE #ExprCont_False
```

```
| NULL #ExprCont_Null
| ME    #ExprCont_ME
| R_PAR expression L_PAR        #ExprCont_ParExp
| NEW typ=type  #ExprCont_New
| IDENTIFIER '[' expression ']' ('[' expression ']')*
expression PERIOD IDENTIFIER R_PAR (expression_list)? L_PAR #ExprCont_Array
| IDENTIFIER R_PAR (expression_list)? L_PAR #ExprCont_IDExpr
;

BOOLEAN
: 'boolean'
;

AND
: 'and'
;

NOT
: 'not'
;

OR
: 'or'
;

BEGIN
: 'begin'
;

CLASS
: 'class'
;

ELSE
: 'else'
;

END
: 'end'
;

FALSE
: 'false'
;

FROM
: 'from'
;

IF
: 'if'
;

INHERITS
: 'inherits'
;

INT
: 'int'
;

IS
: 'is'
;

LOOP
: 'loop'
;

ME
```



```
: 'me'
;

NEW
: 'new'
;

NULL
: 'null'
;

STRING
: 'string'
;

THEN
: 'then'
;

TRUE
: 'true'
;

WHILE
: 'while'
;

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

CR
:('\r\n'|'\n'|'\r')
;

LINE_EXTENSION
: '_' CR -> skip
;

WS
   : [ \t] + -> skip
   ;

IDENTIFIER
//: ('a'..'z' | 'A'..'Z' |  '_')+ ('0'..'9' | 'a'..'z' | 'A'..'Z' |  '_')*
: ('a'..'z' | 'A'..'Z' |  '_') ('0'..'9' | 'a'..'z' | 'A'..'Z' |  '_')*
;

INTEGER_LITERAL
: '-'? ('0'..'9')+
;

STRING_LITERAL
:'"' ( '\\' ([tnfr"\\] | ('0'.. '7')('0'.. '7')('0'.. '7'))  | ~[\r\n\\"] )* '"'

;

UNTERMINATED_STRING_ERROR
:'"' ('\\' ["\\] | ~["\\\r\n])*
;

ILLEGAL_STRING_ERROR
:'"' ('\\' [tnfr"\\01234567] | ~[\r\n"])* '"'
;

AMPERSAND
: '&'
;

PLUS
```

```
: '+'
;

MINUS
: '-'
;

TIMES
: '*'
;

DIV
: '/'
;

GT
: '>'
;

GE
: '>='
;

EQ
: '='
;

ASSIGNMENT_OPERATOR
: ':='
;

R_PAR
: '('
;

L_PAR
: ')'
;

R_BRACKET
: '['
;

L_BRACKET
: ']'
;

COMMA
: ','
;

SEMICOLON
: ';'
;

COLON
: ':'
;

PERIOD
: '.'
;
UNKNOWN_CHAR
: .
;
```

```c
#include <syscall.h>
#include <stdlib.h>
#include "stdlib.h"

//Name: Writer_io_write
//Desc: writes <ch> to standard output (<out> is the predefined Floyd Writer obj
ect)
void Writer_io_write(void *out, int ch) {
  char c = ch;

  write(1, &c, 1);
}

//Name: Reader_io_read
//Desc: reads a character from stdin and returns it (<in> is the predefined Floy
d Reader object)
int Reader_io_read(void *in) {
  char c;

  read(0, &c, 1);

  return c;
}

// ---------------------------------------------------------------------
// String Management Functions
// ---------------------------------------------------------------------
//Name: string_fromlit
//Constructs and returns an Floyd String using chars in <lit>, which must be nul
l terminated
struct String *string_fromlit(char *lit)
{
  struct String *newstr = (struct String *)calloc(sizeof(struct String), 1);
  struct CharNode *cur = NULL;
  while (*lit) {
    struct CharNode *node = (struct CharNode *)calloc(sizeof(struct CharNode), 1
);
    node->ch = *lit;
    if (cur == NULL) {
      newstr->list = node;
    } else {
      cur->next = node;
    }
    cur = node;
    lit++;
  }
  return newstr;
}

//Name; writeint
//Desc: Prints an integer to standard out, used in nullpointertest
void writeint(int num) {
  char buf[20];
  char result[20] = "0\n";
  char *pos = buf;
  char *writeptr = result;
  int numWritten;

  // Handle negative numbers
  if (num < 0) {
    *writeptr++ = '-';
    num = -num;
  }

  if (num > 0) {

    // Build the number in reverse order
    while (num > 0) {
      *pos++ = (num % 10) + '0';
      num /= 10;
    }
    pos--;

    // Now we need to copy the results into the output buffer, reversed
    while (pos > buf) {
      *writeptr++ = *pos--;
    }
    *writeptr++ = *pos;
    *writeptr++ = 10;
    *writeptr++ = 0;
  } else {
    // number is 0; use default result
    writeptr = result + 3;
  }

  write(1, result, (writeptr - result) - 1);

}

//Name: nullpointertest
//Tests for a null pointer. Prints out an error and the line number if null, the
n exits.
void nullpointertest(int lineno, void* ptr) {
  const char msg[] = "Null pointer exception on line ";
  if (ptr == NULL ) {
    write(1, msg, sizeof(msg)-1);
    //Who am I to waste a perfectly good writeint function? jajaja
    writeint(lineno);
    exit(1);
  }
}



//The functions from here on out are used to faciltate the code genereation proc
ess.
//They are c functions that are called in the different CodeGen functions.
int andOp(int x, int y) {
  return x && y;
}

int orOp(int x, int y) {
  return x || y;
}

int eqTo(int x, int y) {
  return x == y;
}
int greaterEqual(int x, int y) {
  return x >= y;
}
int greaterThan(int x, int y) {
  return x > y;
}
int minus(int x, int y) {
  return x - y;
}

int times(int x, int y) {
  return x * y;
}

int division(int x, int y) {
  return x / y;
}
int unaryMinus(int x) {
  return -x;
}
int unaryPlus(int x) {
```

```
  return +x;
}
int unaryNot(int x) {
  return !x;
}
```

```
// These structs duplicate the layout of the CharNode and String Floyd classes
// in stdlib.floyd, using the runtime organization presented in class. If your
// class organization differs, you will have to adjust these.

struct CharNode {
  int reserved1, reserved2;
  int ch;
  struct CharNode *next;
};

struct String {
  int reserved1, reserved2;
  struct CharNode *list;
};

struct String *string_fromlit(char *);
```