COOL COMPILER USING ANTLR AND LLVM

A Project

Presented to the faculty of the Department of Computer Science

California State University, Sacramento

Submitted in partial satisfaction of
the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

Avni Bharne

FALL
2015

COOL COMPILER USING ANTLR AND LLVM

A Project

by

Avni Bharne

Approved by:

_____,Committee Chair

Dr. Theodore Krovetz

_____, Second Reader

Dr. Anne-Louise Radimsky

_____

Date

Student: Avni Bharne

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

_____, Graduate Coordinator _____

Dr. Jinsong Ouyang Date

Department of Computer Science

Abstract

of

COOL COMPILER USING ANTLR AND LLVM

by

Avni Bharne

Computers are very sophisticated machines and the way we control these machines is through programs that are always written in some programming language. Today, the programs written to control the computers are in a high-level language like C, C++, JAVA, Python and so on; while the machine hardware only understands low-level language like assembly language or binary. A compiler is the tool that translates programs in a high-level language into programs in low-level language.

The main goal of this project is to explore the tools ANTLR (ANother Tool for Language Recognition) and LLVM (Low Level Virtual Machine) together for compiler construction. This compiler for COOL (Classroom Object Oriented Language) uses ANTLR version 4 for modules responsible for tokenizing (lexer), checking syntax (parser) and checking semantics (semantic analyzer). The code generation module is implemented using LLVM version 3.7.0.

This project demonstrates how the ANTLR 4 features help simplify compiler construction by auto generating methods and providing interfaces for any user defined grammar.

_____, Committee Chair

Dr. Theodore Krovetz

_____

Date

TABLE OF CONTENTS

Page

LIST OF FIGURES

INTRODUCTION

A compiler is a tool that translates a program written in source language into a program in target language without changing the meaning of the program. This process of translation is known as compiling. The source language is generally a high level language like C, C++, Java, Python which is compiled into a low level language like assembly or machine code that can be directly executed by a processor. The compiling process is broadly divided into four phases, namely Lexical Analyzer, Parser, Semantic Analyzer and Code generator.

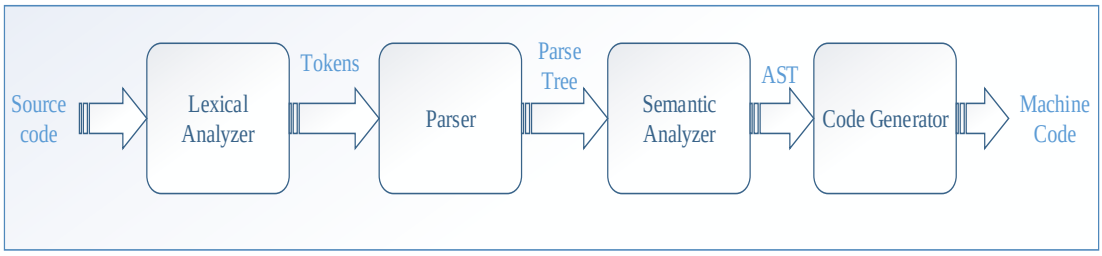

Figure 1: Compiler Overview

In early days, compiler construction was considered to be time-consuming and enervating process. With the advance in computer technology, tools were developed to help make compiler construction less stressful. Early utilities were invented around 1975 and some are still being used today, such as LEX and YACC that help in building a lexical analyzer and a parser respectively. Since then, there has been a lot of improvement in such tools and utilities.

This project analyzes two such tools, namely, ANTLR version 4 (ANother Tool for Language Recognition) and LLVM version 3.7.0 (Low Level Virtual Machine) for compiler construction. The project further implements a compiler using ANTLR and LLVM tools

The motivation for this project is to explore the combination of the tools ANTLR and LLVM for the construction of a compiler for a small language. The project report discusses the inspiration for this

project, followed by an overview of tools being used for the implementation. The rest of the report provides a roadmap with examples for students interested in building their own compiler and to help gain better understanding of the internal workings of a compiler.

The compiler implementation will be attached in the Appendix section as a reference for any interested student.

RELATED WORK

I was introduced to compiler construction during my Graduate Studies course work where I got my hands dirty developing my own compiler for COOL (Classroom Object Oriented Language) using framework and tools, provided and specified by Stanford online course CS 1[2]. This course was delivered by Dr. Alex Aiken, Professor of Computer Science, Stanford University. The course provides a good window into the world of compilation for a newbie. The theory and logic behind the workings of each phase of compiler was coupled with practical implementation of the compiler phase using the framework provided.

The coursework consisted of a detailed study of each phase (viz. Lexical Analyzer, Parser, Semantic Analyzer and Code Generator). The class work encompassed the logic and concepts behind the working of each compiler phase and introduced possibilities about how each phase can be implemented.

This online available course was structured in a way such that on an average a student should be able to complete the course work over the time period of a semester (approximately 16 weeks). To help complete practical assignments in given time frame and also to guide students in the right direction, the course provided a framework which served as a platform, over which the student could build the compiler.

The whole framework is built in a modular fashion isolating each phase from the other. The framework handles the underlying integration of one phase with the other allowing the students to concentrate only on the crux of each phase. The framework also provides a separate test-bench for each phase helping the students to discover and debug any errors. It also enables the student to build a fully functional compiler and confirm this result by verifying with the reference output.

The objective of this project is to analyze the latest available tools (ANTLR 4 and LLVM 3.7.0) for a compiler construction. This analysis is further validated by implementing each phase of the compiler for the Classroom Object Oriented Language (COOL) and testing it against the known comprehensive test-bench provided in the Stanford online course CS1.

TOOLS OVERVIEW

Compiler construction was considered a cumbersome and time-consuming process due to the lack of tools and infrastructure during the 1960s. In 1975 when M. E. Lesk and E. Schmidt came up with utilities LEX and YACC, building compilers got relatively easier.

LEX

LEX is a scanner utility, which was and is still commonly used for implementing lexical analyzers. For using LEX utility, programmer provides rules/patterns for identifying tokens of the language, as the input. The utility then uses these rules/patterns to generate C code that is the lexical analyzer for the language.

YACC (Yet Another Compiler Compiler)

YACC works in a very similar fashion as LEX. It takes in parser rules/patterns as input and generates C code that forms the Parser for the language.

Flex and Bison are free versions of LEX and YACC respectively that can be obtained from GNU.
LEX also has another version namely Jlex which generates lexical analyzer in Java.
For the compiler construction using Stanford framework, the tools used for generating lexical analyzer and parser were LEX and YACC.

ANTLR 4

For this project, I decided to opt for ANTLR 4 framework in lieu of LEX and YACC due to its many advantages. To study the tool and its features I referred to the ANTLR 4 Text Book extensively [1]. Some of its advantages are listed below.

- ANTLR has the ability to generate a parser, specific to the language by using only the language grammar.
- It auto generates a parse tree which is a data structure that represents how the input is matched with the rules and patterns specified in the grammar.
- ANTLR provides multiple command line options that can be used to view the generated parse tree in different forms.
    - option gui, provides graphical representation of the parse tree
    - option tree, prints out the parse tree in lisp form
    - option trace, prints the rule name and current token on rule entry and exit.
- ANTLR also provides mechanisms for parse tree traversals, namely Parse tree listener and Parse tree visitor.

Parse Tree Listener

ANTLR automatically generates a ParseTreeListener (listener) subclass, which is specific to the grammar. This subclass provides entry and exit methods for every rule or sub-rule specified in the grammar. ANTLR also provides a ParseTreeWalker (walker) class, which is responsible for walking the parse tree and triggering ParseTreeListener method calls.

As the walker walks through the parse tree, on encountering each node, it triggers the listener methods. On first encounter, entry method of the node is triggered and exit method is triggered after visiting all children of that node.

An important feature of Parse Tree Listener is that, it is fully automatic. Simply by creating an object of the walker class and providing it the parse tree node context, walker invokes a complete parse tree traversal starting from the node context provided. In this mechanism, the children of every node are implicitly called and therefore tree traversal is completely

automatic. Figure 2 shows a parse tree and Figure 3 shows the corresponding ParseTreeWalker call sequence.

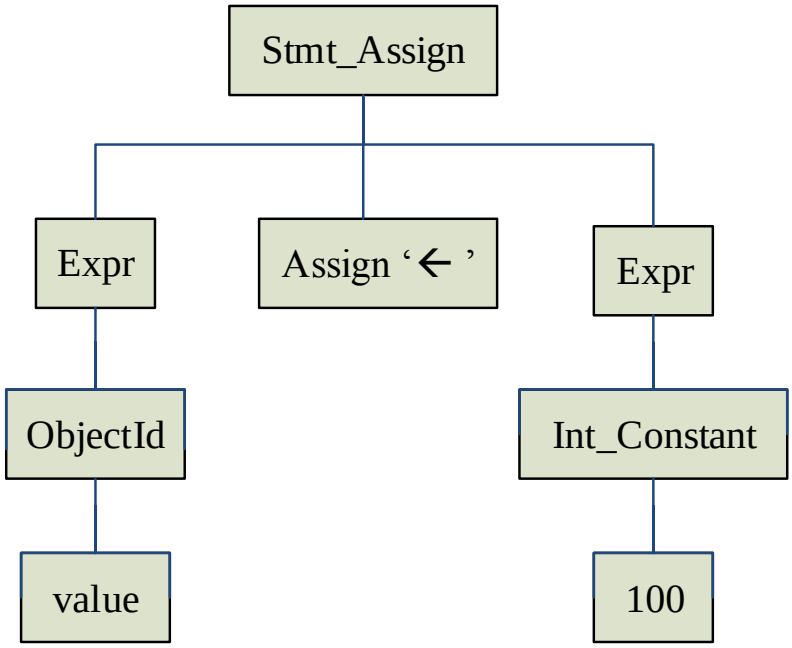

Figure 2: Sample Parse Tree

```
enterStmt_Assign(Stmt_AssignContext)
enterExpr(ExprContext)
enterObjectId(ObjectIdContext)
visitTerminal(TerminalNode)
exitObjectId(ObjectIdContext)
exitExpr(ExprContext)
visitTerminal(TerminalNode)
enterExpr(ExprContext)
enterInt_Constant(Int_ConstantContext)
visitTerminal(TerminalNode)
exitInt_Constant(Int_ConstantContext)
exitExpr(ExprContext)
exitStmt_Assign(Stmt_AssignContext)
```

Figure 3: Parse Tree Walker Call sequence

As shown in the above figures (Figure 2 and 3) the walker performs a depth-first traversal of the parse tree, calling rule/sub-rule entry function on encounter of every new node and the corresponding exit function after visiting every child of the node.

Parse Tree Visitor

ANTLR provides an option called “visitor” which requests generation of ParseTreeVisitor (visitor) interface. The visitor interface provides a visit method for each rule/sub-rule of the grammar. Using visitors, we can govern the traversal of the tree by explicitly calling methods to visit children. This means the programmer can choose exactly which child nodes should be visited and which should be avoided. This is the major difference between listeners and visitors of a parse tree. While listener does not provide this flexibility, it visits all the child nodes regardless. Figure 4 and 5 illustrate how the traversal flow of visitor differs from that of the listener.

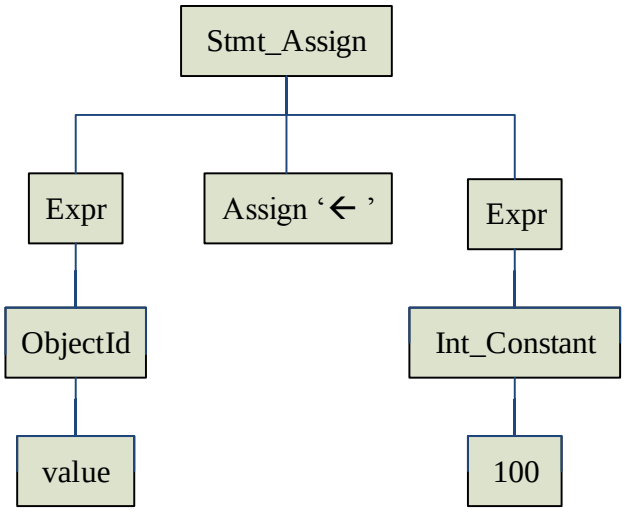

Figure 4: Sample Parse Tree

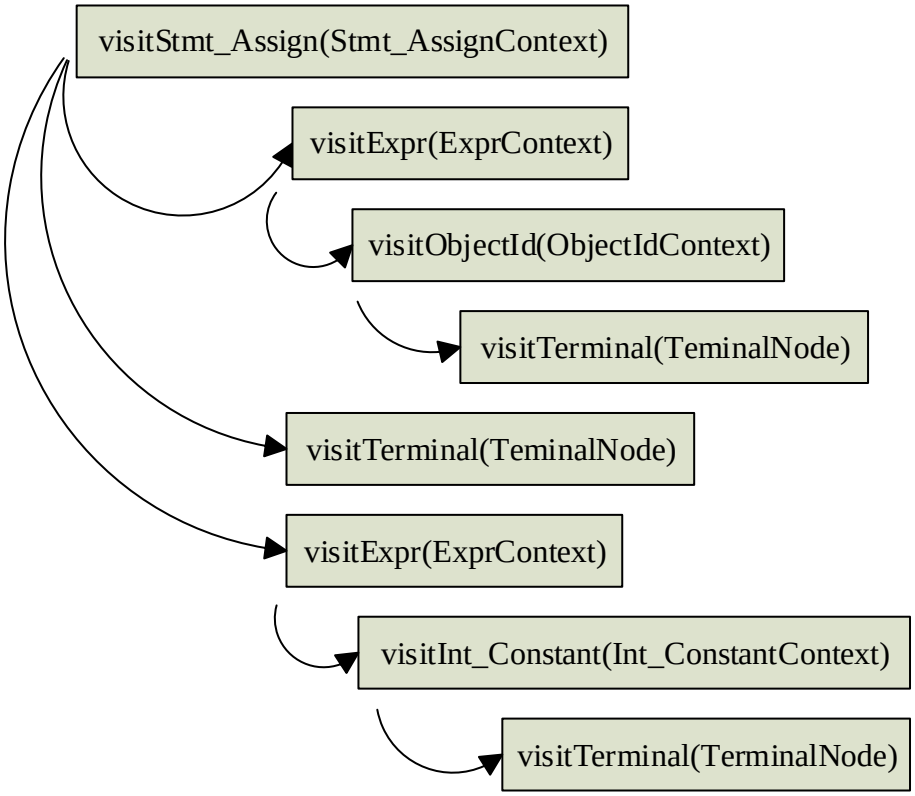

Figure 5: Visitor Call sequence

Working of the Parse Tree Visitor:

- In the default implementation of visitor, every visitor function makes a call to visit all its children nodes.
- The Figure 5 illustrates this default behavior. Indentation is used explicitly to demonstrate the call to a child node.
- This calling sequence of visitor can be altered according to the needs of the programmer.

Another very useful feature provided in ANTLR 4 is the ability to annotate a parse tree. ANTLR auto-generates the signature for the listener and visitor methods. With the auto-generated signature, a programmer does not have the option of having application specific arguments or return values in the case of listeners. Regarding visitors, visitor methods cannot have application specific arguments. To

overcome these shortcomings, ANTLR provides parse tree Annotation. Annotating implies we can store certain information/value for each node of the parse tree. This feature helps to collect information which can be accessed across method calls or events, helping programmers to pass around data without the use of global variables. This results in lesser overhead pertaining to data storage.

LLVM

For implementing the compiler back end, I have chosen LLVM (Low Level Virtual Machine) framework that is written in C++. LLVM is a huge open source project sheltering various libraries under its umbrella. The main feature of LLVM that influenced my choice is that, LLVM provides libraries for constructing Intermediate Representation (IR) of the program that is completely independent of the source as well as the target. This implies two advantages,

- Since LLVM provides library for building IR, programmer does not need to learn the specifics of LLVM IR. Triggering calls to the right IR Builder functions will do the job of generating IR for the code.
- Programmer does not need to understand the target machine specifics. It is taken care of by LLVM under the hood. LLVM provides code generation support for many popular CPUs (x86, ARM, MIPS). By just constructing the LLVM IR using LLVM IR Builder helper functions, we are completely bypassing assembly level machine specific details that have always been difficult to program/debug and stumped the programmers.
- This feature also advocates platform independence at machine level, since the same LLVM IR can be used to generate any target specific machine code supported by LLVM.
- LLVM also provides support for code optimization. It performs optimizations on the generated LLVM IR that is target independent.

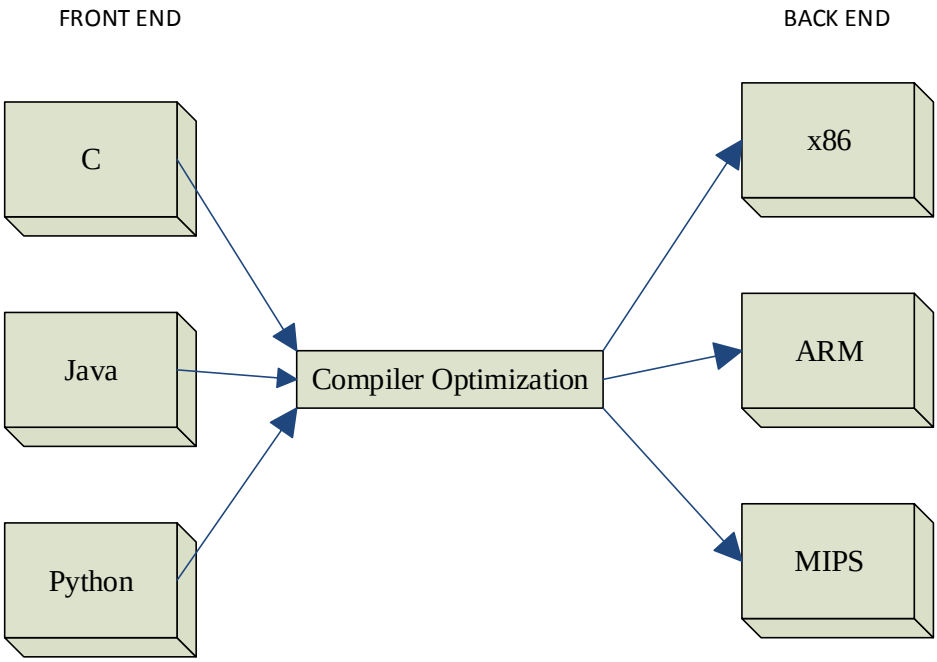

Figure 6 LLVM-Platform Independence

Figure 6 shows the ability of LLVM to perform source and target independent compiler optimizations.

COMPILER IMPLEMENTATION

To implement any language compiler it is important to understand the language constructs and their functionality. The Stanford course provides a fully descriptive COOL Manual [3] which describes COOL grammar specifics and the language semantics. Once we have a clear idea about the language, we can take the next step of implementing the language compiler. Rest of the section describes how I implemented each stage of the compiler.

Lexical Analyzer

A Lexical Analyzer, also known as lexer, tokenizer or scanner, is a program that reads the source code character by character and groups these characters to form words that belong to the programming language. In computer science world, each word of the programming language is called a lexeme. These lexemes are further classified into categories. A lexeme and its classification together as one unit is called a token. This process of producing tokens, known as tokenization is represented in Figure 7.

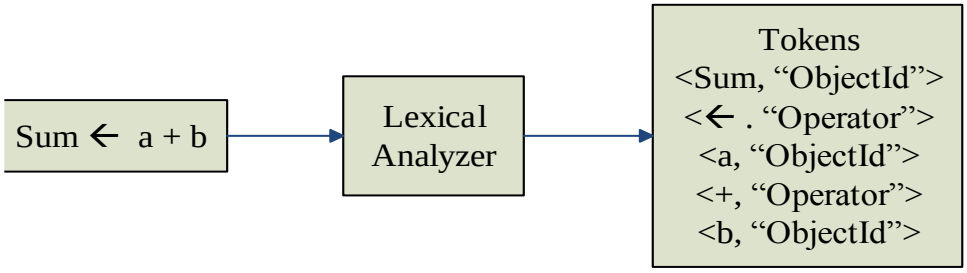

Figure 7: Lexical Analyzer

Implementation

1. To start with, all the different types of tokens that belong to COOL were identified, with the help of COOL grammar (understanding supported data types).
2. Certain specifications regarding data type and data object naming conventions were also taken into consideration.
3. Rules were framed taking into account the scope of each token to ensure that the rule specified covered all the possibilities in which the token could be represented.

To get a better idea of implementing a lexer, let us consider a small portion of COOL that deals with Integers and supports integer arithmetic and comparison operations. Following is a snapshot of a lexer grammar file representing the Integer arithmetic of COOL grammar.

```
lexer grammar IntArith;
Int_Const: [0-9]+;                          //Integer constants
ObjectId: [a-z][a-zA-Z0-9_]*;               //COOL variables
//arithmatic operators
Add:        '+';
Sub:        '-';
Mul:        '*';
Div:        '/';
Assign:     '<-';                           //used for assigning value
//comparison operators
Lt:         '<';
Le:         '<=';
Equal:      '=';                            //compare if two operands are equal

//ignore whilespaces
Space : [ \t\r\f\n]+ ->skip ;
```

Figure 8: Integer Arithmatic Grammar Lexer Rules

Figure 8 contains the rules for identifying tokens of Integer Arithmetic grammar. This grammar file is used by ANTLR to generate a code which functions as a lexer for this grammar.

Parser

Parser is the unit of the compiler which is responsible for checking the language syntax. In a very similar fashion to that of a Lexer, parsing rules are specified by the user, which represent all the possible constructs that the programming language supports. The programming language construct is the assembly of tokens in a particular order which represents a syntactically correct statement of the language.

Parser receives a sequence of tokens as the input, which it then scans to check if the order of tokens follows the language syntax.

Implementation

1. To frame parser rules, first understand the COOL grammar and extract the syntax of the language from the grammar.
2. To simplify grammar debugging, break down complex grammar rules into simpler shorter rules.
3. ANTLR auto-generates a recursive descent parser for the grammar provided. It has to be ensured that all the syntactical constructs provided by the language are taken care of in the parser.

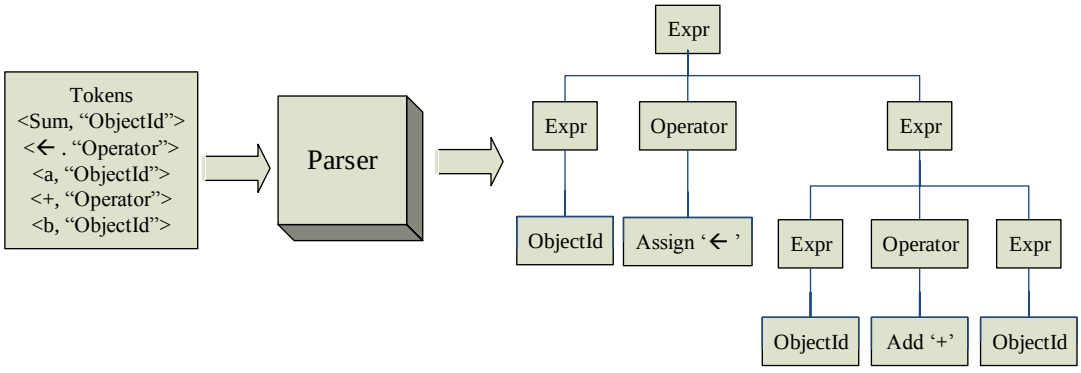

Figure 9: Parser Workflow

Figure 9 illustrates the following:

- Parser is provided with a sequence of Tokens generated by the lexer as the input.
- These tokens are then assembled together by the parser logic to match syntactical constructs provided by the language.
- This assembling of tokens is carried out by the ANTLR auto-generated parser.
- ANTLR also generates the parse tree resulting from the input tokens.
- ANTLR provides options to view the output parse tree in the form of multiple representations. The options I found very useful are option gui provides visual display of the parse tree generated and option tree prints out parse tree in Lisp form.
- The graphical parse tree representation option is very useful, especially when constructing parser rules for huge and complex grammars. It highlights the errors, giving the user the whole picture idea at one glance.

Now let us walk through generating parser rules for the IntegerArithmatic grammar for which we generated the lexer in the previous section.

Figure 10 is the snapshot of the grammar file containing only parser rules for the IntegerArithmatic grammar.

```
grammar IntArith;

expr : ObjectId Assign expr
        |expr (Mul | Div) expr
        |expr (Add | Sub) expr
        |expr (Lt | Le | Equal) expr
        |ObjectId
        |Int_Const;
```

Figure 10: Integer Arithmatic Grammar Parser Rules

Another very important syntactical clause that we need to take care of while specifying parser rules is operator precedence. ANTLR has a very simple way of assigning operator precedence, that is, the precedence is assigned according to the order of the rules written in the grammar file. Rules defined first have higher precedence than the rules defined later. For instance, in the above example (Figure 10), the rule for multiplication and division operation is defined before the rule for addition and subtraction. Hence, when parser is constructing a parse tree for any expression that contains some combination of these arithmetic operators, multiplication and division will be given precedence (evaluated first) over addition and subtraction. Operators specified in the same rule are treated as equal. As we can see, changing the order of evaluation for an expression containing operators with same priority does not change the result.

We can thus conclude that, a parser is responsible for checking if all the statements written in the source code follow the programming language construct syntax.

Semantic Analyzer

Now that we have a syntactically correct program and have generated its parse tree, we need to check the program semantics (meaning of the code). Semantic analysis mainly ensures the meaning or evaluation of the code is consistent with the programming language constructs and the data types used.

To check that our program is semantically correct we first need to understand the meaning and scope of each construct of the programming language. For example, in COOL we can perform arithmetic operations only on Integers and not on Strings or Boolean values.

Implementation

As COOL supports features like forward referencing and user defined data types, semantic analysis is carried out over two stages. In the first stage, information required for performing semantic analysis is collected. In the second stage, semantics of the program are tested using the information gathered in the first stage.

Semantic Analysis Phase 1

For the first stage, it had to be ensured that every node in the parse tree was visited and no node is missed. To guarantee this, Phase1 of semantic analyzer was implemented using ParseTreeListeners.

The information gathered during Phase1 has to be appropriately stored so that it is retrievable in Phase 2 of the semantic analyzer. The information needed for semantic analysis includes:

- Class data
    - class name
    - parent name

- Method data
    - class to which the method belongs
    - method signature
- Attribute data
    - class to which the attribute belongs
    - attribute type

Following figures (Figure 11, 12 and 13) give an idea of the data structures used for storage of required information in Phase1

HashMap<String, String> classTable

Class Name	Class Parent

Figure 11: Class Table

HashMap<String, HashMap<String, method>> methodTable

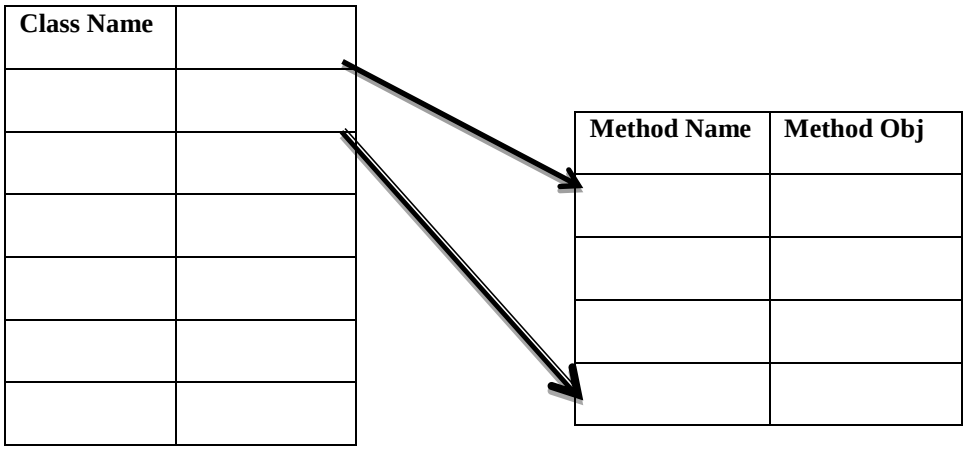

Figure 12: Method Table

HashMap<String, HashMap<String, String>> attributeTabel

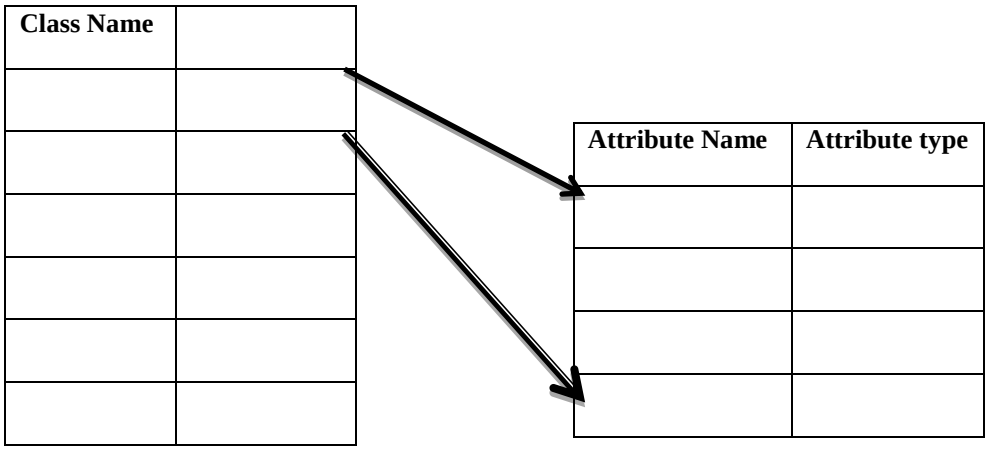

Figure 13: Attribute Table

Semantic Analyzer Phase 2

COOL being an object oriented language, it provides features like encapsulation of data and single inheritance. The fore mentioned, object oriented concepts imply that type checking involves keeping track of basic types and also user defined types, and their inheritance parentage. Type checking is thus a very important step in semantic analysis. It not only involves checking operand types for different kinds of operations but it also involves checking type in other programming language constructs. For example, checking function return type with its return expression. Features like inheritance and data encapsulation add to the challenges in type checking.

The primary reason for implementing Phase-2 of semantic analyzer using parse tree visitor instead of parse tree listener is:

- Option of having application specific return type for visitors. This enables propagation of the type value of each construct up the parse tree.
- Option of visiting only the necessary children and not wasting bandwidth over traversing the whole tree.

Even though, major semantic analysis revolves around type checking, it also includes checking for error conditions such as inheritance cycle, signature mismatch in method overriding, etc.

To sum it up, the main goal of semantic analysis is to check that the program uses language constructs and data types meaningfully.

In most cases, the output of semantic analyzer is an Abstract Syntax Tree (AST). An AST can be considered to be a compact version of parse tree in which the unnecessary nodes are eliminated, retaining only abstract representation of the input. However, in my implementation, I have used the parse tree visitor interface to traverse only the necessary nodes of the parse tree instead of constructing and using AST.

Code Generator

The code generator module is responsible for taking the parse tree and generating the corresponding machine code which can be run directly on the processor.

In this implementation, I have used LLVM for code generation. LLVM provides library functions to generate Intermediate Representation (IR) which is independent of source and target. As explained before, the first three stages of the compiler are implemented using ANTLR, which is a Java based tool, while LLVM is a C++ based tool. Hence there are issues regarding code generation when using LLVM libraries.

Since calling LLVM provided APIs from a Java application is not possible, direct generation of LLVM IR for each construct was implemented as an alternative. The generated LLVM IR is then optimized by the tool and converted into target specific machine code.

Implementation

- For code generation, parse tree visitor is used to traverse only the necessary nodes of the parse tree.
- On visiting any node, corresponding LLVM IR code is generated by my program depending on the rule construct indicated by the node. This generated code is then written into a file.
- Upon completion of the traversal by the visitor, the output file contains LLVM IR for the entire input program.
- This file containing LLVM IR is further optimized and used for generating target specific code.

Figure 14 shows an example illustrating the LLVM IR generated for an expression of COOL code.

To perform arithmetic addition – Expression a + b, in LLVM IR

```
;load value of 1st integer in register
%1 = call i32 @Int_getValue(%Int* %a)

;load value of 2nd integer in register
%2 = call i32 @Int_getValue(%Int*  %b)
;perform addition
%3 = add i32 %1, %2

;create an Integer object to store result
%intObj = alloca %Int
;set value
call void @Int_setValue(%Int* %intObj, i32* %3)
```

Figure 14: LLVM Code Generated

Notes:

- The above example is a part of my code generator implementation.
- In my implementation, I have defined "Int" as a user defined class using LLVM struct construct.

  %intObj = alloca %Int ; allocates sizeof(Int) for %intObj
- Since this is a class storing integer value, I have also implemented getter and setter methods for retrieving and storing the data.

ILLUSTRATION OF THE IMPLEMENTATION

In this section, we walk through an example where we run the implemented COOL compiler on a "HelloWorld" program. Below is the illustration of how each stage of the compiler works on the program given in Figure 15.

```
Class Main inherits IO {
        main():SELF_TYPE {
                out_string("Hello World\n")
        };
};
```

Figure 15: "HelloWorld" Program

Lexer:

In this stage the entire program (Figure 15), taken as input, is broken down into tokens. Figure 16 shows the output of the lexer.

```
C:\Users\Avni\workspace\CoolProject>java org.antlr.v4.runtime.misc.TestRig cool
program -tokens helloWorld.cl
[@0,0:4='class',<1>,1:0]
[@1,6:9='Main',<23>,1:6]
[@2,11:18='inherits',<6>,1:11]
[@3,20:21='IO',<23>,1:20]
[@4,23:23='{',<31>,1:23]
[@5,27:30='main',<22>,2:1]
[@6,31:31='(',<29>,2:5]
[@7,32:32=')',<30>,2:6]
[@8,33:33=':',<40>,2:7]
[@9,34:42='SELF_TYPE',<23>,2:8]
[@10,43:43='{',<31>,2:17]
[@11,48:57='out_string',<22>,3:2]
[@12,58:58='(',<29>,3:12]
[@13,59:71='Hello World',<48>,3:13]
[@14,72:72=')',<30>,3:26]
[@15,76:76='}',<32>,4:1]
[@16,77:77=';',<39>,4:2]
[@17,80:80='}',<32>,5:0]
[@18,81:81=';',<39>,5:1]
[@19,82:81='<EOF>',<-1>,5:2]
```

Figure 16: HelloWorld - Lexer Output

Each line of the output represents a token. Here is the explanation for one of the output lines.

[@3,20:21='IO',<23>,1:20] line indicates the following:

- @3 – fourth token (as they are indexed from zero)
- 20:21 – consists of characters 20 through 21 (again starting from zero)
- 'IO' – token text is "IO"
- <23> - token type is 23 (this token ID is auto-generated by ANTLR and stored in file <grammar_name>.tokens, in this case cool.tokens)
- 1:20 – the token is on line 1 at character position 20.

Parser:

Parser takes in as input the above tokens and with the help of grammar parser rules, it auto-generates the parse tree. With the –gui command line option we can display the parse tree visually in a dialog box as shown in Figure 17.

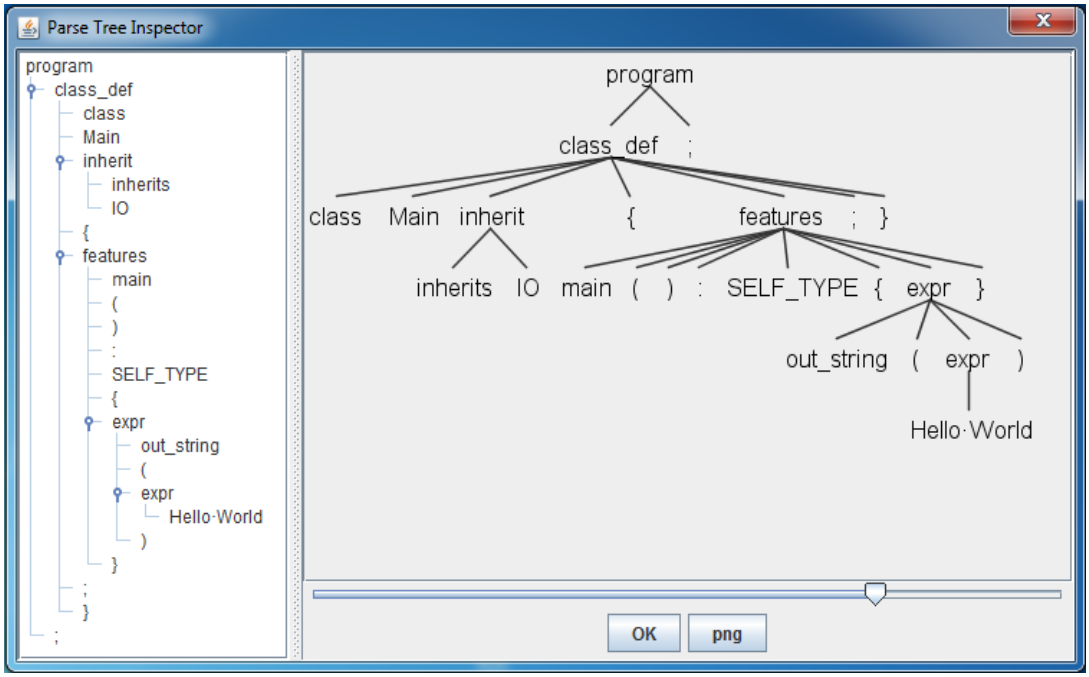

Figure 17: HelloWorld - Parse Tree

Semantic Analyzer:

The semantic analyzer of the compiler checks for semantic errors if any. Following semantic checks are performed on this program

- Check whether the programs contains class Main with a function main( ).
- Check whether class ‘IO’ exists: As class Main inherits class IO, it is mandatory for the existence of class IO. Class IO is one of the built in classes, and hence the check succeeds.
- Check whether function ‘out_string(String)’ is defined
- Check whether the argument of the function ‘out_string’ is of type String.

Since this program is semantically correct, no errors are voiced by the Semantic Analyzer.

Code Generator:

Figure 18 shows the code generated by the implemented compiler.

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-f3
2:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:6
4-S128"
target triple = "x86_64-unknown-linux-gnu"
declare i32 @printf(i8*, ...)
declare i32 @__isoc99_scanf(i8*, ...)
@.msg1 = public constant [12 x i8] c"Hello World\0A\00"
%Object = type { i8* }
%Int = type { i32 }
define void @Int_setValue(%Int* %this, i32 %value) nounwind{
%1 = getelementptr %Int* %this, i32 0, i32 0
store i32 %value, i32* %1ret void
}
define i32 @Int_getValue(%%Int* %%this) nounwind {

%1 = getelementptr %Int* %this, i32 0, i32 0
%2 = load i32* %this
ret i32 %2
}
%String = type { i8, i32 }
%Bool = type { i8 }
define void @Bool_setValue(%Bool* %this, i8 %value) nounwind {
%1 = getelementptr %Bool* %this, i32 0, i32 0
store i8 %value, i8* %1
ret void
}
define i8 @Bool_getValue(%%Bool* %%this) nounwind {
%1 = getelementptr %Bool* %this, i32 0, i32 0
%2 = load i8* %this
ret i8 %2
}
define i32 @main() nounwind uwtable {
entry:
call i32 (i8 *, ...)* @printf(i8* getelementptr inbounds ([11 x i8]* @.msg1, i32
 0, i32 0))
}
```

Figure 18: HelloWorld - Code Generator

The corresponding machine code (x86 code) generated is as shown in Figure 19

```
    .file    "helloworld.ll"
    .text
    .globl    main
    .align    16, 0x90
    .type    main,@function
main:                                   # @main
    .cfi_startproc
# BB#0:                                 # %entry
    pushq    %rbp
.Ltmp2:
    .cfi_def_cfa_offset 16
.Ltmp3:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
.Ltmp4:
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
    leaq    .L.str, %rdi
    movl    $0, -4(%rbp)
    movb    $0, %al
    callq    printf
    movl    $0, %ecx
    movl    %eax, -8(%rbp)          # 4-byte Spill
    movl    %ecx, %eax
    addq    $16, %rsp
    popq    %rbp
    ret
.Ltmp5:
    .size    main, .Ltmp5-main
    .cfi_endproc

    .type    .L.str,@object          # @.str
    .section    .rodata.str1.1,"aMS",@progbits,1
.L.str:
    .asciz     "Hello World"
    .size    .L.str, 12
```

Figure 19: HelloWorld - x86 code

FUTURE SCOPE

The code generator implemented in this project does not cover some aspects of the COOL language. A possible future exercise could be to implement the remaining features of the code generator.

The code generator has been implemented by directly converting COOL constructs into their corresponding LLVM IR. Alternatively, the code generator could also be implemented using StringTemplate Engine.

CONCLUSION

This project report provides an overview of the tools and utilities available for compiler development. Any interested reader can use this report as a guide for implementing their own compiler using ANTLR 4 and LLVM.

This project gave me the opportunity to understand the intricacies of compiler working as well as the power of modern day tools and utilities.

APPENDIX A Lexer Code

```
lexer grammar cool;

@members {
        StringBuilder buf = new StringBuilder();
}

/* LEXER RULES*/

/* for case insensitive matching */

fragment A:('a'|'A');

fragment B:('b'|'B');

fragment C:('c'|'C');

fragment D:('d'|'D');

fragment E:('e'|'E');

fragment F:('f'|'F');

fragment G:('g'|'G');

fragment H:('h'|'H');

fragment I:('i'|'I');

fragment J:('j'|'J');

fragment K:('k'|'K');

fragment L:('l'|'L');

fragment M:('m'|'M');

fragment N:('n'|'N');

fragment O:('o'|'O');

fragment P:('p'|'P');

fragment Q:('q'|'Q');

fragment R:('r'|'R');

fragment S:('s'|'S');

fragment T:('t'|'T');
```

```
fragment U:('u'|'U');

fragment V:('v'|'V');

fragment W:('w'|'W');

fragment X:('x'|'X');

fragment Y:('y'|'Y');

fragment Z:('z'|'Z');


CLASS:      C L A S S;

ELSE:       E L S E;

FI:         F I;

IF:         I F;

IN:         I N;

INHERITS:   I N H E R I T S;

LET:        L E T;

LOOP:       L O O P;

POOL:       P O O L;

THEN:       T H E N;

WHILE:      W H I L E;

CASE:       C A S E;

ESAC:       E S A C;

OF:         O F;

NEW:        N E W;

ISVOID:     I S V O I D;

NOT:        N O T;


TRUE:       [t]R U E;

FALSE:      [f]A L S E;
```

```
BOOL_CONST:  TRUE | FALSE;
INT_CONST:          [0-9]+;

OBJECTID:     [a-z][a-zA-Z0-9_]*;
TYPEID:       [A-Z][a-zA-Z0-9_]*;

ADD:                '+';
SUB:                '-';
DIV:                '/';
MUL:                '*';
COMPLEMENT:'~';
OPEN_PAREN:  '(';
CLOSE_PAREN:        ')';
CURLY_OPEN:  '{';
CURLY_CLOSE:        '}';
EQUAL:              '=';
LT:                 '<';
LE:                 '<=';
//GT:               '>';
//GE:               '>=';
ASSIGN:             '<-';
DARROW:             '=>';
END_OF_LINE: ';';
COLON:              ':';
AT:                 '@';
COMMA:              ',';
```

```
DOT:                    '.';

SPACE : [ \t\r\f\n\u00B]+ ->skip ; // skip spaces, tabs, newlines
COMMENT:    ('(*'(COMMENT|.)*?'*)')->skip;
CLOSE_COMMENT:    '*)' {setText("Error");};
LINE_COMMENT:   '--' ~[\r\n]* -> skip;

STR_CONST :   ""
    (           '\\n'     {buf.append('\n');}
      | '\\\n'    {buf.append('\n');}
      |  '\\t'    {buf.append('\t');}
      |  '\\b'    {buf.append('\b');}
      |  '\\f'    {buf.append('\f');}
      | '\\r'     {buf.append('\r');}
      |  '\\"'    {buf.append('\"');}
      |  '\\\'    {buf.append('\');}
      | '\\\\'    {buf.append('\\');}
        | ~('\\|'"')  {buf.append((char)_input.LA(-1));})*
    ""
    {setText(buf.toString()); buf.setLength(0);};
REST:           . {setText("Error");};
```

APPENDIX B Parser Code

```
grammar cool;

/*Parser Rules */

program:(class_def END_OF_LINE)+;

class_def: CLASSTYPEID (inherit)? CURLY_OPEN (features END_OF_LINE)*CURLY_CLOSE;

inherit: INHERITSTYPEID;

features:OBJECTIDOPEN_PAREN ((formal)(COMMA formal)*)?
        CLOSE_PARENCOLONTYPEIDCURLY_OPEN expr CURLY_CLOSE        #method
        | OBJECTIDCOLONTYPEID (ASSIGN expr)?                     #attribute


formal:   OBJECTIDCOLONTYPEID;

expr:
OBJECTIDASSIGN expr                                              #opAssign
|expr (ATTYPEID)? DOTOBJECTIDOPEN_PAREN ((expr)(COMMA expr)*)? CLOSE_PARENT
                                                                 #staticDispatch
|OBJECTIDOPEN_PAREN ((expr)(COMMA expr)*)? CLOSE_PAREN
                                                                 #dynamicDispatch
|WHILE expr LOOP expr POOL                                       #whileLoop
|CURLY_OPEN expr CURLY_CLOSE                                     #block
|CURLY_OPEN (expr END_OF_LINE)+ CURLY_CLOSE                      #block1
|IF expr THEN expr ELSE expr FI                                  #ifThenElse
|LET let_expr                                                    #let
|CASE expr OF branch+ ESAC                                       #switch
|NEWTYPEID                                                       #newType
|ISVOID expr                                                     #isVoid
|COMPLEMENT expr                                                 #opComplement
|expr (MUL|DIV) expr                                            #mulDiv
```

```
|expr (ADD|SUB) expr                                  #addSub
|expr LT expr                                        #lessThan
|expr LE expr                                        #lessEqual
|expr EQUAL expr                                     #opEqual
|NOT expr                                           #opNot
|OPEN_PAREN expr CLOSE_PAREN                        #group
|OBJECTID                                           #objId
|INT_CONST                                          #integer
|TRUE                                               #boolTrue
|FALSE                                              #boolFalse
|STR_CONST                                          #string   ;

let_expr:   OBJECTIDCOLONTYPEIDCOMMA let_expr                    #nestedLet
          |OBJECTIDCOLONTYPEIDIN expr                            #letIn
          |OBJECTIDCOLONTYPEIDASSIGN expr COMMA let_expr         #letAssignLet
          |OBJECTIDCOLONTYPEIDASSIGN expr IN expr                #letAssignIn;

branch:  OBJECTIDCOLONTYPEIDDARROW expr END_OF_LINE;
```

APPENDIX C Semantic Analyzer Code

semantPhase1.java

```
//Information Gathering Phase
import java.util.HashMap;
import java.util.LinkedHashMap;
import java.util.Map;

publicclass semantPhase1 extends coolBaseListener{

  //class table map which has key=class name , value = parent name
  Map<String, String> class_table = new HashMap<String, String>();

  //method table[class_name]<method name, pointer to method object>
  Map<String, Map<String, method>> method_table = new HashMap<String, Map<String,
method>>();
  Map<String, method> mt = new HashMap<String, method>();
  LinkedHashMap<String, String> formals = new LinkedHashMap<String, String>();

  //attribute table <attribute name, attribute type>
  Map<String, Map<String, String>> attr_table = new HashMap<String, Map<String, String>>();
  Map<String, String> attr = new HashMap<String, String>();

  SymbolTable object_table = new SymbolTable();

  //program
  @Override publicvoid enterProgram(coolParser.ProgramContext ctx) {
        //inserting basic classes
```

```
        class_table.put("Object", "No_class");
        class_table.put("IO", "Object");
        class_table.put("Bool", "Object");
        class_table.put("String", "Object");
        class_table.put("Int", "Object");
        class_table.put("SELF_TYPE", "No_class");
}

//class
@Override publicvoid enterClass_def(coolParser.Class_defContext ctx) {
  //check for class redefinition
        if(class_table.containsKey(ctx.TYPEID().getText())){
                System.out.println("ERROR: Redefinition of Class "+ctx.TYPEID().getText());
        }else{
                if(ctx.TYPEID().getText().equalsIgnoreCase("Self_type")){
                        System.out.println("ERROR: Class name can not be Self_type.");
                }elseif(ctx.inherit()==null){
                        System.out.println("added: class: "+ctx.TYPEID().getText()+"parent:
Object");
                        class_table.put(ctx.TYPEID().getText(), "Object");
                }
        }
        //enter class scope
        object_table.enterScope();
}
```

```
@Override publicvoid exitClass_def(coolParser.Class_defContext ctx) {
        //add methods of a class to method table
        method_table.put(ctx.TYPEID().getText(), new HashMap<String, method>(mt));
        mt.clear();
        //add attributes of a class to attribute table
        attr_table.put(ctx.TYPEID().getText(), new HashMap<String, String>(attr));
        attr.clear();
        //exit class scope
        object_table.exitScope();
}

//inherit
@Override publicvoid enterInherit(coolParser.InheritContext ctx) {
        //check-can not inherit from basic types
        if(ctx.TYPEID().getText().equals("Int")
                || ctx.TYPEID().getText().equals("String")
                || ctx.TYPEID().getText().equals("Bool")
                || ctx.TYPEID().getText().equals("Self_Type")){
                System.out.println("ERROR:Can not Inherit from "+ctx.TYPEID().getText());
        }else{
                class_table.put(ctx.getParent().getChild(1).getText(), ctx.TYPEID().getText());
        }
}

//feature - method
@Override publicvoid enterMethod(coolParser.MethodContext ctx) {
        //enter method scope
```

```
        object_table.enterScope();
}
@Override publicvoid exitMethod(coolParser.MethodContext ctx) {
        //store method details
        mt.put(ctx.OBJECTID().getText(), new method(ctx.OBJECTID().getText(), formals,
                        ctx.TYPEID().getText()));
        formals.clear();
        object_table.exitScope();
}
//feature - attribute
@Override publicvoid enterAttribute(coolParser.AttributeContext ctx) {
        //check "self" in attribute
        if(ctx.OBJECTID().getText().equals("self")==true){
                System.out.println("ERROR: Can not have self as attribute name.");
                return;
        }
        //check attribute redefinition
        if(object_table.probe(ctx.OBJECTID().getText())!=null){
                System.out.println("ERROR: Attribute redefinition.");
                return;
        }
        object_table.addId(ctx.OBJECTID().getText(), new String(ctx.TYPEID().getText()));
        attr.put(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
}
//formal (arguments of method)
@Override publicvoid enterFormal(coolParser.FormalContext ctx) {
        if(ctx.OBJECTID().getText().equals("self")==true){
```

```
                System.out.println("ERROR: Can not have self as a formal name.");
                return;
        }
        if(ctx.TYPEID().getText().equalsIgnoreCase("SELF_TYPE")==true){
System.out.println("ERROR: Can not have formal "+ctx.OBJECTID().getText()+"of SELF_TYPE.");
                return;
        }
        //enter formals in map formals
        if(formals.containsKey(ctx.OBJECTID().getText())==false){
                formals.put(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
                object_table.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
        }else{
                System.out.println("ERROR: Redefinition of a formal");
        }

}
//all lets
@Override publicvoid enterLet(coolParser.LetContext ctx) {
        object_table.enterScope();
}
@Override publicvoid exitLet(coolParser.LetContext ctx) {
        object_table.exitScope();
}
@Override publicvoid enterNestedLet(coolParser.NestedLetContext ctx) {
        //OBJECTID COLON TYPEID COMMA let_expr
        if(ctx.OBJECTID().getText().equals("self")){
                System.out.println("ERROR: self in Let binding");
```

```
            }
            if(object_table.probe(ctx.OBJECTID().getText())!=null){
                  System.out.println("ERROR: Identifier redefinition in let clause.");
                  return;
            }
            object_table.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
}
@Override publicvoid enterLetIn(coolParser.LetInContext ctx) {
            //OBJECTID COLON TYPEID IN expr
            if(ctx.OBJECTID().getText().equals("self")){
                  System.out.println("ERROR: self in Let binding");
            }
            if(object_table.probe(ctx.OBJECTID().getText())!=null){
                  System.out.println("ERROR: Identifier redefinition in let clause.");
                  return;
            }
            object_table.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
}
@Override publicvoid enterLetAssignLet(coolParser.LetAssignLetContext ctx) {
            //OBJECTID COLON TYPEID ASSIGN expr COMMA let_expr
            if(ctx.OBJECTID().getText().equals("self")){
                  System.out.println("ERROR: self in Let binding");
            }
            if(object_table.probe(ctx.OBJECTID().getText())!=null){
                  System.out.println("ERROR: Identifier redefinition in let clause.");
                  return;
            }
```

```
            object_table.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
}
@Override publicvoid enterLetAssignIn(coolParser.LetAssignInContext ctx) {
            //OBJECTID COLON TYPEID ASSIGN expr IN expr
            if(ctx.OBJECTID().getText().equals("self")){
                        System.out.println("ERROR: self in Let binding");
            }
            if(object_table.probe(ctx.OBJECTID().getText())!=null){
                        System.out.println("ERROR: Identifier redefinition in let clause.");
                        return;
            }
            object_table.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
}

//expr - switch case branch
@Override publicvoid enterBranch(coolParser.BranchContext ctx) {
            //OBJECTID COLON TYPEID DARROW expr END_OF_LINE;
            object_table.enterScope();
            object_table.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
}

@Override publicvoid exitBranch(coolParser.BranchContext ctx) {
            object_table.exitScope();
}
}//end semanticPhase1
```

semantPhase2.java

```
//Semantic checking Phase
import java.util.HashMap;
import java.util.Iterator;
import java.util.LinkedHashMap;
import java.util.Map;
import java.util.Vector;

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

publicclass semantPhase2 extends coolBaseVisitor <String>{
Map<String, String> classTable = new HashMap<String, String>();
String current_class;

Map<String, Map<String, method>> methodTable = new HashMap<String, Map<String,
method>>();
Map<String, Map<String, String>> attributes = new HashMap<String, Map<String, String>>();
SymbolTable objectTable = new SymbolTable();
Map<String, String> strings = new HashMap<String, String>();

Vector<String> branch_decl_type= new Vector<String>();
Vector<String> branch_type= new Vector<String>();
String result;
publicvoid Init(ParseTree tree, Map<String, String> class_table,
        Map<String, Map<String, method>> method_table,
        Map<String, Map<String, String>> attribute){
        classTable.putAll(class_table);
```

```
methodTable.putAll(method_table);
//Insert all basic class attributes and functions
LinkedHashMap<String, String> formals = new LinkedHashMap<String, String>();
Map<String, method> met = new HashMap<String, method>();
//class Object
met.put("abort", new method("abort", formals, "Object"));
met.put("type_name", new method("type_name", formals, "String"));
met.put("copy", new method("copy", formals, "SELF_TYPE"));
methodTable.put("Object", new HashMap<String, method>(met));
met.clear();
Map<String, String> attr = new HashMap<String, String>();
attr.put("type_name", "String");
attributes.put("Object", new HashMap<String, String>(attr));
//class IO
//out_string(Str) : SELF_TYPE       writes a string to the output
formals.put("arg", "String");
met.put("out_string", new method("out_string", formals, "SELF_TYPE"));
formals.clear();
//out_int(Int) : SELF_TYPE          writes an integer to the output
formals.put("arg", "Int");
met.put("out_int", new method("out_int", formals, "SELF_TYPE"));
formals.clear();
 //in_string() : Str                reads a string from the input
met.put("in_string", new method("in_string", formals, "String"));
 //in_int() : Int                   reads an integer from the input
met.put("in_int", new method("in_int", formals, "Int"));
methodTable.put("IO", new HashMap<String, method>(met));
```

```
		met.clear();

		//class String
		met.put("length", new method("length", formals, "Int"));
		formals.put("arg", "String");
		met.put("concat", new method("concat", formals, "String"));
		formals.clear();
		formals.put("arg1", "Int");
		formals.put("arg2", "Int");
		met.put("substr", new method("substr", formals, "String"));
		methodTable.put("String", new HashMap<String, method>(met));
		met.clear();
		formals.clear();

		attributes.putAll(attribute);
		result=visit(tree);
}

//return lowest upper bound for type
public String lub(String a, String b){
		if(is_subtype(a,b)){
			return b;
		}
		if(is_subtype(b,a)){
			return a;
		}
```

```
        while(!(is_subtype(a,b))){
                b = classTable.get(b);
        }
        return a;
}

//return true if 'a' is subtype of 'b'
public Boolean is_subtype(String a, String b){
        if(a.equalsIgnoreCase("SELF_TYPE")==true){
                a=current_class;
        }
        if(b.equalsIgnoreCase("SELF_TYPE")==true){
                b=current_class;
        }
        while(a.equalsIgnoreCase("No_class")==false){
                if(a.equals(b)==true){
                        System.out.println("returning true");
                        returntrue;
                }
                else{
                        a=classTable.get(a);
                }
        }
        returnfalse;
}
```

```
//get inheritance path - get all parents of a type
public Vector<String> get_inheritance(String type){
        Vector<String> path = new Vector<String>();
        if(type.equalsIgnoreCase("SELF_TYPE")){
                type = current_class;
        }
        for(; type.equals("No_class")==false; type=classTable.get(type)){
                path.add(type);
        }
        return path;
}

//check inherited attributes
publicvoid check_attributes(){
        //add base class attributes to derived classes
        //get current class attributes
        Map<String, String> current_attrs = new HashMap<String, String>();
        current_attrs = attributes.get(current_class);
        //get list of its parents
        Vector<String> parents = new Vector<String>();
        parents = get_inheritance(current_class);
        //copy attributes of each parent into current class
        Iterator<String> iter_parents = parents.iterator();
        while(iter_parents.hasNext()){
                String parent = iter_parents.next();
                if(!(parent.equals("No_class")) && !(parent.equals("IO"))){
```

```
                        Map<String, String> parent_attrs = new HashMap<String, String>();
                        parent_attrs = attributes.get(parent);
                        Iterator<String> parent_key = parent_attrs.keySet().iterator();
                        while(parent_key.hasNext()){
                                String attr_name = parent_key.next();
                                String attr_value = parent_attrs.get(attr_name);
                                if(current_attrs.containsKey(attr_name) == true){
                                if(attr_value.equals(current_attrs.get(attr_name)) == false){
                                        System.out.println("ERROR: Derived class can not have
                                same name data member("+attr_name+") with different type.");
                                }
                                }else{
                                        current_attrs.put(attr_name, attr_value);
                                }
                        }
                }
        }
}

//check inherited methods
publicvoid check_method(){
        Map<String, method> methodList = new HashMap<String, method>();
        methodList = methodTable.get(current_class);
        Vector<String> parents = new Vector<String>();
        parents = get_inheritance(current_class);
        Iterator<String> iter_parents = parents.iterator();
```

```
        while(iter_parents.hasNext()){
                String parent = iter_parents.next();
                if(!(parent.equals("No_class"))){
                Map<String, method> parent_methods = new HashMap<String, method>();
                        parent_methods = methodTable.get(parent);
                        Iterator<String> parent_key = parent_methods.keySet().iterator();
                        while(parent_key.hasNext()){
                                String method_name = parent_key.next();
                                method parent_method = parent_methods.get(method_name);
                                if(methodList.containsKey(method_name) == true){
                                  method current_method = methodList.get(method_name);

        if(!(parent_method.return_type.equals(current_method.return_type))
                                ||!(parent_method.formal_number == current_method.formal_number)){
                System.out.println("ERROR: Signature Mismatch for method "+method_name);
                        return;
                        }
                if(!(parent_method.formals.equals(current_method.formals))){
                 System.out.println("ERROR: Formals mismatch for method "+method_name);
                                        }
                                }
                        }
                }
        }
}
```

```
@Override public String visitProgram(coolParser.ProgramContext ctx) {
        int i=0;
        String result="NOERROR";
        //check for main class
        if(classTable.containsKey("Main")==false){
                System.out.println("ERROR: Main class not defined.");
                return "ERROR";
        }
        //check for inheritance cycle
        for(String key : classTable.keySet()){
                String parent = classTable.get(key);
                while((!parent.equals("No_class"))&&(!parent.equals(key))){
                        //check if parent exists
                        System.out.println("Parent = "+parent);
                if(classTable.containsKey(parent)==false&&!(parent.equals("No_class"))){
                                System.out.println("ERROR: Parent "+parent+" not defined");
                                return "ERROR";
                        }
                        parent = classTable.get(parent);
                }
                if(parent.equals(key)){
                        System.out.println("ERROR: Inheritance cycle");
                        return "ERROR";
                }
        }
```

```
        while(ctx.class_def(i)!=null){
                objectTable.enterScope();//entering class scope
                result=visit(ctx.class_def(i));
                if(result.equals("ERROR")){
                        break;
                }
                objectTable.exitScope();//exiting class scope
                i++;
        }
        return result;
}

@Override public String visitClass_def(coolParser.Class_defContext ctx) {
        //check if Main class has main method
        int f=0;
        String result="NOERROR";
        current_class= ctx.TYPEID().getText();
        check_attributes();
        check_method();
        if(ctx.TYPEID().getText().equals("Main")){
                Map<String, method> functions = new HashMap<String, method>();
                functions.putAll(methodTable.get(ctx.TYPEID().getText()));
                if(functions.containsKey("main")==false){
                  System.out.println("ERROR: Main class does not contain main() method.");
                        return "ERROR";
                }
        }
```

```
        while(ctx.features(f)!=null){
                result=visit(ctx.features(f));
                if(result.equals("ERROR")){
                        break;
                }
                f++;
        }
        return result;
}
@Override public String visitMethod(coolParser.MethodContext ctx) {
        int f=0;
        objectTable.enterScope();//entering method scope
        Map<String, method> functions = new HashMap<String, method>();
        functions.putAll(methodTable.get(current_class));
        method m = functions.get(ctx.OBJECTID().getText());
        //check if return type exists
        if(classTable.containsKey(m.return_type)==false
        || m.return_type.equalsIgnoreCase("SELF_TYPE")==false){
 System.out.println("ERROR:Return type of method "+ctx.OBJECTID().getText()+"does not exist.");
        }
        //populate objectTable with formals
        while(ctx.formal(f)!=null){
                String formal_type=visitFormal(ctx.formal(f));
                f++;
        }
```

```
        //check if formal types exist
        for(String value: m.formals.values()){
                if(classTable.containsKey(value)==false){
                        System.out.println("ERROR: Formal type "+value+" does not exist");
                }
        }
        //return type should be ancestor of method body expr type
        String expr_type = visit(ctx.expr());
        if(is_subtype(expr_type, ctx.TYPEID().getText())==false){
      System.out.println("ERROR: Method expression type is not a subtype of method return type.");
        }
        objectTable.exitScope();//exiting method scope
                return "NOERROR";
        }

        @Override public String visitFormal(coolParser.FormalContext ctx) {
                if(classTable.containsKey(ctx.TYPEID().getText())==true){
                        objectTable.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
                        return ctx.TYPEID().getText();
                }else{
             System.out.println("ERROR: Formal type "+ctx.TYPEID().getText()+" not defined");
                        return "ERROR";
                }
        }
}
```

```
@Override public String visitAttribute(coolParser.AttributeContext ctx) {
        if(ctx.OBJECTID().getText().equals("self")==true){
                System.out.println("ERROR: Can not have self as Attribute name.");
                return "ERROR";
        }
        if(objectTable.lookup(ctx.OBJECTID().getText())!=null){
                System.out.println("ERROR: Attribute redefinition.");
                return "ERROR";
        }
        objectTable.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
        if(ctx.expr()!=null){
                String result = visit(ctx.expr());
        }
        return "NOERROR";
}

//expr equal comparison operator
@Override public String visitOpEqual(coolParser.OpEqualContext ctx) {
        String type;
        String expr0_type = visit(ctx.expr(0));
        String expr1_type = visit(ctx.expr(1));
        if(expr0_type.equalsIgnoreCase("Int") || expr0_type.equalsIgnoreCase("String")
         || expr0_type.equalsIgnoreCase("Bool")|| expr1_type.equalsIgnoreCase("Int")
         || expr1_type.equalsIgnoreCase("String") || expr1_type.equalsIgnoreCase("Bool")){
                        if(expr0_type.equals(expr1_type)){
                                return "Bool";
```

```
                }else{
                System.out.println("ERROR: Can not Equal Expressions of different type.");
                        type="Object";
                }
        }else{
                type = "Bool";
        }
        return type;
}

//expr assign one expression to the other
@Override public String visitOpAssign(coolParser.OpAssignContext ctx) {
        String type_left=null;
        Map<String, String> attrs = attributes.get(current_class);
        if(objectTable.lookup(ctx.OBJECTID().getText())==null){
                if(attrs.containsKey(ctx.OBJECTID().getText())==true){
                        type_left = attrs.get(ctx.OBJECTID().getText());
                }
        }else{
                type_left = objectTable.lookup(ctx.OBJECTID().getText());
        }
        String type_right = visit(ctx.expr());
        String type;

        if(type_left == null){
          System.out.println("ERROR: Can not find lvalue("+ctx.OBJECTID().getText()+").");
                type="Object";
```

```
                return type;
        }
        if(is_subtype(type_right, type_left)==false){
                System.out.println("ERROR: rvalue is not a subtype of lvalue.");
                type="Object";
                return type;
        }
        return type_right;
}

@Override public String visitNewType(coolParser.NewTypeContext ctx) {
        if(!(ctx.TYPEID().getText().equals("Self_type"))
                &&(classTable.containsKey(ctx.TYPEID().getText())==false)){
                System.out.println("ERROR: Undefined type.");
        }
        return ctx.TYPEID().getText();
}
@Override public String visitOpNot(coolParser.OpNotContext ctx) {
        String not_type = visit(ctx.expr());
        if(!(not_type.equals("Bool"))){
                System.out.println("ERROR: Can not find complement of non-bool value.");
                not_type="Object";
        }
        return not_type;
}
```

```
@Override public String visitWhileLoop(coolParser.WhileLoopContext ctx) {
        if(visit(ctx.expr(0)).equals("Bool")==false){
                System.out.println("ERROR: While loop condition type is not Bool.");
        }
        String loop_type = visit(ctx.expr(1));
        loop_type="Object";
        return loop_type;
}

@Override public String visitBlock1(coolParser.Block1Context ctx) {
        int i=0;
        String block_type="Object";
        while(ctx.expr(i)!=null){
                block_type=visit(ctx.expr(i));
                i++;
        }
        return block_type;
}

@Override public String visitIfThenElse(coolParser.IfThenElseContext ctx) {
        if(visit(ctx.expr(0)).equals("Bool")==false){
                System.out.println("ERROR: Condition(of if then else) type is not Bool.");
        }
        String then_type=visit(ctx.expr(1));
        String else_type=visit(ctx.expr(2));
```

```
        String if_type=lub(then_type, else_type);
        return if_type;
}

//complement of integer(negate)
@Override public String visitOpComplement(coolParser.OpComplementContext ctx) {
        String comp_type="Int";
        if(visit(ctx.expr()).equals("Int")==false){
                System.out.println("ERROR: Can not negate non-integer value");
                comp_type="Object";
        }
        return comp_type;
}

@Override public String visitStaticDispatch(coolParser.StaticDispatchContext ctx) {
        boolean error = false;
        String staticdisp_type;
        String typeId="Self_type";
        String expr_type = visit(ctx.expr(0));
        if(ctx.TYPEID() !=null){
                typeId=ctx.TYPEID().getText();
                if(is_subtype(expr_type, ctx.TYPEID().getText())==false){
        System.out.println("ERROR: Expression ID is not a subtype of Static Dispatch class type");
                }
        }else{
                typeId=expr_type;
        }
```

```
Vector<String> parents = get_inheritance(typeId);
Iterator iter = parents.iterator();
Map<String, method> functs = new HashMap<String, method>();
method m = null;
while(iter.hasNext()){
        functs = methodTable.get(iter.next());
        if(functs !=null&& functs.containsKey(ctx.OBJECTID().getText())){
                m = functs.get(ctx.OBJECTID().getText());
                break;
        }
}
if(m == null){
        System.out.println("ERROR: Undefined method "+ctx.OBJECTID().getText()+".");
        error = true;
}else{
        // check if all actuals are sub-type of formals.
        int arg=1;
        Iterator arguments = m.formals.values().iterator();
        while(arguments.hasNext()){
                //checking formals
                if(ctx.expr(arg)==null){
                   System.out.println("ERROR: Number of arguments do not match.IN");
                        error=true;
                        break;
                }
                expr_type = visit(ctx.expr(arg));
                if(is_subtype(expr_type, arguments.next().toString())==false){
```

```
        System.out.println("ERROR:Actual argument type is not a sub type of formal argument type.");
                                error=true;
                        }
                        arg++;
                }
                if(ctx.expr(arg)!=null){
                        System.out.println("ERROR: Number of arguments do not match.OUT");
                        error=true;
                }
                if(error == true){
                        return "Object";
                }else{
                        staticdisp_type=m.return_type;
                        if(staticdisp_type.equalsIgnoreCase("SELF_TYPE")== true)
                                staticdisp_type = current_class;
                        return staticdisp_type;
                }
        }
        return visitChildren(ctx);
}
@Override public String visitDynamicDispatch(coolParser.DynamicDispatchContext ctx) {
        boolean error = false;
        String disp_type;
        Vector<String> parents = get_inheritance("SELF_TYPE");
        Iterator<String> iter = parents.iterator();
        Map<String, method> functs = new HashMap<String, method>();
        method m = null;
```

```
while(iter.hasNext()){
        functs = methodTable.get(iter.next());
        if(functs !=null){
            if(functs.containsKey(ctx.OBJECTID().getText())==true){
                m = functs.get(ctx.OBJECTID().getText());
                break;
            }
        }
}
if(m == null){
        System.out.println("ERROR: Undefined method "+ctx.OBJECTID().getText()+".");
        error = true;
}else{
        // check if all actuals are sub-type of formals.
        int i=0;
        for (Map.Entry<String, String> entry : m.formals.entrySet()){
            if(ctx.expr(i)==null){
              System.out.println("ERROR: Number of arguments do not match.in");
                error=true;
                break;
            }
            String expr_type = visit(ctx.expr(i));
            if(is_subtype(expr_type, entry.getValue())==false){
System.out.println("ERROR:Actual argument type is not a sub type of formal argument type.");
                error=true;
            }
            i++;
```

```
                }
                if(ctx.expr(i)!=null){
                        System.out.println("ERROR: Number of arguments do not match.out");
                        error=true;
                }
        }
        if(error==true){
                return "Object";
        }else{
                disp_type=m.return_type;
                if(disp_type.equalsIgnoreCase("SELF_TYPE")==true)
                        disp_type = current_class;
                return disp_type;
        }
}

@Override public String visitAddSub(coolParser.AddSubContext ctx) {
        String addSub_type="Int";
        if((visit(ctx.expr(0)).equals("Int")==false) || (visit(ctx.expr(1)).equals("Int")==false)){
                System.out.println("ERROR: Can not add/subtract non-integer values.");
                addSub_type="Object";
        }
        return addSub_type;
}

@Override public String visitMulDiv(coolParser.MulDivContext ctx) {
        String mulDiv_type="Int";
```

```
        if((visit(ctx.expr(0)).equals("Int")==false) || (visit(ctx.expr(1)).equals("Int")==false)){
                System.out.println("ERROR: Can not add/subtract non-integer values.");
                mulDiv_type="Object";
        }
        return mulDiv_type;
}
@Override public String visitSwitch(coolParser.SwitchContext ctx) {
        int i=0;
        String switch_type;
        while(ctx.branch(i)!=null){
                String branch_t = visit(ctx.branch(i));
                branch_type.add(branch_t);
                i++;
        }
        //check for duplicate branch
        for(i=0; i<branch_decl_type.size(); i++){
                for(int j=i+1; j<branch_decl_type.size(); j++){
                        if(branch_decl_type.elementAt(i).equals(branch_decl_type.elementAt(j))){
                           System.out.println("ERROR: Duplicate switch case branch type.");
                        }
                }
        }
        //switch type is lub of all branch types
        switch_type = branch_type.firstElement();
        for(i=1; i<branch_type.size(); i++)
                switch_type = lub(switch_type, branch_type.elementAt(i));
```

```
            branch_type.clear();
            branch_decl_type.clear();
            return switch_type;
}
@Override public String visitBranch(coolParser.BranchContext ctx) {
            branch_decl_type.add(ctx.TYPEID().getText());
            objectTable.enterScope();
            objectTable.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
            String branch_type=visit(ctx.expr());
            objectTable.exitScope();
            return branch_type;
}

@Override public String visitIsVoid(coolParser.IsVoidContext ctx) {
            String void_type="Bool";
            visit(ctx.expr());
            return void_type;
}

@Override public String visitLessThan(coolParser.LessThanContext ctx) {
            String type1 = visit(ctx.expr(0));
            String type2 = visit(ctx.expr(1));
            String lt_type="Bool";
            if((type1.equals("Int")==false) || (type2.equals("Int")==false)){
                        System.out.println("ERROR: Can not compare non-integer Values.");
                        lt_type="Object";
            }
```

```
        return lt_type;
}

@Override public String visitObjId(coolParser.ObjIdContext ctx) {
        String obj_type="Object";
        if(ctx.OBJECTID().getText().equalsIgnoreCase("self")){
                obj_type=current_class;
                return obj_type;
        }
        Map<String, String> attrs = new HashMap<String, String>();
        attrs = attributes.get(current_class);
        if((objectTable.lookup(ctx.OBJECTID().getText()))==null){
                if(attrs.containsKey(ctx.OBJECTID().getText())==false){
                System.out.println("ERROR: Undefined object "+ctx.OBJECTID().getText()+".");
                        return obj_type;
                }else{
                        return attrs.get(ctx.OBJECTID().getText());
                }
        }
        return objectTable.lookup(ctx.OBJECTID().getText());
}
@Override public String visitBlock(coolParser.BlockContext ctx) {
        System.out.println("Block");
        String blk_type=visit(ctx.expr());
        return blk_type;
}
```

```
@Override public String visitLessEqual(coolParser.LessEqualContext ctx) {
        String type1 = visit(ctx.expr(0));
        String type2 = visit(ctx.expr(1));
        String le_type="Bool";
        if((type1.equals("Int")==false) || (type2.equals("Int")==false)){
                System.out.println("ERROR: Can not compare non-integer Values.");
                le_type="Object";
        }
        return le_type;
}

@Override public String visitGroup(coolParser.GroupContext ctx) {
        String grp_type = visit(ctx.expr());
        return grp_type;
}

@Override public String visitLet(coolParser.LetContext ctx) {
        objectTable.enterScope();
        String let_type =visit(ctx.let_expr());
        objectTable.exitScope();
        return let_type;
}

@Override public String visitNestedLet(coolParser.NestedLetContext ctx) {
        //OBJECTID COLON TYPEID COMMA let_expr
        objectTable.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
        String nestLet_type=visit(ctx.let_expr());
```

```
        return nestLet_type;
}

@Override public String visitLetIn(coolParser.LetInContext ctx) {
        //OBJECTID COLON TYPEID IN expr
        objectTable.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
        String letIn_type = visit(ctx.expr());
        return letIn_type;
}

@Override public String visitLetAssignLet(coolParser.LetAssignLetContext ctx) {
        //OBJECTID COLON TYPEID ASSIGN expr COMMA let_expr
        objectTable.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
        String init_type=visit(ctx.expr());
        if(is_subtype(init_type, ctx.TYPEID().getText())==false){
                System.out.println("ERROR: Initialization value is not subtype of Type declared.");
        }
        String letAssign_type=visit(ctx.let_expr());
        return letAssign_type;
}

@Override public String visitLetAssignIn(coolParser.LetAssignInContext ctx) {
        //OBJECTID COLON TYPEID ASSIGN expr IN expr
        objectTable.addId(ctx.OBJECTID().getText(), ctx.TYPEID().getText());
        String init_type=visit(ctx.expr(0));
        if(is_subtype(init_type, ctx.TYPEID().getText())==false){
                System.out.println("ERROR: Initialization value is not subtype of Type declared.");
```

```
            }
            String letAssignIn_type=visit(ctx.expr(1));
            return letAssignIn_type;
}

//basic types
@Override public String visitString(coolParser.StringContext ctx) {
            if(strings.containsKey(ctx.STR_CONST().getText())==false){
                        strings.put(ctx.STR_CONST().getText(),"str_const" );
            }
            return "String";
}
@Override public String visitBoolTrue(coolParser.BoolTrueContext ctx) {
            return "Bool";
}
@Override public String visitBoolFalse(coolParser.BoolFalseContext ctx) {
            return "Bool";
}
@Override public String visitInteger(coolParser.IntegerContext ctx) {
            return "Int";
}
}//end of semantPhase2
```

APPENDIX D Code Generator Code

```
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

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

publicclass codeGeneration extends coolBaseVisitor <String>{
        int reg=1, temp=1, object_count=1, scope=0;
        String current_class;
        Map<String, String> stringConstants = new HashMap<String, String>();
        Map<String, String> attributes = new HashMap<String, String>();
        void Init(ParseTree tree, Map<String, String> strings){
        //target info
        System.out.printf("target datalayout = \"e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-
i64:64:64-f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-
S128\"\n");
        System.out.printf("target triple = \"x86_64-unknown-linux-gnu\"\n");

        //declare printf and scanf
        System.out.printf("declare i32 @printf(i8*, ...)\n");
        System.out.printf("declare i32 @__isoc99_scanf(i8*, ...)\n");

        //global definition of all the strings in the program
        Iterator<String> iter_strings = strings.keySet().iterator();
        while(iter_strings.hasNext()){
```

```
                String string = iter_strings.next();
                int len = string.length();
                System.out.printf("@.msg%d = private unnamed_addr constant [%d x i8]
                c\"%s\\0A\\00\"\n", object_count, len+1, string);
                String stringName = "@.msg";
                stringName = stringName.concat(String.valueOf(object_count));
                object_count++;
                stringConstants.put(string, stringName);
        }
        //IO Class has only methods which are built in provided in llvm
        //so direct calls only

        //Object Class
        System.out.printf("%%Object = type { i8* }\n");
        //define set and get functions for typeName

        //Int Class
        System.out.printf("%%Int = type { i32 }\n");
        //define set and get functions
        //setValue
        System.out.printf("define void @Int_setValue(%%Int* %%this, i32 %%value)
nounwind{\n");
        System.out.printf("%%%d = getelementptr %%Int* %%this, i32 0, i32 0\n", reg);
        System.out.printf("store i32 %%value, i32* %%%d", reg);
        System.out.printf("ret void \n}\n");

        //getValue
```

```
        System.out.printf("define i32 @Int_getValue(%%Int* %%this) nounwind {\n");
        System.out.printf("%%%d = getelementptr %%Int* %%this, i32 0, i32 0\n",reg);
        reg++;
        System.out.printf("%%%d = load i32* %%this\n", reg);
        System.out.printf("ret i32 %%%d \n}\n", reg);
        reg=reg-1;

        //String Class //all string constants declared global
        System.out.printf("%%String = type { i8, i32 }\n");

        //Bool Class
        System.out.printf("%%Bool = type { i8 }\n");
        //setValue
    System.out.printf("define void @Bool_setValue(%%Bool* %%this, i8 %%value) nounwind {\n");
        System.out.printf("%%%d = getelementptr %%Bool* %%this, i32 0, i32 0\n",reg);
        System.out.printf("store i8 %%value, i8* %%%d\n",reg);
        System.out.printf("ret void \n}\n");

        //getValue
        System.out.println("define i8 @Bool_getValue(%%Bool* %%this) nounwind {\n");
        System.out.printf("%%%d = getelementptr %%Bool* %%this, i32 0, i32 0\n",reg);
        reg++;
        System.out.printf("%%%d = load i8* %%this\n", reg);
        System.out.printf("ret i8 %%%d \n}\n", reg);
        reg=reg-1;
        visit(tree);
}
```

```
@Override public String visitClass_def(coolParser.Class_defContext ctx) {
        // %<classname> = type { <member attribute types> } future scope
        //only handling Main class
        scope++;
        if(ctx.TYPEID().getText().equalsIgnoreCase("Main")){
                int i=0;
                while(ctx.features(i)!=null){
                        visit(ctx.features(i));
                        i++;
                }
        }
        scope--;
        return "0";
}

@Override public String visitMethod(coolParser.MethodContext ctx) {
        //define methods
        if(ctx.OBJECTID().getText().equalsIgnoreCase("main")){
                System.out.printf("define i32 @main() nounwind uwtable {\nentry:\n");
                visit(ctx.expr());
                System.out.printf("}\n");
        }
        return "0";
}
@Override public String visitAttribute(coolParser.AttributeContext ctx) {
        //allocate space for attributes
```

```
System.out.printf("%%attr%s = alloca %%%s\n", ctx.OBJECTID().getText(),
ctx.TYPEID().getText());
        // if expr exists assign expr value to attribute
        String attr_name = "%attr";
        attr_name = attr_name.concat(ctx.OBJECTID().getText());
        attributes.put(ctx.OBJECTID().getText(), attr_name);
        if(ctx.ASSIGN()!=null){
                String expr_ret= visit(ctx.expr());
                String type = ctx.TYPEID().getText();
                if(type.equals("Int")){
        System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr_ret);
        System.out.printf("call void @Int_setValue(%%Int* %%attr%s, i8* %%%d)\n",
        ctx.OBJECTID().getText(), reg);
                }
                if(type.equals("Bool")){
System.out.printf("%%%d = call i32 @Bool_getValue(%%Bool* %%%s)\n", reg, expr_ret);
System.out.printf("call void @Bool_setValue(%%Bool* %%attr%s, i8* %%%d)\n",
ctx.OBJECTID().getText(), reg);
                }
        }
        reg++;
        return "0";
}


@Override public String visitDynamicDispatch(coolParser.DynamicDispatchContext ctx) {
        //IO class functions handled here
```

```
        if(ctx.OBJECTID().getText().equals("out_string")){
                String StringName = stringConstants.get(ctx.expr(0).getText());
                int len = ctx.expr(0).getText().length();
System.out.printf("call i32 (i8 *, ...)* @printf(i8* getelementptr inbounds ([%d x i8]* %s, i32 0, i32
0))",len, StringName);
        }
        if(ctx.OBJECTID().getText().equals("out_int")){
                String str = String.valueOf(ctx.expr(0).getText());
                int len = str.length();
System.out.printf("call i32 (i8 *, ...)* @printf(i8* getelementptr inbounds ([%d x i8]* \"%d\", i32 0,
i32 0))",len, str);
        }
        return "0";
}

@Override public String visitOpEqual(coolParser.OpEqualContext ctx) {
        //comparison icmpslt
        //following are names of 2 Int objects
        String expr0_name = visit(ctx.expr(0));
        String expr1_name = visit(ctx.expr(1));
        //load value of 1st int in register
        System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr0_name);
        reg++;
        //load value of 2nd int in register
        System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr1_name);
        reg++;
        //do the comparison
```

```
            System.out.printf("%%%d = icmp eq i32 %%%d, %%%d\n", reg, reg-2, reg-1);
            //create a Bool object and return its name
            System.out.printf("%%bool%d = alloca %%Bool\n", object_count);
            //set value
            System.out.printf("call void @Bool_setValue(%%Bool* %%bool%d, i8* %%%d)\n",
object_count, reg);
            reg++;
            String c = String.valueOf(object_count);
            String name = "%bool";
            name = name.concat(c);
            object_count++;
            return name;
}

@Override public String visitOpAssign(coolParser.OpAssignContext ctx) {
            //store value from rhs to lhs
            String expr_name = visit(ctx.expr());
            String obj_id = attributes.get(ctx.OBJECTID().getText());
            System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr_ret);
            System.out.printf("call void @Int_setValue(%%Int* %%attr%s, i8* %%%d)\n",
tx.OBJECTID().getText(), reg);
                        reg++;
            return visitChildren(ctx);
}

@Override public String visitOpNot(coolParser.OpNotContext ctx) {
            //xor with bool true
```

```
        //get the name of bool object
        String expr_name = visit(ctx.expr());
        //get value of the object
        System.out.printf("%%%d = call i8 @Bool_getValue(%%Bool* %%%s)\n", reg,
expr_name);
        //xor with true to flip it
        System.out.printf("%%%d = xor i1 %%%d, true\n", reg, reg);
        //create a Bool object and return its name
        System.out.printf("%%bool%d = alloca %%Bool\n", object_count);
        //set value
        System.out.printf("call void @Bool_setValue(%%Bool* %%bool%d, i8* %%%d)\n",
bject_count, reg);
        reg++;
        String c = String.valueOf(object_count);
        String name = "%bool";
        name = name.concat(c);
        object_count++;
        return name;
}

@Override public String visitWhileLoop(coolParser.WhileLoopContext ctx) {
        //get condition expr object name (bool)
        System.out.printf("loop%d :\n", temp);
        String cond_expr = visit(ctx.expr(0));
        System.out.printf("%%%d = call i8 @Bool_getValue(%%Bool* %%%s)\n", reg,
cond_expr);
        System.out.printf("br i1 %%%d, label start%d, label end%d\n",reg, temp, temp);
```

```
        reg++;
        System.out.printf("start%d :\n", temp);
        visit(ctx.expr(1));
        System.out.printf("br label loop%d\n", temp);
        System.out.printf("end%d :\n", temp);
        temp++;
        return "0";
}

@Override public String visitBlock1(coolParser.Block1Context ctx) {
        //scope++; after visiting all exprs scope--;
        scope++;
        int i=0;
        while(ctx.expr(i)!=null){
                visit(ctx.expr(i));
                i++;
        }
        scope--;
        return "0";
}
@Override public String visitIfThenElse(coolParser.IfThenElseContext ctx) {
        String cond_expr = visit(ctx.expr(0));
        System.out.printf("%%%d = call i8 @Bool_getValue(%%Bool* %%%s)\n", reg,
cond_expr);
        System.out.printf("br i1 %%%d, label %%True%d, label %%False%d\n",reg, temp, temp);
        reg++;
        System.out.printf("True%d :\n", temp);
```

```
            visit(ctx.expr(1));
            System.out.printf("br label end%d\n", temp);
            System.out.printf("False%d :\n", temp);
            visit(ctx.expr(2));
            System.out.printf("end%d :\n", temp);
            temp++;
            return "0";
    }
    @Override public String visitOpComplement(coolParser.OpComplementContext ctx) {
            //subtract from zero
            //get the name of the integer object
            String expr_name = visit(ctx.expr());
            //load value of the int in a register
            System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr_name);
            //subtract from zero
            System.out.printf("%%%d = sub nsw i32 0, %%%d", reg, reg);
            //create an Int object and return its name
            System.out.printf("%%int%d = alloca %%Int\n", object_count);
            //set value
    System.out.printf("call void @Int_setValue(%%Int* %%int%d, i32* %%%d)\n", object_count, eg);
            reg++;
            String c = String.valueOf(object_count);
            String name = "%int";
            name = name.concat(c);
            object_count++;
            return name;
    }
```

```
@Override public String visitAddSub(coolParser.AddSubContext ctx) {
        //add or sub
        //following are names of 2 Int objects
        String expr0_name = visit(ctx.expr(0));
        String expr1_name = visit(ctx.expr(1));
        //load value of 1st int in register
        System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr0_name);
        reg++;
        //load value of 2nd int in register
        System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr1_name);
        reg++;
        //do addition or subtraction
        if(ctx.ADD()!=null){
                System.out.printf("%%%d = add i32 %%%d, %%%d\n", reg, reg-2, reg-1);
        }else{
                System.out.printf("%%%d = sub i32 %%%d, %%%d\n", reg, reg-2, reg-1);
        }
        //create an Int object and return its name
        System.out.printf("%%int%d = alloca %%Int\n", object_count);
        //set value
System.out.printf("call void @Int_setValue(%%Int* %%int%d, i32* %%%d)\n", object_count, reg);

        reg++;
        String c = String.valueOf(object_count);
        String name = "%int";
```

```
        name = name.concat(c);
        object_count++;
        return name;
}

@Override public String visitMulDiv(coolParser.MulDivContext ctx) {
        //add or sub
        //following are names of 2 Int objects
        String expr0_name = visit(ctx.expr(0));
        String expr1_name = visit(ctx.expr(1));
        //load value of 1st int in register
        System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr0_name);
        reg++;
        //load value of 2nd int in register
        System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr1_name);
        reg++;
        //do addition or subtraction
        if(ctx.MUL()!=null){
                System.out.printf("%%%d = mul i32 %%%d, %%%d\n", reg, reg-2, reg-1);
        }else{
                System.out.printf("%%%d = sdiv i32 %%%d, %%%d\n", reg, reg-2, reg-1);
        }
        //create an Int object and return its name
        System.out.printf("%%int%d = alloca %%Int\n", object_count);
        //set value
        System.out.printf("call void @Int_setValue(%%Int* %%int%d, i32* %%%d)\n",
object_count, eg);
```

```
            reg++;
            String c = String.valueOf(object_count);
            String name = "%int";
            name = name.concat(c);
            object_count++;
            return name;
}

@Override public String visitLessThan(coolParser.LessThanContext ctx) {
            //comparison icmpslt
            //following are names of 2 Int objects
            String expr0_name = visit(ctx.expr(0));
            String expr1_name = visit(ctx.expr(1));
            //load value of 1st int in register
            System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr0_name);
            reg++;
            //load value of 2nd int in register
            System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr1_name);
            reg++;
            //do the comparison
            System.out.printf("%%%d = icmp slt i32 %%%d, %%%d\n", reg, reg-2, reg-1);
            //create a Bool object and return its name
            System.out.printf("%%bool%d = alloca %%Bool\n", object_count);
            //set value
System.out.printf("call void @Bool_setValue(%%Bool* %%bool%d, i8* %%%d)\n", object_count,
reg);
            reg++;
```

```
            String c = String.valueOf(object_count);
            String name = "%bool";
            name = name.concat(c);
            object_count++;
            return name;
}

@Override public String visitObjId(coolParser.ObjIdContext ctx) {
            //load objectId value in a register
            String obj_name;
            obj_name = attributes.get(ctx.OBJECTID().getText());
            return obj_name;
}

@Override public String visitBlock(coolParser.BlockContext ctx) {
            //scope++; after visiting all exprs scope--;
            scope++;
            visit(ctx.expr());
            scope--;
            return "0";
}

@Override public String visitLessEqual(coolParser.LessEqualContext ctx) {
            //comparison icmpslt
            //following are names of 2 Int objects
            String expr0_name = visit(ctx.expr(0));
            String expr1_name = visit(ctx.expr(1));
```

```
            //load value of 1st int in register
System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr0_name);
            reg++;
            //load value of 2nd int in register
            System.out.printf("%%%d = call i32 @Int_getValue(%%Int* %%%s)\n", reg, expr1_name);
            reg++;
            //do the comparison
            System.out.printf("%%%d = icmp sle i32 %%%d, %%%d\n", reg, reg-2, reg-1);

            //create a Bool object and return its name
            System.out.printf("%%bool%d = alloca %%Bool\n", object_count);
            //set value
            System.out.printf("call void @Bool_setValue(%%Bool* %%bool%d, i8* %%%d)\n",
bject_count, reg);
            reg++;

            //bitcast to convert derived class to base
            System.out.printf("%%%d = bitcast %%Bool* %%this to %%Object\n", reg);
            reg++;
            //set type_name
            System.out.printf(" %%%d = getelementptr [5 x i8]* @.bool, i32 0, i32 0");
System.out.printf("call void @Obj_setType(%%Object* %%%d, i8* %%%d)\n", reg-1, reg);
            reg++;

            String c = String.valueOf(object_count);
            String name = "%bool";
            name = name.concat(c);
```

```
        object_count++;
        return name;
}

        @Override public String visitGroup(coolParser.GroupContext ctx) {
                //visit expr
        return visit(ctx.expr());
}
//basic types
@Override public String visitInteger(coolParser.IntegerContext ctx) {
        //create Int object
        System.out.printf("%%int%d = alloca %%Int\n", object_count);
        System.out.printf("%%value%d = add i32 %s, 0\n",object_count,
ctx.INT_CONST().getText());
        System.out.printf("call void @Int_setValue(%%Int* %%int%d, i32* %%value%d)\n",
        object_count, object_count);

        //bitcast to convert derived class to base
        System.out.printf("%%%d = bitcast %%Int* %%this to %%Object\n", reg);
        reg++;
        //set type_name
        System.out.printf(" %%%d = getelementptr [4 x i8]* @.int, i32 0, i32 0");
        System.out.printf("call void @Obj_setType(%%Object* %%%d, i8* %%%d)\n", reg-1,
reg);
        reg++;

        String c = String.valueOf(object_count);
```

```
        String name = "%int";
        name = name.concat(c);
        object_count++;
        return name;
}

@Override public String visitBoolTrue(coolParser.BoolTrueContext ctx) {
        //create Bool object
        System.out.printf("%%bool%d = alloca %%Bool\n", object_count);
        //register to store true
        System.out.printf("%%value%d = alloca i8, align 1\n", object_count);
        System.out.printf("store i8 1, i8* %%value%d, align 1\n", object_count);
        //set value
        System.out.printf("call void @Bool_setValue(%%Bool* %%bool%d, i8* %%value%d)\n",
        object_count, object_count);
        //bitcast to convert derived class to base
        System.out.printf("%%%d = bitcast %%Bool* %%this to %%Object\n", reg);
        reg++;
        //set type_name
        System.out.printf(" %%%d = getelementptr [5 x i8]* @.bool, i32 0, i32 0");
System.out.printf("call void @Obj_setType(%%Object* %%%d, i8* %%%d)\n", reg-1, reg);
        reg++;
        String c = String.valueOf(object_count);
        String name = "%bool";
        name = name.concat(c);
        object_count++;
        return name;
```

```
@Override public String visitBoolFalse(coolParser.BoolFalseContext ctx) {
        //create Bool object
        System.out.printf("%%bool%d = alloca %%Bool\n", object_count);
        //register to store false
        System.out.printf("%%value%d = alloca i8, align 1\n", object_count);
        System.out.printf("store i8 0, i8* %%value%d, align 1\n", object_count);
        //set value
        System.out.printf("call void @Bool_setValue(%%Bool* %%bool%d, i8* %%value%d)\n",
object_cunt, object_count);
        //bitcast to convert derived class to base
        System.out.printf("%%%d = bitcast %%Bool* %%this to %%Object\n", reg);
        reg++;
        //set type_name
        System.out.printf(" %%%d = getelementptr [5 x i8]* @.bool, i32 0, i32 0");
System.out.printf("call void @Obj_setType(%%Object* %%%d, i8* %%%d)\n", reg-1, reg);
        reg++;
        String c = String.valueOf(object_count);
        String name = "%bool";
        name = name.concat(c);
        object_count++;
        return name;
}
@Override public String visitString(coolParser.StringContext ctx) {
        return "0";
}
}//end of codeGeneration
```

APPENDIX E Driver code

```
import java.io.FileInputStream;
import java.io.InputStream;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
import org.antlr.v4.runtime.tree.pattern.ParseTreePattern;
public class semanticAnalyzer{
        public static void main(String args[]) throws Exception{
                String inputFile = null;
                if ( args.length>0 ) inputFile = args[0];
                InputStream is = System.in;
                if ( inputFile!=null ) is = new FileInputStream(inputFile);
                    ANTLRInputStream input = new ANTLRInputStream(is);
                  coolLexer lexer = new coolLexer(input);
                   CommonTokenStream tokens = new CommonTokenStream(lexer);
                   coolParser parser = new coolParser(tokens);
                   ParseTree ast = parser.program();
           ParseTreeWalker walker = new ParseTreeWalker();
           semantPhase1 def = new semantPhase1();
           walker.walk(def, ast);
           semantPhase2 typecheck = new semantPhase2();
           typecheck.Init(ast, def.class_table, def.method_table, def.attr_table);
                   codeGeneration codeGen = new codeGeneration();
           codeGen.Init(ast, typecheck.strings);
        }
}
```

REFERENCES

1. T. Parr. *The Definitive ANTLR 4 Reference*. 2nd ed. Pragmatic Bookshelf. 2013.
2. A. Aiken. Stanford University Compilers CS 1course [Online] January 5, 2015. Available at: https://lagunita.stanford.edu/courses/Engineering/Compilers/Fall2014/about. Accessed on: May 22, 2015.
3. A. Aiken. Stanford University Compilers course – COOL manual [Online] January 5, 2015. Available at: https://theory.stanford.edu/~aiken/software/cool/cool-manual.pdf. Accessed on: September 10, 2015.