

***Report on***

# “Building a mini-compiler based on C++”

*Submitted in partial fulfilment of the requirements for* ***Sem VI***

***Compiler Design Laboratory***

# Bachelor of Technology in

**Computer Science & Engineering**

## Dweepa Honnavalli Ishaan Lodha

## Lakshana Kolur

***Submitted by:***

## 01FB16ECS138

**01FB16ECS142**

**01FB16ECS175**

*Under the*

*guidance of*

## Prakash C O

### Assistant Professor PES University, Bengaluru

#### January – May 2019

**DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING**

FACULTY OF ENGINEERING

#### PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013) 100ft Ring Road, Bengaluru – 560 085, Karnataka, India

# TABLE OF CONTENTS

|  |  |  |
| --- | --- | --- |
| **Chapter No.** | **Title** | **Page No.** |
| **1.** | **INTRODUCTION** | **01** |
| **2.** | **ARCHITECTURE OF LANGUAGE** | **07** |
| **3.** | **REFERENCES** | **07** |
| **4.** | **CONTEXT-FREE GRAMMAR** | **07** |
| **5.** | **DESIGN STRATEGY**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING** | **08** |
| **6.** | **IMPLEMENTATION DETAILS**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING** | **09** |
| **7.** | **RESULTS** | **10** |
| **8.** | **SNAPSHOTS** | **10** |
| **9.** | **CONCLUSIONS** | **15** |
| **10.** | **FURTHER ENHANCEMENTS** | **15** |

## INTRODUCTION

Simple constructs from the language JAVA were implemented. The frontend of the compiler including Symbol table generation, Abstract Syntax tree construction, Intermediate Code generation and Code Optimization was implemented using flex and bison.

**SAMPLE INPUT AND EXPECTED OUTPUT:**

INPUT PROGRAM:

class Test

{

//Comment 1

/\* multi line comment\*/

int b;

public static int main()

{

int a=5;

b=1;

switch(a)

{

case 1: a = 2; break;

case 5: a = 3; while (a>0) { b = b\*9; a=a-1;}

default: a=1;

}

int c=3;

b = a+c;

}

int c=9;

}

Symbol Table:

Final Symbol table

Name Value Type Scope Occurrence

b 4 int 0 1

a 2 int 1 1

c 3 int 1 1

d 0 int 1 1

c 0 int 0 1

AST:

───SEQ

├──SEQ

│ ├──SEQ

│ │ ├──(class,Test)

│ │ │

│ │ └──DECL

│ │ ├──(keyword, int)

│ │ └──=

│ │ ├──(id, b)

│ │ └──(num, 0)

│ SEQ

│ ├─SEQ

│ │ ├──(modifier1,public)

│ │ │

│ │ └──SEQ

│ │ ├──(modifier2, static)

│ │ └──FUNC

│ │ └──(int, main)

│ │

│ │

│ │

│ └──SEQ

│ ├──SEQ

│ │ ├──SEQ

│ │ │ ├──SEQ

│ │ │ │ ├──SEQ

│ │ │ │ │ ├──SEQ

│ │ │ │ │ │ ├──SEQ

│ │ │ │ │ │ │ ├──SEQ

│ │ │ │ │ │ │ │ ├──SEQ

│ │ │ │ │ │ │ │ │ ├──SEQ

│ │ │ │ │ │ │ │ │ │ ├──DECL

│ │ │ │ │ │ │ │ │ │ │ ├──(keyword, int)

│ │ │ │ │ │ │ │ │ │ │ └──,

│ │ │ │ │ │ │ │ │ │ │ ├──,

│ │ │ │ │ │ │ │ │ │ │ │ ├──=

│ │ │ │ │ │ │ │ │ │ │ │ │ ├──(id, a)

│ │ │ │ │ │ │ │ │ │ │ │ │ └──(num, 6)

│ │ │ │ │ │ │ │ │ │ │ │ └──=

│ │ │ │ │ │ │ │ │ │ │ │ ├──(id, c)

│ │ │ │ │ │ │ │ │ │ │ │ └──(num, 7)

│ │ │ │ │ │ │ │ │ │ │ └──=

│ │ │ │ │ │ │ │ │ │ │ ├──(id, b)

│ │ │ │ │ │ │ │ │ │ │ └──(num, 3)

│ │ │ │ │ │ │ │ │ │ └──=

│ │ │ │ │ │ │ │ │ │ ├──(id, a)

│ │ │ │ │ │ │ │ │ │ └──(num, 5)

│ │ │ │ │ │ │ │ │ └──=

│ │ │ │ │ │ │ │ │ ├──(id, c)

│ │ │ │ │ │ │ │ │ └──/

│ │ │ │ │ │ │ │ │ ├──(num, 6)

│ │ │ │ │ │ │ │ │ └──(num, 7)

│ │ │ │ │ │ │ │ └──=

│ │ │ │ │ │ │ │ ├──(id, d)

│ │ │ │ │ │ │ │ └──(num, 7)

│ │ │ │ │ │ │ └──WHILE

│ │ │ │ │ │ │ ├──SEQ

│ │ │ │ │ │ │ │ ├──(>)

│ │ │ │ │ │ │ │ │ │

│ │ │ │ │ │ │ │ │ &&

│ │ │ │ │ │ │ │ │ └─(>)

│ │ │ │ │ │ │ │ │ ├──(id, c)

│ │ │ │ │ │ │ │ │ └──(id, d)

│ │ │ │ │ │ │ ├──>

│ │ │ │ │ │ │ │ ├──(id, a)

│ │ │ │ │ │ │ │ └──(id, b)

│ │ │ │ │ │ │ └──SEQ

│ │ │ │ │ │ │ ├──SEQ

│ │ │ │ │ │ │ │ ├──SEQ

│ │ │ │ │ │ │ │ │ ├──SEQ

│ │ │ │ │ │ │ │ │ │ ├──=

│ │ │ │ │ │ │ │ │ │ │ ├──(id, a)

│ │ │ │ │ │ │ │ │ │ │ └──(num, 7)

│ │ │ │ │ │ │ │ │ │ └──=

│ │ │ │ │ │ │ │ │ │ ├──(id, c)

│ │ │ │ │ │ │ │ │ │ └──(num, 3)

│ │ │ │ │ │ │ │ │ └──WHILE

│ │ │ │ │ │ │ │ │ ├──(a>b)

│ │ │ │ │ │ │ │ │ ├──>

│ │ │ │ │ │ │ │ │ │ ├──(id, a)

│ │ │ │ │ │ │ │ │ │ └──(id, b)

│ │ │ │ │ │ │ │ │ └──SEQ

│ │ │ │ │ │ │ │ │ ├──=

│ │ │ │ │ │ │ │ │ │ ├──(id, a)

│ │ │ │ │ │ │ │ │ │ └──(num, 7)

│ │ │ │ │ │ │ │ │ └──=

│ │ │ │ │ │ │ │ │ ├──(id, c)

│ │ │ │ │ │ │ │ │ └──(num, 3)

│ │ │ │ │ │ │ │ └──(keyword, continue)

│ │ │ │ │ │ │ └──=

│ │ │ │ │ │ │ ├──(id, d)

│ │ │ │ │ │ │ └──(num, 0)

│ │ │ │ │ │ └──=

│ │ │ │ │ │ ├──(id, a)

│ │ │ │ │ │ └──(num, 5)

│ │ │ │ │ └──SWITCH

│ │ │ │ │ ├──+

│ │ │ │ │ │ ├──(id, c)

│ │ │ │ │ │ └──(num, 5)

│ │ │ │ │ └──CASE

│ │ │ │ │ ├──(num, 5)

│ │ │ │ │ └──=

│ │ │ │ │ ├──(id, a)

│ │ │ │ │ └──(num, 0)

│ │ │ │ └──(keyword, break)

│ │ │ └──CASE

│ │ │ ├──(num, 6)

│ │ │ └──=

│ │ │ ├──(id, a)

│ │ │ └──(num, 1)

│ │ └──(keyword, break)

│ └──DEFAULT

│ └──=

│ ├──(id, c)

│ └──(num, 5)

└──DECL

├──(keyword, int)

└──(id, c)

**ICG:**

start

b = 3

c = 7

a = 6

a = 5

T1 = 6 / 7

T2 = c - T1

c = T2

d = 7

L1:T3 = c > d

if T3 go to L2

go to L4

L4:T4 = a > b

if T4 go to L2

go to L3

L2:a = 7

c = 3

L5:T5 = a > b

if T5 go to L6

go to L7

L6:a = 7

c = 3

go to L5

L7:go to L1

d = 0

go to L1

L3:a = 5

T6 = c + 5

T7 = 5 == L9

if T7 go to L9

go to L10

L9:a = 0

go to L8

go to L11

L10:T8 = 6 == L11

if T8 go to L11

go to L12

L11:a = 1

go to L8

go to L13

L12:a = 2

L8:L13:c = 5

T9 = 6 \* 4

T10 = 5 \* 4

T11 = T10 + T9

c = a[T11]

stop

**OPTIMISED ICG:**

start

b = 3

c = 7

a = 6

a = 5

T1 = 0

T2 = 7

c = 7

d = 7

L1:

T3 = 0

if T3 go to L2

go to L4

L4:

T4 = 1

if T4 go to L2

go to L3

L2:

a = 7

c = 3

L5:

T5 = 1

if T5 go to L6

go to L7

L6:

a = 7

c = 3

go to L5

L7:

go to L1

d = 0

go to L1

L3:

a = 5

T6 = 8

T7 = 0

if T7 go to L9

go to L10

L9:

a = 0

go to L8

go to L11

L10:

T8 = 0

if T8 go to L11

go to L12

L11:

a = 1

go to L8

go to L13

L12:

a = 2

L8:

L13:

c = 5

T9 = 24

T10 = 20

T11 = 44

c = a[T11]

stop

## ARCHITECTURE OF LANGUAGE

Compiler for the following constructs:

* while loop
* switch construct
* int data type
* float data type
* char data type
* return, break, continue statements
* modifiers and function calls
* Arrays
* Arithmetic and logical operators
* Comments

## REFERENCES

#### Lex Yacc and its internal working

<https://www.tldp.org/HOWTO/Lex-YACC-HOWTO.html#toc1>

#### Building a mini-compiler - tutorial

<https://www.tutorialspoint.com/compiler_design/index.htm>

#### Expression evaluation using Abstract Syntax Tree

<https://mariusbancila.ro/blog/2009/02/03/evaluating-expressions-part-1>

1. **CONTEXT-FREE GRAMMAR**

**CompilationUnit:**

ImportDec CompilationUnit

|ClassDec

;

**ImportDec:**

T\_IMPORT T\_ID'.'T\_ID'.''\*'';'

;

**ClassDec:**

Modifier T\_CLASS T\_ID '{'ClassBody'}'

;

**ClassBody:**

GlobVarDec ClassBody

|MethodDec ClassBody

|

;

**GlobVarDec:**

Modifier Type DecIdenList';'

;

**MethodDec:**

Modifier Type T\_ID'('Params')'Block

;

**Modifier:**

T\_PUBLIC Modifier1

|T\_PRIVATE Modifier1

|T\_PROTECTED Modifier1

|Modifier1

;

**Modifier1:**

T\_STATIC Modifier2

;

**Modifier2:**

T\_FINAL

|

;

**Params:**

ParamList

;

**ParamList:**

Type T\_ID

|Type T\_ID',' Params

|

;

**Block:**

'{'Statement'}'

**Statement:**

Assign Statement

|T\_BREAK';' Statement

|T\_CONTINUE';' Statement

|T\_WHILE'('Expression')'Block

|T\_WHILE'('Expression')'Statement

|T\_RETURN Expression';'

Statement

|T\_SWITCH'('Expression')' '{'SwitchBlock'}' Statement

|VarDec Statement

|ArrDec Statement

|ArrInit Statement

|

;

**SwitchBlock:**

SwitchLabel':' Statement SwitchBlock

|

;

**SwitchLabel:**

T\_CASE '('Expression')'

| T\_DEFAULT

;

**VarDec:**

Type T\_ID '=' Expression **DecIdenList';'**

|Type T\_ID DecIdenList';'

;

**DecIdenList:**

','T\_ID '=' Expression DecIdenList

|','T\_ID DecIdenList

|

;

**ArrDec:**

T\_INT T\_ID'['']' ArrList';'

|T\_ID'['']''['']' ArrList;

|T\_ID ArrList;

;

**ArrList:**

','T\_ID'['']'

|','T\_ID'['']''['']'

|','T\_ID

|

;

**ArrInit:**

T\_INT T\_ID '['']' '=' T\_NEW T\_INT'['Expression']'';'

|T\_INT T\_ID '['']''['']' '=' T\_NEW T\_INT'['Expression']''['Expression']'';'

|T\_INT'['']' T\_ID '=' T\_NEW T\_INT'['Expression']'';'

|T\_INT'['']''['']' T\_ID '=' T\_NEW T\_INT'['Expression']''['Expression']'';'

|T\_ID '=' T\_NEW T\_INT'['Expression']'';'

|T\_ID '=' T\_NEW T\_INT'['Expression']''['Expression']'';'

;

**ArrExp:**

T\_ID

|T\_ID'['Expression']'

|T\_ID'['Expression']''['Expression]'

;

**Type:**

T\_INT

|T\_DOUBLE

|T\_CHAR

;

**Assign:**

ArrExp AssignOp Expression';'

**AssignOp:**

'='

|T\_SHORTHANDADD

|T\_SHORTHANDSUB

|T\_SHORTHANDMUL

|T\_SHORTHANDDIV

|T\_SHORTHANDAND

|T\_SHORTHANDORE

|T\_SHORTHANDCAR

|T\_SHORTHANDMOD

;

**InfixOp:**

T\_OROR

|T\_ANDAND

|'|'

|'^'

|'&'

|T\_EQCOMP

|T\_NOTEQUAL

|'<'

|'>'

|T\_LESSEREQ

|T\_GREATEREQ

|T\_LEFTSHIFT

|T\_RIGHTSHIFT

|'+'

|'-'

|'\*'

|'/'

|'%'

;

**PrefixOp:**

T\_INC

|T\_DEC

|'!'

|'~'

|'+'

|'-'

;

**PostfixOp:**

T\_INC

|T\_DEC

;

**Expression:**

PrefixOp Expression PostfixOp InfixOp PrefixOp Expression PostfixOp';'

|'('Expression')'

|ArrExp

|T\_NUM

;

## DESIGN STRATEGY

* + **Symbol table creation-** The symbol table was implemented using a linked list with entries as array an structure that contains the identifier, scope, type and its value.
  + **Abstract Syntax Tree-** This tree is constructed as the input is parsed. Each node of this tree contains a pointer to left, a pointer to right and a member for a string.
  + **Intermediate Code Generation-** Intermediate code was generated that makes use of temporary variables and labels. Also all if-else statements were optimized to ifFalse statements to reduce the number of goto statements (an additional optimization provided).
  + **Code Optimization-** Constant folding and Constant propagation were implemented as part of machine independent code optimization.

#### Constant Folding

When an arithmetic expression is encountered, we check to see if all the operands contain digits and are not identifiers. If all the operands are numbers we evaluate the expression.

#### Constant Propagation

When an identifier is encountered, we check the symbol table to see if an entry exists. If the entry exists we perform constant propagation.

* + **Error handling-** Type error and semicolon missing error have been handled

## IMPLEMENTATION DETAILS

Lex and Yacc were used to implement the following:

* + **Symbol table creation-** Implemented in sym.y

The symbol table is a linear array of the following structure

typedef struct **symbol\_table**

{

NODE\* head;

int entries;

}

**TABLE;**

typedef struct **entry\_node**

{

char name[10];

int value;

char type[10];

int scope;

struct NODE\* next;

}

**NODE**;

* + **Abstract Syntax Tree-** Implemented in ASTgen.y

To implement this in lex yacc, we first redefine the YYSTYPE in the yacc file that defaults to int. We create a node structure as follows:

typedef struct **tree**

{

char opr[100];

char value[100];

struct tree\* c1;

struct tree\* c2;

struct tree\* c3;

struct tree\* c4;

}

**TREE**;

typedef struct **ast**

{

TREE\* root;

}

**AST**;

* + **Intermediate Code Generation-** Implemented in ICG.y The given code was converted into 3 address code
  + **Code Optimization-** Implemented in optimicons.y

#### Constant Folding

#### Constant folding is the process of recognizing and evaluating [constant](https://en.wikipedia.org/wiki/Constant_(programming)) expressions at [compile time](https://en.wikipedia.org/wiki/Compile_time) rather than computing them at runtime. Terms in constant expressions are typically simple literals, such as the [integer literal](https://en.wikipedia.org/wiki/Integer_literal) 2, but they may also be variables whose values are known at compile time.

#### This is done using the below function in our code:

#### char\* calculate(char\* opr,char\* op1,char\* op2)

#### Constant Propagation

#### Constant propagation is the process of substituting the values of known constants in expressions at compile time. Such constants include those defined above, as well as [intrinsic functions](https://en.wikipedia.org/wiki/Intrinsic_function) applied to constant values.

#### This is done with the help of the symbol table which has the following structure and the following functions:

typedef struct **symbol\_table\_node**

{

char name[30];

char value[30];

}**NODE;**

#### void add\_or\_update(char\* name,char\* value)

#### char\* getVal(char\* name)

* + **Error Handling-** Implemented in sym.y using conditional statements for type error and semicolon missing is checked using the grammar and unused variables.

## RESULTS

A mini compiler that can compile the chosen constructs was obtained.

## SNAPSHOTS

## CONCLUSIONS

### A compiler for JAVA was thus created using lex and yacc. In addition to the constructs specified, basic building blocks of the language (declaration statements, assignment statements, etc) were handled.

This compiler was built keeping the various stages of Compiler Design, ie, Lexical Analysis, Syntax Analysis, Semantic Analysis and Code Optimisation in mind.

As a part of each stage, an auxillary part of the compiler was built (Symbol Table, Abstract Syntax Tree and Intermediate Code). Each of these components are required to compile code successfully.

In addition to this, basic error handling has also been implemented.

Through this process, all kinds of syntax errors and certain semantic errors in a JAVA program can be caught by the compiler.

1. **FURTHER ENHANCEMENTS**

* Functionality for for and other flavours of while can be implemented.
* The compiler can be constructed to recover from more kinds of errors