

Report on

"Building a Java Mini Compiler"

Submitted in partial fulfillment of the requirements for Sem VI

Compiler Design Laboratory

Bachelor of Technology in Computer Science & Engineering

Submitted by:

Shreya Sri PES1201800132 Vibha Kurpad PES1201800158 Mahah Sadique PES1201801529

Under the guidance of

Prof. Preet Kanwal
Assistant Professor
PES University, Bengaluru

January - May 2021

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	3
2.	ARCHITECTURE OF LANGUAGE:	11
3.	LITERATURE SURVEY	12
4.	CONTEXT FREE GRAMMAR	13
5.	DESIGN STRATEGY	18
6.	IMPLEMENTATION DETAILS	19
7.	RESULTS	22
8.	SNAPSHOTS	22
9.	CONCLUSIONS	27
10.	FURTHER ENHANCEMENTS	28
REFERENCES/BIBLIOGRAPHY		

1. INTRODUCTION

The language we have chosen is the Java programming language. We have implemented the front end of the compiler for Java using Flex and Bison and have handled the following constructs.

- If construct
- If-Else construct
- For Loop

The input to the project are .java class files, and the output is the final optimised intermediate code. 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:

```
/* comment line */
//another comment
public class test {
     public static void main (String[] args) {
     int a = 12/3;
     char cr = 'a';
     float fl = 6.7;
     if (true) {
           int ff = 555;
           int y = 99;
           int c=5+10;
           ff = 5+y;
           int oo =444;
           if(a>=2) {
                 a=10;
           }
           else {
                 a = 8;
           }
     }
     else {
           int ff = 10;
     int fact = 1;
     for ( int i =1; i < 5; i++)
      {
```

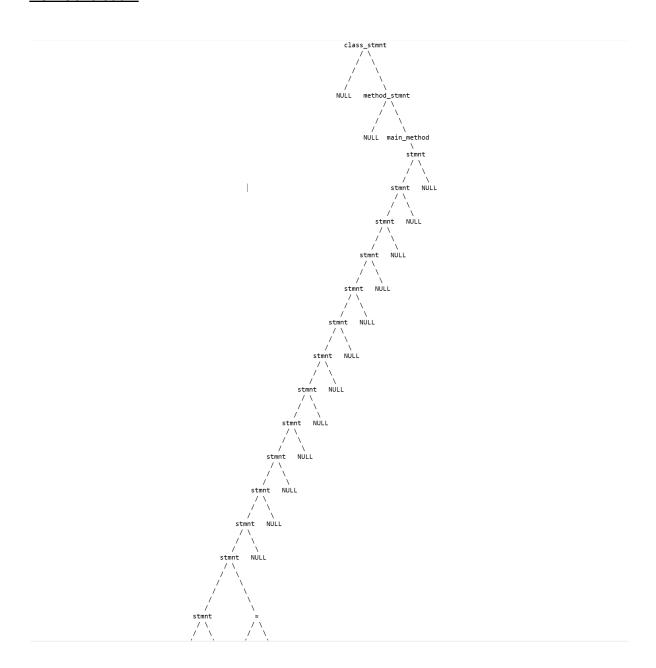
```
fact = fact * i;
    int dummy = 10;
}
int R = 10;
int z = a + 10;
String str = "string";
System.out.print(a);
System.out.print(cr);
System.out.print("hello world");
System.out.print("1234");
System.out.print(fact);
System.out.print(z);
}
```

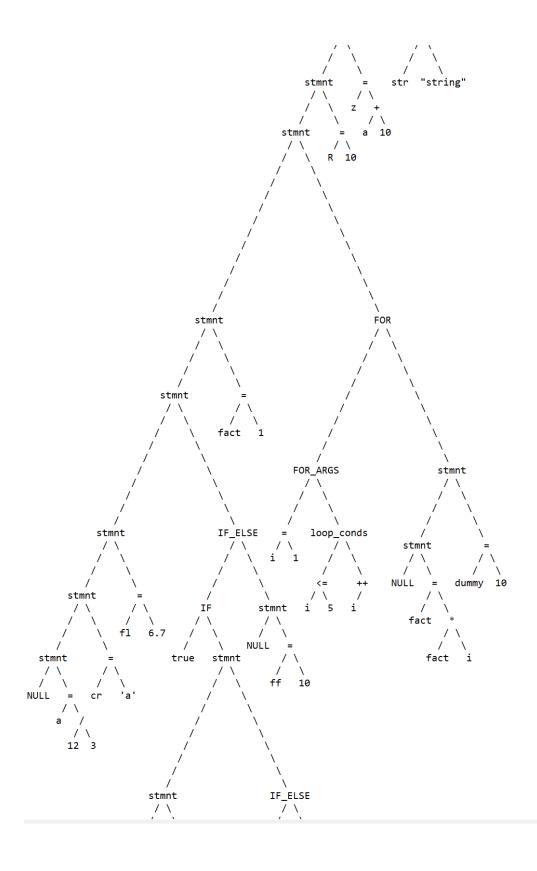
Symbol Table Output:

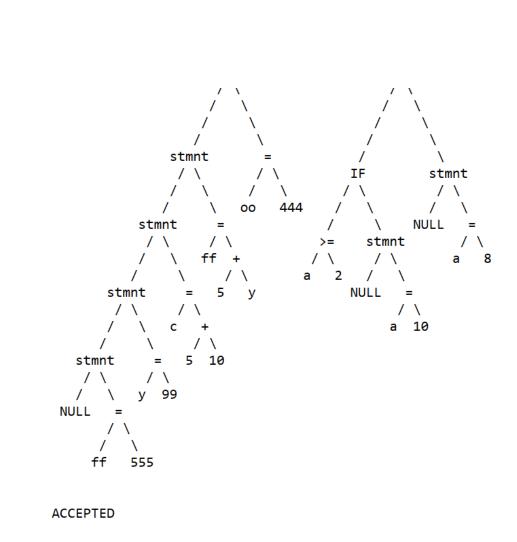
Type	Name	Value	Scope	line	Size
int	a	8	1	6	4
char	cr	a	1	7	2
float	l fl	6.700000	1	8	4
int	ff	104	2	10	4
int	Ìу	99	2	11	4
int	C	15	2	12	4
int	00	444	2	14	4
int	ff	10	2	29	4
int	fact	2	<u> </u>	31	4
int	į i	2	2	32	4
int	dummy	10	3	35	4
int	R	10	į 1	37	4
int	z	18	<u> </u>	38	4
string	str	string	1	39	12

ACCEPTED

AST Generation:







ACCEPTED

Intermediate Code:

```
t0 = 12 / 3
a = t0
cr = 'a'
fl = 6.7
if(not true) goto L1
ff = 555
y = 99
t1 = 5 + 10
c = t1
t2 = 5 + y
ff = t2
00 = 444
t3 = a >= 2
if(not t3) goto L2
a = 10
goto L3
L2:
a = 8
L3:
goto L4
L1:
ff = 10
L4:
fact = 1
i = 1
L5:
t4 = i <= 5
if t4 goto L6
goto L8
L7:
t5 = i + 1
i = t5
goto L5
L6:
t6 = fact * i
fact = t6
dummy = 10
goto L7
L8:
R = 10
t7 = a + 10
```

z = t7
str = "string"
Line No. is 47
ACCEPTED

Intermediate Code in quadruples format:

#	op	arg1	arg2	result
0	/	12	3	t0
1	=	t0		a
2	=	'a'		cr
3	=	6.7		fl
4	iffalse	true		L1
5	=	555		ff
6	=	99	1.0	У
7	+	5	10	t1
8 9	=	t1		C
10	+ =	5 t2	У	t2 ff
11	=	444		00
12	>=	a	2	t3
13	iffalse	t3	2	L2
14	=	10		a
15	goto			L3
16	label			L2
17	=	8		a
18	label			L3
19	goto			L4
20	label			L1
21	=	10		ff
22	label			L4
23	=	1		fact
24	=	1		i
25	label		_	L5
26 27	<= if	i t4	5	t4
28	goto	L4		L6 L8
29	label			L7
30	+	i	1	t5
31	=	t5	_	i
32	goto			L5
33	label			L6
34	*	fact	i	t6
35	=	t6		fact

36	=	10		dummy
37	goto			L7
38	label			L8
39	=	10		R
40	+	a	10	t7
41	=	t7		Z
42	=	"string"		str

2. ARCHITECTURE OF LANGUAGE

Compiler for the following constructs is created:

- for loop
- if, if-else construct
- int data type
- float data type
- char data type
- String data type
- import statements
- return, break, continue statements
- modifiers and function calls
- Arrays
- Arithmetic and logical operators
- Comments (both single line and multiline comments)
- implicit conversion from int/char to float

We show errors for the following

- Redefinition of identifiers within the same scope
- Use of undeclared identifiers
- Unterminated comments
- If length of identifier exceeds maximum limit
- Syntax errors

All the errors and warnings are displayed along with line number

3. LITERATURE SURVEY

Lex Yacc and its internal working

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

Compiler Design fundamentals

https://web.stanford.edu/class/cs143/lectures/lecture01.pdf

Java Grammar

https://docs.oracle.com/javase/specs/jls/se7/html/jls-2.html

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

4. CONTEXT FREE GRAMMAR

```
s:
     compilation unit
compilation unit:
     package_statement import_statement class_stmt
package statement:
     T PACKAGE T IDENTIFIER T SEMC
import statement:
     T_IMPORT package_name T_DOT T_MUL T_SEMC
     | T IMPORT package name T DOT T IDENTIFIER T SEMC
     | T IMPORT T MUL T SEMC
package_name:
     T IDENTIFIER T DOT T IDENTIFIER
class stmt:
     modifier T_CLASS T_IDENTIFIER T_OF stmnt method_stmnt T_CF
modifier:
     T PUBLIC
     T PRIVATE
     | T PROTECTED
method stmnt:
     other method main method
other method:
```

```
stmnt T CF
     | other method T STATIC T VOID T IDENTIFIER T OC T CC T OF
     stmnt T CF
     ;
main method:
     T PUBLIC T STATIC T VOID T MAIN T OC T STRING T OS T CS
     T ARGS T CC T OF stmnt T CF
stmnt:
     stmnt s1
s1:
     variable declaration T SEMC
     | expression T SEMC
     | if else
     |for_stmt
     |print stmnt T SEMC
     | T SEMC
variable declaration:
     dtypes
      ;
dtypes:
     T INT ids1
     | T FLOAT ids2
     | T CHAR ids3
     | T STRING ids4
     | array
array:
     T INT T OS T CS T IDENTIFIER T ASSIGN T NEW T INT T OS T NUM
     T CS | T FLOAT T OS T CS T IDENTIFIER T ASSIGN T NEW
     T FLOAT T OS T NUM T CS
ids1:
     T IDENTIFIER T ASSIGN arithm e
     | ids1 T_COMMA T_IDENTIFIER
     | T IDENTIFIER
```

other method T PUBLIC T VOID T IDENTIFIER T OC T CC T OF

```
;
ids2:
     T IDENTIFIER T ASSIGN float e
     | ids2 T COMMA T IDENTIFIER
     | T IDENTIFIER
ids3:
     T IDENTIFIER T ASSIGN char e
     | ids3 T COMMA T IDENTIFIER
     | T IDENTIFIER
ids4:
     T IDENTIFIER T ASSIGN str e
     | ids4 T COMMA T IDENTIFIER
     | T IDENTIFIER
expression:
     arithm e
     | rel e
rel e:
     arithm e T LT arithm e
     | arithm e T GT arithm e
     | arithm e T LE arithm e
      | arithm e T GE arithm e
      | arithm e T EQ arithm e
      | arithm e T NE OP arithm e
      | T TRUE
      | T FALSE
arithm e:
     arithm e T_MUL arithm e
     | arithm e T DIV arithm e
     | arithm e T ADD arithm e
     | arithm e T SUB arithm e
     | T IDENTIFIER
     | T NUM
     | T IDENTIFIER T INC OP
     | T IDENTIFIER T DEC OP
     T INC OP T IDENTIFIER
     | T DEC OP T IDENTIFIER
     | T IDENTIFIER T ASSIGN arithm e
```

```
float e:
     T IDENTIFIER
     | T IDENTIFIER T ASSIGN float e
     | T DECIMAL
char e:
     T IDENTIFIER
     T IDENTIFIER T ASSIGN char e
     | T CONSTANT
str e:
     T IDENTIFIER
     | T IDENTIFIER T ASSIGN str e
     | T STRING LITERAL
if stmt:
     T IF T OC rel e T CC T OF stmnt T CF else stmnt
     ;
else stmnt:
     T ELSE if stmt
     | T_ELSE T_OF {block = 1;} stmnt T_CF {block = 0;}
for stmt:
     T FOR T OC for args T CC T OF stmnt T CF
for args:
     loop init T SEMC loop cond
loop cond:
     arg2 T SEMC arg3
loop init:
     variable declaration
     | expression
     ;
arg2:
     rel e
```

5. DESIGN STRATEGY

- **Symbol table creation-** The symbol table was implemented using a linear array of a 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, Constant propagation, Copy Propagation and Dead code elimination 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.

Copy Propagation

Copy propagation is the process of replacing the occurrences of targets of direct assignments with their values. A direct assignment is an instruction of the form x = y, which simply assigns the value of y to x.

Removal of dead code

Dead code elimination (also known as DCE, dead code removal, dead code stripping, or dead code strip) is a compiler optimization to remove code which does not affect the program results.

Error handling- Type error and semicolon missing error have been handled. Redefinition of identifiers in the same scope, use of undeclared identifiers, unterminated comments, syntax errors have also been covered. If identifier length exceeds maximum limit error is shown for that as well.

6. IMPLEMENTATION DETAILS

Flex and Bison have been used to implement the following:

Symbol table creation:

Implemented in parsefile.y. The symbol table is a linear array of the following structure

```
struct SymTable {
    char idName[50];
    int value;
    float f val;
    char c val;
    char s val[100];
    int type; //0-int , 1-float, 2-char, 3-String, 4-int[],
5-float[]
    int line no;
    int scope;
    int size;
}; struct SymTable st[50];
bison -d parsefile.y
flex lexfile.l
gcc lex.yy.c parsefile.tab.c
./parsefile.out < input2.java
```

Abstract Syntax Tree:

Implemented in AST.y and AST.c

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

```
typedef struct node
  {
    struct node *left;
    struct node *right;
    char *value;
  } node;
  node *createnode(node *left, node *right, char *value) {
      node *newnode = (node *)malloc(sizeof(node));
      char *newstr = (char *)malloc(strlen(value)+1);
      strcpy(newstr, value);
      newnode->left = left;
      newnode->right = right;
      newnode->value = newstr;
      return (newnode);
bison -d ast.y
flex lexfile.l
gcc lex.yy.c ast.tab.c ast.c
./ast.out < input2.java
```

Intermediate Code Generation:

Implemented in intercode.y

The given code was converted into 3 address code

bison -d intercode.y

flex lexfile.l gcc lex.yy.c intercode.tab.c intercode.c ./intercode.out < input2.java

Code Optimization:

Implemented in optimize.py

Constant Folding

Constant folding is the process of recognizing and evaluating constant expressions at compile time rather than computing them at runtime. Terms in constant expressions are typically simple literals, such as the 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:

```
def constant_folding(list_of_lines)
```

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 applied to constant values.

This is done using the below function in our code:

```
def constant propagation():
```

Copy propagation

Copy propagation is the process of replacing the occurrences of targets of direct assignments with their values. A direct assignment is an instruction of the form x = y, which simply assigns the value of y to x.

```
def copy propagation():
```

Removal of dead code

Dead code elimination (also known as DCE, dead code removal, dead code stripping, or dead code strip) is a compiler optimization to remove code which does not affect the program results.

```
def remove dead code(list of lines) :
```

• Error Handling- Implemented in parsefile.y using conditional statements for type error and semicolon missing is checked using the grammar and unused variables. Redefinition of identifiers in the same scope, use of undeclared identifiers, unterminated comments, syntax errors have also been covered. If identifier length exceeds maximum limit error is shown for that as well.

python3 optimize.py #change the path to the file name to icg2.txt

7. RESULTS

A mini java compiler that can compile the chosen constructs has been created.

8. SNAPSHOTS

SYMBOL TABLE GENERATION

bison -d parsefile.y flex lexfile.l gcc lex.yy.c parsefile.tab.c ./parsefile.out

Input file1:

Output1:

```
vibha@PES1201800158client:~/Desktop/CD$./parsefile.out < input2.java</pre>
hello world
1234
2
18
  Type
int
                     Value
                                        line
                                                 Size
            Name
                               Scope
                     8
                                                 2
8
  char
            сг
                               1
            fl
ff
                     6.700000
  float
                                        1
                                                         | 4
                               2
2
2
2
2
1
                                         10
  int
                     104
                     99
  int
                                        11
            y
c
                                        12
                                                 4
  int
                     15
  int
            00
                     444
                                        14
            ff
                                        23
                     10
                                                 4
4
4
4
  int
  int
            fact
                     2
                                        25
                               2 3
                                        26
  int
                     10
                                        29
  int
            dummy
  int
            R
                     10
                               1
                                        31
                               1
                                        32
  int
            z
                     18
                                                 4
  string
                                                 1
                                                         | 33
                                                                  | 12
                     str
                              string
 ACCEPTED
vibha@PES1201800158client:~/Desktop/CD$
```

Input2:

Output:

```
vibha@PES1201800158client:~/Desktop/CD$./parsefile.out < input.java
Error: syntax error, unexpected T_CLASS, expecting T_PUBLIC or T_PRIVATE or T_PROTECTED
Error occured at Line No. 1
Error before : class
vibha@PES1201800158client:~/Desktop/CD$
```

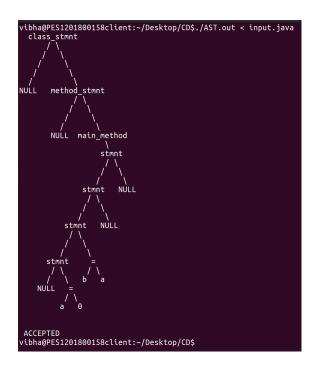
AST Generation:

bison -d ast.y flex lexfile.l gcc lex.yy.c ast.tab.c ast.c ./ast.out

Input:

```
1 public class test {
          public static void main (String[] args)
 2
 3
 4
 5
                 int a=0;
 6
                 int b = a;
 7
                 System.out.print("HELLO WORLD");
 8
          }
 9
10
11 }
```

Output:



INTERMEDIATE CODE GENERATION:

bison -d intercode.y flex lexfile.l gcc lex.yy.c intercode.tab.c intercode.c ./intercode.out

Input:

```
1 public class test {
 2
           public static void main (String[] args)
 3
 4
 5
                  int a=0;
 6
                  int b = a;
 7
                  System.out.print("HELLO WORLD");
 8
           }
 9
10
11 }
```

Output:

```
vibha@PES1201800158client:~/Desktop/CD$./intercode.out < input.java
a = 0
b = a

Line No. is 11
ACCEPTED

the value of inx 2

INTERMEDIATE CODE

# op arg1 arg2 result

0 = 0 a
1 = a b

vibha@PES1201800158client:~/Desktop/CD$</pre>
```

CODE OPTIMIZATION:

python3 optimize.py

Output:

```
vibha@PES1201800158client:~/Desktop/CD$python3 optimize.py
ICG:
a = 0
b = a
dummy = 10
t0 = 5 + 3
c = t0
t1 = c + 10
d = t1

After Copy Propagation:
a = 0
b = a
dummy = 10
t0 = 5 + 3
c = t0
t1 = t0 + 10
d = t1

After Constant Propagation:
a = 0
b = a
dummy = 10
t0 = 5 + 3
c = t0
t1 = t0 + 10
d = t1

After Constant Folding:
a = 0
b = a
dummy = 10
t0 = 5 + 3
c = t0
t1 = t0 + 10
d = t1
After Constant Folding:
a = 0
b = a
dummy = 10
t0 = 8
c = t0
t1 = t0 + 10
d = t1
```

```
After Constant Propagation:
a = 0
b = a
dummy = 10
t0 = 8
c = t0
t1 = 8 + 10
d = t1
After Constant Folding:
a = 0
b = a
dummy = 10
t0 = 8
c = t0
t1 = 18
d = t1
After Dead Code Elimination:
a = 0
b = a
dummy = 10
t0 = 8
c = t0
t1 = 18
d = t1
vibha@PES1201800158client:~/Desktop/CD$
```

9. CONCLUSIONS

A compiler for JAVA was thus created using flex and bison. In addition to the constructs specified, basic building blocks of the language (declaration statements, assignment statements, function creation 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 auxiliary 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.

10. FURTHER ENHANCEMENTS

- Functionality for switch and other flavours of while can be implemented.
- The compiler can be constructed to recover from more kinds of errors.
- More code optimization strategies can be implemented.