Report on

Building a mini java compiler in C using lexx and yacc

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Harshith U	**PES1201700096**
Ziyan Zafar	**PES1201701910**
Mukund Kadlabal	**PES1201700648**

Under the guidance of

Kiran P
Assistant Professor
PES University, Bengaluru

January – May 2020

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

1. INTRODUCTION

Simple constructs from the language JAVA were implemented. The frontend of the compiler includes Symbol table generation, Abstract Syntax tree construction, Intermediate Code generation , Code Optimization and was implemented using lexx and yacc and python.

SAMPLE INPUT AND EXPECTED OUTPUT:

INPUT PROGRAM:

```
class Test
{
//Comment 1

      int b;
    public static int main()
    {
      int a=5;
      b=1;
      for(int i=0;i<10;i++)
      {
           if(a == 2)
           {
                c=4;
           }

           while (a>0)
           {
                b = b*9;
                a=a-1;
           }

      }
      int c=3;
      b = a+c;

    }
 int c=9;

}
```

Symbol Table:

Name	Value	Type	Scope	Line number	size
b	4	int	2	10	4
a	2	int	3	6	4
c	3	int	2	15	4
int	1	keyword	2	1	0
class	1	id	4	1	0

Abstract syntax tree:

```
──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)
```

Intermediate code generation:

```
b = 3
c = 7
a = 6
a = 5
t1 = 6 / 7
t2 = c - t1
c = t2
d = 7
Loop1:
t3 = c > d
if t3 go to Loop2
go to Loop4
Loop4:
t4 = a > b
if t4 go to Loop2
go to Loop3
Loop2:
a = 7
c = 3
Loop5:
t5 = a > b
if t5 go to Loop6
go to Loop7
Loop6:
go to Loop1
d = 0
go to Loop1
Loop3:
a = 5
t6 = c + 5
t7 = 5 == Loop9
if t7 go to Loop9
go to Loop10
Loop9:
a = 0
go to Loop8
go to Loop11
Loop10:
t8 = 6 == Loop11

t10 = 5 * 4
t11 = t10 + t9
c = t11
```

Assembly code:

```
ARM STATEMENT:  MOV REG0,#10
ARM STATEMENT:  STR REG0,a
ARM STATEMENT:  MOV REG1,#10
ARM STATEMENT:  STR REG1,b
ARM STATEMENT:  MOV REG2,#10
ARM STATEMENT:  STR REG2,c
ARM STATEMENT:  MOV REG3,#0
ARM STATEMENT:  STR REG3,i
ARM STATEMENT:  CMP REG3,#5
ARM STATEMENT:  BGE JUMP0
ARM STATEMENT:  MOV REG4,t0
ARM STATEMENT:  STR REG4,i
ARM STATEMENT:  CMP REG4,#5
ARM STATEMENT:  BLE JUMP0
ARM STATEMENT:  MOV REG5,t2

ARM STATEMENT:  BNE LOOP1
ARM STATEMENT:  MOV REG6,t4
ARM STATEMENT:  ADD REG6,REG0,#5
ARM STATEMENT:  STR REG6,a
        LOOP1:
ARM STATEMENT:  MOV REG7,t5
ARM STATEMENT:  ADD REG7,REG6,#5
ARM STATEMENT:  STR REG7,a
ARM STATEMENT:  MOV REG8,t5
ARM STATEMENT:  STR REG8,j
ARM STATEMENT:  MOV REG9,#1
ARM STATEMENT:  STR REG9,j
        LOOP2:
ARM STATEMENT:  CMP REG9,#10
ARM STATEMENT:  BGT LOOP3
ARM STATEMENT:  CMP REG2,#5
ARM STATEMENT:  BLT JUMP2
ARM STATEMENT:  MOV REG10,t9
ARM STATEMENT:  ADD REG10,REG2,#1
ARM STATEMENT:  STR REG10,c
        JUMP2
ARM STATEMENT:  MOV REG11,t10
ARM STATEMENT:  ADD REG11,REG9,#1
ARM STATEMENT:  STR REG11,j
ARM STATEMENT:  B LOOP2
        LOOP3:
ARM STATEMENT:  CMP REG1,#5
ARM STATEMENT:  BNE LOOP4
ARM STATEMENT:  B LOOP3
        LOOP4:
        LOOP5:
ARM STATEMENT:  BNE LOOP6
ARM STATEMENT:  B LOOP5
        LOOP6:
```

ARCHITECTURE OF LANGUAGE

Compiler for the following constructs:

- `while` loop
- for loop
- `if` 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

CONTEXT-FREE GRAMMAR

```
%union
{
   int integer;
   char *string;
}

%token <integer> T_NUM
%token <string> T_ID
%type<integer> T_Const
%type<integer> T_expr
%type<integer> empty

%token   T_PUBLIC   T_PRIVATE
   T_PROTECTED     T_STATIC
   T_VOID          T_CLASS
   T_IMPORT
%token  T_WHILE     T_MAIN     T_DO
   T_FOR      T_IF
%token   T_INT           T_FLOAT
   T_LONG          T_DOUBLE
   T_BOOLEAN  T_CHAR
%token   T_S_PLUSEQ      T_S_MINUSEQ
T_S_MULTEQ            T_S_DIVEQ
   T_TRUE          T_FALSE
%nonassoc     T_S_EQ
%left    T_S_PLUS   T_S_MINUS
   T_S_MULT   T_S_DIV    T_U_DECR
   T_U_INCR
%right   T_GEQ           T_LEQ
   T_LE       T_GE T_ASSG     T_NE

%start start_aug

%%

start_aug
   : import_stmt class_def
{exit(0);}
   | class_def  {exit(0);}
   ;

import_stmt
   : T_IMPORT T_ID
   | import_stmt T_IMPORT T_ID
   ;

class_def
   : modifier class_head
   ;

class_head
   : T_CLASS T_ID '{' main_stmt '}'
{     }
   ;

modifier
   : T_PUBLIC
   | T_STATIC
   | T_PRIVATE
   | T_PROTECTED
   | T_VOID
   ;

main_stmt
   : modifier modifier modifier
T_MAIN '(' ')' '{'    body  '}'
   ;

empty_st
   :
   ;

body
   :  statement_list
   |  empty_st
   ;

statement_list
   :    statement
   |    statement_list   statement
   ;

statement
   : T_ID T_ASSG T_expr ';'
```

```
     { UpdateSymbolTable($1, scope,
$3, &symbolTable);}
     | var_decl
     | iteration_statement
     | conditional_statement
     ;

  iteration_statement
     : T_WHILE '(' cond ')' '{' body
'}'
     | T_DO '{' body '}' T_WHILE'('
cond ')' ';'
     | T_FOR '(' custom_cond ';' cond
';' cond ')'  '{' body '}'
     ;

  custom_cond
     : cond
     | type T_ID T_ASSG T_expr
     { InsertIdentifier($2,
type_helper, scope, $4,
&symbolTable);}
     | T_ID T_ASSG T_expr
     { UpdateSymbolTable($1, scope,
$3, &symbolTable);}
     ;

  conditional_statement
     : T_IF '(' cond ')'  '{' body
'}'
     ;

  T_expr
     : T_expr T_S_PLUS     T_expr
     {$$ = $1 + $3;}
     | T_expr T_S_MINUS    T_expr
     {$$ = $1 - $3;}
     | T_expr T_S_MULT     T_expr  {$$
= $1 * $3;}
     | T_expr T_S_DIV      T_expr  {$$
= $1 / $3;}
     | T_Const             {$$ = $1;}
     | empty
     ;

  empty
     : %empty {}
     ;
```

```
  T_Const
     : T_ID                    {$$ =
LookUpSymbolTable($1, scope,
&symbolTable);}
     | T_NUM               {$$ = $1;}
     ;

  cond
     : T_TRUE
     | T_FALSE
     | T_expr T_S_EQ T_expr
     | T_expr T_GEQ T_expr
     | T_expr T_LEQ T_expr
     | T_expr T_GE T_expr
     | T_expr T_LE T_expr
     | T_expr
     | T_expr T_S_PLUSEQ T_expr  {$1 =
$1 + $3;}
     | T_expr T_S_MINUSEQ T_expr {$1 =
$1 - $3;}
     | T_expr T_S_MULTEQ T_expr  {$1 =
$1 * $3;}
     | T_expr T_S_DIVEQ T_expr   {$1 =
$1 / $3;}
     ;

  type
     : T_INT
     {strcpy(type_helper,"int");}
     | T_FLOAT
     {strcpy(type_helper,"float");}
     | T_LONG
     {strcpy(type_helper,"long");}
     | T_DOUBLE
     {strcpy(type_helper,"float");}
     | T_BOOLEAN
     {strcpy(type_helper,"boolean");}
     ;

  var_decl
     : type T_ID T_ASSG T_expr ';'
     { InsertIdentifier($2,
type_helper, scope, $4,
&symbolTable);}
     | type T_ID ';'
     { InsertIdentifier($2,
type_helper, scope, 0, &symbolTable);}
```

;

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

- **Assembly code generation:** Python program , using arm architecture and immediate addressing was used. Least recently used policy was used to generate and maintain registers.

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 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:

```
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 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.

Assembly code generation:

ARM design rules:

- Instructions: reduced set andfixed length
- Pipeline: decode in one stage
- Registers: a list of 13 general-purpose registers
- Load/store architecture: data processing instructions apply to registers only; load/store to transfer data from memory

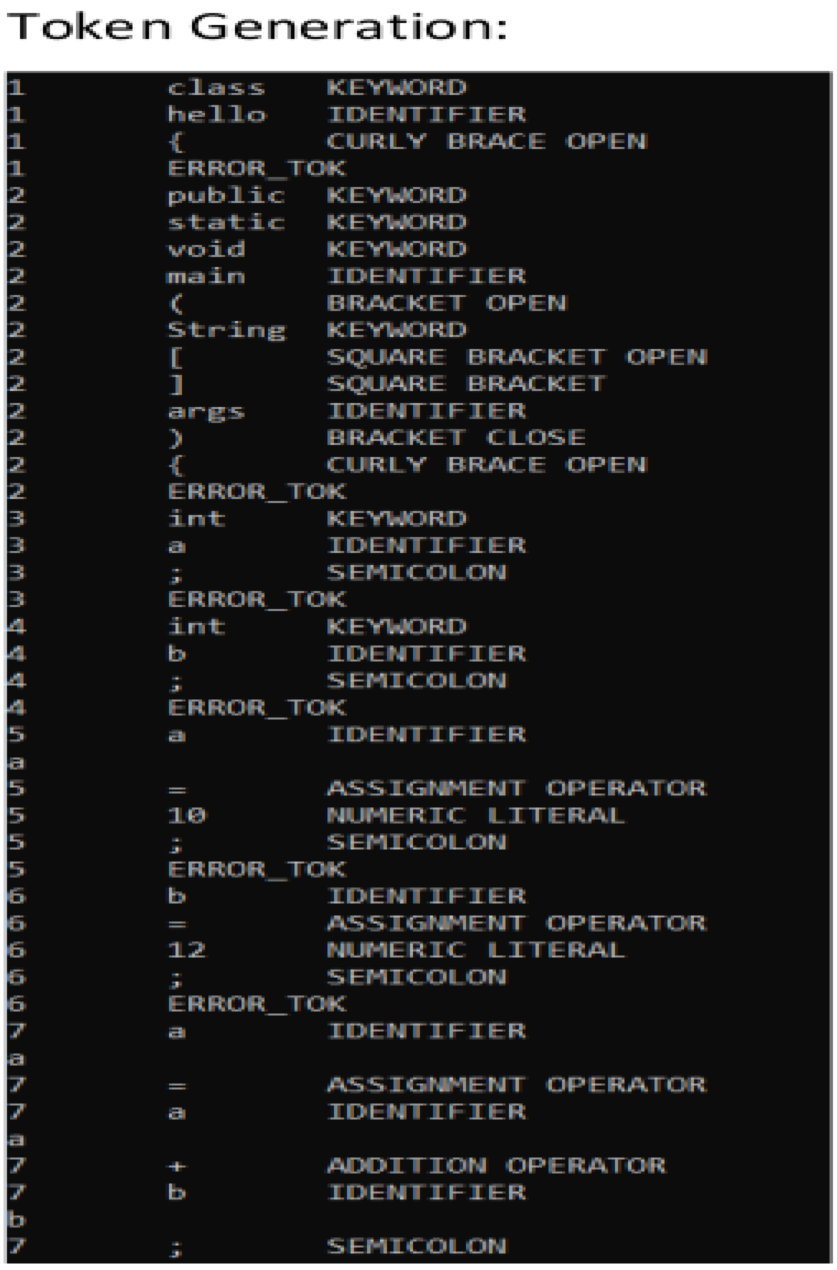
The output of optimized intermediate code is fed to the acg code, if false optimization is done which reduces the load on the code generator.

The assembly code generation is implemented in assembly.py.

Result

A java mini compiler that can handle all the specified constructs was built. All stages of a compiler from symbol table generation to assembly code generation of code generator was implemented.

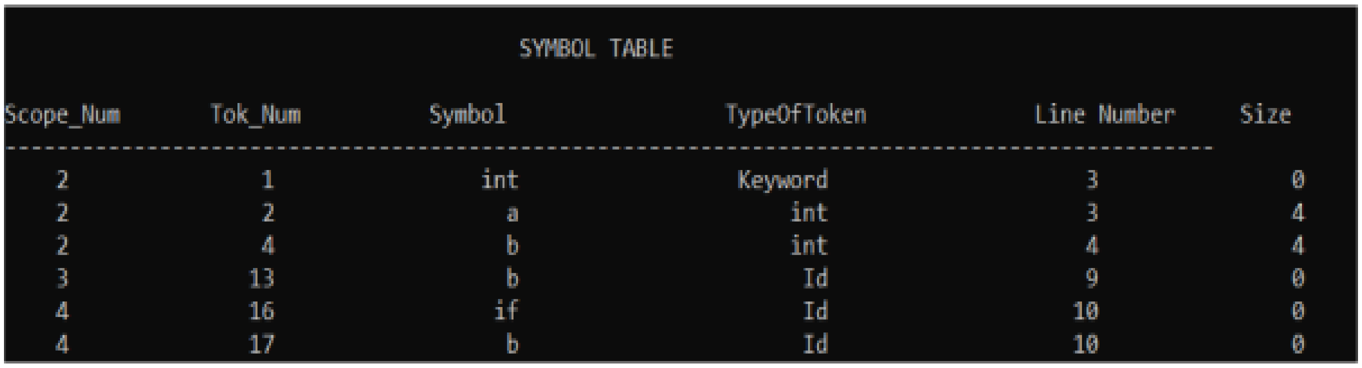
Snapshots:



Token Generation:

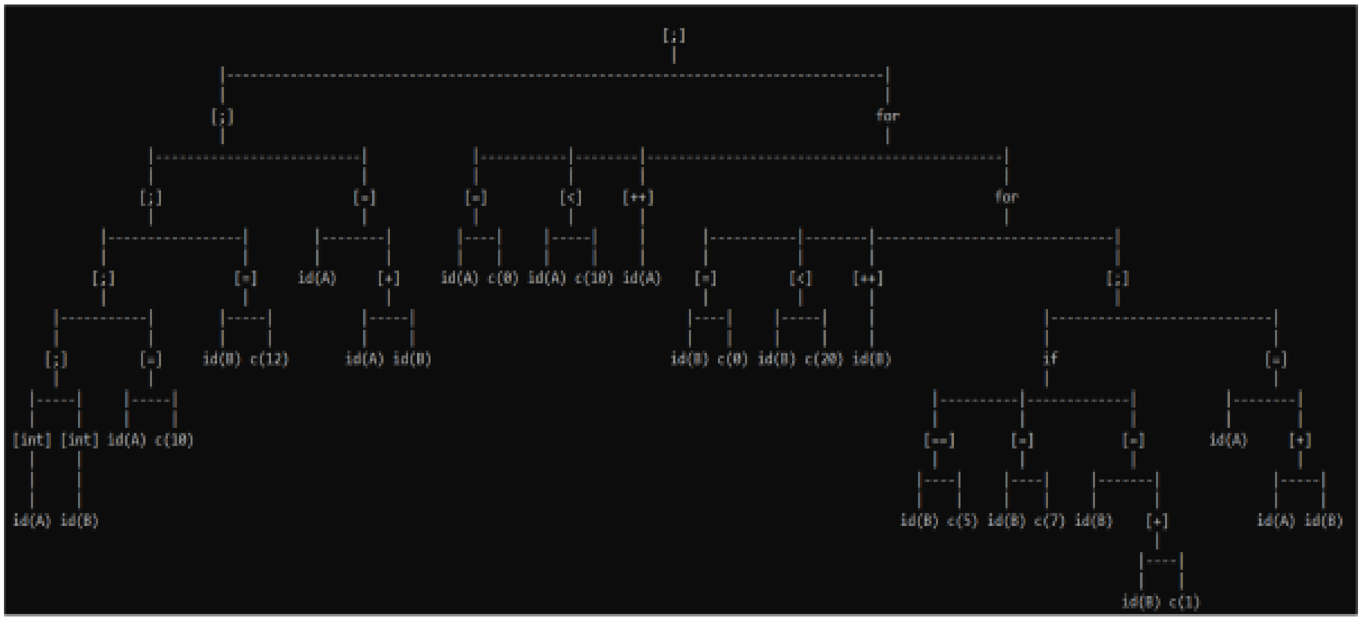
```
1       class    KEYWORD
1       hello    IDENTIFIER
1       {        CURLY BRACE OPEN
1       ERROR_TOK
2       public   KEYWORD
2       static   KEYWORD
2       void     KEYWORD
2       main     IDENTIFIER
2       (        BRACKET OPEN
2       String   KEYWORD
2       [        SQUARE BRACKET OPEN
2       ]        SQUARE BRACKET
2       args     IDENTIFIER
2       )        BRACKET CLOSE
2       {        CURLY BRACE OPEN
2       ERROR_TOK
3       int      KEYWORD
3       a        IDENTIFIER
3       ;        SEMICOLON
3       ERROR_TOK
4       int      KEYWORD
4       b        IDENTIFIER
4       ;        SEMICOLON
4       ERROR_TOK
5       a        IDENTIFIER
a

5       =        ASSIGNMENT OPERATOR
5       10       NUMERIC LITERAL
5       ;        SEMICOLON
5       ERROR_TOK
6       b        IDENTIFIER
6       =        ASSIGNMENT OPERATOR
6       12       NUMERIC LITERAL
6       ;        SEMICOLON
6       ERROR_TOK
7       a        IDENTIFIER
a

7       =        ASSIGNMENT OPERATOR
7       a        IDENTIFIER
a
7       +        ADDITION OPERATOR
7       b        IDENTIFIER
b
7       ;        SEMICOLON
```

Symbol Table:



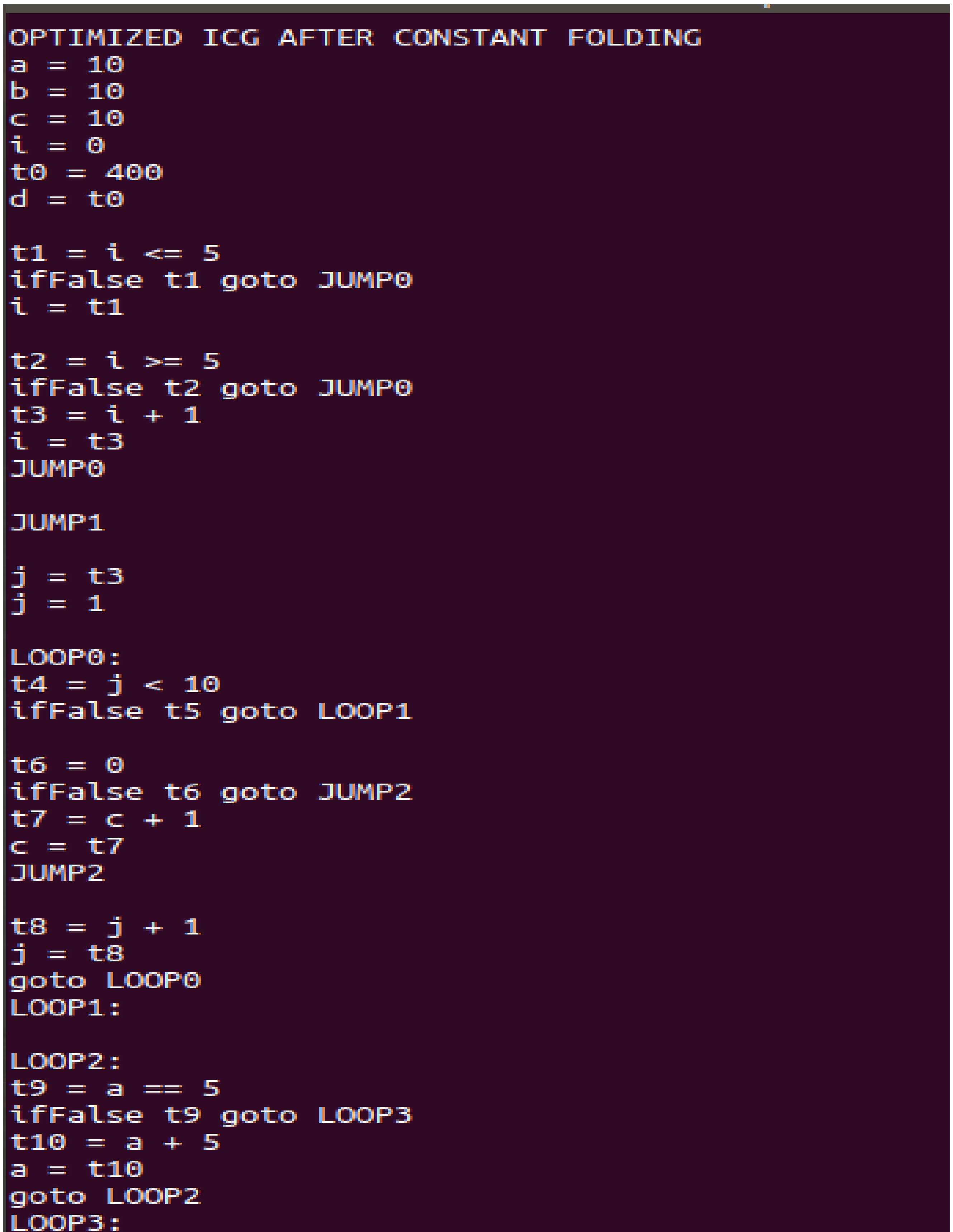
```
                                        SYMBOL TABLE

Scope_Num       Tok_Num          Symbol                  TypeOfToken              Line Number     Size
----------------------------------------------------------------------------------------------
    2              1              int                      Keyword                     3              0
    2              2                a                          int                     3              4
    2              4                b                          int                     4              4
    3             13                b                           Id                     9              0
    4             16               if                           Id                    10              0
    4             17                b                           Id                    10              0
```

Abstract Syntax Tree:



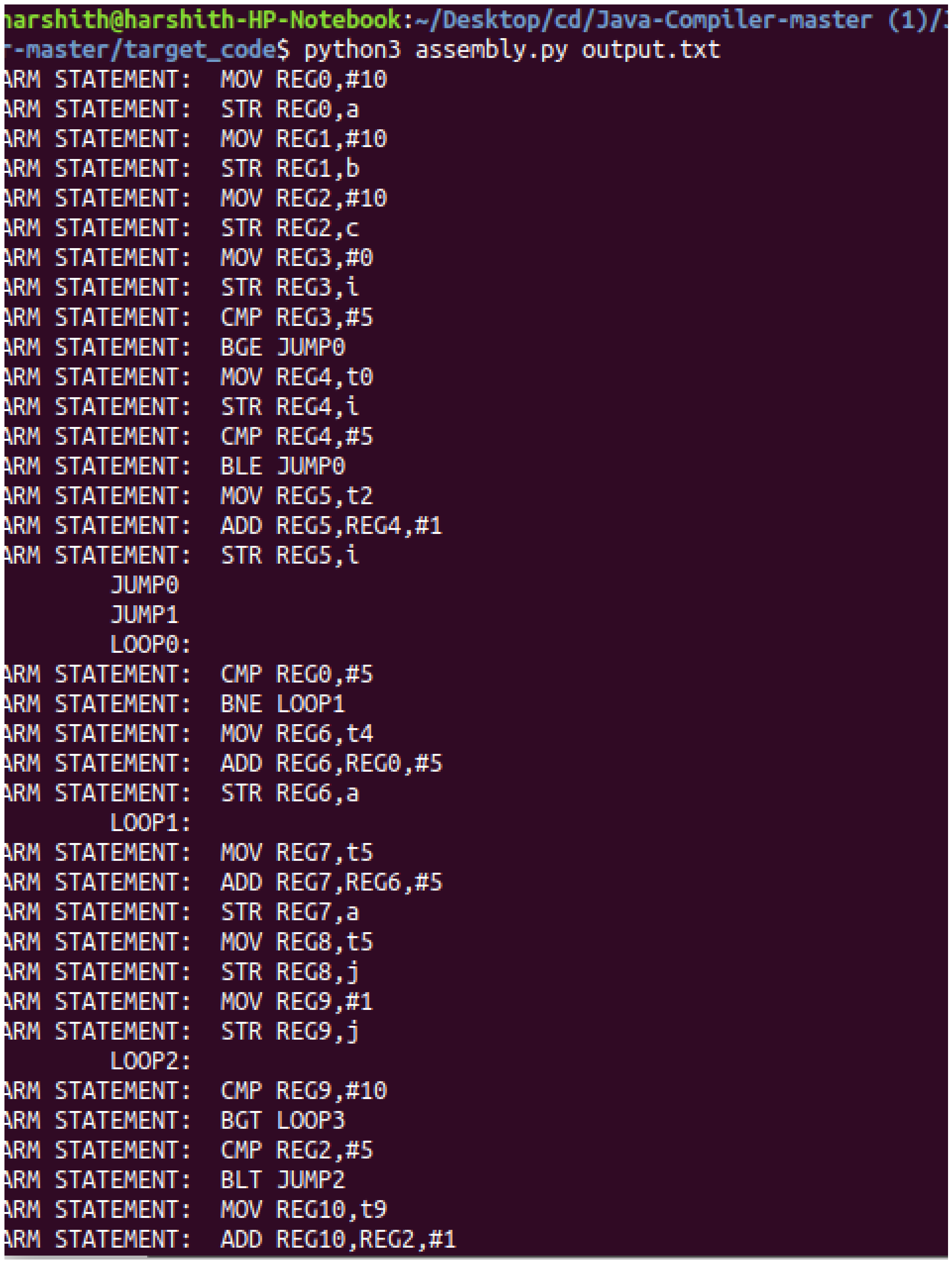
```
                                                                                     [;]
                                                                                      |
                           |-------------------------------------------------------------------------------------|
                           |                                                                                     |
                          [;]                                                                                   for
                           |                                                                                     |
                  |---------------------------|             |-----------|---------|----------------------------------------------|
                  |                           |             |           |         |                                              |
                 [;]                         [=]           [=]         [<]       [++]                                           for
                  |                           |             |           |         |                                              |
            |------------------|        |--------|        |----|     |------|     |       |-----------|--------|-------------------------------|
            |                  |        |        |        |    |     |      |     |       |           |        |                               |
           [;]                [=]     id(A)     [+]     id(A) c(0) id(A)  c(10) id(A)    [=]         [<]      [++]                            [;]
            |                  |                 |                                        |           |        |                               |
      |-----------|        |------|           |-----|                                   |----|     |-----|     |                      |-----------------------------|
      |           |        |      |           |     |                                   |    |     |     |     |                      |                             |
     [;]         [=]     id(B)  c(12)       id(A) id(B)                               id(B) c(0) id(B) c(20) id(B)                    if                           [=]
      |           |                                                                                                                   |                             |
   |-----|     |-----|                                                                                                  |----------|-------------|           |---------|
   |     |     |     |                                                                                                  |          |             |           |         |
 [int] [int] id(A) c(10)                                                                                               [==]       [=]           [=]        id(A)      [+]
   |     |                                                                                                              |          |             |                     |
   |     |                                                                                                            |----|     |----|     |-------|               |-----|
   |     |                                                                                                            |    |     |    |     |       |               |     |
 id(A) id(B)                                                                                                        id(B) c(5) id(B) c(7) id(B)    [+]           id(A) id(B)
                                                                                                                                                    |
                                                                                                                                                  |----|
                                                                                                                                                  |    |
                                                                                                                                                id(B) c(1)
```

Intermediate code generation:



```
OPTIMIZED ICG AFTER CONSTANT FOLDING
a = 10
b = 10
c = 10
i = 0
t0 = 400
d = t0

t1 = i <= 5
ifFalse t1 goto JUMP0
i = t1

t2 = i >= 5
ifFalse t2 goto JUMP0
t3 = i + 1
i = t3
JUMP0

JUMP1

j = t3
j = 1

LOOP0:
t4 = j < 10
ifFalse t5 goto LOOP1

t6 = 0
ifFalse t6 goto JUMP2
t7 = c + 1
c = t7
JUMP2

t8 = j + 1
j = t8
goto LOOP0
LOOP1:

LOOP2:
t9 = a == 5
ifFalse t9 goto LOOP3
t10 = a + 5
a = t10
goto LOOP2
LOOP3:
```

Assembly code generation:



```
harshith@harshith-HP-Notebook:~/Desktop/cd/Java-Compiler-master (1)/
-master/target_code$ python3 assembly.py output.txt
ARM STATEMENT:  MOV REG0,#10
ARM STATEMENT:  STR REG0,a
ARM STATEMENT:  MOV REG1,#10
ARM STATEMENT:  STR REG1,b
ARM STATEMENT:  MOV REG2,#10
ARM STATEMENT:  STR REG2,c
ARM STATEMENT:  MOV REG3,#0
ARM STATEMENT:  STR REG3,i
ARM STATEMENT:  CMP REG3,#5
ARM STATEMENT:  BGE JUMP0
ARM STATEMENT:  MOV REG4,t0
ARM STATEMENT:  STR REG4,i
ARM STATEMENT:  CMP REG4,#5
ARM STATEMENT:  BLE JUMP0
ARM STATEMENT:  MOV REG5,t2
ARM STATEMENT:  ADD REG5,REG4,#1
ARM STATEMENT:  STR REG5,i
        JUMP0
        JUMP1
        LOOP0:
ARM STATEMENT:  CMP REG0,#5
ARM STATEMENT:  BNE LOOP1
ARM STATEMENT:  MOV REG6,t4
ARM STATEMENT:  ADD REG6,REG0,#5
ARM STATEMENT:  STR REG6,a
        LOOP1:
ARM STATEMENT:  MOV REG7,t5
ARM STATEMENT:  ADD REG7,REG6,#5
ARM STATEMENT:  STR REG7,a
ARM STATEMENT:  MOV REG8,t5
ARM STATEMENT:  STR REG8,j
ARM STATEMENT:  MOV REG9,#1
ARM STATEMENT:  STR REG9,j
        LOOP2:
ARM STATEMENT:  CMP REG9,#10
ARM STATEMENT:  BGT LOOP3
ARM STATEMENT:  CMP REG2,#5
ARM STATEMENT:  BLT JUMP2
ARM STATEMENT:  MOV REG10,t9
ARM STATEMENT:  ADD REG10,REG2,#1
```

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 NS assembly generation). 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.

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
- more optimization techniques can be implemented.