Report on

"Building a mini-compiler of Java based on C"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Jnanesh D	**PES1201701822**
Sujay Gad	**PES1201700177**
Mayur RB	**PES1201700714**

Under the guidance of

Preet Kanwal
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

A Java compiler is implemented on C which includes the basic constructs of the language .
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 of C .
The assembly code of MIPS generation was implemented in python.

Sample Input-

```
public class b{
    public static void main(String []args)
    {
        int a=3;
        int c=a+4;
        int d=c-4;
        int e;
        if(a<c)
       {
              e = d + a;
       }
       else{
              e = e*6;
       }
    }
}
```

Output-

Symbol Table-

Name	Value	Type	Scope	lineno	Size
a	3	int	1	4	4
c	7	int	1	5	4
d	0	int	1	14	4
e	6	int	1	10	4

Abstract Syntax Tree-

```
Abstract Syntax Tree
└──CLASS DECLARATION
    ├──modifier
    │   └──(access modifier, public)
    ├──(classname, b)
    └──CLASS BODY
        └──METHOD DECLARATION
            ├──modifier
            │   ├──(access modifier, public)
            │   └──(access modifier, static)
            ├──(datatype, void)
            ├──(datatype, String)
            └──VARIABLE DECLARATION/INITIALISATION STATEMENT
                ├──variable initialisation
                │   ├──(datatype, int)
                │   ├──(id, a)
                │   └──(num, 3)
                └──VARIABLE DECLARATION/INITIALISATION STATEMENT
                    ├──variable initialisation
                    │   ├──(datatype, int)
                    │   ├──(id, c)
                    │   └──+
                    │       ├──(id, a)
                    │       └──(num, 4)
                    └──VARIABLE DECLARATION/INITIALISATION STATEMENT
                        ├──variable initialisation
                        │   ├──(datatype, int)
                        │   ├──(id, d)
                        │   └──-
                        │       ├──(id, c)
                        │       └──(num, 4)
                        └──VARIABLE DECLARATION/INITIALISATION STATEMENT
                            ├──variable declaration
                            │   ├──(datatype, int)
                            │   └──(id, e)
                            └──IF ELSE STATEMENT
                                ├──<
                                │   ├──(id, a)
                                │   └──(id, c)
                                ├──ASSIGNMENT STATEMENT
                                │   └──=
                                │       ├──(id, e)
                                │       └──+
                                │           ├──(id, d)
                                │           └──(id, a)
                                └──ASSIGNMENT STATEMENT
                                    └──=
                                        ├──(id, d)
                                        └──(num, 0)
```

```
│         └──(num, 4)
└──VARIABLE DECLARATION/INITIALISATION STATEMENT
    ├──variable declaration
    │    ├──(datatype, int)
    │    └──(id, e)
    └──IF STATEMENT
         ├──<
         │    ├──(id, a)
         │    └──(id, c)
         ├──ASSIGNMENT STATEMENT
         │    └──=
         │         ├──(id, e)
         │         └──+
         │              ├──(id, d)
         │              └──(id, a)
         └──VARIABLE DECLARATION/INITIALISATION STATEMENT
              ├──variable initialisation
              │    ├──(datatype, int)
              │    ├──(id, num)
              │    └──*
              │         ├──(id, e)
              │         └──(num, 6)
              └──IF STATEMENT
                   ├──>
                   │    ├──(id, num)
                   │    └──(num, 30)
                   └──ASSIGNMENT STATEMENT
                        └──=
                             ├──(id, d)
                             └──(num, 0)
```

Intermediate Code Generation-

```
a = 3
T1 = a + 4
c = T1
T2 = T1 - 4
d = T2
T3 = a < T1
if T3 goto L1
goto LABEL1
L1:T4 = T2 + a
e = T4
goto L3
LABEL1:T5 = T4 * 6
e = T5
L3:
```

Code Optimization -

```
a = 3
T1 = 7
c = 7
T2 = 3
d = 3
T3 = 1
if T3 goto L1
goto LABEL1

L1 : T4 = 6
e = 6
goto L3

LABEL1 : T5 = 36
e = 36

L3 :
```

Target Code generation –

First phase :

```
Basic Block 1
a = 3
T1 = 7
c = 7
T2 = 3
d = 3
T3 = 1
if T3 goto L1

Basic Block 2
goto LABEL1

Basic Block 3


Basic Block 4
L1 : T4 = 6
e = 6
goto L3


Basic Block 5
LABEL1 : T5 = 36
e = 36


Basic Block 6
L3 :
```

Second phase:

```
New Block
mov R1 3
sd R1 a
mov R2 7
mov R3 7
sd R3 c
mov R4 3
mov R5 3
sd R5 d
mov R6 1
bne R6 0 L1

New Block
b LABEL1


New Block
L1 : mov R7 6
mov R8 6
sd R8 e
b L3


New Block
LABEL1 : mov R1 36
mov R8 36
sd R8 e


New Block
L3 :
```

2 . Architecture of language

Compiler for the following constructs:

- `If else` loop
- `switch case` construct
- `int` data type
- `float` data type
- `char` data type
- `String` data type
- `return, break, continue` statements
- modifiers and function calls
- Multidimensional Arrays
- Arithmetic and logical operators
- Comments

3 . LITERATURE SURVEY

Lex Yacc and its internal working
https://www.tldp.org/HOWTO/Lex-YACC-HOWTO.html#toc1

Building a mini-compiler - tutorial
Compiler Design Tutorial

Expression evaluation using Abstract Syntax Tree
https://mariusbancila.ro/blog/2009/02/03/evaluating-expressions-part-1/

Code generation
Code Generation

Compilers – Principles, Techniques and Tools
Second Edition

4 . CONTEXT-FREE GRAMMAR

```
Start:
    Import_S Start
    |Class_declaration
    ;

Import_S:
    T_IMPORT T_ID'.'T_ID'.''*'';'
    ;

Class_declaration:
    Modifier T_CLASS T_ID '{'Class_Body'}'
    ;

Class_Body:
    Global_variable_declaration Class_Body|Method_declaration
    Class_Body
    |
    ;

Global_variable_declaration:
    Modifier Variable_declaration';'


Method_declaration:
    Modifier Type T_ID'('Parameters')'Block
    | Modifier T_VOID T_ID
    '('Parameters')'Block
    ;

Modifier:
    T_PUBLIC Modifier1
    |T_PRIVATE Modifier1
    |T_PROTECTED Modifier1
    |Modifier1
    ;
```

Modifier1:

```
   T_STATIC Modifier2
   ;
```

Modifier2:

```
   T_FINAL
   |
   ;
```

Params:

```
   List_of_parameters
   ;
```

```
List_of_parameters:
   Type T_ID
   |Type T_ID',' Parameters
   |Type'['']' T_ARGS;
   ;
```

Block:

```
   '{'S'}'
```

Statement:

```
   Assignment S
   |T_BREAK';' S
   |T_CONTINUE';' S
   |T_IF'('Expression')'S
   |T_IF '('Expression')' Block S
   |T_IF'('Expression')'Block T_ELSE Block
   |T_SWITCH'('Expression')' `{'SwitchBlock'}' S
   |T_WHILE'('Expression')'Statement
   |T_RETURN Expression';'
Statement
   |T_SWITCH'('Expression')' '{'SwitchBlock'}' Statement
   |Variable_declaration S
   |Array_declaration';' S
   |Array_initialisation';' S
   |error ';' S|H';'|
   ;
```

H:

```
   T_ID T_INC
```

```
    |T_ID T_DEC
    |T_INC T_ID
    |T_DEC T_ID;
```

SwitchBlock:
```
    SwitchLabel S SwitchBlock|;
```

SwitchLabel:
```
    T_CASE Expression
    | T_DEFAULT
    ;
```

Variable_declaration:
```
    Type T_ID '=' Expression
    Identifier_List';'
    |Type T_ID Identifier_List''
    ;
```

Identifier_List:
```
    ','T_ID '=' Expression
    Identifier_List
    |','T_ID Identifier_List
    |;
```
Array_Declaration:
```
    Type B T_ID
    | Type T_ID B;
```
B:
```
    '['']'B|'['']';
```
BB:
```
    '[' BNUM ']' | '[' BNUM']'BB;
```
BNUM :
```
    T_NUM | T_ID;
```
Array_Initialization:
```
Array_declaration Assignment_operator K;
```
K:
```
    V|V','K|T_NEW Type BB;
```
V:
```
    T_NUM|R;
```
R:

```
     '{'K'}';
Type:
     T_INT|T_DOUBLE|T_CHAR|T_STRING;
Assignment:
     T_ID Assignment_operator Expression';' ;
Assignment_operator:

'='|T_SHA|T_SHS|T_SHM|T_SHD|T_SHAND|T_SHO|T_SHC|T_SHMOD|';';
Operators:

T_OR|T_AND|'|'|'^'|'&'|T_EQ|T_NE|'<'|'>'|T_LTE|T_GTE|T_LS|T_RS
|'+'|'-'|'*'|'/'|'%';
Expression:
      Expr | Expr Operators Expression ;
Expr:
      '('Expression')'|T_NUM|T_ID;
```

5 . DESIGN STRATEGY

- **Symbol table creation-** The symbol table was implemented using a linked list with entries as an array structure that contains the identifier, scope, type, lineno, size and its value.
- **Abstract Syntax Tree-** This tree is constructed as the input is parsed. Each node of this tree contains pointers to a maximum of 4 children which refer to non terminals on the right hand side of the grammar.
- **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.

- **Target Code Generation –** Target code generation has been implemented using python. Size of the register set has been kept as 8. R1 - R8 are used as registers. Left hand side operands/values are taken from registers if they are in registers otherwise loaded from memory. Conditional as well as unconditional jumps have also been taken care of using proper labels.
- **Error handling-** We have implemented panic mode recovery indicating line number of error. Variables which are uninitialized and multiple initializations of the same variable has been handled.

6 . 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;
    int lineno;
    int size;
    struct NODE* next;
}
NODE;
```

- **Abstract Syntax Tree-** Implemented in symb.y

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

```
typedef struct tree
{
    char   *opr;
    char *value;
    struct tree* c1;
    struct tree* c2;
    struct tree* c3;
    struct tree* c4;
}
TREE;

typedef struct ast
{
    TREE* root;
}
AST;
```

- **Intermediate Code Generation-**

Implemented in if.y . The given code was converted to the 3 address code. The user defined yacc structure which was used in ICG is

```
typedef struct yacc
{
   char* tr;
   char* fal;
   char* next;
   int i;
   float f;
   char* v;
   char* a;
   char* code;
   char* addr;
   int scope;
   int occur;
   char *type;
   char* val;
   TREE *ptr;
}YACC;
```

Labels in case of nested if-else statements have been taken care of using stack data structure.

```
typedef struct Stack {
 int top;
 unsigned capacity;
 int* array;
 }STACK;
```

Newlabel and newtemp functions are used to generate new labels and temporaries respectively. All the temporaries in the ICG stored in a linked list. They can be searched if the same variable is used again in the program. Each node consists of a variable and corresponding temporary.

```
typedef struct node{
char* temp;
char* var;
struct node* next;
}NODE;

typedef struct list{
NODE* head;
}LIST;
```

- **Code Optimization-** Implemented in opt.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)
```

- **Target Code Generation -** Memory is implemented as a python list. All variables except temporaries are stored into memory. Register set is implemented using 2 lists, one list contains register name and the other contains register value. If the register set is full the least recently used variable is removed and a new variable is added to that corresponding register.

- **Error Handling-** Implemented in symb.y we print the line number where the error syntax has occurred and check for uninitialized variables in the parsing stage. We have also checked if a variable has been initialised multiple times.

```
To run our project
Inside the AST
lex AST.l
yacc -d AST.l
gcc lex.yy.c y.tab.c -ll
./a.out c.java

Inside the ICG

lex  icg.l
yacc -d icg.y
gcc lex.yy.c y.tab.c -ll
./a.out c.java

The output of ICG copied to OPTIM folder
Lex optimicons.l
Yacc -d optimicons.y

The output of the Optimization copied to target folder
Python basic_block.py > input3.txt
Python target.py
```

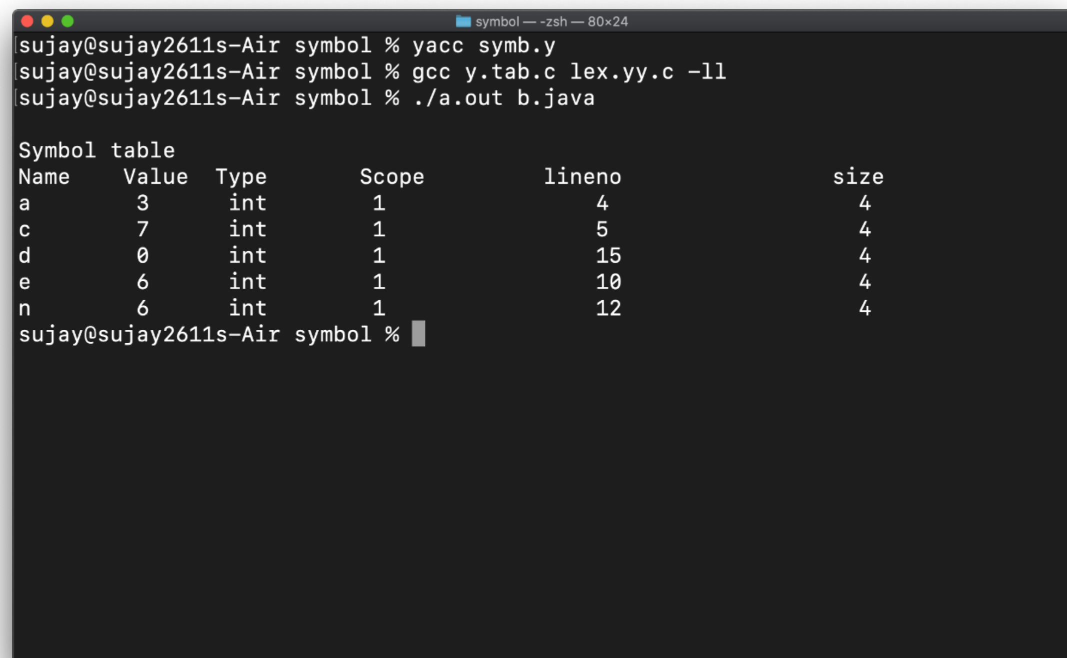
7 . RESULTS

A mini compiler that can compile the chosen constructs of java is obtained. We were successful in implementing our own compiler starting from writing valid context free grammar till target code generation.

Possible shortcomings-

Compiler is unable to detect missing brackets, semicolons. Compiler is unable to handle expressions having more than 2 variables on right hand side.

8 . SNAPSHOTS



```
Last login: Mon May 18 15:25:10 on ttys000
sujay@sujay2611s-Air ~ % cd Desktop/project
sujay@sujay2611s-Air project % cd symbol
sujay@sujay2611s-Air symbol % lex sym.l
sujay@sujay2611s-Air symbol % yacc -d symb.y
sujay@sujay2611s-Air symbol % ./a.out a.java
Panic mode recovery at line : 10
Panic mode recovery at line : 12
Succesful parsing
sujay@sujay2611s-Air symbol % ./a.out c.java
Variable a not declared at line 9
Succesful parsing
sujay@sujay2611s-Air symbol %
```

```
sujay@sujay2611s-Air use % python3 reg.py
New Block
mov R1 45
sd R1 a
mov R2 8
sd R2 b
ld R3 c
mul R4 R2 R3
add R5 R4 R1
mov R3 R5
sd R3 c
L1 : mov R7 0
sd R7 i
bgte R7 50 L3


New Block
mod R8 R7 2
L2 : bne 4 10 L2


New Block
mov R1 0
sd R1 a
b L1
L3 : mov R1 1
sd R1 a
b L1
ld R3 d
mul R4 R3 R3
add R5 R2 R4
ld R6 f
ld R7 t
div R8 R6 R7
```

```
Abstract Syntax Tree
└──CLASS DECLARATION
    ├──modifier
    │   └──(access modifier, public)
    ├──(classname, a)
    └──CLASS BODY
        └──METHOD DECLARATION
            ├──modifier
            │   ├──(access modifier, public)
            │   └──(access modifier, static)
            ├──(datatype, void)
            ├──(datatype, String)
            └──VARIABLE DECLARATION/INITIALISATION STATEMENT
                ├──variable initialisation
                │   ├──(datatype, int)
                │   ├──(id, a)
                │   └──(num, 3)
                └──SWITCH STATEMENT
                    ├──(id, a)
                    └──switch
                        ├──(num, 1)
                        ├──ASSIGNMENT STATEMENT
                        │   └──=
                        │       ├──(id, a)
                        │       └──-
                        │           ├──(id, a)
                        │           └──(num, 5)
                        └──switch
                            ├──(num, 2)
                            └──ASSIGNMENT STATEMENT
                                └──=
                                    ├──(id, a)
                                    └──+
                                        ├──(id, a)
                                        └──(num, 4)
```

9 . CONCLUSION

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 and Target Code generation 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

- Other looping constructs such as for, while and do while can be implemented.
- Including non primitive types other than arrays in the compiler.
- Adding higher levels of error recovery mechanisms such as giving back suggestions to the user.
- Incorporating more complex instructions such as MLA into the target code.
- Adding object oriented functionalities to the compiler.