UE18CS351: Compiler Design
Mini Compiler for Java

Team Members:

Divya S	**PES2201800272**
Sai Shruthi S	**PES2201800361**
Abirami S	**PES2201800495**

Section: 6D

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Table of Content

1. Introduction

Java is a general-purpose, class-based, object-oriented programming language designed for having lesser implementation dependencies. It is a computing platform for application development. Java is fast, secure, and reliable.
This Document will describe the implementation of java compiler using c language. Symbol table generation, Abstract Syntax Tree construction, Intermediate Code generation and Code optimization was implemented using flex and bison of C and the assembly code of MIPS generation was implemented in python.

2. Syntax and Semantics of Java we are handling

The java compiler will have the following constructs:

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

3. Context Free Grammar

CFG we made for this mini compiler is as follows:

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;

4. Symbol Table Creation

The symbol table is implemented using a linked list with entries as an array structure that contains the identifier, scope, type, line no, size and its value. 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;
```

Hence the expected output for this is:

```
divya@Meraki:~/Documents/CD_Project/symbol$ lex lex.l
divya@Meraki:~/Documents/CD_Project/symbol$ yacc -d parse.y
divya@Meraki:~/Documents/CD_Project/symbol$ gcc lex.yy.c y.tab.c
divya@Meraki:~/Documents/CD_Project/symbol$ ./a.out < uniform.java
Succesful parsing
divya@Meraki:~/Documents/CD_Project/symbol$ cat symbol_table.txt
Symbol table         Name        Value        Type        Scope        lineno        size
                        a            3         int            1             4           4
                        b            7         int            1             5           4
```

5. Abstract Tree Generation

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.
It is 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;
```

Hence the expected output for this is:

```
divya@Meraki:~/Documents/CD_Project/AST$ cat AST2.txt
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)
                └──VARIABLE DECLARATION/INITIALISATION STATEMENT
                    ├──variable initialisation
                    │   ├──(datatype, int)
                    │   ├──(id, b)
                    │   └──+
                    │       ├──(id, a)
                    │       └──(num, 4)
                    └──IF ELSE STATEMENT
                        ├──>
                        │   ├──(id, a)
                        │   └──(num, 0)
                        ├──ASSIGNMENT STATEMENT
                        │   └──=
                        │       ├──(id, a)
                        │       └──-
                        │           ├──(id, a)
                        │           └──(num, 2)
                        └──ASSIGNMENT STATEMENT
                            └──=
                                ├──(id, a)
                                └──+
                                    ├──(id, a)
                                    └──(num, 3)
```

6. 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.
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;
```

Hence the expected output for this is:

```
divya@Meraki:~/Documents/CD_Project/ICG$ cat uniform.java
public class a{
        public static void main(String []args)
        {
                int a=3;
                int b=a+4;
                if(a>0)
                {
                        a=a-2;
                }
                else
                {a=a+3;}

        }
}divya@Meraki:~/Documents/CD_Project/ICG$ cat icg1.txt
a = 1
b = 2
c = a
T1 = b + a
b = T1
```

7. 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.
  This is done using the below function in our code:

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

- **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. 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)
```

- **Common Subexpression Elimination**: This is an optimization technique that searches for instances of identical expressions, and replaces them with a single variable holding the computed value.

- **Copy Propagation** : This 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.

This is Implemented in opt.y
Hence the expected output for this is:

```
divya@Meraki:~/Documents/CD_Project/CodeOptimization$ cat sample.txt
e=2
t1=12*4
t2=5/e
t6=4*i
t7=4*i
t8=4*j
t9=a[t8]
a[t7]=t9
t10=4*j
a[t10]=x
t6=4*1
k=t6
t7=k+2
t8=4*2
x=3
y=x
z=y+3
divya@Meraki:~/Documents/CD_Project/CodeOptimization$ python3 optimize.py sample.txt > opt_icg.txt
divya@Meraki:~/Documents/CD_Project/CodeOptimization$ cat opt_icg.txt
e=2
t1=48
t2=2.5
t6=4*i
t7=4
t8=4*j
t9=a[8]
a[t7]=t9
t10=8
a[t10]=3
t6=4
k=t7
t7=k+2
t8=8
x=3
y=3
z=6
```

8. Error Handling

We have implemented panic mode recovery indicating line number of error. Variables which are uninitialized and multiple initializations of the same variable have been handled. This is 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.
Thus **syntax** errors and **semantic** errors have been handled.
Hence the expected output for this is:

Undefined Variable Error:

```
Semantic Error Handling Techniques
divya@Meraki:~/Documents/CD_Project/symbol$ cat s1.java
public class a{
        public static void main(String []args)
        {
                 a=3;
                int b=a+4;
                if(a>0)
                {
                        a=a-2;
                }

                a=a+3;

        }
}divya@Meraki:~/Documents/CD_Project/symbol$ lex lex.l
divya@Meraki:~/Documents/CD_Project/symbol$ yacc -d parse.y
divya@Meraki:~/Documents/CD_Project/symbol$ ./a.out < s1.java
Variable a not declared at line 4
Variable a not declared at line 5
Variable a not declared at line 6
Variable a not declared at line 8
Variable a not declared at line 8
Variable a not declared at line 11
Variable a not declared at line 11
Succesful parsing
```

Multiple Declarations Error:

```
Semantic Error Handling Techniques
divya@Meraki:~/Documents/CD_Project/symbol$ cat s2.java
public class a{
        public static void main(String []args)
        {
                int a=3;
                int a=5;
                int b=a+4;
                if(a>0)
                {
                        a=a-2;
                }

                a=a+3;

        }
}divya@Meraki:~/Documents/CD_Project/symbol$ lex lex.l
divya@Meraki:~/Documents/CD_Project/symbol$ yacc -d parse.y
divya@Meraki:~/Documents/CD_Project/symbol$ gcc lex.yy.c y.tab.c
divya@Meraki:~/Documents/CD_Project/symbol$ ./a.out < s2.java
Variable a already declared
Succesful parsing
```

9. Conclusion

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