Report on

"Switch-Case and Do-While Constructs in C++ Programming Language"

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

Compiler Design Laboratory

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Shailendra Hegde	**01FB16ECS348**
Shailesh Sridhar	**01FB16ECS349**
Sharath Menon	**01FB16ECS351**

Under the guidance of

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

REFERENCES/BIBLIOGRAPHY	

Introduction

A compiler executes four major steps:

- **Scanning**: The scanner reads one character at a time from the source code and keeps track of which character is present in which line
- **Lexical Analysis**: The compiler converts the sequence of characters that appear in the source code into a series of strings of characters (known as tokens), which are associated by a specific rule by a program called a lexical analyzer. A symbol table is used by the lexical analyzer to store the words in the source code that correspond to the token generated.

  The above two functions are handled by flex

- **Syntactic Analysis**: In this step, syntax analysis is performed, which involves preprocessing to determine whether the tokens created during lexical analysis are in proper order as per their usage. The correct order of a set of keywords, which can yield a desired result, is called syntax. The compiler has to check the source code to ensure syntactic accuracy.
- **Semantic Analysis**: First, the structure of tokens is checked, along with their order with respect to the grammar in a given language. The meaning of the token structure is interpreted by the parser and analyzer to finally generate an intermediate code, called object code.Finally, the entire code is parsed and interpreted to check if any optimizations are possible. Once optimizations can be performed, the appropriate modified tokens are inserted in the object code to generate the final object code, which is saved inside a file.

  The above two functions are handled by yacc

  Our compiler Design project is a flex and yacc implementation designed to handle the above crucial steps wrt to the SWITCH and DOWHILE constructs in C++

Inputs:

```
#include<iostream.h>
int qwe[3];
int zxc;
main()
{
        int d=10;int b;int a=20;
        switch(a)
        {
                case 1:b=2*4/3;
                        a=3+2+4/3;
                        break;
                case 2: a=2*4/3;
                        b=3+2*4/3;
                        d=1+2*4/3;
                        break;
                default:break;
        }
}
```

```
#include<iostream.h>
int qwe[3];
int zxc;
main()
{
        int d=10;int b;int a=20;
        do{
                int abc=10*2;
                b=2+10*2;
                do{
                        a=12*4/3;
                        b=1+12*4/3;
                }while(abc);
        }while(b);
}
```

Outputs:

1)Symbol Table:

```
Symbol Table:*********************************
<operator> +
<operator> -
<operator> *
<operator> /
<operator> <
<operator> >
<operator> =
<operator> L
<operator> G
<operator> E
<operator> I
<operator> D
<operator> O
<operator> A
<keyword> do
<keyword> while
<keyword> break
<keyword> default
<keyword> case
<keyword> switch
<keyword> do
<keyword> cin
<keyword> cout
  Variable  Type Value Size line of change
<ID> ARRAY qwe int | 0 0 0 | 12 2
<ID> REGULAR zxc int 0 4 3
<ID> REGULAR d int 3 4 6
<ID> REGULAR b int 5 4 6
<ID> REGULAR a int 2 4 6
flame-alchemist@amaterasu:~/Desktop/CD_Lab/project$
```

2)Intermediate code generation

```
goto Lstart

L1:
t0=4/3
t1=2*t0
b=t1
t2=4/3
t3=t1+t2
t4=t3+t3
a=t4
goto Lnext

L2:
t5=4/t3
t6=t1*t5
a=t1
t7=4/t3
t8=t6*t7
t9=t3+t8
b=t9
t10=4/t3
t10=t8*t7
t10=t7+t8
d=t3
goto Lnext

L3:
goto Lnext

Lstart:
if a==1 goto L1
if a==2 goto L2
Lnext:
```

Architecture of Language

- C++ programs start with the
  Headers:
  #include<iostream.h>
  And
  #include<stdio.h>
- After this declarations may be made in the format
  Datatype identifier; eg. int a;
- Followed by the main function
  main(){
- Within main,further declarations may be made. Our constructs may be coded here as well. Their syntax is as follows
- switch (n)
  {
  case 1: // code to be executed if n = 1;
  break;
  case 2: // code to be executed if n = 2;
  break;
  default: // code to be executed if n doesn't match any cases
  }

- do {
  statement(s);
  }
  while(condition);
- Then the main function is closed

A C++ program has the above mentioned architecture. It may also contain i/o statements such as cin and cout.All of this has been accounted for in the grammar.

Literature Survey

Book:

Compilers - principles, technique and tools by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman

Links:

http://dinosaur.compilertools.net/lex/index.html
http://dinosaur.compilertools.net/yacc/
https://sunilkumarn.wordpress.com/2010/10/19/common-subexpression-elimination-cse/
http://moss.csc.ncsu.edu/~mueller/codeopt/codeopt00/y_man.pdf

Context free grammar

S:
START DECLARE LOOP END | START DECLARE BODY END

START:
HEADER1 HEADER2 HEADER3'{' | HEADER1 HEADER3'{';

LOOP:
DO'{'E'}'WHILE'('COND')";';

BODY:
SWITCH'('T')"{'CASES'}';

CASES:
CASE T':'E | CASE T':'E CASES | DEFAULT':'E;
END:
'}';

T:
ID | NUM | STRING;

E:
ASSIGN E | DECLARE E | X';'E | BREAK';' | OUT E | IN E |/*empty*/ ;

DECLARE:
initInt ID ';'|initInt ID asop X';'

```
ASSIGN:
ID asop X';'

X:
T '+' X
|T '-' X
|T '*' X
|T '/' X
|T '%' X
|T AND X
|T OR X
| unP T
| unN T
| '!' T
| T unP
| T unN
| T
| '('X')'
COND:
X '<' COND
|X '>' COND
|X LE COND
|X GE COND
|X EQ COND
|X NE COND| X ;

OUT:
cout openOut T OUT1 ;

OUT1:
openOut T OUT1 | ';';

IN:
cin openIn ID IN1 ;

IN1:
```

openIn ID IN1 | ';';
asop: '=';

Design Strategies

Symbol Table

We have designed our symbol table as an array and to contain three different type of entries:

1. Operator
2. Keyword
3. Identifier

Apart from its value and name, an identifier entry also contains the following information:

- The type of identifier
- Whether the identifier refers to an array or non-array data
- The line number it was last changed
- The size of the memory referred to by the identifier in bytes

Every time an identifier variable is accessed or declared, the line number in which it was last changed is updated. Value updates are also kept track of.
Keyword and operator entries are added at initialization time itself instead of dynamically during parsing.

Abstract Syntax Tree

Our abstract syntax tree has been designed in such a way that it is an n-ary tree, where each node can have upto 15 children.
The AST has been made for the entire grammar, from the start symbol S to the terminals.

The parentheses format has been made use of for printing the tree.

Intermediate Code Generation

Implementation of icg is done is using an array of temporary variables which will be holding the evaluation value of a three address expression format. The icg would be written on a file named icg.txt instead of displaying it on the terminal.

Code Optimization

Code optimization is done by implementing CSE(common subexpression elimination). Implementation is done with the help of array of temporary variables and searching for existing values; based on this the icg code is written on the file.

Error Handling

Any input (token) that does not match or cannot be parsed by the defined grammar is considered as an error according to our code. Types of error handled include :

- syntax error
- variable redeclaration
- type error
- uninitialized variable
  etc.

Implementation Details

Symbol Table

The symbol table by making use of an array of nodeType structures. Each nodeType structure contains information regarding the entry in the symbol table in the form of an enum(identifier, operator or keyword), as well as the information in the symbol table itself.
The following functions were implemented

- i**nitSymbolTable()**
  Initializes the symbol table along with all keywords and operators
- **check_symbol_table()**
  Every time an identifier is encountered the symbol table array is traversed sequentially from start to end to check whether it is already present in the table. If it is, the value and last line number are updated. Else,the identifier is added to the symbol table.
- **print_symbol_table()**
  Prints the entire symbol table contents along the attributes of each entry, traversing it sequentially.

Abstract Syntax Tree

The AST was implemented by making the datatype corresponding the yacc stack a node in the AST.
The tree node structure contains the token information, value information and other necessary information required for parsing along with an array containing pointers to that node's children. This process occurs at every stage in parsing, thereby forming the entire tree

Every time a semantic action occurs, the parent node(LHS) has memory allocated to it and its children are assigned to it using the **insertChildren()** function. The values corresponding to the nodes in the tree are synthesized attributes and travel up the tree.
printTree() is a function which prints the tree in parentheses format, while carrying out inorder traversal.

Intermediate Code Generation

Implementation of this phases comprises of 4 major functions:

1. icg_func1()
2. icg_func2()
3. func_lstart()
4. icg_label()

icg_func1() and **icg_func2()** is responsible for creation and assignment of temporary variables for every expression present. This generates the three address code format as a result. Our implementation includes an array of temporary variables (size=10) which will hold the value of one computation (ex: t0=1*2). This is done by the function icg_func1(). The job of icg_func2() is to search for the temporary variable and use it in an assignment operation (ex: a=t0).

func_lstart() is used to create a entries for starting label (Lstart) in switch-case construct. It consists of a loop for all the labels generated and adds the line “if <condition> goto <label>” in case of switch-case.

icg_label() is responsible for creating icg code for conditions in do-while construct.

Code Optimization

Implementation of this requires addition of a new function called find_CSE(). This basically searches for whether a value exists in the list of temporary variables, and returns the index if it exists, -1 otherwise.
As a result if it does not exist the similar code of icg is executed. If it exists it returns the icg_func1() function without creating a new temporary variable for the existing value.

Error Handling

On encountering an error the yyerror() function is called which would display the line number in the code as well as a message related to the kind of error.
The line number is determined using the inbuilt yylineno feature in yacc. The required message is sent as a parameter to yyerror().

Results and Shortcomings

Result:

Our compiler parses and handles both the constructs **do-while** and **switch-case** of C++.It ignores the white spaces, tab and comments as mentioned in requirement. It generates the symbol table for necessary tokens; AST as the part of syntax and semantic analysis; and ICG along with optimization of the same as part of last phase of a compiler.

Shortcomings:

Our compiler does not handle data types such as union and structures, linked list (pointers in general), OOPs concepts such as classes and objects

Snapshots

- **Switch-case**

  Input:

```
#include<iostream.h>
int qwe[3];
int zxc;
main()
{
        int d=10;int b;int a=20;
        switch(a)
        {
                case 1:         b=2*4/3;
                        a=3+2+4/3;
                        break;
                case 2: a=2*4/3;
                        b=3+2*4/3;
                        d=1+2*4/3;
                        break;
                default:break;
        }
}
```

1) Symbol table

```
Symbol Table:**********************************
<operator> +
<operator> -
<operator> *
<operator> /
<operator> <
<operator> >
<operator> =
<operator> L
<operator> G
<operator> E
<operator> I
<operator> D
<operator> O
<operator> A
<keyword> do
<keyword> while
<keyword> break
<keyword> default
<keyword> case
<keyword> switch
<keyword> do
<keyword> cin
<keyword> cout
  Variable  Type Value Size line of change
<ID> ARRAY qwe int | 0 0 0 | 12 2
<ID> REGULAR zxc int 0 4 3
<ID> REGULAR d int 3 4 6
<ID> REGULAR b int 5 4 6
<ID> REGULAR a int 2 4 6
flame-alchemist@amaterasu:~/Desktop/CD_Lab/project$
```

2) Abstract Syntax Tree

```
( S ( START ( HEADER1 )( DECLARE2 ( DECLARE1  ID ) DECLARE2 )( HEADER3 ))( DECLA
RE2 ( DECLARE1  ID  ID )( DECLARE2 ( DECLARE1  ID )( DECLARE2 ( DECLARE1  ID  ID
 ) DECLARE2 )))( BODY  SWITCH ( (null)  a )( CASES  CASE ( T  NUM )( E ( ASSIGN
 b ( X ( 2 ) * ( 1 )))( E ( ASSIGN  a ( X ( 3 ) + ( X ( 2 ) + ( X ( 4 ) / ( 3 ))
)))( X  3 )))( CASES  CASE ( T  NUM )( E ( ASSIGN  a ( X ( 2 ) * ( 1 )))( E ( AS
SIGN  b ( X ( 3 ) + ( X ( 2 ) * ( 1 ))))( E ( ASSIGN  d ( X ( 1 ) + ( X ( 2 ) *
( 1 )))) END )))( CASES  DEFAULT  END )))) END )
```

3)Intermediate Code Generation

```
goto Lstart

L1:
t0=4/3
t1=2*t0
b=t1
t2=4/3
t3=t1+t2
t4=t3+t3
a=t4
goto Lnext

L2:
t5=4/t3
t6=t1*t5
a=t1
t7=4/t3
t8=t6*t7
t9=t3+t8
b=t9
t10=4/t3
t10=t8*t7
t10=t7+t8
d=t3
goto Lnext

L3:
goto Lnext

Lstart:
if a==1 goto L1
if a==2 goto L2
Lnext:
```

4) Code Optimization (CSE)

```
goto Lstart

L1:
t0=4/3
t1=2*t0
b=t1
t2=t1+t0
t3=t2+t2
a=t3
goto Lnext

L2:
t0=4/3
t1=2*t0
a=t1
t2=3+t1
b=t2
t3=t0+t1
d=t3
goto Lnext

L3:
goto Lnext

Lstart:
if a==1 goto L1
if a==2 goto L2
Lnext:
```

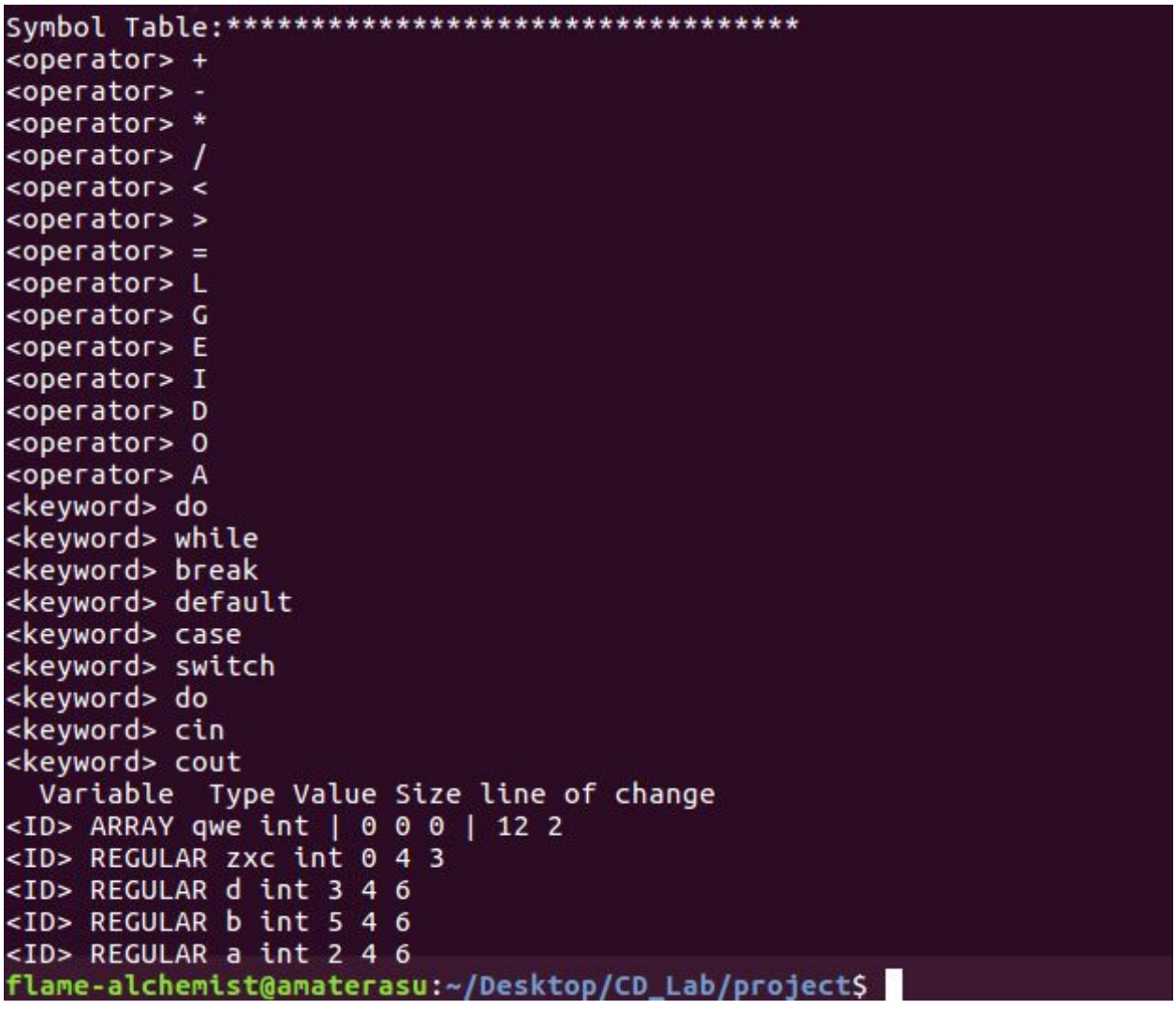
- **Do-while**

  Input:

```
#include<iostream.h>
int qwe[3];
int zxc;
main()
```

```
{
        int d=10;int b;int a=20;
        do{
                int abc=10*2;
                b=2+10*2;
                do{
                        a=12*4/3;
                        b=1+12*4/3;
                }while(abc);
        }while(b);
}
```

1) Symbol Table



2) Abstract Syntax Tree

```
( S ( START ( HEADER1 )( DECLARE2 ( DECLARE1  ID ) DECLARE2 )( HEADER3 ))( DECLA
RE2 ( DECLARE1  ID  ID )( DECLARE2 ( DECLARE1  ID )( DECLARE2 ( DECLARE1  ID  ID
 ) DECLARE2 )))( LOOP  DO ( E ( DECLARE2 ( DECLARE1  ID  ID ) DECLARE2 )( E ( AS
SIGN  b ( X ( 2 ) + ( X ( 10 ) * ( 2 ))))( E ( ASSIGN  a ( X ( 12 ) * ( 1 )))( E
 ( ASSIGN  b ( X ( 1 ) + ( X ( 12 ) * ( 1 ))))( ASSIGN  b ( X ( 1 ) + ( X ( 12 )
 * ( 1 )))))))) WHILE  END ) END )
```

3) Intermediate Code Generation

```
L1:
t0=10*2
abc=t0
t1=10*2
t2=2+t1
b=t2

L2:
t3=4/3
t4=12*t3
a=t4
t5=4/3
t6=t4*t5
t7=t5+t6
b=t7

if t1 goto L2

if t7 goto L1
```

4) Code Optimization (CSE)

```
L1:
t0=10*2
abc=t0
t1=2+t0
b=t1

L2:
t0=4/3
t1=12*t0
a=t1
t2=t0+t1
b=t2

if 20 goto L2

if t2 goto L1
```

Conclusion

- Understanding the working of a compiler
- Hands on experience to build a compiler
- Working of Lex and Yacc programming language
- Different phases of a compiler including
    - Lexical Analyzer (Tokenization)
    - Syntax Analyzer and Semantic Analysis (AST)
    - Intermediate Code Generation
    - Code Optimization (Common Subexpression Elimination)
- Exposure to building CFG of any two constructs of a programming language

Future Enhancements

- Adding more constructs to our code
- Implementing OOPs concept like classes and objects,etc.
- Improvements in error handling techniques
- Incorporating more ranges of data types

References

- LEX & YACC by Tom Niemann
- Compilers - principles, technique and tools by Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman
- www.stackoverflow.com