

***Report on***

**“C++ mini-compiler for while and functions”**

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

***Compiler Design Laboratory***

**Bachelor of Technology**

**in**

**Computer Science & Engineering**

***Submitted by:***

|  |  |
| --- | --- |
| **Sumanth V Rao**  **Sumedh Pb**  **Suraj Aralihalli** | **01FB16ECS402**  **01FB16ECS403**  **01FB16ECS405** |

*Under the guidance of*

|  |
| --- |
| **<Name of the Guide>**  <Designation>  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**

|  |  |  |
| --- | --- | --- |
| **Chapter No.** | **Title** | **Page No.** |
|  | **INTRODUCTION** | **01** |
|  | **ARCHITECTURE OF LANGUAGE:** | **02** |
|  | **LITERATURE SURVEY** | **03** |
|  | **CONTEXT FREE GRAMMAR** |  |
|  | **DESIGN STRATEGY**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING** |  |
|  | **IMPLEMENTATION DETAILS**   * **SYMBOL TABLE CREATION** * **ABSTRACT SYNTAX TREE (internal representation)** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING** |  |
|  | **RESULTS** |  |
|  | **SNAPSHOTS** |  |
|  | **CONCLUSIONS** |  |
|  | **FURTHER ENHANCEMENTS** |  |
| **REFERENCES/BIBLIOGRAPHY** | |  |

**1. INTRODUCTION**

C++ was the language chosen as the basis of this mini compiler. Simple constructs from the

language were implemented. 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. The language the compiler is

designed for is C. The properties we have implemented includes the basic structure of C

which includes initializing variables, creating main and other functions, preprocessor

directives limited to including <stdio.h> and so on.

Sample Input and Expected Output:

Compiler for the following constructs:

do while​ loop

1. while loop
2. if else statements
3. int data typevoid data type
4. char data type
5. basic function declaration and definition
6. arithmetic and logical operators
7. Removal of comments has been taken care of too

**3. 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

**4. CONTEXT-FREE GRAMMAR**

assignment\_expression: conditional\_expression| unary\_expression assignment\_operator

assignment\_expression;

statement: compound\_statement| expression\_statement| selection\_statement|

iteration\_statement| jump\_statement;

compound\_statement: '{' '}'| '{' statement\_list '}'| '{' declaration\_list '}'| '{' declaration\_list

statement\_list '}';

statement\_list: statement| statement\_list statement;expression\_statement: ';'| expression ';';

function\_definition: declaration\_specifiers declarator declaration\_list compound\_statement|

declaration\_specifiers declarator compound\_statement| declarator declaration\_list

compound\_statement| declarator compound\_statement;

primary\_expression: IDENTIFIER | CONSTANT | STRING\_LITERAL| '(' expression ')';

unary\_expression: postfix\_expression| INC\_OP unary\_expression| DEC\_OP unary\_expression|

unary\_operator unary\_expression| SIZEOF unary\_expression| SIZEOF '(' type\_name ')';

unary\_operator: '&'| '\*'| '+'| '-'| '~'| '!';

multiplicative\_expression: cast\_expression| multiplicative\_expression '\*'

cast\_expression|multiplicative\_expression '/' cast\_expression| multiplicative\_expression '%'

cast\_expression;

additive\_expression : multiplicative\_expression | additive\_expression

'+'multiplicative\_expression| additive\_expression '-' multiplicative\_expression;

relational\_expression: shift\_expression| relational\_expression '<' shift\_expression

|relational\_expression '>' shift\_expression| relational\_expression LE\_OP

shift\_expression|relational\_expression GE\_OP shift\_expression;

equality\_expression : relational\_expression| equality\_expression EQ\_OP relational\_expression|

equality\_expression NE\_OP relational\_expression;

assignment\_expression: conditional\_expression| unary\_expression assignment\_operator

assignment\_expression;

assignment\_operator: '=';

selection\_statement: IF '(' expression ')' statement| IF '(' expression ')' statement ELSE

statement;

iteration\_statement : WHILE '(' expression ')' statement| DO statement WHILE '(' expression ')'';'

**5. DESIGN STRATEGY**

**● Symbol table creation** - The symbol table was implemented in a linear array structure

that contains the identifier and its value. Function scope has been implemented as well

● **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 ifFalsestatements 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

**6. IMPLEMENTATION DETAILS**

Lex and Yacc were used to implement the following:

● **Symbol table creation** - Implemented in symtab.y

The symbol table is a linear array of the following structure

struct symtab

{

char identifier[20];

char type[20];

char attribute[20];

int val;

char pars[100];

char scope[20];

int spec;

};

● **Abstract Syntax Tree** - Implemented in ast.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 node

{

struct node \*left;

struct node \*right;

char \*token;

} node;

● **Intermediate Code Generation** - Implemented in icg.y

The given code was converted into 3 address code

● **Code Optimization** - Implemented in opt.y

**Constant Folding**

Using the function alldigits we perform this check. If the operands are all numbers.

We evaluate the expression immediately and store it in the symbol table.

**Constant Propagation**

When an identifier is found in symbol table the synthesised attribute of the

non-terminal, code stores the value of the identifier, else it stores the name of the

identifier itself.

● **Error Handling** - Implemented in symtab.y using conditional statements for type error and

semicolon missing is checked using the grammar

**7. RESULTS**

A mini compiler that can compile the chosen constructs was obtained.

**8. SNAPSHOTS**

**9. CONCLUSIONS**

A compiler for C++ was thus created using lex and yacc.In addition to the

constructs specified, basic building blocks of the language (functions, 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). Each of these components are

required to compile code successfully.

In addition to this, basic error recovery has also been implemented.

Through this process, all kinds of syntax errors and certain semantic errors in a

C++ program can be caught by the compiler.

**10. FURTHER ENHANCEMENTS**

**●** Functionality for for-loop can be implemented.

● The compiler can be constructed to recover from more kinds of errors

● Currently, the input given to the compiler must be delimited by a ~. The need for this

delimiter could be eliminated in the future.