

***Report on***

**MINI COMPILER IN C++ FOR SWITCH AND FOR CONSTRUCT**

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

***Compiler Design Laboratory***

**Bachelor of Technology**

**in**

**Computer Science & Engineering**

***Submitted by:***

|  |  |
| --- | --- |
| **Abhijit Mohanty**  **Gagan Mahesh**  **TS Yogesh** | **PES1201801293**  **PES1201800769**  **PES1201801760** |

*Under the guidance of*

|  |
| --- |
| **<Name of the Course Instructor>**  <Designation>  PES University, Bengaluru |

**January – May 2021**

**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 (Mini-Compiler is built for which language. Provide sample input and output of your project)** | **01** |
|  | **ARCHITECTURE OF LANGUAGE:**   * **What all have you handled in terms of syntax and semantics for the chosen language.** | **02** |
|  | **LITERATURE SURVEY (if any paper referred or link used)** | **03** |
|  | **CONTEXT FREE GRAMMAR (which you used to implement your project)** |  |
|  | **DESIGN STRATEGY (used to implement the following)**   * **SYMBOL TABLE CREATION** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** |  |
|  | **IMPLEMENTATION DETAILS (TOOL AND DATA STRUCTURES USED in order to implement the following):**   * **SYMBOL TABLE CREATION** * **INTERMEDIATE CODE GENERATION** * **CODE OPTIMIZATION** * **ERROR HANDLING *-* strategies and solutions used in your Mini-Compiler implementation (in its scanner, parser, semantic analyzer, and code generator).** * **Provide instructions on how to build and run your program.** |  |
|  | **RESULTS AND possible shortcomings of your Mini-Compiler** |  |
|  | **SNAPSHOTS (of different outputs)** |  |
|  | **CONCLUSIONS** |  |
|  | **FURTHER ENHANCEMENTS** |  |
| **REFERENCES/BIBLIOGRAPHY** | |  |

INTRODUCTION

Compiler design course is one of the most interesting course among core computer fields as this provides a feel of how computers and compilers work, hence we tried to mimic the C++ compiler for switch and for loop construct. We perform lexical analysis, semantic analysis, we generate intermediate code and finally optimized code in quadruple format.

The motivation behind this project was to better explain the communication between programmer and computer and to present how the compiler works, this project can help others in understanding the fine details behind the great engineering work of computers.

The research behind this project comes from several sources including class notes and lectures.

INPUT:

A .cpp file with main function and multiple statements such as assignment, for statement and switch statement and the output will be multiple files which are Tokens.txt, intermediate code.txt(TAC), optimized\_code.txt(TAC) and the quadruple version of intermediate and optimized code.( The files are attached)

#include<iostream>

using namespace std;

int main()

{

    //Single line comment

    float a = 10+5\*4;

    int b = 12\*7-80;

    int k = 3\*3;

    int m = 5;

    int c = m\*2;

    int i = 0;

    int j = 0;

    {

        int i = 5;

        {

            int ad = 8;

            //ad = 10;

        }

        //ad = 1;

    }

    /\*

    Multi line comment;

    \*/

    a = 10+3;

    for(int j=0;j<5;++j)

    {

        int i = j + 2;

    }

    switch(c)

    {

        case 1:

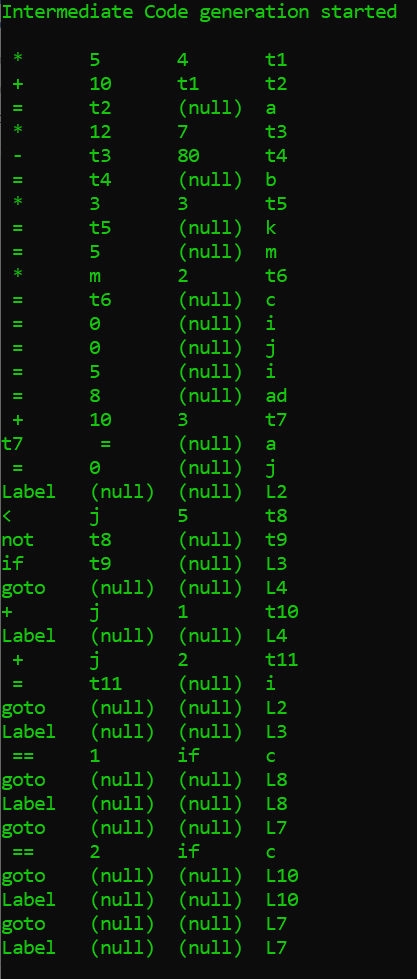
            break;

        case 2:

            break;

    }

}



ARCHITECTURE OF LANGUAGE

We have tried to handle almost all the edge cases ranging from type mismatch, syntax errors, undeclared variables, and for the construct we have handled nested looping statements and for switch we have handled the cases of multiple cases and breaks and as well as defaults.

Examples:

Switch(m)

{

Case 1 :

Break;

}

Output:

Undeclared variable m

Hence we have to define the variable m

int a = “abc”;

error : Type mismatch error

we have also considered the case of redeclaration errors hence we can not declare the variable again but we can declare the variable in the higher or lower scope not the same scope.

The for loop can be nested as well

for(int i=0;i<3;i++)

{

for(int j=0;j<3;j++)

{

}

}

LITERATURE SURVEY

We basically referred to mainly the book titled “LEX & YACC TUTORIAL by Tom Niemann” and documentation by multiple universities such as IIT KGP and document titled “Lex and Yacc: A Brisk Tutorial Saumya K. Debray Department of Computer Science

The University of Arizona”.

These documents and book contains all the required details to start with lex and yacc tool.

These reference materials helped a lot in performing the project successfully.

CONTEXT FREE GRAMMAR

start:T\_INT T\_MAIN T\_OPEN T\_CLOSE comp\_stat

comp\_stat: T\_OPBRACE  stat T\_CLBRACE

         ;

stat:E T\_SC stat

    |comp\_stat

    |{char \* lab = new\_label(); push(lab);} select\_stat

    | {char \* lab = new\_label(); push(lab);} iter\_stat {pop();} stat

    | jump\_stat stat

    |decl stat

    ;

ST  : T\_SWITCH T\_OPEN T\_ID T\_CLOSE T\_OPBRACE

B   : C

    | C D

    | C B

    ;

C   : T\_CASE T\_NUM T\_COLON stat

D   : T\_DEFAULT T\_COLON stat

    ;

select\_stat : ST   {$$=$1;}

           ;

iter\_stat : T\_FOR T\_OPEN Type assign\_expr\_for cond T\_SC unary\_expr T\_CLOSE comp\_stat

assign\_expr1 :T\_ID T\_ASSIGN E T\_COMMA assign\_expr1

           |T\_ID T\_ASSIGN E T\_SC

;

assign\_expr\_for :T\_ID T\_ASSIGN E T\_COMMA assign\_exprfor

           |T\_ID T\_ASSIGN E T\_SC

;

jump\_stat : T\_BREAK T\_SC

         |T\_RETURN E T\_SC

cond :relexp

    |E

    ;

relexp  :E relOp E

      ;

relOp :T\_LESEQ

     |T\_GRTEQ

     |T\_NOTEQ

     |T\_EQEQ

     |T\_LESS

     |T\_GREAT

     ;

decl :Type assign\_expr1

;

Type:T\_INT

    |T\_FLOAT

    | T\_CHAR

    ;

E:E T\_PLUS T

 |E T\_MINUS

T:T T\_MULT F

 |F

 ;

F:T\_

 |T\_NUM

 |T\_OPEN E T\_CLOSE

 |unary\_expr

 |s\_operation

 ;

s\_operation : T\_ID s\_op T\_ID

               | T\_ID s\_op T\_NUM

               | T\_ID s\_op T\_OPEN E T\_CLOSE

               ;

s\_op :T\_SPLUS

    |T\_SMINUS

    |T\_SMULT

    |T\_SDIV

    ;

unary\_expr :        T\_INC T\_ID

          |T\_ID T\_INC

          |T\_DEC T\_

          |T\_ID T\_DEC

          | T\_MINUS T\_ID

          | T\_MINUS T\_NUM

DESIGN STRATEGY

SYMBOL TABLE:

We used a structure entry and created an array of structure called symbol\_table in which we wrote multiple helper functions to check at every stage that whether the variable is declared or not, we are updating the symbol table and before another updation we are calling check\_mult\_decl function which will return true if the variable is declared or not hence it will raise the error as redeclaration error.

struct entry

{

    char name[30];

    char type[10];

    int width;

    int line\_num;

    int scope;

    char symbol\_table\_value[10];

};

struct entry sym\_tab[100];

int ctr = 0;

ctr is taking count of the symbols getting added.

INTERMEDIATE CODE GENERATION:

The structure which we used for intermediate code storage is called quadruples

typedef struct quadruples

{

    char \*op;

    char \*arg1;

    char \*arg2;

    char \*res;

}quad;

int quadlen = 0;

quad q[100];

we created an array of structures in which we updated the intermediate code by using the helper function called code\_gen

In code\_gen function we are passing variable number arguments and for this purpose we included stdarg.h header file. Based on the number of arguments passed we are assigning the values to operator, argument1, argument2 and result.

The interesting point to note is that sometimes we can have 3 arguments sometimes we can have 4 and so many others and based on that we will assign for example arg1 = NULL hence in the output it will show that = 10 (null) a which means a = 10.

OPTIMIZATION:

For optimization also we used a similar structure which is called quad

typedef struct Quad

{

    char \*op;

    char \*arg1;

    char \*arg2;

    char \*res;

}Quad;

int qlen = 0;

quad Q[100];

which will store all the optimized quadruple format code. The underlying working is similar to intermediate code generation. We are storing the buffer outputs in the global variables and we know that a = 10\*4 + 5 will be 45 hence for constant folding we are passing the final buffered output hence it need not store it in temporaries.

ERROR HANDLING

We are handling multiple errors starting from redeclaration errors, undeclared variable error, we are also handling the use of declaring the variables in other scope, we are handling type mismatch errors. For redeclaration, undeclared variable we are traversing the symbol table and we have multiple helper function which helps in identifying if the variable is declared once again or not. We are checking the scope and the variable name to match the whole symbol table.

IMPLEMENTATION DETAILS

The whole project is done using lex and yacc tool, lex is used to create lexemes and pass the tokens to parser, **yacc** stands for Yet Another Compiler Compiler. **yacc** provides a tool to produce a parser for a given grammar. **Yacc** is a program designed to compile a LALR (1) grammar. It is **used** to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.

They are intensively used using c language hence basic knowledge of C and concepts like pointers are required.

The implementation is done in multiple stage from lexer and then multiple stages in parser.

The lexer code generates the lexemes and return those tokens to parser program where the function called yyparse() is called to parse the input.

The structures used in these multiple stages are defined below:

SYMBOL TABLE :

The symbol table is an array of structure called entry which has entry such as

Name: To store the name of the token

Type: Type of the token

Width: this field varies from token to token

Int : 4

Char : 1

Float : 8

Line\_num: This contains the line number in the input program for those particular tokens

Scope: To store the scope of the tokens

Symbol table value: This variable is used to store the value associated with the id or token

We created an array of this structure and populated and finally printed using print\_symbol\_table() function.

INTERMEDIATE CODE GENERATION

As mentioned in the above section a structure called quadruple is used to store the quadruple format of TAC code generated. Multiple helper functions are used to perform intermediate code generation.