

# Intermediate Code Generation for the C Language



**National Institute of Technology Karnataka Surathkal**

**Date: 06-06-2020**

**Submitted To:  
Prof. P. Santhi Thilagam**

**Group Members:  
Chandan Naik - 17CO212  
Jelwin Rodrigues - 17CO218  
Siddhartha M - 17CO246**

## Abstract:

This report contains the details on the ICG phase ( Phase four of the Compiler Design). Also, the report summarizes the four phases of compiler design. We have seen the following phases of compiler design:

*Lexical Analysis:* The first phase of the scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens.

*Syntax Analysis:* The next phase takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

*Semantic Analysis:* Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, the assignment of values is between compatible data types and adding a string to an integer. Also, the semantic analyzer keeps track of identifiers, their types, and expressions; whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.

*Intermediate Code Generation:* After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

There are several intermediate codes like postfix notation, syntax tree, 3 address code. We use 3 address code in our project. Thus, the annotated parse tree obtained from the previous semantic phase is converted in a machine-independent linear representation like three address code.

# Objectives:

Following constructs will be handled by the mini-compiler :

- Data Types: char, int with all its subtypes
- Comments: Single line and multiline comments,
- Keywords: char, else, for, if, int, long, return, short, signed, struct, unsigned, void, while, main
- Identification of valid identifiers used in the language,
- Looping Constructs: It will support nested for and while loops.
- Conditional Constructs: if...else-if...else statements,
- Operators: addition(+), subtraction(-), multiplication(\*), division(/), modulo(%) etc.
- Delimiters: semicolon(;),comma(,)
- Function construct of the language: Syntax: int func(int x)
- Support for nested conditional statements and nested loops.
- Support for a 1-dimensional array

## Contents:

1. Introduction
  - a. Intermediate Code Generation
  - b. Yacc Script
  - c. C program
2. Design of Programs
  - a. Code
  - b. Explanation
3. Test Cases
  - a. Without Errors
  - b. With Errors
4. Implementation
5. Results / Future Work
6. References

## List of Figures and Tables:

1. **Figure 1:** Output for test case of a simple c program to validate the compiler
2. **Figure 2:** Output for test case containing loop, function and assignment operation
3. **Figure 3:** Output for test case having arithmetic operations
4. **Figure 4:** Output for test case having conditional statements
5. **Figure 5:** Output for test case having illegal identifier
6. **Figure 6:** Output for test case with use of function before declaration/definition
7. **Figure 7:** Output for test case with mismatched number of braces
8. **Figure 8:** Output for test case having varying return type

# Introduction:

## Intermediate Code Generation:

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine). Because of the machine-independent intermediate code, portability will be enhanced. For example, suppose, if a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required. Because, obviously, there were some modifications in the compiler itself according to the machine specifications. Retargeting is facilitated and it is easier to apply source code modification to improve the performance of source code by optimizing the intermediate code.

Intermediate Code can be represented in many different formats depending on whether it is language-specific (e.g. Bytecode for Java) or language-independent (three-address-code). We have used a three-address-code to make it independent.

Most common independent intermediate representations are:

1. Postfix notation
2. Three Address Code
3. Syntax tree

### Three Address Code

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent expression and the value computed at each instruction is stored in a temporary variable generated by the compiler. The compiler decides the order of operation given by three address code.

General representation:  $x = y \text{ op } z$

An address can be a name, constant or temporary.

- Assignments  $x = y \text{ op } z$ ;  $x = \text{op } y$ .
- Copy  $x = y$ .
- Unconditional jump `goto L`.
- Conditional jumps `if x RELOP y goto L`.
- Parameters `param x`.
- Function call `y = call p`

## Yacc Script:

Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures of his input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process frequently, it is convenient and appropriate to have most of the flow of control in the user's application handled by this subroutine. The lexer can be used to make a simple parser. But it needs making extensive use of the user-defined states.

The input subroutine produced by Yacc calls a user-supplied routine to return the next basic input item. Thus, the user can specify his input in terms of individual input characters, or in terms of higher-level constructs such as names and numbers. The user-supplied routine may also handle idiomatic features such as comment and continuation conventions, which typically defy easy grammatical specifications.

Yacc is written in portable C. The class of specifications accepted is a very general one: LALR(1) grammars with disambiguating rules.

The structure of our Yacc script is given below; files are divided into three sections, separated by lines that contain only two percent signs, as follows:

```
Definition section
%%
Rules section
%%
C code section
```

The definition section defines macros and imports header files written in C. It is also possible to write any C code here, which will be copied verbatim into the generated source file.

In the rules section, each grammar rule defines a symbol in terms of:

1. Other symbols
2. Tokens (or terminal symbols) that come from the lexer.

Each rule can have an associated action, which is executed *after* all the component symbols of the rule have been parsed. Actions are basically C-program statements surrounded by curly braces.

The C code section contains C statements and functions that are copied verbatim to the generated source file. These statements presumably contain code called by the rules in the rules section. In large programs, it is more convenient to place this code in a separate file linked in at compile time.

## C Program:

This section describes the input C program which is fed to the yacc script for parsing. The workflow is explained as under:

1. Compile the script using Yacc tool

```
$ yacc -d parser.y
```

2. Compile the flex script using Flex tool

```
$ flex lexer.l
```

3. After compiling the lex file, a lex.yy.c file is generated. Also, y.tab.c and y.tab.h files are generated after compiling the yacc script.
4. The three files, lex.yy.c, y.tab.c and y.tab.h are compiled together with the options `-ll` and `-ly`

```
$ gcc -o compiler lex.yy.c y.tab.h y.tab.c -ll -ly
```

5. The executable file is generated, which on running parses the C file given as a command-line input

```
$ ./compiler test.c
```

The script also has an option to take standard input instead of taking input from a file.

# Design of Programs:

## Lexer Code:

```
%{  
  
#include <stdio.h>  
#include <string.h>  
#include <stdlib.h>  
#include "y.tab.h"  
  
#define ANSI_COLOR_RED          "\x1b[31m"  
#define ANSI_COLOR_GREEN "\x1b[32m"  
#define ANSI_COLOR_YELLOW "\x1b[33m"  
#define ANSI_COLOR_BLUE "\x1b[34m"  
#define ANSI_COLOR_MAGENTA "\x1b[35m"  
#define ANSI_COLOR_CYAN "\x1b[36m"  
#define ANSI_COLOR_RESET "\x1b[0m"  
  
struct symboltable  
{  
    char name[100];  
    char class[100];  
    char type[100];  
    char value[100];  
    int nestval;  
    int lineno;  
    int length;  
    int params_count;  
}ST[1001];  
  
struct constanttable  
{  
    char name[100];  
    char type[100];  
    int length;  
}CT[1001];  
  
int currnest = 0;  
extern int yynval;  
  
int hash(char *str)  
{  
    int value = 0;  
    for(int i = 0 ; i < strlen(str) ; i++)  
    {  
        value = 10*value + (str[i] - 'A');  
    }  
}
```



```

        value = value % 1001;
        while(value < 0)
            value = value + 1001;
    }
    return value;
}

int lookupST(char *str)
{
    int value = hash(str);
    if(ST[value].length == 0)
    {
        return 0;
    }
    else if(strcmp(ST[value].name, str) == 0)
    {
        return value;
    }
    else
    {
        for(int i = value + 1 ; i != value ; i = (i+1)%1001)
        {
            if(strcmp(ST[i].name, str) == 0)
            {
                return i;
            }
        }
        return 0;
    }
}

int lookupCT(char *str)
{
    int value = hash(str);
    if(CT[value].length == 0)
        return 0;
    else if(strcmp(CT[value].name, str) == 0)
        return 1;
    else
    {
        for(int i = value + 1 ; i != value ; i = (i+1)%1001)
        {
            if(strcmp(CT[i].name, str) == 0)
            {
                return 1;
            }
        }
    }
}

```

```

        return 0;
    }
}

void insertSTline(char *str1, int line)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,str1)==0)
        {
            ST[i].lineno = line;
        }
    }
}

void insertST(char *str1, char *str2)
{
    if(lookupST(str1))
    {
        if(strcmp(ST[lookupST(str1)].class,"Identifier")==0 &&
strcmp(str2,"Array Identifier")==0)
        {
            printf("Error use of array\n");
            exit(0);
        }
        return;
    }
    else
    {
        int value = hash(str1);
        if(ST[value].length == 0)
        {
            strcpy(ST[value].name,str1);
            strcpy(ST[value].class,str2);
            ST[value].length = strlen(str1);
            ST[value].nestval = 9999;
            ST[value].params_count = -1;
            insertSTline(str1,yylineno);
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
        {
            if(ST[i].length == 0)
            {
                pos = i;
            }
        }
    }
}

```

```

        break;
    }
}

strcpy(ST[pos].name, str1);
strcpy(ST[pos].class, str2);
ST[pos].length = strlen(str1);
ST[pos].nestval = 9999;
ST[pos].params_count = -1;
}
}

void insertSTtype(char *str1, char *str2)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name, str1) == 0)
        {
            strcpy(ST[i].type, str2);
        }
    }
}

void insertSTvalue(char *str1, char *str2)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name, str1) == 0 && ST[i].nestval == currnest)
        {
            strcpy(ST[i].value, str2);
        }
    }
}

void insertSTnest(char *s, int nest)
{
    if(lookupST(s) && ST[lookupST(s)].nestval != 9999)
    {
        int pos = 0;
        int value = hash(s);
        for (int i = value + 1 ; i != value ; i = (i+1)%1001)
        {
            if(ST[i].length == 0)
            {
                pos = i;
                break;
            }
        }
    }
}

```

```

        strcpy(ST[pos].name,s);
        strcpy(ST[pos].class,"Identifier");
        ST[pos].length = strlen(s);
        ST[pos].nestval = nest;
        ST[pos].params_count = -1;
        ST[pos].lineno = yylineno;
    }
    else
    {
        for(int i = 0 ; i < 1001 ; i++)
        {
            if(strcmp(ST[i].name,s)==0 )
            {
                ST[i].nestval = nest;
            }
        }
    }
}

void insertSTparamscount(char *s, int count1)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,s)==0 )
        {
            ST[i].params_count = count1;
        }
    }
}

int getSTparamscount(char *s)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,s)==0 )
        {
            return ST[i].params_count;
        }
    }
    return -1;
}

void insertSTF(char *s)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(strcmp(ST[i].name,s)==0 )

```

```

        {
            strcpy(ST[i].class,"Function");
            return;
        }
    }

}

void insertCT(char *str1, char *str2)
{
    if(lookupCT(str1))
        return;
    else
    {
        int value = hash(str1);
        if(CT[value].length == 0)
        {
            strcpy(CT[value].name,str1);
            strcpy(CT[value].type,str2);
            CT[value].length = strlen(str1);
            return;
        }

        int pos = 0;

        for (int i = value + 1 ; i!=value ; i = (i+1)%1001)
        {
            if(CT[i].length == 0)
            {
                pos = i;
                break;
            }
        }

        strcpy(CT[pos].name,str1);
        strcpy(CT[pos].type,str2);
        CT[pos].length = strlen(str1);
    }
}

void deletedata (int nesting)
{
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(ST[i].nestval == nesting)
        {
            ST[i].nestval = 99999;
        }
    }
}

```

```
}
```

```
int checkscope(char *s)
```

```
{
```

```
    int flag = 0;
```

```
    for(int i = 0 ; i < 1000 ; i++)
```

```
    {
```

```
        if(strcmp(ST[i].name,s)==0)
```

```
        {
```

```
            if(ST[i].nestval > currnest)
```

```
            {
```

```
                flag = 1;
```

```
            }
```

```
        else
```

```
        {
```

```
            flag = 0;
```

```
            break;
```

```
        }
```

```
    }
```

```
}
```

```
if(!flag)
```

```
{
```

```
    return 1;
```

```
}
```

```
else
```

```
{
```

```
    return 0;
```

```
}
```

```
}
```

```
int check_id_is_func(char *s)
```

```
{
```

```
    for(int i = 0 ; i < 1000 ; i++)
```

```
    {
```

```
        if(strcmp(ST[i].name,s)==0)
```

```
        {
```

```
            if(strcmp(ST[i].class,"Function")==0)
```

```
                return 1;
```

```
        }
```

```
    }
```

```
    return 0;
```

```
}
```

```
int checkarray(char *s)
```

```
{
```

```
    for(int i = 0 ; i < 1000 ; i++)
```

```
    {
```

```

        if(strcmp(ST[i].name,s)==0)
        {
            if(strcmp(ST[i].class,"Array Identifier")==0)
            {
                return 0;
            }
        }
    }
    return 1;
}

int duplicate(char *s)
{
    for(int i = 0 ; i < 1000 ; i++)
    {
        if(strcmp(ST[i].name,s)==0)
        {
            if(ST[i].nestval == currnest)
            {
                return 1;
            }
        }
    }

    return 0;
}

int check_duplicate(char* str)
{
    for(int i=0; i<1001; i++)
    {
        if(strcmp(ST[i].name, str) == 0 && strcmp(ST[i].class,
"Function") == 0)
        {
            printf("Function redeclaration not allowed\n");
            exit(0);
        }
    }
}

int check_declaration(char* str, char *check_type)
{
    for(int i=0; i<1001; i++)
    {
        if(strcmp(ST[i].name, str) == 0 && strcmp(ST[i].class,
"Function") == 0 || strcmp(ST[i].name,"printf")==0 )
        {
            return 1;
        }
    }
}

```

```

    }
    return 0;
}

int check_params(char* type_specifier)
{
    if(!strcmp(type_specifier, "void"))
    {
        printf("Parameters cannot be of type void\n");
        exit(0);
    }
    return 0;
}

char gettype(char *s, int flag)
{
    for(int i = 0 ; i < 1001 ; i++ )
    {
        if(strcmp(ST[i].name,s)==0)
        {
            return ST[i].type[0];
        }
    }
}

void printST()
{
    printf("%10s | %15s | %10s | %10s | %10s | %15s | %10s |\n", "SYMBOL",
"CLASS", "TYPE", "VALUE", "LINE NO", "NESTING", "PARAMS COUNT");
    for(int i=0;i<100;i++) {
        printf("-");
    }
    printf("\n");
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(ST[i].length == 0)
        {
            continue;
        }
        printf("%10s | %15s | %10s | %10s | %10d | %15d | %10d
|\n",ST[i].name, ST[i].class, ST[i].type, ST[i].value, ST[i].lineno, ST[i].nestval,
ST[i].params_count);
    }
}

void printCT()
{
    printf("%10s | %15s\n", "NAME", "TYPE");
    for(int i=0;i<81;i++) {

```



```

        printf("-");
    }
    printf("\n");
    for(int i = 0 ; i < 1001 ; i++)
    {
        if(CT[i].length == 0)
            continue;

        printf("%10s | %15s\n",CT[i].name, CT[i].type);
    }
}
char curid[20];
char curtype[20];
char curval[20];

%}
DE "define"
IN "include"
%%
\n      {yylineno++;}
([#][ " "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)(>?))/["\n"|\|| " "|\t"]    {
}
([#][ " "]*({DE})[ " "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\|| " "|\t"]
    { }
\\\/(.*)
                                { }
\\\/*([^*]|[\r\n]|(\*+([^\r\n])*)\*+\\\/
                                { }

[ \n\t] ;
";"                { return(';'); }
","                { return(','); }
("{")              { return('{'); }
("}")              { return('}'); }
"("                { return('('); }
")"                { return(')'); }
"["| "<:"          { return('['); }
"]"| ">:"          { return(']'); }
":"                { return(':'); }
"."                { return('.'); }

"char"              { strcpy(curtype,yytext); insertST(yytext, "Keyword");return
CHAR;}
"double"            { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
DOUBLE;}
"else"              { insertST(yytext, "Keyword"); return ELSE;}
"float"              { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return FLOAT;}
"while"              { insertST(yytext, "Keyword"); return WHILE;}
"do"                { insertST(yytext, "Keyword"); return DO;}

```

```

"for"          { insertST(yytext, "Keyword"); return FOR;}
"if"           { insertST(yytext, "Keyword"); return IF;}
"int"          { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
INT;}
"long"         { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
LONG;}
"return"       { insertST(yytext, "Keyword"); return RETURN;}
"short"        { strcpy(curtype,yytext); insertST(yytext, "Keyword");
return SHORT;}
"signed"       { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
SIGNED;}
"sizeof"       { insertST(yytext, "Keyword"); return sizeof;}
"struct"       { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
STRUCT;}
"unsigned"     { insertST(yytext, "Keyword"); return UNSIGNED;}
"void"         { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
VOID;}
"break"        { insertST(yytext, "Keyword"); return BREAK;}


"++"          { return increment_operator; }
"--"          { return decrement_operator; }
"<<"         { return leftshift_operator; }
">>"         { return rightshift_operator; }
"<="         { return lessthan_assignment_operator; }
"<"          { return lessthan_operator; }
">="         { return greaterthan_assignment_operator; }
">"          { return greaterthan_operator; }
"=="          { return equality_operator; }
"!="          { return inequality_operator; }
"&&"         { return AND_operator; }
"||"          { return OR_operator; }
"^"           { return caret_operator; }
"*="          { return multiplication_assignment_operator; }
"/="          { return division_assignment_operator; }
"%="          { return modulo_assignment_operator; }
"+="          { return addition_assignment_operator; }
"-="          { return subtraction_assignment_operator; }
"<<="        { return leftshift_assignment_operator; }
">>="        { return rightshift_assignment_operator; }
"&="          { return AND_assignment_operator; }
"^="          { return XOR_assignment_operator; }
"|="          { return OR_assignment_operator; }
"&"           { return amp_operator; }
"!"           { return exclamation_operator; }
"~"           { return tilde_operator; }
"_"           { return subtract_operator; }
"+"           { return add_operator; }

```

```

"*"                { return multiplication_operator; }
"/"                { return division_operator; }
%"                { return modulo_operator; }
"|"                { return pipe_operator; }
\"                { return assignment_operator;}

\"^[^\\n]*\"/[;|,|\\)]          {strcpy(curval,yytext);
insertCT(yytext,"String Constant"); return string_constant;}
\\'[A-Z|a-z]\\'/[;|,|\\)]|:]          {strcpy(curval,yytext);
insertCT(yytext,"Character Constant"); return character_constant;}
[a-zA-Z]([a-zA-Z]|\\[0-9])*\\/\\[ {strcpy(curid,yytext); insertST(yytext, "Array
Identifier"); return array_identifier;}
[1-9][0-9]*|0/[;|,|" "\\)|<|>|=|\\!|\\||&|\\+|\\-|\\*|\\/|\\%|~|\\]|\\}|:|\\n|\\t|\\^]
{strcpy(curval,yytext); insertCT(yytext, "Number Constant"); yylval = atoi(yytext);
return integer_constant;}
([0-9]*\\.([0-9]+)/[;|,|" "\\)|<|>|=|\\!|\\||&|\\+|\\-|\\*|\\/|\\%|~|\\]|\\}|:|\\n|\\t|\\^]
{strcpy(curval,yytext); insertCT(yytext, "Floating Constant"); return
float_constant;}
[A-Za-z_][A-Za-z_0-9]* {strcpy(curid,yytext); insertST(curid,"Identifier"); return
identifier;}
(\\.?) {
    if(yytext[0]=='#')
    {
        printf("Error in Pre-Processor directive at line no.
%d\\n",yylineno);
    }
    else if(yytext[0]=='/')
    {
        printf("ERR_UNMATCHED_COMMENT at line no. %d\\n",yylineno);
    }
    else if(yytext[0]=='"')
    {
        printf("ERR_INCOMPLETE_STRING at line no. %d\\n",yylineno);
    }
    else
    {
        printf("ERROR at line no. %d\\n",yylineno);
    }
    printf("%s\\n", yytext);
    return 0;
}
%%

```

Parser Code:

```
%{
```

```

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

void yyerror(char* s);
int yylex();
void ins();
void insV();
int flag=0;
#define ANSI_COLOR_RED          "\x1b[31m"
#define ANSI_COLOR_GREEN        "\x1b[32m"
#define ANSI_COLOR_CYAN         "\x1b[36m"
#define ANSI_COLOR_RESET        "\x1b[0m"
extern char curid[20];
extern char curtype[20];
extern char curval[20];
extern int currnest;
void deletedata (int );
int checkscope(char*);
int check_id_is_func(char *);
void insertST(char*, char*);
void insertSTnest(char*, int);
void insertSTparamscount(char*, int);
int getSTparamscount(char*);
int check_duplicate(char*);
int check_declaration(char*, char *);
int check_params(char*);
int duplicate(char *s);
int checkarray(char*);
char currfunctype[100];
char currfunc[100];
char currfunccall[100];
void insertSTF(char*);
char gettype(char*,int);
char getfirst(char*);
void push(char *s);
void codegen();
void codeassign();
char* itoa(int num, char* str, int base);
void reverse(char str[], int length);
void swap(char*,char*);
void label1();

```

```

void label2();
void label3();
void label4();
void label5();
void label6();
void genunary();
void codegencon();
void funcgen();
void funcgenend();
void arggen();
void callgen();

int params_count=0;
int call_params_count=0;
int top = 0,count=0,ltop=0,lno=0;
char temp[3] = "t";
%}

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
%token RETURN MAIN
%token VOID
%token WHILE FOR DO
%token BREAK
%token ENDIF
%expect 1

%token identifier array_identifier func_identifier
%token integer_constant string_constant float_constant character_constant

%nonassoc ELSE

%right leftshift_assignment_operator rightshift_assignment_operator
%right XOR_assignment_operator OR_assignment_operator
%right AND_assignment_operator modulo_assignment_operator
%right multiplication_assignment_operator division_assignment_operator
%right addition_assignment_operator subtraction_assignment_operator
%right assignment_operator

%left OR_operator
%left AND_operator
%left pipe_operator

```

```

%left caret_operator
%left amp_operator
%left equality_operator inequality_operator
%left lessthan_assignment_operator lessthan_operator
greaterthan_assignment_operator greaterthan_operator
%left leftshift_operator rightshift_operator
%left add_operator subtract_operator
%left multiplication_operator division_operator modulo_operator

%right SIZEOF
%right tilde_operator exclamation_operator
%left increment_operator decrement_operator


%start program

%%
program
    : declaration_list;

declaration_list
    : declaration D

D
    : declaration_list
    | ;

declaration
    : variable_declaration
    | function_declaration

variable_declaration
    : type_specifier variable_declaration_list ';'

variable_declaration_list
    : variable_declaration_list ','
variable_declaration_identifier | variable_declaration_identifier;

variable_declaration_identifier
    : identifier
{if(duplicate(curid)){printf("Duplicate\n");exit(0);}}insertSTnest(curid,c
urrnest); ins(); } vdi

```

```

        | array_identifier
{if(duplicate(curid)){printf("Duplicate\n");exit(0);}insertSTnest(curid,c
urnnest); ins(); } vdi;

```

```

vdi : identifier_array_type | assignment_operator simple_expression ;

```

```

identifier_array_type
    : '[' initialization_params
    | ;

```

```

initialization_params
    : integer_constant '[' initialization {if($$ < 1)
{printf("Wrong array size\n"); exit(0);} }
    | '[' string_initialization;

```

```

initialization
    : string_initialization
    | array_initialization
    | ;

```

```

type_specifier
    : INT | CHAR | FLOAT | DOUBLE
    | LONG long_grammar
    | SHORT short_grammar
    | UNSIGNED unsigned_grammar
    | SIGNED signed_grammar
    | VOID ;

```

```

unsigned_grammar
    : INT | LONG long_grammar | SHORT short_grammar | ;

```

```

signed_grammar
    : INT | LONG long_grammar | SHORT short_grammar | ;

```

```

long_grammar
    : INT | ;

```

```

short_grammar
    : INT | ;

```

```

function_declaration
    : function_declaration_type
function_declaration_param_statement;

function_declaration_type
    : type_specifier identifier '(' { strcpy(currfunctype,
curtype); strcpy(currfunc, curid); check_duplicate(curid);
insertSTF(curid); ins(); };

function_declaration_param_statement
    : {params_count=0;}params ')' {funcgen();} statement
{funcgenend();};

params
    : parameters_list { insertSTparamscount(currfunc,
params_count); }| { insertSTparamscount(currfunc, params_count); };

parameters_list
    : type_specifier { check_params(curtype);}
parameters_identifier_list ;

parameters_identifier_list
    : param_identifier parameters_identifier_list_breakup;

parameters_identifier_list_breakup
    : ',' parameters_list
    | ;

param_identifier
    : identifier { ins();insertSTnest(curid,1);
params_count++; } param_identifier_breakup;

param_identifier_breakup
    : '[' ']'
    | ;

statement
    : expression_statment | compound_statement
    | conditional_statements | iterative_statements
    | return_statement | break_statement
    | variable_declaration;

```



```

compound_statement
    : {currnest++;} '{' statment_list '}'
{deletedata(currnest);currnest--;} ;

statment_list
    : statement statment_list
    | ;

expression_statment
    : expression ';'
    | ';' ;

conditional_statements
    : IF '(' simple_expression ')'
{label1();if($3!=1){printf("Condition checking is not of type
int\n");exit(0);}} statement {label2();} conditional_statements_breakup;

conditional_statements_breakup
    : ELSE statement {label3();}
    | {label3();};

iterative_statements
    : WHILE '(' {label4();} simple_expression ')'
{label1();if($4!=1){printf("Condition checking is not of type
int\n");exit(0);}} statement {label5();}
    | FOR '(' expression ';' {label4();} simple_expression
    ';' {label1();if($6!=1){printf("Condition checking is not of type
int\n");exit(0);}} expression ')'statement {label5();}
    | {label4();}DO statement WHILE '(' simple_expression
    ')' {label1();label5();if($6!=1){printf("Condition checking is not of type
int\n");exit(0);}} ';' ;

return_statement
    : RETURN ';' {if(strcmp(currfunctype,"void"))
{printf("Returning void of a non-void function\n"); exit(0);}}
    | RETURN expression ';' { if(!strcmp(currfunctype,
"void"))

{

yyerror("Function is void");

}

```

```

if((currfunctype[0]=='i' || currfunctype[0]=='c') && $2!=1)
    {
printf("Expression doesn't match return type of function\n"); exit(0);
    }

};

```

```

break_statement
    : BREAK ';' ;

```

```

string_initilization
    : assignment_operator string_constant {insV();} ;

```

```

array_initialization
    : assignment_operator '{' array_int_declarations '}';

```

```

array_int_declarations
    : integer_constant array_int_declarations_breakup;

```

```

array_int_declarations_breakup
    : ',' array_int_declarations
    | ;

```

```

expression
    : mutable assignment_operator {push("=");} expression
{

```

```

    if($1==1 && $4==1)

```

```

    {

```

```

    $$=1;

```

```

}

```

```

else

```

```

{ $$=-1; printf("Type mismatch\n"); exit(0);}

```

```

codeassign();

```

```

}
```

```

| mutable addition_assignment_operator
{push("+=");}expression {

    if($1==1 && $4==1)

$$=1;

else

{$$=-1; printf("Type mismatch\n"); exit(0);}

codeassign();
}

| mutable subtraction_assignment_operator {push("-=");}
expression {

    if($1==1 && $4==1)

$$=1;

else

{$$=-1; printf("Type mismatch\n"); exit(0);}

codeassign();
}

| mutable multiplication_assignment_operator
{push("*=");} expression {

    if($1==1 && $4==1)

$$=1;

else

{$$=-1; printf("Type mismatch\n"); exit(0);}

codeassign();
}

| mutable division_assignment_operator
{push("/=");}expression {

```

```

        if($1==1 && $4==1)

$$=1;

else

{ $$=-1; printf("Type mismatch\n"); exit(0);}

        | mutable modulo_assignment_operator
{push("%=");}expression
        {

                if($1==1 && $3==1)

                $$=1;

                else

                { $$=-1; printf("Type mismatch\n"); exit(0);}

codeassign();

        }

        | mutable increment_operator
{ push("++");if($1 == 1) $$=1; else $$=-1; genunary();}

        | mutable decrement_operator
{push("--");if($1 == 1) $$=1; else $$=-1;}

        | simple_expression {if($1 == 1) $$=1; else $$=-1;} ;

simple_expression
        : simple_expression OR_operator and_expression
{push("||");} {if($1 == 1 && $3==1) $$=1; else $$=-1; codegen();}

        | and_expression {if($1 == 1) $$=1; else $$=-1;}

and_expression
        : and_expression AND_operator {push("&&");}
unary_relation_expression {if($1 == 1 && $3==1) $$=1; else $$=-1;
codegen();}

        | unary_relation_expression {if($1 == 1) $$=1; else
$$=-1;} ;

```

```

unary_relation_expression
    : exclamation_operator {push("!");}
unary_relation_expression {if($2==1) $$=1; else $$=-1; codegen();}
    | regular_expression {if($1 == 1) $$=1; else $$=-1;} ;

regular_expression
    : regular_expression relational_operators sum_expression
    {if($1 == 1 && $3==1) $$=1; else $$=-1; codegen();}
    | sum_expression {if($1 == 1) $$=1; else $$=-1;} ;

relational_operators
    : greaterthan_assignment_operator {push(">=");} |
    lessthan_assignment_operator {push("<=");} | greaterthan_operator
    {push(">");} | lessthan_operator {push("<");} | equality_operator
    {push("==");} | inequality_operator {push("!=");} ;

sum_expression
    : sum_expression sum_operators term {if($1 == 1 &&
    $3==1) $$=1; else $$=-1; codegen();}
    | term {if($1 == 1) $$=1; else $$=-1;};

sum_operators
    : add_operator {push("+");}
    | subtract_operator {push("-");} ;

term
    : term MULOP factor {if($1 == 1 && $3==1) $$=1; else
    $$=-1; codegen();}
    | factor {if($1 == 1) $$=1; else $$=-1;} ;

MULOP
    : multiplication_operator {push("*");} |
    division_operator {push("/");} | modulo_operator {push("%");} ;

factor
    : immutable {if($1 == 1) $$=1; else $$=-1;}
    | mutable {if($1 == 1) $$=1; else $$=-1;} ;

mutable
    : identifier {
        push(curid);
        if(check_id_is_func(curid))

```

```

                                {printf("Function name used as
Identifier\n"); exit(8);}
                                if(!checkscope(curid))

{printf("%s\n",curid);printf("Undeclared\n");exit(0);}
                                if(!checkarray(curid))
                                {printf("%s\n",curid);printf("Array ID has
no subscript\n");exit(0);}
                                if(gettype(curid,0)=='i' ||
gettype(curid,1)=='c')
                                $$ = 1;
                                else
                                $$ = -1;
                                }
                                | array_identifier
{if(!checkscope(curid)){printf("%s\n",curid);printf("Undeclared\n");exit(
0);}} '[' expression ']'
                                {if(gettype(curid,0)=='i' ||
gettype(curid,1)=='c')
                                $$ = 1;
                                else
                                $$ = -1;
                                };

immutable
                                : '(' expression ')' {if($2==1) $$=1; else $$=-1;}
                                | call {if($1==-1) $$=-1; else $$=1;}
                                | constant {if($1==1) $$=1; else $$=-1;};

call
                                : identifier '('{

                                if(!check_declaration(curid, "Function"))
                                { printf("Function not declared");
exit(0);}

                                insertSTF(curid);
                                strcpy(currfunccall,curid);
                                if(gettype(curid,0)=='i' ||
gettype(curid,1)=='c')
                                {
                                $$ = 1;
                                }

```

```

        else
            $$ = -1;
        call_params_count=0;
    }
    arguments ')'
        { if(strcmp(currfunccall,"printf"))
            {

if(getSTparamscount(currfunccall)!=call_params_count)
            {
                yyerror("Number of
arguments in function call doesn't match number of parameters");
                exit(8);
            }
        }
        callgen();
    };

arguments
    : arguments_list | ;

arguments_list
    : arguments_list ',' exp { call_params_count++; }
    | exp { call_params_count++; };

exp : identifier {arggen(1);} | integer_constant {arggen(2);} |
string_constant {arggen(3);} | float_constant {arggen(4);} |
character_constant {arggen(5);} ;

constant
    : integer_constant { insV(); codegencon(); $$=1; }
    | string_constant { insV(); codegencon(); $$=-1;}
    | float_constant{ insV(); codegencon();}
    | character_constant{ insV(); codegencon(); $$=1; };

%%

extern FILE *yyin;
extern int yylineno;
extern char *yytext;
void insertSTtype(char *,char *);
void insertSTvalue(char *, char *);

```

```

void incertCT(char *, char *);
void printST();
void printCT();

struct stack
{
    char value[100];
    int labelvalue;
}s[100],label[100];

void push(char *x)
{
    strcpy(s[++top].value,x);
}

void swap(char *x, char *y)
{
    char temp = *x;
    *x = *y;
    *y = temp;
}

void reverse(char str[], int length)
{
    int start = 0;
    int end = length -1;
    while (start < end)
    {
        swap((str+start), (str+end));
        start++;
        end--;
    }
}

char* itoa(int num, char* str, int base)
{
    int i = 0;
    int isNegative = 0;

    if (num == 0)

```



```

{
    str[i++] = '0';
    str[i] = '\\0';
    return str;
}

if (num < 0 && base == 10)
{
    isNegative = 1;
    num = -num;
}

while (num != 0)
{
    int rem = num % base;
    str[i++] = (rem > 9)? (rem-10) + 'a' : rem + '0';
    num = num/base;
}

if (isNegative)
    str[i++] = '-';

str[i] = '\\0';

reverse(str, i);

return str;
}

void codegen()
{
    strcpy(temp, "t");
    char buffer[100];
    itoa(count, buffer, 10);
    strcat(temp, buffer);
    printf("%s = %s %s\n", temp, s[top-2].value, s[top-1].value, s[top].value);
    top = top - 2;
    strcpy(s[top].value, temp);
    count++;
}

```

```

}

void codegencon()
{
    strcpy(temp, "t");
    char buffer[100];
    itoa(count, buffer, 10);
    strcat(temp, buffer);
    printf("%s = %s\n", temp, curval);
    push(temp);
    count++;
}

int isunary(char *s)
{
    if(strcmp(s, "--")==0 || strcmp(s, "++")==0)
    {
        return 1;
    }
    return 0;
}

void genunary()
{
    char temp1[100], temp2[100], temp3[100];
    strcpy(temp1, s[top].value);
    strcpy(temp2, s[top-1].value);

    if(isunary(temp1))
    {
        strcpy(temp3, temp1);
        strcpy(temp1, temp2);
        strcpy(temp2, temp3);
    }
    strcpy(temp, "t");
    char buffer[100];
    itoa(count, buffer, 10);
    strcat(temp, buffer);
    count++;

    if(strcmp(temp2, "--")==0)

```

```

    {
        printf("%s = %s - 1\n", temp, temp1);
        printf("%s = %s\n", temp1, temp);
    }

    if(strcmp(temp2,"++")==0)
    {
        printf("%s = %s + 1\n", temp, temp1);
        printf("%s = %s\n", temp1, temp);
    }

    top = top -2;
}

void codeassign()
{
    printf("%s = %s\n",s[top-2].value,s[top].value);
    top = top - 2;
}

void label1()
{
    strcpy(temp,"L");
    char buffer[100];
    itoa(lno,buffer,10);
    strcat(temp,buffer);
    printf("IF not %s GoTo %s\n",s[top].value,temp);
    label[++ltop].labelvalue = lno++;
}

void label2()
{
    strcpy(temp,"L");
    char buffer[100];
    itoa(lno,buffer,10);
    strcat(temp,buffer);
    printf("GoTo %s\n",temp);
    strcpy(temp,"L");
    itoa(label[ltop].labelvalue,buffer,10);
    strcat(temp,buffer);
    printf("%s:\n",temp);
    ltop--;
}

```

```

        label[++ltop].labelvalue=lno++;
    }

void label3()
{
    strcpy(temp,"L");
    char buffer[100];
    itoa(label[ltop].labelvalue,buffer,10);
    strcat(temp,buffer);
    printf("%s:\n",temp);
    ltop--;
}

void label4()
{
    strcpy(temp,"L");
    char buffer[100];
    itoa(lno,buffer,10);
    strcat(temp,buffer);
    printf("%s:\n",temp);
    label[++ltop].labelvalue = lno++;
}

void label5()
{
    strcpy(temp,"L");
    char buffer[100];
    itoa(label[ltop-1].labelvalue,buffer,10);
    strcat(temp,buffer);
    printf("GoTo %s:\n",temp);
    strcpy(temp,"L");
    itoa(label[ltop].labelvalue,buffer,10);
    strcat(temp,buffer);
    printf("%s:\n",temp);
    ltop = ltop - 2;
}

void funcgen()

```

```

{
    printf("func begin %s\n", currfunc);
}

void funcgenend()
{
    printf("func end\n\n");
}

void arggen(int i)
{
    if(i==1)
    {
        printf("refparam %s\n", curid);
    }
    else
    {
        printf("refparam %s\n", curval);
    }
}

void callgen()
{
    printf("refparam result\n");
    push("result");
    printf("call %s, %d\n", currfuncall, call_params_count);
}

int main(int argc , char **argv)
{
    yyin = fopen(argv[1], "r");
    yyparse();

    if(flag == 0)
    {
        printf(ANSI_COLOR_GREEN "Status: Parsing Complete - Valid"
ANSI_COLOR_RESET "\n");
        printf("%30s" ANSI_COLOR_CYAN "SYMBOL TABLE" ANSI_COLOR_RESET
"\n", " ");
        printf("%30s %s\n", " ", "-----");
    }
}

```

```

        printST();

        printf("\n\n%30s" ANSI_COLOR_CYAN "CONSTANT TABLE"
ANSI_COLOR_RESET "\n", " ");
        printf("%30s %s\n", " ", "-----");
        printCT();
    }
}

void yyerror(char *s)
{
    printf(ANSI_COLOR_RED "%d %s %s\n", yylineno, s, yytext);
    flag=1;
    printf(ANSI_COLOR_RED "Status: Parsing Failed - Invalid\n"
ANSI_COLOR_RESET);
    exit(7);
}

void ins()
{
    insertSTtype(curid, curtype);
}

void insV()
{
    insertSTvalue(curid, curval);
}

int yywrap()
{
    return 1;
}

```

## **Explanation:**

The lex code is detecting the tokens from the source code and returning the corresponding token to the parser. In phase 1 we were just printing the token and now we are returning the token so that parser uses it for further computation. We are using the symbol table and the constant table of the previous phase only. We added functions like insertSTnest(), insertST(), paramscount(), checkscope(), deletedata(), duplicate() etc., in order to check the semantics. In the production rules of the grammar semantic actions are written and these are performed by the functions listed above. Along with semantic actions SDT also included function to generate the 3 address code.

## **Declaration Section**

In this section we have included all the necessary header files, function declaration and flag that was needed in the code.

Between declaration and rules section we have listed all the tokens which are returned by the lexer according to the precedence order. We also declared the operators here according to their associativity and precedence. This ensures the grammar we are giving to the parser is unambiguous as LALR(1) parser cannot work with ambiguous grammar.

## **Rules Section**

In this section production rules for entire C language is written. The grammar productions does the syntax analysis of the source code. When a complete statement with proper syntax is matched by the parser. Along with rules semantic actions associated with the rules are also written and corresponding functions are called to do the necessary actions. Then code generation function was also associated with the production so that we can get the desired 3 address code for the given input program.

## **C-Program Section**

In this section the parser links the extern functions, variables declared in the lexer, external files generated by the lexer etc. The main function takes the input source code file and prints the final symbol table and the 3 address code. Apart from these several functions are also written that generates the 3 address code.

## Test Cases:

Without Errors:

Test 1: Error Free Code

(a simple c program to validate the compiler)

```
#include <stdio.h>
void main()
{
    int a = 10, i = 0;
    do
    {
        printf("hello world");
        i = i + 1;
    }while(i<a);
}
```

Output:

**STATUS: PASS**

```
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$ ./compiler ef-test1.c
```

```
func begin main
t0 = 10
t1 = 0
L0:
refparam "hello world"
refparam result
call printf, 1
t2 = 1
t3 = i + t2
i = t3
t4 = i < a
IF not t4 GoTo L1
GoTo L0:
L1:
func end
```

Status: Parsing Complete - Valid

### SYMBOL TABLE

SYMBOL	CLASS	TYPE	VALUE	LINE NO	NESTING	PARAMS COUNT
a	Identifier	int	10	5	99999	-1
i	Identifier	int	0	5	99999	-1
do	Keyword			7	9999	-1
int	Keyword			5	9999	-1
main	Function	void		3	9999	0
printf	Function			9	9999	-1
while	Keyword			11	9999	-1
void	Keyword			3	9999	-1

### CONSTANT TABLE

NAME	TYPE
"hello world"	String Constant
10	Number Constant
0	Number Constant
1	Number Constant

```
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$
```

Figure 1.



Test 2: Error free code  
(For loop, function, assignment)

```
#include<stdio.h>

int add(int a , int b){
    return a+b;
}

int main()
{
    int a = 5, b =10, i = 0;
    a = a+b;
    for(i=0;i<b;i++)
    {
        b++;
        add(a,b);

    }
    a++;

    while(a>0)
    {
        printf("Hello world");
        a--;
    }

    a=4;
    while(a>0)
    {
        printf("%d",a);
        a--;
        int b;
        b= 4;
        while(b>0)
        {
            printf("%d", b);
            b--;
        }
    }
}
```

```
}
```

Output:  
**STATUS: Pass**

```
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$ ./compiler ef-test2.c
func begin add
t0 = a + b
func end

func begin main
t1 = 5
t2 = 10
t3 = 0
t4 = a + b
a = t4
t5 = 0
i = t5
L0:
t6 = i < b
IF not t6 GoTo L1
t7 = i + 1
i = t7
t8 = b + 1
b = t8
refparam a
refparam b
refparam result
call add, 2
GoTo L0:
L1:
t9 = a + 1
a = t9
L2:
t10 = 0
t11 = a > t10
IF not t11 GoTo L3
refparam "Hello world"
refparam result
call printf, 1
GoTo L2:
L3:
t12 = 4
```

```
L3:
t12 = 4
a = t12
L4:
t13 = 0
t14 = a > t13
IF not t14 GoTo L5
refparam "%d"
refparam a
refparam result
call printf, 2
t15 = 4
b = t15
L6:
t16 = 0
t17 = b > t16
IF not t17 GoTo L7
refparam "%d"
refparam b
refparam result
call printf, 2
GoTo L6:
L7:
GoTo L4:
L5:
func end
```

Status: Parsing Complete - Valid

SYMBOL TABLE

SYMBOL	CLASS	TYPE	VALUE	LINE NO	NESTING	PARAMS COUNT
a	Identifier	int		3	99999	-1
b	Identifier	int		3	99999	-1
a	Identifier	int	0	9	99999	-1
b	Identifier	int	10	9	99999	-1
b	Identifier	int	0	32	99999	-1
i	Identifier	int	0	9	99999	-1

```

call printf, 2
GoTo L6:
L7:
GoTo L4:
L5:
func end

Status: Parsing Complete - Valid

```

SYMBOL	CLASS	TYPE	VALUE	LINE NO	NESTING	PARAMS COUNT
a	Identifier	int		3	99999	-1
b	Identifier	int		3	99999	-1
a	Identifier	int	0	9	99999	-1
b	Identifier	int	10	9	99999	-1
b	Identifier	int	0	32	99999	-1
i	Identifier	int	0	9	99999	-1
for	Keyword			11	9999	-1
return	Keyword			4	9999	-1
int	Keyword			3	9999	-1
add	Function	int		3	9999	2
main	Function	int		7	9999	0
printf	Function			23	9999	-1
while	Keyword			21	9999	-1

  

NAME	TYPE
"Hello world"	String Constant
"%d"	String Constant
10	Number Constant
0	Number Constant
4	Number Constant
5	Number Constant

```

chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$

```

Figure 2.

### Test 3: Error Free Code

(multiplication, addition, subtraction, assignment operators)

```

#include<stdio.h>

int main()
{
    int a,b,c,d;

    a = d * b + c;
    a = (b-c)*d + (b-c)*d;
    a = b+c*d;
    a = b-c + b*c;
}

```

Output:

STATUS: Pass

```
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$ ./compiler ef-test3.c
func begin main
t0 = d * b
t1 = t0 + c
a = t1
t2 = b - c
t3 = t2 * d
t4 = b - c
t5 = t4 * d
t6 = t3 + t5
a = t6
t7 = c * d
t8 = b + t7
a = t8
t9 = b - c
t10 = b * c
t11 = t9 + t10
a = t11
func end

Status: Parsing Complete - Valid

          SYMBOL TABLE
          -----
SYMBOL | CLASS | TYPE | VALUE | LINE NO | NESTING | PARAMS COUNT |
-----|-----|-----|-----|-----|-----|-----|
a | Identifier | int |  | 5 | 99999 | -1 |
b | Identifier | int |  | 5 | 99999 | -1 |
c | Identifier | int |  | 5 | 99999 | -1 |
d | Identifier | int |  | 5 | 99999 | -1 |
int | Keyword |  |  | 3 | 9999 | -1 |
main | Function | int |  | 3 | 9999 | 0 |

          CONSTANT TABLE
          -----
NAME | TYPE
-----|-----
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$
```

Figure 3.

Test4: Error free code:  
(conditional statements)

```
#include <stdio.h>
void main()
{int a,b,c,d;
  while(a < c)
  {
    if( a < b){
      if (b < c){
        a = b+c;
      }
      else{
        b = 10;
      }
    }
  }
}}
```

Output:

**STATUS:** Pass

```
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$ ./compiler ef-test4.c
func begin main
L0:
t0 = a < c
IF not t0 GoTo L1
t1 = a < b
IF not t1 GoTo L2
t2 = b < c
IF not t2 GoTo L3
t3 = b + c
a = t3
GoTo L4
L3:
t4 = 10
b = t4
L4:
GoTo L5
L2:
L5:
GoTo L0:
L1:
func end

Status: Parsing Complete - Valid
```

SYMBOL TABLE						
SYMBOL	CLASS	TYPE	VALUE	LINE NO	NESTING	PARAMS COUNT
a	Identifier	int		5	99999	-1
b	Identifier	int		5	99999	-1
c	Identifier	int		5	99999	-1
d	Identifier	int		5	99999	-1
if	Keyword			8	9999	-1
int	Keyword			5	9999	-1
main	Function	void		3	9999	0
else	Keyword			12	9999	-1
while	Keyword			6	9999	-1
void	Keyword			3	9999	-1

CONSTANT TABLE	
NAME	TYPE
10	Number Constant

```
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$
```

Figure 4.

## With Errors:

Test1: Code with Error  
(Illegal Identifier)

```
// # cannot be the starting character of a variable

#include<stdio.h>

int main() {
    char #c;
    #c = 10

    int a = #c
    while(a<3)
    {
        a = a+b;
        for(i=0;i<b;i++)
        {
            b++;
            myfunc(a,b);

        }
        a++;
    }
}
```

Output:

STATUS: Pass

```
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$ ./compiler e-test1.c
func begin main
Error in Pre-Processor directive at line no. 6
#
6 syntax error #
Status: Parsing Failed - Invalid
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$
```

Figure 5.

Test2: Code with Error

(Identifier not declared, here function not defined or declared)

```
#include<stdio.h>
int cube(int a)
{
    return a*a*a;
}

int main()
{
    int num = 2;
    if(num < 0){
        num = cube(num)
    }
    else{
        num = square(num)
    }
}
```

Output:

**STATUS:** Pass

```
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$ ./compiler e-test2.c
func begin cube
t0 = a * a
t1 = t0 * a
func end

func begin main
t2 = 2
t3 = 0
t4 = num < t3
IF not t4 GoTo L0
refparam num
refparam result
call cube, 1
num = result
12 syntax error }
$status: Parsing Failed - Invalid
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$
```

Figure 6.

Test3: Code with error:  
(Improper number of braces)

```
#include <stdio.h>
int main() {
    int year;
    printf("Enter a year: ");
    scanf("%d", &year);

    if (year % 400 == 0) {
        printf("%d is a leap year.", year);
    }
    else if (year % 100 == 0) {
        printf("%d is not a leap year.", year);
    }
    else if (year % 4 == 0) {
        printf("%d is a leap year.", year);
    }
    else
        printf("%d is not a leap year.", year);
}

return 0;
}
```

Output:

STATUS: Pass

```
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$ ./compiler e-test3.c
func begin main
refparam "Enter a year: "
refparam result
call printf, 1
refparam "%d"
$ syntax error &
status: Parsing Failed - Invalid
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$ █
```

Figure 7.



Test4: Code with error  
(Error in return type)

```
#include<stdio.h>

int facto(int n){
    int i, fact=1;
    for(i=1;i<=n;i++){
        fact=fact*i;
    }
    return fact;
}

int main()
{
    int i,number;
    printf("Enter a number: ");
    scanf("%d",&number);
    char answer[10] =""
    answer = facto(number)
    printf("Factorial of %d is: %d",number,answer);
    return 0;
}
```

Output:

STATUS: Pass

```
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$ ./compiler e-test4.c
func begin facto
t0 = 1
t1 = 1
i = t1
L0:
t2 = i <= n
IF not t2 GoTo L1
t3 = i + 1
i = t3
t4 = fact * i
fact = t4
GoTo L0:
L1:
func end

func begin main
refparam "Enter a number: "
refparam result
call printf, 1
refparam "%d"
14 syntax error &
Status: Parsing Failed - Invalid
chandan@chandan-HP-Pavilion-Notebook:~/mini-c-compiler/ICG$
```

Figure 8.

# Implementation:

The lexer code submitted in the previous phase took care of most of the features of C using regular expressions. Some special corner cases were taken care of using custom regex.

These were:

- A. The Regex for Identifiers
- B. Multiline comments should be supported
- C. Literals
- D. Error Handling for Incomplete String
- E. Error Handling for Nested Comment

The parser code requires exhaustive token recognition and because of this reason, we utilised the lexer code given under the C specifications with the parser. The parser implements C grammar using a number of production rules.

The parser takes tokens from the lexer output, one at a time and applies the corresponding production rules to append to the symbol table with type, value and line of declaration. If the parsing is not successful, the parser outputs the line number with the corresponding error. Along with this semantic actions were also added to each production rule to check if the structure created has some meaning or not. Then we added the function to generate the 3 address code with production so that we can generate the desired intermediate code. In order to generate 3 address code we made use of explicit stack. Whenever we came across an operator, operand or constant we pushed it to stack. Whenever reduction occurred (Since LALR(1) parser is bottom up parser it evaluates SDT when reduction occurs) `codegen()` function generated the 3 address code by creating a new temporary variable and by making use of the entries in the stack, after that it popped those entries from the stack and pushed the temporary variable to the stack so that it gets used in further computation. Similarly functions like `labels` were used to assign appropriate labels while using conditional statements or iterative statements. All the functions used are described below :

1. `codegen()` : This function is called whenever a reduction of an expression takes place. It creates the temporary variable and displays the desired 3 address code i.e  $x = y \text{ op } z$ .
2. `codegencon()` : This function is especially written for reductions of expression involving constants since its 3 address code is  $x \text{ op } z$ .
3. `isunary()` : This function checks if the operator is an unary operator like '++'. If so it returns true else false.
4. `genunary()` : This function is specifically designed to generate 3 address code for unary operations. It makes use of `isunary` function mentioned above. E.g. if  $a = i++$  then it converts into  $t0 = i + 1, a = t0$ .
5. `codeassign()` : This function is specifically designed for assignment operator. It assigns the final temp variable value (after all the evaluation) to the desired variable.
6. `label1()` : It is used while evaluating conditions of loops or if statement. If the condition is not satisfied then it states where to jump to i.e. on which label the control should go.
7. `label2()` : It is used when the statement block pertaining to if statement is over. It tells where

the control flow should go once that block is over i.e. it jumps the else statement block.

8. `label3()` : it is used after the whole if else construct is over. It gives label that tells where to jump after the if block is executed.
9. `label4()` : it is used to give labels to starting of loops.
10. `label5()` : it is used after the statement block of the loop. It indicates the label to jump to and also generates the label where the control should go once the loop is terminated.
11. `funcgen()` : it indicates the beginning of a function.
12. `funcgenend()` : it indicates the ending of a function.
13. `arggen()` : it displays all the reference parameters that are used in a function call.
14. `callgen()` : it calls the function i.e. displays the appropriate function call according to 3 address code.
15. `Itoa()` : it worked as utility function since we had to name temporary variables and labels it was used to convert int to string and used several functions like reverse , swap to do it.

## Results:

The code can parse the tokens produced by the flex script. The output displays the set of identifiers and constants present in the program with their types, values and line of declaration. Also nesting values changes dynamically as the program ends its made infinite. The parser generates error messages in case of any syntactical errors in the test program or any semantic error. Also we are displaying the 3 address code generated by our yacc script.

## Future work:

The yacc script can now handle the test cases and most of the rules in the C language. Although this is true, there are still some rules that cannot be parsed.

The future work would be to handle all test cases of the C language. Also more test cases can be written so as to increase the validation criteria and make the script robust in nature. The script can also be made more modular so that the compiler can be used out of the box without too many configurations.

## References:

1. <https://www.geeksforgeeks.org/compiler-design-tutorials/>
2. <http://dinosaur.compilertools.net/>
3. Compilers: Principles, Techniques, and Tools: Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman