CS304- COMPILER DESIGN LAB

A REPORT ON THE PROJECT ENTITLED

INTERMEDIATE CODE GENERATOR FOR THE C LANGUAGE

Group Members:

B Anagha	Gouri M R	P Devi Deepika
221CS114	221CS125	221CS138

V SEMESTER B-TECH CSE- S1

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL

2024 – 2025

ABSTRACT

A compiler is a software tool that translates code written in a high-level programming language, like C, into machine code or an intermediate form that a computer's hardware can understand and execute. This process is essential because high-level languages are designed to be human-readable, with syntax and abstractions that make coding more intuitive, whereas machines operate on binary instructions. The primary role of a compiler is to bridge this gap, converting complex instructions into efficient, executable code while ensuring that the program's intended logic and functionality remain intact.

This document presents the implementation of the intermediate code generation phase (Phase 4) in C, an essential step in compiler design. Intermediate code generation acts as a bridge between high-level language syntax and machine-level instructions, providing an abstracted, portable representation that simplifies optimization and target code generation. By implementing this phase, we achieve language and machine independence while enhancing the compiler's ability to perform optimizations. This phase converts syntax tree representations or parse tree outputs from previous compiler phases into an intermediate form, often using three-address code (TAC). Our focus is on outlining the data structures, algorithms, and code structure required to create intermediate code, including the handling of arithmetic expressions, conditional statements, and iterative constructs. This document aims to facilitate an understanding of the intermediate code generation process, highlighting the key challenges and design choices encountered in the implementation.

INTRODUCTION

Intermediate code generation is a core phase in the compiler's translation process, transforming a high-level program's source code into an intermediate representation (IR) that sits between syntax analysis and machine code generation. The IR serves as an abstract layer, simplifying the transition to machine-specific code and providing a platform for optimizations. Typical IR forms include three-address code (TAC), quadruples, and abstract syntax trees. This document focuses on implementing the TAC-based IR in C, aiming to make the IR compact, expressive, and conducive to both subsequent optimization and code generation phases.

Phase 4 of our compiler project leverages the output from syntax and semantic analysis to produce TAC. TAC is chosen due to its balance between simplicity and the expressive power needed to capture control flow and data dependencies efficiently. Through this phase, we translate high-level language constructs into three-address statements, handling arithmetic expressions, conditional and looping structures, and function calls. The implementation strategy emphasizes modularity, using data structures like linked lists and hash tables to manage symbol information, temporary variables, and labels, which are crucial for ensuring correct IR generation.

This document outlines the approach, design considerations, data structures, and algorithms used to construct the IR. The goal is to provide a practical reference for understanding intermediate code generation, ultimately serving as a foundation for further compiler development, such as optimization and final code generation.

Here are the key activities carried out in the ICG phase:

1. Convert High-Level Constructs to Intermediate Representation

- Translate complex high-level language constructs (such as arithmetic expressions, control statements, and loops) into a standardized intermediate form, often using **Three-Address Code (TAC)**. TAC simplifies these constructs into a series of instructions that involve at most three operands, such as `a = b + c`.
- This conversion makes it easier to optimize and later translate the code into machine-level instructions.

2. Generate Temporary Variables

- Introduce **temporary variables** to hold intermediate values of complex expressions. This is especially important in expressions that require multiple operations, where each result needs to be stored temporarily before further computation.
- Example: For an expression like `a + b * c`, a temporary variable might store `b * c` before adding `a`.

3. Handle Control Flow Statements

- Translate conditional (`if`, `else`, `switch`) and looping (`for`, `while`) constructs into IR, typically using **labels and jump instructions** to represent control flow.
- For example, `if (x > y)` might be represented as a series of comparison and jump instructions.

4. Represent Function Calls and Parameter Passing

- Standardize the intermediate representation for **function calls, argument passing, and return values** to ensure consistency across different high-level function implementations.
- This includes handling the calling conventions, such as setting up parameters, managing return addresses, and defining the calling sequence in TAC.

5. Maintain a Symbol Table

- Use a **symbol table** to track variables, constants, function names, and other identifiers along with their types, scope, and memory locations.
- This ensures that the generated IR correctly references variables and provides necessary information for code generation and optimization phases.

6. Labeling for Code Blocks

- Assign unique **labels** to different blocks of code, especially for conditional branches, loops, and function calls. Labels help manage control flow within the IR and aid in later phases, like optimization and code generation.

7. Simplify Complex Expressions

- Break down complex expressions and statements into simpler, manageable instructions in TAC. Each TAC instruction represents a single operation, making it easier for the compiler to process and optimize them.

8. Platform Independence

- Generate IR that is independent of machine-specific instructions. This ensures that the compiler can later target different hardware architectures by translating this IR into machine code during the code generation phase.

Overall, ICG provides a machine-independent code structure that simplifies the process of generating optimized machine code and supports platform versatility.

IMPLEMENTATION IN C COMPILER

Here's a high-level outline of the steps involved in implementing the **Intermediate Code Generation (ICG)** in a C compiler:

Step 1: Initialize Symbol Table and Temporary Management

- Set up a **symbol table** to store information about variables, constants, and functions.
- Create a mechanism for generating and managing **temporary variables** (often named t1, t2, etc.) for storing intermediate results.

Step 2: Implement TAC Data Structure

- Define a data structure for **Three-Address Code (TAC)**, where each instruction typically includes an operation, two operands, and a result location.
- This structure will help in representing expressions, control flow, and function calls uniformly.

Step 3: Translate Expressions to TAC

- For each expression (like arithmetic and logical operations), generate a sequence of TAC instructions.

- Use temporary variables to store intermediate results of sub-expressions, breaking complex expressions into simple, manageable steps.

Step 4: Handle Control Flow Statements

- Implement TAC generation for **conditional and looping statements** by using **conditional jumps** and labels.
- For example, `if (condition) { ... } else { ... }` would translate into conditional jump instructions and labels for branching.

Step 5: Implement Function Call and Parameter Handling

- Define TAC for **function calls**, **parameter passing**, and **return statements**.
- Use a standardized format to handle function prologues, parameter setup, and returns, ensuring consistency across function calls in TAC.

Step 6: Manage Scope and Symbol Resolution

- Ensure that variable scopes are correctly managed to avoid conflicts, especially in cases of nested scopes or recursive functions.
- Use the symbol table to look up and assign values to identifiers, ensuring that variables are consistently referenced.

Step 7: Generate Labels for Jumps

- For control statements and loops, generate **unique labels** to manage control flow transitions accurately.
- Assign labels to start, end, and any break/continue points to facilitate easier back-patching.

Step 8: Output Intermediate Code

- Store the generated TAC instructions in a list or array.
- Implement a function to output or print the TAC, which is useful for debugging and further processing in later compiler phases.

Step 9: Handle Back-Patching for Unresolved Jumps

- Manage **back-patching** for unresolved labels, which can occur in branching and loops.
- This involves updating placeholder labels once the target locations are known.

By following these steps, the compiler can produce an intermediate code representation that is both machine-independent and suitable for optimization and final code generation phases.

CODE

LEXICAL ANALYSER CODE

```
%option yylineno

%{
#include <stdio.h>
#include <string.h>
#include<stdlib.h>
#include "pa4.tab.h"

struct ConstantTable{
char constant_name[100] ;
char constant_type[100] ;
int exist ;
}CT[1000] ;

struct SymbolTable{
char symbol_name[100] ;
char symbol_type[100] ;
char array_dimensions[100] ;
char class[100] ;
char value[100] ;
char parameters[100] ;
int line_number ;
int exist ;
int define ;
int nested_val ;
int curr_nest_val ;
int params_count ;
}ST[1000] ;

int current_nested_val = 0 ;
```

```
int params_count = 0 ;

unsigned long hash(unsigned char *str)
{
unsigned long hash = 5381 ;
int c ;

while (c = *str++)
hash = ((hash << 5) + hash) + c ;

return hash ;
}

int search_ConstantTable(char* str){
unsigned long temp_val = hash(str) ;
int val = temp_val%1000 ;

if(CT[val].exist == 0){
return 0 ;
}

else if(strcmp(CT[val].constant_name, str) == 0)
{
return 1 ;
}
else
{
for(int i = val+1                    ; i!=val ; i = (i+1)%1000)
{
if(strcmp(CT[i].constant_name,str)==0)
{
return 1 ;
}
}
return 0 ;
}
}

int search_SymbolTable(char* str){
unsigned long temp_val = hash(str) ;
int val = temp_val%1000 ;
```

```
if(ST[val].exist == 0){
return 0 ;
}

else if(strcmp(ST[val].symbol_name, str) == 0)
{
return val ;
}
else
{
for(int i = val+1                    ; i!=val ; i = (i+1)%1000)
{
if(strcmp(ST[i].symbol_name,str)==0)
{
return i ;
}
}
return 0 ;
}
}

void insert_ConstantTable(char* name, char* type){
int index = 0 ;
if(search_ConstantTable(name)){
return ;
}
else{
unsigned long temp_val = hash(name) ;
int val = temp_val%1000 ;
if(CT[val].exist == 0){
strcpy(CT[val].constant_name, name) ;
strcpy(CT[val].constant_type, type) ;
CT[val].exist = 1 ;
return ;
}

for(int i = val+1 ; i != val ; i = (i+1)%1000){
if(CT[i].exist == 0){
index = i ;
break ;
```

```
}
}
strcpy(CT[index].constant_name, name) ;
strcpy(CT[index].constant_type, type) ;
CT[index].exist = 1 ;
}
}

void insert_SymbolTable(char* name, char* class){
int index = 0 ;
if(search_SymbolTable(name)){
return ;
}
else{
unsigned long temp_val = hash(name) ;
int val = temp_val%1000 ;
if(ST[val].exist == 0){
strcpy(ST[val].symbol_name, name) ;
strcpy(ST[val].class, class) ;
ST[val].nested_val = 100 ;
ST[val].curr_nest_val = current_nested_val ;
//ST[val].params_count = -1 ;
ST[val].line_number = yylineno ;
ST[val].exist = 1 ;
strcpy(ST[val].symbol_type,"");
return ;
}

for(int i = val+1 ; i != val ; i = (i+1)%1000){
if(ST[i].exist == 0){
index = i ;
break ;
}
}

strcpy(ST[index].symbol_name, name) ;
strcpy(ST[val].class, class) ;
ST[index].nested_val = 100 ;
ST[index].curr_nest_val = current_nested_val ;
//ST[index].params_count = -1 ;
ST[index].exist = 1 ;
```

```
if(strcmp("Function", class) == 0)ST[index].define = 1 ;
}
}

void insert_SymbolTable_type(char *str1, char *str2)
{

for(int i = 0                    ; i < 1000 ; i++)
{
if(strcmp(ST[i].symbol_name,str1)==0)
{
if(strcmp(ST[i].symbol_type,"") == 0)
strcpy(ST[i].symbol_type,str2) ;
}
}
}

void insert_SymbolTable_value(char *str1, char *str2)
{
for(int i = 0                                                    ; i < 1001 ; i++)
{
if(strcmp(ST[i].symbol_name,str1)==0 && ST[i].nested_val != current_nested_val)
{
strcpy(ST[i].value,str2) ;
}
}
}

void insert_SymbolTable_arraydim(char *str1, char *dim)
{
for(int i = 0                          ; i < 1000 ; i++)
{
if(strcmp(ST[i].symbol_name,str1)==0)
{
strcpy(ST[i].array_dimensions,dim) ;
}
}
}

void insert_SymbolTable_funcparam(char *str1, char *param)
{
```

```
for(int i = 0                                    ; i < 1000 ; i++)
{
if(strcmp(ST[i].symbol_name,str1)==0)
{
strcat(ST[i].parameters," ");
strcat(ST[i].parameters,param) ;
}
}
}

void insert_SymbolTable_line(char *str1, int line)
{
for(int i = 0                                ; i < 1000 ; i++)
{
if(strcmp(ST[i].symbol_name,str1)==0)
{
ST[i].line_number = yylineno ;
}
}
}

void insert_SymbolTable_nest(char *s, int nest)
{
if(search_SymbolTable(s) && ST[search_SymbolTable(s)].nested_val != 100)
{
int pos = 0 ;
int value = hash(s) ;
value = value%1001 ;
for (int i = value + 1                                    ; i!=value ; i = (i+1)%1001)
{
if(ST[i].exist == 0)
{
pos = i ;
break ;
}
}

strcpy(ST[pos].symbol_name,s) ;
strcpy(ST[pos].class,"Identifier");
ST[pos].nested_val = nest ;
ST[pos].curr_nest_val = current_nested_val ;
```

```
ST[pos].line_number = yylineno ;
ST[pos].exist = 1 ;
}
else
{
for(int i = 0                                   ; i < 1001 ; i++)
{
if(strcmp(ST[i].symbol_name,s)==0 )
{
ST[i].nested_val = nest ;
ST[i].curr_nest_val = current_nested_val ;
}
}
}
}

int check_scope(char *s)
{
int flag = 0 ;
for(int i = 0                           ; i < 1000 ; i++)
{
if(strcmp(ST[i].symbol_name,s)==0)
{
if(ST[i].nested_val > current_nested_val)
{
flag = 1 ;
}
else
{
flag = 0 ;
break ;
}
}
}
if(!flag)
{
return 1 ;
}
else
{
return 0 ;
```

```
}
}

void remove_scope (int nesting)
{
for(int i = 0                    ; i < 1000 ; i++)
{
if(ST[i].nested_val == nesting)
{
ST[i].nested_val = 100 ;
ST[i].curr_nest_val = current_nested_val ;
}
}
}

void insert_SymbolTable_function(char *s)
{
for(int i = 0                    ; i < 1001 ; i++)
{
if(strcmp(ST[i].symbol_name,s)==0 )
{
strcpy(ST[i].class,"Function");
ST[i].define=0 ;
return ;
}
}

}

void updateDefine(char *s)
{

for(int i = 0                    ; i < 1001 ; i++)
{
if(strcmp(ST[i].symbol_name,s)==0 )
{

ST[i].define=1 ;
return ;
}
}
```

```
}

void insert_SymbolTable_function2(char *s)
{
for(int i = 0                    ; i < 1001 ; i++)
{
if(strcmp(ST[i].symbol_name,s)==0 )
{
strcpy(ST[i].class,"Function");
ST[i].define=1 ;
return ;
}
}

}
int isNotPointer(char *s)
{

for(int i = 0                    ; i < 1001 ; i++)
{
if(strcmp(ST[i].symbol_name,s)==0 )
{
int j=0 ;
for(;ST[i].symbol_type[j+1]!='\0';j++);

char ch=ST[i].symbol_type[j] ;
if(ch=='*') return 0 ;

return 1 ;
}
}

}

int check_function(char *s)
{
for(int i = 0                    ; i < 1000 ; i++)
{
if(strcmp(ST[i].symbol_name,s)==0)
{
```

```
if(strcmp(ST[i].class,"Function")==0)
return 1 ;
}
}
return 0 ;
}

int check_array(char *s)
{
for(int i = 0                                ; i < 1000 ; i++)
{
if(strcmp(ST[i].symbol_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].symbol_name,s)==0)
{
if(ST[i].nested_val == current_nested_val)
{
return 1 ;
}
}
}

return 0 ;
}

int check_duplicate(char* str)
{
for(int i=0 ; i<1000 ; i++)
```

```
{
if(strcmp(ST[i].symbol_name, str) == 0 && strcmp(ST[i].class, "Function") == 0)
{
if(ST[i].define==0){
printf("ERROR: Cannot Redeclare same function!\n");
printf("\nUNSUCCESSFUL: INVALID PARSE\n");
exit(0) ;}
else ST[i].define=1 ;
}
}
}

int check_declaration(char* str, char *check_type)
{
for(int i=0 ; i<1000 ; i++)
{
if(strcmp(ST[i].symbol_name, str) == 0 && strcmp(ST[i].class, "Function") == 0 &&
ST[i].define==1|| strcmp(ST[i].symbol_name,"printf")==0 )
{
return 1 ;
}
}
return 0 ;
}
int func_exist(char* str)
{
printf("IN func exist ++++++++++++++++++++++++++++++\n");
for(int i=0 ; i<1000 ; i++)
{
if((strcmp(ST[i].symbol_name, str) == 0 )&& (strcmp(ST[i].class, "Function") == 0) &&
(strcmp(ST[i].symbol_name,"printf")!=0) && (ST[i].define==1))
{

return 1 ;
}
}

return 0 ;
}

int check_params(char* type_specifier)
```

```
{
if(!strcmp(type_specifier, "void"))
{
printf("ERROR: Here, Parameter cannot be of void type\n");
printf("\nUNSUCCESSFUL: INVALID PARSE\n");
exit(0) ;
}
return 0 ;
}

void insert_SymbolTable_paramscount(char *s, int count)
{
for(int i = 0                                ; i < 1000 ; i++)
{
if(strcmp(ST[i].symbol_name,s)==0 )
{
ST[i].params_count = count ;
}
}
}

int getSTparamscount(char *s)
{
for(int i = 0                          ; i < 1000 ; i++)
{
if(strcmp(ST[i].symbol_name,s)==0 )
{
return ST[i].params_count ;
}
}
return -2 ;
}
int getSTdimensioncount(char *s) {
for (int i = 0 ; i < 1000 ; i++) {
if (strcmp(ST[i].symbol_name, s) == 0) {
return atoi(ST[i].array_dimensions) ;
}
}
return -2 ;
}
char gettype(char *s, int flag)
```

```
{
for(int i = 0                    ; i < 1001 ; i++ )
{
if(strcmp(ST[i].symbol_name,s)==0)
{
return ST[i].symbol_type[0] ;
}
}

}

void printConstantTable(){
printf("%20s | %20s\n", "CONSTANT","TYPE");
for(int i = 0 ; i < 1000 ; ++i){
if(CT[i].exist == 0)
continue ;

printf("%20s | %20s\n", CT[i].constant_name, CT[i].constant_type) ;
}
}

void printSymbolTable(){
printf("%10s | %18s | %10s | %10s | %10s | %10s | %10s | %10s\n","SYMBOL",
"CLASS", "TYPE","VALUE","DIMENSIONS","PARAMETERS","PARAMETER COUNT",
"NESTING", "LINE NO");
for(int i = 0 ; i < 1000 ; ++i){
if(ST[i].exist == 0)
continue ;
printf("%10s | %18s | %10s | %10s | %10s | %10s | %15d | %10d | %d\n",
ST[i].symbol_name, ST[i].class, ST[i].symbol_type,
ST[i].value,ST[i].array_dimensions,ST[i].parameters, ST[i].params_count,
ST[i].curr_nest_val,ST[i].line_number) ;
}
}
char current_identifier[20] ;
char current_whatever[20] ;
char current_type[20] ;
char current_value[20] ;
char current_function[20] ;
char previous_operator[20] ;
int flag ;
```

```
int line_no = 1 ;

%}

num     [0-9]
alpha     [a-zA-Z]
alphanum {alpha}|{num}
escape_sequences 0|a|b|f|n|r|t|v|"\\"|"\'"|"\""
ws                          [ \t\r\f\v]+
%x MLCOMMENT
DE "define"
IN "include"

%%

int nested_count = 0 ;
int check_nested = 0 ;

\n     {}
"#include"[ ]*"<"{alpha}({alphanum})*".h>"     { }
"#define"[ ]+(_|{alpha})({alphanum})*[ ]*(.)+ { }
"//".*
                                        { }

"/*"                                        { BEGIN MLCOMMENT ; }
<MLCOMMENT>"/*"                                                 { ++nested_count ;
check_nested = 1 ;
}
<MLCOMMENT>"*"+"/"                              { if (nested_count) --nested_count ;
else{ if(check_nested){
check_nested = 0 ;
BEGIN INITIAL ;
}
else{
BEGIN INITIAL ;
}
}
}
<MLCOMMENT>"*"+
          ;
```

```
<MLCOMMENT>[^/*\n]+
        ;
<MLCOMMENT>[/]
        ;
<MLCOMMENT>\n
        ;
<MLCOMMENT><<EOF>>                                              { printf("Line No. %d
ERROR: MULTI LINE COMMENT NOT CLOSED\n", yylineno) ; return 0 ;}

"[" {return *yytext ;}
"]" {return *yytext ;}
"(" {return *yytext ;}
")" {return *yytext ;}
"{" {return *yytext ;}
"}" {return *yytext ;}
"," {return *yytext ;}
";" {return *yytext ;}

"char"                  { strcpy(current_type,yytext) ; insert_SymbolTable(yytext,
"Keyword"); return CHAR ;}
"double"        { strcpy(current_type,yytext) ; insert_SymbolTable(yytext, "Keyword");
return DOUBLE ;}
"else"                  { insert_SymbolTable(yytext, "Keyword"); return ELSE ;}
"float"                 { strcpy(current_type,yytext) ; insert_SymbolTable(yytext,
"Keyword");return FLOAT ;}
"NULL" {return NULL_CONST ;}
"while"                 { insert_SymbolTable(yytext, "Keyword"); return WHILE ;}
"do"            { insert_SymbolTable(yytext, "Keyword"); return DO ;}
"for"           { insert_SymbolTable(yytext, "Keyword"); return FOR ;}
"if"            { insert_SymbolTable(yytext, "Keyword"); return IF ;}
"int"           { strcpy(current_type,yytext) ; insert_SymbolTable(yytext,
"Keyword");return INT ;}
"long"                  { strcpy(current_type,yytext) ; insert_SymbolTable(yytext,
"Keyword"); return LONG ;}
"return"        { insert_SymbolTable(yytext, "Keyword"); return RETURN ;}
"short"                 { strcpy(current_type,yytext) ; insert_SymbolTable(yytext,
"Keyword"); return SHORT ;}
"signed"        { strcpy(current_type,yytext) ; insert_SymbolTable(yytext, "Keyword");
return SIGNED ;}
"sizeof"        { insert_SymbolTable(yytext, "Keyword"); return SIZEOF ;}
```

```
"struct"        { strcpy(current_type,yytext) ; insert_SymbolTable(yytext, "Keyword");
return STRUCT ;}
"unsigned"      { insert_SymbolTable(yytext, "Keyword"); return UNSIGNED ;}
"void"                  { strcpy(current_type,yytext) ; insert_SymbolTable(yytext,
"Keyword"); return VOID ;}
"break"                 { insert_SymbolTable(yytext, "Keyword"); return BREAK ;}
"continue"      { insert_SymbolTable(yytext, "Keyword"); return CONTINUE ;}
"goto"                  { insert_SymbolTable(yytext, "Keyword"); return GOTO ;}
"switch"        { insert_SymbolTable(yytext, "Keyword"); return SWITCH ;}
"case"                  { insert_SymbolTable(yytext, "Keyword"); return CASE ;}
"default"       { insert_SymbolTable(yytext, "Keyword"); return DEFAULT ;}

("\"")[^\n\"]*("\"")
{strcpy(current_whatever,yytext);strcpy(current_value,yytext);
insert_ConstantTable(yytext,"String Constant"); return string_constant ;}
("\"")[^\n\"]*                          { printf("Line No. %d ERROR: UNCLOSED STRING -
%s\n", yylineno, yytext) ; return 0 ;}
("\'")(("\\"({escape_sequences}))|.)("\'") {strcpy(current_whatever,yytext)
;strcpy(current_value,yytext); insert_ConstantTable(yytext,"Character Constant");
return character_constant ;}
("\'")((((("\\")[^0abfnrtv\\\"\'][^\n\']*))|[^\n\"][^\n\"]+)("\'") {printf("Line No. %d
ERROR: NOT A CHARACTER - %s\n", yylineno, yytext) ; return 0 ; }
{num}+(\.{num}+)?e{num}+        {strcpy(current_whatever,yytext)
;strcpy(current_value,yytext); insert_ConstantTable(yytext, "Floating Constant"); return
float_constant ;}
{num}+\.{num}+      {strcpy(current_whatever,yytext) ;strcpy(current_value,yytext);
insert_ConstantTable(yytext, "Floating Constant"); return float_constant ;}
{num}+      {strcpy(current_whatever,yytext) ;strcpy(current_value,yytext);
insert_ConstantTable(yytext, "Number Constant"); yylval = atoi(yytext) ; return
integer_constant ;}
(_|{alpha})({alpha}|{num}|_)*
{strcpy(current_whatever,yytext);strcpy(current_identifier,yytext);insert_SymbolTable(
yytext,"Identifier"); return identifier ;}
(_|{alpha})({alpha}|{num}|_)*/\[
{strcpy(current_whatever,yytext);strcpy(current_identifier,yytext);insert_SymbolTable(
yytext,"Array Identifier"); return array_identifier ;}
{ws}
        ;

"++"                                                    {return INCREMENT ;}
"--"                                                    {return DECREMENT ;}
```

```
"!"                                                     {return NOT ;}
"+="                                                    {return ADD_EQUAL ;}
"-="                                                    {return
SUBTRACT_EQUAL ;}
"*="                                                    {return
MULTIPLY_EQUAL ;}
"/="                                                    {return DIVIDE_EQUAL ;}
"%="                                                    {return MOD_EQUAL ;}
"&&"                                                    {return AND_AND ;}
"||"                                                    {return OR_OR ;}
">"                                                     {return GREAT ;}
"<"                                                     {return LESS ;}
">="                                                    {return GREAT_EQUAL ;}
"<="                                                    {return LESS_EQUAL ;}
"=="                                                    {return EQUAL ;}
"!="                                                    {return NOT_EQUAL ;}
"." {return DOT ;}
":" {return *yytext ;}
"+"                                                     {return *yytext ;}
"-"                                                     {return *yytext ;}
"*"                                                     {return *yytext ;}
"/"                                                     {return *yytext ;}
"="                                                     {return *yytext ;}
"%"                                                     {return *yytext ;}
"&"                                                     {return *yytext ; }
"^"                                                     {return *yytext ; }
.                                                       { flag = 1 ;
if(yytext[0] == '#')
printf("Line No. %d PREPROCESSOR ERROR - %s\n", yylineno, yytext) ;
else
printf("Line No. %d ERROR: ILLEGAL CHARACTER - %s\n", yylineno, yytext) ;
return 0 ;}

%%
```

PARSER CODE

```
%{
void yyerror(char *s) ;
int yylex() ;
#include "stdio.h"
#include "stdlib.h"
#include "ctype.h"
#include "string.h"
#include "semantic.h"
void insert_type() ;
void insert_value() ;
void top_() ;
void insert_dimensions() ;
void insert_parameters() ;
void remove_scope(int) ;
int check_scope(char *) ;
int check_function(char *) ;
void insert_SymbolTable_nest(char *, int) ;
void insert_SymbolTable_paramscount(char *, int) ;
int isNotPointer(char *) ;
int getSTparamscount(char *) ;
int getSTdimensioncount(char *) ;
int check_duplicate(char *) ;
int check_declaration(char *, char *) ;
int check_params(char *) ;
int func_exist(char*) ;
void updateDefine(char *) ;
int duplicate(char *s) ;
int check_array(char *) ;
void insert_SymbolTable_function(char *) ;
char gettype(char *, int) ;
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 appendTop(char*) ;
    void start_function() ;
    void end_function() ;
    void arggen() ;
    void callgen() ;
extern int flag ;
int insert_flag = 0 ;
int call_params_count=0 ;
    int array_flag = 0 ;
    int array_tac_flag = 0 ;
    int top = 0,count=0,ltop=0,lno=0 ;
    char temp[3] = "t";
extern char current_identifier[20] ;
extern char current_whatever[20] ;
extern char current_type[20] ;
extern char current_value[20] ;
extern char current_function[20] ;
extern char previous_operator[20] ;
extern int current_nested_val ;
extern int switch_index ;
extern char switch_array[20][20] ;
char currfunctype[100] ;
char currfunccall[100] ;
extern int params_count ;
int call_params_count ;
int current_array_dimensions ;
int actual_dimensions ;
char* getVariableType(char* identifier) ;
int checkTypeCompatibility(char* expectedType, char* actualType) ;
char* getParameterType(char* funcName, int paramIndex) ;
char* getArgumentType(int argIndex) ;
char current_call_argument_types[10][20] ;
int current_exp_type = -1 ;
int printCount=0 ;

int switch_index=0 ;
char switch_array[20][20] ;
extern FILE *yyin ;
```

```
extern int yylineno ;
extern char *yytext ;
void insert_SymbolTable_type(char *,char *) ;
void insert_SymbolTable_value(char *, char *) ;
void insert_ConstantTable(char *, char *) ;
void insert_SymbolTable_arraydim(char *, char *) ;
void insert_SymbolTable_funcparam(char *, char *) ;
void printSymbolTable() ;
void printConstantTable() ;
int newCheckCompatibility(int a, int b) ;
int giveIndex(char* name) ;
int checkConstantTable(char* s) ;
struct SymbolTable{
char symbol_name[100] ;
char symbol_type[100] ;
char array_dimensions[100] ;
char class[100] ;
char value[100] ;
char parameters[100] ;
int line_number ;
int exist ;
int define ;
int nested_val ;
int curr_nest_val ;
int params_count ;
} ;

struct ConstantTable{
char constant_name[100] ;
char constant_type[100] ;
int exist ;
} ;

extern struct ConstantTable CT[1000] ;

extern struct SymbolTable ST[1000] ;

int newGetType(char* s) ;
int current_exp_type2 = -1 ;
char* identifier2 ;
char* struck_var ;
```

```
char* curr_indi ;
//char* onlyVariable ;

%}

%nonassoc IF
%token INT CHAR FLOAT DOUBLE LONG SHORT SIGNED UNSIGNED STRUCT
STRUCT_ASSIGN
%token RETURN MAIN DOT

%token VOID NULL_CONST
%token WHILE FOR DO
%token BREAK CONTINUE GOTO
%token ENDIF
%token SWITCH CASE DEFAULT
%token identifier array_identifier
%token integer_constant string_constant float_constant character_constant
%nonassoc ELSE

%right MOD_EQUAL
%right MULTIPLY_EQUAL DIVIDE_EQUAL
%right ADD_EQUAL SUBTRACT_EQUAL
%right '='

%left OR_OR
%left AND_AND
%left '^'
%left EQUAL NOT_EQUAL
%left LESS_EQUAL LESS GREAT_EQUAL GREAT
%left '+' '-'
%left '*' '/' '%'

%right SIZEOF
%right NOT
%left INCREMENT DECREMENT

%start begin_parse

%%

begin_parse
```

```
: declarations
;

declarations
: declaration declarations
|
;

declaration
: variable_dec
| function_dec
| structure_dec
| fun_decl
;

structure_dec
: STRUCT identifier { insert_type() ; } '{' structure_content '}' struct_name ';'
;

structure_content
: variable_dec structure_content
| structure_dec structure_content
|
;

struct_name
: variables
|
;

variable_dec
: structure_initialize ';'
| datatype variables ';'
;

structure_initialize
: STRUCT struct_init_identifier struct_star variables
;

struct_init_identifier
```

```
: identifier { if (gettype(current_identifier, 0) != 's') { yyerror("Identifier undeclared or
out of scope\n"); }}
;

struct_star
: '*' struct_star
|
;

variables
: identifier_name multiple_variables
;

multiple_variables
: ',' variables
|
;

identifier_name
: identifier {
push(current_identifier) ;
identifier2 = strdup(current_identifier) ;
if (duplicate(current_identifier)) { yyerror("Duplicate value!\n"); }
insert_SymbolTable_nest(current_identifier, current_nested_val) ;
insert_type() ;
} extended_identifier { if (($3<= 5) &&
(newCheckCompatibility(newGetType(identifier2), $3) != 1)) {yyerror("ERROR: Type
Mismatch\n"); exit(0) ;}}
| array_identifier {
push(current_identifier) ;
if (duplicate(current_identifier)) { yyerror("Duplicate value!\n"); }
insert_SymbolTable_nest(current_identifier, current_nested_val) ;
insert_type() ;
} extended_identifier
;

extended_identifier
: array_iden {$$ = 100 ;}
| '=' NULL_CONST {if(isNotPointer(current_identifier) )yyerror("NULL ERROR!!!!"); $$ = 6
;}
```

```
| '=' {strcpy(previous_operator, "="); push("=");} condition_ternary{ if($3 <1)
{yyerror("Invalid expression.\n");} $$ = $3 ; codeassign() ;}
;

array_iden
: '[' array_dims
|
;

array_dims
: integer_constant { insert_dimensions() ; } ']' initialization{if($$ < 1) {yyerror("Array
must have size greater than 1!\n");}}
| ']' string_initialization
| float_constant { insert_dimensions() ; } ']' initialization{yyerror("Array size must be an
integer!\n");}
| string_constant { insert_dimensions() ; } ']' initialization{yyerror("Array size must be an
integer!\n");}
| expression { insert_dimensions() ; } ']' initialization{if($1 != 1 && $1 != 2) {yyerror("Array
size must be an integer!\n");}else if($$ < 1) {yyerror("Array must have size greater than
1!\n");} current_exp_type = -1 ;}
;

initialization
: string_initialization
| array_initialization
|
;

string_initialization
: '=' { strcpy(previous_operator, "="); } string_constant { codegencon() ; insert_value() ; }
;

array_initialization
: '=' { strcpy(previous_operator, "="); } multi_dim
;

multi_dim
: '{' arr_elements '}'
;

arr_elements
```

```
: array_values
| '{' array_values '}'
| multi_dim
| multi_dim ',' '{' arr_elements '}'
| '{' array_values '}' ',' arr_elements
;

array_values
: constant multiple_array_values
;

multiple_array_values
: ',' array_values
|
;

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

star
: star '*' { strcat(current_type, "*"); }
| '*' { strcat(current_type, "*"); }
;

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_dec
: function_datatype function_parameters
;

fun_decl : ';';

function_datatype
: datatype identifier '(' {
strcpy(currfunctype, current_type) ;
check_duplicate(current_identifier) ;
if(func_exist(current_function)==1) yyerror(" ERROR:Function already defined\n");
insert_SymbolTable_function(current_identifier) ;
strcpy(current_function, current_identifier) ;
insert_type() ;
}
;

func_param : func_param ',' datatype | datatype ;

function_parameters
: parameters ')' { params_count = 0 ; start_function() ;} body
;
body : { updateDefine(current_function) ; }multiple_statement { end_function() ;}
| ';'                                                                                ;
parameters
: datatype { check_params(current_type) ; } all_parameter_identifiers {
insert_SymbolTable_paramscount(current_function, params_count) ; }
|
;

all_parameter_identifiers
: parameter_identifier multiple_parameters
;
```

```
multiple_parameters
: ',' parameters
|
;

parameter_identifier
: identifier {
insert_parameters() ;
insert_type() ;
insert_SymbolTable_nest(current_identifier, 1) ;
params_count++ ;
} extended_parameter
;

extended_parameter
: '[' ']'
|
;

statement
: expression_statement {$$ = $1 ;}
| multiple_statement
| conditional_statements
| iterative_statements
| return_statement
| break_statement
| switch_statement
| variable_dec
| function_dec
;

switch_statement
: SWITCH '(' expression ')' {if ($3 != 1) { yyerror("ERROR: Condition must have integer
value!\n"); }} '{' case_statement'}'
;

case_statement
: CASE int_char_const {
if (switch_check(current_value, switch_array, switch_index)) {
yyerror("ERROR: Duplicate case value");
```

```
} else {
switch_insert(current_value, switch_array, &switch_index) ;
}
} ':' case_body case_break case_statement
| DEFAULT ':' case_body case_break
|
;

case_body
: statement {$$ = $1 ;}
|
;

case_break
: break_statement
|
;

int_char_const
: integer_constant {$$ = 1 ; codegencon() ;}
| character_constant {$$ = 2 ;codegencon();}
;

multiple_statement
: { current_nested_val++ ; } '{' statements '}' { remove_scope(current_nested_val) ;
current_nested_val-- ; }
;

statements
: statement statements {$$ = $2>$2? $1 : $2 ; }
|
;

expression_statement
: expression ';'{if($1 == -1) {yyerror("Cannot perform this operation\n."); $$ = -1 ;} else
{$$ = 1 ;}current_exp_type = -1 ;}
| expression {current_exp_type = -1 ;} ',' expression_statement {$$ = $1>$4? $1 : $4 ;}
| ';'
;

conditional_statements
```

```
: IF '(' expression ')' {label1() ; if ($3 != 1) { yyerror("ERROR: Condition must have integer value!\n"); $$ = -1 ;} else $$ = 1 ; } statement {label2() ;}
extended_conditional_statements
| condition_ternary {$$ = $1 ;}
;

extended_conditional_statements
: ELSE statement {label3() ; $$ = $2 ;}
| {label3() ;}
;

condition_ternary
: '(' expression {current_exp_type = -1 ;} ')' '?' condition_ternary ':' condition_ternary
{if($2>0 && $2<4 && $6 > 0 && $6<4 && $8>0 && $8<4) $$ = 1 ; else $$ = -1 ;}
| '(' conditional_statements ')' {$$ = $2 ;}
| expression { $$ = $1 ; current_exp_type = -1 ;}
;

iterative_statements
: WHILE '(' {label4() ;} expression ')' {label1() ; if ($4 < 1 || $4 > 3) { yyerror("ERROR: Condition must have integer value!\n"); } $$ = 1 ;} statement {label5() ;}
| FOR '(' for_initialization {label4() ;} expression ';' { if($5<1 || $5 > 3){yyerror("Here, condition must have integer value!\n");} label1() ;} expression {current_exp_type = -1 ; $$ = 1 ;} ')' statement {label5() ;}
|{label4() ;} DO statement WHILE '(' expression ')' {label1() ; label5() ; if ($6 < 1 || $6>3) { yyerror("ERROR: Condition must have integer value!\n"); } $$ = 1 ;} ';'
;

for_initialization
: variable_dec
| mutable ',' for_initialization
| expression ';' {label4() ; current_exp_type = -1 ;}
| expression ','{current_exp_type = -1 ;} for_initialization
| variable_dec ',' for_initialization
| ';'
;

simple_expression_for
: simple_expression
| simple_expression ',' simple_expression_for
| ';'
```

```
;

expression_for
: expression {current_exp_type = -1 ;}
| expression ','{current_exp_type = -1 ;} expression_for
|
;

return_statement
: RETURN ';' { if (strcmp(currfunctype, "void")) { yyerror("ERROR: Cannot have void
return for non-void function!\n"); }}
| RETURN expression ';' { if (!strcmp(currfunctype, "void")) {
yyerror("Non-void return for void function!"); current_exp_type = -1 ;
}

//printSymbolTable() ;
if (((strcmp(currfunctype,"int") == 0|| strcmp(currfunctype,"char") == 0) && ($2 != 1) &&
($2 != 2))) {
yyerror("Expression doesn't match return type of function\n");
}
else if (strcmp(currfunctype,"char*") == 0 && ($2 != 0)) {
yyerror("Expression doesn't match return type of function\n");
}
else if (strcmp(currfunctype,"int*") == 0 && ($2 != 4)) {
yyerror("Expression doesn't match return type of function\n");
}
else if (strcmp(currfunctype,"float*") == 0 && ($2 != 5)) {
yyerror("Expression doesn't match return type of function\n");
}

}
;

break_statement
: BREAK ';'
;

expression
:
{}NULL_CONST {if(isNotPointer(current_identifier) )yyerror("NULL ERROR!!!!"); else
$$=6 ; push("NULL"); }
```

```
| expression '+' term {
if (!($1 >= 1 && $1 <4 && $3 >= 1 && $3 <4)) {
yyerror("Cannot perform arithmetic operations on strings/pointers");
$$ = -1 ;
} else {
$$ = $1>$3? $1 : $3 ;
}
}
|mutable '=' { push("=");} expression { strcpy(previous_operator, "=");
if (($1 >= 1 && $1 <4 && $4 >= 1 && $4 <4) || ($1 == $4) || $4 == 6) { $$ = $1>$4? $1 : $4 ; }
else { $$ = -1 ; yyerror("Type Mismatch\n"); }
codeassign() ;
}
| mutable ADD_EQUAL {push(current_identifier);push("+=");} expression {
strcpy(previous_operator, "+=");
if ($1 >= 1 && $1 <4 && $4 >= 1 && $4 <4) { $$ = $1>$4? $1 : $4 ; }
else { $$ = -1 ; yyerror("Type Mismatch\n"); }
codeassign() ;
}
| mutable SUBTRACT_EQUAL {push(current_identifier) ; push("-=");} expression {
strcpy(previous_operator, "-=");
if ($1 >= 1 && $1 <4 && $4 >= 1 && $4 <4) { $$ = $1>$4? $1 : $4 ; }
else { $$ = -1 ; yyerror("Type Mismatch\n"); }
codeassign() ;
}
| mutable MULTIPLY_EQUAL {push(current_identifier) ; push("*=");} expression {
strcpy(previous_operator, "*=");
if ($1 >= 1 && $1 <4 && $4 >= 1 && $4 <4) { $$ = $1>$4? $1 : $4 ; }
else { $$ = -1 ; yyerror("Type Mismatch\n"); }
codeassign() ;

}
| mutable DIVIDE_EQUAL {push(current_identifier) ; push("/=");} expression {
strcpy(previous_operator, "/=");
if ($1 >= 1 && $1 <4 && $4 >= 1 && $4 <4) { $$ = $1>$4? $1 : $4 ; }
else { $$ = -1 ; yyerror("Type Mismatch\n"); }
codeassign() ;
}
| mutable MOD_EQUAL {push(current_identifier) ; push("%=");} expression {
strcpy(previous_operator, "%=");
if ($1 >= 1 && $1 <4 && $4 >= 1 && $4 <4) { $$ = $1>$4? $1 : $4 ; }
```

```
else { $$ = -1 ; yyerror("Type Mismatch\n"); }
codeassign() ;
}
| mutable INCREMENT {push(current_identifier) ; push("++"); if ($1 >= 1) $$ = $1 ; else
$$ = -1 ; genunary() ;}
| mutable DECREMENT {push(current_identifier) ; push("--"); if ($1 >= 1) $$ = $1 ; else $$
= -1 ; genunary() ;}
| simple_expression { if ($1 >= 0) $$ = $1 ; else $$ = -1 ;}
;

simple_expression
: simple_expression OR_OR and_expression { if ($1 >= 1 && $1 <4 && $3 >= 1 && $3 <4)
{$$ = $1>$3? $1 : $3 ;} else {$$ = -1 ; }codegen() ;}
| and_expression { if ($1 >= 0) {$$ = $1 ;} else {$$ = -1 ; }}
;

and_expression
: and_expression AND_AND unary_relation_expression { if ($1 >= 0 && $3 >= 0) $$ =
$1>$3? $1 : $3 ; else $$ = -1 ; codegen() ;}
| unary_relation_expression { if ($1 >= 0) $$ = $1 ; else $$ = -1 ; }
;

unary_relation_expression
: NOT unary_relation_expression { if ($2 >= 0) $$ = $2 ; else $$ = -1 ; codegen() ;}
| regular_expression { if ($1 >= 0) $$ = $1 ; else $$ = -1 ; }
;

regular_expression
: regular_expression relational_operators sum_expression { if ($1 >= 0 && $3 >=0) $$ =
1 ; else $$ = -1 ; codegen() ;}
| sum_expression { $$ = $1 ;}
;

relational_operators
                : GREAT_EQUAL{strcpy(previous_operator,">="); push(">=");}
                | LESS_EQUAL{strcpy(previous_operator,"<="); push("<=");}
                | GREAT{strcpy(previous_operator,">"); push(">");}
                | LESS{strcpy(previous_operator,"<"); push("<");}
                | EQUAL{strcpy(previous_operator,"=="); push("==");}
                | NOT_EQUAL{strcpy(previous_operator,"!="); push("!=");}   ;
```

```
sum_expression
: sum_expression sum_operators term { if ($1 >= 1 && $3 >= 1) $$ = $1>$3? $1 : $3 ; else
$$ = -1 ; codegen() ;}
| term { if ($1 >= 0) $$ = $1 ; else $$ = -1 ; }
;

sum_operators
                    : '+' {push("+");}
                    | '-' {push("-");}
| '='{yyerror("Line no. Error: Invalid lhs value\n"); exit(0) ;}
;

term
: term MULOP factor { if ($1 >= 1 && $1 <4 && $3 >= 1 && $3 <4) $$ = $1>$3? $1 : $3 ; else
$$ = -1 ; codegen() ;}
| factor { if ($1 >= 0) $$ = $1 ; else $$ = -1 ; }
;

MULOP
: '*' {push("*");}
                    | '/' {push("/");}
                    | '%' {push("%");} ;
;

factor
: immutable { if ($1 >= 0) $$ = $1 ; else $$ = -1 ; }
| mutable { if ($1 >= 0) $$ = $1 ; else $$ = -1 ; }
;

mutable
: identifier { push(current_identifier) ;
if (!check_scope(current_identifier)) {
yyerror("Identifier undeclared or out of scope");

}

int typ = newGetType(current_identifier) ;

$$ = typ ;
current_exp_type = current_exp_type<$$? $$: current_exp_type ;
```

```
} stuck ;
| '&' identifier {
if (!check_scope(current_identifier)) { yyerror("Identifier undeclared or out of
scope\n"); }
if (!check_array(current_identifier)) { yyerror("Array Identifier has No Subscript\n"); }
int typ = newGetType(current_identifier) ;

if(typ == 1)
$$ = 4 ;
else if(typ == 3)
$$ = 5 ;
current_exp_type = current_exp_type<$$? $$: current_exp_type
;
char str2[100] ;
snprintf(str2, sizeof(str2), "&%s", current_identifier) ;
strcpy(current_value, str2) ;
codegencon() ;
}
| identifier INCREMENT {
if (!check_scope(current_identifier)) { yyerror("Identifier undeclared or out of
scope\n"); }
if (!check_array(current_identifier)) { yyerror("Array Identifier has No Subscript\n"); }
int typ = newGetType(current_identifier) ;

$$ = typ ;
current_exp_type = current_exp_type<$$? $$: current_exp_type
;
push(current_identifier) ;
push("++");
genunary() ;
}
| identifier DECREMENT {
if (!check_scope(current_identifier)) { yyerror("Identifier undeclared or out of
scope\n"); }
if (!check_array(current_identifier)) { yyerror("Array Identifier has No Subscript\n"); }
int typ = newGetType(current_identifier) ;

$$ = typ ;
current_exp_type = current_exp_type<$$? $$: current_exp_type
;
push(current_identifier) ;
```

```
push("--");
genunary() ;
}
| INCREMENT identifier {
if (!check_scope(current_identifier)) { yyerror("Identifier undeclared or out of
scope\n"); }
if (!check_array(current_identifier)) { yyerror("Array Identifier has No Subscript\n"); }
int typ = newGetType(current_identifier) ;

$$ = typ ;
current_exp_type = current_exp_type<$$? $$: current_exp_type
;
push("++");
push(current_identifier) ;
genunary() ;
}
| DECREMENT identifier {
if (!check_scope(current_identifier)) { yyerror("Identifier undeclared or out of
scope\n"); }
if (!check_array(current_identifier)) { yyerror("Array Identifier has No Subscript\n"); }
int typ = newGetType(current_identifier) ;

$$ = typ ;
current_exp_type = current_exp_type<$$? $$: current_exp_type
;
push("--");
push(current_identifier) ;
genunary() ;
}
| '+' identifier {
if (!check_scope(current_identifier)) { yyerror("Identifier undeclared or out of
scope\n"); }
if (!check_array(current_identifier)) { yyerror("Array Identifier has No Subscript\n"); }
int typ = newGetType(current_identifier) ;

$$ = typ ;
current_exp_type = current_exp_type<$$? $$: current_exp_type
;
}
| '-' identifier {
```

```
if (!check_scope(current_identifier)) { yyerror("Identifier undeclared or out of
scope\n"); }
if (!check_array(current_identifier)) { yyerror("Array Identifier has No Subscript\n"); }
int typ = newGetType(current_identifier) ;

$$ = typ ;
current_exp_type = current_exp_type<$$? $$: current_exp_type
;
}
| array_identifier { push(current_identifier) ;
array_flag = 1 ;
actual_dimensions = getSTdimensioncount(current_identifier) ;
current_array_dimensions = 0 ;
if (!check_scope(current_identifier)) { yyerror("Identifier undeclared or out of
scope\n"); }
} '[' { current_array_dimensions++ ; } expression ']' extended_array_dimension {
int typ = newGetType(current_identifier) ;

$$ = typ ;
current_exp_type = current_exp_type<$$? $$: current_exp_type
;
}
| struct_identifier STRUCT_ASSIGN mutable {
if (!check_scope(current_identifier)) { yyerror("Identifier undeclared or out of
scope\n"); }
if (!check_array(current_identifier)) { yyerror("Array Identifier has No Subscript\n"); }
int typ = newGetType(current_identifier) ;

$$ = typ ;
current_exp_type = current_exp_type<$$? $$: current_exp_type
        ;
}
;
stuck: | DOT{ appendTop("."); } identifier { struck_var = strdup(current_identifier) ;
appendTop(current_identifier) ;
push("=");}'='{top_();}expression{if(newCheckCompatibility(newGetType(struck_var),
$7) != 1){ char* error_message ;sprintf(error_message, "Type mismatch in function call:
expected %s but got %s\n", newGetType(struck_var), $7) ; yyerror(error_message)
;}top_();codeassign();};

struct_identifier
```

```
: identifier {
if (!check_scope(current_identifier)) { yyerror("Identifier undeclared or out of
scope\n"); }
if (!check_array(current_identifier)) { yyerror("Array Identifier has No Subscript\n"); }
if (gettype(current_identifier, 0) == 's') $$ = 0 ;
else { yyerror("Identifier not of struct type\n"); $$ = -1 ; }
}
;

extended_array_dimension
: '[' { current_array_dimensions++ ; } expression ']' extended_array_dimension
| { if (current_array_dimensions > actual_dimensions) { yyerror("Dimensions mismatch
of the array\n"); }}
;

immutable
: '(' expression ')' { if ($2 >= 1) $$ = $2 ; else $$ = -1 ; }
| call {$$ = $1 ;}
| constant_noString { if ($1 >= 0) $$ = $1 ; else $$ = -1 ; }
;
constant_noString : integer_constant { insert_type() ; $$ = 1 ; codegencon() ;}
| string_constant { codegencon() ; insert_type() ; $$ = 0 ;}
| float_constant {codegencon() ; insert_type() ; $$ = 3 ; }
| character_constant { insert_type() ; codegencon() ; $$ = 2 ; }
;

call
: identifier '(' {
strcpy(previous_operator, "(");
if (!check_declaration(current_identifier, "Function")) {
yyerror("Function not defined");
}
char tp = gettype(current_identifier,0) ;
int typ = newGetType(current_identifier) ;
if (strcmp(current_identifier, "printf") == 0)
typ = 1 ;

$$ = typ ;
current_exp_type = current_exp_type<$$? $$: current_exp_type ;

//insert_SymbolTable_function(current_identifier) ;
```

```
strcpy(currfunccall, current_identifier) ;
call_params_count = 0 ;
current_exp_type2 = $$ ;
} arguments ')' {
if (strcmp(currfunccall, "printf")) {
if (getSTparamscount(currfunccall) != call_params_count) {
yyerror("Number of parameters not same as number of arguments during function
call!");

}
for (int i = 0 ; i < call_params_count ; i++) {
char* expectedType = getParameterType(currfunccall, i) ;
char* actualType = getArgumentType(i) ;

if (!checkTypeCompatibility(expectedType, actualType)) {
char error_message[100] ;
sprintf(error_message, "Type mismatch in function call: expected %s but got %s for
argument %d",
expectedType, actualType, i+1) ;
yyerror(error_message) ;

}
}
}else printCount=call_params_count ;
call_params_count = 0 ;
$$ = current_exp_type2 ;
callgen() ;
}
;

arguments
: arguments_list
|
;

arguments_list
: expression {
arggen() ;
strcpy(current_call_argument_types[call_params_count],
getVariableType(current_whatever)) ;
call_params_count++ ;
```

```
} A
;

A
: ',' expression {
arggen() ;
strcpy(current_call_argument_types[call_params_count],
getVariableType(current_whatever)) ;
call_params_count++ ;
} A
|
;

constant
: integer_constant { insert_type() ; codegencon() ; $$ = 1 ; }
| string_constant { insert_type() ; codegencon() ; $$ = 0 ; }
| float_constant { insert_type() ; codegencon() ; $$ = 3 ; }
| character_constant { insert_type() ; codegencon() ; $$ = 2 ; }
;

%%

extern FILE *yyin ;
extern int yylineno ;
extern char *yytext ;
void insert_SymbolTable_type(char *,char *) ;
void insert_SymbolTable_value(char *, char *) ;
void insert_ConstantTable(char *, char *) ;
void insert_SymbolTable_arraydim(char *, char *) ;
void insert_SymbolTable_funcparam(char *, char *) ;

void yyerror(char *s) {
printf("Line No. : %d %s %s\n", yylineno, s, yytext) ;
flag = 1 ;
printf("\nUNSUCCESSFUL: INVALID PARSE\n");
}

void insert_type() {
insert_SymbolTable_type(current_identifier, current_type) ;
}
```

```
void insert_value() {
if (strcmp(previous_operator, "=") == 0) {
insert_SymbolTable_value(current_identifier, current_value) ;
}
}

void insert_dimensions() {
insert_SymbolTable_arraydim(current_identifier, current_value) ;
}

void insert_parameters() {
insert_SymbolTable_funcparam(current_function, current_type) ;
}

//int yywrap() {
// return 1 ;
//}*/
char* getVariableType(char* id) {

for(int i = 0                         ; i < 1001 ; i++ )
{
if(strcmp(ST[i].symbol_name,id)==0)
{
return (ST[i].symbol_type) ;
break ;
}
}

int t = newGetType(id) ;
if(t == 0)
return "char*";
else if(t == 1)
return "int";
else if(t == 2)
return "char";
else if(t == 3)
return "float";
else if(t == 4)
return "int*";
else if(t == 5)
return "float*";
```

```
return "error";
}

char* getParameterType(char* funcName, int paramIndex) {
char* p = NULL ;
int pc = 0 ;
for(int i = 0                                          ; i < 1000 ; i++)
{
if(strcmp(ST[i].symbol_name,funcName)==0 )
{
p = ST[i].parameters ;
pc = ST[i].params_count ;
break ;
}
}
if(p == NULL)
return "void";

if(paramIndex >= pc)
return "error";

char res[100] = "";
int count = -1 ;
for(int i = 0 ;i<100;i++){
if(p[i] == '\0')
break ;
if(p[i] == ' '){
count++ ;
continue ;
}

if(count == paramIndex){
int len = strlen(res) ; // Find the current length of the string.
res[len] = p[i] ; // Add the new character at the end of the string.
res[len + 1] = '\0';
}
else if(count>paramIndex){
break ;
}
```

```
}

return strdup(res) ;
}

char* getArgumentType(int argIndex) {

return current_call_argument_types[argIndex] ;
}

int checkTypeCompatibility(char* expectedType, char* actualType) {
if (strcmp(expectedType, actualType) == 0) {
return 1 ;
}
if (strcmp(expectedType, "float") == 0 && strcmp(actualType, "int") == 0) {
return 1 ;
}
return 0 ;
}
void printSymbolTable() ;
void printConstantTable() ;
struct stack
{
   char value[100] ;
   int labelvalue ;
}s[100],label[100] ;

void push(char *x)
{

   strcpy(s[++top].value,x) ;
}

void codegen()
{

   printf("t%d = %s %s %s\n",count,s[top-2].value,s[top-1].value,s[top].value);
   top = top - 2 ;
sprintf(temp, "t%d", count) ;
   strcpy(s[top].value,temp) ;
   count++ ;
```

```
}

void appendTop(char* s1)
{
   strcat(s[top].value,s1) ;

}

void codegencon()
{

   if(array_flag == 1){
         printf("t%d = 4 * %s\n",count ,current_whatever) ;
         count++ ;
         printf("t%d = &arr + t%d\n",count ,count-1) ;
         array_tac_flag = 1 ;
   }
   else
         printf("t%d = %s\n",count ,current_value) ;
   sprintf(temp, "t%d", count) ;
   push(temp) ;
   count++ ;
   array_flag = 0 ;
}

void codeassign()
{

   if(array_tac_flag == 1)
         printf("*%s = %s\n",s[top-2].value,s[top].value);
   else
         printf("%s = %s\n",s[top-2].value,s[top].value);
   array_tac_flag = 0 ;
   top = top - 2 ;
}

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) ;
    }

    if(strcmp(temp2,"--")==0)
    {
            printf("t%d = %s - 1\n", count, temp1) ;
            printf("%s = t%d\n", temp1, count) ;
    }

    if(strcmp(temp2,"++")==0)
    {
            printf("t%d = %s + 1\n", count, temp1) ;
            printf("%s = t%d\n", temp1, count) ;
    }
    count++ ;
    top = top -2 ;
}

void label1()
{
    printf("IF not %s goto L%d\n",s[top].value,lno);
    label[++ltop].labelvalue = lno++ ;
}

void label2()
{
    printf("goto L%d\n",lno);
```

```
   printf("L%d:\n",label[ltop].labelvalue);
   ltop-- ;
   label[++ltop].labelvalue=lno++ ;
}

void label3()
{
   printf("L%d:\n",label[ltop].labelvalue);
   ltop-- ;
}

void label4()
{
   printf("L%d:\n",lno);
   label[++ltop].labelvalue = lno++ ;
}

void label5()
{
   printf("goto L%d:\n",label[ltop-1].labelvalue);
   printf("L%d:\n",label[ltop].labelvalue);
   ltop = ltop - 2 ;
}

void start_function()
{
   printf("func begin %s\n",current_function);
}

void end_function()
{
   printf("func end\n\n");
}
void top_(){
int i=top ;
while(i>=0){ printf(" %s ",s[i].value);i--;}
printf("\n\n");
}

void arggen()
{
```

```
   printf("param %s\n", s[top].value) ;
top-- ;

}

void callgen()
{
   printf("refparam result\n");
   push("result");
if(strcmp(currfunccall,"printf"))
   printf("call %s, %d\n",currfunccall,getSTparamscount(currfunccall));
else printf("call %s, %d\n",currfunccall,printCount);
}

int main() {
yyin = fopen("input.c", "r");
if(yyin==NULL) printf("Unable to open file");
yyparse() ;

if (flag == 0) {
printf("VALID PARSE\n");
printf("%30s SYMBOL TABLE \n", " ");
printf("%30s %s\n", " ", "------------");
printSymbolTable() ;

printf("\n\n%30s CONSTANT TABLE \n", " ");
printf("%30s %s\n", " ", "--------------");
printConstantTable() ;
}
}
/*void yyerror(char *s) {
printf("Line No.: %d %s %s\n", yylineno, s, yytext) ;
flag = 1 ;
printf("\nUNSUCCESSFUL: INVALID PARSE\n");
}

void insert_type() {

insert_SymbolTable_type(current_identifier, current_type) ;
}
```

```
void insert_value() {
if (strcmp(previous_operator, "=") == 0) {
insert_SymbolTable_value(current_identifier, current_value) ;
}
}

void insert_dimensions() {
insert_SymbolTable_arraydim(current_identifier, current_value) ;
}

void insert_parameters() {
insert_SymbolTable_funcparam(current_function, current_type) ;
}*/

int yywrap() {
return 1 ;
}

int newGetType(char* s){
char* res = "";
int t = -1 ;

for(int i = 0                    ; i < 1001 ; i++ )
{
if(strcmp(ST[i].symbol_name,s)==0)
{
res = ST[i].symbol_type ;
break ;
}
}

if(strcmp(res,"char*") == 0 ){
t = 0 ;
}
else if (res[0] == 's') {
t = 0 ; // Indicate it's a string
}
else if (strcmp(res,"int") == 0) {
t = 1 ; // Indicate it's a numeric type
//current_exp_type = current_exp_type<1? 1: current_exp_type ;
```

```
}
else if (strcmp(res,"char") == 0) {
t = 2 ; // Indicate it's a numeric type
//current_exp_type = current_exp_type<2? 2: current_exp_type ;
}
else if (strcmp(res,"float") == 0) {
t = 3 ; // Indicate it's a numeric type
//current_exp_type = current_exp_type<3? 3: current_exp_type ;
}
else if (strcmp(res,"int*") == 0) {
t = 4 ; // Indicate it's a numeric type
//current_exp_type = current_exp_type<3? 3: current_exp_type ;
}
else if (strcmp(res,"float*") == 0) {
t = 5 ; // Indicate it's a numeric type
//current_exp_type = current_exp_type<3? 3: current_exp_type ;
}

else {
t = checkConstantTable(s) ;
}

return t ;
}

int checkConstantTable(char* s){
for(int i = 0                    ; i < 1001 ; i++ )
{
if(strcmp(CT[i].constant_name,s)==0)
{

if(strcmp(CT[i].constant_type,"Floating Constant") == 0)
return 3 ;
if(strcmp(CT[i].constant_type,"Number Constant") == 0)
return 1 ;
if(strcmp(CT[i].constant_type,"Character Constant") == 0)
return 2 ;
if(strcmp(CT[i].constant_type,"String Constant") == 0)
return 0 ;
}
}
```

```
return -1 ;
}

int newCheckCompatibility(int a, int b){
if(a == 1 || a== 2){
if(b == 1||b == 2)
return 1 ;
}
if(a == 3)
if(b == 1 || b == 2 || b == 3){
return 1 ;
}

return (a == b || b == 6) ;
}

int giveIndex(char* str){
for(int i=0 ; i<1000 ; i++)
{
if(strcmp(ST[i].symbol_name, str) == 0 && strcmp(ST[i].class, "Function") == 0
&&ST[i].define==1|| strcmp(ST[i].symbol_name,"printf")==0 )
{
return i ;
}
}
return -1 ;
}
```

OVERVIEW OF CODE:

LEX CODE

The lexer works in conjunction with a parser (indicated by the inclusion of "y.tab.h") and provides comprehensive token recognition along with sophisticated symbol table management. It's designed to handle both the basic lexical analysis tasks like tokenization and the more complex requirements of tracking symbols, scopes, and program structure.

Overview:

1. Data Structures:
    - Implements two main tables:
        - ConstantTable: Stores constants and their types
        - SymbolTable: A more complex table storing identifiers, functions, and various attributes like scope, parameters, line numbers etc.
2. Core Functionality:
    - Handles symbol table management for a C-like programming language
    - Uses hash-based storage for efficient lookup and insertion
    - Maintains scoping information through nesting levels
    - Tracks function declarations, definitions and parameters
    - Manages array declarations and dimensions
    - Keeps track of identifiers' line numbers and nested levels
3. Key Features:
    - Symbol management with scope handling
    - Function declaration and definition tracking
    - Support for arrays
    - Parameter counting and tracking
    - Type checking capabilities
    - Multi-level nested scope support
    - Line number tracking
    - Duplicate declaration checking
4. Pattern Recognition:
    - Recognizes C language keywords
    - Handles string and character constants
    - Processes numeric constants (integers and floating point)
    - Identifies identifiers and array identifiers
    - Recognizes operators and special symbols
    - Handles multi-line and single-line comments
    - Processes preprocessor directives
5. Error Handling:
    - Detects unclosed strings
    - Identifies illegal characters
    - Catches invalid character constants
    - Checks for function redeclarations
    - Validates parameter types

This is the foundation for a compiler, providing comprehensive token recognition and symbol management capabilities while maintaining detailed information about the

program structure and catching various syntax and semantic errors at the lexical analysis stage.

PARSER CODE

Here's an overview of the key components:

1. Header Section:

- Contains numerous function declarations for semantic analysis, symbol table management, and code generation
- Includes standard C libraries and a custom "semantic.h" header
- Defines important global variables and data structures including:
    - Symbol Table (ST) structure for tracking variables, functions, and their properties
    - Constant Table (CT) structure for managing constants
    - Various current state tracking variables (current_identifier, current_type, etc.)

2. Grammar Rules : The file defines grammar rules for parsing:

- Basic program structure and declarations
- Structure/struct definitions
- Variable declarations and initializations
- Function declarations and definitions
- Array declarations
- Expression handling including:
    - Arithmetic operations
    - Assignment operations
    - Conditional expressions
    - Loop constructs (while, for, do-while)
    - Switch statements
    - Return statements

3. Type Checking:

- Validates function parameters and return types
- Handles array dimensions and pointer declarations
- Managing type compatibility between expected and actual argument types with checkTypeCompatibility, getParameterType, and getArgumentType, which is robust for complex expressions.
- Using newGetType to determine types dynamically is effective. Make sure that getVariableType and getParameterType cover all types to avoid potential

runtime issues.

4. Symbol Table Management:
- Includes scope management functionality
- Tracks variable declarations and definitions
- Handles function parameters and nested scopes

5. Code Generation:
- Contains placeholders for generating intermediate or target code
- Includes functions for handling:
    - Unary operations (genunary)
    - Labels for control flow
    - codegen, codegencon, and codeassign effectively generate code for expressions, constants, and assignments.
    - The temporary variable count (count) for intermediate results (t0, t1, ...) appears accurate.

6. Array and Struct Handling:

    - array_identifier correctly checks for array bounds and validates dimensions, which is crucial for multi-dimensional arrays.
    - The use of extended_array_dimension to count and check dimensions is well-implemented.
    - For struct assignments (e.g., struct_identifier), you could expand to allow nested structures if needed.

7. Error Handling:

    - Your yyerror function prints error messages with line numbers, which is helpful.
    - The usage of yyerror within various grammar rules for undeclared identifiers, scope checks, and array bounds is excellent.
    - For complex expressions, especially involving nested structures or arrays, ensure sufficient testing to capture edge cases.

TEST CASES

TEST CASES WITHOUT CASES

1. //ERROR FREE - This test includes a declaration and a print statement

```
#include<stdio.h>
int main()
{
	//This is the first test program.
	int a;
	/* This is the declaration
	of an integer value */
	printf("Hello World");
	return 0;

}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
 func begin main
 t0 = "Hello World"
 param t0
 refparam result
 call printf, 1
 t1 = 0
 func end

 VALID PARSE
                          SYMBOL TABLE
                          ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |           Keyword |            |            |            |            |               0 |          0 | 4
      main |          Function |        int |            |            |            |               0 |          0 | 4
    printf |        Identifier |        int |            |            |            |               0 |          1 | 11
         a |        Identifier |        int |            |            |            |               0 |          1 | 7
    return |           Keyword |            |            |            |            |               0 |          1 | 12


                          CONSTANT TABLE
                          --------------
         CONSTANT |                TYPE
    "Hello World" |      String Constant
                0 |      Number Constant
```

2.//ERROR FREE - This test case includes a function

```
#include<stdio.h>
int multiply(int a)
{
	return 2*a;
}
int main()
{
	int a = 5;
```

```
        int b = multiply(a);
        printf("%d ", b);
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin multiply
t0 = 2
t1 = t0 * a
func end

func begin main
t2 = 5
a = t2
param a
refparam result
call multiply, 1
b = result
t3 = "%d "
param t3
param b
refparam result
call printf, 2
func end

VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
  multiply |          Function |        int |            |            |        int |               1 |          0 | 4
       int |           Keyword |            |            |            |            |               0 |          0 | 4
      main |          Function |        int |            |            |            |               0 |          0 | 9
    printf |        Identifier |        int |            |            |            |               0 |          1 | 13
         a |        Identifier |        int |            |            |            |               0 |          1 | 4
         b |        Identifier |        int |            |            |            |               0 |          1 | 12
    return |           Keyword |            |            |            |            |               0 |          1 | 6


                            CONSTANT TABLE
                            --------------
           CONSTANT |                TYPE
                  2 |      Number Constant
                  5 |      Number Constant
               "%d " |      String Constant
```

3.//ERROR LESS

```
#include<stdio.h>
int main() {
        int a[2];
        a[0] = 5;
        a[1] = 2;  char b = 'a';
        if(b == 'a'){
                if(a[0] == 5)
                        printf("Hello 1");
                else    printf("Hello 2");  }
                else   printf("Not Hello"); }
```

OUTPUT:

```
func begin main
t0 = 4 * 0
t1 = &arr + t0
t2 = 5
*t1 = t2
t3 = 4 * 1
t4 = &arr + t3
t5 = 2
*t4 = t5
t6 = 'a'
b = t6
t7 = 'a'
t8 = b == t7
IF not t8 goto L0
t9 = 4 * 0
t10 = &arr + t9
t11 = 5
t12 = t10 == t11
IF not t12 goto L1
t13 = "Hello 1"
param t13
refparam result
call printf, 1
goto L2
L1:
t14 = "Hello 2"
param t14
refparam result
call printf, 1
L2:
goto L3
L0:
t15 = "Not Hello"
param t15
refparam result
call printf, 1
L3:
func end

VALID PARSE
                        SYMBOL TABLE
                        ------------
SYMBOL |            CLASS |      TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
  else |          Keyword |           |            |            |            |               0 |          2 | 14
   int |          Keyword |           |            |            |            |               0 |          0 | 4
  char |          Keyword |           |            |            |            |               0 |          1 | 9
  main |         Function |       int |            |            |            |               0 |          0 | 4
printf |       Identifier |      char |            |            |            |               0 |          2 | 13
    if |          Keyword |           |            |            |            |               0 |          1 | 11
     a |  Array Identifier |       int |            |          2 |            |               0 |          1 | 6
     b |       Identifier |      char |            |            |            |               0 |          1 | 9


                        CONSTANT TABLE
                        --------------
      CONSTANT |              TYPE
  "Not Hello" |     String Constant
             0 |     Number Constant
             1 |     Number Constant
             2 |     Number Constant
             5 |     Number Constant
           'a' |  Character Constant
       "Hello 1" |     String Constant
       "Hello 2" |     String Constant
```

4. //ERROR FREE

```
#include<stdio.h>
int main()
{
	int num = 3;
	for(int i = 0; i < num; i++)
		printf("Hello");

	while(num > 0){
		printf("Hello");
		Num--;
	}
}
```

OUTPUT:

```
func begin main
t0 = 3
num = t0
t1 = 0
i = t1
L0:
t2 = i < num
IF not t2 goto L1
t3 = i + 1
i = t3
t4 = "Hello"
param t4
refparam result
call printf, 1
goto L0:
L1:
L2:
t5 = 0
t6 = num > t5
IF not t6 goto L3
t7 = "Hello"
param t7
refparam result
call printf, 1
t8 = num - 1
num = t8
goto L2:
L3:
func end

VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |           Keyword |            |            |            |            |               0 |          0 | 4
      main |          Function |        int |            |            |            |               0 |          0 | 4
    printf |        Identifier |        int |            |            |            |               0 |          1 | 9
         i |        Identifier |        int |            |            |            |               0 |          1 | 8
       num |        Identifier |        int |            |            |            |               0 |          1 | 6
     while |           Keyword |            |            |            |            |               0 |          1 | 11
       for |           Keyword |            |            |            |            |               0 |          1 | 8


                            CONSTANT TABLE
                            --------------
          CONSTANT |                TYPE
           "Hello" |      String Constant
                 0 |      Number Constant
                 3 |      Number Constant
```

5. #include<stdio.h>
int main() {
	int num = 3;
	for(int i = 0; i<num; i++) {
		for(int j = 0; j < num; j++)
			printf("Hello"); } }

OUTPUT:

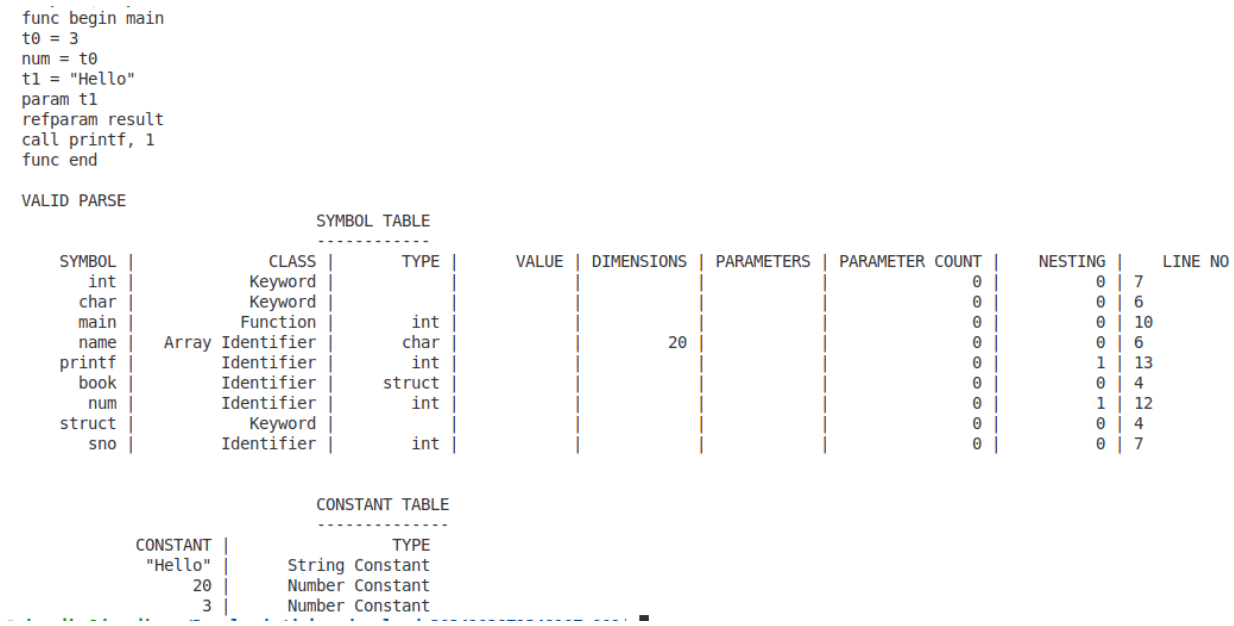
```
^[[Adeepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
t0 = 3
num = t0
t1 = 0
i = t1
L0:
t2 = i < num
IF not t2 goto L1
t3 = i + 1
i = t3
t4 = 0
j = t4
L2:
t5 = j < num
IF not t5 goto L3
t6 = j + 1
j = t6
t7 = "Hello"
param t7
refparam result
call printf, 1
goto L2:
L3:
goto L0:
L1:
func end

VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |           Keyword |            |            |            |            |               0 |          0 | 4
      main |          Function |        int |            |            |            |               0 |          0 | 4
    printf |        Identifier |        int |            |            |            |               0 |          2 | 11
         i |        Identifier |        int |            |            |            |               0 |          1 | 8
         j |        Identifier |        int |            |            |            |               0 |          2 | 10
       num |        Identifier |        int |            |            |            |               0 |          1 | 6
       for |           Keyword |            |            |            |            |               0 |          1 | 8


                            CONSTANT TABLE
                            --------------
          CONSTANT |               TYPE
           "Hello" |      String Constant
                 0 |      Number Constant
                 3 |      Number Constant
```

6. //ERROR FREE - This test case includes declaration of a structure

#include<stdio.h>
struct book {
	char name[20];
	int sno;
};

```
int main() {
        int num = 3;
        printf("Hello");
}
```

OUTPUT:



```
func begin main
t0 = 3
num = t0
t1 = "Hello"
param t1
refparam result
call printf, 1
func end

VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |           Keyword |            |            |            |            |               0 |          0 | 7
      char |           Keyword |            |            |            |            |               0 |          0 | 6
      main |          Function |        int |            |            |            |               0 |          0 | 10
      name |   Array Identifier |       char |            |         20 |            |               0 |          0 | 6
    printf |         Identifier |        int |            |            |            |               0 |          1 | 13
      book |         Identifier |     struct |            |            |            |               0 |          0 | 4
       num |         Identifier |        int |            |            |            |               0 |          1 | 12
    struct |           Keyword |            |            |            |            |               0 |          0 | 4
       sno |         Identifier |        int |            |            |            |               0 |          0 | 7


                            CONSTANT TABLE
                            --------------
          CONSTANT |                TYPE
           "Hello" |      String Constant
                20 |      Number Constant
                 3 |      Number Constant
```

7. //ERROR FREE - This test case includes escape sequences

```
#include<stdio.h>
int main()
{
        char es = '\a';
        printf("Hello");
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
t0 = '\a'
es = t0
t1 = "Hello"
param t1
refparam result
call printf, 1
func end

VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |           Keyword |            |            |            |            |               0 |          0 | 4
      char |           Keyword |            |            |            |            |               0 |          1 | 6
      main |          Function |        int |            |            |            |               0 |          0 | 4
    printf |        Identifier |       char |            |            |            |               0 |          1 | 7
        es |        Identifier |       char |            |            |            |               0 |          1 | 6


                            CONSTANT TABLE
                            --------------
           CONSTANT |                TYPE
            "Hello" |      String Constant
               '\a' |   Character Constant
```

8. //ERROR FREE - This test case includes nested comments

```
#include<stdio.h>
int main()
{
        /*This is /* nested comment */!!*/
        /*This is a
        normal comment*/
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
func end

VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |           Keyword |            |            |            |            |               0 |          0 | 4
      main |          Function |        int |            |            |            |               0 |          0 | 4


                            CONSTANT TABLE
                            --------------
           CONSTANT |                TYPE
```

9. //ERROR FREE: This testcase contains a program that evaluates expressions

```
#include<stdio.h>
void main()
{
```

```
        int a = 0;
        int b = 2;
        int c = 5;
        int d = 10;
        int result;
        result = a + b*(c + d);
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
t0 = 0
a = t0
t1 = 2
b = t1
t2 = 5
c = t2
t3 = 10
d = t3
t4 = c + d
t5 = b * t4
t6 = a + t5
result = t6
func end

VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |             CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |           Keyword |            |            |            |            |               0 |          1 | 5
      main |          Function |       void |            |            |            |               0 |          0 | 3
         a |        Identifier |        int |            |            |            |               0 |          1 | 5
         b |        Identifier |        int |            |            |            |               0 |          1 | 6
         c |        Identifier |        int |            |            |            |               0 |          1 | 7
         d |        Identifier |        int |            |            |            |               0 |          1 | 8
    result |        Identifier |        int |            |            |            |               0 |          1 | 9
      void |           Keyword |            |            |            |            |               0 |          0 | 3


                            CONSTANT TABLE
                            --------------
          CONSTANT |               TYPE
                10 |      Number Constant
                 0 |      Number Constant
                 2 |      Number Constant
                 5 |      Number Constant
```

10. //ERROR FREE: This testcase includes multiple functions

```
#include<stdio.h>
int add(int a, int b, int c)
{
        int res;
        res = a + b + c;
        return res;
}
int multiply(int a, int b, int c)
{
        int res;
```

```
        res = a * b * c;
        return res;
}
void main()
{
        int a = 2, b = 3, c = 4;
        int sum = add(a,b,c);
        int product = multiply(a,b,c);
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin add
t0 = a + b
t1 = t0 + c
res = t1
func end

func begin multiply
t2 = a * b
t3 = t2 * c
res = t3
func end

func begin main
t4 = 2
a = t4
t5 = 3
b = t5
t6 = 4
c = t6
param a
param b
param c
refparam result
call add, 3
sum = result
param a
param b
param c
refparam result
call multiply, 3
product = result
func end

VALID PARSE
                        SYMBOL TABLE
                        ------------
    SYMBOL |            CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
  multiply |          Function |        int |            |            |  int int int |               3 |          0 | 9
       add |          Function |        int |            |            |  int int int |               3 |          0 | 3
       int |           Keyword |            |            |            |              |               0 |          0 | 3
       sum |        Identifier |        int |            |            |              |               0 |          1 | 18
      main |          Function |       void |            |            |              |               0 |          0 | 15
   product |        Identifier |        int |            |            |              |               0 |          1 | 19
       res |        Identifier |        int |            |            |              |               0 |          1 | 5
         a |        Identifier |        int |            |            |              |               0 |          1 | 3
         b |        Identifier |        int |            |            |              |               0 |          1 | 3
         c |        Identifier |        int |            |            |              |               0 |          1 | 3
      void |           Keyword |            |            |            |              |               0 |          0 | 15
    return |           Keyword |            |            |            |              |               0 |          1 | 7


                        CONSTANT TABLE
                        --------------
          CONSTANT |               TYPE
                 2 |      Number Constant
                 3 |      Number Constant
                 4 |      Number Constant
```

11. //ERROR FREE: This testcase contains a program tp swap 2 numbers

```
#include<stdio.h>
void main()
{
        int a = 10, b = 20;
```

```
        printf("\na = %d, b = %d", a,b);
        int temp = a;
        a = b;
        b = temp;
        printf("\na = %d, b = %d", a,b);
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
t0 = 10
a = t0
t1 = 20
b = t1
t2 = "\na = %d, b = %d"
param t2
param a
param b
refparam result
call printf, 3
temp = a
a = b
b = temp
t3 = "\na = %d, b = %d"
param t3
param a
param b
refparam result
call printf, 3
func end

VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |              CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
       int |            Keyword |            |            |            |            |               0 |          1 | 5
      main |           Function |       void |            |            |            |               0 |          0 | 3
      temp |         Identifier |        int |            |            |            |               0 |          1 | 7
    printf |         Identifier |        int |            |            |            |               0 |          1 | 6
         a |         Identifier |        int |            |            |            |               0 |          1 | 5
         b |         Identifier |        int |            |            |            |               0 |          1 | 5
      void |            Keyword |            |            |            |            |               0 |          0 | 3


                            CONSTANT TABLE
                            --------------
            CONSTANT |                TYPE
  "\na = %d, b = %d" |      String Constant
                  10 |      Number Constant
                  20 |      Number Constant
```

12. //ERROR FREE: This testcase includes a program to find factorial of a number

```
#include <stdio.h>
int main()
{
        int n = 7, i;
        int fact = 1;
        if (n < 0)
                printf("\nError!");
        Else
```

```
	{
		for (i = 1; i <= n; i++)
		{
			fact = fact * i;
		}
		printf("Factorial of %d = %d", n, fact);
	}
	return 0;
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
t0 = 7
n = t0
t1 = 1
fact = t1
t2 = 0
t3 = n < t2
IF not t3 goto L0
t4 = "\nError!"
param t4
refparam result
call printf, 1
goto L1
L0:
t5 = 1
i = t5
L2:
L3:
t6 = i <= n
 AT FORRR 1
IF not t6 goto L4
t7 = i + 1
i = t7
t8 = fact * i
fact = t8
goto L3:
L4:
t9 = "Factorial of %d = %d"
param t9
param n
param fact
refparam result
call printf, 3
L2:
t10 = 0
func end

VALID PARSE
                            SYMBOL TABLE
                            ------------
    SYMBOL |              CLASS |       TYPE |      VALUE | DIMENSIONS | PARAMETERS | PARAMETER COUNT |    NESTING |    LINE NO
      else |            Keyword |            |            |            |            |               0 |          1 | 9
       int |            Keyword |            |            |            |            |               0 |          0 | 3
      main |           Function |        int |            |            |            |               0 |          0 | 3
    printf |         Identifier |        int |            |            |            |               0 |          1 | 8
        if |            Keyword |            |            |            |            |               0 |          1 | 7
      fact |         Identifier |        int |            |            |            |               0 |          1 | 6
         i |         Identifier |        int |            |            |            |               0 |          1 | 5
         n |         Identifier |        int |            |            |            |               0 |          1 | 5
       for |            Keyword |            |            |            |            |               0 |          2 | 11
    return |            Keyword |            |            |            |            |               0 |          1 | 18


                            CONSTANT TABLE
                            --------------
          CONSTANT |                TYPE
         "\nError!" |      String Constant
"Factorial of %d = %d" |      String Constant
                   0 |      Number Constant
                   1 |      Number Constant
                   7 |      Number Constant
```

TEST CASES WITH ERRORS

13. //WITH ERROR - This test case includes duplicate declaration of identifier

```
#include<stdio.h>
void main()
{
        int a = 1;
```

```
        int a = 2;
        printf("%d", a);
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
t0 = 1
a = t0
Line No. : 7 Duplicate value!
 =

UNSUCCESSFUL: INVALID PARSE
t1 = 2
a = t1
t2 = "%d"
param t2
param a
refparam result
call printf, 2
func end
```

14. //WITH ERROR - This test case includes array size less than 1

```
#include<stdio.h>
void main()
{
        int a[0];
        printf("hello\n");
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
Line No. : 6 Array must have size greater than 1!
 ;

UNSUCCESSFUL: INVALID PARSE
t0 = "hello\n"
param t0
refparam result
call printf, 1
func end
```

15. //WITH ERROR - This test case includes duplicate function declaration

```
#include<stdio.h>
void func()
{
        printf("hello\n");
}
void func()
```

```
{
	printf("hello\n");
}
void main()
{
	printf("hello\n");
}
```

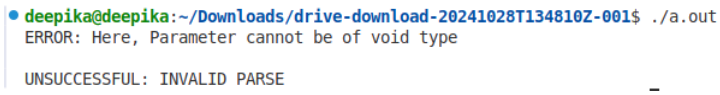
OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
 func begin func
 t0 = "hello\n"
 param t0
 refparam result
 call printf, 1
 func end

 Line No. : 7  ERROR:Function already defined
  (

 UNSUCCESSFUL: INVALID PARSE
 func begin func
 t1 = "hello\n"
 param t1
 refparam result
 call printf, 1
 func end

 Line No. : 11  ERROR:Function already defined
  (

 UNSUCCESSFUL: INVALID PARSE
 func begin main
 t2 = "hello\n"
 param t2
 refparam result
 call printf, 1
 func end
```

16. //WITH ERROR - This test case includes void parameter for function

```
#include<stdio.h>
void func(void x)
{
	printf("hello\n");
}
void main()
{
	printf("hello\n");
}
```

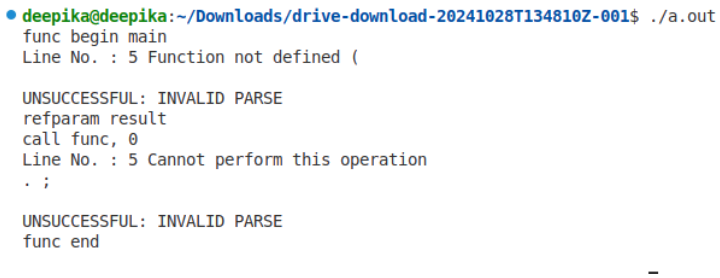
OUTPUT:



17. //WITH ERROR - This test case includes a function call to undeclared function

```
#include<stdio.h>
void main()
{
        func();
}
```

OUTPUT:



18.//WITH ERROR - This test case includes a function call to function declared after main

```
#include<stdio.h>
void main()
{
        func();
}
void func()
{
        printf("hello\n");
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
Line No. : 5 Function not defined (

UNSUCCESSFUL: INVALID PARSE
refparam result
call func, 0
Line No. : 5 Cannot perform this operation
. ;

UNSUCCESSFUL: INVALID PARSE
func end

Line No. : 7  ERROR:Function already defined
 (

UNSUCCESSFUL: INVALID PARSE
func begin func
t0 = "hello\n"
param t0
refparam result
call printf, 1
func end
```

19. //WITH ERROR - This test case includes void return for int function

```
#include<stdio.h>
int func()
{
        printf("hello\n");
        return;
}
void main()
{
        func();
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin func
t0 = "hello\n"
param t0
refparam result
call printf, 1
Line No. : 6 ERROR: Cannot have void return for non-void function!
 ;

UNSUCCESSFUL: INVALID PARSE
func end

Line No. : 8  ERROR:Function already defined
 (

UNSUCCESSFUL: INVALID PARSE
func begin main
refparam result
call func, 0
func end
```

20. //WITH ERROR - This test case includes int return for void function

```
#include<stdio.h>
void func()
{
	printf("hello\n");
	return 0;
}
void main()
{
	func();
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin func
t0 = "hello\n"
param t0
refparam result
call printf, 1
t1 = 0
Line No. : 6 Non-void return for void function! ;

UNSUCCESSFUL: INVALID PARSE
func end

Line No. : 8  ERROR:Function already defined
 (

UNSUCCESSFUL: INVALID PARSE
func begin main
refparam result
call func, 0
Line No. : 10 Cannot perform this operation
. ;

UNSUCCESSFUL: INVALID PARSE
func end
```

21. //WITH ERROR - This test case includes function call with incorrect number of parameters during call

```
#include<stdio.h>
int func(int a, int b)
{
        return a+b;
}
void main()
{
        int a = 1;
        int x;
        x = func(a);
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin func
t0 = a + b
Line No. : 5 Expression doesn't match return type of function
 ;

UNSUCCESSFUL: INVALID PARSE
func end

Line No. : 7  ERROR:Function already defined
 (

UNSUCCESSFUL: INVALID PARSE
func begin main
t1 = 1
a = t1
param a
Line No. : 11 Number of parameters not same as number of arguments during function call!

UNSUCCESSFUL: INVALID PARSE
refparam result
call func, 2
x = result
func end
```

22. //WITH ERROR - This test case includes if condition not of type int

```
#include<stdio.h>
void main()
{
        float x = 1.0;
        if(x)
                print("hello\n");
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
t0 = 1.0
x = t0
IF not x goto L0
Line No. : 6 ERROR: Condition must have integer value!
 )

UNSUCCESSFUL: INVALID PARSE
Line No. : 7 Function not defined (

UNSUCCESSFUL: INVALID PARSE
t1 = "hello\n"
param t1
Line No. : 7 Number of parameters not same as number of arguments during function call! )

UNSUCCESSFUL: INVALID PARSE
Line No. : 7 Type mismatch in function call: expected error but got char* for argument 1 )

UNSUCCESSFUL: INVALID PARSE
refparam result
call print, 0
Line No. : 7 Cannot perform this operation
. ;

UNSUCCESSFUL: INVALID PARSE
goto L1
L0:
L1:
func end
```

23. //WITH ERROR - This test case includes case where array identifier has no subscript

```
#include<stdio.h>
void main()
{
        int ar[2] = {1,2};
        ar = 3;
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
t0 = 1
t1 = 2
t2 = 3
Line No. : 6 Type Mismatch
 ;

UNSUCCESSFUL: INVALID PARSE
ar = t2
Line No. : 6 Cannot perform this operation
. ;

UNSUCCESSFUL: INVALID PARSE
func end
```

24. //WITH ERROR - This test case includes case value in subscript not integer

```
#include<stdio.h>
void main()
{
	int ar[2] = {1, 2};
	float y = 1.0;
	ar[y] = 1;

}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
t0 = 1
t1 = 2
t2 = 1.0
y = t2
Line No. : 6 ERROR: Type Mismatch
 ;

UNSUCCESSFUL: INVALID PARSE
```

25. //WITH ERROR - This test case includes case where lhs of assignment has more than 1 single variable

```
#include<stdio.h>
void main()
{
	int x = 1;
	int y = 1;
	int z = 1;
	x + y = z;

}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
t0 = 1
x = t0
t1 = 1
y = t1
t2 = 1
z = t2
t3 = x + y
Line No. : 8 Line no. Error: Invalid lhs value
 =

UNSUCCESSFUL: INVALID PARSE
```

26. //WITH ERROR- This test case includes use of out of scope id

```
#include <stdio.h>
void main()
{
        int x = 1;
        if(1)
        {
                int y = 2;
        }
        y = 3;
}
```

OUTPUT:

```
deepika@deepika:~/Downloads/drive-download-20241028T134810Z-001$ ./a.out
func begin main
t0 = 1
x = t0
t1 = 1
IF not t1 goto L0
t2 = 2
y = t2
goto L1
L0:
L1:
Line No. : 10 Identifier undeclared or out of scope =

UNSUCCESSFUL: INVALID PARSE
t3 = 3
y = t3
func end
```

Future work

The yacc script presented in this report takes care of all the rules of C language, but is not fully exhaustive in nature. Our future work would include making the script even more robust in order to handle all aspects of C language and making it more efficient.

Results

The parser was able to be successfully parse the tokens recognized by the flex script. The program gives us the list of the symbols and constants. The program also detects syntactical errors and semantics errors if found. Thus the semantic stage is an essential part of the compiler and is needed for the simplifications of the compiler. It makes the compiler more efficient and robust.