COMPILER FOR C LANGUAGE

2020380029 RAIHAN MD RAKIBUL ISLAM
2020380043 SHOAIB MUHAMMAD
Bachelor of Engineering in
Computer Science and Technology
Northwestern Polytechnical University

Dr. 林奕
Computer Science and Technology
Northwestern Polytechnical University

Abstract:
Our aim is to design a compiler for subset of C Programming Language. We are trying to implement the following features

Modules:
1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator

1. Lexical Analyzer:

- Identification of Keywords, Identifiers, Operators (Relational, Logical and Arithmetic), Punctuators, Constants (Integer, Character and Float) and String Literals with invalid string error handling
- Single and Multi-Line Comments with error handling.
- Data Types (int, float and char) with modifiers (unsigned and signed) and types (short, long).
- Procedures with return type int, float and char.
- Parenthesis matching with error reporting.
- Invalid character identification and error reporting.

2. Syntax Analyzer:

In this phase syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Checks the expression from this token are syntactically correct. We are trying to implement following features.

- Syntax Checking for arithmetic, logical and relational expressions.
- Productions rules for control statements such as if, if else and nested if statements.
- Production rules for unary plus and minus.
- Production rule and syntax checking for ‘for’ loop and nested for loop.
- Syntax checking for function declaration and parameter passing.
- Array indexing syntax errors.
- Scanf and printf syntax errors.
- Missing semicolon and unbalanced parenthesis.

COMPILER FOR C LANGUAGE

3. Semantic Analyzer:

In this phase, we extract necessary semantic information from the source code which is impossible to detect in parsing. We are trying to implement following features

- Scope of the identifiers declared
- Duplicate declaration of identifiers
- Function declaration and its scope
- Actual and formal parameter matching (number and type of parameters)
- Matching function return type
- Checking array indexing with in the given bound
- Invalid array indexing while declaration (array limit less than one)
- Type mismatch of variables

4. Intermediate Code Generator:

In this phase, we are trying to generate language independent three-address code for given source program which is lexically, syntactically and semantically correct.

It receives input in the form of annotated syntax tree That syntax tree then can be converted into a linear representation, (postfix notation). Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code. This three-address code can be converted to MIPS assembly code .

SCANNER FOR C

Introduction:

A compiler is a special program that processes statements written in a particular programming language and turns them into machine language or "code" that a computer's processor uses. A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes. When executing (running), the compiler first parses (or analyzes) all of the language statements syntactically one after the other and then, in one or more successive stages or "passes", builds the output code, making sure that statements that refer to other statements are referred to correctly in the final code.

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. We basically have two phases of compilers, namely Analysis phase and Synthesis phase. Analysis phase creates an intermediate representation from the given source code. It is also termed as front end of the compiler. Analysis phase consists of:

1. Lexical Analysis:
    a. The lexical phase reads the characters in the source program and groups them into a stream of tokens in which each token represents a logically cohesive sequence of characters, such as, an identifier, a keyword, a punctuation character.
    b. The character sequence forming a token is called the lexeme for the token.

COMPILER FOR C LANGUAGE

2. Syntax Analysis :

a. Syntax analysis imposes a hierarchical structure on the token stream. This hierarchical structure is called syntax tree.
b. A syntax tree has an interior node is a record with a field for the operator and two fields containing pointers to the records for the left and right children.
c. A leaf is a record with two or more fields, one to identify the token at the leaf, and the other to record information about the token.

3. Semantic Analysis:

a. This phase checks the source program for semantic errors and gathers type information for the subsequent code-generation phase.
b. It uses the hierarchical structure determined by the syntax-analysis phase to identify the operators and operands of expressions and statements.
c. An important component of semantic analysis is type checking.

4. Intermediate Code Generation :

The syntax and semantic analysis generate a explicit intermediate representation of the source program.
The intermediate representation should have two important properties:

I. It should be easy to produce,
II. And easy to translate into target program.

Intermediate representation can have a variety of forms. One of the forms is: three address code; which is like the assembly language for a machine in which every location can act like a register. Three address code consists of a sequence of instructions, each of which has at most three operands.

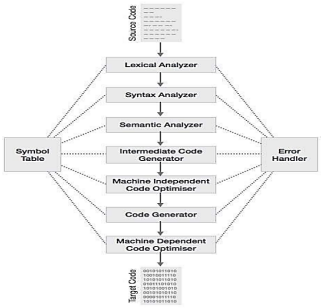
Synthesis phase creates an equivalent target program from intermediate representation. It is the back end of the compiler. Synthesis phase consists of:

1. Code Optimization :

Code optimization phase attempts to improve the intermediate code, so that faster-running machine code will result.

2. Code Generation:

a. The final phase of the compiler is the generation of target code, consisting normally of relocatable machine code or assembly code.
b. Memory locations are selected for each of the variables used by the program.
c. Then, each intermediate instruction is translated into a sequence of machine instructions that perform the same task. The following diagram shows the various phases of a compiler.



COMPILER FOR C LANGUAGE

Lexical Analysis:

The word "lexical" in the traditional sense means "pertaining to words". In terms of programming languages, words are objects like variable names, numbers, keywords etc. Such words are traditionally called tokens.

Lexical analysis is the first phase of compiler which is also termed as scanning. A token is a sequence of characters that represent lexical unit, which matches with the pattern, such as keywords, operators, identifiers etc. Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called lexemes which produces token as output.

A lexical analyzer, or lexer for short, will as its input take a string of individual letters and divide this string into tokens. Additionally, it will filter out whatever separates the tokens (the so-called white-space), i.e., lay-out characters (spaces, newlines, etc.) and comments.

Tokens, Patterns and Lexemes Token:

It is a valid sequence of characters which are given by lexeme. In a programming language, keywords, constant, identifiers, numbers, operators and punctuations symbols are possible tokens to be identified. Example of tokens:

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

Pattern: A pattern describes a rule that must be matched by sequence of characters (lexemes) to form a token. It can be defined by regular expressions or grammar rules.
For example - [A-Za-z][A-Za-z_0-9]*
Lexeme: A lexeme is a sequence of characters that matches the pattern for a token i.e., instance of a token. Eg: c=a+b*5;

The sequence of tokens produced by lexical analyzer helps the parser in analyzing the syntax of programming languages.

Role of a Lexical Analyzer:

A lexical analyzer performs the following tasks:

- Reads the source program, scans the input characters, group them into lexemes and produce the token as output.
- Enters the identified token into the symbol table.
- Strips out white spaces and comments from source program.
- Correlates error messages with the source program i.e., displays error message with its occurrence by specifying the line number.
- Expands the macros if it is found in the source program.

Need of Lexical Analyzer:

- Simplicity of design of compiler - The removal of white spaces and comments enables the syntax analyzer for efficient syntactic constructs.

COMPILER FOR C LANGUAGE

- Compiler efficiency is improved - Specialized buffering techniques for reading characters speed up the compiler process.
- Compiler portability is enhanced

Lexical Errors:

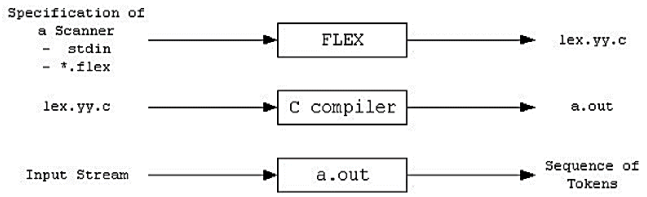
A character sequence that cannot be scanned into any valid token is a lexical error. Lexical errors are uncommon, but they still must be handled by a scanner. Usually, a lexical error is caused by the appearance of some illegal character, mostly at the beginning of a token.

Lexical Analyzer also generates errors in the following cases:
1) Unterminated String: When the right number of inverted commas are not provided.
2) Nested Comments: Nested comments are not supported.
3) Unmatched Parenthesis: If there are missing parenthesis, an error message is generated.
4) Invalid Identifier: If the entered identifier does not match the identifier forming rules, an error message is displayed.

FLEX:

The scanner performs lexical analysis of a certain program. It reads the source program as a sequence of characters and recognizes "larger" textual units called tokens.

FLEX stands for Fast Lexical Analyzer Generator. It is a tool for generating scanners. Instead of writing a scanner from scratch, you only need to identify the vocabulary of a certain language, write a specification of patterns using regular expressions (e.g. DIGIT [0-9]), and FLEX will construct a scanner for you. FLEX is generally used in the manner depicted here:



First, FLEX reads a specification of a scanner either from an input file *.lex, or from standard input, and it generates as output a C source file lex.yy.c. Then, lex.yy.c is compiled and linked with the "-lfl" library to produce an executable a.out. Finally, a.out analyzes its input stream and transforms it into a sequence of tokens.

- *.lex is in the form of pairs of regular expressions and C code.
- lex.yy.c defines a routine yylex() that uses the specification to recognize tokens.
- a.out is actually the scanner.

These programs perform character parsing and tokenizing via the use of a deterministic finite automaton (DFA). A DFA is a theoretical machine accepting regular languages. These machines are a subset of the collection of Turing machines. DFAs are equivalent to read-only right moving Turing machines. The syntax is based on the use of regular expressions.

COMPILER FOR C LANGUAGE

Format Of the Flex File:

The flex input file consists of three sections, separated by a line with just `%%' in it: definitions %% rules %% user code The definitions section contains declarations of simple name definitions to simplify the scanner specification, and declarations of start conditions, which are explained in a later section. Name definitions have the form:

name definition

The "name" is a word beginning with a letter or an underscore ('_') followed by zero or more letters, digits, '_', or '-' (dash). The definition is taken to begin at the first non-whitespace character following the name and continuing to the end of the line. The definition can subsequently be referred to using "{name}", which will expand to "(definition)". For example,

DIGIT [0-9]
*ID [a-z][a-z0-9]**

defines "DIGIT" to be a regular expression which matches a single digit, and "ID" to be a regular expression which matches a letter followed by zero-or-more letters-or-digits. A subsequent reference to

*{DIGIT}+"."{DIGIT}**
is identical to
*([0-9])+"."([0-9])**

and matches one-or-more digits followed by a '.' followed by zero-or-more digits.

The rules section of the flex input contains a series of rules of the form:

pattern action

where the pattern must be unintended and the action must begin on the same line.

Finally, the user code section is simply copied to `lex.yy.c' verbatim. It is used for companion routines which call or are called by the scanner. The presence of this section is optional; if it is missing, the second `%%' in the input file may be skipped, too.

In the definitions and rules sections, any indented text or text enclosed in `%{' and `%}' is copied verbatim to the output (with the `%{}"s removed). The `%{}"s must appear unintended on lines by themselves. In the rules section, any indented or %{} text appearing before the first rule may be used to declare variables which are local to the scanning routine and (after the declarations) code which is to be executed whenever the scanning routine is entered. Other indented or %{} text in the rule section is still copied to the output, but its meaning is not well-defined and it may well cause compile-time errors. In the definitions section (but not in the rules section), an unintended comment (i.e., a line beginning with "/*") is also copied verbatim to the output up to the next "*/".

Implementation:

scanner.l:

This is the lex program that contains various regular expressions for all the specific actions that are to be carried out by a lexical analyzer. This file is converted to lex.yy.c which is compiled to get the executable a.out

COMPILER FOR C LANGUAGE

Code:

```
scanner.l
%{
#include <stdio.h>
#include <stdlib.h>
#include <malloc.h>
#include <string.h>
int var=0,i,nc=0,cLine=0,flag=0;
int lineNumber=1;
int cBrac=0;

FILE *symbol,*constants;
char *comment,*inputFile, s_comment[1000];

void insertToTable(char *yytext,char type);
void displayComment(char *yytext);
void storeSingleLineComment(char *yytext);

struct Node {
    char *tname;
    int av;
    struct Node *next;
}*head=NULL;

%}

digit  [0-9]
letter  [a-zA-Z]
keyword "auto"|"break"|"case"|"char"|"const"|"continue"|"default"|"do"|"double"|"else"|"enum"|"extern"|"float"|"for"|"goto"|"if"|"int"|"long"|"register"|"return"
datatype "int"|"char"|"void"
sign "signed"|"unsigned"
modifiers "long"|"short"
relational >|<|<=|>=|!=|==
logical \&|\^|\~
arithmatic \+|\-|\*|\/|\%
puncuator \(|\)|\[|\]|\;|\,|\:|\.
assignment =
quote \'|\"|\\
whitespace [ \t]+
newline "\n"
singlelinecomment (\/\/.*)
multilinecommentstart (\/\*)
multilinecommentend (\*\/)
Line 1, Column 1                                Tab Size: 4        Lex/Flex
```

Fig. 1

```
scanner.l
                current->next = temp;
        }

            var++;
        }
    }

    if(type =='i' || type == 'a' || type == 'q' || type=='j')
    {
        fprintf(symbol,"\n%20s%30s%30d%35d",yytext,token,i,lineNumber);
    }
    switch(type)
    {
        case 'c' : fprintf(constants,"\n%20s%20s%30d%35d",yytext,"int",i,lineNumber);
                    break;

        case 'f' : fprintf(constants,"\n%20s%20s%30d%35d",yytext,"float",i,lineNumber);
                    break;

        case 'z' : fprintf(constants,"\n%20s%20s%30d%35d",yytext,"char",i,lineNumber);
                    break;
    }

    fprintf(yyout,"\n%20s%30s%30d%35d",yytext,token,i,lineNumber);
}

int yywrap()
{
 return(1);
}
Line 296, Column 2                              Tab Size: 4        Lex/Flex
```

Fig. 2

COMPILER FOR C LANGUAGE

Input.c

This is the C source program for which the lexical analysis is done. Based on this, we generate the Symbol and Constants Table.

Code:

```
input.c
C: > Users > Raihan > Desktop > Lexical Analyzer > src > input.c
#include <stdio.h>
int main()
{
  /* My name is Raihan.
  This is
  a sample
  multi-line
  comment*/
  struct node{
  int a;
  char name;
  };

  float f1 = 5.01;
  char l1 = 'a', l2 ='1';
  int _a=0,b=0,c=5;
  int arr[50];
  char *ptr;
  scanf("%d %d",&a,&b);
  int sum=0;
  sum=a+b;
  printf("\n Sum : %d \n",sum);

  //Enter Name
  char *name = "Raihan";

  //End the main function
  return 1;
}
void abc()
{
  printf("\nTest function ");
}
```

Execution Of Code:

The lex code can be executed by the following commands :

- flex<filename1>
- cc lex.yy.c
- ./a.out <filenmae2>

Where <filename1> is the lex program (here scanner.l)

<filenmae2>is the C source program for which Lexical Analysis is done. (here input.c)

Output:

symbolTable.txt :

```
symbolTable.txt
C: > Users > Raihan > Desktop > Lexical Analyzer > src > symbolTable.txt

Symbol Table:
   Lexeme        Type       Attribute Value     Line Number

              main                Procedure                    2                    2
              node               Identifier                    5                    9
                 a               Identifier                    6                   10
              name               Identifier                    9                   11
                fl               Identifier                   12                   14
                l1               Identifier                   15                   15
                l2               Identifier                   17                   15
                _a               Identifier                   19                   16
                 b               Identifier                   21                   16
                 c               Identifier                   22                   16
               arr                    Array                   24                   17
               ptr                  Pointer                   25                   18
             scanf               Identifier                   26                   19
                 a               Identifier                    6                   19
                 b               Identifier                   21                   19
               sum               Identifier                   31                   20
               sum               Identifier                   31                   21
                 a               Identifier                    6                   21
                 b               Identifier                   21                   21
            printf               Identifier                   33                   22
               sum               Identifier                   31                   22
              name                  Pointer                    9                   25
               abc                Procedure                   38                   30
            printf               Identifier                   33                   32
```

COMPILER FOR C LANGUAGE

constantTable.txt

constantTable.txt

C: > Users > Raihan > Desktop > Lexical Analyzer > src > constantTable.txt

Constants Table:

Lexeme	Type	Attribute Value	Line Number	
5.01	float	14	14	
a	char	6	15	
1	char	18	15	
0	int	20	16	
0	int	20	16	
5	int	23	16	
0	int	20	20	
1	int	18	28	

parsedTable.txt

C: > Users > Raihan > Desktop > Lexical Analyzer > src > parsedTable.txt

Table:

Lexeme	Token	Attribute Value	Line Number	
#include <stdio.h>	Preprocessor Statement	0	1	
int	Keyword	1	2	
main	Procedure	2	2	
(	Punctuator	3	3	
struct	Keyword	4	9	
node	Identifier	5	9	
(	Punctuator	3	9	
int	Keyword	1	10	
a	Identifier	6	10	
;	Punctuator	7	10	
char	Keyword	8	11	
name	Identifier	9	11	
;	Punctuator	7	11	
}	Punctuator	10	12	
;	Punctuator	7	12	
float	Keyword	11	14	
f1	Identifier	12	14	
=	Assignment Op	13	14	
5.01	Float Constant	14	14	
;	Punctuator	7	14	
char	Keyword	8	15	
l1	Identifier	15	15	
=	Assignment Op	13	15	
a	Character Constant	6	15	
,	Punctuator	16	15	
l2	Identifier	17	15	
=	Assignment Op	13	15	
1	Character Constant	18	15	
;	Punctuator	7	15	
int	Keyword	1	16	
a	Identifier	19	16	
=	Assignment Op	13	16	
0	Integer Constant	20	16	
,	Punctuator	16	16	
b	Identifier	21	16	
=	Assignment Op	13	16	
0	Integer Constant	20	16	
,	Punctuator	16	16	
c	Identifier	22	16	
=	Assignment Op	13	16	
5	Integer Constant	23	16	
;	Punctuator	7	16	
int	Keyword	1	17	
arr	Array	24	17	
;	Punctuator	7	17	
char	Keyword	8	18	
ptr	Pointer	25	18	
;	Punctuator	7	18	
scanf	Identifier	26	19	
(	Punctuator	27	19	
"%d %d"	String Literal	28	19	
,	Punctuator	16	19	
&	Logical Op	29	19	
a	Identifier	6	19	
,	Punctuator	16	19	
&	Logical Op	29	19	
b	Identifier	21	19	
)	Punctuator	30	19	
;	Punctuator	7	19	
int	Keyword	1	20	
sum	Identifier	31	20	
=	Assignment Op	13	20	
0	Integer Constant	20	20	
;	Punctuator	7	20	
sum	Identifier	31	21	
=	Assignment Op	13	21	
a	Identifier	6	21	
+	Arithmetic Op	32	21	
b	Identifier	21	21	
;	Punctuator	7	21	
printf	Identifier	33	22	
b	Identifier	21	21	
;	Punctuator	7	21	
printf	Identifier	33	22	
(	Punctuator	27	22	
"\n Sum : %d \n"	String Literal	34	22	
,	Punctuator	16	22	
sum	Identifier	31	22	
)	Punctuator	30	22	
;	Punctuator	7	22	
char	Keyword	8	25	
name	Pointer	9	25	
=	Assignment Op	13	25	
"Arvind"	String Literal	35	25	
;	Punctuator	7	25	
return	Keyword	36	28	
1	Integer Constant	18	28	
;	Punctuator	7	28	
}	Punctuator	10	29	
void	Keyword	37	30	
abc	Procedure	38	30	
(	Punctuator	3	31	
printf	Identifier	33	32	
(	Punctuator	27	32	
"\nTest function "	String Literal	39	32	
)	Punctuator	30	32	
;	Punctuator	7	32	
}	Punctuator	10	33	

MultilineComment (5 lines):
My name is Raihan.
This is
a sample
multi-line
comment

SingleLineComment :
Enter Name
End the main function

COMPILER FOR C LANGUAGE

Test Cases:

Test Case Filename	Test Case Type	Code	Status
Case1.c	Pre-Processor statements	#include<stdio.h> #define count 10	Passed

Case2.c	Constants
	Keywords
	Punctuators
	Variables
		int a=25;
		
		char c='h';
		
		char arr1[10]="hello";
		
		char arr2[10]="hello;
			Passed
			
			Constants:
			
			Integer: 25
			
			Character: 'h'
			
			String: "Hello"
			
			Key Words: int char
			
			Identifiers: a c arr1 arr2

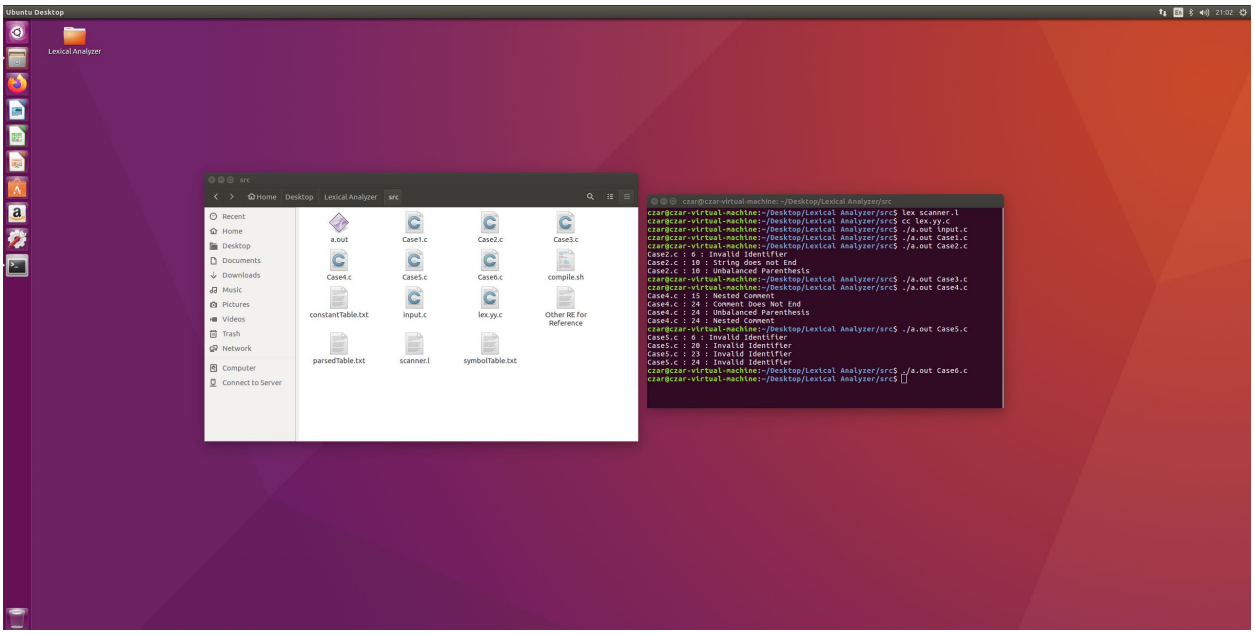
Case3.c	Relational Operators
	
	Logical Operators
	
	Arithmetic Operators
		int a=1,b=2,c=3,d; a++;
		--b; c=a+b;
		d+=a;
		d=a||b;
		c=a%b;
		d=a++c;
		a=c+/c;
		d=a>b;
			Passed
			
			Relational: >
			Logical: | |
			
			Arithmetic: + -
			/ % \

Case4.c	Single Line Comments Multi Line Comments	// single line comment /// this is valid comment /* this is a multi line comment */ /* this is a /* nested */ comment */ invalid comment */ /* invalid comment	Passed Passed Passed Passed Error Passed Error Passed Error

Case5.c	Control Statements Looping Statements	int a=1,b=2,c=10;
		if(a>b){ printf("\nInside if");
		} else{
		printf("\nInside else"); }
		if(b>a){ if(b<c){
		printf("\nNested if "); }
		}
		for(int i=1;i<=10;i++){
		printf("\nIteration %d",i); }
		for(int i=0;i<10;i++){ for(int J=0;J<20;J++){
		printf("\nNestedloop "); }
		}
			Passed
			Keywords If else for
			Datatype Int
			Identifiers J a b c i
			Procedures :
			printf ()

Case6.c	Function Declaration	struct student{ int a; char c;
		};
		union teacher{ int q;
		};
		void abc(){ printf("\nHello World"); }
		void main(){ student *S; teacher *T;
			Passed
			Procedures :
			main() abc() printf()
			Identifiers :
			Student teacher

COMPILER FOR C LANGUAGE



Remarks:

We have made a scanner for C Programming Language. The following features have been taken care of:

1) Identification of Keywords, Identifiers, Operators (Relational, Logical and Arithmetic), Punctuators, Constants (Integer, Character and Float) and String Literals.
2) Arrays
3) Pointers
4) Single and Multi-Line Comments
5) Data Types (int, float and char) with modifiers (unsigned and signed) and types (short, long).
6) Procedures

The following lexical errors have been handled:

1) Nested Comments
2) Invalid Identifiers
3) Invalid String
4) Balancing of Parentheses

PARSER FOR C

Introduction:

When an input string (source code or a program in some language) is given to a compiler, the compiler processes it in several phases, starting from lexical analysis (scans the input and divides it into tokens) to target code generation. The aim of this phase of a compiler is to implement a parser for C language. The lexical analyzer generated in the first phase reads the source program and generates tokens that are given as input to the parser which then creates a syntax tree in accordance with the grammar, consequently leading to the generation of intermediate code that is fed into the synthesis phase, to obtain the correct, equivalent machine level code. We have seen

that a lexical analyzer can identify tokens with the help of regular expressions and pattern rules. But a lexical analyzer cannot check the syntax of a given sentence due to the limitations of the regular expressions. Regular expressions cannot check balancing tokens, such as parenthesis. Therefore, this phase uses context-free grammar (CFG), which is recognized by push-down automata.

Syntax Analysis:

Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. It checks the syntactic structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not. It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the predefined Grammar of the language and the input string. If the given input string can be produced with the help of the syntax tree (in the derivation process), the input string is found to be in the correct syntax. The Grammar for a Language consists of Production rules.

Context-Free Grammar:

A context-free grammar has four components:

● A set of non-terminals (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.

● A set of tokens, known as terminal symbols (Σ). Terminals are the basic symbols from which strings are formed.

● A set of productions (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a non-terminal called the left side of the production, an arrow, and a sequence of tokens and/or on- terminals, called the right side of the production.

● One of the non-terminals is designated as the start symbol (S); from where the production begins.

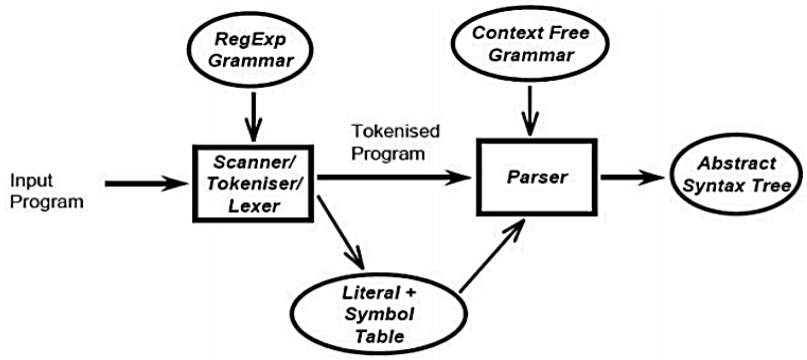


Fig 1: Flowchart of a parser

The strings are derived from the start symbol by repeatedly replacing a non-terminal (initially the start symbol) by the right side of a production, for that non-terminal.

COMPILER FOR C LANGUAGE

Parse Tree:

A parse tree is a graphical depiction of a derivation. It is convenient to see how strings are derived from the start symbol. The start symbol of the derivation becomes the root of the parse tree. Consider the following example with the given Production rules and Input String.

Input string : id + id * id

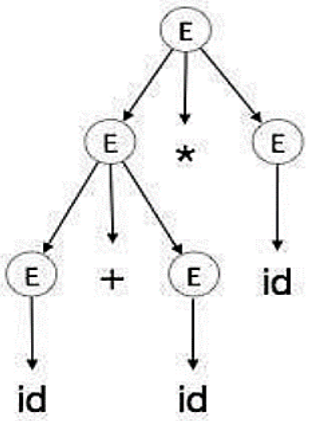
Production rules:

- E → E + E
- E → E * E
- E → id

The left-most derivation is:

- E → E * E
- E → E + E * E
- E → id + E * E
- E → id + id * E
- E → id + id * id

The parse tree for the given input string is:



In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string. A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

Yacc Script:

YACC stands for Yet Another Compiler-Compiler. It is a parser generator for LALR (1) grammars. Given a description of the grammar, it generates a C source for the parser. The input is a file that contains the grammar description with a formalism similar to the BNF (Backus-Naur Form) notation for language specification:

- Non-terminal symbols - lowercase identifiers 4
  - Expr, stmt

COMPILER FOR C LANGUAGE

● Terminal symbols - uppercase identifiers or single characters
○ INTEGER, FLOAT, IF, WHILE, ‘;’, ‘.’
● Grammar rules (Production rules) ○ expr: expr ‘+’ expr | expr ‘*’ expr ; E→E+E|E*E

Yacc provides a general tool for imposing structure on the input to a computer program. The Yacc user prepares a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. Yacc then generates a function to control the input process. This function, called a parser, calls the user-supplied low-level input routine (the lexical analyzer) to pick up the basic items (called tokens) from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, then user code supplied for this rule, an action, is invoked; actions have the ability to return values and make use of the values of other actions.

Basic Specifications:

Names refer to either tokens or nonterminal symbols. Yacc requires token names to be declared as such. Every specification file consists of three sections: the declarations, (grammar) rules, and programs. The sections are separated by double percent ``%%" marks. (The percent ``%" is generally used in Yacc specifications as an escape character.)

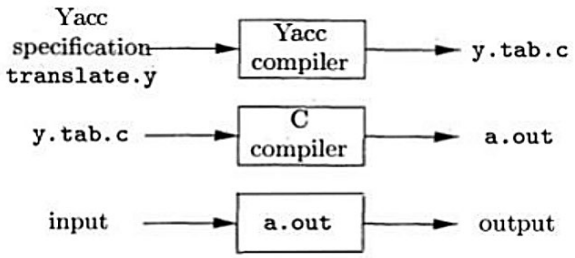


Fig 2: Creating a syntactical analyzer with Yacc

In other words, a full specification file looks like:

declarations
%%
rules
%% programs

The declaration section may be empty. Moreover, if the programs section is omitted, the second %% mark may be omitted also; thus, the smallest legal Yacc specification is

%%
rules

The rules section is made up of one or more grammar rules. A grammar rule has the form:

A: BODY; A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals.

COMPILER FOR C LANGUAGE

Actions:

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. Moreover, the lexical analyzer can return values for tokens, if desired. An action is an arbitrary C statement, and as such can do input and output, call subprograms, and alter external vectors and variables. An action is specified by one or more statements, enclosed in curly braces "{"and "}".

How the Parser Works:

Yacc turns the specification file into a C program, which parses the input according to the specification given. The parser produced by Yacc consists of a finite state machine with a stack. The parser is also capable of reading and remembering the next input token (called the lookahead token). The current state is always the one on the top of the stack. A move of the parser is done as:

1. Based on current state, the parser decides whether it needs a lookahead token to decide what action should be done; if it needs and does not have one, it calls yylex to obtain the next token.
2. Using the current state, and the lookahead token if needed, the parser decides on its next action, and carries it out. This may result in states being pushed onto the stack, or popped off of the stack, and in the lookahead token being processed or left alone.

Source Code:

Lexical Analyzer: This is the lex program that contains regular expressions and returns whatever tokens are required.

Syntax Analyzer: This is the main parser code, a yacc file which contains the declarations, rules and programs and defines the actions which should be taken for various cases.

Execution Of the Code:

The following is a shell script to automate the compilation and execution of the parser.

```
#!/bin/sh
lex lexicalAnalyzer.l
yacc -d syntaxChecker.y
gcc lex.yy.c y.tab.c -w -g
./a.out input.c
rm y.tab.c y.tab.h lex.yy.c
```

Implementation:

The implementation is as follows:

● Firstly, a file is created with a ". y" extension, which represents a yacc file. The file created here is "syntaxChecker.y ".

● The file, like all yacc files, contains three main sections: Declarations, Rules, and Programs.

COMPILER FOR C LANGUAGE

● The declaration section contains the structure for each entries of symbol table that is identifier name, data type and attribute values (in our case, line number)

● Also, the declaration section contains all the pointer used in lexical phase to track each token that is each pointer from “lexicalAnalyzer.l” is imported here as extern type and used accordingly to raise lexical errors as well as syntactical errors using the production rules for each token

● Also, declaration contains function declaration for symbol table and constant table management (that is void makList(char *, char, int)

● The Rules section contains the various grammatical rules that are to be followed while parsing.

● The Programs section is where the yyparse() and the yyerror() functions are called,in order to generate the error message at the line where the syntax error has occurred.

● The Program section makes symbol table (“symblTable.txt”) and constant table (“constantTable.txt”) for each node of the link list struct tokenList

● If a production rule contains identifier or constants makeList( ) function will be invoked with respective type as ‘v’ or ‘c’ along with line number accordingly linked list for them is maintained

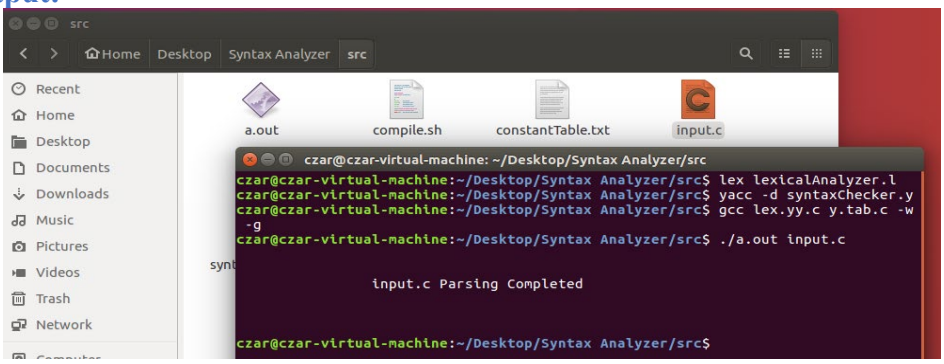
● void makeList(char *tokenName , char tokenType , int lineNumber) will add the identifier and constants accordingly by using linked list if an identifier is again came which is already there then it’s line number is appended to the first entry hence no duplicate entry for each identifier in symbol table

● Once the file is made, it is run using the shell script by giving the following command: ./compile.sh

Simple Input:

```c
#include<stdio.h>
void sum(int ,char);
void main()
{
        int a=5;
        int ;
        short ;
        char b;
        printf("\nEnter Number:");
        scanf("%d",&b);
        sum(5,b);
}
void sum(int inp1,char inp2){
        int s=0;
        s=inp1+inp2;
        printf("\nSum : %d",s);
        return ;
}
```

Simple Output:



COMPILER FOR C LANGUAGE

Symbol Table:

SymbolTable

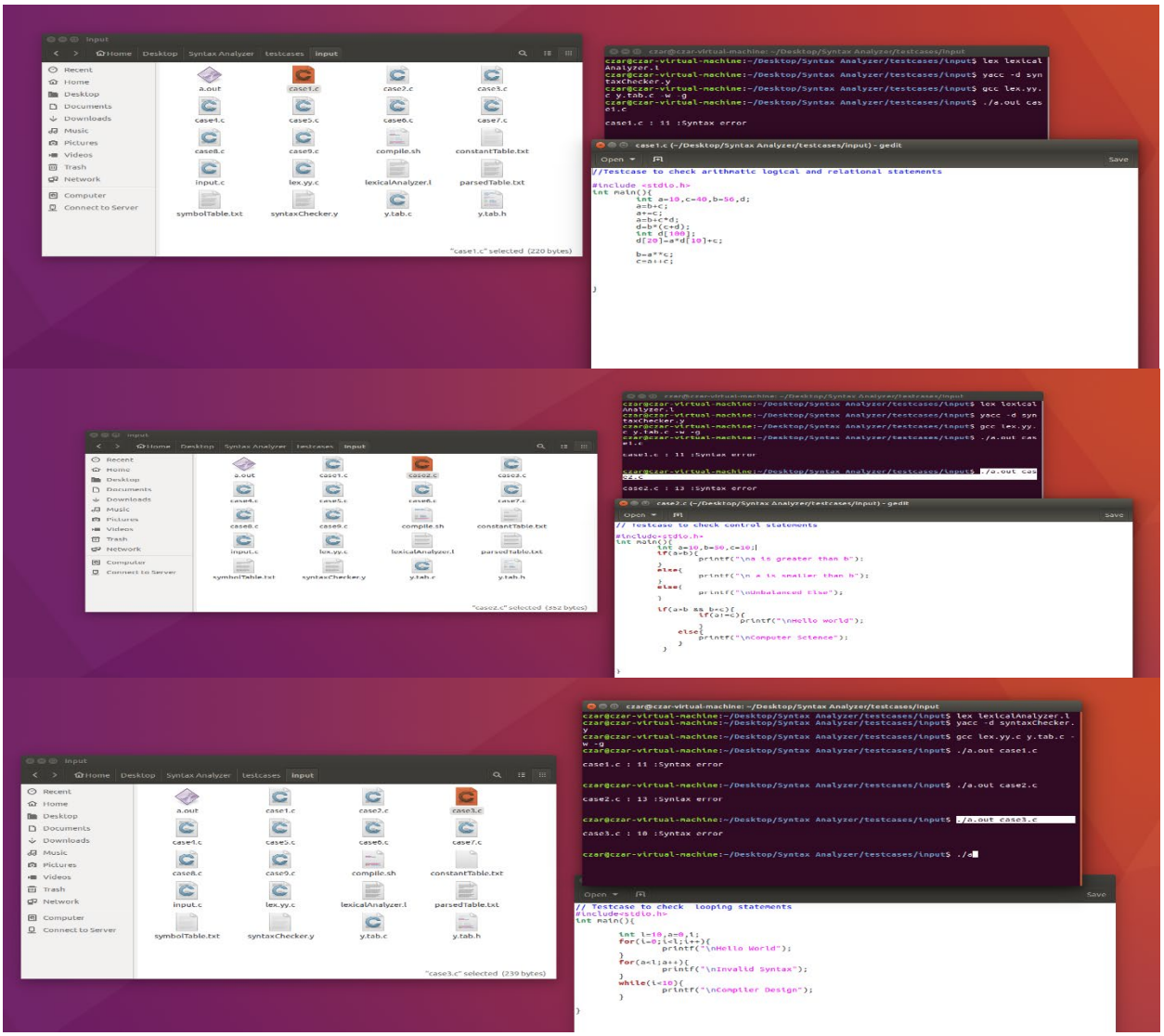
Token	Type	Line Number
sum	VOID	2 11 13
main	VOID	3
a	INT	5
b	CHAR	8 10 11
printf		9 16
scanf		10
inp1	INT	13 15
inp2	CHAR	13 15
s	INT	14 15 16

Constant Table:

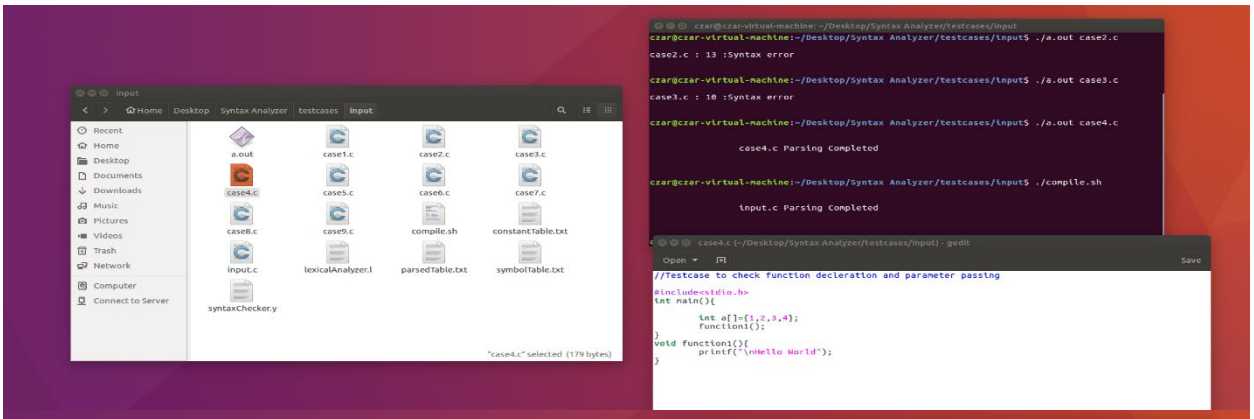
Constant Table

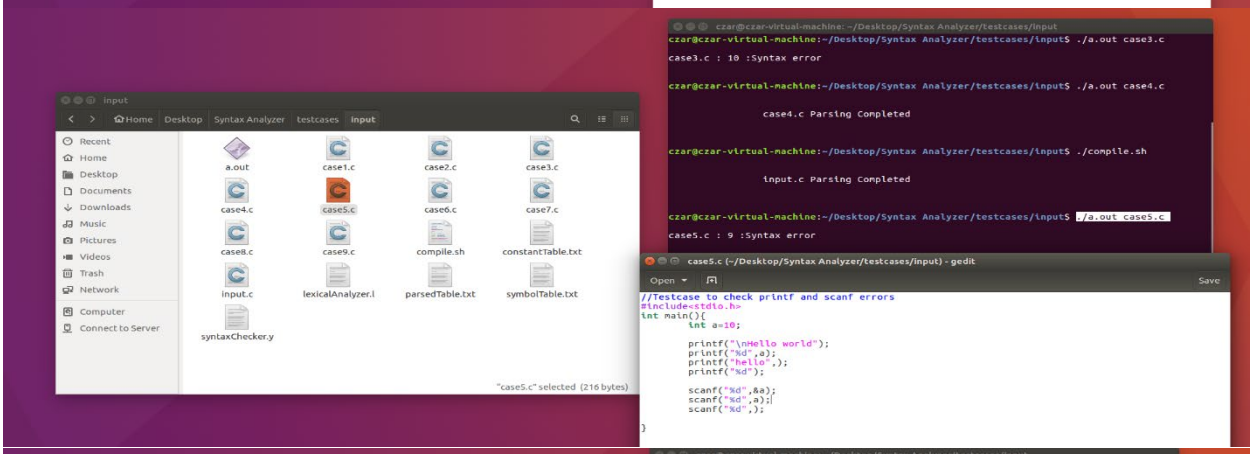
Value	Line Number
5	5 11
0	14

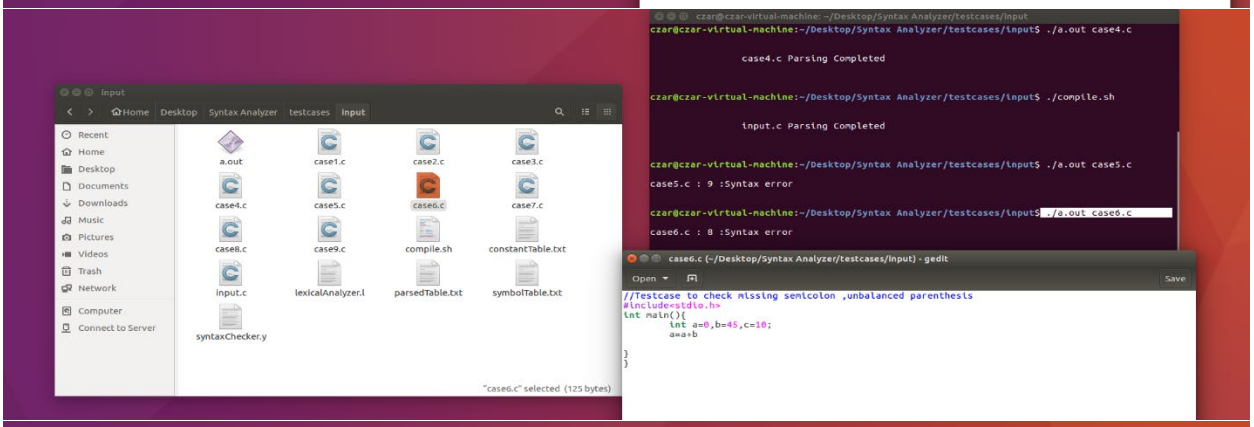
Test Case Evaluation:

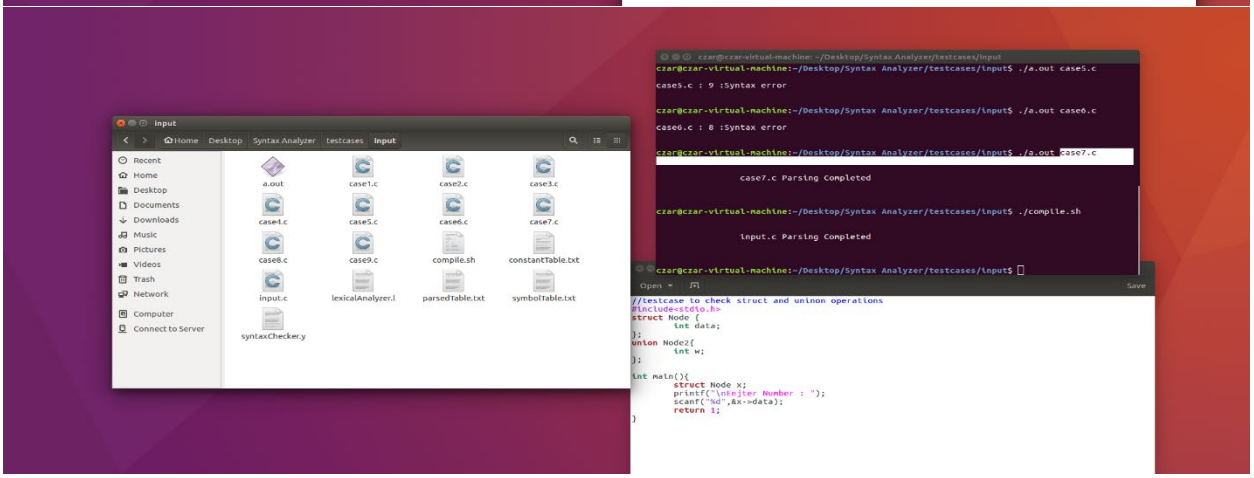


COMPILER FOR C LANGUAGE









COMPILER FOR C LANGUAGE

File Name	Purpose	Expected Output	Explanation	Status
Case1.c	Check arithmetic ,relational and logical statements	Case1.c : 11 : Syntax Error	Cannot use two arithmetic operator consecutively Unary plus should not have more than one operand	Passed
Case2.c	Check control and selection statements	Case2.c : 13 : Syntax Error	There is an unbalanced else in line number 13 ,ie there should be else for each if statements	Passed
Case3.c	Check for iteration statement validity	Case3.c : 10 : Syntax Error	There should be exactly 2 semicolon separates three statements in for loop syntax	Passed
Case4.c	Check for function declaration and parameter passing	Case4.c : Parsing Complete	All the function declaration and function invoking follows the grammar	Passed
Case5.c	Check printf and scanf function invoke errors	Case5.c : 8 : Syntax Error	Printf arguments should be separated by comma and should not end with comma	Passed
Case6.c	Check unbalanced parenthesis and missing semicolon	Case6.c : 8: Syntax Error	Unbalanced parenthesis at the end of the code	Passed
Case7.c	Check declaration syntax for structure and union variable	Case7.c : Parsing Complete	All the declaration follow the grammar rule	Passed

Conclusion:

After the second phase of this project, we have successfully implemented and added a Syntactic Analyzer/Parser to the C Compiler. This takes in the stream of tokens generated by the Lexical Analyzer implemented in the first phase as input.

For the output, it displays syntax errors along with their corresponding line number. Symbol table and Constant table have also been added. The following list of features have been included in this phase:

1. Syntax checking for Arithmetic, Logical and Relational Expressions
2. IF, IF...ELSE and Nested IF statements
3. Validation of Unary Operators
4. Validation of all loops
    a. FOR
    b. WHILE
    c. DO...WHILE
5. Proper declaration of functions and parameter passing
6. Array declaration
7. Structure and Union
8. Scanf and Printf errors
9. Missing semicolon at the end of statements
10. Unbalanced Parentheses

COMPILER FOR C LANGUAGE

SEMANTIC CHECKER FOR C

Introduction:

A parser constructs parse trees in the syntax analysis phase. The plain parse-tree constructed in that phase is generally of no use for a compiler, as it does not carry any information of how to evaluate the tree. The productions of context-free grammar, which makes the rules of the language, do not accommodate how to interpret them. For example,

$E \rightarrow E + T$

The above CFG production has no semantic rule associated with it, and it cannot help in making any sense of the production. Parsing only verifies that the program consists of tokens arranged in a syntactically valid combination.

Semantic analysis:

Semantics of a language provide meaning to its constructs, like tokens and syntax structure. Semantics help interpret symbols, their types, and their relations with each other. Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.
CFG + semantic rules = Syntax Directed Definitions

For example:
int a = “value”; should not issue an error in lexical and syntax analysis phase, as it is lexically and structurally correct, but it should generate a semantic error as the type of the assignment differs. These rules are set by the grammar of the language and evaluated in semantic analysis.

Semantic analysis is the task of ensuring that the declarations and statements of a program are semantically correct, i.e , that their meaning is clear and consistent with the way in which control structures and data types are supposed to be used. Semantic analysis is not a separate module within a compiler. It is usually a collection of procedures called at appropriate times by the parser as the grammar requires it. Implementing the semantic actions is conceptually simpler in recursive descent parsing because they are simply added to the recursive procedures. Implementing the semantic actions in a table - action driven LL(1) parser requires the addition of a third type of variable to the productions and the necessary software routines to process it.

Part of semantic analysis is producing some sort of representation of the program, either object code or an intermediate representation of the program. One - pass compilers will generate object code without using an intermediate representation; code generation is part of the semantic actions performed during parsing. Other compilers will produce an intermediate representation during semantic analysis; most often it will be an abstract syntax tree or quadruples. Semantic analysis typically involves:

- *Type checking* - Data types are used in a manner that is consistent with their definition (i. e., only with compatible data types, only with operations that are defined for them, etc.)
- *Label Checking* - Labels references in a program must exist.
- *Flow control checks* - Control structures must be used in their proper fashion (no GOTOs into a FORTRAN DO statement, no breaks outside a loop or switch statement, etc.)
- Array-bound Checking - Variables being used as an array index should be within the

bounds of the array.

● Scope Resolution - We need to determine what identifiers are accessible at different points in the program. Abstract syntax trees have one enormous advantage over other intermediate representations: they can be "decorated" , i.e., each node on the AST can have their attributes saved in the AST nodes, which can simplify the task of type checking as the parsing process continues.

Attribute grammar:

An attribute is a property whose value is assigned to a grammar symbol. Attribute computation functions (or semantic functions) are associated with the productions of a grammar and are used to compute the values of an attribute. An attribute grammar is an extension to a context - free grammar that is used to describe features of a programming language that cannot be described in BNF or can only be described in BNF with great difficulty. Each attribute has well-defined domain of values, such as integer, float, character, string, and expressions. Attribute grammar is a medium to provide semantics to the context-free grammar and it can help specify the syntax and semantics of a programming language. Attribute grammar (when viewed as a parse-tree) can pass values or information among the nodes of a tree.

E → E + T { E.value = E.value + T.value }

The right part of the CFG contains the semantic rules that specify how the grammar should be interpreted. Here, the values of non-terminals E and T are added together and the result is copied to the non-terminal E. Semantic attributes may be assigned to their values from their domain at the time of parsing and evaluated at the time of assignment or conditions. Based on the way the attributes get their values, they can be broadly divided into two categories:

● Synthesized attributes - These attributes get values from the attribute values of their child nodes. Synthesized attributes never take values from their parent nodes or any sibling nodes. For example,
S → ABC

If S is taking values from its child nodes (A, B, C), then it is said to be a synthesized attribute, as the values of ABC are synthesized to S.

● Inherited attributes - In contrast to synthesized attributes, inherited attributes can take values from parent and/or siblings. For example,
S → ABC

A can get values from S, B and C. B can take values from S, A, and C. Likewise, C can take values from S, A, and B.

Syntax Directed Translation:

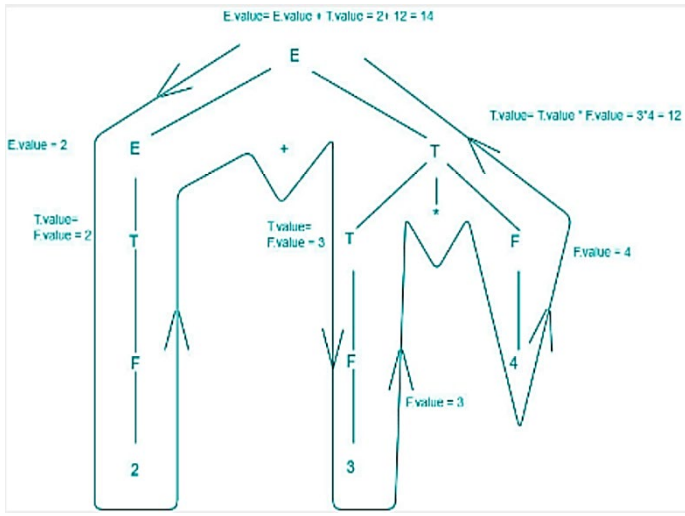
Parser uses a CFG(Context-free-Grammar) to validate the input string and produce output for next phase of the compiler. Output could be either a parse tree or abstract syntax tree. Now to interleave semantic analysis with syntax analysis phase of the compiler, we use Syntax Directed Translation. Syntax Directed Translation are augmented rules to the grammar that facilitate semantic analysis. SDT involves passing information bottom-up and/or top-down the parse tree in form of attributes attached to the nodes. Syntax directed translation rules use :

1) lexical values of nodes,
2) constants

3) attributes associated to the non-terminals in their definitions.

For example,
E -> E+T { E.val = E.val + T.val } PR#1
E -> T { E.val = T.val } PR#2
T -> T*F { T.val = T.val * F.val } PR#3
T -> F { T.val = F.val } PR#4
F -> INTLIT { F.val = INTLIT.lexval } PR#5

Suppose we take the first SDT augmented to [E -> E+T] production rule. The translation rule in consideration has val as attribute for both the non-terminals – E & T. Right hand side of the translation rule corresponds to attribute values of right side nodes of the production rule and vice-versa.



Above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children attributes are computed before parents.

● S-attributed SDT - If an SDT uses only synthesized attributes, it is called as S-attributed SDT. These attributes are evaluated using S-attributed SDTs that have their semantic actions written after the production (right hand side). Attributes in S-attributed SDTs are evaluated in bottom-up parsing, as the values of the parent nodes depend upon the values of the child nodes.

● L-attributed SDT - This form of SDT uses both synthesized and inherited attributes with restriction of not taking values from right siblings. In L-attributed SDTs, a non-terminal can get values from its parent, child, and sibling nodes. Attributes in L-attributed SDTs are evaluated by depth-first and left-to-right parsing manner.

Implementation:

Scope: Implemented using stack. Variable called scope is used which is initially set to 0. Increment and push to stack when opening brace is found. On finding closing brace, pop out the last element from stack. This will assign unique value to each block of code. Stack as a whole contains the scope-id, which is unique. In case of functions, the passed parameters are also part of the function block, therefore increment the scope variable when you encounter.

COMPILER FOR C LANGUAGE

Declaration Checking: To check whether an identifier is declared or not, check the symbol table corresponding to the value having scope-id equal to the current stack (which gives scope-id of the matched identifier). If not present, then undeclared.

Type Checking: When you encounter the type, use a global variable called flag which will indicate the type. For example, flag = 1 for int. When you encounter an assignment statement the current flag and the stored type value of the identifier has to match which is found out by consulting the symbol table.

Re-declaration Checking: When inserting a variable, check if the variable with same name exists with same scope-id. This is taken care by hashing the variable name along with the scope-id. Redeclaration is checked every time we insert identifier into the symbol table.

Array Dimension: When you encounter array declaration, pass the array dimension number into the symbol table. This makes sure that it's only positive integer integer.
Symbol Table. It contains fields:

1. Name: identifier name.
2. Token: token is constant or identifier.
3. Type: type of identifier whether int, float, void etc.
4. Scope: Scope could be global, function.
5. Scope-id: Unique value given to each block of code.
6. Function-id: Unique for each function declaration

All the semantic actions are written in semantic.h file, including maintaining symbol table. Necessary functions are called from syntaxChecker.y and lexicalAnalyzer.l for performing semantic actions and to report semantic error if any. Source code for all the files are included below.

Source Code:

Lexical Analyzer: This is the lex program that contains regular expressions and returns whatever tokens are required.

```
%{
#include <stdio.h>
#include "y.tab.h"
int lineCount=1;
int nestedCommentCount=0;
int commentFlag=0;
char *tablePtr;
void addToken(char*);
%}

digit                   [0-9]
letter                  [a-zA-Z_]
hex                     [a-fA-F0-9]
E                       [Ee][+-]?{digit}+
FS                      (f|F|l|letter)
IS                      (u|U|l|letter)*

singlelineComment (\/\/.*)
multilineCommentStart (\/\*)
multilineCommentEnd (\*\/)
%x DETECT_COMMENT

%%


{singlelineComment}                     { lineCount++; }

{multilineCommentStart}                 { BEGIN(DETECT_COMMENT);
                                          nestedCommentCount++;
                                        }


<DETECT_COMMENT>{multilineCommentStart} { nestedCommentCount++;
                                          if(nestedCommentCount>1)
                                                commentFlag = 1;
                                        }


<DETECT_COMMENT>{multilineCommentEnd}   { BEGIN(INITIAL); lineCount++;
                                                if(nestedCommentCount>0)
                                                        nestedCommentCount--;
                                                if(nestedCommentCount==0)
                                                        BEGIN(INITIAL);
                                        }

<DETECT_COMMENT>\n              {lineCount++;}
<DETECT_COMMENT>.               {}
```

COMPILER FOR C LANGUAGE

```
"auto"                  {  return(AUTO); }
"break"                 {  return(BREAK); }
"case"                  {  return(CASE); }
"char"                  {  return(CHAR); }
"const"                 {  return(CONST); }
"continue"              {  return(CONTINUE); }
"default"               {  return(DEFAULT); }
"do"                    {  return(DO); }
"double"                {  return(DOUBLE); }
"else"                  {  return(ELSE); }
"enum"                  {  return(ENUM); }
"extern"                {  return(EXTERN); }
"float"                 {  return(FLOAT); }
"for"                   {  return(FOR); }
"goto"                  {  return(GOTO); }
"if"                    {  return(IF); }
"int"                   {  return(INT); }
"long"                  {  return(LONG); }
"register"              {  return(REGISTER); }
"return"                {  return(RETURN); }
"short"                 {  return(SHORT); }
"signed"                {  return(SIGNED); }
"sizeof"                {  return(SIZEOF); }
"static"                {  return(STATIC); }
"struct"                {  return(STRUCT); }
"switch"                {  return(SWITCH); }
"typedef"               {  return(TYPEDEF); }
"union"                 {  return(UNION); }
"unsigned"              {  return(UNSIGNED); }
"void"                  {  return(VOID); }
"volatile"              {  return(VOLATILE); }
"while"                 {  return(WHILE); }

{letter}({letter}|{digit})*             {  addToken(yytext); return(IDENTIFIER); }
{letter}?\"(\\.|[^\\"])*\"              {  addToken(yytext); return(STRING_LITERAL); }

0[xX]{hex}+{IS}?                        {  addToken(yytext); return(CONSTANT); }
0{digit}+{IS}?                          {  addToken(yytext); return(CONSTANT); }
{hex}+{IS}?                             {  addToken(yytext); return(CONSTANT); }
{letter}?'(\\.|[^\\'])+'                {  addToken(yytext); return(CONSTANT); }
{digit}+{E}{FS}?                        {  addToken(yytext); return(CONSTANT); }
{digit}*"."{digit}+({E})?{FS}?          {  addToken(yytext); return(CONSTANT); }
{digit}+"."{digit}*({E})?{FS}?          {  addToken(yytext); return(CONSTANT); }

"..."                   {  return(ELLIPSIS); }
">>="                   {  return(RIGHT_ASSIGN); }
"<<="                   {  return(LEFT_ASSIGN); }
"+="                    {  return(ADD_ASSIGN); }
"-="                    {  return(SUB_ASSIGN); }
"*="                    {  return(MUL_ASSIGN); }
"/="                    {  return(DIV_ASSIGN); }
"%="                    {  return(MOD_ASSIGN); }
"&="                    {  return(AND_ASSIGN); }
"^="                    {  return(XOR_ASSIGN); }
"|="                    {  return(OR_ASSIGN); }
">>"                    {  return(RIGHT_OP); }
"<<"                    {  return(LEFT_OP); }
"++"                    {  return(INC_OP); }
"--"                    {  return(DEC_OP); }
"->"                    {  return(PTR_OP); }
"&&"                    {  return(AND_OP); }
"||"                    {  return(OR_OP); }
"<="                    {  return(LE_OP); }
">="                    {  return(GE_OP); }
"=="                    {  return(EQ_OP); }
"!="                    {  return(NE_OP); }
";"                     {  return(';'); }
("{"|"<%")              {  return('{'); }
("}"|"%>")              {  return('}'); }
","                     {  return(','); }
":"                     {  return(':'); }
"="                     {  return('='); }
"("                     {  return('('); }
")"                     {  return(')'); }
("["|"<:")              {  return('['); }
("]"|":>")              {  return(']'); }
"."                     {  return('.'); }
"&"                     {  return('&'); }
"!"                     {  return('!'); }
"~"                     {  return('~'); }
"-"                     {  return('-'); }
"+"                     {  return('+'); }
"*"                     {  return('*'); }
"/"                     {  return('/'); }
"%"                     {  return('%'); }
"<"                     {  return('<'); }
">"                     {  return('>'); }
"^"                     {  return('^'); }
"|"                     {  return('|'); }
"?"                     {  return('?'); }


"#include"(.)*"\n"      { lineCount++; }
"#define"(.)*"\n"       { lineCount++; }

[ ]                     {}
[\t\v\f]                {}
[\n]                    {lineCount++;}
.                       { }

%%
yywrap()
{
        return(1);
}

void addToken(char *yytext)
{
  int len = strlen(yytext);
  tablePtr = (char*)malloc((len+1)*sizeof(char));
  strcpy(tablePtr, yytext);
}
```

COMPILER FOR C LANGUAGE

Syntax Analyzer: This is the main parser code, a yacc file which contains the declarations, rules and programs and defines the actions which should be taken for various cases.

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "y.tab.h"
#include "semantic.h"

extern FILE *yyin;
extern int lineCount;
extern char *tablePtr;
extern char *tablePtr;
extern int nestedCommentCount;
extern int commentFlag;
extern int arrayIndexErr;

char *sourceCode=NULL;
int errorFlag=0;
void makeList(char *,char,int);
%}

%token  AUTO BREAK  CASE CHAR  CONST  CONTINUE  DEFAULT  DO DOUBLE  ELSE ENUM
%token EXTERN FLOAT  FOR GOTO  IF INT LONG REGISTER  RETURN SHORT SIGNED

%token SIZEOF STATIC STRUCT SWITCH TYPEDEF UNION UNSIGNED VOID VOLATILE WHILE

%token IDENTIFIER CONSTANT FLCONSTANT STRING_LITERAL

%token ELLIPSIS

%token PTR_OP INC_OP DEC_OP LEFT_OP RIGHT_OP LE_OP GE_OP EQ_OP NE_OP
%token AND_OP OR_OP MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN ADD_ASSIGN
%token SUB_ASSIGN LEFT_ASSIGN RIGHT_ASSIGN AND_ASSIGN
%token XOR_ASSIGN OR_ASSIGN TYPE_NAME

%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE
%start translation_unit
%%

primary_expression
        : IDENTIFIER            { $$=checkScope(tablePtr,lineCount); }
        | FLCONSTANT            {tempCheckType=4;}
        | CONSTANT              { addConstant(tablePtr, lineCount);}
        | STRING_LITERAL        { makeList(tablePtr, 's', lineCount);}
        | '(' expression ')'    { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); $$=$2; }
        ;
postfix_expression
        : primary_expression   {$$=$1;}
        | postfix_expression '[' expression ']'                 { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | postfix_expression '(' ')'                            { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | postfix_expression '(' argument_expression_list ')'   { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | postfix_expression '.' IDENTIFIER                     { makeList(tablePtr, 'v', lineCount);}
        | postfix_expression PTR_OP IDENTIFIER                  { makeList(tablePtr, 'v', lineCount);}
        | postfix_expression INC_OP                             { makeList(tablePtr, 'o', lineCount);}
        | postfix_expression DEC_OP                             { makeList(tablePtr, 'o', lineCount);}
        ;
argument_expression_list
        : assignment_expression {$$=$1;}
        | argument_expression_list ',' assignment_expression { makeList(",",'p', lineCount); }
        ;
unary_expression
        : postfix_expression {$$=$1;}
        | INC_OP unary_expression       { makeList("++",'o', lineCount); }
        | DEC_OP unary_expression       { makeList("--",'o', lineCount); }
        | unary_operator cast_expression { makeList("sizeof",'o', lineCount); }
        | SIZEOF unary_expression       { makeList("sizeof",'o', lineCount); }
        | SIZEOF '(' type_name ')'      { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
unary_operator
        : '&' { makeList("&",'o', lineCount); }
        | '*' { makeList("*",'o', lineCount); }
        | '+' { makeList("+",'o', lineCount); }
        | '-' { makeList("-",'o', lineCount); }
        | '~' { makeList("~",'o', lineCount); }
        | '!' { makeList("!",'o', lineCount); }
        ;
cast_expression
        : unary_expression   {$$=$1;}
        | '(' type_name ')' cast_expression { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
multiplicative_expression
        : cast_expression {$$=$1;}
        | multiplicative_expression '*' cast_expression { makeList("*",'o', lineCount); }
        | multiplicative_expression '/' cast_expression { makeList("/",'o', lineCount); }
        | multiplicative_expression '%' cast_expression { makeList("%",'o', lineCount); }
        ;
additive_expression
        : multiplicative_expression {$$=$1;}
        | additive_expression '+' multiplicative_expression { makeList("+",'o', lineCount); }
        | additive_expression '-' multiplicative_expression { makeList("-",'o', lineCount); }
        ;
shift_expression
        : additive_expression {$$=$1;}
        | shift_expression LEFT_OP additive_expression  { makeList("<<",'o', lineCount); }
        | shift_expression RIGHT_OP additive_expression { makeList(">>",'o', lineCount); }
        ;
relational_expression
        : shift_expression {$$=$1;}
        | relational_expression '<' shift_expression
        | relational_expression '>' shift_expression
        | relational_expression LE_OP shift_expression { makeList("<=",'o', lineCount); }
        | relational_expression GE_OP shift_expression { makeList(">=",'o', lineCount); }
        ;
equality_expression
        : relational_expression {$$=$1;}
        | equality_expression EQ_OP relational_expression { makeList("==",'o', lineCount); }
        | equality_expression NE_OP relational_expression { makeList("!=",'o', lineCount); }
        ;
and_expression
        : equality_expression {$$=$1;}
        | and_expression '&' equality_expression        { makeList("&", 'o', lineCount);}
        ;
exclusive_or_expression
        : and_expression {$$=$1;}
        | exclusive_or_expression '^' and_expression    { makeList("^", 'o', lineCount); }
        ;
inclusive_or_expression
        : exclusive_or_expression {$$=$1;}
        | inclusive_or_expression '|' exclusive_or_expression { makeList("|", 'o', lineCount); }
        ;
logical_and_expression
        : inclusive_or_expression {$$=$1;}
        | logical_and_expression AND_OP inclusive_or_expression { makeList("&&", 'o', lineCount); }
        ;
logical_or_expression
        : logical_and_expression {$$=$1;}
        | logical_or_expression OR_OP logical_and_expression { makeList("||", 'o', lineCount); }
        ;
conditional_expression
        : logical_or_expression {$$=$1;}
        | logical_or_expression '?' expression ':' conditional_expression { makeList("?:",'o', lineCount); }
        ;
assignment_expression
        : conditional_expression {$$=$1;}
        | unary_expression assignment_operator assignment_expression {$$=$3; checkType($1,$3,lineCount);}
```

COMPILER FOR C LANGUAGE

```
        ;
assignment_operator
        : '='           { makeList("=",'o', lineCount); }
        | MUL_ASSIGN    { makeList("*=",'o', lineCount); }
        | DIV_ASSIGN    { makeList("/=",'o', lineCount); }
        | MOD_ASSIGN    { makeList("%=",'o', lineCount); }
        | ADD_ASSIGN    { makeList("+=",'o', lineCount); }
        | SUB_ASSIGN    { makeList("-=",'o', lineCount); }
        | LEFT_ASSIGN   { makeList("<<=",'o', lineCount); }
        | RIGHT_ASSIGN  { makeList(">>=",'o', lineCount); }
        | AND_ASSIGN    { makeList("&=",'o', lineCount); }
        | XOR_ASSIGN    { makeList("^=",'o', lineCount); }
        | OR_ASSIGN     { makeList("|=",'o', lineCount); }
        ;
expression
        : assignment_expression {$$=$1;}
        | expression ',' assignment_expression { makeList(",", 'p', lineCount); }
        ;
constant_expression
        : conditional_expression
        ;
declaration
        : declaration_specifiers ';'                        { makeList(";", 'p', lineCount);typeBuffer=' '; }
        | declaration_specifiers init_declarator_list ';' { makeList(";", 'p', lineCount); typeBuffer=' ';}
        ;
declaration_specifiers
        : storage_class_specifier
        | storage_class_specifier declaration_specifiers
        | type_specifier
        | type_specifier declaration_specifiers
        | type_qualifier
        | type_qualifier declaration_specifiers
        ;
init_declarator_list
        : init_declarator
        | init_declarator_list ',' init_declarator { makeList(",", 'p', lineCount); }
        ;
init_declarator
        : declarator
        | declarator '=' initializer { makeList("=", 'o', lineCount); }
        ;
storage_class_specifier
        : TYPEDEF       { makeList("typedef", 'k', lineCount);}
        | EXTERN        { makeList("extern", 'k', lineCount);}
        | STATIC        { makeList("static", 'k', lineCount);}
        | AUTO          { makeList("auto", 'k', lineCount);}
        | REGISTER      { makeList("register", 'k', lineCount);}
        ;
type_specifier
        : VOID          { makeList("void", 'k', lineCount);typeBuffer='v';}
        | CHAR          { makeList("char", 'k', lineCount); typeBuffer='c';}
        | SHORT         { makeList("short", 'k', lineCount);}
        | INT           { makeList("int", 'k', lineCount); typeBuffer='i';}
        | LONG          { makeList("long", 'k', lineCount);}
        | FLOAT         { makeList("float", 'k', lineCount);    typeBuffer='f';}
        | DOUBLE        { makeList("double", 'k', lineCount);}
        | SIGNED        { makeList("signed", 'k', lineCount);}
        | UNSIGNED      { makeList("unsigned", 'k', lineCount);}
        | struct_or_union_specifier
        | enum_specifier
        | TYPE_NAME
        ;
struct_or_union_specifier
        : struct_or_union IDENTIFIER '{' struct_declaration_list '}'
        | struct_or_union '{' struct_declaration_list '}'
        | struct_or_union IDENTIFIER
        ;
struct_or_union
        : STRUCT        { makeList("struct", 'k', lineCount);}
        | UNION         { makeList("union", 'k', lineCount);}
        ;
struct_declaration_list
        : struct_declaration
        | struct_declaration_list struct_declaration
        ;
struct_declaration
        : specifier_qualifier_list struct_declarator_list ';' { makeList(";", 'p', lineCount); }
        ;
specifier_qualifier_list
        : type_specifier specifier_qualifier_list
        | type_specifier
        | type_qualifier specifier_qualifier_list
        | type_qualifier
        ;
struct_declarator_list
        : struct_declarator
        | struct_declarator_list ',' struct_declarator { makeList(",", 'p', lineCount); }
        ;
struct_declarator
        : declarator
        | ':' constant_expression               { makeList(":", 'p', lineCount); }
        | declarator ':' constant_expression    { makeList(":", 'p', lineCount); }
        ;
enum_specifier
        : ENUM '{' enumerator_list '}'                  { makeList("enum", 'k', lineCount);}
        | ENUM IDENTIFIER '{' enumerator_list '}'       { makeList("enum", 'k', lineCount); makeList(tablePtr, 'v', lineCount); }
        | ENUM IDENTIFIER                               { makeList("enum", 'k', lineCount); makeList(tablePtr, 'v', lineCount); }
        ;
enumerator_list
        : enumerator
        | enumerator_list ',' enumerator { makeList(",", 'p', lineCount); }
        ;
enumerator
        : IDENTIFIER                            { makeList(tablePtr, 'v', lineCount); }
        | IDENTIFIER '=' constant_expression    { makeList("=", 'o', lineCount); makeList("tablePtr", 'v', lineCount); }
        ;
type_qualifier
        : CONST         { makeList("const", 'k', lineCount); }
        | VOLATILE      { makeList("volatile", 'k', lineCount); }
        ;
declarator
        : pointer direct_declarator
        | direct_declarator
        ;
direct_declarator
        : IDENTIFIER                                        {  checkDeclaration(tablePtr,lineCount,scopeCount);}
        | '(' declarator ')'                                {  makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | direct_declarator '[' constant_expression ']'     { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | direct_declarator '[' ']'                         { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | direct_declarator '(' parameter_type_list ')'     { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | direct_declarator '(' identifier_list ')'         { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | direct_declarator '(' ')'                         { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
pointer
        : '*'                                   { makeList("*", 'o', lineCount); }
        | '*' type_qualifier_list               { makeList("*", 'o', lineCount); }
        | '*' pointer                           { makeList("*", 'o', lineCount); }
        | '*' type_qualifier_list pointer       { makeList("*", 'o', lineCount); }
        ;
type_qualifier_list
        : type_qualifier
        | type_qualifier_list type_qualifier
        ;
parameter_type_list
        : parameter_list
```

COMPILER FOR C LANGUAGE

```
        | parameter_list ',' ELLIPSIS          { makeList(",", 'p', lineCount); makeList("::", 'o', lineCount); }
        ;
parameter_list
        : parameter_declaration
        | parameter_list ',' parameter_declaration { makeList(",", 'p', lineCount); }
        ;
parameter_declaration
        : declaration_specifiers declarator
        | declaration_specifiers abstract_declarator
        | declaration_specifiers
        ;
identifier_list
        : IDENTIFIER                           { checkDeclaration(tablePtr,lineCount,scopeCount);}
        | identifier_list ',' IDENTIFIER       { checkDeclaration(tablePtr,lineCount,scopeCount);makeList(",", 'p', lineCount); }
        ;
type_name
        : specifier_qualifier_list
        | specifier_qualifier_list abstract_declarator
        ;
abstract_declarator
        : pointer
        | direct_abstract_declarator
        | pointer direct_abstract_declarator
        ;
direct_abstract_declarator
        : '(' abstract_declarator ')'                                  { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | '[' ']'                                                      { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | '[' constant_expression ']'                                  { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | direct_abstract_declarator '[' ']'                           { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | direct_abstract_declarator '[' constant_expression ']'       { makeList("[", 'p', lineCount); makeList("]", 'p', lineCount); }
        | '(' ')'                                                      { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | '(' parameter_type_list ')'                                  { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | direct_abstract_declarator '(' ')'                           { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | direct_abstract_declarator '(' parameter_type_list ')'       { makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
initializer
        : assignment_expression {$$=$1;}
        | '{' initializer_list '}'
        | '{' initializer_list ',' '}'
        ;
initializer_list
        : initializer
        | initializer_list ',' initializer { makeList(",", 'p', lineCount); }
        ;
statement
        : labeled_statement
        | compound_statement
        | expression_statement
        | selection_statement
        | iteration_statement
        | jump_statement
        ;
labeled_statement
        : IDENTIFIER ':' statement                    { makeList(tablePtr, 'v', lineCount);  }
        | CASE constant_expression ':'  statement     { makeList(":", 'p', lineCount); makeList("case", 'k', lineCount);}
        | DEFAULT ':' statement                       { makeList(":", 'p', lineCount); makeList("default", 'k', lineCount); }
        ;
compound_statement
        : '{' '}'
        | '{' statement_list '}'
        | '{' declaration_list '}'
        | '{' declaration_list statement_list '}'
        ;
declaration_list
        : declaration
        | declaration_list declaration
        ;
statement_list
        : statement
        | statement_list statement
        ;
expression_statement
        : ';'                          { makeList(";", 'p', lineCount); }
        | expression ';'               { makeList(";", 'p', lineCount); }
        ;
selection_statement
        : IF '(' expression ')' statement %prec LOWER_THAN_ELSE
                                 { makeList("if", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount);}
        | IF '(' expression ')' statement ELSE statement
                                 { makeList("if", 'k', lineCount);  makeList("else", 'k', lineCount); makeList("(", 'p', lineCount);
                                 }
        | SWITCH '(' expression ')' statement
                                 { makeList("switch", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
iteration_statement
        : WHILE '(' expression ')' statement
                        { makeList("while", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | DO statement WHILE '(' expression ')' ';'
                        { makeList("do", 'k', lineCount); makeList("while", 'k', lineCount); makeList("(", 'p', lineCount);
                        }
        | FOR '(' expression_statement expression_statement ')' statement
                        { makeList("for", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        | FOR '(' expression_statement expression_statement expression ')' statement
                        { makeList("for", 'k', lineCount); makeList("(", 'p', lineCount); makeList(")", 'p', lineCount); }
        ;
jump_statement
        : GOTO IDENTIFIER ';'   { makeList("goto", 'k', lineCount); makeList(";", 'p', lineCount); makeList(tablePtr, 'v', lineCount);}
        | CONTINUE ';'          { makeList("continue", 'k', lineCount); makeList(";", 'p', lineCount); }
        | BREAK ';'             { makeList("break", 'k', lineCount); makeList(";", 'p', lineCount);}
        | RETURN ';'            { makeList("return", 'k', lineCount); makeList(";", 'p', lineCount);}
        | RETURN expression ';' { makeList("return", 'k', lineCount); makeList(";", 'p', lineCount);}
        ;
translation_unit
        : external_declaration
        | translation_unit external_declaration
        ;
external_declaration
        : function_definition
        | declaration
        ;
function_definition
        : declaration_specifiers declarator declaration_list compound_statement
        | declaration_specifiers declarator compound_statement
        | declarator declaration_list compound_statement
        | declarator compound_statement
        ;
%%

void yyerror()
{
        errorFlag=1;
        fflush(stdout);
        printf("\n%s : %d :Syntax error \n",sourceCode,lineCount);
}

void main(int argc,char **argv){
        if(argc<=1){

                printf("Invalid ,Expected Format : ./a.out <\"sourceCode\"> \n");
                return 0;
        }

        yyin=fopen(argv[1],"r");

        sourceCode=(char *)malloc(strlen(argv[1])*sizeof(char));
```

COMPILER FOR C LANGUAGE

```c
        sourceCode=argv[1];
        yyparse();

        if(nestedCommentCount!=0){
                errorFlag=1;
                printf("%s : %d : Comment Does Not End\n",sourceCode,lineCount);

        }

        if(commentFlag==1){
                errorFlag=1;
                printf("%s : %d : Nested Comment\n",sourceCode,lineCount);
        }

        if(!errorFlag  && !semanticErr  && arrayIndexErr!=1){

                printf("\n\n\t\t%s Parsing Completed\n\n",sourceCode);

                FILE *writeParsed=fopen("parsedTable.txt","w");
                fprintf(writeParsed,"\n\t\t\t\tParsed Table\n\n\t\tToken\t\t\t\t\t\tType\t\t\t\t\t\t\tLine Number\n");
                for(tokenList *ptr=parsedPtr;ptr!=NULL;ptr=ptr->next){
                        fprintf(writeParsed,"\n%20s%30.30s%60s",ptr->token,ptr->type,ptr->line);
                }

                FILE *writeSymbol=fopen("symbolTable.txt","w");
                fprintf(writeSymbol,"\n\t\t\t\tSymbolTable\n\n\t\tToken\t\t\t\tType\t\tLine Number\t\t\t\tScope\t\tFunction Number\n");
                for(tokenList *ptr=symbolPtr;ptr!=NULL;ptr=ptr->next){
                        fprintf(writeSymbol,"\n%20s%30.30s%30s%30s\t%d\t%d",ptr->token,ptr->type,ptr->line,ptr->scope,ptr->scopeValue,ptr->funcCount
                }

                FILE *writeConstant=fopen("constantTable.txt","w");
                fprintf(writeConstant,"\n    \t\t\t\t\t\t\t\tConstant Table \n\n \t\t\t\t\t\tValue\t\t\t\t\t\t\tLine Number\n");
                for(tokenList *ptr=constantPtr;ptr!=NULL;ptr=ptr->next)
                fprintf(writeConstant,"\n%50s%60s",ptr->token,ptr->line);

                fclose(writeSymbol);
                fclose(writeConstant);
                fclose(writeParsed);
        }
printf("\n\n");
}
```

Semantic Analyzer: This is the semantic phase code, a c header file which contains the function that is called by different production rule in parser phase and taken care of symbol table entry.

```c
struct tokenList
{
        char *token,type[20],line[100];
        char *scope[20];
        int scopeValue;
        int funcCount;
        struct tokenList *next;
};
typedef struct tokenList tokenList;
struct funcNode{
        char funcName[30];
        int line;
        char funcReturn[20];
        struct  funcNode *next;
};
typedef struct funcNode funcNode;


tokenList *symbolPtr = NULL;
tokenList *constantPtr = NULL;
tokenList *parsedPtr =NULL;
extern int functionCount;
extern int scopeCount;
char typeBuffer=' ';
char *sourceCode;
int tempCheckType=3;

int semanticErr=0,lineSemanticCount;
int checkScope(char *tempToken,int lineCount)
{       tokenList *temp=NULL;
        char type[20];
        int flag=0,tempFlag=0;
        for(tokenList *p=symbolPtr;p!=NULL;p=p->next){
                if(strcmp(tempToken,"printf")==0 ||  strcmp(tempToken,"scanf")==0){
                        tempFlag=1;
                }
                else{
                        if(strcmp(tempToken,p->token)==0){
                                strcpy(type,p->type);
                                flag=1;
                                break;
                        }
                }

        }
        if (flag == 0 && tempFlag ==0 )
        {
                printf("\n%s : %d :Undeclared variable  \n",sourceCode,lineCount-1);
                semanticErr=1;
        }
        else
        {
                addSymbol(tempToken,lineCount);
                if(strcmp(type,"VOID")==0)
                        return(1);
                if(strcmp(type,"CHAR")==0)
                        return(2);
                if(strcmp(type,"INT")==0)
                        return(3);
                if(strcmp(type,"FLOAT")==0)
                        return(4);
        }

}

void checkType(int value1,int value2,int lineCount)
{       lineSemanticCount=lineCount;
        if(value2 == 0)
                value2 = tempCheckType;
        if(value1!=value2)
        {
                printf("\n%s : %d :Type Mismatch error \n",sourceCode,lineSemanticCount-1);
                semanticErr=1;
        }
        tempCheckType=3;
}
void checkDeclaration(char *tokenName,int tokenLine,int scopeVal){
        char type[20];
        char line[39],lineBuffer[19];
        snprintf(lineBuffer, 19, "%d", tokenLine);
        strcpy(line,"  ");
        strcat(line,lineBuffer);
        switch(typeBuffer){
                case 'i': strcpy(type,"INT"); break;
                case 'f': strcpy(type,"FLOAT");break;
                case 'v': strcpy(type,"VOID");break;
                case 'c': strcpy(type,"CHAR");break;

        }
        for(tokenList *p=symbolPtr;p!=NULL;p=p->next){
```

COMPILER FOR C LANGUAGE

```c
                if(strcmp(p->token,tokenName)==0 && p->scopeValue == scopeCount && p->funcCount == functionCount){
                        semanticErr=1;
                        if(strcmp(p->type,type)==0){
                                printf("\n%s : %d :Multiple Declaration \n",sourceCode,tokenLine);
                                return;
                        }
                        else{
                                printf("\n%s : %d :Multiple Declration with Different Type \n",sourceCode,tokenLine);
                                return;
                        }
                }
        }
        addSymbol(tokenName,tokenLine,scopeCount);

}
void checkArray(int val,int lineCount){
        if(val<0){
                semanticErr=1;
                printf("\n%s : %d :Array Index error\n",sourceCode,lineCount-1);
        }
}
void addSymbol(char *tokenName,int tokenLine,int scopeVal){
        char line[39],lineBuffer[19];
        snprintf(lineBuffer, 19, "%d", tokenLine);
        strcpy(line," ");
        strcat(line,lineBuffer);
        char type[20];
        for(tokenList *p=symbolPtr;p!=NULL;p=p->next)
                if(strcmp(p->token,tokenName)==0 && p->scopeValue == scopeCount && p->funcCount ==functionCount ){
                        strcat(p->line,line);
                        return;
                }
        tokenList *temp=(tokenList *)malloc(sizeof(tokenList));
        temp->token=(char *)malloc(strlen(tokenName)+1);
        strcpy(temp->token,tokenName);
        switch(typeBuffer){
                case 'i': strcpy(temp->type,"INT"); break;
                case 'f': strcpy(temp->type,"FLOAT");break;
                case 'v': strcpy(temp->type,"VOID");break;
                case 'c': strcpy(temp->type,"CHAR");break;
        }
        temp->funcCount=functionCount;
        if(scopeCount==0){
                strcpy(temp->scope,"GLOBAL");
                temp->scopeValue=scopeCount;

        }
        else{
                strcpy(temp->scope,"NESTING");
                temp->scopeValue=scopeCount;
        }
        strcpy(temp->line,line);
        temp->next=NULL;
        tokenList *p=symbolPtr;
        if(p==NULL){
                symbolPtr=temp;
        }
        else{
                while(p->next!=NULL){
                        p=p->next;
                }
                p->next=temp;
        }


}
void addConstant(char *tokenName,int tokenLine){
        char line[39],lineBuffer[19];
        snprintf(lineBuffer, 19, "%d", tokenLine);
        strcpy(line," ");
        strcat(line,lineBuffer);
        for(tokenList *p=constantPtr;p!=NULL;p=p->next)
                if(strcmp(p->token,tokenName)==0){
                        strcat(p->line,line);
                        return;
                }
                tokenList *temp=(tokenList *)malloc(sizeof(tokenList));
                temp->token=(char *)malloc(strlen(tokenName)+1);
                strcpy(temp->token,tokenName);
                strcpy(temp->line,line);
                temp->next=NULL;

                tokenList *p=constantPtr;
                if(p==NULL){
                        constantPtr=temp;
                }
                else{
                        while(p->next!=NULL){
                                p=p->next;
                        }
                        p->next=temp;
                }


}
void makeList(char *tokenName,char tokenType, int tokenLine)
{
        char line[39],lineBuffer[19];

        snprintf(lineBuffer, 19, "%d", tokenLine);
        strcpy(line," ");
        strcat(line,lineBuffer);
        char type[20];
        switch(tokenType)
        {
                        case 'c':
                                        strcpy(type,"Constant");
                                        break;
                        case 'v':
                                        strcpy(type,"Identifier");
                                        break;
                        case 'p':
                                        strcpy(type,"Punctuator");
                                        break;
                        case 'o':
                                        strcpy(type,"Operator");
                                        break;
                        case 'k':
                                        strcpy(type,"Keyword");
                                        break;
                        case 's':
                                        strcpy(type,"String Literal");
                                        break;
                        case 'd':
                                        strcpy(type,"Preprocessor Statement");
                                        break;
        }


        for(tokenList *p=parsedPtr;p!=NULL;p=p->next)
                if(strcmp(p->token,tokenName)==0){
                        strcat(p->line,line);
                        return;
                }
                tokenList *temp=(tokenList *)malloc(sizeof(tokenList));
```

COMPILER FOR C LANGUAGE

```c
                        case 'k':
                                        strcpy(type,"Keyword");
                                        break;
                        case 's':
                                        strcpy(type,"String Literal");
                                        break;
                        case 'd':
                                        strcpy(type,"Preprocessor Statement");
                                        break;
        }


        for(tokenList *p=parsedPtr;p!=NULL;p=p->next)
                        if(strcmp(p->token,tokenName)==0){
                                strcat(p->line,line);
                                return;
                        }
                tokenList *temp=(tokenList *)malloc(sizeof(tokenList));
                temp->token=(char *)malloc(strlen(tokenName)+1);
                strcpy(temp->token,tokenName);
                strcpy(temp->type,type);
                strcpy(temp->line,line);
                temp->next=NULL;

                tokenList *p=parsedPtr;
                if(p==NULL){
                        parsedPtr=temp;
                }
                else{
                        while(p->next!=NULL){
                                p=p->next;
                        }
                        p->next=temp;
                }
}
```

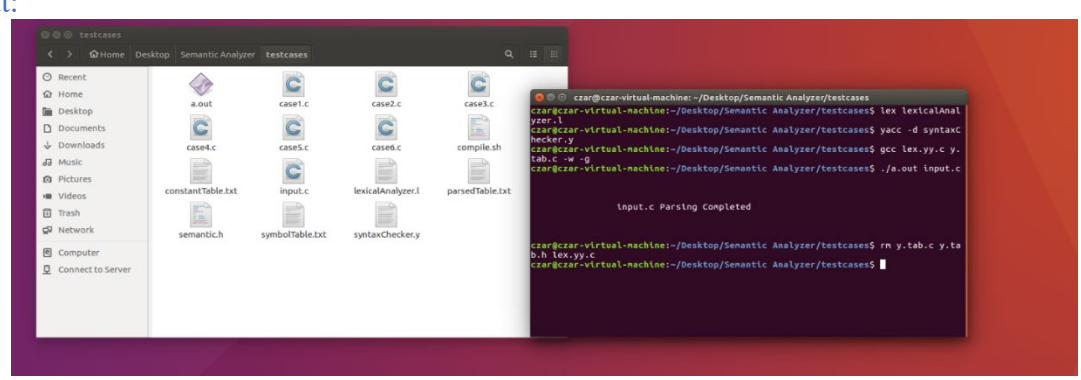
Execution Of the Code:

The following is a shell script to automate the compilation and execution of the mini compiler consist of lexical, syntax and semantic phases. There is sample input source code given. Execution of the same is done and output is shown as sample output and respective symbol table is generated.

Input:

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

        int a=10,b;
        char c;
        int f[100][10];
        a=b;
        a=b+12;
        printf("\nHello World");
        scanf();
        {
        int y;
        int a;
        }
```

Output:



COMPILER FOR C LANGUAGE

Symbol Table:

SymbolTable

Token	Type	Line Number	Scope	Function Number
func	VOID	2	GLOBAL 0	1
a	INT	3	NESTING 1	1
main	VOID	5	GLOBAL 0	2
a	INT	8 11 12	NESTING 1	2
b	INT	8 11 12	NESTING 1	2
c	CHAR	9	NESTING 1	2
f	INT	10	NESTING 1	2
printf		13	NESTING 1	2
scanf		14	NESTING 1	2
y	INT	16	NESTING 2	2
a	INT	17	NESTING 2	2

Test Cases:

Different test cases are added to check implementation of semantic phase. The validation of the same is done below in a table format. Source code for the 6 test cases is added as screenshot below.

Test Case Evaluation

File Name	Purpose	Expected Output	Explanation	Status
case1.c	Check Multiple Declaration	case1.c : 9 :Multiple declaration with different Type case1.c : 11 :Multiple Declaration	Multiple declaration of variable a and b	**Passed**
case2.c	Check Undeclared variable	case2.c : 6 :Undeclared variable case2.c : 6 :Type Mismatch error case2.c : 7 :Undeclared variable	Variable d is undeclared	**Passed**
case3.c	Check Type mismatch variable	case3.c : 7 :Multiple Declaration with Different Type case3.c : 7 :Type Mismatch error	Variable l is of type *int* but assignment expression of type float	**Passed**
case4.c	Check scope of variable	case4.c Parsing Completed	Though multiple declaration of a but all under different scope	**Passed**
case5.c	Check scope of variable with functions	case5.c Parsing Completed	Multiple declaration under different compound statement	**Passed**
case6.c	Check array dimension	case6.c : 7 : Invalid array Index	*arr2* has array dimension negative	**Passed**

COMPILER FOR C LANGUAGE

Conclusion:

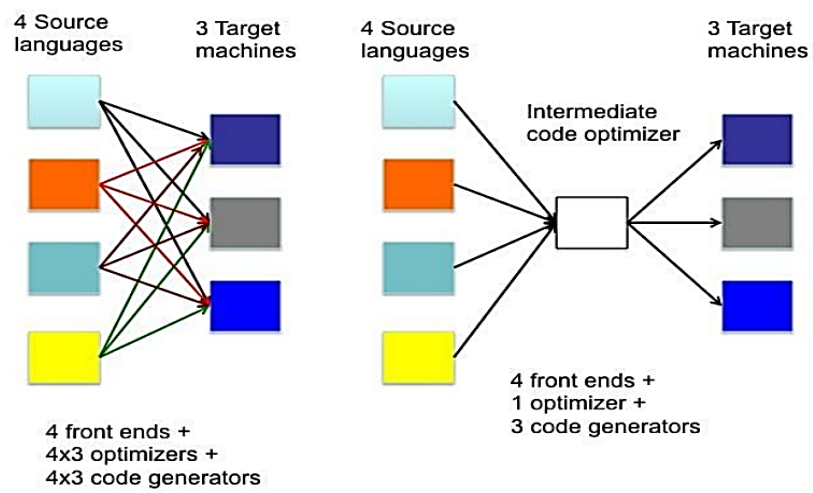
After the third phase of this project, we have successfully implemented and added a Semantic Analyzer to the C Compiler. Semantic analysis checks whether the parse tree constructed follows the rules of language. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.

INTERMEDIATE CODE GENERATION FOR C LANGUAGE

Introduction:

Compilers generate machine code, whereas interpreters interpret intermediate code. Interpreters are easier to write and can provide better error messages (symbol table is still available). However, they are at least 5 times slower than machine code generated by compilers and also require much more memory than machine code generated by compilers. While generating machine code directly from source code is possible, it entails two problems:

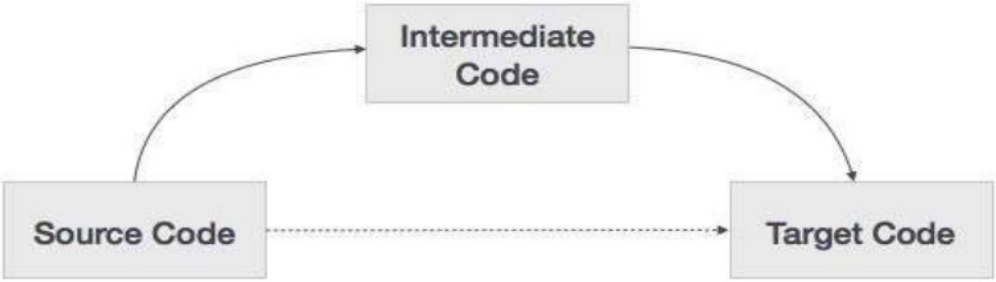
1. With m languages and n target machines, we need to write m front ends, m × n optimizers, and m × n code generators
2. The code optimizer which is one of the largest and very-difficult-to-write components of a compiler, cannot be reused.



By converting source code to an intermediate code, a machine-independent code optimizer may be written. This means just m front ends, n code generators and 1 optimizer. A source code can directly be translated into its target machine code, but we need to translate the source code into an intermediate code which is then translated to its target code. The main reasons for this are:

1. If a compiler translates the source language to its target machine language without having the option for generating intermediate code, then for each new machine, a full native compiler is required.
2. Intermediate code eliminates the need of a new full compiler for every unique machine by keeping the analysis portion same for all the compilers.
3. The second part of compiler, synthesis, is changed according to the target machine.

4. It becomes easier to apply the source code modifications to improve code performance by applying code optimization techniques on the intermediate code.



Intermediate code can be either language specific (e.g., Bytecode for Java) or language independent (three-address code).

Intermediate Representation:

The aim of Intermediate Code Generation is to translate the program into a format expected by the compiler back-end. Intermediate code must be easy to produce and easy to translate to machine code.

Why use an intermediate representation?

I. It's easy to change the source or the target language by adapting only the front-end or back-end (portability).
II. It makes optimization easier: one needs to write optimization methods only for the intermediate representation.
III. The intermediate representation can be directly interpreted. Intermediate codes can be represented in a variety of ways and they have their own benefits.

High Level IR - High-level intermediate code representation is very close to the source language itself. They can be easily generated from the source code and we can easily apply code modifications to enhance performance. But for target machine optimization, it is less preferred.

Low Level IR - This one is close to the target machine, which makes it suitable for register and memory allocation, instruction set selection, etc. It is good for machine-dependent optimizations. Some general forms of intermediate representation are:

1) Graphical IR (parse tree, abstract syntax trees, DAG. . .)
2) Linear IR (ie., non-graphical)
3) Three Address Code (TAC): instructions of the form "result=op1 operator op2"
4) Static single assignment (SSA) form: each variable is assigned once
5) Continuation-passing style (CPS): general form of IR for functional language

Three-Address Code:

Intermediate code generator receives input from its predecessor phase, semantic analyzer, in the form of an annotated syntax tree. That syntax tree then can be converted into a linear representation, e.g., postfix notation. Intermediate code tends to be machine independent code. Therefore, code generator assumes to have unlimited number of memory storage (register) to generate code. The

intermediate code generator will try to divide this expression into sub-expressions and then generate the corresponding code.

A statement involving no more than three references (two for operands and one for result) is known as three address statement. A sequence of three address statements is known as three address code. Three address statement is of the form x = y op z , here x, y, z will have address (memory location). Sometimes a statement might contain less than three references but it is still called three address statement. Example – The three address code for the expression a + b * c + d :

- T 1 = b * c
- T 2 = a + T 1
- T 3 = T 2 + d

where T 1 , T 2 , T 3 are temporary variables. A three-address code has at most three address locations to calculate the expression. A three-address code can be represented in two forms: quadruples and triples.

1) Quadruples - Each instruction in quadruples presentation is divided into four fields: operator, arg1, arg2, and result. Consider the example, a = b + c * d; It is represented below in quadruples format:

Op	arg1	arg2	result
*	c	d	r1
+	b	r1	r2
+	r2	r1	r3
=	r3		d

2) Triples - Each instruction in triples presentation has three fields: op, arg1, and arg2.The results of respective sub-expressions are denoted by the position of expression. Triples represent similarity with DAG and syntax tree. They are equivalent to DAG while representing expressions.

Op	arg1	arg2
*	c	d
+	b	0
+	(1)	0
=	(2)	

Triples face the problem of code immovability while optimization, as the results are positional and changing the order or position of an expression may cause problems.

3) Indirect Triples - This representation is an enhancement over triples representation. It uses pointers instead of position to store results. This enables the optimizers to freely re-position the sub-expression to produce an optimized code.

Implementation:

When generating IR at this level, do not need to worry about optimizing it. It's okay to generate IR that has lots of unnecessary assignments, redundant computations, etc. All the lexical analyzer part are done in lexicalAnalyzer .l which returns each token along with its value or type based on whether it is variable or constant or keyword. Syntax and

semantic check has been done in syntax Checker.y. All the productions and respective semantic actions are written here. The functions that are generating intermediate code has been written in icg{.h,.c}. The main functions in icg files are

- three Address Code* append Code (char *code) Append the three-address code provided as argument to the main set of three address code
- void backpatch (back Patch List * list, int go to L) It is to add the line number “go to L” to back Patch List
- back Patch List* merge lists (back Patch List * a, back Patch List * b) It is to merge to back patch list if it is resolved.
- back Patch List* append To Back Patch (back Patch List * list, Three Address Code * entry) It is to append a single line three address code entry to the back patch list.
- token List* append To Symbol Table (char * name, token Type, token Return Type return Type, long size, long line, char * scope, long parameter) It is used add token “name” to the symbol table along with all the properties
- void write Code (FILE * icg Out)

This is writing the three-address code generated till now into a file provided as argument

- void write Symbol Table (FILE * sym Out)

This is to write the symbol table entries till now to the file provided as argument These functions are called from respective productions in syntaxAnalyzer.y whenever needed. All the code returned are stored in a global pointer such that can write it to a file at the end. The source code for the same has been provided below.

Source Code:

Lexical Analyzer .l: This is the lex program that contains regular expressions and returns whatever tokens are required.

```
%option noyywrap
%option nomain
%option yylineno
%{
#include "icg.h"
#include "y.tab.h"
int f=0;
%}
NEWLINE (\n|\n\l|\r\n)
WHITESPACE (" "|\t)
SINGLELINE_COMMENT ("//".*{NEWLINE})
MULTILINE_COMMENT ("/*".*"*/")
PREPROCESSOR ("#".*{NEWLINE})
INTEGER (0|[1-9][0-9]*)
FLOAT (0\.0|[1-9][0-9]*\.[0-9]+)
IDENTIFIER ([a-zA-Z_][a-zA-Z0-9_]*)
LIBFUNC   "printf"|"scanf"
%%
({NEWLINE}+|{WHITESPACE}+) {}
({SINGLELINE_COMMENT}|{MULTILINE_COMMENT}) {}
{PREPROCESSOR} {}
^("+"|"-") {
                if('+' == yytext[0]) {
                        return(U_PLUS);
                } else {
                        return(U_MINUS);
                }
                f = 0;
        }
("("|")"|"{"|"}"|"<"|">"|"="|";"|","|"!")       {f = 1;return(yytext[0]);}
("/"|"*"|"%")                                   {f = 1;return(yytext[0]);}
("+"|"-") {
                if(f==1) {
                        if('+' == yytext[0]) {
                        return(U_PLUS);
                }
                else {
                        return(U_MINUS);
                }
                f = 0;
                } else {
                        return(yytext[0]);
                }
        }

"void"                                          (return(VOID);)
"int"                                           (return(INT);)
"float"                                         (return(FLOAT);)
"constant"                                      (return(CONSTANT);)
"if"                                            (return(IF);)
"else"                                          (return(ELSE);)
"return"                                        (return(RETURN);)
"do"                                            (return(DO);)
"while"                                         (return(WHILE);)
"for"                                           (return(FOR);)
"++"                                            (return(INC_OP);)
"--"                                            (return(DEC_OP);)
"=="                                            (return(EQUAL);)
"!="                                            (return(NOT_EQUAL);)
">="                                            (return(GREATER_OR_EQUAL);)
"<="                                            (return(LESS_OR_EQUAL);)
"<<"                                            (return(SHIFTLEFT);)
"&&"                                            (return(LOG_AND);)
"||"                                            (return(LOG_OR);)
{INTEGER} {
                yylval.expr.value = strdup(yytext);
                yylval.expr.cType = CONST_type;
                yylval.expr.type = INT_type;
                f = 0;
                return(CONSTANT);
        }
{FLOAT} {
                yylval.expr.value = strdup(yytext);
                yylval.expr.cType = CONST_type;
                yylval.expr.type = FLOAT_type;
                f = 0;
                return(CONSTANT);
        }
{IDENTIFIER} {
                yylval.str = malloc(strlen(yytext)*sizeof(char)+1);
                strcpy(yylval.str, yytext);
                f = 0;
                return(IDENTIFIER);
        }

%%
```

Syntax Checker.y: This is the main parser code, a yacc file which contains the declarations, rules and programs and defines the actions which should be taken for various cases.

COMPILER FOR C LANGUAGE

```
%{
#include "icg.h"
extern int yylineno;
extern char *yytext;
struct label{
        char *funcName;
        int label;
        struct label *next;
};
struct label *labelHead=NULL;

int label[10];
int funcCount=0;
int paramCount;
char icgQuad[50];
int funcLineNumber = 0;
%}
%token VOID INT FLOAT CONSTANT IDENTIFIER
%token IF ELSE RETURN DO WHILE FOR
%token INC_OP DEC_OP U_PLUS U_MINUS
%token EQUAL NOT_EQUAL GREATER_OR_EQUAL LESS_OR_EQUAL SHIFTLEFT LOG_AND LOG_OR

%right '='
%left LOG_OR
%left LOG_AND
%left '<' '>' LESS_OR_EQUAL GREATER_OR_EQUAL
%left EQUAL NOT_EQUAL
%left SHIFTLEFT
%left '+' '-'
%left '*' '/' '%'
%right U_PLUS U_MINUS '!'
%left INC_OP DEC_OP
%union
{
    char                *str;
    int                 integer;
    float               real;
    int                 type;
        struct
        {
            char                        *value;
            int                         type;
            int                         cType;
            struct BackpatchList*       trueList;
            struct BackpatchList*       falseList;
        } expr;
        struct
        {
          struct BackpatchList*         nextList;
        } stmt;
        struct
        {
          int                           quad;
          struct BackpatchList*         nextList;
        } mark;
        struct
        {
            int                         count;
            struct tokenList *  queue;
        } exp_list;
}

%type <str> id IDENTIFIER
%type <type> declaration var_type
%type <expr> expression assignment CONSTANT
%type <stmt> statement statement_list matched_statement unmatched_statement program function_body function
%type <exp_list> exp_list
%type <mark> marker jump_marker
%start program_head
%%
program_head
    : program
        {
            tokenList * mainFunc = getSymbol("main");
            if(mainFunc == NULL){
                printf("ERROR: Main function not found!\n");
                yyerror();
            }
            backpatch($1.nextList,mainFunc->line+1);
        }
    ;
program
    : jump_marker function
        {
            $$.nextList = $1.nextList;
            backpatch($2.nextList, nextquad());
        }
    | program function
        {
            $$.nextList = $1.nextList;
            backpatch($2.nextList, nextquad());
        }
    ;
function
    : var_type id '(' parameter_list ')' ';'
        {
            if(labelHead==NULL){
                struct label *temp=(struct label *)malloc(sizeof(label));
                temp->name=strdup(id);
                temp->label=funcCount;
                temp->next=NULL;
            }
            else{
                struct label *temp=(struct label *)malloc(sizeof(label));
                temp->name=strdup(id);
                temp->label=funcCount;
                temp->next=NULL;
                struct label *p=labelHead;
                while(p->next!=NULL){
                        p=p->next;
                }
                p->next=temp;
            }

            funcCount++;
            addFunctionPrototype($2, paramCount, $1);
            paramCount = 0;
            $$.nextList = NULL;
        }
    | var_type id '(' parameter_list ')' function_body
        {

            addFunction($2, paramCount, $1, funcLineNumber);
            paramCount = 0;
            funcLineNumber = nextquad();
            $$.nextList = $6.nextList;
        }
    ;
function_body
    : '{' statement_list  '}'
        {
            $$.nextList = $2.nextList;
        }
    | '{' declaration_list statement_list '}'
        {
```

COMPILER FOR C LANGUAGE

```
            $$.nextList = $3.nextList;
        }
    ;
declaration_list
    : declaration ';'
    | declaration_list declaration ';'
    ;
declaration
    : INT id
        {
            $$ = INT_type;
            addSymbolToQueue($2, INT_type, 0);
        }
    | FLOAT id
        {
            $$ = FLOAT_type;
            addSymbolToQueue($2, FLOAT_type, 0);
        }
    | declaration ',' id
        {
            if(INT_type == $1) {
                addSymbolToQueue($3, INT_type, 0);
            } else if(FLOAT_type == $1) {
                addSymbolToQueue($3, FLOAT_type, 0);
            }
        }
    ;
parameter_list
    : INT id
        {
            paramCount++;
            addSymbolToQueue($2, INT_type, paramCount);
        }
    | FLOAT id
        {
            paramCount++;
            addSymbolToQueue($2, FLOAT_type, paramCount);
        }
    | parameter_list ',' INT id
        {
            paramCount++;
            addSymbolToQueue($4, INT_type, paramCount);
        }
    | parameter_list ',' FLOAT id
        {
            paramCount++;
            addSymbolToQueue($4, FLOAT_type, paramCount);
        }
    | VOID
    |
    ;
var_type
    : VOID
        {
            $$ = Return_VOID;
        }
    | INT
        {
            $$ = Return_INT;
        }
    | FLOAT
        {
            $$ = Return_FLOAT;
        }
    ;
statement_list
    : statement
        {
            $$.nextList = $1.nextList;
        }
    | statement_list marker statement
        {
            backpatch($1.nextList,$2.quad);
            $$.nextList = $3.nextList;
        }
    ;
statement
    : matched_statement
        {
            $$.nextList = $1.nextList;
        }
    | unmatched_statement
        {
            $$.nextList = $1.nextList;
        }
    ;
matched_statement
    : IF '(' assignment ')' marker matched_statement jump_marker ELSE marker matched_statement
        {
            backpatch($3.trueList,$5.quad);
            backpatch($3.falseList,$9.quad);
            $$.nextList = mergelists($7.nextList,$10.nextList);
            $$.nextList = mergelists($$.nextList,$6.nextList);
        }
    | assignment ';'
        {
            $$.nextList = NULL;
        }
    | RETURN ';'
        {
            $$.nextList = NULL;
            sprintf(icgQuad,"RETURN");
            appendCode(icgQuad);
        }
    | RETURN assignment ';'
        {
            $$.nextList = NULL;
            sprintf(icgQuad,"RETURN %s",$2.value);
            appendCode(icgQuad);
        }
    | WHILE marker '(' assignment ')' marker matched_statement jump_marker
        {
            backpatch($4.trueList,$6.quad);
            $$.nextList = $4.falseList;
            backpatch($7.nextList,$2.quad);
            backpatch($8.nextList,$2.quad);
        }
    | DO marker statement WHILE '(' marker assignment ')' ';'
        {
            backpatch($3.nextList,$6.quad);
            backpatch($7.trueList,$2.quad);
            $$.nextList = $7.falseList;
        }
    | FOR '(' assignment ';' marker assignment ';' marker assignment jump_marker ')' marker matched_statement jump_marker
        {
            if(BOOL_type == $3.type || BOOL_type == $9.type) {
                printf("error, no boolean statements allowed as 1st or 3rd assignment in for loop\n");
                yyerror();
            }
            if(BOOL_type != $6.type) {
                printf("error, 2nd argument of for loop must be boolean\n");
                yyerror();
            }
            backpatch($3.trueList, $5.quad);
            backpatch($13.nextList, $8.quad);
```

COMPILER FOR C LANGUAGE

```
            backpatch($14.nextList, $8.quad);
            $$.nextList = $6.falseList;
            backpatch($6.trueList, $12.quad);
            backpatch($9.trueList, $5.quad);
            backpatch($10.nextList, $5.quad);

        }
    | '{' statement_list '}'
        {
            $$.nextList = $2.nextList;
        }
    | '{' '}'
        {
            $$.nextList = NULL;
        }
    ;
unmatched_statement
    : IF '(' assignment ')' marker statement
        {
            backpatch($3.trueList,$5.quad);
            $$.nextList = mergelists($3.falseList,$6.nextList);
        }
    | WHILE marker '(' assignment ')' marker unmatched_statement jump_marker
        {
            backpatch($4.trueList,$6.quad);
            $$.nextList = $4.falseList;
            backpatch($7.nextList,$2.quad);
            backpatch($8.nextList,$2.quad);
        }
    | FOR '(' assignment ';' marker assignment ';' marker assignment jump_marker ')' marker unmatched_statement jump_marker
        {
            if(BOOL_type == $3.type || BOOL_type == $9.type) {
                printf("error, no boolean statements allowed as 1st or 3rd assignment in for loop\n");
                yyerror();
            }
            if(BOOL_type != $6.type) {
                printf("error, 2nd argument of for loop must be boolean\n");
                yyerror();
            }
            backpatch($3.trueList, $5.quad);
            backpatch($13.nextList, $8.quad);
            backpatch($14.nextList, $8.quad);
            $$.nextList = $6.falseList;
            backpatch($6.trueList, $12.quad);
            backpatch($9.trueList, $5.quad);
            backpatch($10.nextList, $5.quad);
        }
    | IF '(' assignment ')' marker matched_statement jump_marker ELSE marker unmatched_statement
        {
            backpatch($3.trueList,$5.quad);
            backpatch($3.falseList,$9.quad);
            $$.nextList = mergelists($7.nextList,$10.nextList);
            $$.nextList = mergelists($$.nextList,$6.nextList);
        }
    ;
assignment
    : expression
        {
            $$=$1;
        }
    | id '=' expression
        {
            int destType = getSymbolType($1);
                if(destType == 0){
                        printf("ERROR: Not in scope");
                }
            if(destType != $3.type) {
                printf("Type error on line: %d\n", yylineno);
                yyerror();
            }

            sprintf(icgQuad,"%s := %s",$1,$3.value);
            appendCode(icgQuad);
            $$.type = destType;
            $$.trueList = $3.trueList;
            $$.cType = VAR_type;
            $$.value = $1;
        }
    ;
expression
    : INC_OP expression
        {
            if($2.type != INT_type){
                    printf("ERROR: Increment not allowed for types different than Integer.\n");
                    yyerror();
            }
            //Create a variable if needed
            if($2.cType != VAR_type){
                    char *var = nextIntVar();
                    sprintf(icgQuad,"%s := %s",var,$2.value);
                    appendCode(icgQuad);
                    free($2.value);
                    $2.value = var;
                    $2.type = INT_type;
                    $2.cType = VAR_type;
            }
            sprintf(icgQuad,"%s := %s + 1",$2.value,$2.value);
            appendCode(icgQuad);
            //Set the attributes
            $$ = $2;
            $$.trueList = NULL;
            $$.falseList = NULL;
        }
    | DEC_OP expression
        {
            if($2.type != INT_type){
                    printf("ERROR: Decrement not allowed for types different than Integer.\n");
                    yyerror();
            }
            //Create a variable if needed
            if($2.cType != VAR_type){
                    char *var = nextIntVar();
                    sprintf(icgQuad,"%s := %s",var,$2.value);
                appendCode(icgQuad);
                    free($2.value);
                    $2.value = var;
                    $2.type = INT_type;
                    $2.cType = VAR_type;
            }
            sprintf(icgQuad,"%s := %s - 1",$2.value,$2.value);
            appendCode(icgQuad);
            //Set the attributes
            $$ = $2;
            $$.trueList = NULL;
            $$.falseList = NULL;
        }
    | expression LOG_OR marker expression
        {
            if(BOOL_type != $1.type) {
                sprintf(icgQuad, "IF (%s <> 0) GOTO", $1.value);
                $1.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
                sprintf(icgQuad, "GOTO");
                $1.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            }
            if(BOOL_type != $4.type) {
```

COMPILER FOR C LANGUAGE

```
                sprintf(icgQuad, "IF (%s <> 0) GOTO", $4.value);
                $4.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
                sprintf(icgQuad, "GOTO");
                $4.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            }
            $$.trueList = mergelists($1.trueList, $4.trueList);
            backpatch($1.falseList, $3.quad);
            $$.falseList = $4.falseList;
            $$.type = BOOL_type;
        }
    | expression LOG_AND marker expression
        {
            if(BOOL_type != $1.type) {
                sprintf(icgQuad, "IF (%s <> 0) GOTO", $1.value);
                $1.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
                sprintf(icgQuad, "GOTO");
                $1.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            }
            if(BOOL_type != $4.type) {
                sprintf(icgQuad, "IF (%s <> 0) GOTO", $4.value);
                $4.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
                sprintf(icgQuad, "GOTO");
                $4.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            }
            $$.falseList = mergelists($1.falseList, $4.falseList);
            backpatch($1.trueList, $3.quad);
            $$.trueList = $4.trueList;
            $$.type = BOOL_type;
         }
    | expression NOT_EQUAL expression
        {


            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
                yyerror();
            }
            sprintf(icgQuad,"IF (%s <> %s) GOTO",$1.value,$3.value);
            $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad,"GOTO");
            $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            $$.value = "TrueFalse Only!";
            $$.type = BOOL_type;
            $$.cType = NONE_type;


        }
    | expression EQUAL expression
        {
            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
                yyerror();
            }
            sprintf(icgQuad,"IF (%s = %s) GOTO",$1.value,$3.value);
            $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad,"GOTO");
            $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
        if(BOOL_type == $1.type) {
            $$.trueList = mergelists($$.trueList, $1.trueList);
            $$.falseList = mergelists($$.falseList, $1.falseList);
        }
        if(BOOL_type == $3.type) {
            $$.trueList = mergelists($$.trueList, $3.trueList);
            $$.falseList = mergelists($$.falseList, $3.falseList);
        }
            $$.value = "TrueFalse Only!";
            $$.type = BOOL_type;
            $$.cType = NONE_type;
        }
    | expression GREATER_OR_EQUAL expression
        {
            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
                yyerror();
            }
            sprintf(icgQuad,"IF (%s >= %s) GOTO",$1.value,$3.value);
            $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad,"GOTO");
            $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            $$.value = "TrueFalse Only!";
            $$.type = BOOL_type;
            $$.cType = NONE_type;
        }
    | expression LESS_OR_EQUAL expression
        {
            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
                yyerror();
            }
            sprintf(icgQuad,"IF (%s <= %s) GOTO",$1.value,$3.value);
            $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad,"GOTO");
            $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            $$.value = "TrueFalse Only!";
            $$.type = BOOL_type;
            $$.cType = NONE_type;
        }
    | expression '>' expression
        {
            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
                yyerror();
            }
            sprintf(icgQuad,"IF (%s > %s) GOTO",$1.value,$3.value);
            $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad,"GOTO");
            $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            $$.value = "TrueFalse Only!";
            $$.type = BOOL_type;
            $$.cType = NONE_type;
        }
    | expression '<' expression
        {
            if($1.type != INT_type && $1.type != FLOAT_type){
                printf("ERROR: Only Integer, Float and Bool values allowed in comparsions.\n");
                yyerror();
            }
            sprintf(icgQuad,"IF (%s < %s) GOTO",$1.value,$3.value);
            $$.trueList = appendToBackPatch(NULL, appendCode(icgQuad));
            sprintf(icgQuad,"GOTO");
            $$.falseList = appendToBackPatch(NULL, appendCode(icgQuad));
            $$.value = "TrueFalse Only!";
            $$.type = BOOL_type;
            $$.cType = NONE_type;
        }
    | expression '+' expression
        {
            if($1.type != INT_type && $1.type!= FLOAT_type &&  $3.type != INT_type && $3.type != FLOAT_type){
                printf("ERROR: Only integer and float values allowed when adding numbers.\n");
                yyerror();
            }
            int type = 0;
            if($1.type == $3.type){
                type = $1.type;
            }
            else{
                type = FLOAT_type;
```

COMPILER FOR C LANGUAGE

icg.c

```c
#include "icg.h"
#include "y.tab.h"

tokenList  *symPtr = NULL;
tokenList  *bufferPtr = NULL;
threeAddressCode   *codePtr = NULL;

extern int yylineno;
extern int yyin;

int globalOffset = 0, tempIntCounter = 0,tempFloatCounter= 0,currentLine = -1;

void yyerror() {
    printf("ERROR: %d : Syntactical Error\n",yylineno);
    exit(1);
}

int main(int argc, char **argv) {
    yyin = fopen(argv[1],"r");
    yyparse();
    printf("\n\n\t Parsing Completed\n");
    printf("Three Address Code and Symbol table are generated\n\n");
    FILE *icgPtr= fopen("threeAddressCode.txt","w");
    writeCode(icgPtr);
    FILE *symPtr = fopen("symbolTable.txt", "w");
    writeSymbolTable(symPtr);
    fclose(icgPtr);
    fclose(symPtr);
    printf("\n");
    return(0);
}

static const char* returnTypeToString(tokenReturnType type)
{
    switch(type)
    {
        case Return_VOID:
            return "None";

        case Return_INT:
            return "Int";

        case Return_FLOAT:
            return "Real";
    }
}

static const char* dataTypeToString(tokenType type)
{
    switch(type)
    {

        case INT_type:
            return "Int";

        case FLOAT_type:
            return "Real";

         case BOOL_type:
          return "Bool";

        case FUNC:
            return "Func";
        case PROTO:
            return "Proto";

        case MAIN:
            return "Main";
    }
}

threeAddressCode* appendCode(char *code){

        threeAddressCode *temp = malloc(sizeof(threeAddressCode ));

        temp->code=(char *)malloc(sizeof(char)*strlen(code));
        strcpy(temp->code,code);
        temp->gotoLine = -1;
        temp->next = NULL;


        if(codePtr == NULL){
                codePtr=temp;
        }
        else{
                threeAddressCode *p=codePtr;
                while(p->next!=NULL){
                        p=p->next;
                }
                p->next=temp;
        while(cur != NULL){
                if((cur->type == FUNC || cur->type == PROTO) && 0 == strcmp(name,cur->name)){
                        break;
                }
        }
        if(cur == NULL)
                return -1;
        int foundParams = 0;
        cur = symPtr;
        do{
                while(cur != NULL){
                        if(cur->scope != NULL && 0 == strcmp(name,cur->scope) && cur->parameter != 0){
                                break;
                        }
                        cur = cur->next;
                }
                if(cur == NULL || queue == NULL){
                        if(parameterCount == 0 && foundParams == 0)
                                return 0;
                        else
                                return -2;
                }
                else if(cur->type != queue->type){
                        return -3;
                }
                foundParams++;
                sprintf(buffer,"PARAM %s",queue->name);
                appendCode(buffer);
                cur = cur->next;
                tokenList  *temp = queue;
                queue = queue->next;
                free(temp);
        }while(foundParams != parameterCount);
        return 0;
}
```

COMPILER FOR C LANGUAGE

icg.h

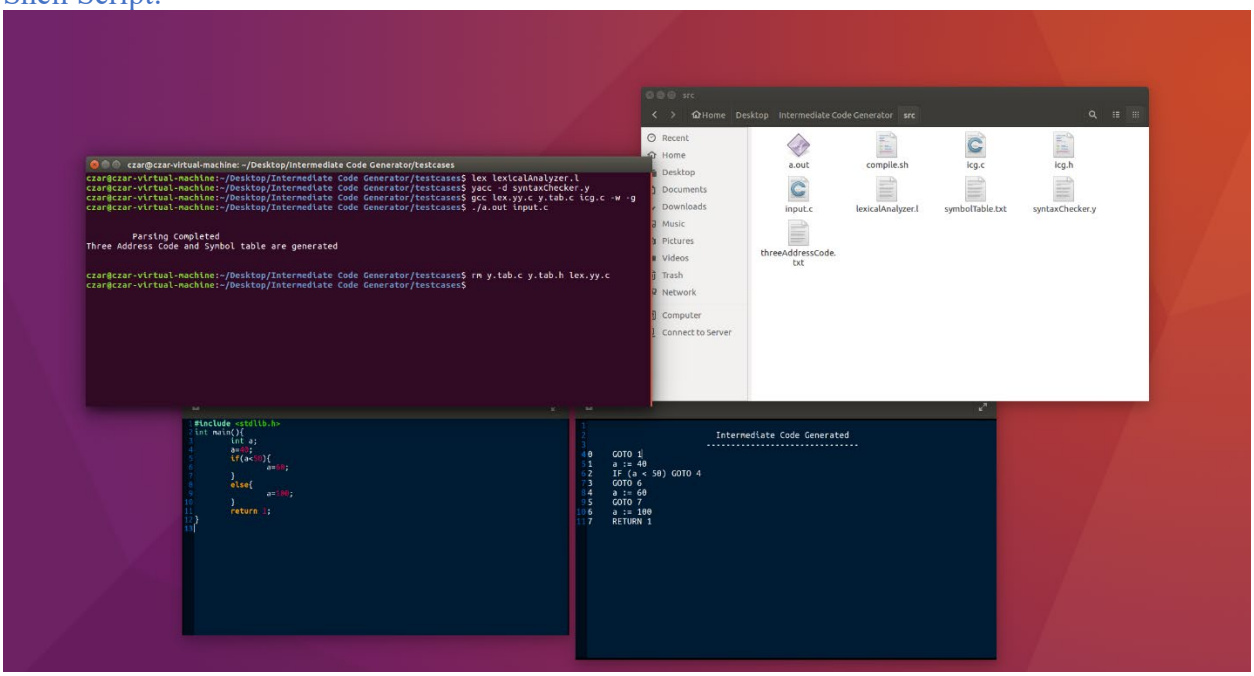
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
#include <malloc.h>
#define PARENT_NONE NULL

typedef enum
{
     BOOL_type,
    INT_type,
    FLOAT_type,
     FUNC,
     MAIN,
     PROTO,
} tokenType;
typedef enum
{
    Return_VOID,
    Return_INT,
    Return_FLOAT

} tokenReturnType;
typedef enum
{
        CONST_type,
        VAR_type,
        NONE_type

} tokenConstType;

struct tokenList
{
    char *name;
    tokenType type;
    tokenReturnType returnType;
    long size;
    char *scope;
    long line;
    long parameter;

    struct tokenList *next;

};
typedef struct tokenList tokenList;

struct threeAddressCode
{
    char *code;
    int gotoLine;

    struct threeAddressCode *next;

};
typedef struct threeAddressCode threeAddressCode;

struct backPatchList
{
    threeAddressCode   *entry;
    struct backPatchList  *next;

};
typedef struct backPatchList   backPatchList ;

threeAddressCode* appendCode(char *code);
void backpatch(backPatchList * list, int gotoL);

backPatchList* mergelists(backPatchList  * a, backPatchList  * b);

backPatchList* appendToBackPatch(backPatchList  * list, threeAddressCode  * entry);

tokenList* appendToSymbolTable(char *name,tokenType type,tokenReturnType returnType,long size,long line,char *scope,long parameter);
void writeCode(FILE *icgOut);
void writeSymbolTable(FILE *symOut);

tokenList * addSymbolToParameterQueue(tokenList * queue, char *name, tokenType type);

int checkAndGenerateParams(tokenList * queue, char* name ,int parameterCount);

int getFunctionType(char *name);
int getSymbolType(char *name);
char* nextFloatVar();
char* nextIntVar();
char* nextBoolVar();
int nextquad();

void addFunction(char *name, unsigned int parameter_count, tokenReturnType  ret_type, int line);
tokenList * lookup(char *name);
extern tokenList *symbolTable;
```

Execution Of the Code:

The following is a shell script to automate the compilation and execution of the mini compiler 24 consist of all phases. There is sample input source code given. Execution of the same is done and output is shown as sample output and respective symbol table is generated.

COMPILER FOR C LANGUAGE

Shell Script:



Symbol table:

```
                    Symbol Table
                ------------------

  Name          DataType      ReturnType      Offset        Scope
-------------------------------------------------------------------------
a                 Int           None            0           main
main              Main          Int             4           None
```

Sample Three-address Code:

```
                    Intermediate Code Generated
                ------------------------------
0     GOTO 1
1     a := 40
2     IF (a < 50) GOTO 4
3     GOTO 6
4     a := 60
5     GOTO 7
6     a := 100
7     RETURN 1
```

COMPILER FOR C LANGUAGE

Test Cases:

Different test cases are added to check implementation of ICG phase. Source code for the 6 tests cases are added as screenshot below along with their three-address code.

Case1.c

```c
//Testcase generate simple expresion statements
#include<stdio.h>
int main(){
        int a;
        float t;
        a=10;
        t=t*10;
        return 1;
}
```

```
                        Intermediate Code Generated
                      -------------------------------
0     GOTO 1
1     a := 10
2     Tf_1 := t * 10
3     t := Tf_1
4     RETURN 1
```

Case2.c

```c
//testcase to generate control statments
#include<stdio.h>
int main(){
        int a;
        float t;
        a=20;
        if(a<30){
                t=3.0;
        }
        else{
                t=4.0;
        }
        return 2;
}
```

```
                        Intermediate Code Generated
                      -------------------------------
0     GOTO 1
1     a := 20
2     IF (a < 30) GOTO 4
3     GOTO 6
4     t := 3.0
5     GOTO 7
6     t := 4.0
7     RETURN 2
```

COMPILER FOR C LANGUAGE

Case3.c

```c
//testcase to check looping statements
#include<stdio.h>
int main(){
        int i,s;
        s=0;
        for(i=0;i<10;i=i+2){
                s=s+i;
                while(s>30)
                {
                s=10;
                if(s<10)
                {

                                }
                                else
                                {
                                }
                }
        }
        return 2;
}
```

```
                        Intermediate Code Generated
                      -------------------------------
0     GOTO 1
1     s := 0
2     i := 0
3     IF (i < 10) GOTO 8
4     GOTO 18
5     Ti_1 := i + 2
6     i := Ti_1
7     GOTO 3
8     Ti_2 := s + i
9     s := Ti_2
10    IF (s > 30) GOTO 12
11    GOTO 5
12    s := 10
13    IF (s < 10) GOTO 15
14    GOTO 16
15    GOTO 10
16    GOTO 10
17    GOTO 5
18    RETURN 2
```

Case4.c

```c
//testcase to generate different functions code
#include<stdio.h>
int func1(){
        int s;
        s=56;
}
int func(int x,int y){
        x=x+y;
        func1();
}

int main(){
        int a;
        a=20;
        func1();
        return 1;
}
```

COMPILER FOR C LANGUAGE

```
                        Intermediate Code Generated
                      -------------------------------
0     GOTO 3
1     s := 56
2     a := 20
3     RETURN 1
```

Case5.c

```
//testcase to check main function errors
#include<stdio.h>
int main();
int main(){
        int y;
        y=20;
}
```

```
                        Intermediate Code Generated
                      -------------------------------
0     GOTO 1
1     y := 20
```

Conclusion:

Intermediate Code Generation (ICG) is the fourth phase of a compiler. It is the final phase of the front end of a compiler. It generates intermediate code, that is a form which can be readily executed by machine. Intermediate code is converted to machine language using the last two phases which are platform dependent. We can take the intermediate code from the already existing compiler and build the last two parts. After this, the intermediate code is ready and it is then sent to the Code Optimizer phase which transforms the code so that it consumes fewer resources and produces more speed.