Compiler for C Language

Automata and Compiler Design (IT250) Report

Submitted in partial fulfillment of the requirements for the degree of

BACHELOR OF TECHNOLOGY

In

INFORMATION TECHNOLOGY

by

NITHIN S 221IT085

JAY CHAVAN 221IT020

AYUSH KUMAR 221IT015

DEPARTMENT OF INFORMATION TECHNOLOGY

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA

SURATHKAL, MANGALORE -575025

March, 2024

D E C L A R A T I O N

We hereby *declare* that the *ACD Project Report* entitled **"C Compiler Design"** which is being submitted to the National Institute of Technology Karnataka Surathkal, in partial fulfillment of the requirements for the award of the Degree of Bachelor of Technology in the Department of Information Technology, is a ***bonafide report of the work carried out by us.*** The material contained in this project report has not been submitted to any University or Institution for the award of any degree.

Nithin S	Ayush Kumar	Jay Chavan
Signature Department of IT	Signature Department of IT	Signature Department of IT

CERTIFICATE

This is to certify that the Seminar entitled **"C Compiler Phases"** has been presented by Nithin S (221IT085), Ayush Kumar (221IT015) & Jay Chavan (221IT020), students of IV semester B.Tech.(I.T), Department of Information Technology, National Institute of Technology Karnataka, Surathkal, on 25 March, 2024, during the even semester of the academic year 2023 - 2024, in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in Information Technology.

Examiner-1 Name
Signature of the Examiner-1 with Date

Examiner-2 Name
Signature of the Examiner-2 with Date

Place :

Date:

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

Our aim is to design a compiler for C Programming Language and present phase by phase output of the Compiler . We are trying to implement the following features

Modules

1. Lexical Analyzer
2. Syntax Analyzer
3. Semantic Analyzer
4. Intermediate Code Generator
5. Code Optimizer
6. Target Code Generator

1. 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

1.2 Syntax Analyzer

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

1.3 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

1.4 Intermediate Code Generator

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

It receives input in the form of an 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 generators assume an unlimited number of memory storage (registers) to generate code.This three-address code can be converted to MIPS assembly code .

1.5 Code Optimizer

In this phase,the machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means faster, but other objectives may be desired, such as shorter code, or target code that consumes less power.

- The optimizer can convert the integer 60 to floating point at compile time, just once.
- Different compilers vary greatly in how much they optimize code.
- Simple optimizations can greatly enhance program running time without much impact on compilation speed.

1.6 Target Code Generator

The final phase in our compiler model is the code generator. It takes as input the intermediate representation (IR) produced by the front end of the compiler, along with relevant symbol table information, and produces semantically equivalent target programs.

- Code generators must ensure the target program accurately preserves the semantic meaning of the source program.
- The target program must be of high quality, efficiently using the target machine's resources.
- Mathematically, generating an optimal target program from a given source program is an undecidable problem.
- The intermediate representation (IR) may undergo multiple passes during the optimization and code generation phases.

CHAPTER 2 : OBJECTIVE

The objective of designing a compiler for the C Programming Language is to systematically translate C code into an efficient, machine-understandable format through several phases. Initially, the Lexical Analyzer identifies basic language constructs like keywords and operators, handling errors such as invalid characters. The Syntax Analyzer then ensures the grammatical correctness of the code, focusing on expressions, control statements, and syntax errors like missing semicolons. The Semantic Analyzer adds a layer of context, checking for scope, type mismatches, and array bounds to ensure logical consistency. The Intermediate Code Generator converts validated code into a machine-independent format, optimizing for simplicity and efficiency. The Code Optimizer enhances this intermediate code by improving performance and reducing resource consumption without altering the program's semantics. Finally, the Target Code Generator produces the final machine code, optimizing for the target hardware's specifics while ensuring the output remains true to the original program's intent. This multi-phase process aims to create efficient, optimized, and error-free machine code from high-level C language input, balancing speed, resource use, and maintainability.

CHAPTER 3 : Phases of Compiler

3.1 Lexical Analyzer

3.1.1 Methodology

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 units, 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 a token as output.

- **Tokens** : It is a valid sequence of characters which are given by lexeme. In a programming language, keywords, constants, identifiers, numbers, operators and punctuation symbols are possible tokens to be identified.
- **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.
- **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.

Role of a Lexical Analyzer

A lexical analyzer performs the following tasks :

- Reads the source program, scans the input characters, groups them into lexemes and produces the token as output.
- Enter the identified token into the symbol table.
- Strips out white spaces and comments from the 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 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 parentheses, an error message is generated.

4) Invalid Identifier : If the entered identifier does not match the identifier forming rules, an error message is displayed.

3.1.2 Implementation

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:

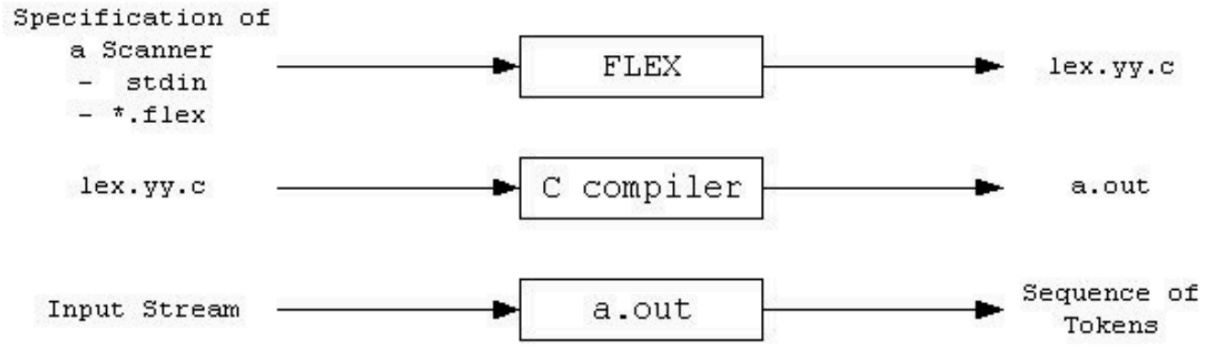


fig 3.1.1 Block diagram of lexical analyzer

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.

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 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)".

*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 unindented 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 unindented 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 unindented comment (i.e., a line beginning with "/*") is also copied verbatim to the output up to the next "*/".

3.1.3 Result

lexAnalyzer.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.

CODE :

```
%{
#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;

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"
datatype "int"|"char"|"void"
sign "signed"|"unsigned"
modifiers "long"|"short"
relational >|<|<=|>=|!=|==
logical \&|\^|\~
arithmatic \+|\-|\*|\/|\%
puncuator \(|\)|\[|\]|\;|\,|\:|\.
assignment =
quote \'|\"|\\
whitespace [ \t]+
newline "\n"
singlelinecomment (\/\/.*)
multilinecommentstart (\/\*)
multilinecommentend (\*\/)
identifier ({letter}({letter}|{digit})*)|"_"({letter}|{digit})+

%x DETECT_COMMENT

%%

^#([-a-zA-Z0-9.]|{relational}|{whitespace})* insertToTable(yytext,'d'); //preprocessor directive rule

{keyword} insertToTable(yytext,'k');
{sign}?{whitespace}{modifiers}?{whitespace}{datatype} insertToTable(yytext,'k'); //keyword rule

^{datatype}{whitespace}*{identifier}\(.*\)  {   int i,j=0;char temp1[50]={'\0'}, temp2[50]={'\0'};

for(i=0;yytext[i]!=' ';i++)
                                                                                {

temp1[i] = yytext[i];
                                                                                }


insertToTable(temp1,'k');

for(;yytext[i]!='(';i++){

temp2[j]=yytext[i];

j++;
                                                                                }
```

```
"{"  {  cBrac++;
        insertToTable(yytext,'p');
         }

"}"  {  cBrac--;
        insertToTable(yytext,'p');
         }

{singlelinecomment} {storeSingleLineComment(yytext);}

{multilinecommentstart}  {
                BEGIN(DETECT_COMMENT);
                nc++;
                cLine++;
                displayComment("\n\t");
            }

<DETECT_COMMENT>{multilinecommentstart} {
                            nc++;
                            if(nc>1)
                            {
                                                                printf("%s : %d : Nested
Comment\n",inputFile,lineNumber);
                                flag = 1;
                            }
                        }

<DETECT_COMMENT>{multilinecommentend} {
                            if(nc>0)
                                nc--;
                            else
                                                            printf("%s : %d : */ found before /
*\n",inputFile,lineNumber);

                            if(nc==0)
                                BEGIN(INITIAL);
                    }

<DETECT_COMMENT>\n   {
                 cLine++;
                 lineNumber++;
                 displayComment("\n");
                }

<DETECT_COMMENT>. {displayComment(yytext);}


%%

int main(int argc,char **argv)
{
        comment = (char*)malloc(100*sizeof(char));
```

```
int main(int argc,char **argv)
{
        comment = (char*)malloc(100*sizeof(char));
        yyin=fopen(argv[1],"r");
        inputFile=argv[1];

        yyout=fopen("symbolTable.txt","w"); // File to write all token in source program
        fprintf(yyout,"\n Table:\n \t\tLexeme\t\t\tToken\t\t\t\tAttribute Value\t\t\tLine Number\n");

        // Initialize symbol and constants file pointers to NULL


        yylex();

        if(nc!=0)
                printf("%s : %d : Comment Does Not End\n",inputFile,lineNumber);

        if(cBrac!=0)
                printf("%s : %d : Unbalanced Parenthesis\n",inputFile,lineNumber);

        fprintf(yyout,"\n");
        if(flag==1)
        {
                cLine = 0;
                fprintf(yyout,"\n\nComment (%d lines):\n",cLine);
                printf("%s : %d : Nested Comment\n",inputFile,lineNumber);
        }
        else
        {
                int i;
                fprintf(yyout,"\n\nMultiLineComment (%d lines):",cLine);
                fputs(comment,yyout);
                fprintf(yyout,"\n\nSingleLineComment :\n");
                fputs(s_comment,yyout);
        }

        fclose(yyout);

        // Close symbol and constants files if they were opened

}



void storeSingleLineComment(char *yytext)
{
        int len = strlen(yytext);
        int i, j=0;
        char *temp;
        temp = (char*)malloc((len+1)*sizeof(char));
        for(i=2;yytext[i]!='\0';i++)
        {
```

```
    case 'p': strcpy(token,"Punctuator");break;

    case 'o': strcpy(token,"Arithmetic Op");break;

    case 'c': strcpy(token,"Integer Constant");break;

    case 'f': strcpy(token,"Float Constant");break;

    case 'z': strcpy(token,"Character Constant");break;

    case 'e': strcpy(token,"Assignment Op");break;

    case 'l': strcpy(token,"Logical Op");break;

    case 's': strcpy(token,"String Literal");break;
}

if(nc<=0)
{
        current = head;
        for(i=0;i<var;i++)
        {
            if(strcmp(current->tname,yytext)==0)
                    {
                            break;
                    }
                    current = current->next;
        }

        if(i==var)
        {
            temp = (struct Node *)malloc(sizeof(struct Node));
            temp->av = i;
            temp->tname = (char *)malloc(sizeof(char)*(l1+1));
            strcpy(temp->tname,yytext);
            temp->next = NULL;

            if(head==NULL)
            {
                    head = temp;
            }
            else
            {
                    current = head;
                    while(current->next!=NULL)
                    {
                            current = current->next;
                    }
                    current->next = temp;
            }
```

Output

Test Case 1

```c
// test case to check loop statements

int main(){

        int i, a, b;
        int nume=3.45;
        for(i = 0;  i < 10; i++){
                a=i;
        }
        i=1;
}
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/1_Lexical_Analyzer$ lex lexAnalyzer.l
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/1_Lexical_Analyzer$ cc lex.yy.c
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/1_Lexical_Analyzer$ ./a.out < code.c
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/1_Lexical_Analyzer$
```

Symbol table generated in .txt file so that we can store the table and use it in other phases

```
Table:
              Lexeme                  Token                  Attribute Value              Line Number

               int                       Keyword                        0                        3
              main                     Procedure                       1                        3
                 {                    Punctuator                       2                        3
               int                       Keyword                        0                        5
                 i                   Identifier                        3                        5
                 ,                   Punctuator                        4                        5
                 a                   Identifier                        5                        5
                 ,                   Punctuator                        4                        5
                 b                   Identifier                        6                        5
                 ;                   Punctuator                        7                        5
               int                       Keyword                        0                        6
              nume                     Identifier                       8                        6
                 =                  Assignment Op                      9                        6
              3.45                 Float Constant                     10                        6
                 ;                   Punctuator                        7                        6
               for                       Keyword                       11                        7
                 (                   Punctuator                       12                        7
                 i                   Identifier                        3                        7
                 =                  Assignment Op                      9                        7
                 0                Integer Constant                     13                        7
                 ;                   Punctuator                        7                        7
                 i                   Identifier                        3                        7
                 <                  Relational Op                      14                        7
                10                Integer Constant                     15                        7
                 ;                   Punctuator                        7                        7
                 i                   Identifier                        3                        7
                 +                  Arithmetic Op                      16                        7
                 +                  Arithmetic Op                      16                        7
                 )                   Punctuator                       17                        7
                 {                   Punctuator                        2                        7
                 a                   Identifier                        5                        8
                 =                  Assignment Op                      9                        8
                 i                   Identifier                        3                        8
                 ;                   Punctuator                        7                        8
                 }                   Punctuator                       18                        9
                 i                   Identifier                        3                       10
                 =                  Assignment Op                      9                       10
                 1                Integer Constant                     19                       10
                 ;                   Punctuator                        7                       10
                 }                   Punctuator                       18                       11


MultiLineComment (0 lines):

SingleLineComment :
 test case to check loop statements
```

fig. 3.1.2 Output of Test Case1

Test Case 2

```c
//Test case for if else
void main()
{
        int a=1,b=2,c=10;

        if(a>b){
                printf("\nInside if");
        }
        else{
                printf("\nInside else");
        }

}
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/1_Lexical_Analyzer$ ./a.out < TestCases/ifelse.c
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/1_Lexical_Analyzer$
```

Table:

Lexeme	Token	Attribute Value	Line Number
void	Keyword	0	2
main	Procedure	1	2
{	Punctuator	2	3
int	Keyword	3	4
a	Identifier	4	4
=	Assignment Op	5	4
1	Integer Constant	6	4
,	Punctuator	7	4
b	Identifier	8	4
=	Assignment Op	5	4
2	Integer Constant	9	4
,	Punctuator	7	4
c	Identifier	10	4
=	Assignment Op	5	4
10	Integer Constant	11	4
;	Punctuator	12	4
if	Keyword	13	6
(	Punctuator	14	6
a	Identifier	4	6
>	Relational Op	15	6
b	Identifier	8	6
)	Punctuator	16	6
{	Punctuator	2	6
printf	Identifier	17	7
(	Punctuator	14	7
"\nInside if"	String Literal	18	7
)	Punctuator	16	7
;	Punctuator	12	7
}	Punctuator	19	8
else	Keyword	20	9
{	Punctuator	2	9
printf	Identifier	17	10
(	Punctuator	14	10
"\nInside else"	String Literal	21	10
)	Punctuator	16	10
;	Punctuator	12	10
}	Punctuator	19	11
}	Punctuator	19	13

MultiLineComment (0 lines):

SingleLineComment :
Test case for if else

fig. 3.1.3 Output of Test Case 2

3.1.4 Analyses

The lexical analyzer, being the first phase of the compiler, plays a crucial role in breaking down the source code into manageable pieces. By converting the raw source code into a series of tokens, it simplifies the complexity of the code for the subsequent phases. Tokens represent the fundamental elements like keywords, constants, and operators. Essentially, this phase acts as the compiler's initial filter, stripping away the non-essential components such as comments and whitespace, and preparing a streamlined version of the code for further analysis.

3.2 Syntax Analyzer

3.2.1 Methodology

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 nonterminals 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.

The strings are derived from

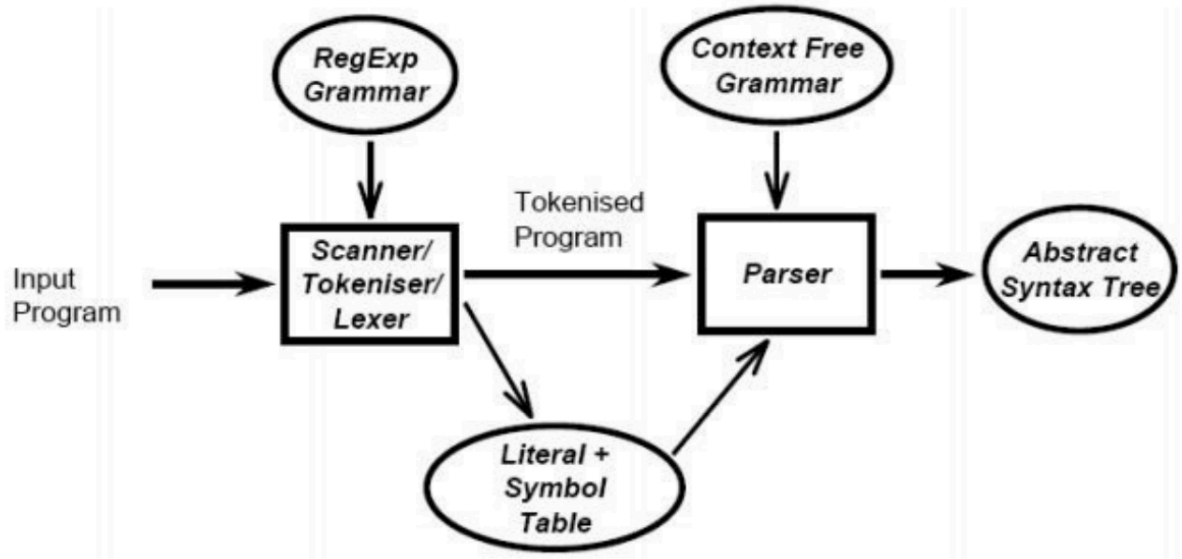


Fig 1: Flowchart of a parser

fig. 3.2.1 Flowchart of a parser

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

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.

Consider the following example with the given Production rules and Input String.

Input string : id + id * id

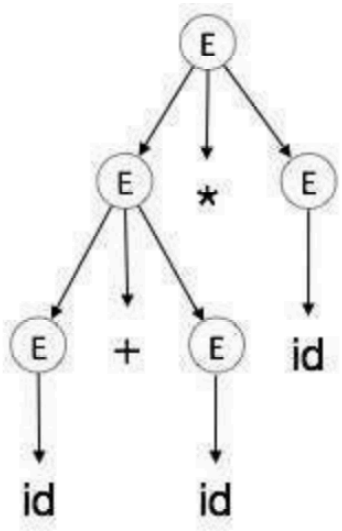
Production rules:

- 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 the 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
    - Expr, stmt

- 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.

3.2.2 Implementation

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.)

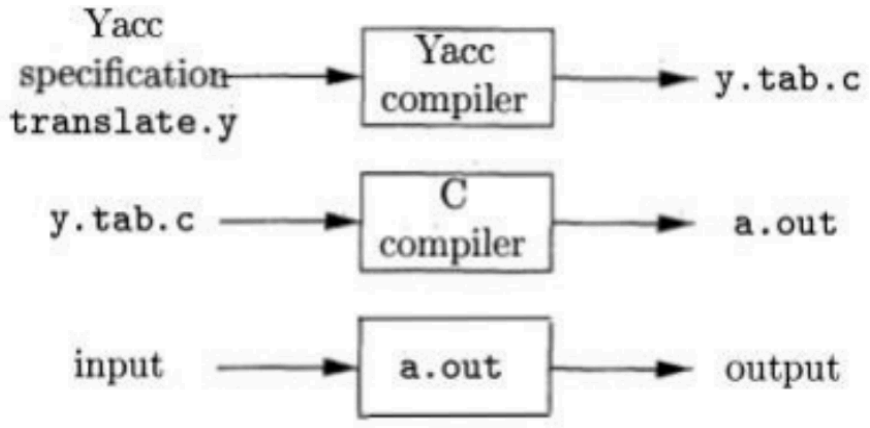


fig 3.2.2 Creating a syntactical analyzer with YACC

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.

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 the 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.

3.2.3 Results

parser.l

```
%{
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>

        #include "y.tab.h"


        int line = 1;
        extern int scope;

        extern void yyerror(const char *);
        static void comment(void);

        extern struct node * checksym(char *);


%}


D               [0-9]
L               [a-zA-Z_]
A               [a-zA-Z_0-9]
WS              [ \t\v\f]


%%

[\n]            { fprintf(yyout, "%s", yytext); line++; }
"/*"            { comment(); }
"//"[^\n]*      { /* Consume Comment */ }


        /* Data Types */
int                     { fprintf(yyout, "%s", yytext);  yylval.ival=0; return(INT);    }
float           { fprintf(yyout, "%s", yytext);  yylval.ival=1; return(FLOAT); }
char            { fprintf(yyout, "%s", yytext);  yylval.ival=2; return(CHAR); }
void            { fprintf(yyout, "%s", yytext);  yylval.ival=3; return(VOID); }



        /* Headers */
"#"                     { fprintf(yyout, "%s", yytext);  return HASH; }
include         { fprintf(yyout, "%s", yytext);  return INCLUDE; }


        /* C++ Libraries */
"iostream"      { fprintf(yyout, "%s", yytext);  return IOSTREAM; }


        /* Control Structures */
```

```
"iostream"      { fprintf(yyout, "%s", yytext);  return IOSTREAM; }


        /* Control Structures */
for                     { fprintf(yyout, "%s", yytext);  return FOR; }
while           { fprintf(yyout, "%s", yytext);  return WHILE; }
if                      { fprintf(yyout, "%s", yytext);  return IF; }


printf          { fprintf(yyout, "%s", yytext);  return PRINT; }
return          { fprintf(yyout, "%s", yytext);  return RETURN; }

        /* User Defined Data Types, Identifiers */

{L}{A}*                                                 {       fprintf(yyout, "%s", yytext);
                                                                yylval.ptr = checksym(yyt
                                                                return IDENTIFIER;
                                                        }
{D}+\.{D}+                      {       fprintf(yyout, "%s", yytext);
                                                                yylval.fval=atof(yytext);
                                                                return FLOAT_LITERAL;
                                                        }
{D}+                                                    {       fprintf(yyout, "%s", yytext);
                                                                yylval.ival=atoi(yytext);
                                                                return INTEGER_LITERAL;
                                                        }

"'"."'"                                                 {       fprintf(yyout, "%s", yytext);
                                                                yylval.cval= yytext[1];
                                                                return CHARACTER_LITERAL;
                                                        }

\"{A}+(".h"|".c")\"                             {fprintf(yyout, "%s", yytext);  return HEADER_LIT

\".*\"                                                  {fprintf(yyout, "%s", yytext);  return ST


        /* Assignment Operators */
"+="            {fprintf(yyout, "%s", yytext);  return(ADD_ASSIGN); }
"-="            {fprintf(yyout, "%s", yytext);  return(SUB_ASSIGN); }
"*="            {fprintf(yyout, "%s", yytext);  return(MUL_ASSIGN); }
"/="            {fprintf(yyout, "%s", yytext);  return(DIV_ASSIGN); }
"%="            {fprintf(yyout, "%s", yytext);  return(MOD_ASSIGN); }

        /* Relational Operators */
"++"            {fprintf(yyout, "%s", yytext);  return(INC_OP); }
"--"            {fprintf(yyout, "%s", yytext);  return(DEC_OP); }
"<="            {fprintf(yyout, "%s", yytext);  return(LE_OP); }
">="            {fprintf(yyout, "%s", yytext);  return(GE_OP); }
"=="            {fprintf(yyout, "%s", yytext);  return(EQ_OP); }
"!="            {fprintf(yyout, "%s", yytext);  return(NE_OP); }
```

```
";"                     {fprintf(yyout, "%s", yytext);  return(';'); }
"{"                     {fprintf(yyout, "%s", yytext);  scope++; return('{'); }
"}"                     {fprintf(yyout, "%s", yytext);  return('}'); }
","                     {fprintf(yyout, "%s", yytext);  return(','); }
":"                     {fprintf(yyout, "%s", yytext);  return(':'); }
"="                     {fprintf(yyout, "%s", yytext);  return('='); }
"("                     {fprintf(yyout, "%s", yytext);  return('('); }
")"                     {fprintf(yyout, "%s", yytext);  return(')'); }
("["|"<:")      {fprintf(yyout, "%s", yytext);  return('['); }
("]"|":>")      {fprintf(yyout, "%s", yytext);  return(']'); }
"."                     {fprintf(yyout, "%s", yytext);  return('.'); }
"&"                     {fprintf(yyout, "%s", yytext);  return('&'); }
"!"                     {fprintf(yyout, "%s", yytext);  return('!'); }
"~"                     {fprintf(yyout, "%s", yytext);  return('~'); }
"-"                     {fprintf(yyout, "%s", yytext);  return('-'); }
"+"                     {fprintf(yyout, "%s", yytext);  return('+'); }
"*"                     {fprintf(yyout, "%s", yytext);  return('*'); }
"/"                     {fprintf(yyout, "%s", yytext);  return('/'); }
"%"                     {fprintf(yyout, "%s", yytext);  return('%'); }
"<"                     {fprintf(yyout, "%s", yytext);  return('<'); }
">"                     {fprintf(yyout, "%s", yytext);  return('>'); }
"^"                     {fprintf(yyout, "%s", yytext);  return('^'); }
"|"                     {fprintf(yyout, "%s", yytext);  return('|'); }
"?"                     {fprintf(yyout, "%s", yytext);  return('?'); }


{WS}+           {fprintf(yyout, "%s", yytext); /* whitespace separates tokens */}

.                       { printf("No Match, Invalid Expression %s\n", yytext); return yytext[0];}

%%

int yywrap(void)
{
        return 1;
}


static void comment(void)
{
    int c;

    while ((c = input()) != 0)
        if (c == '*')
        {
            while ((c = input()) == '*');
            if (c == '/')
                return;

            if (c == 0)
                break;
        }
```

parser.y

```
%{
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>
        #include <limits.h>
        // #define COUNT 10

        void yyerror(const char*);
        int yylex();
        extern FILE * yyin, *yyout;

        int x=0;

        extern int line;

        int scope = 0;

        int unaryop = -1;               //unary operator type
        int assignop = -1;              //assignment operator == += -=
        int datatype = -1;              //type specifier
        int assigntype = -1;    //RHS type
        int idcheck = -1;               //check if ID
        int check_un = 0;               //check for undeclared variables


        char tempStr[100];              //sprintf

        struct node{
                char token[20];
                char name[20];
                int dtype;
                int scope;
                int lineno;
                int valid;
                union value{
                        float f;
                        int i;
                        char c;
                }val;

                struct node *link;

        }*first = NULL, *tmp, *crt, *lhs;

        typedef struct Node{
                struct Node *left;
                struct Node *right;
                char token[100];
                struct Node *val;
                int level;
        }Node;
```

```
        struct node * checksym(char *);
        void addsymbol(struct node *,char *);
        void addInt(struct node *, int, int);
        void addFloat(struct node *, int, float);
        void addChar(struct node *, int, char);
        void addfunc(struct node *t, int, char *);
        void printsymtable();

        struct node * addtosymbol(struct node * n);
        void cleansymbol();


        //AST
        void create_node(char *token, int leaf);
        void push_tree(Node *newnode);
        Node *pop_tree();
        void preorder(Node* root);
        void printtree(Node* root);
        int getmaxlevel(Node *root);
        void printGivenLevel(Node* root, int level, int h);
        void get_levels(Node *root, int level);


%}

%token  HASH INCLUDE IOSTREAM
%token  STRING_LITERAL HEADER_LITERAL PRINT RETURN
%left   '+' '-'
%left   '/' '*' '%'
%right  '='

%union{
        int ival;
        float fval;
        char cval;
        char string[128];
        struct node *ptr;
}


%token <ival>   INTEGER_LITERAL
%token <cval>   CHARACTER_LITERAL
%token <fval>   FLOAT_LITERAL
%token <ptr>    IDENTIFIER

%token  INC_OP  DEC_OP  LE_OP   GE_OP   EQ_OP   NE_OP
%token  MUL_ASSIGN      DIV_ASSIGN      MOD_ASSIGN      ADD_ASSIGN      SUB_ASSIGN
%token  <ival>  CHAR    INT     FLOAT   VOID
%token  FOR  WHILE  IF
```

```
program
        : HASH INCLUDE '<' libraries '>' program
        | HASH INCLUDE HEADER_LITERAL    program
        | translation_unit
        ;


translation_unit
        : ext_dec
        | translation_unit ext_dec
        ;


ext_dec
        : declaration
        | function_definition
        ;


libraries
        : IOSTREAM
        ;


compound_statement
        : '{' '}'
        | '{' block_item_list '}'
        ;


block_item_list
        : block_item
        | block_item_list block_item
                        {
                                create_node("stmt", 0);
                        }
        ;


block_item
        : declaration
        | statement
        | function_call ';'
        | RETURN expression_statement
                        {
                                create_node("return", 1);
                        }
        | printstat ';'
        ;
```

```
        : type_specifier init_declarator_list ';'
        ;


statement
        : compound_statement {
                                                struct node *ftp;
                                                ftp = first;
                                                while(ftp!=NULL){
                                                        if(ftp->scope == scope && ftp->valid ==
                                                                ftp->valid = 0;
                                                        }
                                                        ftp=ftp->link;
                                                }
                                                scope--;

                                        }
        | expression_statement
        | iteration_statement
        | condition_statement
        ;

condition_statement
        : IF '(' relational_expression ')' statement {create_node("if", 0);}
        ;


iteration_statement
        :
         FOR '(' expression_statement  expression_statement expression  ')' statement
                        {
                                create_node("for", 0);
                        }
        | WHILE '(' relational_expression ')' statement
        {
                create_node("while", 0);
        }
        ;


type_specifier
        : VOID  {       datatype = $1; }
        | CHAR  {       datatype = $1; }
        | INT   {       datatype = $1; }
        | FLOAT {       datatype = $1; }
        ;


init_declarator_list
        : init_declarator
        | init_declarator_list ',' init_declarator
        ;
```

```
                struct node *ftp, *nnode;
                nnode = (struct node *)malloc(sizeof(struct node));
                ftp = first;
                while(ftp->link!=NULL){
                        ftp = ftp->link;
                }
                addsymbol(nnode,$1->name);
                ftp->link = nnode;
                nnode->link = NULL;
                $1 = nnode;

                if (datatype == 0){

                        addInt($1, 0, $4);
                        if(assigntype == 1){
                                printf("Line:%d: ", line);
                                printf("\033[1;35m");
                                printf("warning: ");
                                printf("\033[0m");
                                printf("\'float\' to \'int\' \n\n");
                        }
                }
                else if(datatype == 1){

                        addFloat($1, 1, $4);
                        if(assigntype == 2){
                                printf("Line:%d: ", line);
                                printf("\033[1;35m");
                                printf("warning: ");
                                printf("\033[0m");
                                printf(" \'char\' to \'float\' \n\n");
                        }
                }
                else if(datatype == 2){
                        float tempf = (float)$4;
                        addChar($1, 2, (int)tempf);

                        if(assigntype == 1){
                                printf("Line:%d: ", line);

                                printf(" \'float\' to \'char\' \n\n");
                        }
                }
                x = datatype;

                create_node("=", 0);

        }


        else if($1->dtype !=- 1){
```

```
                        printf(" ");
                }
            printf("%s", root->token);
        }
    else if (level > 1){
        printGivenLevel(root->left,level-1,h);
                for(int j=0; j<=h-1-level; j++){
                        printf(" ");
                }
        printGivenLevel(root->right,level-1,h);
    }
}


void preorder(Node * node){
        if (node == NULL)
                return;

        if(node->left!=NULL && node->right!=NULL)
                strcat(preBuf, "   ( ");
        strcat(preBuf, node->token);
        strcat(preBuf, "   ");

        preorder(node-> left);


        if(node->right){
                preorder(node-> right);
        }

        if(node->left!=NULL && node->right!=NULL)
                strcat(preBuf, ")   ");
        // printf("\n");

}

void get_levels(Node *root, int level){
        root->level = level;
        if(root->left == NULL && root->right == NULL){
                return;
        }
        if(root->left == NULL){
                get_levels(root->right, level+1);
        }
        else if(root->right == NULL){
                get_levels(root->left, level+1);
        }
        else{
                get_levels(root->left, level+1);
                get_levels(root->right, level+1);
        }
}
```

OUTPUT

Test Case 1

```c
int main(){

        int i, a, b;
        int nume=3.45;
        for(i = 0;  i < 10; i++){
                a=i;
        }
        i=1;
}
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/2_Syntax_Analyzer$ lex parseTree.l
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/2_Syntax_Analyzer$ yacc -d parseTree.y
parseTree.y:787.11-18: warning: POSIX Yacc does not support string literals [-Wyacc]
  787 |         | "INC_OP"      {       unaryop = 5;    }
      |           ^~~~~~~
parseTree.y:788.11-18: warning: POSIX Yacc does not support string literals [-Wyacc]
  788 |         | "DEC_OP"      {       unaryop = 6;    }
      |           ^~~~~~~
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/2_Syntax_Analyzer$ cc lex.yy.c y.tab.c
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/2_Syntax_Analyzer$ ./a.out < TestCases/forloop.c

Line:6: 'float' to 'int'


ST

   SYMBOL               NAME      TYPE        SCOPE       LINE #             VALUE
 identifier               i       int           1             5              1
 identifier               a       int           1             5              0
 identifier               b       int           1             5              -
 identifier            nume       int           1             6              3




Parse Tree
                                                                    main

                                                              stmt

                                                        stmt          =

                                                  =           for         i          1

                                            i           0         ++          =

                                                  <           i         a          i

                                        i           10


Preorder Traversal of Parse Tree:
main      ( stmt      ( stmt      ( =   i   0   )      ( for      ( ++      ( <   i   10   )   i   )      ( =   a   i   )   )   )      ( =   i   1   )   )
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/2_Syntax_Analyzer$
```

fig. 3.2.3 Output of Test Case 1

Test Case 2

```c
void main()
{
        int a=1,b=2,c=10;

        if(a>b){
                printf("\nInside if");
        }
        else{
                printf("\nInside else");
        }

}
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/2_Syntax_Analyzer$ ./a.out < TestCases/ifelse.c

Line:8: use of undeclared identifier 'else'

Line:8: syntax error


Parse Tree

                                        stmt

                                =   if

                        b   2   =   >


Preorder Traversal of Parse Tree:
   ( stmt      ( =    b    2    )      ( if      ( =    c    10    )      ( >    a    b    )    )    )
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/2_Syntax_Analyzer$
```

fig. 3.2.4 Output of Test Case 2

3.2.4 Analyses

Following the lexical analysis, the syntax analyzer takes the sequence of tokens and scrutinizes their structural arrangement. This phase ensures that the code's syntax aligns with the programming language's rules, akin to checking grammar in a language. The output of this process is a parse tree, a hierarchical structure that represents the syntactic structure of the code.

3.3 Semantic Analyzer

3.3.1 Methodology

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 → 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.

3.3.2 Implementation

An attribute is a property whose value is assigned to a grammar symbol. Attribute computation functions (or semantic functions) are associated with the production 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 a well-defined domain of values, such as integer, float, character, string, and expressions. Attribute grammar is a medium to provide semantics to 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

$E \rightarrow E + T \{ E.value = E.value + T.value \}$

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 parents 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 the next phase of the compiler. Output could be either a parse tree or abstract syntax tree. Now to interleave semantic analysis with the 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 the form of attributes attached to the nodes. Syntax directed translation rules use 1) lexical values of nodes, 2) constants, 3) attributes associated with 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.

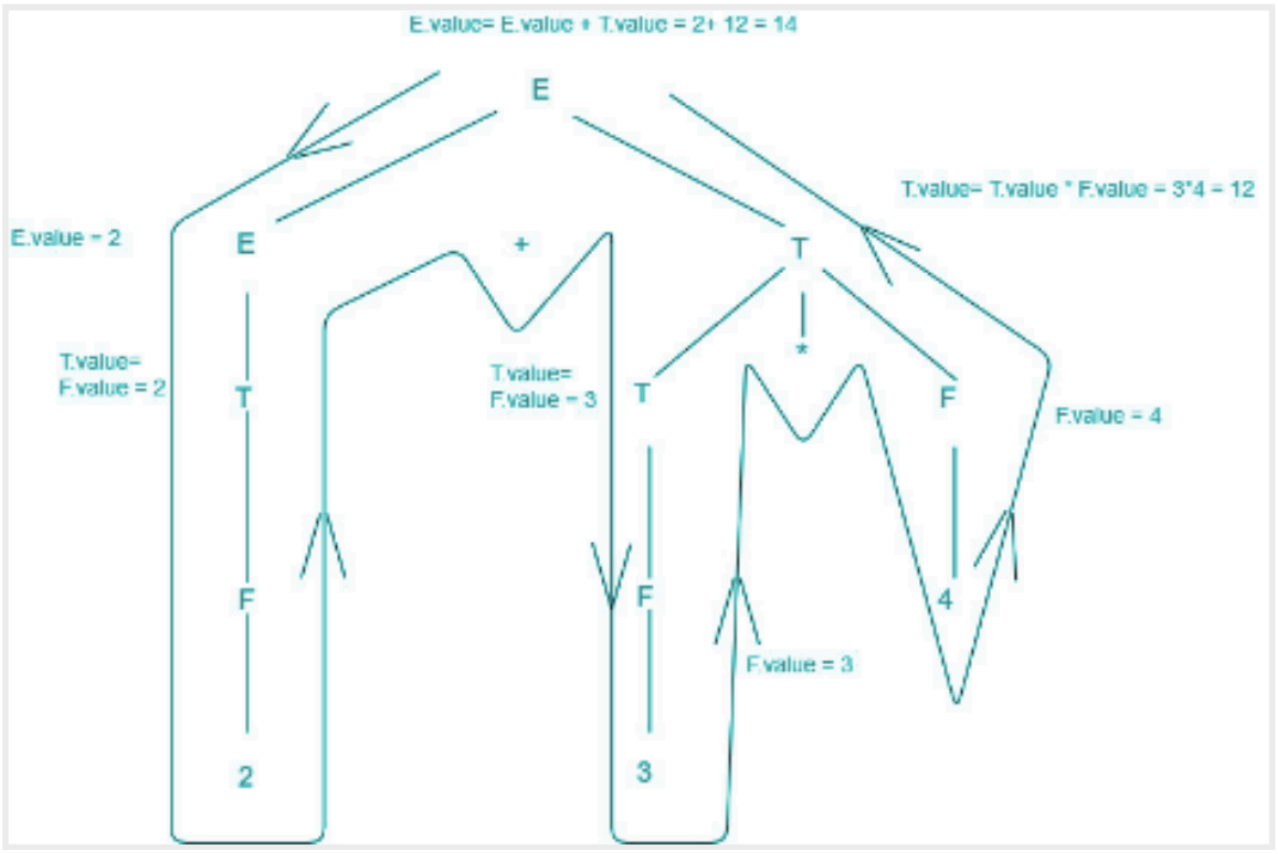


fig. 3.3.1 Flow of Semantic Analysis

Above diagram shows how semantic analysis could happen. The flow of information happens bottom-up and all the children's 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.

3.3.3 Results

scanner.l

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

        struct symboltable
        {
                char name[100];
                char class[100];
                char type[100];
                char value[100];
                int nesting_val;
                int lineno;
                int length;
                int params_count;
        }ST[1007];

        struct constanttable
        {
                char name[100];
                char type[100];
                int length;
        }CT[1007];

        int current_nesting = 0;
        int params_count = 0;
        extern int yylval;


        int hash(char *str)
        {
                int value = 0;
                for(int i = 0 ; i < strlen(str) ; i++)
                {
                        value = 10*value + (str[i] - 'A');
                        value = value % 1007;
                        while(value < 0)
                                value = value + 1007;
                }
                return value;
        }

        int lookupST(char *str)
        {
                int value = hash(str);
                if(ST[value].length == 0)
                {
                        return 0;
                }
                else if(strcmp(ST[value].name,str)==0)
                {

                        return value;
                }
```

```
	void insertSTline(char *str1, int line)
	{
		for(int i = 0 ; i < 1007 ; i++)
		{
			if(strcmp(ST[i].name,str1)==0)
			{
				ST[i].lineno = line;
			}
		}
	}


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

			int pos = 0;

			for (int i = value + 1 ; i!=value ; i = (i+1)%1007)
			{
				if(ST[i].length == 0)
				{
					pos = i;
					break;
				}
			}

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

```
                {
                        if(ST[i].length == 0)
                        {
                                pos = i;
                                break;
                        }
                }

                strcpy(ST[pos].name,s);
                strcpy(ST[pos].class,"Identifier");
                ST[pos].length = strlen(s);
                ST[pos].nesting_val = nest;
                ST[pos].params_count = -1;
                ST[pos].lineno = yylineno;
        }
        else
        {
                for(int i = 0 ; i < 1007 ; i++)
                {
                        if(strcmp(ST[i].name,s)==0 )
                        {
                                ST[i].nesting_val = nest;
                        }
                }
        }
}

void insertSTparamscount(char *s, int count)
{
        for(int i = 0 ; i < 1007 ; i++)
        {
                if(strcmp(ST[i].name,s)==0 )
                {
                        ST[i].params_count = count;
                }
        }
}

int getSTparamscount(char *s)
{
        for(int i = 0 ; i < 1007 ; i++)
        {
                if(strcmp(ST[i].name,s)==0 )
                {
                        return ST[i].params_count;
                }
        }
        return -2;
}

void insertSTF(char *s)
{
        for(int i = 0 ; i < 1007 ; i++)
        {
```

```
                        if(CT[i].length == 0)
                        {
                                pos = i;
                                break;
                        }
                }

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

void deletedata (int nesting)
{
        for(int i = 0 ; i < 1007 ; i++)
        {
                if(ST[i].nesting_val == nesting)
                {
                        ST[i].nesting_val = 99999;
                }
        }


}

int checkscope(char *s)
{
        int flag = 0;
        for(int i = 0 ; i < 1000 ; i++)
        {
                if(strcmp(ST[i].name,s)==0)
                {
                        if(ST[i].nesting_val > current_nesting)
                        {
                                flag = 1;
                        }
                        else
                        {
                                flag = 0;
                                break;
                        }
                }
        }
        if(!flag)
        {
                return 1;
        }
        else
        {
                return 0;
        }
}

int check_id_is_func(char *s)
```

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

        return 0;
}

int check_duplicate(char* str)
{
        for(int i=0; i<1007; i++)
        {
                if(strcmp(ST[i].name, str) == 0 && strcmp(ST[i].class, "Functi
                {
                        printf("Function redeclaration not allowed\n");
                        exit(0);
                }
        }
}

int check_declaration(char* str, char *check_type)
{
        for(int i=0; i<1007; i++)
        {
                if(strcmp(ST[i].name, str) == 0 && strcmp(ST[i].class, "Functi
strcmp(ST[i].name,"printf")==0 )
                {
                        return 1;
                }
        }
        return 0;
}

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

char gettype(char *s, int flag)
{
                for(int i = 0 ; i < 1007 ; i++ )
```

```
        }

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

        char gettype(char *s, int flag)
        {
                        for(int i = 0 ; i < 1007 ; i++ )
                        {
                                if(strcmp(ST[i].name,s)==0)
                                {
                                        return ST[i].type[0];
                                }
                        }

        }

        void printST()
        {
                printf("%10s | %15s | %10s | %10s | %10s | %15s | %10s |\n","symbol na
"Type","Value", "Line No.", "Nesting Count", "Count of Params");
                for(int i=0;i<100;i++) {
                        printf("_");
                }
                printf("\n");
                for(int i = 0 ; i < 1007 ; i++)
                {
                        if(ST[i].length == 0)
                        {
                                continue;
                        }
                        printf("%10s | %15s | %10s | %10s | %10d | %15d | %10d |\n",ST
ST[i].class, ST[i].type, ST[i].value, ST[i].lineno, ST[i].nesting_val, ST[i].params_co
                }
    }


        void printCT()
        {
                printf("%10s | %15s\n","constant name", "constant type");
                for(int i=0;i<85;i++) {
                        printf("_");
                }
                printf("\n");
                for(int i = 0 ; i < 1007 ; i++)
                {
                        if(CT[i].length == 0)
```

```
                                continue;

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

%}

DE "define"
IN "include"

%%
\n      {yylineno++;}
([#][" "]*({IN})[ ]*([<]?)([A-Za-z]+)[.]?([A-Za-z]*)([>]?))/["\n"|\/|" "|"\t"]  { }
([#][" "]*({DE})[" "]*([A-Za-z]+)(" ")*[0-9]+)/["\n"|\/|" "|"\t"]
\/\/(.*)
{ }
\/\*([^*]|[\r\n]|(\*+([^*/]|[\r\n])))*\*+\/
{ }
[ \n\t] ;
";"                             { return(';'); }
","                             { return(','); }
("{")                   { return('{'); }
("}")                   { return('}'); }
"("                             { return('('); }
")"                             { return(')'); }
("["|"<:")              { return('['); }
("]"|":>")              { return(']'); }
":"                             { return(':'); }
"."                             { return('.'); }

"char"                  { strcpy(curtype,yytext); insertST(yytext, "Keyword");return
"double"                { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
"else"                  { insertST(yytext, "Keyword"); return ELSE;}
"float"                 { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
"while"                 { insertST(yytext, "Keyword"); return WHILE;}
"do"                    { insertST(yytext, "Keyword"); return DO;}
"for"                   { insertST(yytext, "Keyword"); return FOR;}
"if"                    { insertST(yytext, "Keyword"); return IF;}
"int"                   { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
"long"                  { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
"return"                { insertST(yytext, "Keyword"); return RETURN;}
"short"                 { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
"signed"                { strcpy(curtype,yytext); insertST(yytext, "Keyword"); return
"sizeof"                { insertST(yytext, "Keyword"); return SIZEOF;}
"struct"                { strcpy(curtype,yytext);   insertST(yytext, "Keyword");  retu
"unsigned"              { insertST(yytext, "Keyword");   return UNSIGNED;}
"void"                  { strcpy(curtype,yytext);   insertST(yytext, "Keyword");  retu
"break"                 { insertST(yytext, "Keyword");  return BREAK;}
```

```
"||"                    { return OR_operator; }
"^"                     { return caret_operator; }
"*="                    { return multiplication_assignment_operator; }
"/="                    { return division_assignment_operator; }
"%="                    { return modulo_assignment_operator; }
"+="                    { return addition_assignment_operator; }
"-="                    { return subtraction_assignment_operator; }
"<<="                   { return leftshift_assignment_operator; }
">>="                   { return rightshift_assignment_operator; }
"&="                    { return AND_assignment_operator; }
"^="                    { return XOR_assignment_operator; }
"|="                    { return OR_assignment_operator; }
"&"                     { return amp_operator; }
"!"                     { return exclamation_operator; }
"~"                     { return tilde_operator; }
"-"                     { return subtract_operator; }
"+"                     { return add_operator; }
"*"                     { return multiplication_operator; }
"/"                     { return division_operator; }
"%"                     { return modulo_operator; }
"|"                     { return pipe_operator; }
\=                      { return assignment_operator;}

\"[^\n]*\"/[;|,|\)]                    {strcpy(curval,yytext); insertCT(yytext,"Strin
return string_constant;}
\'[A-Z|a-z]\'/[;|,|\)|:]               {strcpy(curval,yytext); insertCT(yytext,"Chara
return character_constant;}
[a-z|A-Z]([a-z|A-Z]|[0-9])*/\[  {strcpy(curid,yytext); insertST(yytext, "Array Identi
array_identifier;}
[1-9][0-9]*|0/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\]|\}|:|\n|\t|\^]
{strcpy(curval,yytext); insertCT(yytext, "Number Constant"); yylval = atoi(yytext); re
integer_constant;}
([0-9]*)\.([0-9]+)/[;|,|" "|\)|<|>|=|\!|\||&|\+|\-|\*|\/|\%|~|\n|\t|\^]
{strcpy(curval,yytext); insertCT(yytext, "Floating Constant"); return float_constant;}
[A-Za-z_][A-Za-z_0-9]* {strcpy(curid,yytext); insertST(curid,"Identifier"); return ide

(.?) {
                if(yytext[0]=='#')
                {
                        printf("Error in Pre-Processor directive at line no. %d\n",yyl
                }
                else if(yytext[0]=='/')
                {
                        printf("ERR_UNMATCHED_COMMENT at line no. %d\n",yylineno);
                }
                else if(yytext[0]=='"')
                {
                        printf("ERR_INCOMPLETE_STRING at line no. %d\n",yylineno);
                }
                else
                {
                        printf("ERROR at line no. %d\n",yylineno);
                }
                printf("%s\n", yytext);
```

parser.y

```
%{
        void yyerror(char* s);
        int yylex();

        #include "stdio.h"
        #include "stdlib.h"
        #include "ctype.h"
        #include "string.h"

        void ins();
        void insV();

        int flag=0;

        extern char curid[20];
        extern char curtype[20];
        extern char curval[20];

        extern int current_nesting;

        void deletedata (int );
        int checkscope(char*);
        int check_id_is_func(char *);
        void insertST(char*, char*);
        void insertSTnest(char*, int);
        void insertSTparamscount(char*, int);
        int getSTparamscount(char*);
        int check_duplicate(char*);
        int check_declaration(char*, char *);
        int check_params(char*);
        int duplicate(char *s);
        int checkarray(char*);
        char currfunctype[100];
        char currfunc[100];
        char currfunccall[100];
        void insertSTF(char*);
        char gettype(char*,int);
        char getfirst(char*);
        extern int params_count;
        int call_params_count;
%}

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

%token identifier array_identifier func_identifier
%token integer_constant string_constant float_constant character_constant
```

```
%left leftshift_operator rightshift_operator
%left add_operator subtract_operator
%left multiplication_operator division_operator modulo_operator

%right SIZEOF
%right tilde_operator exclamation_operator
%left increment_operator decrement_operator


%start program

%%
program
                        : declaration_list;

declaration_list
                        : declaration D

D
                        : declaration_list
                        | ;

declaration
                        : variable_declaration
                        | function_declaration

variable_declaration
                        : type_specifier variable_declaration_list ';'

variable_declaration_list
                        : variable_declaration_list ',' variable_declaration_identifie
variable_declaration_identifier;

variable_declaration_identifier
                        : identifier {if(duplicate(curid)){printf("Duplicate\n");exit(
insertSTnest(curid,current_nesting); ins();  } vdi
                          | array_identifier {if(duplicate(curid)){printf("Duplicate\n
insertSTnest(curid,current_nesting); ins();  } vdi;



vdi : identifier_array_type | assignment_operator simple_expression  ;

identifier_array_type
                        : '[' initilization_params
                        | ;

initilization_params
                        : integer_constant ']' initilization {if($$ < 1) {printf("Wron
size\n"); exit(0);} }
                        | ']' string_initilization;

initilization
                        : string_initilization
```

```
long_grammar
                        : INT  | ;

short_grammar
                        : INT | ;

function_declaration
                        : function_declaration_type function_declaration_param_stateme

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

function_declaration_param_statement
                        : params ')' statement;

params
                        : parameters_list | ;

parameters_list
                        : type_specifier { check_params(curtype); } parameters_identi
{ insertSTparamscount(currfunc, params_count); };

parameters_identifier_list
                        : param_identifier parameters_identifier_list_breakup;

parameters_identifier_list_breakup
                        : ',' parameters_list
                        | ;

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

param_identifier_breakup
                        : '[' ']'
                        | ;

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

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

statment_list
                        : statement statment_list
                        | ;

expression_statment
                        : expression ';'
```

```
return_statement
                        : RETURN ';' {if(strcmp(currfunctype,"void")) {printf("Returni
non-void function\n"); exit(0);}}
                        | RETURN expression ';' {       if(!strcmp(currfunctype, "voi
                                                                                {
yyerror("Function returns something but is declared void");
                                                                                }

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

break_statement
                        : BREAK ';' ;

string_initilization
                        : assignment_operator string_constant {insV();} ;

array_initialization
                        : assignment_operator '{' array_int_declarations '}';

array_int_declarations
                        : integer_constant array_int_declarations_breakup;

array_int_declarations_breakup
                        : ',' array_int_declarations
                        | ;

expression
                        : mutable assignment_operator expression              {
if($1==1 && $3==1)
{
                                                                                $$=1
                                                                                }
                                                                                else
                                                                                {$$=
mismatch\n"); exit(0);}
                                                                              }
                        | mutable addition_assignment_operator expression     {
if($1==1 && $3==1)
                                                                                $$=1
                                                                                else
                                                                                {$$=
mismatch\n"); exit(0);}
                                                                              }
```

```
                                                                                }
                        | mutable modulo_assignment_operator expression

if($1==1 && $3==1)
                                                                                    $$=1
                                                                                    else
                                                                                    {$$=
mismatch\n"); exit(0);}
                                                                                }
                        | mutable increment_operator
{if($1 == 1) $$=1; else $$=-1;}
                        | mutable decrement_operator
{if($1 == 1) $$=1; else $$=-1;}
                        | simple_expression {if($1 == 1) $$=1; else $$=-1;} ;


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

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


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

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

relational_operators
                        : greaterthan_assignment_operator | lessthan_assignment_operat
greaterthan_operator
                        | lessthan_operator | equality_operator | inequality_operator

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

sum_operators
                        : add_operator
                        | subtract_operator ;

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

MULOP
```

```
call
                        : identifier '('{
                                        if(!check_declaration(curid, "Function"))
                                        { printf("Need to declare function"); exit(0);}
                                        insertSTF(curid);
                                                strcpy(currfunccall,curid);
                                        } arguments ')'
                                                { if(strcmp(currfunccall,"printf"))
                                                        {
                                                                if(getSTparamscount(cu
=call_params_count)
                                                                {
                                                                        yyerror("funct
required does not match the passed arguments...");
                                                                        exit(8);
                                                                }
                                                        }
                                                };

arguments
                        : arguments_list | ;

arguments_list
                        : expression { call_params_count++; } A ;

A
                        : ',' expression { call_params_count++; } A
                        | ;

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

%%

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

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

        if(flag == 0)
        {
```

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

%%

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

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

        if(flag == 0)
        {
                printf( "PASSED: Semantic Phase\n");
                printf("%30s"  "PRINTING SYMBOL TABLE"  "\n\n", " ");
                printST();
                printf("%30s %s\n", " ", "------------------------------");
                printf("\n\n%30s"  "PRINTING CONSTANT TABLE"  "\n\n", " ");
                printCT();
        }
}

void yyerror(char *s)
{
        printf( "%d %s %s\n", yylineno, s, yytext);
        flag=1;
        printf( "FAILED: Semantic Phase Parsing failed\n" );
        exit(7);
}

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

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

int yywrap()
{
        return 1;
```

OUTPUT

Test Case 1

```
// test case to check loop statements

int main(){

        int i, a, b;
        int nume=3.45;
        for(i = 0;  i < 10; i++){
                a=i;
        }
        i=1;
}
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$ lex scanner.l
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$ yacc -d parser.y
parser.y:50.1-7: warning: POSIX Yacc does not support %expect [-Wyacc]
   50 | %expect 1
      | ^~~~~~~
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$ cc lex.yy.c y.tab.c
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$ ./a.out < TestCases/forloop.c
PASSED: Semantic Phase
                        PRINTING SYMBOL TABLE

symbol name |          Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |      Identifier |        int |            |          5 |           99999 |          -1 |
          b |      Identifier |        int |            |          5 |           99999 |          -1 |
          i |      Identifier |        int |          1 |          5 |           99999 |          -1 |
        for |         Keyword |            |            |          7 |            9999 |          -1 |
       main |        Function |        int |            |          3 |            9999 |          -1 |
       nume |      Identifier |        int |       3.45 |          6 |           99999 |          -1 |
        int |         Keyword |            |            |          3 |            9999 |          -1 |
                          ------------------------------


                        PRINTING CONSTANT TABLE

constant name |   constant type
_____________________________________________________________________________
       3.45 | Floating Constant
         10 | Number Constant
          0 | Number Constant
          1 | Number Constant
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$
```

fig. 3.3.2 Output of Test Case 1

Test Case 2

```c
void main()
{
        int a=1,b=2,c=10;

        if(a>b){
                printf("\nInside if");
        }
        else{
                printf("\nInside else");
        }

}
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$ lex scanner.l
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$ yacc -d parser.y
parser.y:50.1-7: warning: POSIX Yacc does not support %expect [-Wyacc]
   50 | %expect 1
      | ^~~~~~~
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$ cc lex.yy.c y.tab.c
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$ ./a.out < TestCases/ifelse.c
PASSED: Semantic Phase
                        PRINTING SYMBOL TABLE

symbol name |           Class |       Type |      Value |   Line No. |   Nesting Count | Count of Params |
_____________________________________________________________________________________________________
          a |      Identifier |        int |          1 |          3 |           99999 |          -1 |
          b |      Identifier |        int |          2 |          3 |           99999 |          -1 |
          c |      Identifier |        int |         10 |          3 |           99999 |          -1 |
       main |        Function |       void |            |          1 |            9999 |          -1 |
         if |         Keyword |            |            |          5 |            9999 |          -1 |
        int |         Keyword |            |            |          3 |            9999 |          -1 |
       else |         Keyword |            |            |          8 |            9999 |          -1 |
       void |         Keyword |            |            |          1 |            9999 |          -1 |
     printf |        Function |            |            |          6 |            9999 |          -1 |
                          ------------------------------


                        PRINTING CONSTANT TABLE

constant name |   constant type
_____________________________________________________________________________
"\nInside if" | String Constant
"\nInside else" | String Constant
         10 | Number Constant
          1 | Number Constant
          2 | Number Constant
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$
```

fig. 3.3.3 Output of Test Case 2

Test Case 3

```c
#include <stdio.h>

int main() {
  int i

  for (i = 1; i < 11; i++)
  {
    printf("%d ", i);
  }
  return 0
}
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$ ./a.out < code.c
6 syntax error for
FAILED: Semantic Phase Parsing failed
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$
```

fig. 3.3.4 Output of Test Case 3

Test Case 4

```c
#include <stdio.h>

int main() {
  int i;

  for (i = 1; i < 11; i++)
  {
    printf("%d ", i);

  return 0;
}
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$ ./a.out < code.c
12 syntax error
FAILED: Semantic Phase Parsing failed
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/3_Semantic_Analyzer$
```

fig. 3.3.5 Output of Test Case 4

3.3.4 Analyses

The semantic analyzer enriches the syntactical structure with semantic rules, ensuring that the operations and expressions are meaningful within the context of the language. This phase checks for type compatibility, scope resolution, and other semantic considerations, ensuring the code not only looks correct but also makes logical sense.

3.4 Intermediate Code Generator

3.4.1 Methodology

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.

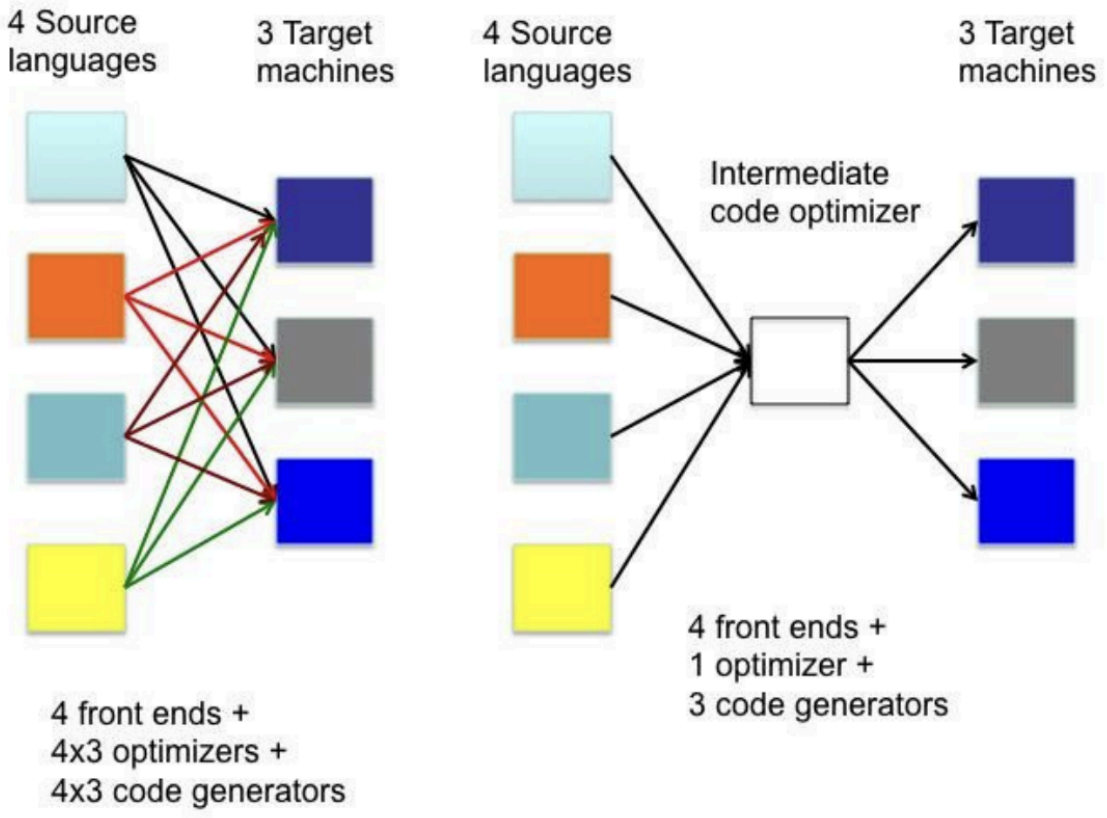


fig. 3.4.1 Flow of Intermediate code generator

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 the same for all the compilers.

3) The second part of the 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).

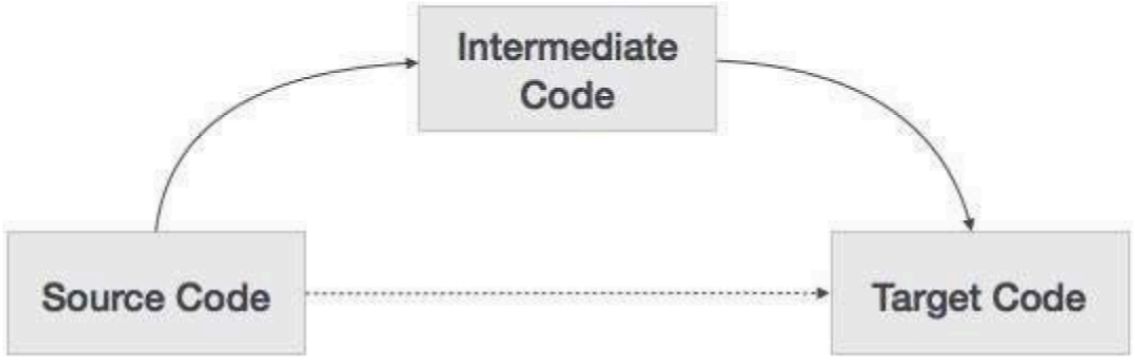


fig. 3.4.2 Path of Intermediate code

3.4.2 Implementation

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 ?

1) It's easy to change the source or the target language by adapting only the front-end or back-end (portability).

2) It makes optimization easier: one needs to write optimization methods only for the intermediate representation.

3) The intermediate representation can be directly interpreted.

Intermediate codes can be represented in a variety of ways and they have their own benefits.

1) 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.

2) 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 generators assume an 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 statements. A sequence of three address statements is known as three address

code. Three address statements are 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 statements.

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		a

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.

3.4.3 Results

ICG.l

```
%{
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>

        #include "y.tab.h"

        int line = 1;

        extern FILE * fp;

        extern void yyerror(const char *);
        static void comment(void);



%}


D               [0-9]
L               [a-zA-Z_]
A               [a-zA-Z_0-9]
WS              [ \t\v\f]


%%

[\n]            {   line++;     }
"/*"            { comment(); }
"//"[^\n]*      { /* Consume Comment */ }


        /* Data Types */
int                     {    return(INT); }
float           {    return(FLOAT); }
char            {    return(CHAR); }
void            {    return(VOID); }


        /*COUT STUFF*/
"endl" {return ENDL;}
"cout" {return COUT;}

        /* Headers */
"#"                     {   return HASH; }
include         {   return INCLUDE; }

"using namespace std" {return NAMESPACE;}
```

```
";"                     {   return(';'); }
"{"                     {   return('{'); }
"}"                     {   return('}'); }
","                     {   return(','); }
":"                     {   return(':'); }
"="                     {   return('='); }
"("                     {   return('('); }
")"                     {   return(')'); }
("["|"<:")      {   return('['); }
("]"|":>")      {   return(']'); }
"."                     {   return('.'); }
"&"                     {   return('&'); }
"!"                     {   return('!'); }
"~"                     {   return('~'); }
"-"                     {   return('-'); }
"+"                     {   return('+'); }
"*"                     {   return('*'); }
"/"                     {   return('/'); }
"%"                     {   return('%'); }
"<"                     {   return('<'); }
">"                     {   return('>'); }
"^"                     {   return('^'); }
"|"                     {   return('|'); }
"?"                     {   return('?'); }


{WS}+           {  /* whitespace separates tokens */}

.                       { printf("No Match, Invalid Expression %s\n", yytext);

%%

int yywrap(void)
{
        return 1;
}


static void comment(void)
{
    int c;

    while ((c = input()) != 0)
        if (c == '*')
        {
            while ((c = input()) == '*');
            if (c == '/')
                return;

            if (c == 0)
                break;
        }
    yyerror("Unterminated comment");
}
```

ICG.y

```
%{
        #include <stdio.h>
        #include <stdlib.h>
        #include <string.h>

        void yyerror(const char*);
        int yylex();

        extern FILE * yyin, *yyout;

        FILE *fp_icg, *fp_quad;

        extern int line;

        char buffer[100];
        int ln = 0, tempno = 0;

        int assignop = -1;              //assignment operator == += -=
        int unaryop = -1;               //unary operator type
        int exprno = -1;                //FOR loop , 3rd expression
        int m = 0;                              //string length for buffer

        int paramno = 0;                //Number of parameters
        char intbuf[20];
        char secIDbuf[20];



%}

%token  HASH INCLUDE CSTDIO STDLIB MATH STRING TIME IOSTREAM CONIO
%token  NAMESPACE
%token COUT ENDL
%token  STRING_LITERAL HEADER_LITERAL PRINT RETURN
%left   '+' '-'
%left   '/' '*' '%'
%right  '='
%right UMINUS


%union{
        char sval[300];
}


%token  <sval> INTEGER_LITERAL
%token  <sval> CHARACTER_LITERAL
%token  <sval> FLOAT_LITERAL
%token  <sval> IDENTIFIER
%token  <sval> FOR
%token  <sval> WHILE
```

```
%type <sval>    expression
%type <sval>    postfix_expression
%type <sval>    assignment_expression
%type <sval>    conditional_expression
%type <sval>    equality_expression
%type <sval>    unary_operator
%type <sval>    expr
%type <sval>    declarator


%start S

%%
S : program
        ;

program
        : HASH INCLUDE '<' libraries '>'  program
        | HASH INCLUDE HEADER_LITERAL    program
        | NSPACE translation_unit
        ;
NSPACE
        : NAMESPACE ';'
translation_unit
        : ext_dec
        | translation_unit ext_dec
        ;


ext_dec
        : declaration
        | function_definition
        ;


libraries
        : CSTDIO
        | STDLIB
        | MATH
        | STRING
        | TIME
        | IOSTREAM
        | CONIO
        ;


compound_statement
        : '{' '}'
        | '{' block_item_list '}'
        ;
```

```
        | STDLIB
        | MATH
        | STRING
        | TIME
        | IOSTREAM
        | CONIO
        ;


compound_statement
        : '{' '}'
        | '{' block_item_list '}'
        ;


block_item_list
        : block_item
        | block_item_list block_item
        ;


block_item
        : declaration
        | statement
        | RETURN expression_statement
        | function_call ';'
        | printstat ';'
        ;


printstat
        : PRINT '(' STRING_LITERAL ')'
        | PRINT '(' STRING_LITERAL ',' expression ')'
        | COUT '<''<' STRING_LITERAL
        ;


declaration
        : type_specifier init_declarator_list ';'
        ;


statement
        : compound_statement
        | expression_statement
        | iteration_statement
        | conditional_statement
        ;

conditional_statement
        : IF '('

          conditional expression ')'
```

```
        {
                        fprintf(fp_icg, "L%d:\n", ln);
                        fprintf(fp_quad, "\tLabel\t\t  \t\t  \t\tL%d\n", ln);
                        ln++;
        }
        |
        ;

iteration_statement
        :FOR '(' expression_statement
                {
                        fprintf(fp_icg,"L%d:\n", ln);
                        fprintf(fp_quad, "\tLabel\t\t  \t\t  \t\tL%d\n", ln);
                        ln++;
                }
        expression_statement
                {
                        fprintf(fp_icg, "ifFalse t%d goto L%d\n", --tempno, ln);
                        fprintf(fp_quad, "\tifFalse\t\tt%d\t\t  \t\tL%d\n", tempno, ln);
                        tempno++;
                }
        expr  ')' statement
                {

                        //expression 3

                        switch(exprno){
                                case 0:
                                                sprintf(buffer,"t%d",tempno++);
                                                m = strlen(buffer);
                                                buffer[m] = '\0';
                                                fprintf(fp_icg, "%s = %s + 1\n",buffer, $7);
                                                fprintf(fp_quad, "\t+\t\t%s\t\t1 \t\t%s\n", $7, buffer);

                                                fprintf(fp_icg, "%s = %s\n", $7, buffer);
                                                fprintf(fp_quad, "\t=\t\t%s\t\t  \t\t%s\n", buffer, $7);
                                                break;

                                        case 1:
                                                sprintf(buffer,"t%d",tempno++);
                                                m = strlen(buffer);
                                                buffer[m] = '\0';
                                                fprintf(fp_icg, "%s = %s - 1\n",buffer, $7);
                                                fprintf(fp_quad, "\t-\t\t%s\t\t1 \t\t%s\n", $7, buffer);

                                                fprintf(fp_icg, "%s = %s\n", $7, buffer);
                                                fprintf(fp_quad, "\t=\t\t%s\t\t  \t\t%s\n", buffer, $7);
                                                break;

                                        case 2:
                                                sprintf(buffer,"t%d",tempno++);
                                                m = strlen(buffer);
```

```
}


int main(int argc, char *argv[]){
        int printflag=0;
        if(argc>1)
        {
                if(!strcmp(argv[1],"--print"))
                        printflag=1;
                else
                        {
                                printf("Invalid Option!\n");
                                return 0;
                        }
        }

        fp_icg          = fopen("icg.txt", "w");
        fp_quad         = fopen("quad.txt", "w");
        printf("\n");


        fprintf(fp_quad, "\tOp\t\tArg1\t\tArg2\t\tRes\n");
        fprintf(fp_quad, "--------------------------------------------------------------------------------\n");

        yyparse();

        fclose(fp_icg);


        if(printflag)
        {
                printf("\n\nIntermediate Code\n\n");
                system("cat icg.txt");
        }




        fclose(fp_quad);
        if(printflag)
        {
                printf("\n\nQuadruple Format\n\n");
                system("cat quad.txt");
        }



        return 0;
}
```

OUTPUT

Test Case 1

```
#include<iostream>
using namespace std;
int main()
{
        int i=0;
        int a=0;
        for(i=0;i<10;i++)
        {
                a=a+i;
        }
        a=2*a-1;
}
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/4_Intermediate_Code_Generator$ lex ICG.l
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/4_Intermediate_Code_Generator$ lex -d ICG.l
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/4_Intermediate_Code_Generator$ yacc -d ICG.y
ICG.y:783.11-18: warning: POSIX Yacc does not support string literals [-Wyacc]
  783 |         | "INC_OP"      {       unaryop = 4; }
      |           ^~~~~~~~
ICG.y:784.11-18: warning: POSIX Yacc does not support string literals [-Wyacc]
  784 |         | "DEC_OP"      {       unaryop = 5; }
      |           ^~~~~~~~
ICG.y: warning: 120 shift/reduce conflicts [-Wconflicts-sr]
ICG.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/4_Intermediate_Code_Generator$ cc lex.yy.c y.tab.c
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/4_Intermediate_Code_Generator$ ./a.out < TestCases/forloop.c

--(end of buffer or a NUL)
--accepting rule at line 45 ("#")
--accepting rule at line 46 ("include")
--accepting rule at line 145 ("<")
--accepting rule at line 53 ("iostream")
--accepting rule at line 146 (">")
--accepting rule at line 28 ("
")
--accepting rule at line 48 ("using namespace std")
--accepting rule at line 126 (";")
--accepting rule at line 28 ("
")
--accepting rule at line 34 ("int")
--accepting rule at line 152 (" ")
--accepting rule at line 73 ("main")
--accepting rule at line 132 ("(")
--accepting rule at line 133 (")")
--accepting rule at line 28 ("
")
--accepting rule at line 127 ("{")
--accepting rule at line 28 ("
")
--accepting rule at line 152 (" ")
--accepting rule at line 34 ("int")
--accepting rule at line 152 (" ")
--accepting rule at line 73 ("i")
--accepting rule at line 129 (",")
--accepting rule at line 152 (" ")
--accepting rule at line 73 ("a")
--accepting rule at line 129 (",")
--accepting rule at line 152 (" ")
--accepting rule at line 73 ("b")
--accepting rule at line 126 (";")
--accepting rule at line 28 ("
")
--accepting rule at line 152 (" ")
--accepting rule at line 34 ("int")
--accepting rule at line 152 (" ")
--accepting rule at line 73 ("nume")
--accepting rule at line 131 ("=")
--accepting rule at line 78 ("3.45")
--accepting rule at line 126 (";")
--accepting rule at line 152 (" ")
--accepting rule at line 28 ("
")
```

```
--accepting rule at line 83 ("0")
--accepting rule at line 126 (";")
--accepting rule at line 152 ("   ")
--accepting rule at line 73 ("i")
--accepting rule at line 152 (" ")
--accepting rule at line 145 ("<")
--accepting rule at line 152 (" ")
--accepting rule at line 83 ("10")
--accepting rule at line 126 (";")
--accepting rule at line 152 (" ")
--accepting rule at line 73 ("i")
--accepting rule at line 116 ("++")
--accepting rule at line 133 (")")
--accepting rule at line 127 ("{")
--accepting rule at line 28 ("
")
--accepting rule at line 152 ("         ")
--accepting rule at line 73 ("a")
--accepting rule at line 131 ("=")
--accepting rule at line 73 ("i")
--accepting rule at line 126 (";")
--accepting rule at line 28 ("
")
--accepting rule at line 152 (" ")
--accepting rule at line 128 ("}")
--accepting rule at line 28 ("
")
--accepting rule at line 152 (" ")
--accepting rule at line 73 ("i")
--accepting rule at line 131 ("=")
--accepting rule at line 83 ("1")
--accepting rule at line 126 (";")
--accepting rule at line 28 ("
")
--accepting rule at line 128 ("}")
--accepting rule at line 28 ("
")
--accepting rule at line 28 ("
")
--accepting rule at line 28 ("
")
--accepting rule at line 28 ("
")
--(end of buffer or a NUL)
```

In ICG.txt file we get the ICG

```
nume = 3.45
i = 0
L0:
t0 = i < 10
ifFalse t0 goto L1
a = t0
t1 = i + 1
i = t1
goto L0
L1:
i = t1
```

fig. 3.4.3 Output of Test Case 1

Test Case 2

```
#include<iostream>
using namespace std;
int main()
{
        int a=1,b=2,c=10;

        if(a>b){
                printf("\nInside if");
        }
        else{
                printf("\nInside else");
        }

}
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/4_Intermediate_Code_Generator$ ./a.out < TestCases/ifelse.c

--(end of buffer or a NUL)
--accepting rule at line 45 ("#")
--accepting rule at line 46 ("include")
--accepting rule at line 145 ("<")
--accepting rule at line 53 ("iostream")
--accepting rule at line 146 (">")
--accepting rule at line 28 ("
")
--accepting rule at line 48 ("using namespace std")
--accepting rule at line 126 (";")
--accepting rule at line 28 ("
")
--accepting rule at line 34 ("int")
--accepting rule at line 152 (" ")
--accepting rule at line 73 ("main")
--accepting rule at line 132 ("(")
--accepting rule at line 133 (")")
--accepting rule at line 28 ("
")
--accepting rule at line 127 ("{")
--accepting rule at line 28 ("
")
--accepting rule at line 152 (" ")
--accepting rule at line 34 ("int")
--accepting rule at line 152 (" ")
--accepting rule at line 73 ("a")
--accepting rule at line 131 ("=")
--accepting rule at line 83 ("1")
--accepting rule at line 129 (",")
--accepting rule at line 73 ("b")
--accepting rule at line 131 ("=")
--accepting rule at line 83 ("2")
--accepting rule at line 129 (",")
--accepting rule at line 73 ("c")
--accepting rule at line 131 ("=")
--accepting rule at line 83 ("10")
--accepting rule at line 126 (";")
--accepting rule at line 28 ("
")
```

```
--accepting rule at line 132 ("(")
--accepting rule at line 73 ("a")
--accepting rule at line 146 (">")
--accepting rule at line 73 ("b")
--accepting rule at line 133 (")")
--accepting rule at line 127 ("{")
--accepting rule at line 28 ("
")
--accepting rule at line 152 ("         ")
--accepting rule at line 68 ("printf")
--accepting rule at line 132 ("(")
--accepting rule at line 101 (""\nInside if"")
--accepting rule at line 133 (")")
--accepting rule at line 126 (";")
--accepting rule at line 28 ("
")
--accepting rule at line 152 (" ")
--accepting rule at line 128 ("}")
--accepting rule at line 28 ("
")
--accepting rule at line 152 (" ")
--accepting rule at line 66 ("else")
--accepting rule at line 127 ("{")
--accepting rule at line 28 ("
")
--accepting rule at line 152 ("         ")
--accepting rule at line 68 ("printf")
--accepting rule at line 132 ("(")
--accepting rule at line 101 (""\nInside else"")
--accepting rule at line 133 (")")
--accepting rule at line 126 (";")
--accepting rule at line 152 (" ")
--accepting rule at line 28 ("
")
--accepting rule at line 152 (" ")
--accepting rule at line 128 ("}")
--accepting rule at line 28 ("
")
--accepting rule at line 152 (" ")
--accepting rule at line 28 ("
")
--accepting rule at line 128 ("}")
--accepting rule at line 152 (" ")
--accepting rule at line 28 ("
")
--accepting rule at line 28 ("
")
--accepting rule at line 28 ("
")
--(end of buffer or a NUL)
--EOF (start condition 0)
```

```
a = 1
b = 2
c = 10
t0 = a > b
ifFalse t0 goto L0
goto L1
L0:
L1:
```

fig. 3.4.4 Output of Test Case 2

3.4.4 Analyses

The intermediate representation (IR) is a platform-independent, lower-level form of the original code, which retains the logic and structure but is closer to machine code. This representation serves as a bridge, making the code more manageable for optimization and further translation into the target machine language.

3.5 Code Optimizer

3.5.1 Methodology

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand less resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Efforts for an optimized code can be made at various levels of compiling the process.

- Efforts for an optimized code can be made at various levels of compiling the process.
- After generating intermediate code, the compiler can modify the intermediate code by address calculations and improving loops.
- While producing the target machine code, the compiler can make use of memory hierarchy and CPU registers.

Types of Code Optimization

Optimization can be categorized broadly into two types : machine independent and machine dependent.

1.Machine Independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
  item = 10;
  value = value + item;
}while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
  value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

1.Machine Dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block identification

We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:

    a. First statement of a program.
    b. Statements that are the target of any branch (conditional/unconditional).
    c. Statements that follow any branch statement.

- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

Basic blocks are important concepts from both code generation and optimization point of view.

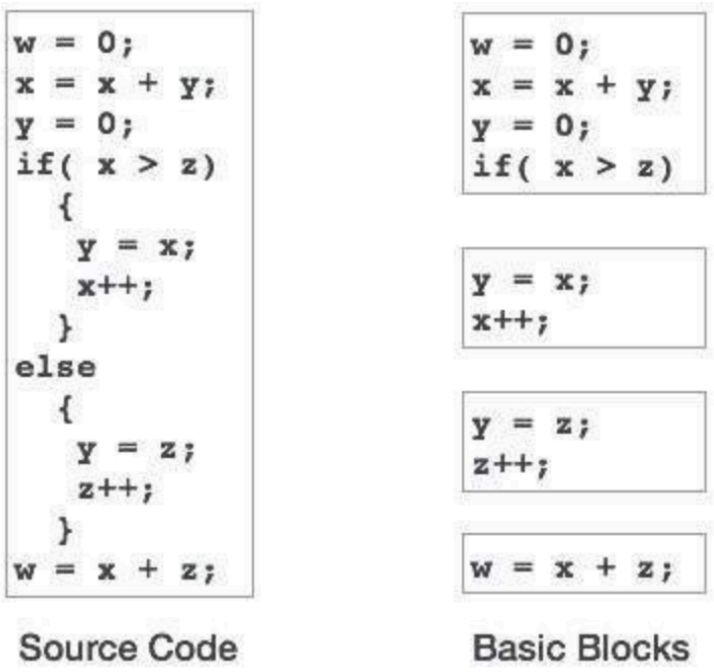


fig 3.5.1 Basic Blocks

Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.

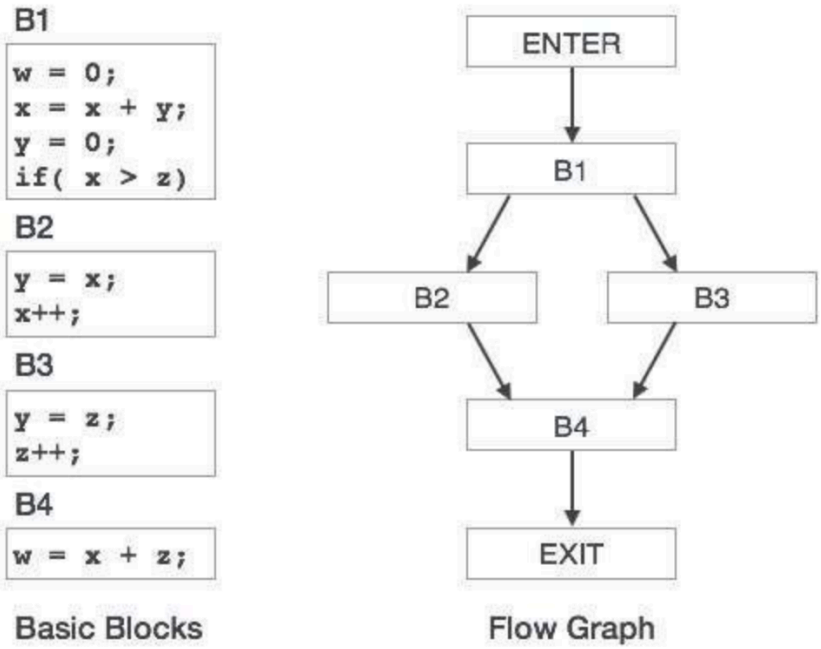


fig. 3.5.2 Flow Graph of Basic Blocks

Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

- **Invariant code** : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
- **Induction analysis** : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
- **Strength reduction** : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

Dead-code Elimination

Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code

There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-codes.

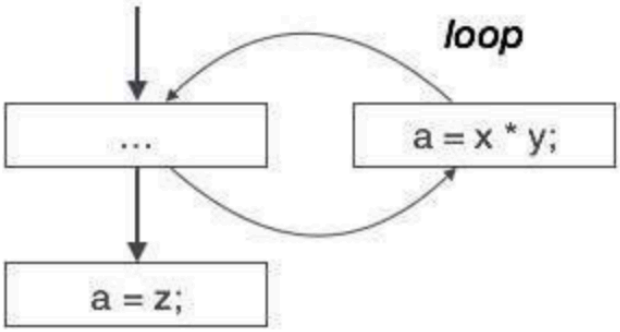


fig. 3.5.3 Loop of partially dead code

The above control flow graph depicts a chunk of program where variable ‘a’ is used to assign the output of expression ‘x * y’. Let us assume that the value assigned to ‘a’ is never used inside the loop.Immediately after the control leaves the loop, ‘a’ is assigned the value of variable ‘z’, which would be used later in the program. We conclude here that the assignment code of ‘a’ is never used anywhere, therefore it is eligible to be eliminated.

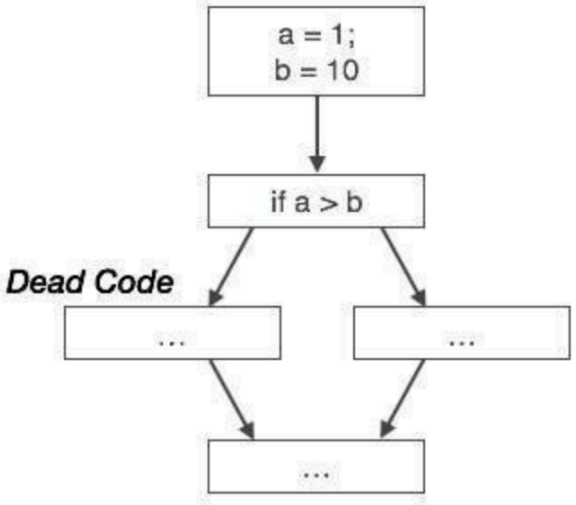


fig. 3.5.4 Flowchart of Dead Code

Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.

3.5 Implementation

Redundant expressions are computed more than once in parallel path, without any change in operands.whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,

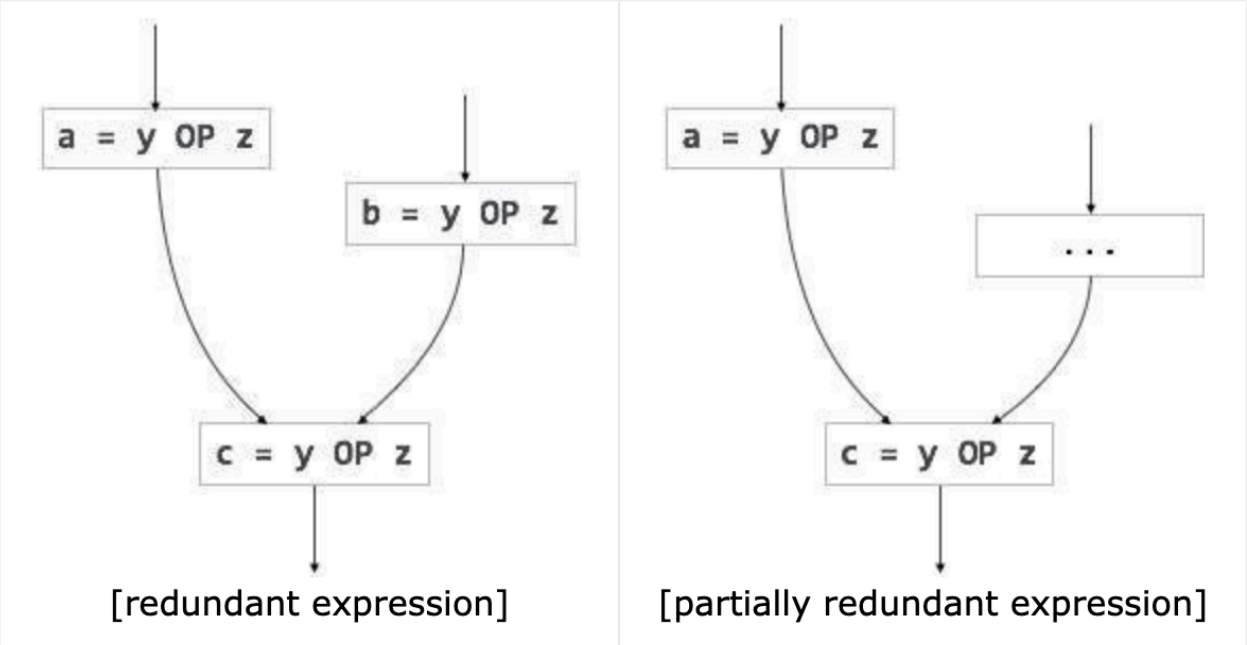


fig. 3.5.5 Redundant Expression

Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique.

Another example of a partially redundant code can be:

```
If (condition)
{
  a = y OP z;
}
else
```

```
{
  ...
}
c = y OP z
```

We assume that the values of operands (**y** and **z**) are not changed from assignment of variable **a** to variable **c**. Here, if the condition statement is true, then y OP z is computed twice, otherwise once. Code motion can be used to eliminate this redundancy, as shown below:

```
If (condition)
{
  ...
  tmp = y OP z;
  a = tmp;
  ...
}
else
{
  ...
  tmp = y OP z;
}
c = tmp;
```

Here, whether the condition is true or false; y OP z should be computed only once.

3.5.3 Results

optimizer.py

```
import re
import sys
import operator

printContent=False

istemp = lambda s : bool(re.match(r"^t[0-9]*$", s))
isid = lambda s : bool(re.match(r"^[A-Za-z][A-Za-z0-9_]*$", s))

if len(sys.argv) == 2 :
    icg_file = str(sys.argv[1])
elif len(sys.argv) == 3:
    if(sys.argv[2]=="--print"):
        icg_file = str(sys.argv[1])
        printContent=True


def eval_binary_expr(op1, oper, op2):
    op1,op2 = int(op1), int(op2)
    if(oper=='+'):
        return op1+op2
    elif(oper=='-'):
        return op1-op2
    elif(oper=='*'):
        return op1*op2
    elif(oper=="/"):
        return op1/op2
    elif(oper=="<"):
        return op1<op2
    elif(oper==">"):
        return op1>op2
    elif(oper=="||"):
        return op1 or op2
    elif(oper=="&&"):
        return op1 and op2



def printICG(list_of_lines):
    for i in list_of_lines:
        print(i.strip())

def add_to_dict(x,y):
    temp_vars[x]=y
```

```
                s=s.join([x[0],x[1],x[2],x[3],val,"\n"])
                list_of_lines[i]=s
                return list_of_lines
        elif x[4].isdigit():
            if(x[2] in temp_vars):
                s=" "
                val=temp_vars[x[2]]
                s=s.join([x[0],x[1],val,x[3],x[4],"\n"])
                list_of_lines[i]=s
                return list_of_lines
    else:
        if(x[2] in temp_vars):
            if(x[4] in temp_vars):
                s=" "
                val1=temp_vars[x[2]]
                val2=temp_vars[x[4]]
                s=s.join([x[0],x[1],val1,x[3],val2,"\n"])
                list_of_lines[i]=s
                return list_of_lines
            else:
                val=temp_vars[x[2]]
                s=" "
                s=s.join([x[0],x[1],val,x[3],x[4],"\n"])
                list_of_lines[i]=s
                return list_of_lines
        if(x[4] in temp_vars):
            val=temp_vars[x[4]]
            s=" "
            s=s.join([x[0],x[1],x[2],x[3],val,"\n"])
            list_of_lines[i]=s
            return list_of_lines


def constant_propagation():
    #print(list_of_lines)
    for i in range(len(list_of_lines)):
        temp_list=list_of_lines[i].split()
        l=len(temp_list)
        if(l==3):
            add_to_dict(temp_list[0],temp_list[2])
        elif(l==5):
            func1(temp_list,i)

def remove_dead_code(list_of_lines) :
    num_lines = len(list_of_lines)
    temps_on_lhs = set()
    for line in list_of_lines :
        tokens = line.split()
        if istemp(tokens[0]) :
            temps_on_lhs.add(tokens[0])

    useful_temps = set()
```

```
            if tokens[1] :
                useful_temps.add(tokens[1])

    temps_to_remove = temps_on_lhs - useful_temps
    new_list_of_lines = []
    for line in list_of_lines :
        tokens = line.split()
        if tokens[0] not in temps_to_remove :
            new_list_of_lines.append(line)
    if num_lines == len(new_list_of_lines) :
        return new_list_of_lines
    return remove_dead_code(new_list_of_lines)

if __name__ == "__main__" :
        list_of_lines = []
        f = open(icg_file, "r")
        for line in f :
            list_of_lines.append(line)
        f.close()
        temp_list=list()
        temp_vars=dict()

        if(printContent):
            print('ICG: ')
            printICG(list_of_lines)
            print("\n\n")

            print('After Constant Propagation')
            constant_propagation()
            printICG(list_of_lines)
            print("\n\n")


            print('After Constant Folding: ')
            constant_folding(list_of_lines)
            printICG(list_of_lines)
            print("\n\n")

            print('After Dead Code Elimination: ')
            list_of_lines=remove_dead_code(list_of_lines)
            list_of_lines=remove_dead_code(list_of_lines)
            printICG(list_of_lines)
            fopt = open("optimized_icg.txt", "w")
            fopt.writelines(list_of_lines)
            fopt.close()
        else:
            constant_propagation()
            constant_folding(list_of_lines)
            list_of_lines=remove_dead_code(list_of_lines)
            list_of_lines=remove_dead_code(list_of_lines)
            fopt = open("optimized_icg.txt", "w")
            fopt.writelines(list_of_lines)
            fopt.close()
```

OUTPUT

Test Case 1

ICG Output of previous phase

```
nume = 3.45
i = 0
L0:
t0 = i < 10
ifFalse t0 goto L1
a = t0
t1 = i + 1
i = t1
goto L0
L1:
i = t1
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/5_Code_Optimizer$ python optimizer.py input.txt
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/5_Code_Optimizer$
```

Optimized ICG

```
1 nume = 3.45
2 i = 0
3 L0:
4 t0 = True
5 ifFalse t0 goto L1
6 a = t0
7 t1 = 1
8 i = t1
9 goto L0
0 L1:
1 i = t1
```

fig. 3.5.6 Output of Test Case 1

Test Case 2

ICG Output

```
a = 1
b = 2
c = 10
t0 = a > b
ifFalse t0 goto L0
goto L1
L0:
L1:
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/5_Code_Optimizer$ python optimizer.py input.txt
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/5_Code_Optimizer$
```

Optimized Output

```
a = 1
b = 2
c = 10
t0 = False
ifFalse t0 goto L0
goto L1
L0:
L1:
```

fig. 3.5.7 Output of Test Case 2

3.5.4 Analyses

The optimized code is a modified version of the IR, where unnecessary operations are eliminated, and the code is restructured for better performance. This phase aims to reduce the execution time and resource consumption without altering the code's semantics, preparing it for efficient execution on the target machine.

3.6 Target Code Generator

3.6.1 Methodology

Code generation can be considered as the final phase of compilation. Through post code generation, optimization processes can be applied on the code, but that can be seen as a part of the code generation phase itself. The code generated by the compiler is an object code of some lower-level programming language, for example, assembly language. We have seen that the source code written in a higher-level language is transformed into a lower-level language that results in a lower-level object code, which should have the following minimum properties:

- It should carry the exact meaning of the source code.
- It should be efficient in terms of CPU usage and memory management.

We will now see how the intermediate code is transformed into target object code (assembly code, in this case).

Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

For example:

$t_0 = a + b$

$t_1 = t_0 + c$

$d = t_0 + t_1$

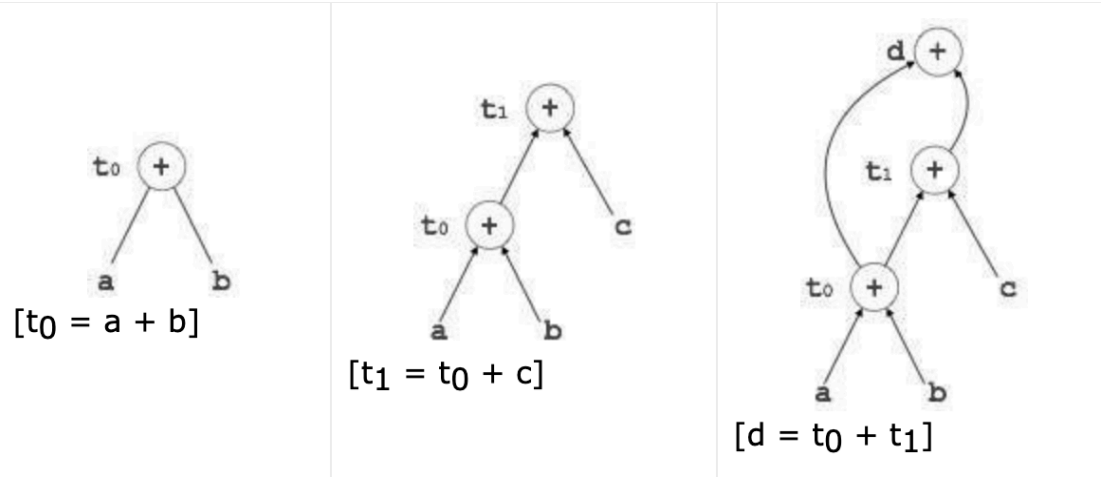


fig. 3.6.1 Directed Acyclic graph

Peephole Optimization

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination

At source code level, the following can be done by the user:

```
int add_ten(int x)
  {
  int y, z;
  y = 10;
  z = x + y;
  return z;
  }
```

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- MOV x, R0
- MOV R0, R1

We can delete the first instruction and rewrite the sentence as:

```
MOV x, R1
```

Unreachable code

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidentally written a piece of code that can never be reached.

For example:

```
void add_ten(int x)
{
  return x + 10;
  printf("value of x is %d", x);
}
```

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

Flow of control optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...
MOV R1, R2
GOTO L1
...
L1 :   GOTO L2
L2 :   INC R1
```

In this code,label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...
MOV R1, R2
GOTO L2
...
L2 :   INC R1
```

Algebraic expression simplification

There are occasions where algebraic expressions can be made simple. For example, the expression **a = a + 0** can be replaced by **a** itself and the expression a = a + 1 can simply be replaced by INC a.

Strength reduction

There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

For example, **x * 2** can be replaced by **x << 1**, which involves only one left shift. Though the output of a * a and a^2 are the same, a^2 is much more efficient to implement.

Accessing machine instructions

The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much more efficiently. If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results.

3.6.2 Implementation

A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

- **Target language** : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.
- **IR Type** : Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.
- **Selection of instruction** : The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.
- **Register allocation** : A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.

- **Ordering of instructions** : At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

Descriptors

The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:

- **Register descriptor** : Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.
- **Address descriptor** : Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

Code generator keeps both the descriptor updated in real-time. For a load statement, LD R1, x, the code generator:

- updates the Register Descriptor R1 that has value of x and
- updates the Address Descriptor (x) to show that one instance of x is in R1.

Code Generation

Basic blocks consist of a sequence of three-address instructions. Code generator takes these sequences of instructions as input.

Note : If the value of a name is found at more than one place (register, cache, or memory), the register's value will be preferred over the cache and main memory. Likewise cache's value will be preferred over the main memory. Main memory is barely given any preference.

getReg : Code generator uses getReg function to determine the status of available registers and the location of name values. getReg works as follows:

- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

For an instruction x = y OP z, the code generator may perform the following actions. Let us assume that L is the location (preferably register) where the output of y OP z is to be saved:

- Call function getReg, to decide the location of L.
- Determine the present location (register or memory) of y by consulting the Address Descriptor of y. If y is not presently in register L, then generate the following instruction to copy the value of y to L:

  MOV y', L
  where **y'** represents the copied value of **y**.

- Determine the present location of z using the same method used in step 2 for y and generate the following instruction:

  OP z', L

  where z' represents the copied value of z.

- Now L contains the value of y OP z, that is intended to be assigned to **x**. So, if L is a register, update its descriptor to indicate that it contains the value of **x**. Update the descriptor of **x** to indicate that it is stored at location **L**.
- If y and z have no further use, they can be given back to the system.

Other code constructs like loops and conditional statements are transformed into assembly language in a general assembly way.

3.6.3 Results

assembly.py

```
import time
import sys
def constant_reg(stmt, regval, value):
    lstmt = "MOV "+"R"+str(regval)+"," + "#" + value
    stmt.append(lstmt)
    r1 = regval
    regval = (regval + 1)%13
    return stmt, regval, r1

def variable_reg(stmt, regval, value):
    st1 = "MOV "+"R" + str(regval) + ","+"="+str(value)
    r1 = regval
    regval = (regval + 1)%13
    stmt.append(st1)
    st2 = "MOV "+"R" + str(regval) +","+ "[R" + str(r1) + "]"
    stmt.append(st2)
    r2 = regval
    regval = (regval + 1)%13
    return stmt, regval, r1, r2

def arith_op(stmt, lhs, arg1, op, arg2,type):
    if(type==1):
        if(arg1.isdigit()):
            if(op == "+"):
                st = "ADD "+"R"+str(lhs)+","+"#"+arg1+",R"+arg2
                stmt.append(st)

            elif(op == "-"):
                st = "SUBS "+"R"+str(lhs)+","+"#"+arg1+",R"+arg2
                stmt.append(st)

            elif(op == "*"):
                st = "MUL "+"R"+str(lhs)+","+"#"+arg1+",R"+arg2
                stmt.append(st)

            elif(op == "/"):
                st = "SDIV "+"R"+str(lhs)+","+"#"+arg1+",R"+arg2
                stmt.append(st)
            return stmt
        elif(arg2.isdigit()):
            if(op == "+"):
                st = "ADD "+"R"+str(lhs)+","+"R"+arg1+",#"+arg2
                stmt.append(st)
```

```
                    stmt.append(b_stmt)
                else:
                    stmt, regval, r1,r2 = variable_reg(stmt, regval, p
                    cmp_stmt="CMP "+"R"+str(r2)+",#"+prev_line[4]
                    stmt.append(cmp_stmt)
                    b_stmt="B"+conditions[prev_line[3]]+" "+lines[inde
                    stmt.append(b_stmt)
            else:
                stmt, regval, r1,r2 = variable_reg(stmt, regval, prev_
                stmt, regval, r3,r4 = variable_reg(stmt, regval, prev_
                cmp_stmt="CMP "+"R"+str(r2)+",R"+str(r4)
                stmt.append(cmp_stmt)
                b_stmt="B"+conditions[prev_line[3]]+" "+lines[index-1]
                stmt.append(b_stmt)
        elif(len(i.split()) == 3):
            variable = i.split()[0]
            value = i.split()[2]
            variable = str(variable)
            if variable not in varlist:
                out = ""
                out = out + variable + ":" + " .WORD " + str(value)
                vardec.append(out)
                varlist.append(variable)
            else:
                stmt, regval, r1, r2 = variable_reg(stmt, regval, vari
                stmt, regval, r3 = constant_reg(stmt, regval, value)
                st = "STR R"+str(r3)+", [R" + str(r1) + "]"
                stmt.append(st)
    return vardec, stmt


def file_write(stmt, vardec, File):
    File.write(".text\n")
    for i in stmt:
        File.write("%s\n"%(i))
    File.write("SWI 0x011\n")
    File.write(".DATA\n")
    for i in vardec:
        time.sleep(0.01)
        File.write("%s\n"%(i))

if __name__=="__main__":
    if(len(sys.argv)!=2):
        print("Usage: python gen.py [filename]")
        exit()
    fin = open(sys.argv[1], "r")
    fout = open(sys.argv[1].split('.')[0]+".s", "w")
    lines = fin.readlines()
    print('Compiling......')
    vardec, stmt = icg_to_asm(lines, fout)
    file_write(stmt, vardec, fout)
```

OUTPUT

Test Case 1

```
nume = 3.45
i = 0
L0:
t0 = i < 10
ifFalse t0 goto L1
a = t0
t1 = i + 1
i = t1
goto L0
L1:
i = t1
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/6_Target_Code_Generator$ python assembly.py icg.txt
Compiling......
Assembly code dumped to:  icg.s
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/6_Target_Code_Generator$
```

```
.text
L0:
MOV R0,=i
MOV R1,[R0]
CMP R1,#10
BGE L1
MOV R2,=i
MOV R3,[R2]
MOV R4,=t1
MOV R5,[R4]
ADD R5,#3,R1
STR R5, [R4]
MOV R6,=i
MOV R7,[R6]
MOV R8,#t1
STR R8, [R6]
B L0
L1:
MOV R9,=i
MOV R10,[R9]
MOV R11,#t1
STR R11, [R9]
SWI 0x011
.DATA
nume: .WORD 3.45
i: .WORD 0
a: .WORD t0
```

fig. 3.6.2 Output of Test Case 1

Test Case 2

```
a = 1
b = 2
c = 10
t0 = a > b
ifFalse t0 goto L0
goto L1
L0:
L1:
```

```
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/6_Target_Code_Generator$ python assembly.py icg.txt
Compiling......
Assembly code dumped to:  icg.s
nithin@nithin1729s:~/Codes/Projects/C Compiler Phases/6_Target_Code_Generator$
```

```
.text
MOV R0,=a
MOV R1,[R0]
MOV R2,=b
MOV R3,[R2]
CMP R1,R3
BLE L0
B L1
L0:
L1:
SWI 0x011
.DATA
a: .WORD 1
b: .WORD 2
c: .WORD 10
```

fig. 3.6.3 Output of Test Case 2

3.6.4 Analyses

The final output is the executable machine code tailored to the target platform's specifications. This phase translates the IR into the specific instructions understood by the target machine, considering its architecture, instruction set, and conventions. The result is a ready-to-run program that faithfully executes the logic defined in the original source code.

CHAPTER 4 : Conclusion

A simple C compiler was successfully built and the output for each phase is shown for a sufficient number of test cases.

GitHub Link: https://github.com/Nithin1729S/C-Compiler-Phases

CHAPTER 5 : Future Scope

The future scope of compiler research and development holds exciting possibilities for advancing the state-of-the-art in programming language implementation, optimization techniques, and tooling support. Several key areas present opportunities for innovation and exploration:

Machine Learning-Based Optimization: Integrating machine learning techniques into compiler optimization opens up new avenues for automatic program analysis and transformation. Deep learning models can be trained to recognize patterns in code and make informed decisions for optimization, such as loop unrolling, vectorization, and parallelization. Reinforcement learning algorithms can dynamically adapt optimization strategies based on feedback from program execution, leading to more efficient and adaptive compilers.

Quantum Computing Compilation: With the advent of quantum computing, there is a pressing need for compilers tailored to quantum programming languages and architectures. Compiler research in this domain focuses on translating high-level quantum algorithms into executable quantum circuits, optimizing quantum gate placement and resource utilization, and addressing error correction and noise mitigation challenges. Quantum-aware compilation techniques aim to exploit the unique properties of quantum hardware to maximize performance and scalability.

Heterogeneous Computing Compilation: As computing architectures become increasingly heterogeneous, compilers must support efficient code generation for diverse hardware accelerators such as GPUs, TPUs, and FPGAs. Compiler research in this area focuses on polyhedral optimization techniques, domain-specific languages (DSLs), and compiler frameworks for expressing and optimizing parallelism across heterogeneous computing resources. Additionally, tools for automatic offloading and task scheduling help leverage the full potential of heterogeneous systems for accelerating compute-intensive workloads.

High-Level Synthesis (HLS): HLS compilers enable the synthesis of hardware designs from high-level programming languages such as C, C++, and OpenCL. Future research in HLS focuses on improving the productivity, performance, and scalability of hardware design synthesis by integrating advanced optimization techniques, automated design space exploration, and architectural modeling. HLS tools that target emerging technologies such as neuromorphic computing and approximate computing enable rapid prototyping and exploration of novel hardware architectures.

Domain-Specific Compilation: Domain-specific languages (DSLs) and compilers tailored to specific application domains enable developers to express complex algorithms and optimizations concisely and efficiently. Future research in domain-specific compilation focuses on language design, compiler optimization, and tooling support for emerging domains such as machine learning, data analytics, bioinformatics, and quantum computing. Additionally, techniques for cross-domain optimization and interoperability facilitate the integration of domain-specific languages with general-purpose programming languages and libraries.

Compiler Tooling and Developer Productivity: Improving compiler tooling and developer productivity is essential for enhancing the software development experience. Future research in this area includes the development of interactive programming environments, intelligent code completion and refactoring tools, and compiler diagnostics for detecting common programming errors and performance bottlenecks. Integration with IDEs, version control systems, and collaborative development platforms enhances collaboration and code quality assurance.

CHAPTER 6 : References

M. Upadhyaya, "Simple calculator compiler using Lex and YACC," 2011 3rd International Conference on Electronics Computer Technology, Kanyakumari, India, 2011, pp. 182-187, doi: 10.1109/ICECTECH.2011.5942077.

D. A. Ladd and J. C. Ramming, "A*: a language for implementing language processors," in IEEE Transactions on Software Engineering, vol. 21, no. 11, pp. 894-901, Nov. 1995, doi: 10.1109/32.473218.

A. N. Likhith, K. Gurunadh, V. Chinthapalli and M. Belwal, "Compiler For Mathematical Operations Using English Like Sentences," 2023 7th International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS), Bangalore, India, 2023, pp. 1-6, doi: 10.1109/CSITSS60515.2023.10334225.

S. Sridhar and S. Sanagavarapu, "A Compiler-based Approach for Natural Language to Code Conversion," 2020 3rd International Conference on Computer and Informatics Engineering (IC2IE), Yogyakarta, Indonesia, 2020, pp. 1-6, doi: 10.1109/IC2IE50715.2020.9274674. ,

M. Mernik and V. Zumer, "An educational tool for teaching compiler construction," in IEEE Transactions on Education, vol. 46, no. 1, pp. 61-68, Feb. 2003, doi: 10.1109/TE.2002.808277.