Syntax Analysis Phase Report

20.02.2018

Harita Reddy - 15CO217

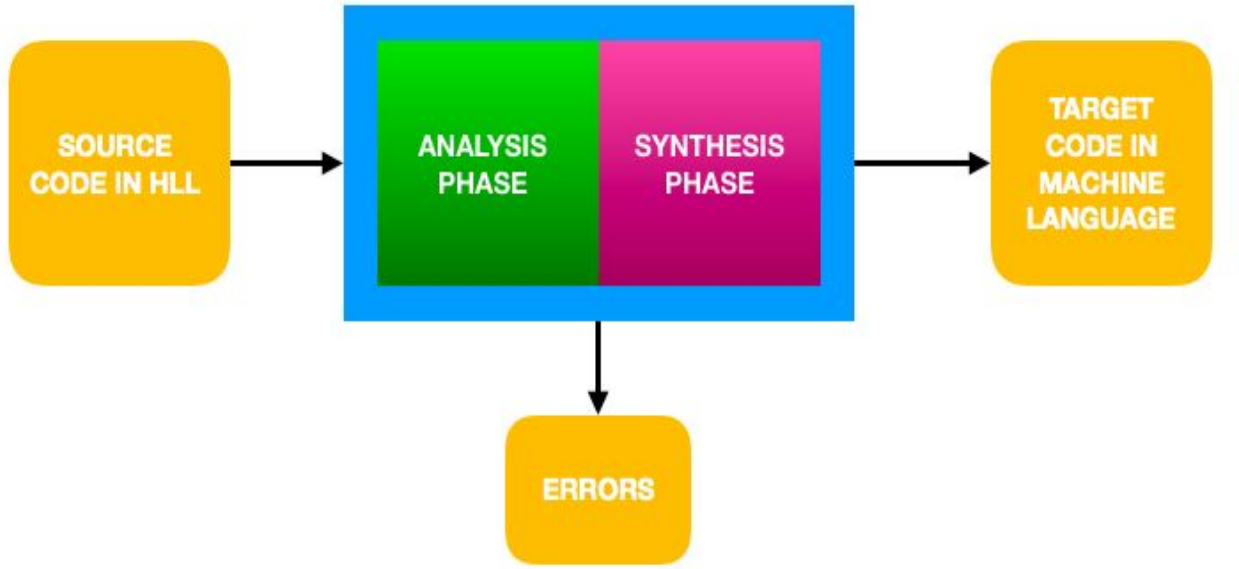
Manali Gala - 15CO214

Contents

1. Introduction to Compiler 2
2. Phases of Compiler 3
3. Syntax Analyser (Parser)
    - Introduction 4
    - Limitations of Regular Grammar 5
    - Context Free Grammar 5
    - Parse Tree 5
    - Left Recursion 6
    - Type of Parsers 6
4. Our implementation of Syntax Analyser
    - Yacc 7
    - Tokens Returned by Lexical Analyzer 7
    - Expressions 7
    - Data Types and Functions 8
    - Basic Code Body 9
    - Statements 9
    - If- Else 9
    - Conditionals 10
    - Keywords 10
    - Printing Errors 10
    - Symbol and Constant Tables 10
5. Lexical and Parser Code 11
6. Output and Screenshots
    - Test Cases and Screenshots 19
    - Parse Trees 22
7. Conclusion 24

Introduction to Compiler

A compiler is a software which converts a source code (program) written in high level language like C, which is also known as **Source Language** to low level language called as **Target Language**, which is machine understandable code. A compiler also throws errors in the source code.



The objectives of a compiler are as follows:

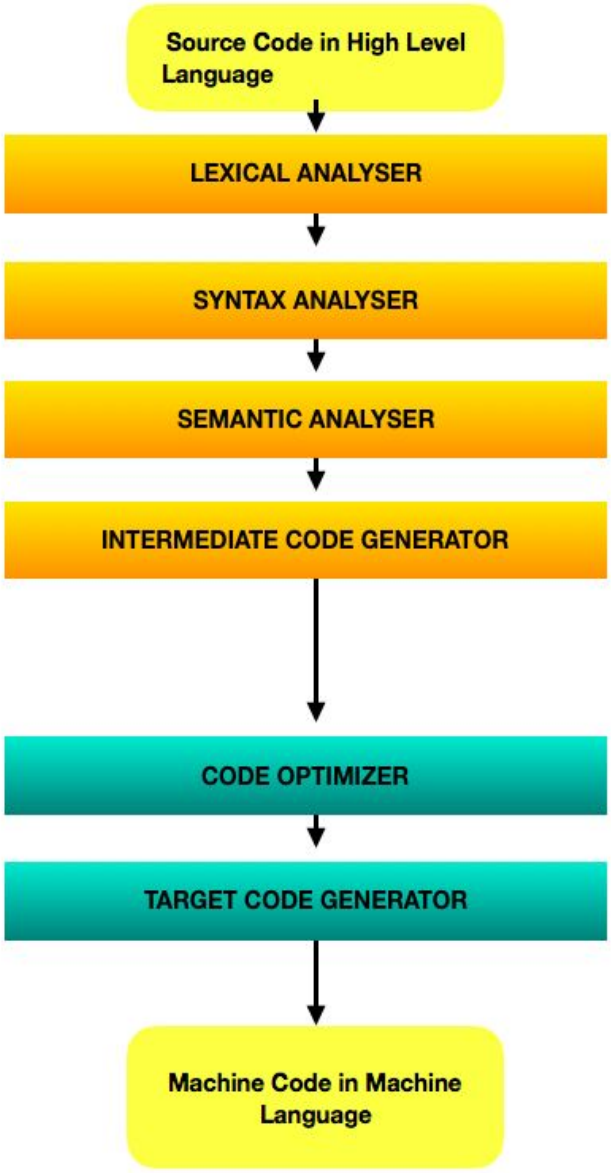
1. **Error Handling Feature:** A compiler generally gives the line where the error is found, i.e, the line number, error message and a possible suggestion. The error message has to be given in a way that programmer can understand.
2. **Machine Code should be proportional to size of code:** The machine code should not be very large compared to the source code; it will affect both storage and the speed.
3. **Should be fast:** The compiler itself should run fast enough.

Phases of a Compiler

The compiler is mainly divided into two phases-

i) **Frontend**- This comprises of the Lexical Analyzer, Syntax Analyzer, Semantic Analyzer and the Intermediate Code Generator. It is also known as the **Analysis Phase.**

ii) **Backend**- This comprises of Code Optimization and Final Code Generation. This is also known as **Synthesis Phase.**



LEXICAL ANALYSIS: In the this phase, the source code is read character by character and line by line to form groups of meaningful words from the code.

SYNTACTICAL ANALYSIS: It produces syntactical errors and represents the information in the form of tree structure, also known as **parse tree.**

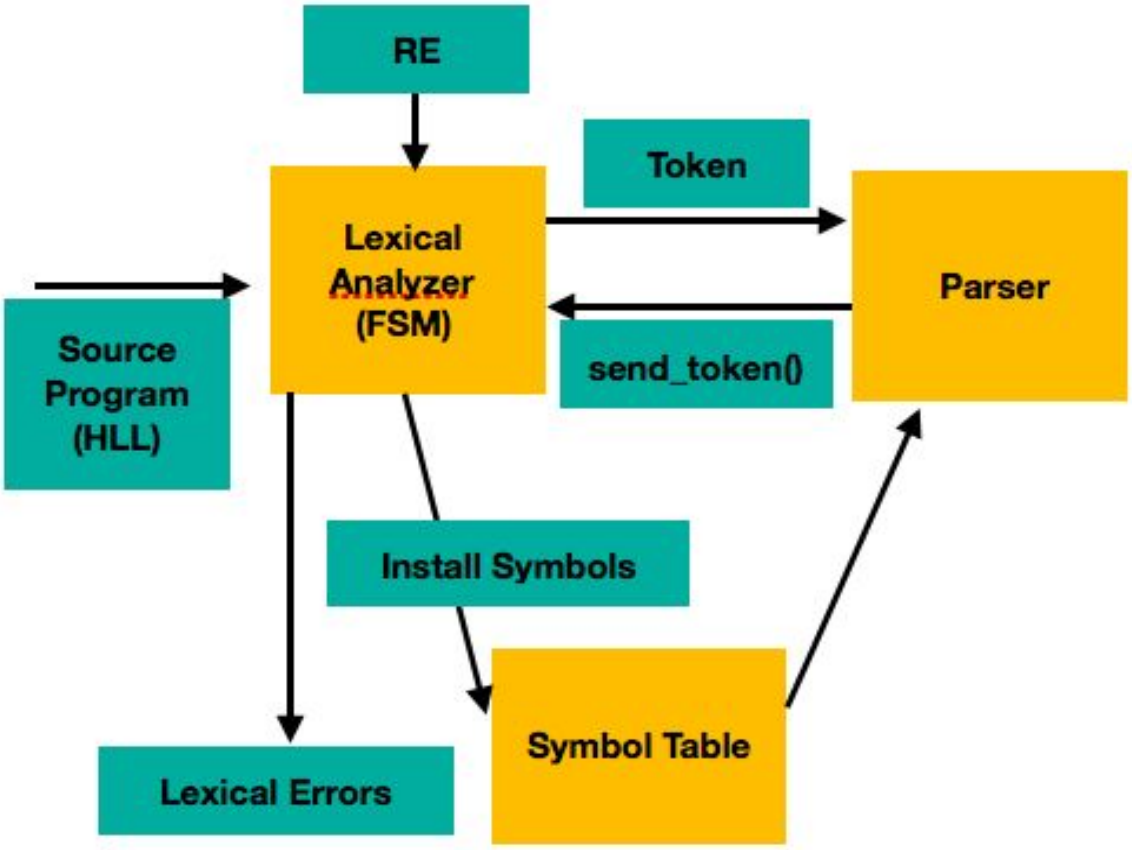
SEMANTIC ANALYSIS: Semantic analysis judges whether the syntax structure constructed in the source program derives any meaning or not.

INTERMEDIATE CODE GENERATOR: The semantically verified parse tree is given as an input to the intermediate code generator that generates the three address code.

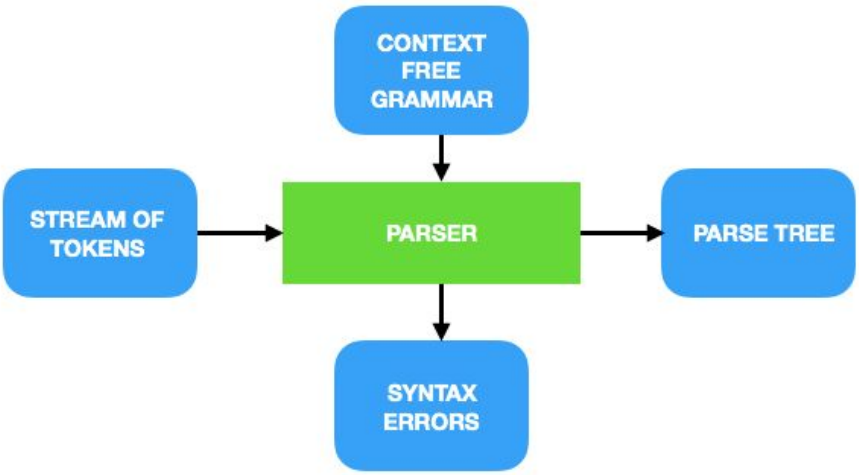
CODE OPTIMIZATION: In this phase, the unwanted information is removed and optimization is performed.

FINAL CODE GENERATION: The object code is generated by this phase and this is machine code in machine language.

Syntax Analyser (Parser)



The syntax of a given sentence cannot be checked by a lexical analyser due to the limitations of the regular expressions. This phase uses context free grammar and is also called as **parsing.** It produces syntactical errors and represents the information in the form of tree structure, also known as **parse tree.** Tokens provided by lexical analyser are grouped into meaningful sentences(larger constructs) at this stage. This stage aims to recover the structure of the series of tokens in a hierarchical manner and give **syntax**



errors if the tokens do not encode a structure defined by our grammar.

Limitations of Regular Grammar

Regular grammar is not very aggressive and has some limitations. They are usually too weak to define programming languages. For example:

1. We cannot create a regular expression that matches all expressions with balanced parentheses
2. We cannot define a regular expression matching all functions with a properly nested block structure.

Context Free Grammar or CFG

A context-free grammar (CFG) is a set of recursive rewriting rules used to generate patterns of strings. These rules are also called as productions.
It is used to describe the source language as regular grammar is not aggressive enough.
Most of the programming languages can be described by Context Free Grammars.
The components of CFG are:

- **Terminal Symbols:** These are characters of the alphabet which can come in the strings generated by the grammar.
- **Nonterminal Symbols:** These are placeholders for patterns of terminal symbols that can be generated by the nonterminal symbols.
- **Production rules:** These are rules for replacing non terminal symbols with combinations of non terminal and terminal symbols provided on the right hand side of the rule.
- **Start Symbol:** This is a special nonterminal symbol that appears in the initial string generated by the grammar.

Parse Tree

A parse tree or parsing tree or syntax tree is an ordered, rooted tree that represents the syntactic structure of a string according to some context-free grammar.
A syntax tree is a compressed representation of the parse tree in which the operators appear as the interior nodes, and the operands of an operator are the children of the node for that operator.
It has the following properties :

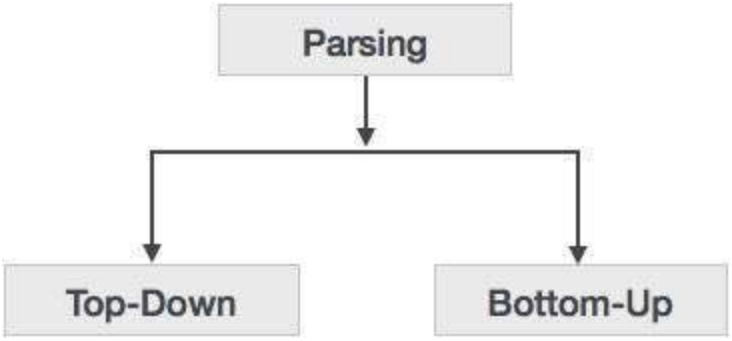
- The root is labeled by the start symbol
- Each leaf is labeled by a token or by epsilon.
- Each interior node is labeled by a non terminal.

- If A is the non terminal labeling some interior node and $X_1,X_2,...,X_n$ are the labels of the children of that node from left to right, then $A->X_1X_2...X_n$ is a production. Here $X_1,X_2,...,X_n$ stand for a symbol that is either a terminal or nonterminal.

Left Recursion

A grammar is said to be left –recursive if it has a non-terminal A such that there is a derivation A =>, for some string . Top–down parsing methods cannot handle left recursive grammars, so a transformation that eliminates left-recursion is needed.

Types of Parsers



Top-down Parsing

When the parser starts constructing the parse tree from the start symbol and then tries to transform the start symbol to the input, it is called top-down parsing.

- Recursive descent parsing : It is a common form of top-down parsing. It is called recursive as it uses recursive procedures to process the input. Recursive descent parsing suffers from backtracking.
- Backtracking : It means, if one derivation of a production fails, the syntax analyzer restarts the process using different rules of same production. This technique may process the input string more than once to determine the right production.

Bottom-up Parsing

As the name suggests, bottom-up parsing starts with the input symbols and tries to construct the parse tree up to the start symbol.

Symbol Table is a common data structure (usually implemented by hashing) used to store the identifiers and their related information.

Our Implementation of Syntax Analyzer

We have implemented the syntactical analysis phase using yacc (yet another compiler compiler). It is a parser generator for the Unix Operating System.

Yacc

- Yacc provides a general tool for imposing structure on the input to a computer program. It 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.
- Yacc is written in a portable dialect of C and the actions, and output subroutine, are in C as well. Moreover, many of the syntactic conventions of Yacc follow C.
- The heart of the input specification is a collection of grammar rules. Each rule describes an allowable structure and gives it a name.

Tokens returned by the lexical analyser

The following tokens are returned by the lexical analyser to the parser while scanning the program:

ID NUM WHILE TYPE CHARCONST COMPARE PREPRO MAIN INT RETURN IF ELSE STRUCT UNARYOP STATEKW STRING

Also, precedence is given to the operators at the beginning of the program with * and / having the highest precedence and + and - having the lowest. It follows left associativity.

Both single line and multiline comments are ignored by the code.

Expressions

Expressions are specified by the following production rules:

```
E : E'+'E
|E'-'E
```

|E'*'E
|E'/'E
|'('E')'
|ID
|NUM
|CHARCONST
|STRING
|ID '(' arguments ')'
| UNARYOP ID
| ID UNARYOP

We support addition, subtraction, multiplication and division operations between two expressions. Expressions can also be present between parentheses (for eg (a+b)). Identifier, number (float or integer), character constant and string are the terminals which E can give. We have also included function call (for eg func(10)) as an expression. Apart from this unary operators like ++ and -- applied on expressions are also considered as expressions. As an example, a++, ++b, --a, c-- all are expressions in our implementation.

Data Types and Functions

The data types our parser supports are int, float and char. The data types for variable, constants, and functions are inserted in the symbol table and constant table.
The declarations are given by the following production rules :
declarations: declarations B
|B
;
B: TYPE ID ';'
;

We have handled function declarations, definitions and call statements through the following production rules.

function : TYPE ID '(' arguments ')' '{' code return '}'
| ID '(' arguments ')' ';'
|TYPE ID '(' arguments ')' ';'
;

The parameters to a function are comma separated values. The parameters/arguments to functions are handled as follows:

arguments: arguments ',' D
|D
|

;
D: TYPE ID
|ID
;

Basic Code Body

The basic code, whether it is inside the main function or outside the main is implemented using the following production rule:

code: code A
|
;
A can be either a statement ending with a semicolon, a while loop, an if else statement, a function declaration, function definition, function call statement or a unary statement ending with a semicolon. All these can be nested inside each other. Few examples are as follows:

A: whileloop
|statement ';'
|ifelse
|function
|unaryst ';'
;

Statements

Statements in the code can be of the following form:

TYPE ID	(Variable Declaration)
ID'='E	(Identifier = Expression)
TYPE ID'='E	(Identifier declaration and assignment to an expression)
STRUCT ID '{' declarations '}'	(Structure Declaration, with variable declarations inside it)
TYPE '*' ID	(Pointer declaration)
TYPE '*' '*'ID	(Double Pointer declaration)
STATEKW	(Keywords that can form statements, eg: break;, continue;)
TYPE '*' ID '=' E	(Pointer declaration and assignment)
TYPE '*''*' ID '=' E	(Double Pointer declaration and assignment)

If Else Statement

Our code supports if -else statements, with multiple else if's, as well as nested if else statements. There is no ambiguity in the given production rules.

ifelse : IF '(' conditions ')' '{' code '}'
| IF '(' conditions ')' S ';'
| IF '(' conditions ')' '{' code '}' ELSE '{' code '}'

|IF '(' conditions ')' S ';' ELSE '{' code '}'
|IF '(' conditions ')' '{' code '}' ELSE A
|IF '(' conditions ')' S ';' ELSE A

Conditions

Conditions inside if statement and and while loop can be of the form:

E	(A simple expression)
statement	(A statement like an assignment operation)
E COMPARE E	(Comparison between expressions using >, <, <=, >= ==, etc)

Keywords

We support a range of keywords in C through our lexical analyzer through the following regular expression:

("float")|("char")|("auto")|("break")|("case")|("const")|("continue")|("main")|("default")|("do")|("double")|("else")|("enum")|("extern")|("for")|("goto")|("if")|("int")|("long")|("register")|("return")|("short")|("signed")|("sizeof")|("static")|("struct")|("switch")|("typedef")|("union")|("unsigned")|("void")|("volatile")|("while")

Printing Errors

If no errors are found, it prints “Parsing Completed Successfully”. If there is a construct that cannot be derived by the grammar (syntax error) it prints the line number and mentions that it is a syntax error.

Symbol Table and Constant Table

We have implemented the symbol and constant table using hashing. In case of collision, chaining (using linked lists) is done. The hash function is based on the sum of the ASCII values of the characters in the tokens. Based on that, the position of a token in the hash table is calculated. The token name, its type and the number of times it has appeared in the source code is stored in the symbol table. We have used an array of structure pointers which are further chained at their respective places using structure pointers. Whenever a token is identified, we check if it is already present in the symbol table and simply increment the count if it does. Else, we insert the token into the table. Chaining is done through dynamic memory allocation.

The syntax analyser adds the **data type of the declared identifiers** to the symbol table.

Constants and their specific type (string, character, int or float constant) are stored in the constant table in similar fashion.

Code

Lexical code:

```
%{
   #include "y.tab.h"
   #include<stdlib.h>
   extern int yylval;
   struct node
   {
            char name[50];
            char class[50];
            int count;
            struct node* next;
            char type[10];
   } ;
   struct node* symtable[53],*constable[53];
   struct node* ptr;
   int linecount=0;
   int poscalc(char* name) //Calculates position of token in symbol table( hash function)
   {
            int i,pos=0;
            for(i=0;i<strlen(name);i++)
            {
                     pos=pos+name[i]; //ASCII values of chacacters added
            }
            pos=pos%53;
            return pos; //returns hash value
   }

   void insert(char* name,char* class) //Inserts the token in symbol table
   {
            int pos=poscalc(name);
            if(symtable[pos]==NULL) //If there is no element already at that index
            {
                      symtable[pos]=(struct node*)malloc(sizeof(struct node));
                      strcpy(symtable[pos]->name,name);
                     strcpy(symtable[pos]->class,class);
                      symtable[pos]->next=NULL;
                      symtable[pos]->count=1;
            }
            else //Chaining is required
            {
                      int flag=0;
                      struct node* check=symtable[pos];
                      while(check!=NULL)
```

```
                {
                        if(strcmp(check->name,name)==0) //Increments count if token exists
                        {
                        flag=1;
                        check->count++;
                        break;
                }
                        check=check->next;
                }
                if(flag==1)
                return;
                struct node* ptr=(struct node*)malloc(sizeof(struct node));
                strcpy(ptr->name,symtable[pos]->name);
                strcpy(ptr->class,symtable[pos]->class);
                ptr->count=symtable[pos]->count;
                ptr->next=symtable[pos]->next;
                strcpy(symtable[pos]->name,name);
                strcpy(symtable[pos]->class,class);
                symtable[pos]->next=ptr;
                symtable[pos]->count=1;
        }
  }
int poscalccons(char* name) //Calculates position of constant in constant table( hash function)
        {
                int i,pos=0;
                for(i=0;i<strlen(name);i++)
                {
                        pos=pos+name[i]; //ASCII values of chacacters added
                }
                pos=pos%53;
                return pos; //returns hash value

        }
  void insertincons(char* name,char* class) //Inserts token into constant table
  {
        int pos=poscalc(name);
        if(constable[pos]==NULL) //If there is no element already at that index
        {
                constable[pos]=(struct node*)malloc(sizeof(struct node));
                strcpy(constable[pos]->name,name);

                strcpy(constable[pos]->class,class);
                constable[pos]->next=NULL;
                constable[pos]->count=1;
        }
        else //Chaining is required
        {
                int flag=0;
```

```
                struct node* check=constable[pos];
                while(check!=NULL)
                {
                        if(strcmp(check->name,name)==0) //Increments count if already exists
                        {
                        flag=1;
                        check->count++;
                        break;
                }
                        check=check->next;
                }
                if(flag==1)
                return;
                struct node* ptr=(struct node*)malloc(sizeof(struct node));
                strcpy(ptr->name,constable[pos]->name);
                strcpy(ptr->class,constable[pos]->class);
                ptr->count=constable[pos]->count;
                ptr->next=constable[pos]->next;
                strcpy(constable[pos]->name,name);
                strcpy(constable[pos]->class,class);
                constable[pos]->next=ptr;
                constable[pos]->count=1;
        }
  }

%}
letter [a-zA-Z]
digit [0-9]
scomment [/][/].*[\n]
mcommentstart [/][*](.|\n)*[*][/]
WHILE "while"
PREPRO [#].+
MAIN ("main")
RETURN ("return")
IF ("if")
ELSE ("else")
STRUCT ("struct")
kw
("auto")|("case")|("const")|("default")|("do")|("double")|("enum")|("extern")|("for")|("goto")|("long")
|("register")|("short")|("signed")|("sizeof")|("static")|("switch")|("typedef")|("union")|("unsigned")|("
void")|("volatile")
STATEKW ("break")|("continue")
TYPE ("float")|("char")|("int")
ID [a-zA-Z][a-zA-Z0-9_]*
NUM (([0-9]+)|([0-9]*\.[0-9]+)([eE][-+]?[0-9]+)?)
CHARCONST [\'][a-zA-Z][\']
COMPARE [<>]|"<="|">="|"=="|"!="
UNARYOP "++"|"--"
```

```
STRING ["].*["]
%%
[\t ] ;
[\n] {linecount++;}
{scomment} {linecount++;}
{mcommentstart} ;
{WHILE} { insert(yytext,"Keyword"); return WHILE;}
{PREPRO} {return PREPRO;}
{MAIN} {insert(yytext,"Keyword"); return MAIN;}
{RETURN} { insert(yytext,"Keyword"); return RETURN;}
{IF} {insert(yytext,"Keyword");return IF;}
{ELSE} {insert(yytext,"Keyword");return ELSE;}
{STRUCT} {insert(yytext,"Keyword");return STRUCT;}
{kw} {insert(yytext,"Keyword");printf("%s:Keyword\n",yytext);}
{STATEKW} {insert(yytext,"Keyword"); return STATEKW;}
{TYPE} {insert(yytext,"Keyword");yylval = strdup(yytext);
return TYPE;}
{ID} {insert(yytext,"Identifier");yylval = strdup(yytext);
return ID;}
{NUM} { insert(yytext,"Constant");
int isfloat=0; int i;
   for(i=0;i<yyleng;i++)
   {
            if(yytext[i]=='.')
            {
                     isfloat=1;
                     break;
            }
   }
   if(isfloat==1)
   insertincons(yytext,"float");
   else
   insertincons(yytext,"int");
yylval=strdup(yytext);
return NUM;}
{CHARCONST} {insert(yytext,"Character Constant");
insertincons(yytext,"character");
return CHARCONST;}
{COMPARE} {//insert(yytext,"Comparison Operator");
                     return COMPARE;}
{UNARYOP} {//insert(yytext,"Unary Operator");
                       return UNARYOP;}
{STRING} {insert(yytext,"String Constant");
insertincons(yytext,"string");return STRING; }
. return yytext[0];

%%
```

Parser Code:

```
%{
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node
	{
		char name[50];
		char class[50];
		int count;
		struct node* next;
		char type[10];
	} ;
extern struct node* symtable[53];
extern struct node* constable[53];
extern struct node* ptr;
extern int linecount;
void inserttype(char* type,char* name) //Inserts the token in symbol table
	{
		int pos=poscalc(name);

			int flag=0;
			struct node* check=symtable[pos];
			while(check!=NULL&&check->name!=name)
			{
				check=check->next;
			}

			strcpy(symtable[pos]->type,type);
	}
%}
%token ID NUM WHILE TYPE CHARCONST COMPARE PREPRO MAIN INT RETURN IF ELSE STRUCT
UNARYOP STATEKW STRING
%left '+' '-'
%left '*' '/'
%%
ED: program { printf("Parsing Successfully Completed (No error)\n"); }
;
program : PREPRO code main
|main
;
main: TYPE MAIN '(' ')' '{' code return '}'
;
return: RETURN '(' NUM ')' ';'
|RETURN  ';'
```

```
|RETURN  NUM ';'
|RETURN ID ';'
|
;
code: code A
|
;
A: whileloop
|statement ';'
|ifelse
|function
|unaryst ';'
;
whileloop : WHILE '(' E ')' '{' code '}'
|WHILE '(' statement ')' '{' code '}'
|WHILE '(' E COMPARE E ')' '{' code '}'
;
conditions : E
|statement
|E COMPARE E
;
S: statement
|unaryst
;
ifelse :  IF '(' conditions ')' '{' code '}'
| IF '(' conditions ')'  S ';'
| IF '(' conditions ')' '{' code '}' ELSE '{' code '}'
|IF '(' conditions ')' S ';'  ELSE '{' code '}'
|IF '(' conditions ')' '{' code '}' ELSE A
|IF '(' conditions ')' S ';' ELSE A
;
arguments: arguments ',' D
|D
|
;
D: TYPE ID
|ID
;
function : TYPE ID '(' arguments ')' '{' code return '}'
| ID '(' arguments ')' ';'
|TYPE ID '(' arguments ')' ';'
;
statement :ID'='E
| TYPE ID  { inserttype($1,$2);}
| TYPE ID'='E { inserttype($1,$2);}
| STRUCT ID '{' declarations '}'
| STATEKW
| TYPE '*' ID {
```

```
                                        char point[20];
                                        strcpy(point,$1); strcat(point," pointer"); inserttype(point,$3);}
| TYPE '*' '*' ID {
                                        char point[20];
                                        strcpy(point,$1); strcat(point," double pointer"); inserttype(point,$4);}
| TYPE '*' ID '=' E {
                                        char point[20];
                                        strcpy(point,$1); strcat(point," pointer"); inserttype(point,$3);}
| TYPE '*' '*' ID '=' E {
                                        char point[20];
                                        strcpy(point,$1); strcat(point," double pointer"); inserttype(point,$4);}
;
declarations: declarations B
|B
;
B: TYPE ID ';'
;
unaryst: UNARYOP ID
| ID UNARYOP
;
E : E'+'E
|E'-'E
|E'*'E
|E'/'E
|'('E')'
|ID
|NUM
|CHARCONST
|STRING
|ID '(' arguments ')'
| UNARYOP ID
| ID UNARYOP
;
%%
#include <stdio.h>
extern int yylex();
extern int yyparse();
extern FILE *yyin;
main() {
        // open a file handle to a particular file:
        FILE *myfile = fopen("program.txt", "r");
        // make sure it is valid:
        // set lex to read from it
        yyin = myfile;
        // parse through the input until there is no more:
        do {
                yyparse();
        } while (!feof(yyin));
```

```
	int j;
	printf("\n\n\n");
	printf("--SYMBOL TABLE--\n"); //Printing Symbol Table
	printf("\t\tName	Class	\t\tCount\tDataType\n\n");
	for(j=0;j<53;j++)
	{
		ptr=symtable[j];
		if(ptr==NULL)
		continue;
		printf("Position: %d\t",j);
		while(ptr!=NULL)
		{
			printf("%-10s %-15s\t\t%-2d\t",ptr->name,ptr->class,ptr->count);
			if(strlen(ptr->type)!=0)
			printf("%-20s",ptr->type);
			else
			printf("\t\t  ");
			printf("\t|");
			ptr=ptr->next;
		}
		printf("\n");
	}
	printf("\n\n");
	printf("--CONSTANT TABLE--\n"); //Printing Constant Table
	printf("\t\tConstant   Datatype\t\t\tCount\n\n");
	for(j=0;j<53;j++)
	{
		ptr=constable[j];
		if(ptr==NULL)
		continue;
		printf("Position: %d\t",j);
		while(ptr!=NULL)
		{
			printf("%-10s %-15s\t\t%-2d|\t",ptr->name,ptr->class,ptr->count);
			ptr=ptr->next;
		}
		printf("\n");
	}

}
yyerror(char *s) {
	printf("Parsing Unsuccessful Due to Syntax Error\n");
	printf("Error in line: %d\n",linecount+1);
}
```

Output Screenshots

The symbol table and constant tables' screenshots are attached below along with the test cases. Parse Trees are drawn for Test Case 1 and 3.

Test Cases and Screenshots

1.

```
#include<stdio.h>
int main()
{
   int a=5;
   int b=5;
   int d=9;
   while(a==b)
   {
   a=a+b*d;
   b=b-a;
   }
}
```

2.

```
#include<stdio.h>

int main()
{
```

```
 //This is test case 2 of the project
  float f;
  f=0.19;

  char a='m';
  a=a+5

  f=a*0.5;
}
```

```
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$ ./a.out
Parsing Unsuccessful Due to Syntax Error
Error in line: 10



--SYMBOL TABLE--
Position: 0     5          Constant                  1               |
Position: 4     float      Keyword                   1               |
Position: 13    int        Keyword                   1               |
Position: 28    'm'        Character Constant        1               |
Position: 41    0.19       Constant                  1               |
Position: 43    char       Keyword                   1               |
Position: 44    a          Identifier                3       char            |
Position: 49    f          Identifier                3       float           |
Position: 50    main       Keyword                   1               |


--CONSTANT TABLE--
Position: 0     5          int                       1 |
Position: 28    'm'        character                 1 |
Position: 41    0.19       float                     1 |
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$
```

3.

```
#include<stdio.h>

struct node
{
   int var;
   char c;
   float f;
};

int main()
{
   int a=10;

   while(a)
   {
    a=a-1;
    if(a==5)
          break;
   }

   return (0);
}
```

```
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$ ./a.out
Parsing Successfully Completed (No error)



--SYMBOL TABLE--
Position: 0     5          Constant                  1               |
Position: 4     float      Keyword                   1               |
Position: 7     while      Keyword                   1               |
Position: 11    var        Identifier                1               |
Position: 13    int        Keyword                   3               |
Position: 16    ==         Comparison Operator       1               |
Position: 36    return     Keyword                   1               |
Position: 40    break      Keyword                   1               |
Position: 41    struct     Keyword                   1               |
Position: 43    char       Keyword                   1               |
Position: 44    10         Constant                  1       int             |   a          Identifier                  5
 |
Position: 46    c          Identifier                1               |
Position: 48    0          Constant                  1               |     if         Keyword                     1               |
Position: 49    1          Constant                  1               |     f          Identifier                  1               |
Position: 50    main       Keyword                   1               |
Position: 51    node       Identifier                1               |


--CONSTANT TABLE--
Position: 0     5          int                       1 |
Position: 44    10         int                       1 |
Position: 48    0          int                       1 |
Position: 49    1          int                       1 |
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$
```

4.

```
#include<stdio.h>
int main()
{
   int a;
   while()
   {
   a++;
   }
   return (0);
}
```

```
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$ ./a.out
Parsing Unsuccessful Due to Syntax Error
Error in line: 6



--SYMBOL TABLE--
Position: 7     while      Keyword                   1               |
Position: 13    int        Keyword                   2               |
Position: 44    a          Identifier                1       int             |
Position: 50    main       Keyword                   1               |


--CONSTANT TABLE--
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$
```

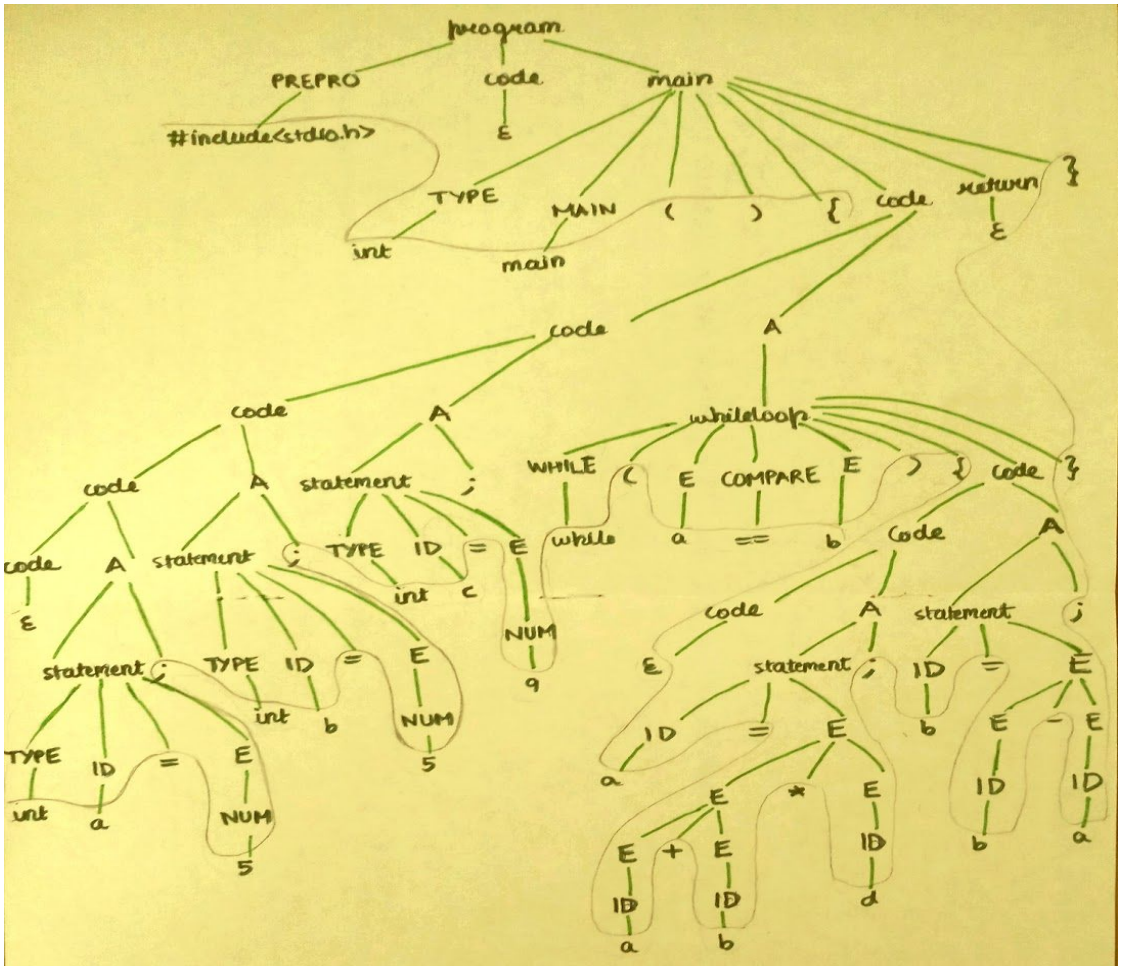
5.

```
#include<stdio.h>

int main()
{
   int a=0;
   int b=2;
   int c=6;
   c=a+*b;
```

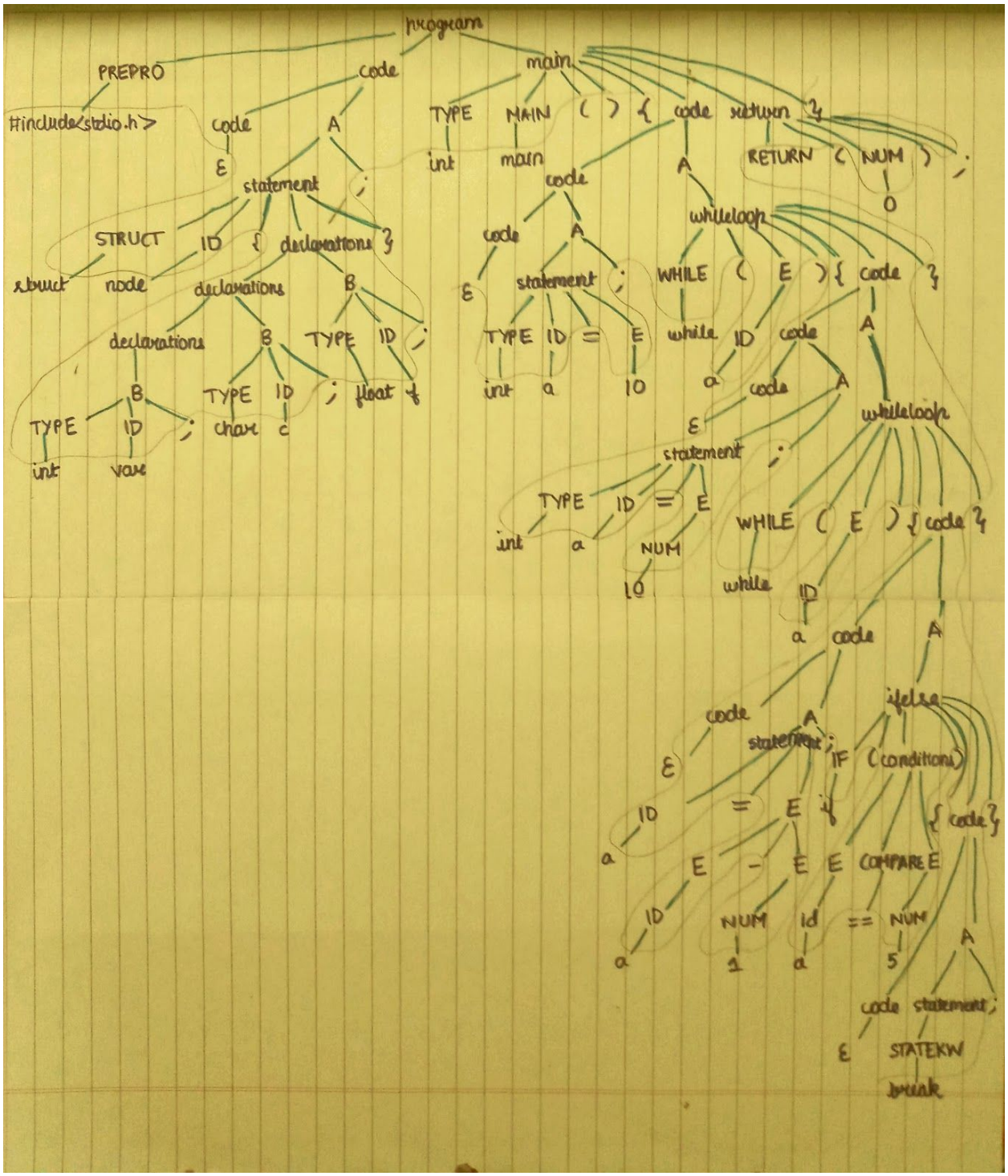
```
}
```

```
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$ ./a.out
Parsing Unsuccessful Due to Syntax Error
Error in line: 8



--SYMBOL TABLE--
Position: 1     6          Constant                    1               |
Position: 13    int        Keyword                     4               |
Position: 44    a          Identifier                  2       int             |
Position: 45    b          Identifier                  1       int             |
Position: 46    c          Identifier                  2       int             |
Position: 48    0          Constant                    1               |
Position: 50    2          Constant                    1               |     main       Keyword                     1               |


--CONSTANT TABLE--
Position: 1     6          int                         1 |
Position: 48    0          int                         1 |
Position: 50    2          int                         1 |
manali@manali-Aspire-E5-573:~/Documents/Computer Science 6th sem/CD/Compiler_Design$
```

PARSE TREE FOR TEST CASE 1



PARSE TREE FOR TEST CASE 3


program
PREPRO
#include<stdio.h>
code
ε
code
A
statement
;
STRUCT
ID
{
declarations
}
struct
node
declarations
B
declarations
B
TYPE
ID
;
B
TYPE
ID
;
float
TYPE
ID
;
char
c
int
var
main
TYPE
MAIN
( )
{
code
return
}
int
main
code
A
RETURN
(
NUM
)
;
0
code
A
whileloop
ε
statement
;
WHILE
(
E
)
{
code
}
TYPE
ID
=
E
while
ID
code
A
int
a
10
a
code
A
ε
whileloop
statement
;
TYPE
ID
=
E
WHILE
(
E
)
{
code
}
int
a
NUM
while
ID
10
a
code
A
code
A
ifelse
ε
statement
;
IF
(conditions)
{
code
}
ID
=
E
a
E
-
E
E
COMPARE
E
ID
NUM
Id
==
NUM
A
a
1
a
5
code
statement
;
ε
STATEKW
break

Conclusion

We have thus implemented and integrated the lexical and parser codes so that the project gives abstract syntax tree as output(internally within yacc). We now need to get a semantically verified parse tree through the semantic analysis phase and then generate the intermediate code, thus building a full-fledged mini-compiler.

We have implemented most constructs of C language through our lexical and syntax codes and have no shift reduce conflicts in our code (no ambiguity). Also, the syntax phase adds the data type of the identifiers into the symbol table.