Compiler Design Project
Part Two

Syntax Analysis

Project Guided by :

Project done by :

Menezes Elvis Edward (15CO254)
Somnath Krishna Sarkar (15CO247)

Contents.....

Compiler Design

The Compiler Design Process is broken down mainly into two parts namely:

1. Front-End
    a) Lexical Analysis
    b) Syntax Analysis
    c) Semantic Analysis

2. Back-End
    a)Code Optimizer
    b)Code Generator

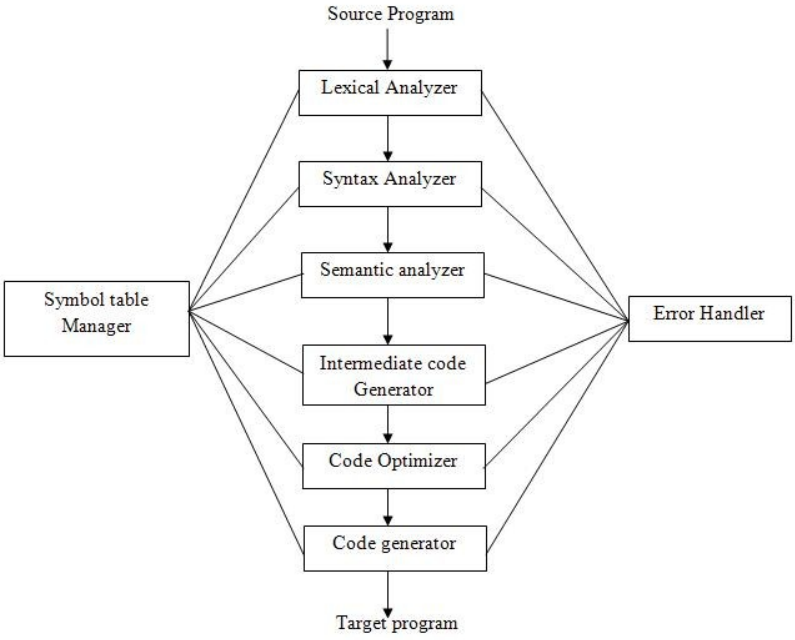


Fig 1.3 Phases of a Compiler

Lexical Analysis:

Lexical Analysis is the first phase of compiler. It is also known as scanner. The input program is converted into a sequence of tokens that can be treated as a unit in the grammar of the programming languages.

Syntax Analysis:

The second phase of the compiler is Syntax Analysis also known as Parsing. The syntactical structure of the given input is checked in this phase, i.e. whether it is the correct syntax. A data structure is built and used in this process called as a Parse tree and also as 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.

Semantic Analysis:

In this phase the parse tree is verified, if it is meaningful or not and produces a verified parse tree.

Intermediate Code Generator:

A form which can be readily executed by machine is generated in this phase of the compiler. One of the many popular intermediate codes is the Three address code.

Code Optimizer:

In this phase the code is transformed into more efficient code so that it consumes less resources and is executed faster without the meaning of the code not altered. Optimization can be categorized into two types: machine dependent and machine independent.

Code Generator:

This is the final stage of the compiler and the main purpose of this stage is write a code that the machine can understand. The output is dependent on the assembler.

This Report mainly focuses on the second phase of the compiler that is the syntax analyzer.

Syntax Analyzer:

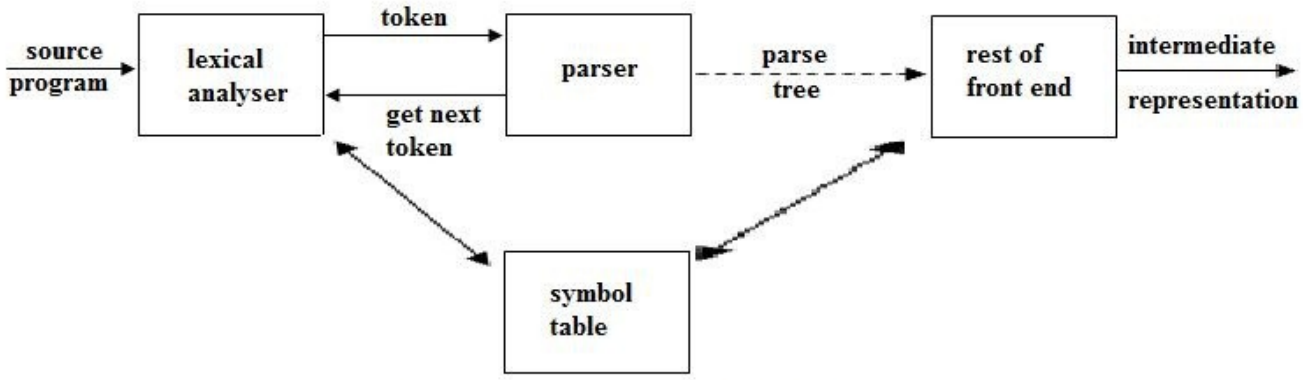


Fig 2.1 Position of parser in compiler model

The **four** components of Context free Grammar are as follows:

1. 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.

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

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

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

Syntax Analyser:

The input to the parser is the stream of tokens which in turn is the output of the lexical analyzer. The parser analyzes the source code (token stream) against the production rules to detect any errors in the code. The output of this phase is a parse tree.

Context Free Grammar is defined and the syntax analyzer uses the productions defined therein to determine the correctness of the syntax.

Syntax analysis or syntactic analysis is the process of analysing a string of symbols, either in natural language, computer languages or data structures, conforming to the rules of a formal grammar. The term parsing comes from Latin pars (orationis), meaning part (of speech)

Syntax Errors:

In computer science, a syntax error is an error in the syntax of a sequence of characters or tokens that is intended to be written in a particular programming language.

For compiled languages, syntax errors are detected at compile-time. A program will not compile until all syntax errors are corrected.

A compiler will flag a syntax error when given source code that does not meet the requirements of the language grammar.

Most frequent syntax errors are:

1. Missing Parenthesis (})
2. Printing the value of variable without declaring it
3. Missing semicolon

Recovery Techniques :

Two classes of recovery:

1. Local Recovery: adjust input at point where error was detected.
2. Global Recovery: adjust input before point where error was detected.

Explanation:

Lexical Analyzer

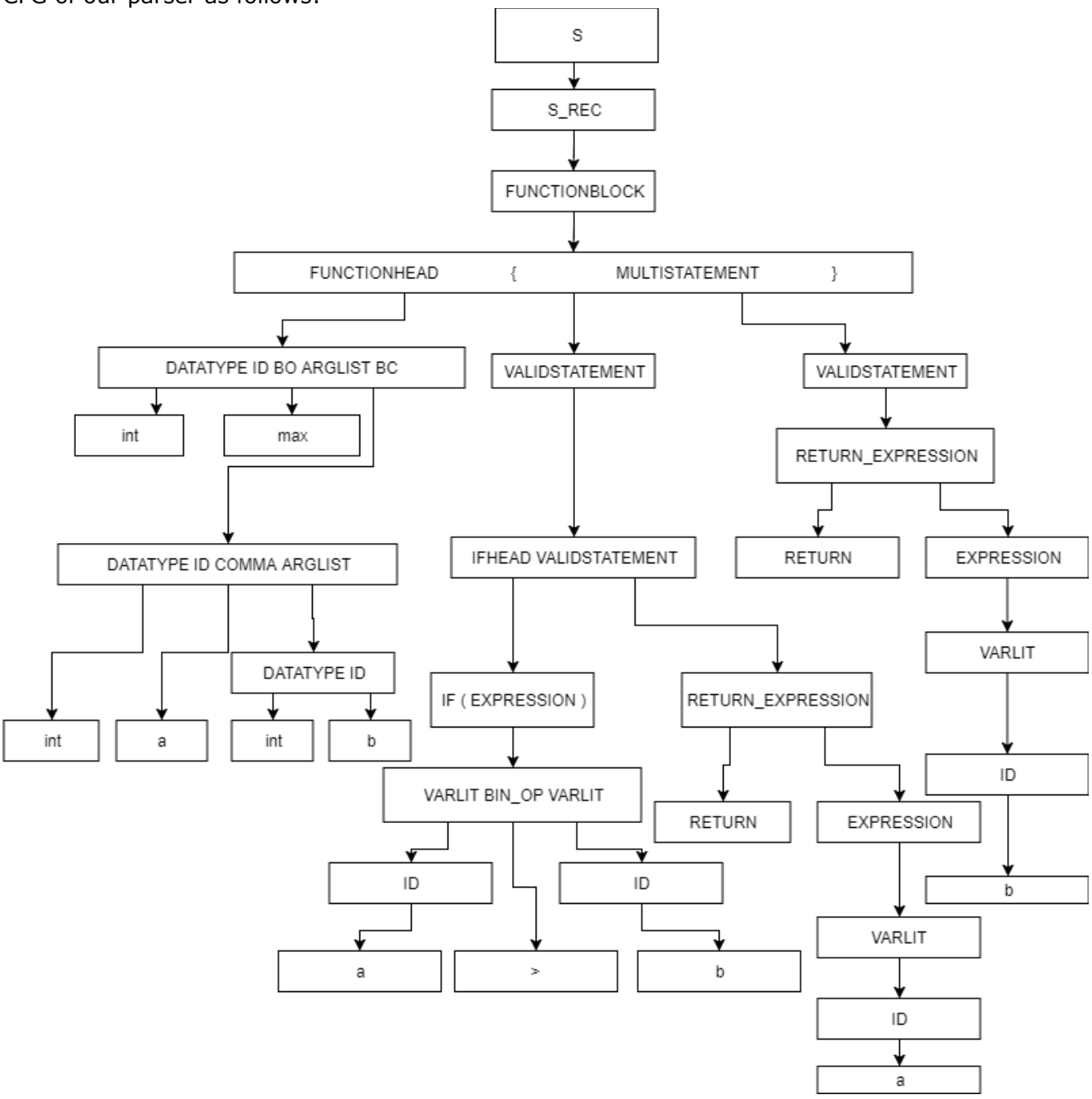
- Handles Single Line comments, Multi-Line comments, Preprocessor Statements, and white space are recognized but not passed to the parser.
- Lexical Errors returned are invalid characters, and dangling string literals.
- Line numbers are calculated here and are used by the hash map and the parser.
- We pass identifier names and datatype keywords as strings to yylval so that the symbol table can be computed on the parser side.
- The constant table is created here, with support for String, Integer and Floating Point constants. The Hash Function used is the summation of ASCII values of characters involved in the constant string (stored in yytext), multiplied by a prime raised to the index of the respective character. The whole hash value is taken modulo a large prime. The Collision handling method used is separate chaining using linked lists.

Syntax Analyzer

- We redefine yylval as a union containing a character pointer, so we can accept strings from the lexical analyzer.
- We make use of a stack to handle scope for the symbol table. An integer a, defined in a struct str, will be stored in the symbol table as str.a, whereas an integer a, defined in a function func will be stored as func.a, so there is no issue of collision for non-overlapping variables.
- The line numbers, datatype and identifier strings are taken from the parser and used to populate the symbol table.
- There is support for all standard C binary and unary operators, with the normal C precedence and associativity rules.
- Syntax errors are returned using line number provided by scanner. Presence of errors interrupts parsing.
- There is support for 3 syntax warnings which are printed to output with corresponding line number, such syntax warnings do not interrupt parsing.
- The first syntax warning is for no return in a non-void returning function. This warning is triggered if there is no return statement found in a function block and the return type included in the function header is not void. If the return type is void no such error is raised.
- The second syntax warning is for a duplicate declaration. If the same identifier name is declared twice in the same scope, this warning is triggered. We implement it by using the lookup function in the corresponding symbol table prior to insertion.
- Undeclared Identifier is the last syntax warning. If a variable is referenced before it is declared in the scope or in any parent scope, this warning is triggered if identifier could not be located in the symbol table.

Parse Tree:

The parse tree for the first function of the valid test case (test case 5) can be derived from the CFG of our parser as follows:



Code for Lexer:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "y.tab.h"
extern YYSTYPE yylval;
const char * keywords[] =
{"int","float","char","long","short","double","void","if","else","while","struct","for","switch","ca
se","break","union","return"};
int kc = 17;
int lineno = 1;
const int MOD = 104729,PRIME = 1009;
struct hash_node{
	char *value,*dtype;
	int line;
	struct hash_node *next;
}*hash_map[104729];
int compute_hash(const char *str){
	int ans = 0;
	int len = strlen(str);
	int i;
	for(i = 0; i<len; i++){
		ans*=PRIME;
		ans%=MOD;
		ans+=str[i];
		ans%=MOD;
	}
	return ans;
}
int hash_lookup(const char *str){
	int hash_value = compute_hash(str);
	struct hash_node *focus = hash_map[hash_value];
	while(focus){
		if(!strcmp(focus->value,str))
		return 1;
		focus = focus->next;
	}
```

```
	return 0;
}
void insert_hash(const char *str){
	if (hash_lookup(str))
		return;
	int hash_value = compute_hash(str);
	if(hash_map[hash_value]==NULL){
		hash_map[hash_value] = (struct hash_node *)malloc(sizeof(struct hash_node));
		hash_map[hash_value]->value = strdup(str);
		if(strstr(str,"\""))
			hash_map[hash_value]->dtype = strdup("String");
		else if(strstr(str,"."))
			hash_map[hash_value]->dtype = strdup("Floating-Point");
		else
			hash_map[hash_value]->dtype = strdup("Integer");
		hash_map[hash_value]->line = lineno;
		hash_map[hash_value]->next = NULL;
	}else{
		struct hash_node *focus = hash_map[hash_value];
		while(focus->next)
			focus = focus->next;
		focus->next = (struct hash_node *)malloc(sizeof(struct hash_node));
		focus->next->value = strdup(str);
		if(strstr(str,"\""))
			focus->next->dtype = strdup("String");
		else if(strstr(str,"."))
			focus->next->dtype = strdup("Floating-Point");
		else
			focus->next->dtype = strdup("Integer");
		focus->next->line = lineno;
		focus->next->next = NULL;
	}
}
void printconstanttable(){
	printf("Constant Table\nValue\tLine\tType\n");
	int i;
	for(i=0; i<MOD; i++){
		struct hash_node *focus = hash_map[i];
		while(focus){
```

```
                        printf("%s\t%d\t%s\n",focus->value,focus->line,focus->dtype);
                        focus = focus->next;
                }
        }
}
void lexinit(){
        int i;
        for(i = 0; i<MOD; i++)
                hash_map[i] = NULL;
}
%}
letter [a-zA-Z]
digit [0-9]
singlecomment "//".*
multicomment "/*"([^*]|\*[^/])*"*/"
comment {singlecomment}|{multicomment}
danglingcomment "/*"
preprocessor "#".*
ws [\t ]+
newline [\n]
datatype ("int"|"float"|"double"|"long"|"short"|"char"|"void")
identifier {letter}({digit}|{letter}|[_])*
if_keyword "if"
else_keyword "else"
while_keyword "while"
switch_keyword "switch"|"case"|"break"
for_keyword "for"
return_keyword "return"
keyword {if_keyword}|{else_keyword}|{while_keyword}|{switch_keyword}|{return_keyword}
intliteral {digit}+
floatliteral (({digit}+("."{digit}*)?)|("."{digit}+))
stringliteral "\""(\\.|[^\\"])*["]
literal {intliteral}|{floatliteral}|{stringliteral}
binary_operator "+="|"-="|"*="|"/="|"^="|"&="|"|="|"=="|"&&"|"||"|"->"|"!="|"<="|">="|[+\-*/\^&|<>]
unary_operator "++"|"--"
semicolon [;]
comma [,]
%%
{singlecomment} ;
```

```
{multicomment} {
	int i = 0;
	for(i = 0; i < yyleng; i++)
		lineno+=(yytext[i]=='\n');
}
{preprocessor} ;
{ws} ;
{newline} lineno++;
{identifier} {
	int match = 0,i = 0;
	for(i = 0; i<kc; i++){
		if(strcmp(keywords[i],yytext)==0){
			match = 1;
			break;
		}
	}
	if(match){
		REJECT;
	}else{
		yylval.str = strdup(yytext);
		return ID;
	}
}
"if" return IF;
"else" return ELSE;
"while" return WHILE;
"return" return RETURN;
"struct"|"union" {yylval.str = strdup(yytext); return STRUCTUNION;}
{unary_operator} {
	return UNARY_OPERATOR;
}
"+=" return PE;
"-=" return ME;
"^=" return XE;
"&=" return AE;
"|=" return OE;
"==" return EQU;
"!=" return NE;
"<=" return LE;
```

```
">=" return GE;
"->" return PNT;
"&&" return AA;
"||" return OO;
[+\-*/\^&|<>] return *yytext;
"=" return EQ;
{datatype} {
       yylval.str = strdup(yytext);
       return DATATYPE;
}
{literal} {
       insert_hash(yytext);
       return CONSTANT;
}
";" return *yytext;
"(" return BO;
")" return BC;
"{" return CBO;
"}" return CBC;
{comma} {
       return COMMA;
}
"\r" ;
. {
       printf("Lexical Error: Line %d\n",lineno);
}
%%
```

Code for Parser:

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "symboltable.h"
extern FILE* yyin;
extern int lineno;
void lexinit();
void printconstanttable();
int was_return;
char datatype[20];
struct symboltable* symstack[100];
char *scstack[100];
int sp = 0;
void init(){
	lexinit();
	scstack[0] = strdup("");
	symstack[0] = createsymboltable();
}
void push(char *scope){
	symstack[++sp] = createsymboltable();
	scstack[sp] = strdup(scope);
}
void pop(){
	apply_scope(symstack[sp],scstack[sp]);
	merge_table(symstack[sp-1],symstack[sp]);
	sp--;
}
void printsymboltable(){
	printf("Symbol Table\n");
	struct node *focus = (symstack[0])->root;
	printf("Identifier Name\tLine\tDatatype\n");
	while(focus){
		printf("%s\t\t%d\t%s\n",focus->id,focus->line,focus->dtype);
		focus = focus->next;
	}
```

```
}
void undeclared_identifier(char * x){
	int i;
	for(i = sp; i>=0; i--){
		if(lookup(symstack[i],x))
			return;
	}
	printf("Warning: Undeclared Identifier %s on Line %d\n",x,lineno);
}
%}
%union{
	char * str;
}
%token DATATYPE ID CONSTANT IF WHILE RETURN STRUCTUNION
%token BO BC CBO CBC
%left COMMA
%right EQ PE XE AE OE ME
%left OO
%left AA
%left '|'
%left '^'
%left '&'
%left EQU NE
%left '<' '>' LE GE
%left '+' '-'
%left '*' '/'
%left UNARY_OPERATOR PNT
%right ELSE
%%
S
	: S_REC {printsymboltable();printconstanttable();}
	;
S_REC
	: /* empty */
	| VALIDSTATEMENT S_REC
	| FUNCTIONBLOCK S_REC
	;
VALIDSTATEMENT
```

```
        : STATEMENT ';'
        | IFHEAD VALIDSTATEMENT ELSEBLOCK
        | IFHEAD CBO MULTISTATEMENT CBC ELSEBLOCK
        | WHILEHEAD VALIDSTATEMENT
        | WHILEHEAD CBO MULTISTATEMENT CBC
        ;
MULTISTATEMENT
        : /* empty */
        | VALIDSTATEMENT MULTISTATEMENT
        ;
STATEMENT
        : /* empty */
        | ASSIGNMENT
        | COMPOUNDDECLARATION
        | COMPOUNDDECLARATION IDCHAIN
        | EXPRESSION
        | RETURN EXPRESSION {was_return = 1;}
        ;
ASSIGNMENT
        : DATATYPE {strcpy(datatype,$<str>1);} DECLARATIONLIST
        | STRUCTUNION ID {undeclared_identifier($<str>2); strcpy(datatype,$<str>1);} IDCHAIN
        ;
DECLARATIONLIST
        : ID {add_identifier(symstack[sp],$<str>1,datatype);}
        | ID COMMA DECLARATIONLIST {add_identifier(symstack[sp],$<str>1,datatype);}
        | ID EQ RVALUE {add_identifier(symstack[sp],$<str>1,datatype);}
        | ID EQ RVALUE COMMA DECLARATIONLIST {add_identifier(symstack[sp],$<str>1,datatype);}
        ;
COMPOUNDDECLARATION
        : STRUCTUNION ID {add_identifier(symstack[sp],$<str>2,$<str>1); push($<str>2); $<str>$ =
strdup($<str>1);} CBO ASSIGNMENTLIST CBC {strcpy(datatype,$<str>1);pop();}
        ;
ASSIGNMENTLIST
        : /* empty */
        | DATATYPE {strcpy(datatype,$<str>1);} IDCHAIN ';' ASSIGNMENTLIST
        ;
IDCHAIN
        : ID {add_identifier(symstack[sp],$<str>1,datatype);}
        | ID COMMA IDCHAIN {add_identifier(symstack[sp],$<str>1,datatype);}
```

```
        ;
IFHEAD
        : IF BO EXPRESSION BC
        ;
ELSEBLOCK
        : /* empty */
        | ELSE VALIDSTATEMENT
        | ELSE CBO MULTISTATEMENT CBC
        ;
WHILEHEAD
        : WHILE BO EXPRESSION BC
        ;
FUNCTIONBLOCK
        : FUNCTIONHEAD {if (strcmp($<str>1,"void")){was_return = 0;} $<str>$ = strdup($<str>1);} CBO
MULTISTATEMENT CBC {if(strcmp($<str>1,"void")&&!was_return)printf("Warning: Return value expected
for non-void function on Line %d\n",lineno);was_return=0;pop();}
        ;
FUNCTIONID
        : DATATYPE ID BO {$<str>$ = strdup($<str>1); strcpy(datatype,$<str>1); strcat(datatype,"()");
add_identifier(symstack[sp],$<str>2,datatype); push($<str>2); }
        ;
FUNCTIONHEAD
        : FUNCTIONID ARGLIST BC {}
        | FUNCTIONID BC
        ;
ARGLIST
        : DATATYPE ID {add_identifier(symstack[sp],$<str>2,$<str>1);}
        | DATATYPE ID COMMA ARGLIST {add_identifier(symstack[sp],$<str>2,$<str>1);}
        ;
RVALUE
        : VARLIT
        | FUNCTIONCALL
        | VARLIT BIN_OP RVALUE
        | BO RVALUE BC
        ;
EXPRESSION
        : VARLIT
        | FUNCTIONCALL
        | VARLIT BIN_OP EXPRESSION
```

```
        | VARLIT COMMA EXPRESSION
        | BO EXPRESSION BC
        | UN_OP EXPRESSION
        | EXPRESSION UN_OP
        ;
FUNCTIONCALL
        : ID BO EXPRESSION BC {undeclared_identifier($<str>1);}
        | ID BO BC {undeclared_identifier($<str>1);}
        ;
BIN_OP
        : EQ
        | PE
        | ME
        | XE
        | AE
        | OE
        | OO
        | AA
        | '+'
        | '-'
        | '&'
        | '*'
        | '/'
        | '^'
        | '<'
        | LE
        | GE
        | '>'
        | EQU
        | NE
        | PNT
        ;
UN_OP
        : UNARY_OPERATOR
        ;
VARLIT
        : ID {undeclared_identifier($<str>1);}
        | CONSTANT
        ;
```

```
%%
int main(){
	init();
	yyin = fopen("../Parser Test Files/test_5.c","r");
	yyparse();
	return 0;
}
int yyerror(const char *s){
	return printf("Syntax Error: Line %d\n",lineno);
}
int yywrap(){
	return 1;
}
```

Code for Symbol Table:

```
#include "symboltable.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
extern int lineno;
struct symboltable *createsymboltable(){
	struct symboltable *ret = (struct symboltable *)malloc(sizeof(struct symboltable));
	ret->root = NULL;
	return ret;
}
void add_identifier(struct symboltable *st, char *id, char *dtype){
	if(lookup(st,id)){
		printf("Warning: Duplicate Definition of %s in this scope on Line %d\n",id,lineno);
		return;
	}
	if((st->root)==NULL){
		st->root = (struct node *)malloc(sizeof(struct node));
		(st->root)->id = strdup(id);
		(st->root)->dtype = strdup(dtype);
		(st->root)->line = lineno;
		(st->root)->next = NULL;
	}else{
		struct node* focus = st->root;
		while(focus->next) focus = focus->next;
			focus->next = (struct node *)malloc(sizeof(struct node));
		(focus->next)->id = strdup(id);
		(focus->next)->dtype = strdup(dtype);
		(focus->next)->line = lineno;
		(focus->next)->next = NULL;
	}
}
void apply_scope(struct symboltable *st, char *scope){
	struct node *focus = st->root;
	int sclen = strlen(scope);
	while(focus!=NULL){
```

```
                int idlen = strlen(focus->id);
                char *tmpstr = (char *)malloc(sclen+idlen+2);
                strcpy(tmpstr,scope);
                strcat(tmpstr,".");
                strcat(tmpstr,focus->id);
                focus->id = tmpstr;
                focus = focus->next;
        }
}
int lookup(struct symboltable *st, char *key){
        struct node *focus = st->root;
        while(focus!=NULL){
                if(!strcmp(key,focus->id))
                return 1;
                focus=(focus->next);
        }
        return 0;
}
void merge_table(struct symboltable* dest, struct symboltable *src){
        struct node* focus = dest->root;
        if(focus==NULL)
                dest->root = src->root;
        else{
                while(focus->next) focus = focus->next;
                focus->next = src->root;
        }
}
```

Code for header file of symbol table:

```
struct node{
	char *id, *dtype;
	int line;
	struct node* next;
};
struct symboltable{
	struct node* root;
};
struct symboltable *createsymboltable();
void add_identifier(struct symboltable* st, char* id, char* dtype);
void apply_scope(struct symboltable* st, char* scope);
int lookup(struct symboltable* st, char *key);
void merge_table(struct symboltable* dest, struct symboltable *src);
```

Test Cases And Screenshots:

Test Case 1

```
//TestCase for Missing SemiColon
#include<stdio.h>
void main(){
        //Valid Test cases
        printf("Hello World");
        5;
        5 + 6;
        int a;
        //Invalid Test cases
        int a
        10
        printf("Missing Semicolon")
}
```

```
C:\Users\guestcheap\Documents\School Work\Year 3\Sem 6\CO351\MiniC\Project2>lex.yy.exe
Warning: Undeclared Identifier printf on Line 6
Warning: Duplicate Definition of a in this scope on Line 15
Syntax Error: Line 15
```

Test Case 2

```
//TestCase for Unmatched paranthesis and unterminated Strings
#include<stdio.h>
void main(){
    //Valid Test cases
    int y=10;
    while(y>0){
       y--;
   }
    printf("Hello World");
   //Invalid Test cases
   int x =10;
   while(x>0){
       x--;

   while({

   }

   char *a = "This is a nice test case

   printf("This is an error");

}
```

```
C:\Users\guestcheap\Documents\School Work\Year 3\Sem 6\CO351\MiniC\Project2>lex.yy.exe
Warning: Undeclared Identifier printf on Line 12
Syntax Error: Line 21
```

Test Case 3

```
//Testcase for function syntax error
#include<stdio.h>

// Valid function declaration
int min(int a,int b){
    if(a<b)
        return a;
    return b;
}

// Invalid function declaration: no parameters
int max{
    return 1;
}

int main(){

    //Valid Test  cases
    int a=7;
    int b=8;
    int c;

    // Valid function call
    c = min(a,b);

    // Invalid function call: No closing parenthesis
    c = min(a,b;

    return 0;
}
```

```
C:\Users\guestcheap\Documents\School Work\Year 3\Sem 6\CO351\MiniC\Project2>lex.yy.exe
Syntax Error: Line 12
```

Test Case 4

```
//Testcase for loop statements and if else statements

#include<stdio.h>

int main(){


    int a=10;
    int b=15;

    // Valid if else statements
    if(a>b){
        printf("hi 1");
    }
    else{
        printf(" hi 2")
    }

    // Valid while loop
```

```
    while(a>0)
        a--;

    // Valid while loop
    while(b>0)
    {
        printf("Hi 3")
        b--;
    }

    // Invalid if statement: No condition
    if(){
        printf("Error");
    }

    a = 5;

    // Invalid else statement: No matching if
    else{

    }

    return 0;
}
```

```
C:\Users\guestcheap\Documents\School Work\Year 3\Sem 6\CO351\MiniC\Project2>lex.yy.exe
Warning: Duplicate Definition of b in this scope on Line 10
Warning: Undeclared Identifier printf on Line 14
Warning: Undeclared Identifier printf on Line 17
Syntax Error: Line 18
```

Test Case 5

```
// Valid test case
#include <stdio.h>
int max(int a, int b){
    if(a>b)
        return a;
    return b;
}
int main(){
    int a = 10,b;
    b = a+5;
    int c = max(a,b);
    printf("%d\n",c);
    while(a<=b){
        int d = 5;
        while(d>0){
            d--;
            a+=2;
        }
    }
    c = max(a,b);
    printf("%d",c);
    return 0;
}
```

```
C:\Users\guestcheap\Documents\School Work\Year 3\Sem 6\CO351\MiniC\Project2>lex.yy.exe
Warning: Undeclared Identifier printf on Line 14
Warning: Undeclared Identifier printf on Line 23
Symbol Table
Identifier Name Line    Datatype
max             4       int()
max.b           4       int
max.a           4       int
main            10      int()
main.b          11      int
main.a          11      int
main.c          13      int
main.d          16      int
Constant Table
Value   Line    Type
0       17      Integer
2       19      Integer
5       12      Integer
10      11      Integer
"%d"    23      String
"%d\n"  14      String
```

Conclusion:

Thus we have finished implementation of both the scanner and the parser. We build upon the lexical analyzer created in the first project in several ways. Firstly, by adding support for standard C operator precedence and associativity. Secondly we have support for undeclared and duplicate declarations warnings using the symbol table with scope support. Moreover, we can return warnings for absence of return statements in non-void functions, as well as general purpose syntax errors with line numbers. The symbol table and constant table are successfully populated by the parser and scanner respectively. The code that passes through the parser with no syntax errors should be ready to pass through the next stage of the compiler process: The Semantic Analyzer.