

Project Report

Syntax Analysis for C Compiler

Submitted by

Agnitha Mohanram-15CO201

Vasudha Boddukuri-15CO211

Under Guidance of

Sushmita

Umapriya



National Institute of Technology-Karnataka

Contents

- 1 Acknowledgement
- 2 Introduction
 - 2.1 Overview
 - 2.2 Motivation
 - 2.3 Objectives
 - 2.4 Concepts and tools
- 3 Compiler Design
- 4 Syntax Phase
- 5 Implementation
- 6 Source Code
- 7 Output
- 8 Parse Tree
- 9 Conclusion
- 10 References

Acknowledgement

We would like to express our gratitude to Ms.Santhi Thilagam, Ms. Sushmita and Ms.Umapriya for their constant support, guidance and help. A special thanks to NITK for providing us with the resources and facilities required to complete our project successfully. This learning experience was extremely helpful and valuable. Last but not the least, we're very thankful to our parents for their unwavering support and encouragement.

Introduction

Overview

The report elaborates on the syntax analysis phase of building a C-compiler. It describes the procedure used to build the syntax analyzer and contains the output obtained from several test cases. It also enlists the various tools and concepts used in the implementation.

Motivation

A compiler is a software that aims to transform code submitted to it which is written in one programming language (the source language) into another programming language (the target language).

Compilers are extensively used in the programming world and it's essential for a computer science engineer to understand the working of a compiler. By doing so, he/she gains thorough knowledge of the system software and can manipulate it with ease if need arises.

The project aims to build a primitive compiler from scratch through the various phases of compiler design.

This report is on the lexical analysis phase where the input program is converted into sequence of tokens which are stored in a symbol table.

Objective

The objectives of the project are as follows:

- to study the format of a typical yacc script
- to enter the grammar in the yacc format, integrate with the scanner generated in the previous stage and generate the parser
- to test the generated parser on sample programs

Concepts and Tools

Compiler

Compilers are system softwares that convert code written in one language (usually high level language) into an equivalent machine understandable language to generate an executable.

Syntax Analyzer

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 syntactical errors in the code.

Hashing

Hashing is the process of converting given string into a key value by using some mathematical functions. These key values are used to optimize time complexity for searching, insertion and deletion from memory.

Hash Table

A hash table is a data structure which is used to map keys generated by hashing to the values they were generated from.

Flex

Lex is a computer program that generates lexical analyzers ("scanners" or "lexers"). Lex reads an input stream specifying the lexical analyzer and outputs source code implementing the lexer in the C programming language.

Git

Git is a version control system(VCS) which helps keep track of changes made in projects. It is used to maintain records of changes made to any set of files.

Yacc

YACC stands for "Yet Another Compiler Compiler". This is because this kind of analysis of text files is normally associated with writing compilers. Yacc is officially known as a "parser". Its job is to analyze the structure of the input stream, and operate of the "big picture". In the course of its normal work, the parser also verifies that the input is syntactically sound.

Parse tree

Parse tree is a hierarchical structure which represents the derivation of the grammar to yield input strings. Root node of parse tree has the start symbol of the given grammar from where the derivation proceeds. Leaves of parse tree represent terminals. Each interior node represents productions of grammar.

Production rules

One of the important component of Context Free Grammar (CFG) is set of Productions or Production Rules. 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 non-terminals, called the right side of the production.

Compiler Design

Compiler

A compiler is computer software that transforms computer code written in one programming language(the source language) into another programming language (the target language) without changing the meaning of the program.

The compiler is expected to make the target code efficient and optimized in terms of time and space complexities.

Compiler design deals with the translation and optimization processes. It covers basic translation mechanism, error detection and errors recovery.

Compiler design has two phases:

FRONT END

- a) Lexical Analysis
- b) Syntax Analysis
- c) Semantic Analysis
- d) Intermediate Code Generation

BACK END

- e) Code Optimization
- f) Code Generation

The front end builds an intermediate representation(IR) of the source code. It also manages the symbol table and mapping of each token in the program to its associated information.

The back end on the other hand deals with the CPU architecture specific optimizations and code generation.

Lexical Analysis

Lexical analysis is the first phase of compiler design. It takes the source code as input and translates it into a sequence of tokens. It removes the unnecessary elements of the source code such as white spaces and comments and generates appropriate error messages if any invalid token is encountered.

Syntax Analysis

A syntax analyzer checks the syntactical correctness of the source program. It takes the token of streams generated by the lexical phase as input and analyzes the source code against the production rules to detect errors. The output is in the form of a parse tree.

Semantic Analysis

Semantics of the language add meaning to its tokens and syntax structure. The semantic analyzer analysis the parse tree and determines if it derives any meaning. The output of this phase is a more verified parse tree.

Intermediate Code Generation

Intermediate code is generated from the verified parse tree. It is then converted to machine language through the last two phases. All the phases till the intermediate code generation are platform independent, whereas the next two phases are not. In order to build a compiler for a platform, one need not start from scratch. He/she could take the intermediate code from an already existing compiler and build the last two phases.

Code optimization

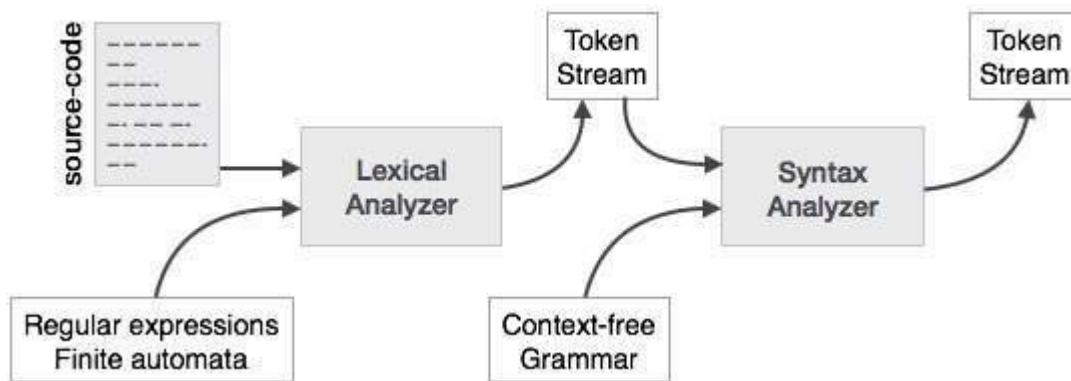
The code optimizer transforms the code into an optimized version which is designed to consume fewer resources and work faster. There are two types of optimizations, machine dependent and machine independent.

Target Code Generator

This can be looked at as the final stage of compilation. Its function is to write a code that the machine can understand. The output is assembler dependent.

Syntax Analysis

The second phase of a compiler is the Syntax 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. The output of this phase is a parse tree.



This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase. Parsers are expected to parse the whole code even if some errors exist in the program.

Limitations of Syntax Analyzers:

Syntax analyzers receive their inputs, in the form of tokens, from lexical analyzers. Lexical analyzers are responsible for the validity of a token supplied by the syntax analyzer. Syntax analyzers have the following drawbacks -

- it cannot determine if a token is valid,
- it cannot determine if a token is declared before it is being used,
- it cannot determine if a token is initialized before it is being used,
- it cannot determine if an operation performed on a token type is valid or not.

These tasks are accomplished by the semantic analyzer

Implementation

Scanner:

This part of the program was implemented with the help of the lex tool. Regular expressions were written in order to extract tokens from the c program. The tokens thus extracted were inserted into the symbol and constant tables accordingly.

Symbol Table and Constant Table:

The symbol and constant tables were made efficient through the use of hash functions. The implementation of the hashing concepts is elaborated below.

Hash Table:

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index where an element is to be inserted or is to be located from.

Hashing:

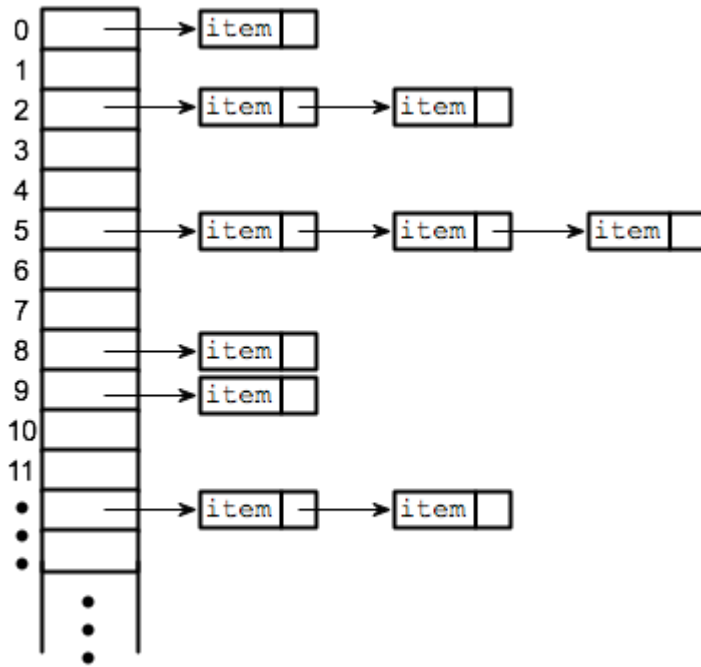
Hashing is a technique to convert a range of key values into a range of indexes of an array. The sum of the ASCII values of each of the characters in the matched string was obtained and the modulo operator(mod table size) was used on the value thus obtained to get an index in the table.

To resolve clashes, the chaining technique was used .

Chaining Technique:

Separate chaining is one of the most commonly used collision resolution techniques. It was implemented using linked lists. Each element of the hash table is a linked list. To store an element in the hash table, the element was inserted into a specific linked list. In

case of any collision (i.e. two different elements have same hash value) both the elements were stored in the same linked list.



Parser:

This part of the program was implemented with the help of the yacc tool. Production rules were written and the syntactical errors were checked for. If the given program satisfied the grammar, the parsing was successful. Otherwise, appropriate error messages were displayed.

Code Explanation:

Code segments for important sections are discussed below:

1) Handling nested comments in Flex

```
%X SC_COMMENT

%%
    int comment_nesting = 0; /* Line 4 */

    "/"* "          { BEGIN(SC_COMMENT); }
<SC_COMMENT>{
    "/"* "          { ++comment_nesting; }
    "*"+"/"         { if (comment_nesting)
--comment_nesting;
                      else BEGIN(INITIAL); }
    "*" +           ; /* Line 11 */
    [^/*\n]+        ; /* Line 12 */
    [/]             ; /* Line 13 */
    \n              ; /* Line 14 */
}
```

Line 4: Indented lines before the first rule are inserted at the top of the `yyllex` function where they can be used to declare and initialize local variables. We use this to initialize the comment nesting depth to 0 on every call to `yyllex`. The invariant which must be maintained is that `comment_nesting` is always 0 in the `INITIAL` state.

Lines 11-13: A simpler solution would have been the single pattern `.\n.`, but that would result in every comment character being treated as a separate subtoken. Even though the corresponding action does nothing, this would have caused the scan loop to be broken and the action switch statement to be executed for every character. So it is usually better to try to match several characters at once.

We need to be careful about `/` and `*` characters, though; we can only ignore those asterisks which we are certain are not part of the `*/` which terminates the (possibly nested) comment. Hence lines 11 and 12. (Line 12 won't match a sequence of asterisks which is followed by a `/` because those will already have been matched by the pattern above, at line 9.) And we need to ignore `/` if it is not followed by a `*`. Hence line 13.

Line 14: However, it can also be sub-optimal to match too large a token.

First, flex is not optimized for large tokens, and comments can be very large. If flex

needs to refill its buffer in the middle of a token, it will retain the open token in the new buffer, and then rescan from the beginning of the token.

Second, flex scanners can automatically track the current line number, and they do so relatively efficiently. The scanner checks for newlines only in tokens matched by patterns which could possibly match a newline. But the entire match needs to be scanned.

We reduce the impact of both of these issues by matching newline characters inside comments as individual tokens. (Line 14, also see line 12) This limits the `yylineno` scan to a single character, and it also limits the expected length of internal comment tokens. The comment itself might be very large, but each line is likely to be limited to a reasonable length, thus avoiding the potentially quadratic rescan on buffer refill.

2) Dangling Else

```
%nonassoc LOWER_THAN_ELSE
%nonassoc ELSE

If_else_stmt
: IF '(' exp ')' stmt %prec LOWER_THAN_ELSE
| IF '(' exp ')' stmt ELSE stmt
;
```

ELSE and LOWER_THAN_ELSE (a pseudo-token) are made non associative by the first two statements.

This gives ELSE more precedence over LOWER_THAN_ELSE simply because LOWER_THAN_ELSE is declared first.

Then in the conflicting rule a precedence is assigned to either the shift or reduce action. Here, higher precedence is given to shifting.

Source Code

Source Code:

LEX CODE:

```
%{
#include "y.tab.h"
#include<string.h>
#include<stdio.h>
int lineno=1;
int comment_nesting = 0;
int eleCount=50;

struct t
{
    char key[20];
    int lno;
}type;

    struct node {

        char name[20];
        char type[20];
        int key;

        struct node *next;

    };

    struct hash {

        struct node *head;
```

```

        int count;

};

//struct hash *hashTable = (struct hash *) calloc(50,
sizeof(struct hash));

struct hash hashTable[50];
struct hash chashTable[50];

struct node * createNode(int key, char *name, char
*type) {

    struct node *newnode;

    newnode = (struct node *) malloc(sizeof(struct
node));

    newnode->key=key;
    strcpy(newnode->name, name);
    strcpy(newnode->type, type);

    newnode->next = NULL;

    return newnode;

}

void insertToHash(int key, char *name, char *type) {

    int hashIndex = key % eleCount;

    struct node *newnode = createNode(key, name, type);

```

```

if (!hashTable[hashIndex].head) {

    hashTable[hashIndex].head = newnode;

    hashTable[hashIndex].count = 1;

    return;

}

newnode->next = (hashTable[hashIndex].head);

hashTable[hashIndex].head = newnode;

hashTable[hashIndex].count++;

return;

}

```

```

int searchInHash(int key, char *name) {

    int hashIndex = key % eleCount, flag = 0;

    struct node *myNode;

    myNode = hashTable[hashIndex].head;

    if (!myNode) {

```



```

    return flag;

}

while (myNode != NULL) {

    if (strcmp(myNode->name,name)==0) {

        flag = 1;

        break;

    }

    myNode = myNode->next;

}

return flag;

}

```

```

void display() {

    struct node *myNode;

    int i;
    printf("\n\n
SYMBOL TABLE");

    printf("\n\n\n");

printf("\t|-----
-----|\n");

```

```

        printf("\t|
|\n");

        printf("\t|          Name
|   Type          |\n");

printf("\t|-----
-----|\n");

        printf("\t|
|          |\n");

        for (i = 0; i < eleCount; i++) {

            if (hashTable[i].count == 0)

                continue;

            myNode = hashTable[i].head;

            if (!myNode)

                continue;

            while (myNode != NULL) {

                printf("\t|\t%-30s", myNode->name);

                printf("\t|\t%-15s\n", myNode->type);

printf("\t|-----
-----|\n");

```

```

        myNode = myNode->next;

    }

}

return;

}

```

```

int hash_func(char val[20])
{
    int sum=0;
    for(int i=0; i<strlen(val);i++)
    {
        sum+=val[i];
    }
    sum%=50;

    return sum;
}

```

```

void insertToHash(int key, char *name, char *type) {

    int hashIndex = key % eleCount;

    struct node *newnode = createNode(key, name, type);

    if (!chashTable[hashIndex].head) {

        chashTable[hashIndex].head = newnode;

        chashTable[hashIndex].count = 1;

        return;
    }
}

```

```
}
```

```
newnode->next = (chashTable[hashIndex].head);
```

```
chashTable[hashIndex].head = newnode;
```

```
chashTable[hashIndex].count++;
```

```
return;
```

```
}
```

```
int searchIncHash(int key, char *name) {
```

```
    int hashIndex = key % eleCount, flag = 0;
```

```
    struct node *myNode;
```

```
    myNode = chashTable[hashIndex].head;
```

```
    if (!myNode) {
```

```
        return flag;
```

```
    }
```

```
    while (myNode != NULL) {
```

```

        if (strcmp(myNode->name,name)==0) {

            flag = 1;

            break;

        }

        myNode = myNode->next;

    }

    return flag;

}

```

```

void cdisplay() {

    struct node *myNode;

    int i;

    printf("\n\n
CONSTANT TABLE");

    printf("\n\n\n");

    printf("\t|-----|
-----|\n");

    printf("\t|
|\n");

    printf("\t|          Constant
|      Type          |\n");

```

```

printf("\t|-----
-----|\n");

        printf("\t|
|                                     |\n");

        for (i = 0; i < eleCount; i++) {

            if (chashTable[i].count == 0)

                continue;

            myNode = chashTable[i].head;

            if (!myNode)

                continue;

            while (myNode != NULL) {

                printf("\t|\t%-30s", myNode->name);

                printf("\t|\t%-15s\n", myNode->type);

printf("\t|-----
-----|\n");

                myNode = myNode->next;

            }

        }

```

```

        return;

    }

    int chash_func(char val[20])
    {
        int sum=0;
        for(int i=0; i<strlen(val);i++)
        {
            sum+=val[i];
        }
        sum%=50;

        return sum;
    }

```

```

%}
alpha [a-zA-Z_]
digit [0-9]
ID {alpha}({alpha}|{digit})*
WHITESPACE " " | " " | "\n"

```

```

%x SC_COMMENT

```

```

%%

```

```

#include[ ]?<{alpha}*.h> {}

```

```

"/*"          { comment_nesting++; BEGIN(SC_COMMENT); }
<SC_COMMENT>{
    "/"        { comment_nesting++; }

    "*"+"/"    { comment_nesting=0;
                  BEGIN(INITIAL); }

```

```

    "*" +
    [/]
    \n
}

\\\/[^\n]*[\n]? { lineno++; }

```

```

"break"          { return(BREAK); }
"char"           {strcpy(type.key,yytext);
                  type.lno=lineno;
                  return(CHAR); }
"double"         { strcpy(type.key,yytext);
                  type.lno=lineno;
                  return(DOUBLE); }
"else"           { return(ELSE); }
"float"          {strcpy(type.key,yytext);
                  type.lno=lineno;
                  return(FLOAT); }
"for"            { return(FOR); }
"if"             { return(IF); }
"int"            { strcpy(type.key,yytext);
                  type.lno=lineno;
                  return(INT); }
"long"           {  strcpy(type.key,yytext);
                  type.lno=lineno;
                  return(LONG); }
"return"         {  strcpy(type.key,yytext);
                  type.lno=lineno;
                  return(RETURN); }
"short"          {  strcpy(type.key,yytext);
                  type.lno=lineno;
                  return(SHORT); }
"signed"         {  strcpy(type.key,yytext);
                  type.lno=lineno;
                  return(SIGNED); }
"unsigned"       {  strcpy(type.key,yytext);
                  type.lno=lineno;
                  return(UNSIGNED); }

```



```

"void"          { strcpy(type.key,yytext);
                  type.lno=lineno;
                  return(VOID); }

{ID}           {
                  int key=hash_func(yytext);
                  if (searchInHash(key,yytext)==0)
                  {
                  if (type.lno==lineno)
                      //printf("%s", type.key);
                  insertToHash(key,yytext,type.key); }
                  return ID;}

{digit}+       {int key=chash_func(yytext);
                  if (searchInHash(key,yytext)==0)
                  insertToHash(key,yytext,"constant");
                  return CONSTANT;}

{digit}*\\. {digit}+ {int key=chash_func(yytext);
                  if (searchInHash(key,yytext)==0)
                  insertToHash(key,yytext,"constant");
                  return CONSTANT;}

0[xX][0-9a-fA-F]+ {int key=chash_func(yytext);
                  if (searchInHash(key,yytext)==0)
                  insertToHash(key,yytext,"constant");
                  return CONSTANT;}

'0'[0-7]+      {int key=chash_func(yytext);
                  if (searchInHash(key,yytext)==0)
                  insertToHash(key,yytext,"constant");
                  return CONSTANT;}

[+-]?[digit]+[.]?[digit]*([eE][+-]?[digit]+)? {int
key=chash_func(yytext);
                  if (searchInHash(key,yytext)==0)
                  insertToHash(key,yytext,"constant");
                  return CONSTANT;}

```

```

\" [^\n]*\"    {
    int key=chash_func(yytext);
    if (searchIncHash(key,yytext)==0)
        insertToHash(key,yytext,"string");
        return (STRING);
    }

"+=" | "-=" | "*=" | "/=" | "%=" { return (ASSIGN_OP); }
"==" | "!=" { return (EQU_OP); }
"<=" | ">=" | ">" | "<" { return (REL_OP); }
"+" | "-" { return (ADD_OP); }
"*" | "/" | "%" { return (MUL_OP); }
"++" | "--" { return (INCDEC_OP); }
"&&" {return (LAND);}
"||" {return (LOR);}
"=" { return ('='); }
"{" { return ('{'); }
"}" { return ('}'); }
"(" { return ('('); }
")" { return (')'); }
"[" { return ('['); }
"]" { return (']'); }
";" {strcpy(type.key," ");
    type.lno=0;
    return(';'); }
", " { return(','); }
"." { return('.'); }
"? " { return('?'); }
": " { return(':'); }

{WHITESPACE} { if (yytext[0]=='\n') { lineno++; } };

. {printf("LEXICAL ERROR - %s : invalid character at line
[%d]\n",yytext,lineno);}

```

%%

```
int yywrap()  
{  
    return 1;  
}
```

Yacc Code:

```
%token BREAK CHAR DOUBLE ELSE FLOAT FOR IF INT LONG  
RETURN SHORT SIGNED UNSIGNED VOID  
%token ID CONSTANT STRING ASSIGN_OP REL_OP ADD_OP  
MUL_OP INCDEC_OP EQU_OP LAND LOR
```

```
%{  
#include<stdio.h>  
#include<stdlib.h>  
#include<ctype.h>  
  
int lineno;  
extern int comment_nesting;
```

```
int flag = 0;  
extern void display();  
extern void cdisplay();  
%}
```

```
%nonassoc LOWER_THAN_ELSE  
%nonassoc ELSE
```

```
%right ASSIGN_OP '='  
%left LOR  
%left LAND  
%left EQU_OP  
%left REL_OP  
%left ADD_OP  
%left MUL_OP  
%left INCDEC_OP  
%start start
```

```
%%
```

```
start
```

```

        :ext
        |start ext
        ;

ext
    :func_def
    |decn
    ;

decn
    : decn_spec init_decr_list ';'
    ;

decn_spec
    : type_spec
    | type_spec decn_spec
    ;

type_spec
    : VOID
    | CHAR
    | SHORT
    | INT
    | LONG
    | FLOAT
    | DOUBLE
    | SIGNED
    | UNSIGNED
    ;

init_decr_list
    : init_decr
    | init_decr_list ',' init_decr
    ;

init_decr

```

```
    : decr
    | decr '=' exp
    | decr '=' STRING
    ;
```

```
decr
    : ID
    | '(' decr ')'
    | func_decn
    ;
```

```
func_decn
    : ID '(' decn_spec_list ')'
    | ID '(' ' ' ')'
    ;
```

```
func_def
    : decn_spec ID '(' ' ' ')' cmpnd_stmt
    | decn_spec ID '(' def_spec_list ')' cmpnd_stmt
    ;
```

```
decn_spec_list
    : decn_spec_list ',' decn_spec
    | decn_spec
    ;
```

```
def_spec_list
    : var_decn
    | def_spec_list ',' var_decn
    ;
```

```
var_decn
    : decn_spec ID
    ;
```

```
cmpnd_stmt
    : '{' stmt_list '}'
    | '{' '}'
    ;
```

;

```
stmt_list
: stmt_list stmt
| stmt
;
```

```
stmt
: cmpnd_stmt
| exp_stmt
| for_loop_stmt
| if_else_stmt
| jump_stmt
| decn
| error ';'
;
```

```
exp_stmt
: ';'
| ID ASSIGN_OP exp ';'
| ID '=' exp ';'
| exp INCDEC_OP ';'
| INCDEC_OP exp ';'
| func_call
;
```

```
exp
: exp REL_OP exp
| exp EQU_OP exp
| exp ADD_OP exp
| exp MUL_OP exp
| exp LAND exp
| exp LOR exp
```

```
| exp INCDEC_OP
| INCDEC_OP exp
| '(' exp ')'
| ID
| CONSTANT
| func_call
;
```

```
func_call
: ID '(' ')'
| ID '(' exp_list ')'
;
```

```
exp_list
: exp_list ',' exp
| exp
;
```

```
if_else_stmt
: IF '(' exp ')' stmt %prec LOWER_THAN_ELSE
| IF '(' exp ')' stmt ELSE stmt
;
```

```
for_loop_stmt
: FOR '(' exp_stmt exp_stmt ')' stmt
| FOR '(' exp_stmt exp ';' exp ')' stmt
;
```

```
jump_stmt
: BREAK ';'
| RETURN ';'
| RETURN exp
;
```


%%

```
int yyerror()  
{  
    flag = 1;  
    printf("PARSING ERROR at Line Number -  
%d\n",lineno);  
    return (1);  
}  
  
main()  
{  
  
    yyparse();  
    if(comment_nesting!=0)  
        printf("LEXICAL ERROR : Unterminated  
Comment\n");  
  
    if(!flag)  
    {  
        printf("\nParsing successful!\n");  
    }  
  
    display();  
    cdisplay();  
}
```

Output

Test Case-1

Input:

```
/*program to calculate factorial of  
a positive integer */
```

```
#include <stdio.h>
```

```
void main()  
{  
    int num, i, value = 1;  
  
    num=10;  
  
    for(i = 1; i <= num; ++i)  
    {  
        value *= i;  
    }  
  
    return 0;  
}
```

```
agnitha@agnitha-Inspiron-3442: ~/Desktop/Project/Parser
|-----|
agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser$ ./a.out<Test/
basic-if-else.png          error-nested-for.c          missing-semi.png
dangling-else.png          error-nested-for.png        No_error/
dangling-if-else.c         missing-semicolon-quotes-brackets.c
agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser$ ./a.out<Test/No_error/basic-for.c
program to calculate factorial of      a positive integer
Parsing successful!

SYMBOL TABLE

|-----|
| Name | | Type |
|-----|
| i     | | int  |
|-----|
| main  | | void |
|-----|
| num   | | int  |
|-----|
| value | | int  |
|-----|

CONSTANT TABLE

|-----|
| Constant | | Type |
|-----|
| 10        | | constant |
|-----|
| 0         | | constant |
|-----|
| 1         | | constant |
|-----|
agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser$
```

Test Case-2

Input:

```
#include<stdio.h>
```

```
void main()
```

```
{
```

```
    float c;
```

```
    int a=5;
```

```
    int z,b;
```

```
    if(a<10)
```

```
    {
```

```
        a=a+1;
```

```
    }
```

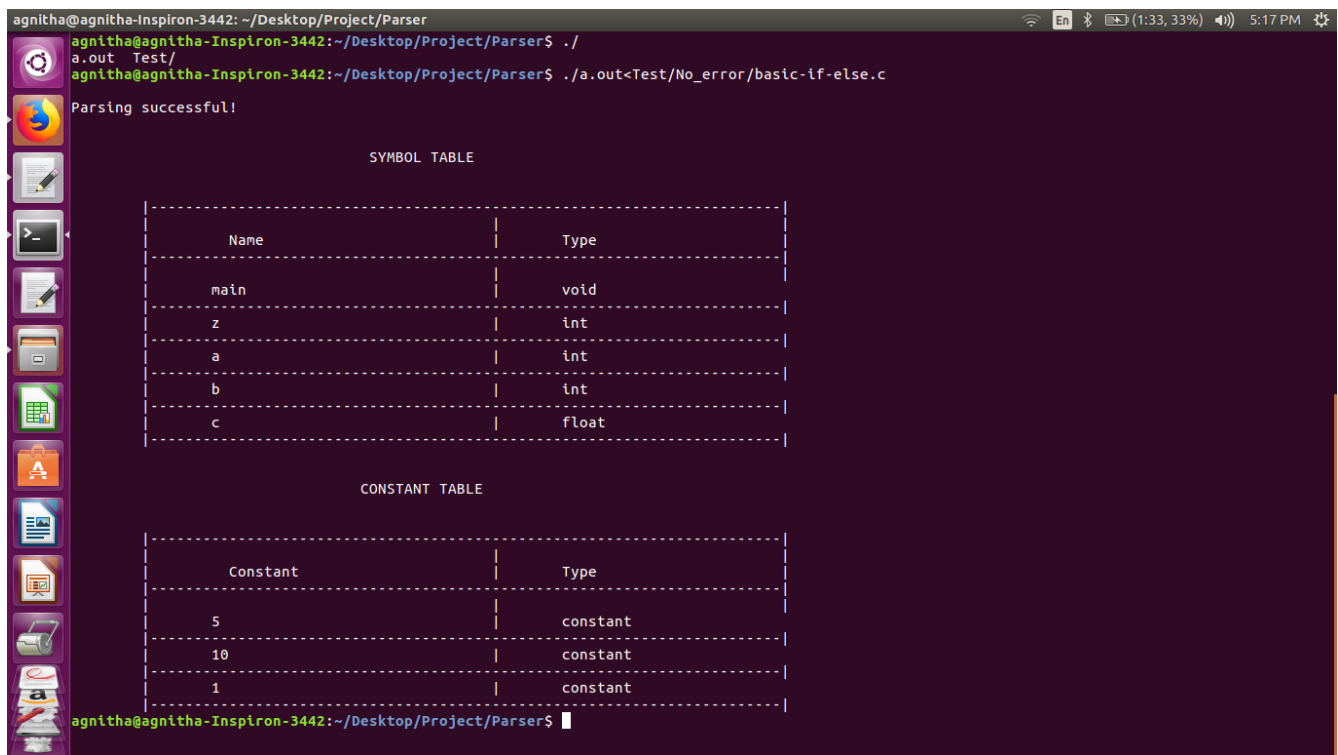
```
    else
```

```
    {
```

```
        a=a-1;
```

```
    }
```

```
}
```



The terminal window shows the execution of a parser. The command `./a.out<Test/No_error/basic-if-else.c` was run, resulting in the message "Parsing successful!". Below this, two tables are displayed: the SYMBOL TABLE and the CONSTANT TABLE.

SYMBOL TABLE

Name	Type
main	void
z	int
a	int
b	int
c	float

CONSTANT TABLE

Constant	Type
5	constant
10	constant
1	constant

The terminal prompt is `agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser$`.

Test Case-3

Input:

```
int count=0;
```

```
int i, j;
```

```
void find_prime(int n)
```

```
{
```

```
    int flag;
```

```
        for(i=2 ; i<=n ; i++)
```

```
        {
```

```
            count = 0;
```

```
            for(j=1 ; j<=i ; j++)
```

```
            {
```

```
                if(i%j==0)
```

```
                    count++;
```

```
            }
```

```
            if(count == 2)
```

```
                flag = 1;
```

```
            else
```

```
                flag = 0;
```

```
        }
```

```
    }
```

```
void main()
```

```
{
```

```
    find_prime(10);
```

```
}
```

```
agnitha@agnitha-Inspiron-3442: ~/Desktop/Project/Parser
agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser$ ./a.out<Test/No_error/nested-for.c
Parsing successful!

SYMBOL TABLE

|-----|
| Name | | Type |
|-----|
| find_prime | | void |
|-----|
| count | | int |
|-----|
| i | | int |
|-----|
| j | | int |
|-----|
| flag | | int |
|-----|
| n | | int |
|-----|
| main | | void |
|-----|

CONSTANT TABLE

|-----|
| Constant | | Type |
|-----|
| 2 | | constant |
|-----|
| 10 | | constant |
|-----|
| 0 | | constant |
|-----|
| 1 | | constant |
|-----|

agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser$
```

Test Case-4

Input:

```
void main()
{
    int a=5,b;
    if(a>10)
    {
        b=1;
    }
    else if(a<10)
    {
        b=-1;
    }
    else
    {
        b=0;
    }
}

else
{
    b=100;
}

}
```

agnitha@agnitha-Inspiron-3442: ~/Desktop/Project/Parser

En (1:50, 21%) 4:59 PM

```
agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser$ ./a.out<Test/dangling-if-else.c
PARSING ERROR at Line Number - 18
```

SYMBOL TABLE

Name	Type
main	void
a	int
b	int

CONSTANT TABLE

Constant	Type
5	constant
-1	constant
100	constant
10	constant
0	constant
1	constant

agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser\$

Test Case-5

Input:

```
int main()
{
    int max=10;
    int c;
    int i, j;
    for( i=0;i<max;)
    {
    for( j=0;j<20;j++)
    {
        c=2;
    }
    }
}
```

```
agnitha@agnitha-Inspiron-3442: ~/Desktop/Project/Parser
agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser$ ./a.out<Test/error-nested-for.c
PARSING ERROR at Line Number - 6
PARSING ERROR at Line Number - 8
PARSING ERROR at Line Number - 8
```

SYMBOL TABLE

Name	Type
i	int
j	int
main	int
max	int
c	int

CONSTANT TABLE

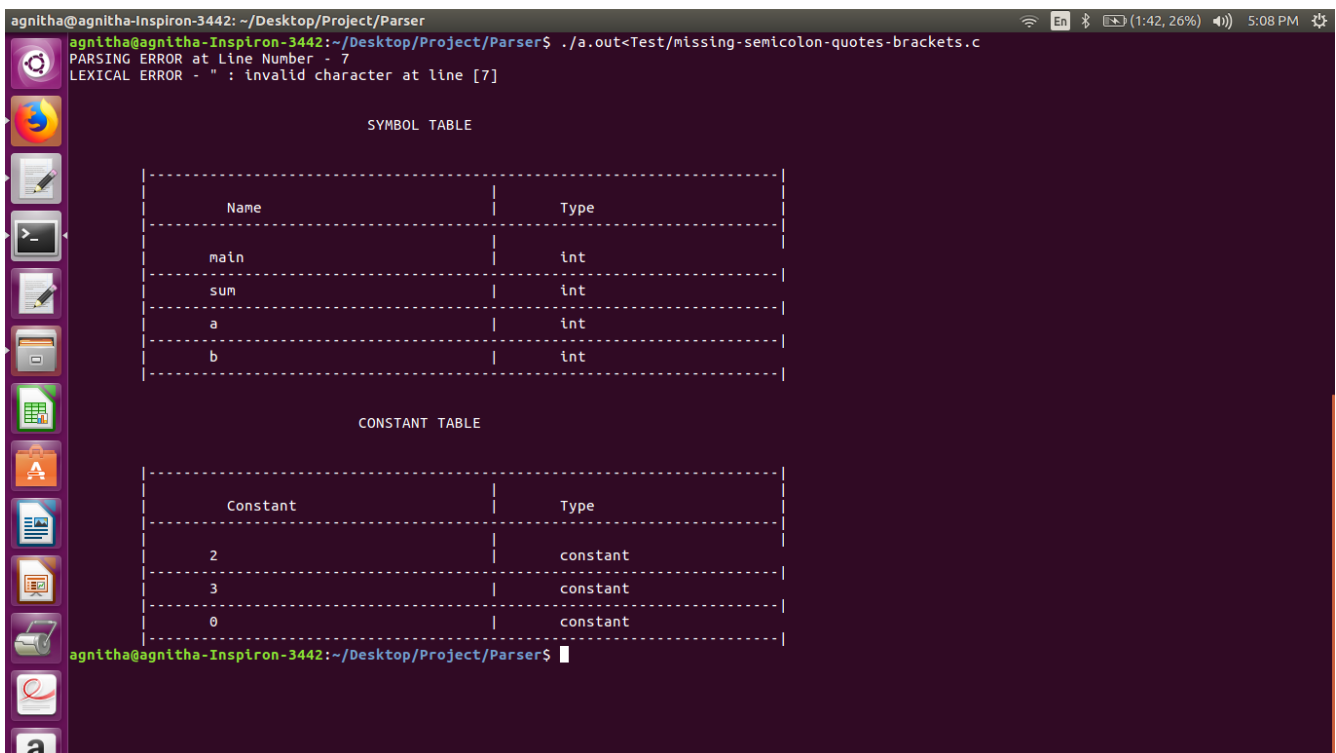
Constant	Type
2	constant
10	constant
20	constant
0	constant

```
agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser$
```

Testcase-6

Input:

```
#include<stdio.h>
int main()
{
    int a=2,b=3;
    int sum=0;
    sum=a+b
```



The terminal window shows the execution of a parser on a C program. The program has a parsing error at line 7. The symbol table and constant table are displayed below the error message.

```
agnitha@agnitha-Inspiron-3442: ~/Desktop/Project/Parser
agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser$ ./a.out<Test/missing-semicolon-quotes-brackets.c
PARSING ERROR at Line Number - 7
LEXICAL ERROR - " : invalid character at line [7]
```

SYMBOL TABLE

Name	Type
main	int
sum	int
a	int
b	int

CONSTANT TABLE

Constant	Type
2	constant
3	constant
0	constant

```
agnitha@agnitha-Inspiron-3442:~/Desktop/Project/Parser$
```

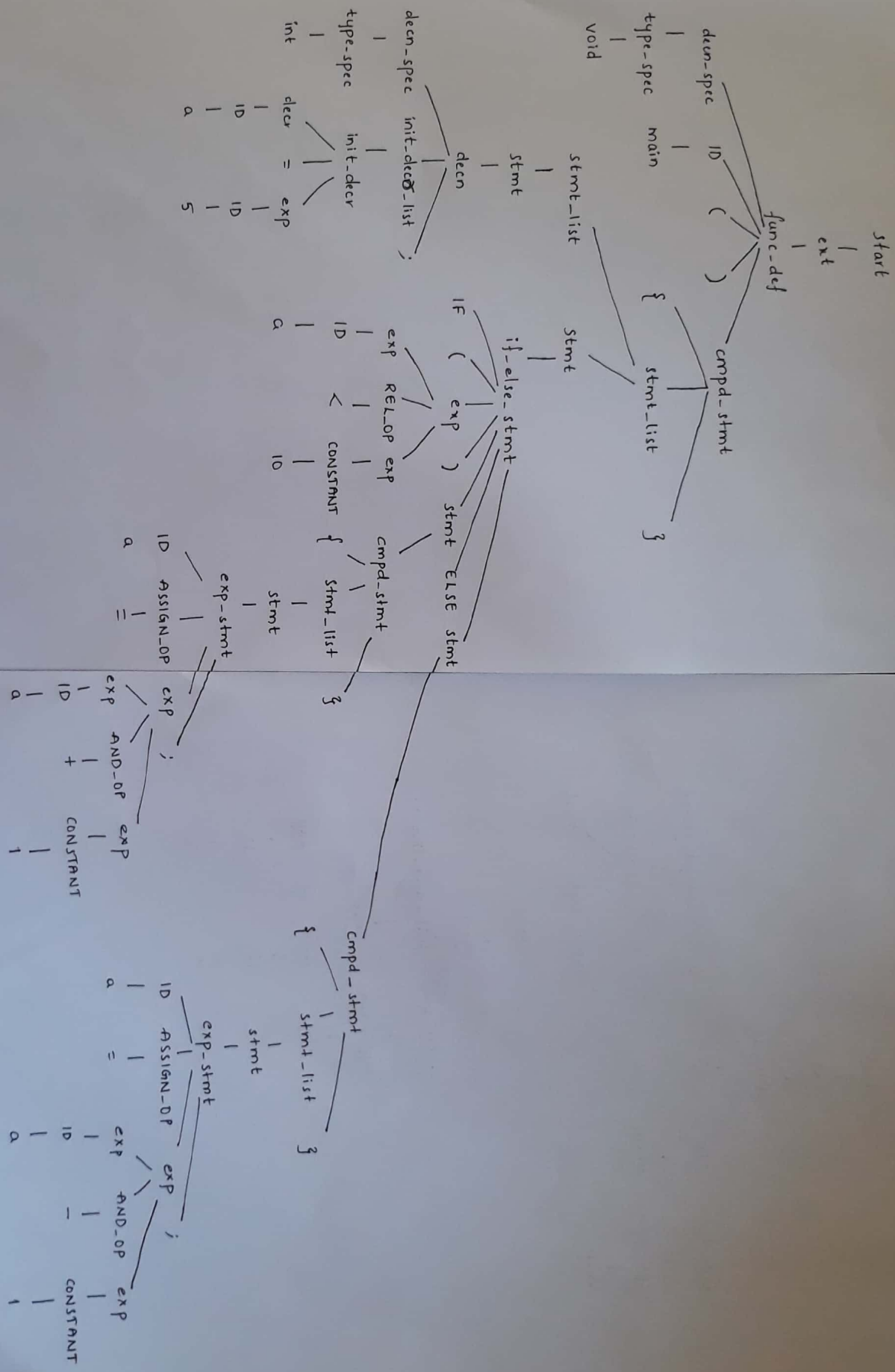
Description and Status of Test Cases:

TEST CASE	DESCRIPTION	STATUS
1	Simple for loop program to calculate factorial	PASS
2	Basic if-else	PASS
3	Nested-for-loop for finding prime numbers within a range	PASS
4	Dangling else	PASS
5	Nested for with error	PASS
6	Program with errors like: <ul style="list-style-type: none">• Unmatched paranthesis• Unmatched curly braces• Unmatched quotes• Unmatched multi line comment• Mismatched paranthesis Invalid identifier	PASS

Parse Tree

Input Program:

```
void main()
{
    float c;
    int a=5;
    int z,b;
    if(a<10)
    {
        a=a+1;
    }
    else
    {
        a=a-1;
    }
}
```



Input program:

```
void main()
{
    int num=50, i, sum = 1;

    for(i = 1; i <= num; ++i)
    {
        sum+=i;
    }
}
```


Conclusion

The project aimed to build a syntax analyzer which took stream of tokens as input from the lexical phase and checked for syntactical correctness of the program

It took care of lexical and syntax errors and used hashing concepts to optimize the storage of tokens. It populates the symbol table and constant table. The symbol table enlists the identifiers along with their data types, while the constant table enlists the constants.

References

- Compilers – Principles , Techniques and tools by Alfred V. Aho , Monica S. Larn , Ravi Sethi , Jeffrey D. Ullman
- Lex and Yacc by John R. Levine , Tony Mason , Doug Brown
- <https://en.wikipedia.org/wiki/>
- <https://www.tutorialspoint.>
- <https://www.lysator.liu.se/c/ANSI-C-grammar-y.html>
- <https://stackoverflow.com/questions/34493467/how-to-handle-nested-comment-in-flex>