CONTENTS:-

INTRODUCTION

What is a compiler?

In order to reduce the complexity of designing and building computers, nearly all of these are made to execute relatively simple commands (but do so very quickly). A program for a computer must be built by combining these very simple commands into a program in what is called machine language. Since this is a tedious and error prone process most programming is, instead, done using a high-level programming language. This language can be very different from the machine language that the computer can execute, so some means of bridging the gap is required.

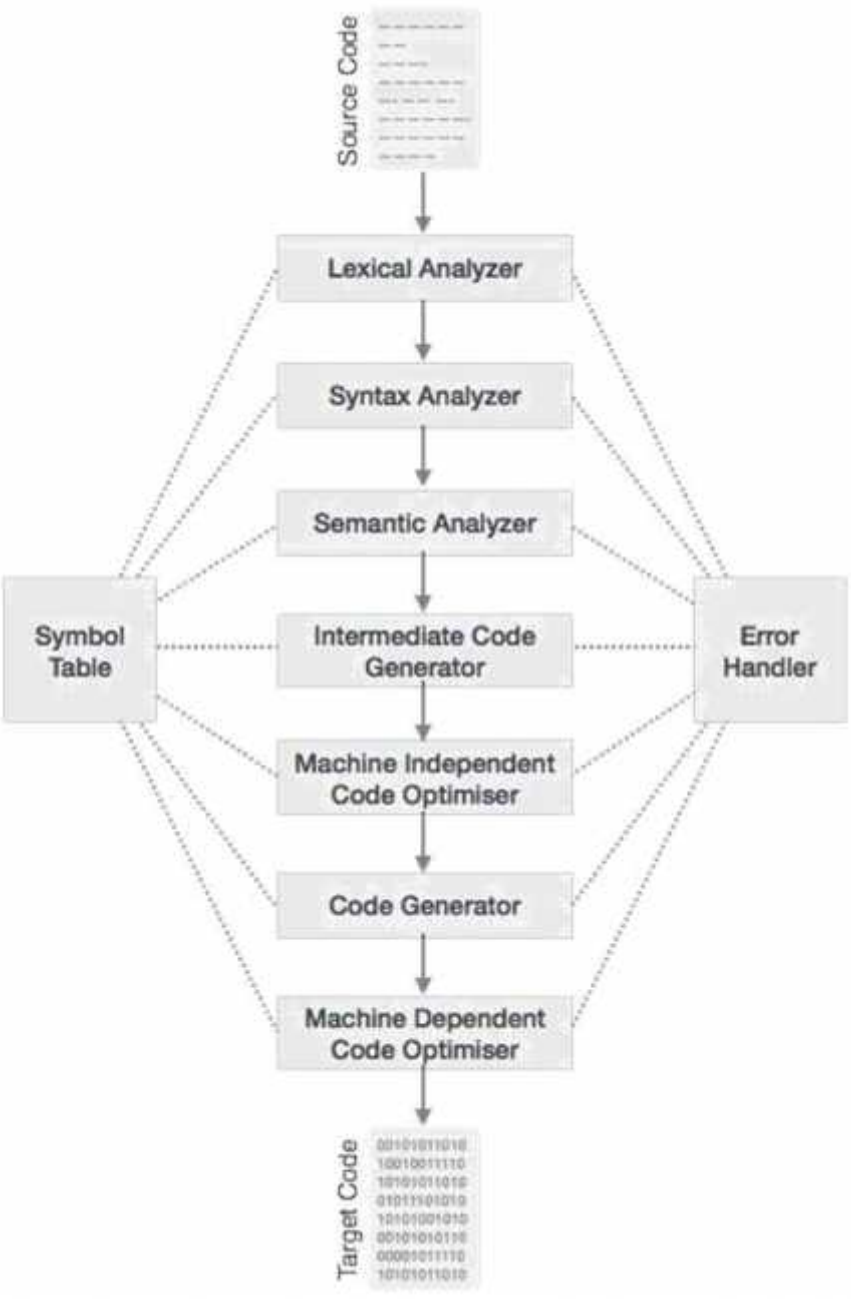
This is where the compiler comes in. A compiler translates (or compiles) a program written in a high-level programming language that is suitable for human programmers into the low-level machine language that is required by computers. During this process, the compiler will also attempt to spot and report obvious programmer mistakes. Using a high-level language for programming has a large impact on how fast programs can be developed.

The main reasons for this are:

- Compared to machine language, the notation used by programming languages is closer to the way humans think about problems.
- The compiler can spot some obvious programming mistakes.
- Programs written in a high-level language tend to be shorter than equivalent programs written in machine language.
- Another advantage of using a high-level level language is that the same program can be compiled to many different machine languages and, hence, be brought to run on many different machines.
- On the other hand, programs that are written in a high-level language and automatically translated to machine language may run somewhat slower than programs that are hand-coded in machine language. Hence, some time-critical programs are still written partly in machine language.
- A good compiler will, however, be able to get very close to the speed of hand-written machine code when translating well-structured programs.

THE PHASES OF A COMPILER

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.



Lexical Analysis

The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<Token-name, attribute-value>

Syntax Analysis

The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e., the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis

Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not, etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation

After semantic analysis, the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization

The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code Generation

In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do

Symbol Table

It is a data-structure maintained throughout all the phases of a compiler. All the identifiers' names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

LEXICAL ANALYSIS

Lexical analysis is the first phase of a compiler. It takes the modified source code from language pre processors that are written in the form of sentences. The lexical analyzer breaks these syntaxes into a series of tokens, by removing any hitespace or comments in the source code. If the lexical analyzer finds a token invalid, it generates an error. The lexical analyzer works closely with the syntax analyzer.
It reads character streams from the source code, checks for legal tokens, and passes the data to the syntax analyzer when it demands. The main purpose of lexical analysis is to make life easier for the subsequent syntax analysis phase. In theory, the work that is done during lexical analysis can be made an integral part of syntax analysis, and in simple systems this is indeed often done. However, there are reasons for keeping the phases separate:

- Efficiency: A lexer may do the simple parts of the work faster than the more general parser can. Furthermore, the size of a system that is split in two may be smaller than a combined system. This may seem paradoxical

but, as we shall see, there is a non-linear factor involved which may make a separated system smaller than a combined system.

- Modularity: The syntactical description of the language need not be cluttered with small lexical details such as white-space and comments.
- Tradition: Languages are often designed with separate lexical and syntactical phases in mind, and the standard documents of such languages typically separate lexical and syntactical elements of the languages.

Token: Token is a sequence of characters that can be treated as a single logical entity. Typical tokens are,

1) Identifiers
2) Keyword
3) operators
4) special symbols
5) constants

Pattern: A set of strings in the input for which the same token is produced as output. This set of strings is described by a rule called a pattern associated with the token.

Lexeme: A lexeme is a sequence of characters in the source program that is matched by the pattern for a token.

DESCRIPTION OF THE PROGRAMMING LANGUAGE

Your scanner should break up a program into a stream of tokens. The tokens are described in the description of the programming language. Here are some suggestions for building your scanner. These are only suggestions and you may construct it any way you like. However, your scanner must:

1. Find tokens
2. Install a new identifier in the symbol table and return the address
3. Return the symbol table address of a previously installed identifier
4. Produce a numbered listing of the source program
5. Produce error messages for **LEXICAL** errors.

The first decision you must make is what the format of your tokens is. One idea might be to return an object for each token. The first field might be the token type and the second field might be the address (if applicable). If the token is a number, the value must be returned

For example:

Iden 23
begin
assign
string 12
plus
iden 15
div
number 26437

You mighy (but do not have to) use named constants for your tokens

final int ident= 1;
final int while$ = 2;
final int lessThan = 3; etc.

Be careful, do not use a reserved word as one of your token names list (eg use **while$** not **while**)

a) Lexical Units and Spacing

The lexical units of a program are identifiers, reserved words, numbers, strings, and delimiters.

A delimiter is any one of the following special characters:

( ) [ ] ; : . , * - + / < = >

or one of the following compound symbols:

<> := <= >=

Spaces may be inserted freely with no effect on meaning between lexical units. At least one space must separate adjacent identifiers or numbers. The end of a line is equivalent to a space except (1) it terminates comments and (2) is illegal in quoted character strings (see below). Thus each lexical unit must fit on one line.

b) **IDENTIFIERS**

Identifiers (names) consist of a letter followed by a sequence of letters or digits. Upper and lower case are considered equivalent. An identifier may be any length but must be distinguishable within the first 32 characters. Thus the two names a11111111111111111111111111112222 and a11111111111111111111111111119999 are not distinguishable.

c) NUMBERS

The only kind of number is the integer. An integer is a sequence of digits. Maximum and minimum values of integers are determined by our implementation.

d) STRINGS

A character string is a sequence of characters prefixed and terminated by the apostrophe character. The backslash character (\) acts as an escape. When followed by an n or a t, the backslash denotes a new line (CR) or tab. When followed by another character it denotes that character. Thus \' denotes the single quote. For example, the string **DON'T** would be represented as 'DON\'T'.

e) COMMENTS

A comment starts with an exclamation point (!) and is terminated by the end of a line. It may appear following a lexical unit or at the beginning of a program.

```
!This is a comment
```

f) RESERVED WORDS

The identifiers listed below are reserved words and may not be declared by the programmer:

and
array
begin
integer
do
else
end
function
if
of
or
not
procedure
program
read
then
var
while
write

How To Begin

The basic purpose of your scanner is to return the next token to the caller. (The caller will be the parser. How should you begin writing the scanner?

1. Design a finite state machine to model the scanner.

2. Implement your FSM (use a table). This version will assign 0 to the second field of the token. Produce a listing file i.e. a version of the program with line numbers. You can also do the symbol table simultaneously.

3. Test this version of your scanner on sample programs which contain no errors.

4. Add the string table and a symbol table, if you have not already.
The symbol table can be an array of objects and you can use a linear search

5. Modify the scanner so that it can handle lexical errors.

6. Your program should prompt the user for an input file. All input files should end with the extension .pas

7. Your scanner should output a listing file. The listing file has the same name as the source file but the extension is .lis

8. For now you might write the token stream to a file called tokens.dat

Here is a typical source file. You should try many variations. Don't worry about syntax now.
The design of your program and even your implementation language is up to you. However, your compiler should be object oriented. In fact, your scanner should be an object. There should probably be a method get Token () which returns a token.

Problem statement:

Design a c/c++ compiler for the following pseudo code:-

```
int main()
begin
    int L[10];
    int maxval=L[0];
    for i=1 to n-1 do
        if L[i]>maxval
            maxval=L[i];
        endif
    endfor
    return(maxval)
```

end

Scanner code for the given hypothetical code:-

```
#include<stdio.h>
#include<ctype.h>
#include<string.h>
void keyw(char *p);
int i=0,id=0,kw=0,num=0,op=0;
char keys[2][10]={"for","int"};
main()
{
        char ch,str[25],seps[15]=" \t\n,;(){}[]#\"<>",oper[]="!%^&*-  +=~|.<>/?";
        int j;
        char fname[50];
        FILE *f1;
        //clrscr();
printf("enter file path (drive:\\fold\\filename)\n");
scanf("%s",fname);
f1 = fopen(fname,"r");
//f1 = fopen("Input","r");
        if(f1==NULL)
        {
         printf("file not found");
         exit(0);
        }
        while((ch=fgetc(f1))!=EOF)
        {
for(j=0;j<=14;j++)
{
if(ch==oper[j])
{
printf("%c is an operator\n",ch);
op++;
str[i]='\0';
keyw(str);
}
}
for(j=0;j<=14;j++)
{
```

```
if(i==-1)
break;
if(ch==seps[j])
{
if(ch=='#')
{
while(ch!='>')
{
printf("%c",ch);
ch=fgetc(f1);
}
printf("%c is a header file\n",ch);
i=-1;
break;
}
if(ch=='"')
{
do
{
ch=fgetc(f1);
printf("%c",ch);
}while(ch!='"');
printf("\b is an argument\n");
i=-1;
break;
}
str[i]='\0';
keyw(str);
}
}
if(i!=-1)
{
str[i]=ch;
i++;
}
else
i=0;
        }
printf("Keywords: %d\nIdentifiers: %d\nOperators: %d\nNumbers:
%d\n",kw,id,op,num);
//getch();
}
void keyw(char *p)
```

```
{
int k,flag=0;
for(k=0;k<=1;k++)
{
if(strcmp(keys[k],p)==0)
{
printf("%s is a keyword\n",p);
kw++;
flag=1;
break;
}
}
if(flag==0)
{
if(isdigit(p[0]))
{
printf("%s is a number\n",p);
num++;
}
else
{
//if(p[0]!=13&&p[0]!=10)
if(p[0]!='\0')
{
printf("%s is an identifier\n",p);
id++;
}
}
}
i=-1;
}
```

Explanation for the above code

In the code above, an array of keywords that are used in the program have been declared.

When a statement of the program is given as input to the program, the strtok function divides the input with space, semicolon and comma as delimiter and assigns it to a variable named token every time in the loop.

Every time the token is compared with each keyword in the array and if it matches any of them,then it is displayed as a keyword.

If it matches with any of the operators, then it’s an operator.

If it matches with numbers, then it’s a number.

Else it is an identifier.

Output:-

```
sandesh@sandesh-Lenovo-G50-80:~$ ./a.out
enter file path (drive:\fold\filename)
input.txt
int is a keyword
main is an identifier
begin is an identifier
int is a keyword
l is an identifier
10 is a number
int is a keyword
max is an identifier
for is a keyword
= is an operator
i is an identifier
0 is a number
to is an identifier
- is an operator
n is an identifier
1 is a number
do is an identifier
if is an identifier
l is an identifier
i is an identifier
> is an operator
max is an identifier
= is an operator
max is an identifier
l is an identifier
i is an identifier
endif is an identifier
endfor is an identifier
return is an identifier
max is an identifier
end is an identifier
Keywords: 4
Identifiers: 20
Operators: 4
Numbers: 3
```

SYNTAX ANALYSIS

Syntax analysis or parsing is the second phase of a compiler. In this chapter, we shall learn the basic concepts used in the construction of a parser.

Role of the Parser

- In the compiler model, the parser obtains a string of tokens from the lexical analyser, and verifies that the string can be generated by the grammar for the source language.
- The parser returns any syntax error for the source language.

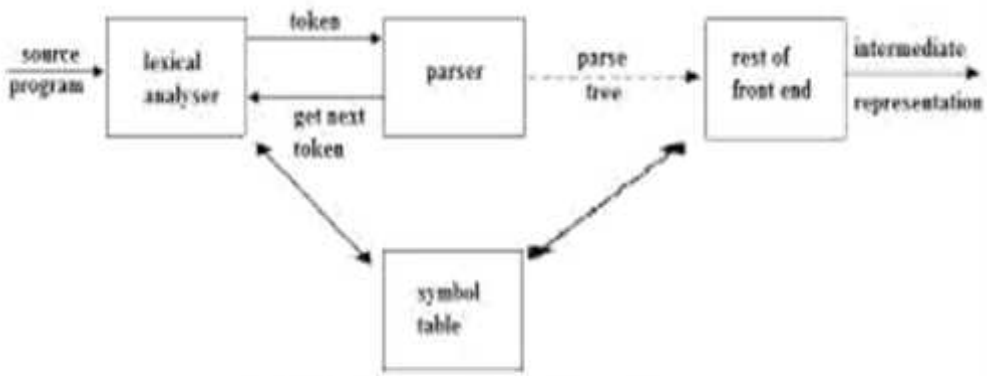


Fig 2.1 Position of parser in compiler model

- There are three general types' parsers for grammars.
  1. Universal parsing methods such as the Cocke-Younger-Kasami algorithm and Earley's algorithm can parse any grammar. These methods are too inefficient to use in production compilers.
  2. The methods commonly used in compilers are classified as either top-down parsing or bottom-up parsing.
  3. Top-down parsers build parse trees from the top (root) to the bottom (leaves).

4. Bottom-up parsers build parse trees from the leaves and work up to the root.
5. In both case input to the parser is scanned from left to right, one symbol at a time.
6. The output of the parser is some representation of the parse tree for the stream of token.
7. There are number of tasks that might be conducted during parsing. Such as; o Collecting information about various tokens into the symbol table.
8. Performing type checking and other kinds of semantic analysis. o Generating intermediate code.

Syntax Error Handling:

- Planning the error handling right from the start can both simplify the structure of a compiler and improve its response to errors.
- The program can contain errors at many different levels. e.g.,

1. Lexical – such as misspelling an identifier, keyword, or operator.
2. Syntax – such as an arithmetic expression with unbalanced parenthesis.
3. Semantic – such as an operator applied to an incompatible operand.
4. Logical – such as an infinitely recursive call.
5. Much of the error detection and recovery in a compiler is centred on the syntax analysis phase.
6. One reason for this is that many errors are syntactic in nature or are exposed when the stream of tokens coming from the lexical analyser disobeys the grammatical rules defining the programming language.
7. Another is the precision of modern parsing methods; they can detect the presence of syntactic errors in programs very efficiently.
8. The error handler in a parser has simple goals:
9. It should the presence of errors clearly and accurately.
10. It should recover from each error quickly enough to be able to detect subsequent errors.
11. It should not significantly slow down the processing of correct programs.

Error-Recovery Strategies:

- There are many different general strategies that a parser can employ to recover from a syntactic error.

1. Panic mode
2. Phrase level
3. Error production
4. Global correction

CONTEXT-FREE GRAMMAR

In this section, we will first see the definition of context-free grammar and introduce terminologies used in parsing technology.

A context-free grammar has four components:

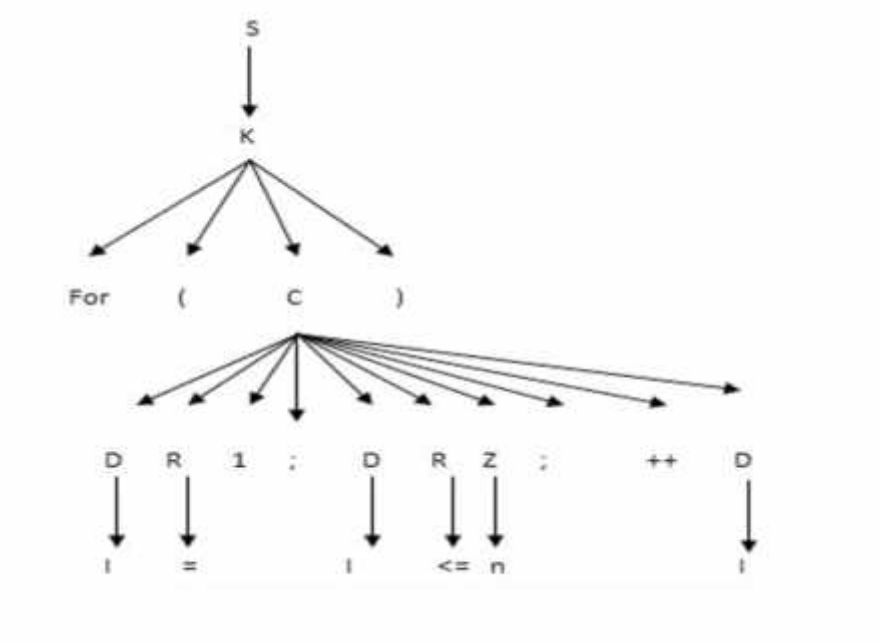
- A set of **non-terminals** (V). Non-terminals are syntactic variables that denote sets of strings. The non-terminals define sets of strings that help define the language generated by the grammar.
- A set of tokens, known as **terminal symbols** (Σ). Terminals are the basic symbols from which strings are formed.
- A set of **productions** (P). The productions of a grammar specify the manner in which the terminals and non-terminals can be combined to form strings. Each production consists of a **non-terminal** called the left side of the production, an arrow, and a sequence of tokens and/or **on- terminals**, called the right side of the production.
- One of the non-terminals is designated as the start symbol (S); from where the production begins.

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

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. Let us see this by an example from the last topic.

We take the left-most derivation of a +
b * c The left-most derivation is:



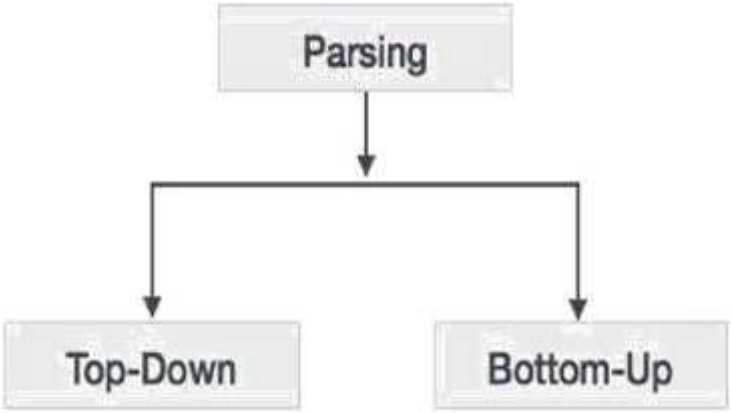
In a parse tree:

- All leaf nodes are terminals.
- All interior nodes are non-terminals.
- In-order traversal gives original input string.

A parse tree depicts associativity and precedence of operators. The deepest sub-tree is traversed first, therefore the operator in that sub-tree gets precedence over the operator which is in the parent nodes.

Types of Parser:-

Syntax analysers follow production rules defined by means of context-free grammar. The way the production rules are implemented (derivation) divides parsing into two types: top-down parsing and bottom-up parsing



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.

NOTE: THE TYPE OF PARSER WE HAVE USED IS TOP-DOWN PARSER

LL (1) Grammars

LL (1) GRAMMARS AND LANGUAGES. A context-free grammar G = (VT, VN, S, P) whose parsing table has no multiple entries is said to be *LL*(1). In the name *LL* (1),

- the first *L* stands for scanning the input from **l**eft to right,
- the second *L* stands for producing a **l**eftmost derivation,
- And the 1 stands for using **one** input symbol of look ahead at each step to make parsing action decision.

A language is said to be LL (1) if it can be generated by a LL (1) grammar. It can be shown that LL (1) grammars are

1. not ambiguous and
2. Not left-recursive.

eg: LL(1) grammar for above problem statement is

S->K
K->intA
K->begin
K->endif
K->end
K->returnRDR
K->ifB
A->main()
A->DR10R;
A->TR0;
A->MRDR10R;
B->DRTRRM
D->l
T->i
M->maxval
R->[
R->]
R->(
R->)
R->=

R-> >
K->maxvalRDRTR

First and Follow Sets

An important part of parser table construction is to create first and follow sets. These sets can provide the actual position of any terminal in the derivation. This is done to create the parsing table where the decision of replacing T[A, t] = α with some production rule.

First Set

This set is created to know what terminal symbol is derived in the first position by a non-terminal. For example,

$\alpha \rightarrow t\,\beta$ $\in$

That is, α derives t (terminal) in the very first position. So, t

FIRST(α).

Algorithm for Calculating First Set

Look at the definition of FIRST(α)

set:

if αis a terminal, then FIRST(α) = { α}.

if αis a non-terminal and $\alpha \rightarrow \mathcal{E}$ is a production, then FIRST(α) = { Ɛ }.

if αis a non-terminal and $\alpha \rightarrow 1\ 2\ 3 \ldots$ n and any FIRST( ) contains t, then t is in FIRST(α).

First set can be seen as: FIRST(α) = { t | $\alpha \rightarrow^* t\,\beta$ } ∪ { Ɛ | $\alpha \rightarrow^* \varepsilon$}

Follow Set

Likewise, we calculate what terminal symbol immediately follows a non-terminal α in production rules. We do not consider what the non-terminal can generate but instead, .we see what would be the next terminal symbol that follows the productions of a non-terminal

Algorithm for Calculating Follow Set:

if αis a start symbol, then FOLLOW() = $

if αisƐ a non-terminal and has a production α → AB, then FIRST(B) is in

FOLLOW(A) except .

if αis a non-terminal and has a production α → AB, where B Ɛ, then

FOLLOW(A) is in FOLLOW(α).

Follow set can be seen as: FOLLOW(α) = { t | S *αt*}
eg : first and follow for the above problem statement is

FIRST(S)={int,if,begin,endif,return,maxval}
FIRST(K)={int,maxval,begin,endif,end,return,if}
FIRST(A)={i,l,main,maxval}
FIRST(D)={l}
FIRST(B)={l}
FIRST(T)={i}
FIRST(M)={maxval}
FIRST(R)={[,],(,),=,>}

FOLLOW(S)={$}
FOLLOW(K)={$}
FOLLOW(A)={$}
FOLLOW(B)={$}
FOLLOW(T)={$}
FOLLOW(R)={$}
FOLLOW(M)={$}
FOLLOW(D')={$}

First Code to accept the line from the source:-

The following code accepts the input (single line from the source) and checks for the matching.

```
#include <stdio.h>
#include <stdlib.h>
#include<string.h>
int r=0,c=0;
char item[50],item[50];
char a[25][25][11]={
{"S","K","$"},
{"K","int","A","$"},
{"K","begin","$"},
{"K","endif","$"},
{"K","end","$"},
{"K","return ","R","D","R","$"},
{"K","if","B","$"},
{"A","main","(",")","$"},
{"A","D","R","10","R",";","$"},
{"A","T","R","0",";","$"},
{"A","M","R","D","R","10","R",";","$"},
{"B","D","R","T","R","R","M","$"},
{"D","l","$"},
{"T","i","$"},
{"M","maxval","$"},
{"R","[","$"},
{"R","]","$"},
{"R","(","$"},
{"R",")","$"},
{"R","=","$"},
{"R",">","$"},
{"K","maxval","R","D","R","T","R",";","$"}
};

char b[40][40][10]={
{"S","int","0"},
{"S","if","0"},
{"S","begin","0"},
{"S","end","0"},
{"S","endif","0"},
{"S","return","0"},
{"S","maxval","0"},
{"K","int","1"},
{"K","begin","2"},
{"K","maxval","21"},
{"K","endif","3"},

{"K","end","4"},
```

```
{"K","return","5"},

{"K","if","6"},
{"A","main","7"},
{"A","l","8"},
{"A","i","9"},
{"A","maxval","10"},
{"B","l","11"},
{"D","l","12"},
{"T","i","13"},
{"M","maxval","14"},
{"R","[","15"},
{"R","]","16"},
{"R","(","17"},
{"R",")","18"},
{"R","=","19"},
{"R",">","20"},

};
int prod(int x , char str[]);
char stack[10][110];
char queue[10][100];
char *token;
int count;
int l=0,i,k,front=0,rear=-1,state=1,r;
char str[50],stp[100];
int main()
{
int x;strcpy(stack[r],"S");
printf("Enter the string to be parsed\n (with $ at the
end )\n");
while(strcmp(queue[rear],"$")!=0)
{
rear++;
scanf("%s",queue[rear]);
}
l1:
strcpy(stp,queue[front]);
if((strcmp(stp,")")==0)||(strcmp(stp,"(")==0)||(strcmp(
stp,";")==0)||(strcmp(stp,",")==0)||
(strcmp(stp,"1")==0)||(strcmp(stp,"0")==0))
{
printf("%s matched\n",stp);
front++;
```

```
r++;
goto l1;
}
while(strcmp(stp,"$")!=0)
{
for(i=0;i<40;i++)
{
strcpy(str,stack[r]);
strcpy(stp,queue[front]);
if((strcmp(b[i][0],str)==0)&&(strcmp(b[i][1],stp)==0))
{
x=atoi(b[i][2]);
r=prod(x,str);
}
else if((strcmp(str,stp))==0)
{
strcpy(item,queue[front]);
printf("%s matched\n",item);
front++;
strcpy(str,queue[front]);
strcpy(item,stack[r]);
r++;
goto l1;
}
}
}
if(strcmp(stack[r],"\0")!=0)
{
printf("not correct syntax\n");
}
printf("String is Accepted\n");
return 0;
}
int prod(int x,char str[]){
char stq[100];
int y=1,count=0;
printf("%s ->",a[x][0]);
l2 : strcpy(stq,a[x][y]);
while(strcmp(stq,"$")!=0)
{
strcpy(str,stq);
printf("%s",str);
strcpy(stack[r],str);
count++;
r++;y++;
```

```
goto l2;
}
if(count>r)
{
r = count-r;
}
else
{
r = r-count;
}
printf("\n");
return r;
}
```

Output:-

```
sandesh@sandesh-Lenovo-G50-80:~$ cc int.c
sandesh@sandesh-Lenovo-G50-80:~$ ./a.out
Enter the string to be parsed
 (with $ at the end )
int main ( )
$
S ->K
K ->intA
int matched
A ->main()
main matched
( matched
) matched
String is Accepted
sandesh@sandesh-Lenovo-G50-80:~$ ./a.out
Enter the string to be parsed
 (with $ at the end )
maxval = l [ i ] ;
$
S ->K
K ->maxvalRDRTR;
maxval matched
R ->=
= matched
D ->l
l matched
R ->[
[ matched
T ->i
i matched
R ->]
] matched
; matched
String is Accepted
sandesh@sandesh-Lenovo-G50-80:~$
```

Second Code to accept the entire code from the source:-

Here this program accepts entire languages, if there is a single mismatch then it casts an error.

main.c is code file includes main.h and parser.c.
parser.c file includes intstack.c file.

$ cat main.h

```
#define max 100
struct stack {
  char stk[max][80];
  int top;
}tokens;
void push( struct stack *x,char a[80])
{
      if(x->top == max -1)
            return;
      else
      {
            x->top+= 1;
            strcpy(x->stk[x->top], a);
      } // else
} // push

char * pop(struct stack * x)
{
      if(x->top == -1)
            return "\0";
      else
      {
    x->top -= 1;
            return x->stk[x->top+1];
      } // else
} // pop
```

$ cat intstack.h

```
#define maxint 100
struct stacki {
  int stk[maxint];
  int top;
}nest;
void ipush( struct stacki *x,int a)
{
      if(x->top == maxint -1)
            return;
      else
      {
            x->top+= 1;
            x->stk[x->top]= a;
```

```
     } // else
} // push

int  ipop(struct stacki * x)
{
     if(x->top == -1)
          return -1;
     else
     {
    x->top -= 1;
          return x->stk[x->top+1];
     } // else
} // pop
```

$ cat parser.c

```
#include "intstack.h"
//data
char t_col[21][10]={"int",  //0
                    "main",//1
                    "begin",//2
                    "end",//3
                    ";",//4
                    "id",//5
                    "[",//6
                    "]",//7
                    "number",//8
                    "=",//9
                    "endfor",//10
                    "do",//11
                    "operator",//12
                    "to",//13
                    "for",//14
                    "endif",//15
                    "if",//16
                    "return",//17
                    "(",//18
                    ")",//19
                    "$"//20
                  };
int t_size = sizeof(t_col)/sizeof(t_col[0]);

int parse_table[36][21]={
//0   1   2   3   4   5   6   7   8   9     10  11  12
13  14  15  16  17  18  19  20
 {0  ,0  ,0  ,1  ,4  ,0  ,0  ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//0
```

```
 {0  ,0  ,2  ,0  ,4  ,0  ,0  ,0  ,0   ,0    ,8  ,0  ,0
,0  ,0  ,19 ,0  ,0  ,0  ,33 ,0  },//1
 {0  ,3  ,0  ,0  ,0  ,0  ,0  ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,32 ,0  },//2
 {5  ,0  ,2  ,0  ,0  ,0  ,0  ,0  ,0   ,4    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//3
 {0  ,0  ,0  ,0  ,0  ,3  ,0  ,6  ,3   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//4
 {0  ,0  ,2  ,0  ,4  ,0  ,0  ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,-1 },//5
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,7   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//6
 {0  ,0  ,0  ,0  ,0  ,0  ,4  ,6  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//7
 {0  ,0  ,0  ,0  ,23 ,0  ,0  ,0  ,0   ,0    ,8  ,9  ,0
,0  ,0  ,19 ,0  ,0  ,0  ,0  ,0  },//8
 {0  ,0  ,0  ,0  ,0  ,10 ,0  ,0  ,10  ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//9
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0   ,0    ,0  ,0  ,11
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//10
 {0  ,0  ,0  ,0  ,0  ,12 ,0  ,0  ,12  ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//11
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0   ,0    ,0  ,0  ,13
,14 ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//12
 {0  ,0  ,0  ,0  ,0  ,13 ,0  ,0  ,13  ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//13
 {0  ,0  ,0  ,0  ,0  ,15 ,0  ,0  ,15  ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//14
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0   ,16   ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//15
 {0  ,0  ,0  ,0  ,0  ,17 ,0  ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//16
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0   ,18   ,0  ,0  ,0
,0  ,1  ,0  ,0  ,0  ,0  ,0  ,0  },//17
 {0  ,0  ,0  ,0  ,0  ,17 ,0  ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//18
 {0  ,0  ,0  ,0  ,23 ,20 ,0  ,0  ,20  ,0    ,8  ,9  ,0
,0  ,0  ,19 ,0  ,0  ,0  ,0  ,0  },//19
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0   ,0    ,0  ,0  ,21
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//20
 {0  ,0  ,0  ,0  ,0  ,22 ,0  ,24 ,22  ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//21
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,8  ,0  ,0  ,0  ,0  },//22
```

```
 {0  ,0  ,0  ,0  ,0  ,27 ,0  ,24 ,27  ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//23
 {0  ,0  ,0  ,0  ,0  ,25  ,0  ,0  ,25  ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//24
 {0  ,0  ,0  ,0  ,0  ,0  ,26 ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//25
 {0  ,0  ,0  ,0  ,0  ,27 ,0  ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//26
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0   ,28   ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//27
 {0  ,0  ,0  ,0  ,0  ,19 ,0  ,29 ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//28
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,30  ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//29
 {0  ,0  ,0  ,0  ,0  ,0  ,31 ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//30
 {0  ,0  ,0  ,0  ,0  ,19 ,0  ,0  ,0   ,0    ,0  ,9  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//31
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,2  ,0  ,0  },//32
 {0  ,0  ,0  ,0  ,0  ,34 ,0  ,0  ,34  ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  },//33
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,0  ,35 ,0  ,0  },//34
 {0  ,0  ,0  ,0  ,0  ,0  ,0  ,0  ,0   ,0    ,0  ,0  ,0
,0  ,0  ,0  ,0  ,1  ,0  ,0  ,0  }//35
};


//functions
int stoi(char * a){
  char c;
  int i=0,val=0;
  while((c=a[i++])!='\0'){
    if((c-'0')>9||(c-'0')<0) return -1;
  //  printf("%d\n",c-'0');
    val=val*10+(c-'0');
  }
  return val;
}

int getIndex(char * a){
  int i;
  for(i=0;i<t_size;i++){
    if(strcmp(t_col[i],a)==0)
```

```
      return i;
    }
  //printf("\nid:%d\n",a[0]-'0');
  if(a[0]=='-'||a[0]=='>'){
    return getIndex("operator");
  }
  //identifier
  if((a[0]-'0')>9||a[0]=='_')
      return getIndex("id");
  //number
  if(stoi(a)!=-1)
    return getIndex("number");
 return -1;
}

int getNextState(int state,int index){
  return parse_table[state][index];
}



int parse(struct stack *st){

char *p;
int state=0;
nest.top=-1;
while(st->top!=-1){
 p = pop(st);
 if(strcmp("endfor",p)==0||strcmp("endif",p)==0){
   ipush(&nest,state);
 }
 if(nest.top>=0){
   if(strcmp("for",p)==0||strcmp("if",p)==0){
     state=ipop(&nest);
     continue;
   }
 }
 state = getNextState(state,getIndex(p));
 if(getIndex(p)==-1||state==0){
   printf("\n\nsyntax error near %s\n\n", p);

  printf("symbol
%s\tnextstate:%d\t\tindex:%d\n",p,state,getIndex(p));
   break;
 }
```

```
printf("symbol
%s\tnextstate:%d\t\tindex:%d\n",p,state,getIndex(p));
}


if(st->top==-1&&state==-1&&nest.top==-1){
  printf("accepted\n" );
}
else{
  printf("rejected\n" );
}
return 0;
}
```

$ cat main.c

```
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<unistd.h>
#include<fcntl.h>
#include "main.h"
#include "parser.c"

int main(int argc,char* argv[]){
 int fd,ttracker=0,ivar=0,t=0,i;
 char *a ;
 char in;
 char token[20]={'\0'};

/*
//testing
printf("\n%d\n",getIndex("$s"));
return 0;
//testing
*/
tokens.top=-1;

push(&tokens,"$");
 if(argc<2){
    printf("Error: Input file unspecified %d",']');
    exit(0);
 }
  fd = open(argv[1],O_RDONLY);
```

```
  while((read(fd,&in,1))==1){

if(in==59||in==61||in==45||in==62||in==91||in==93||in==
'('||in==')'){
      if(ttracker)
      push(&tokens,token);
      while(ttracker--){
            token[ttracker]='\0';
        }
      ttracker=0;
      token[0]=in;
      push(&tokens,token);
    }
    else if(in==32||in==10){
         if(ttracker==0){
            continue;
        }
        //printf("\ntoken : %s",token);

    push(&tokens,token);
        token[ttracker]='\0';
        while(ttracker--){
            token[ttracker]='\0';
        }
        ttracker=0;
       }
     else{
        token[ttracker] = in;
        ttracker++;
    }
  }
  parse(&tokens);
/*
  while(tokens.top){
    printf("\ntoken : %s",pop(&tokens));
  }*/


  printf("\n");
  close(fd);
}
```

Output:-

```
sandesh@sandesh-Lenovo-G50-80:~$ ./a.out input.txt
symbol end      nextstate:1             index:3
symbol )        nextstate:33            index:19
symbol max      nextstate:34            index:5
symbol (        nextstate:35            index:18
symbol return   nextstate:1             index:17
symbol endfor   nextstate:8             index:18
symbol endif    nextstate:19            index:15
symbol ;        nextstate:23            index:4
symbol ]        nextstate:24            index:7
symbol i        nextstate:25            index:5
symbol [        nextstate:26            index:6
symbol l        nextstate:27            index:5
symbol =        nextstate:28            index:9
symbol max      nextstate:19            index:5
symbol max      nextstate:28            index:5
symbol >        nextstate:21            index:12
symbol ]        nextstate:24            index:7
symbol i        nextstate:25            index:5
symbol [        nextstate:26            index:6
symbol l        nextstate:27            index:5
symbol do       nextstate:9             index:11
symbol 1        nextstate:10            index:8
symbol -        nextstate:11            index:12
symbol n        nextstate:12            index:5
symbol to       nextstate:14            index:13
symbol 0        nextstate:15            index:8
symbol =        nextstate:16            index:9
symbol i        nextstate:17            index:5
symbol ;        nextstate:4             index:4
symbol max      nextstate:3             index:5
symbol int      nextstate:5             index:0
symbol ;        nextstate:4             index:4
symbol ]        nextstate:6             index:7
symbol 10       nextstate:7             index:8
symbol [        nextstate:4             index:6
symbol l        nextstate:3             index:5
symbol int      nextstate:5             index:0
symbol begin    nextstate:2             index:2
symbol )        nextstate:32            index:19
symbol (        nextstate:2             index:18
symbol main     nextstate:3             index:1
symbol int      nextstate:5             index:0
symbol $        nextstate:-1            index:28
accepted
```

```
sandesh@sandesh-Lenovo-G50-80:~$ cat input.B
int main()
begin
int L[10];
int maxval=L[0];
for i=1 to n-1 do
if L[i]>maxval
maxval=L[i];
endif
endfor
return(maxval)

sandesh@sandesh-Lenovo-G50-80:~$ cat input.txt
int main()
begin
  int l[10];
  int max;
  for i=0 to n-1 do
    if l[i]>max
      max=l[i];
    endif
  endfor
return(max)
end
sandesh@sandesh-Lenovo-G50-80:~$ ./a.out input.B


syntax error near )

symbol )       nextstate:0          index:19
rejected
```

CONCLUSION

In lexical analysis when we give a program statement as input, the keywords, identifiers, operators and numbers are displayed. Each of these are the tokens of the statement.

In top down parser, on giving an input string, the string is parsed as per the grammar and parsing table given in the program. If the string was parsed completely then the leftmost derivation of the string is displayed in the output else a syntax error is displayed.

The hands on experience on project gave more practical experience of designing the compiler by ourselves.