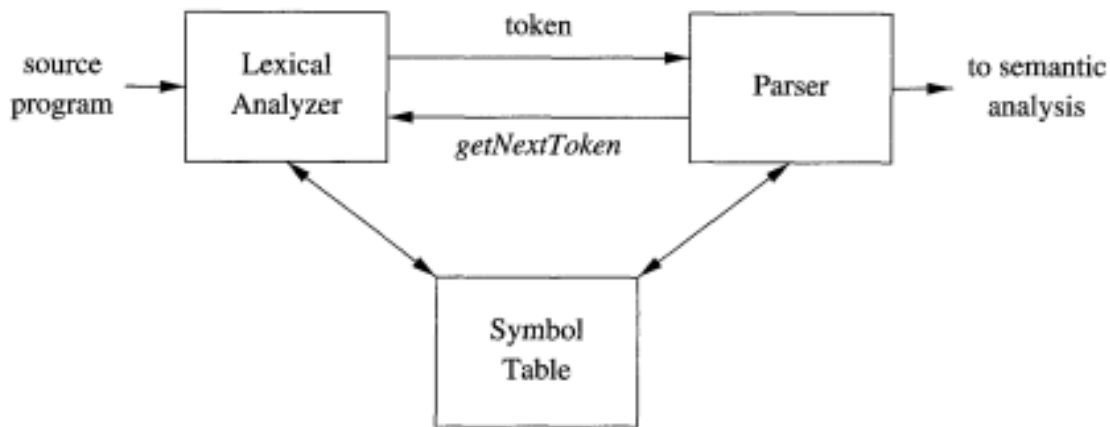


Compiler Design

Lexical Analysis



As the first phase of a compiler, the main task of the lexical analyser:

- read the input characters of the source program, for eg. 'i', 'n', 't', ' ', 'm', 'a', 'i', 'n', '(', ')', ...
- group them into lexemes. For eg. `int main()` ...
- produce as output a sequence of tokens for each lexeme in the source program. For eg. `<int>`, `<id,1>`, ...
- The stream of tokens is sent to the parser for syntax analysis.
- the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
- If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyzer. For eg.
 `#include<stdio.h> ===> File Inclusion`
 `#define x = 6; ===> macro expansion`

Tokens, Patterns, and Lexemes

1. A *token* is a pair consisting of a token name and an optional attribute value, i.e. `<token-name, attribute-value>`. The token name is an abstract symbol representing a kind of lexical unit, e.g., a particular keyword, or a sequence of input characters denoting an identifier. The token names are the input symbols that the parser processes. Eg. `int x = a - b + c`
2. A *lexeme* is a sequence of characters in the source program that matches the **pattern** for a token and is identified by the lexical analyzer as an instance of that token.
3. A **pattern** is a description of the form that the lexemes of a token may take.
 - a. keywords: sequence of characters
 - b. identifiers: starts with any alphabet or an underscore followed by any number of alphanumeric characters. For eg. `abc`, `_abc123`
 - c. numbers: any sequence of digits where each digit can be from 0 to 9.
- d. Operators: `[+, -, *, /, ...]`

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters i, f	if
else	characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	letter followed by letters and digits	pi, score, D2
number	any numeric constant	3.14159, 0, 6.02e23
literal	anything but ", surrounded by "'s	"core dumped"

Q1. Find the number of tokens in the following:

a.

```
main(){
    printf("cd");
    // prints the message
}
```

main, (,), {, , printf, (, "cd",),, }, } → 10 tokens

b.

```
while(i>0){
    printf(i);
    i++;
}
```

while, (, i,>,0,){,printf,(i,),,;,i,++,;, } → 16 tokens

c.

```
char *(a+5) = "abcd";
```

 → 10 Tokens

d.

```
char a[5] = "abcd";
```

 →

```
char *(a+5) = "abcd";
```

e.

```
int a[5][4];
```

 →

```
int *(*a+5)+4;
```

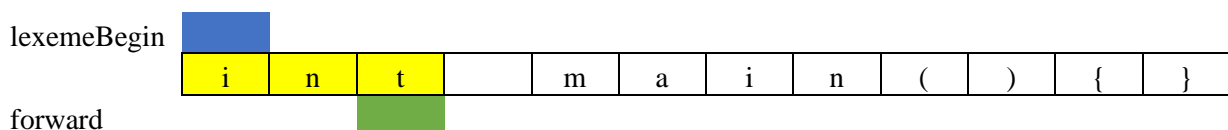
Input Buffering

- lexemeBegin pointer:** points to the beginning character of the current lexeme
- forward pointer:** Initially points to the beginning of the current lexeme. Advances forward as the LA scans the source program.

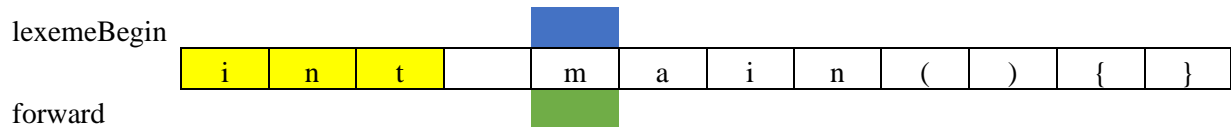
For eg

```
int main() { }
```

Once the next lexeme is determined, forward is set to the character at its right end.



After the lexeme is recorded as an attribute value of a token returned to the parser, lexemeBegin is set to the character immediately after the lexeme just found.



reading a character from hard disk/secondary memory → one system call
entire source program → system calls will be as many as the number of characters in the source program

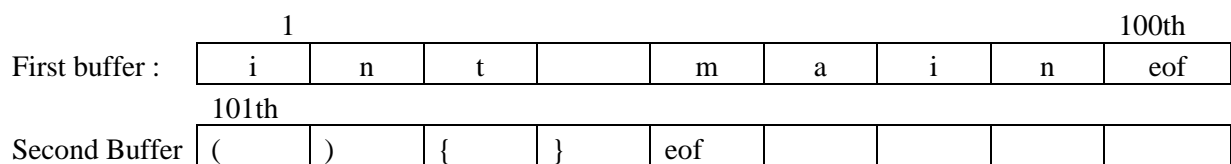
System calls are costly and put burden on the system.

Buffering

Instead of reading one character at a time, a block of characters is read into a buffer at a time using only one system call.

Buffering is implemented using:

- One buffer system: only one buffer is used.
Disadvantage: Overwriting
- Two buffer system: Two buffers are used. If the first buffer is full, then only the second buffer will be used and filled. If fewer than N(size of buffer) character remains in the input, eof is used. eof marks the end of source file or end of input stream and it also specifies the end of the buffer. eof → sentinel character



```

switch ( *forward++ ) {
    case eof:
        if (forward is at end of first buffer ) {
            reload second buffer;
            forward = beginning of second buffer;
        }
        else if (forward is at end of second buffer ) {
            reload first buffer;
            forward = beginning of first buffer;
        }
        else /* eof within a buffer marks the end of input */
            terminate lexical analysis;
        break;
    Cases for the other characters
}

```

=====

Specification of Tokens

- a. **Symbol:** Any character
- b. **Alphabet:** An alphabet is any finite set of symbols. Typical examples of symbols are letters: {a,b,c,d}, digits: {1,2,3,4}, and punctuation: {.,'}.
The set {0,1} is the binary alphabet
ASCII is an important example of an alphabet.
- c. **Strings:** A string over an alphabet is a finite sequence of symbols drawn from that alphabet.
w1 = abcd
w2 = 123
|w| or |s| → length of the string
The empty string, denoted ϵ , is the string of length zero
- d. **Language:** Any countable set of strings over some alphabets.
Denoted by, $L = \{ \dots \}$
for eg.

Empty set, $\{ \epsilon \}$

 $L = \{ a, aa, ab, bb \} \rightarrow a+b^*$

Language with empty string, $L = \{ \epsilon \}$, $\epsilon \rightarrow$ empty string

Set of syntactically well-formed C program

Set of grammatically correct English language

Operations on Strings

1. A **prefix** of string s is any string obtained by removing zero or more symbols from the end of s. For example, ban, banana, and E are prefixes of banana.
2. A **suffix** of string s is any string obtained by removing zero or more symbols from the beginning of s. For example, nana, banana, and E are suffixes of banana.
3. A **substring** of s is obtained by deleting any prefix and any suffix from s. For instance, banana, nan, and E are substrings of banana.
4. The proper prefixes, suffixes, and substrings of a string s are those, prefixes, suffixes, and substrings, respectively, of s that are not E or not equal to s itself.
5. A **subsequence** of s is any string formed by deleting zero or more not necessarily consecutive positions of s. For example, baan is a subsequence of banana
6. **Concatenation:** If x and y are strings, the concatenation of x and y is denoted by 'xy', i.e. the string y is appended after x. For eg.

x = PSIT , y = Kanpur

xy = PSITKanpur

Empty string is identity under concatenation, i.e. for any string s, $\epsilon.s = s.\epsilon = s$

7. **Exponentiation:** if $S^0 = \epsilon$ & for all $i > 0$, define S^i to be $S^{i-1}.S$

$$S^1 = S^0.S = E.S = S$$

$$S^2 = S^1.S = SS$$

$$S^3 = S^2.S = SSS$$

8. **Reverse of a String:** $w = abc$, $w^R = cba$, $w \rightarrow \text{String}$

9. Length of a String: $|s| \rightarrow \text{Length of String}$

Length of empty string is 0

Operations on Languages

OPERATION	DEFINITION AND NOTATION
<i>Union of L and M</i>	$L \cup M = \{s \mid s \text{ is in } L \text{ or } s \text{ is in } M\}$
<i>Concatenation of L and M</i>	$LM = \{st \mid s \text{ is in } L \text{ and } t \text{ is in } M\}$
<i>Kleene closure of L</i>	$L^* = \bigcup_{i=0}^{\infty} L^i$
<i>Positive closure of L</i>	$L^+ = \bigcup_{i=1}^{\infty} L^i$

a. Union:

$$L = \{a,b,c\} \quad M = \{d,e,f\}$$

$$L \cup M = \{a,b,c,d,e,f\}$$

$$L = \{a,b,c\} \quad M = \{a,d,e,f\}$$

$$L \cup M = \{a,b,c,d,e,f\}$$

b. Concatenation:

$$L = \{a, b, ab, ba\} \quad M = \{a, b\}$$

$$L.M = LM = \{aa, ba, aba, baa, ab, bb, abb, bab\}$$

c. Kleene Closure:

$$L^0 = \text{Set of strings of length 0}$$

$$L^1 = \text{Set of strings of length 1}$$

$$L^2 = \text{Set of strings of length 2}$$

$$\text{Kleene Closure} = L^0 \cup L^1 \cup L^2 \cup L^3 \cup L^4 \dots$$

$$L = \{a,b\}$$

$$L^* = \{E, a, b, aa, bb, ab, ba \dots\}$$

d. Positive Closure:

L^1 = Set of strings of length 1

L^2 = Set of strings of length 2

Positive Closure = $L^1 \cup L^2 \cup L^3 \cup L^4 \dots$

$L = \{a,b\}$

$L^+ = \{a,b,aa,bb,ab,ba \dots\}$

Eg.

Let L be the set of letters $\{A, B, \dots, Z, a, b, \dots, z\}$ and let D be the set of digits $\{0,1,\dots,9\}$

- $L \cup D$ is the set of letters and digits
 - LD is the set of 520 strings of length two, each consisting of one letter followed by one digit
 - L^4 is the set of all 4-letter strings.
 - L^* is the set of all strings of letters, including ϵ , the empty string.
 - $L(L \cup D)^*$: identifiers(abc, abc123,...), int
-

Regular Expression

- Ways to represent the regular language.
- It is an expression of string and operators

Operations

- $*$: Kleene Closure
- $^+$: Positive Closure
- $.$: Concatenation
- $+$ or $|$: Union

BASIS: There are two rules that form the basis:

- ϵ is RE, $L(\epsilon) = \{\epsilon\}$
- a is RE, $L = \{a\}$

Induction: There are four parts to the induction whereby larger regular expressions are built from smaller ones. Suppose r and s are regular expressions denoting languages $L(r)$ and $L(s)$, respectively.

- $(r)|(s)$ is a regular expression denoting the language $L(r) \cup L(s)$
- $(r)(s)$ is a regular expression denoting the language $L(r).L(s)$
- $(r)^*$ is a regular expression denoting $(L(r))^*$
- (r) is a regular expression denoting $L(r)$. This last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

Eg.

Define language for given RE:

- $(a|b)$: $L(a) = \{a\}, L(b) = \{b\}; L(a) \cup L(b); L = \{a,b\}$
- $(a+b)(a+b)$: $L = \{aa,ab,bb,ba\}$
- $(a|b)(a|b)(a|b)(a|b)$: $L = \{aaaa, aaab,aabb,\dots\}$
- $(a|b)^*$: $L = \{\epsilon, a,b,aa,bb,ab,ba,\dots\}$
- $(a^*b^*)^*$: $L = \{\epsilon,a,b,aa,aab,\dots\}$
- $a|a^*b$: $L = \{a,b,ab,aab,aaab,\dots\}$

Regular Definitions

- a. It is the name given to a regular expression.
b. regular definition is a sequence of definitions of the form:
- $\mathbf{d_i} \rightarrow \mathbf{r_i}$
- $\text{roll_number} \rightarrow (\text{Year_of_study}).(\text{college_code}).(\text{branch_code}).(\text{serial_number})$
- $\text{phone_number} \rightarrow \text{country_code}.\text{area_code}.\text{unique_number}$
- i.
- $\text{letter_} \rightarrow _ | A | B | C | D \dots | Z | a | b | c | d | \dots | z$
- $\text{digit} \rightarrow 0 | 1 | 2 | 3 | 4 | \dots | 9$
- $\text{id} \rightarrow \text{letter_}(\text{letter_} / \text{digit})^*$
- ii.
- $\text{digit} \rightarrow 0 | 1 | 2 | 3 | 4 | \dots | 9$
- $\text{digits} \rightarrow \text{digit} \text{ digit}^*$
- $\text{optionalFraction} \rightarrow .\text{digits} / \varepsilon$
- $\text{optionalExponent} \rightarrow (E(+/-/\varepsilon)\text{digits}) / \varepsilon$
- $\text{unsignednumber} \rightarrow \text{digits optionalFraction optionalExponent}$**
- iii.
- $\text{sign} \rightarrow + / - / \varepsilon$
- $\text{digit} \rightarrow 0 | 1 | 2 | 3 | 4 | \dots | 9$
- $\text{digits} \rightarrow \text{digit} \text{ digit}^*$
- $\text{optionalFraction} \rightarrow .\text{digits} / \varepsilon$
- $\text{optionalExponent} \rightarrow (E \text{ sign digits}) / \varepsilon$
- $\text{signednumber} \rightarrow \text{sign digits optionalFraction optionalExponent}$**
-
-

Extension of Regular Expression

- a. *One or more instances:* positive closure of RE. If r is RE, then $(r)^+$ denotes the language $(L(r))^+$
 $r^+ = r.r^*$;
 $r^* = r^+ / \varepsilon$
- b. *Zero or one instance:* ?
 $r? = r / \varepsilon$
- c. *Character classes:* A regular expression $a_1/a_2 | \dots | a_n$, where a_i 's are each symbols of the alphabet, can be written as the shorthand $[a_1a_2\dots a_n]$

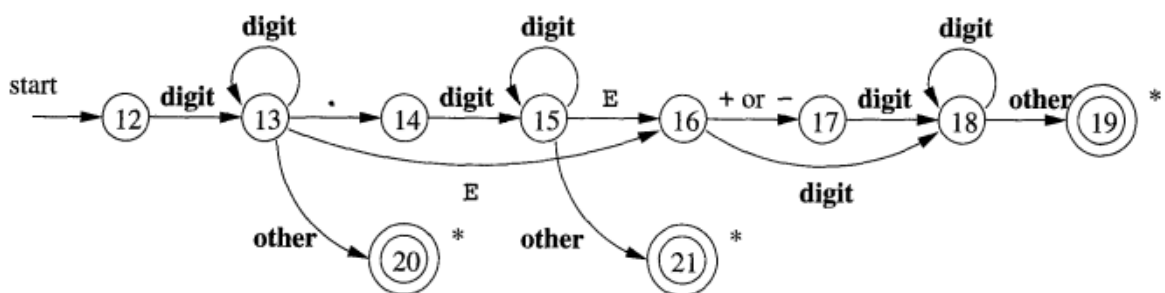
$\text{sign} \rightarrow [+ -]?$

$\text{digit} \rightarrow [0-9]$

$\text{letter_} \rightarrow [A-Za-z_]$

- i.
- $\text{letter_} \rightarrow [_A-Za-z]$
- $\text{digit} \rightarrow [0-9]$
- $\text{id} \rightarrow \text{letter}(\text{letter_} / \text{digit})^*$

- ii.
 - digit* $\rightarrow [0-9]$
 - digits* $\rightarrow \textit{digit}^+$
 - optionalFraction* $\rightarrow \textit{.digits}?$
 - optionalExponent* $\rightarrow (E [+ -]?)\textit{digits}?)$
 - unsignednumber*** $\rightarrow \textit{digits optionalFraction optionalExponent}$



A transition diagram for unsigned numbers

a) $\underline{5200}$:

digits optional fraction optional Exponent

$\epsilon \qquad \qquad \qquad \epsilon$

b) $\underline{0.01234}$:

digits optional fraction optional Exponent

0 . digits ϵ

 . 01234

c) $\underline{6.336E4}$:

digits optional fraction optional Exponent

6 . digits E ϵ digits

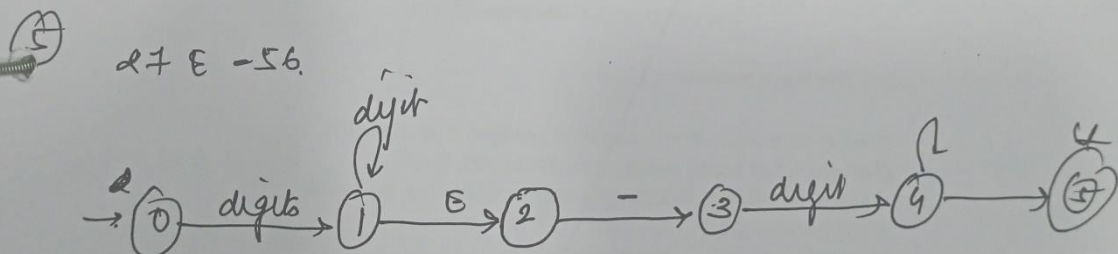
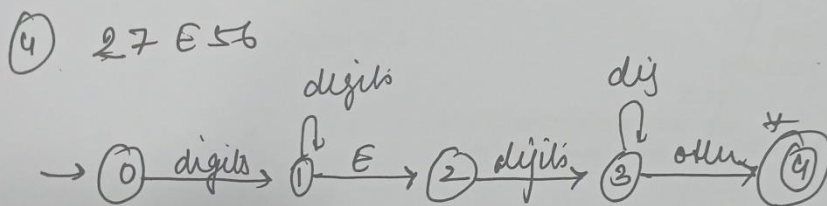
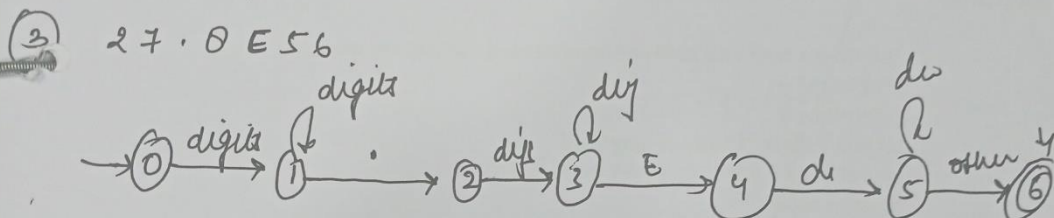
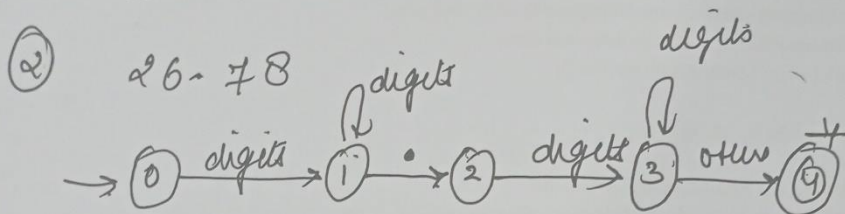
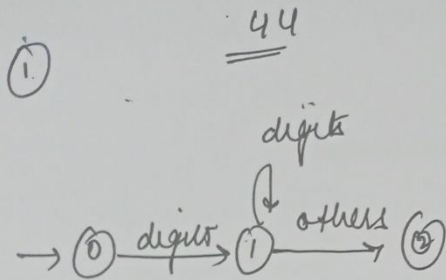
 .. 336 4

d) $\underline{1.09E-4}$

digits optional fraction optional Exponent

1 . digits E - digits

 . 09 0 4.



Recognition of Tokens

Transition diagrams: state, transitions

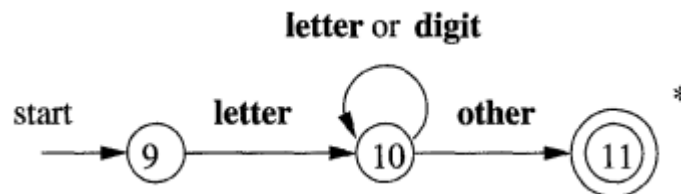
a. Recognition of identifiers:

$letter \rightarrow [_A-Za-z]$

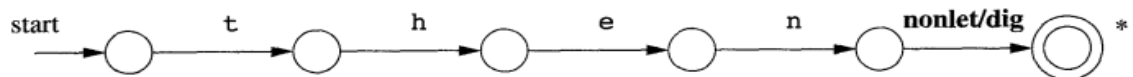
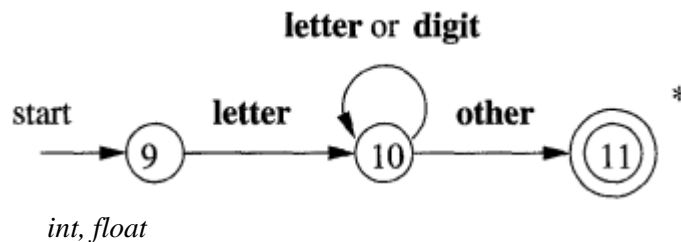
$digit \rightarrow [0-9]$

$id \rightarrow letter (letter | digits)^*$

int no_Of_Books= 9



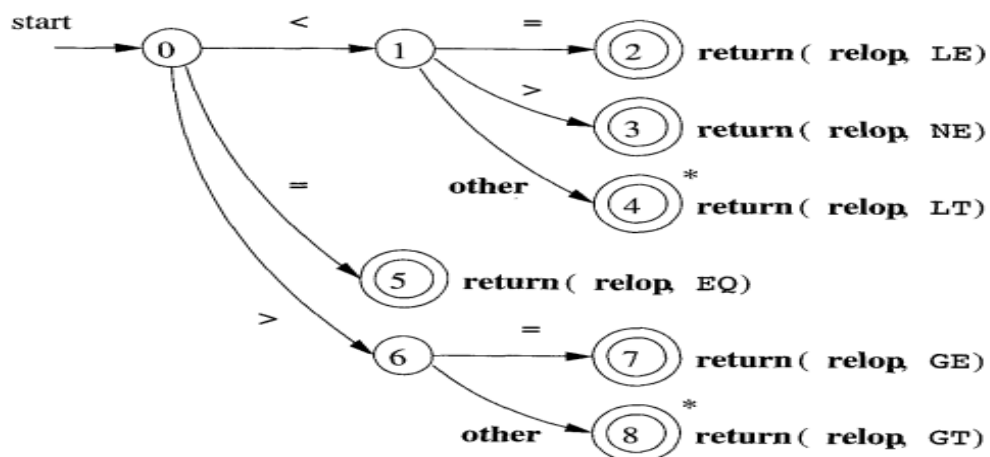
b. Recognition of an keywords/reserved words:



then
else
int

c. Recognition of relational operators:

$\leq, =, \geq, !=, <>, <$



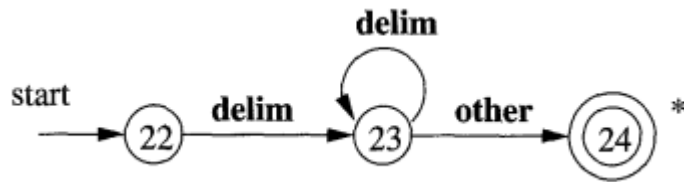
d. **Recognition of Whitespaces:**

$delim \rightarrow blank/tab/newline$

$ws \rightarrow (blank/tab/newline)^+$

$ws \rightarrow delim^+$

$ws \rightarrow delim(delim)^*$



Construction of NFA from Regular expression

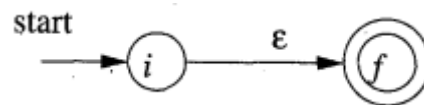
Algorithm: The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA

INPUT: A regular expression r over alphabet Σ .

OUTPUT: An NFA N accepting $L(r)$.

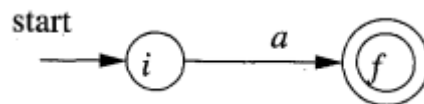
BASIS: For regular expression ϵ , construct the NFA

$r = \epsilon; L(r) = \{ \epsilon \}$



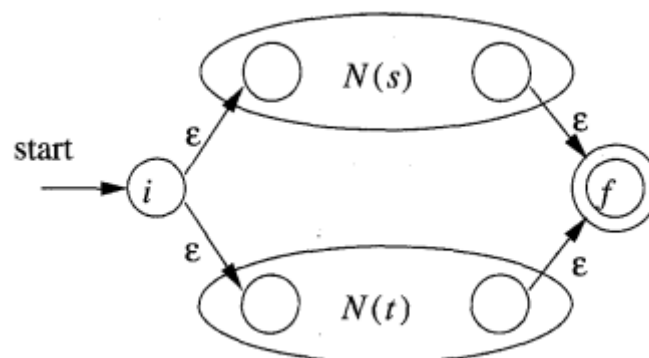
For any regular expression a in Σ , construct the NFA

$r = a; L(r) = \{ a \}$

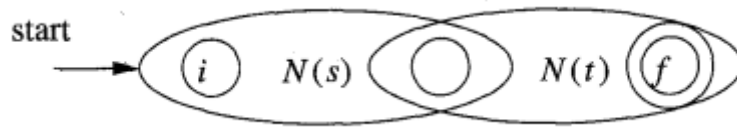


INDUCTION: Suppose $N(s)$ and $N(t)$ are NFA's for regular expressions s and t , respectively.

a. Suppose $r = s / t$



b. Suppose $r = st$



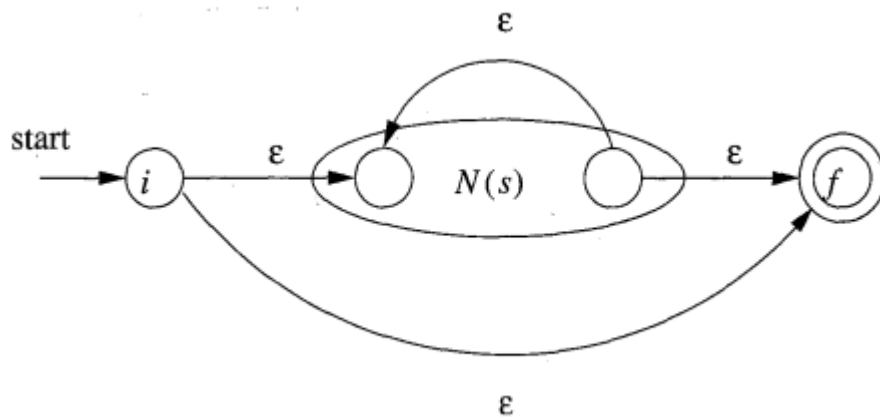
c. Suppose $r = s^*$

$$\epsilon = \epsilon$$

$$X = X$$

$$X. \epsilon. X = XX$$

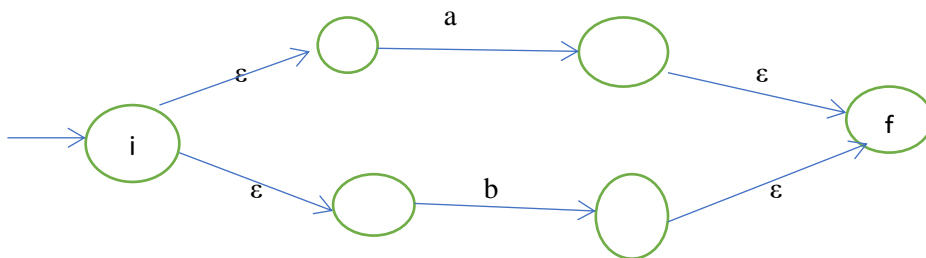
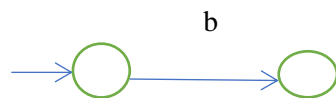
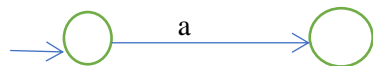
$$X. \epsilon. X. \epsilon. X = XXX$$



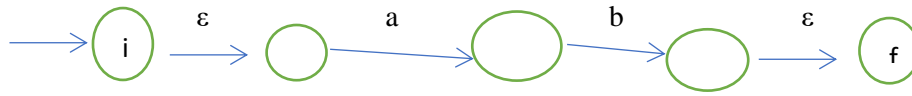
$$r = s^*; L(r) = \{ \epsilon, s, ss, sss, \dots \}$$

For eg.

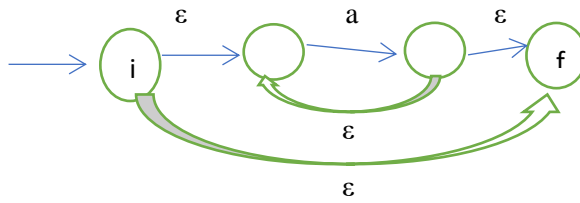
i. $(a|b); L(r) = \{a, b\}$



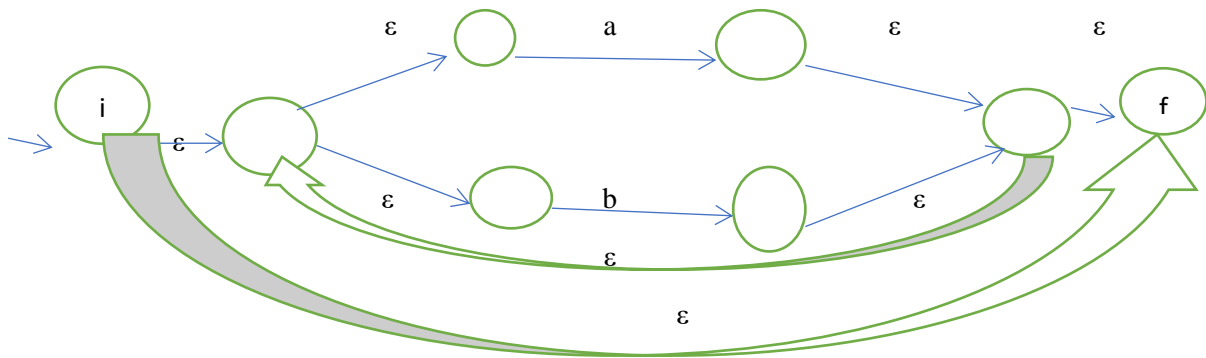
ii. ab



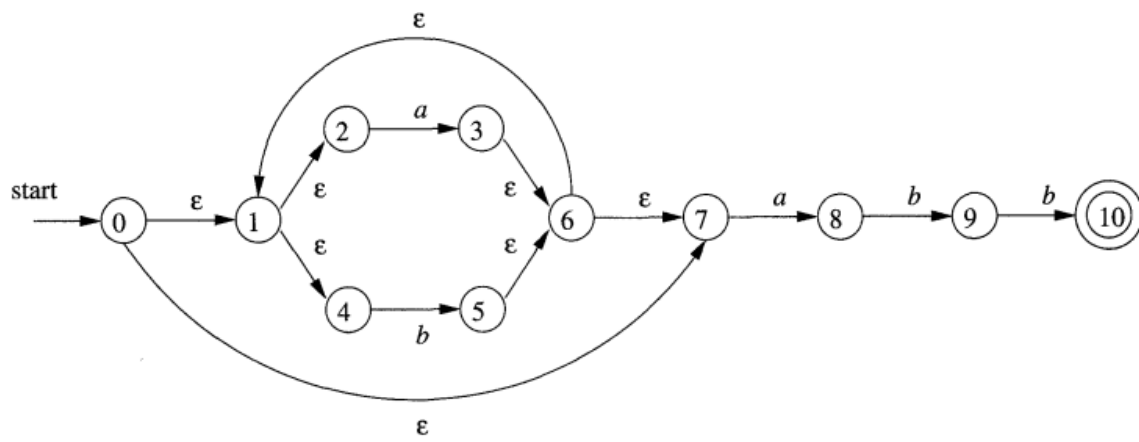
iii. $a^* : L(r) = \{ \epsilon, a, aa, \dots \}$



iv. $(a|b)^*$; $L(r) = \{ \epsilon, a, b, aa, ab, bb, ba, \dots \}$



v. $(a|b)^*abb$

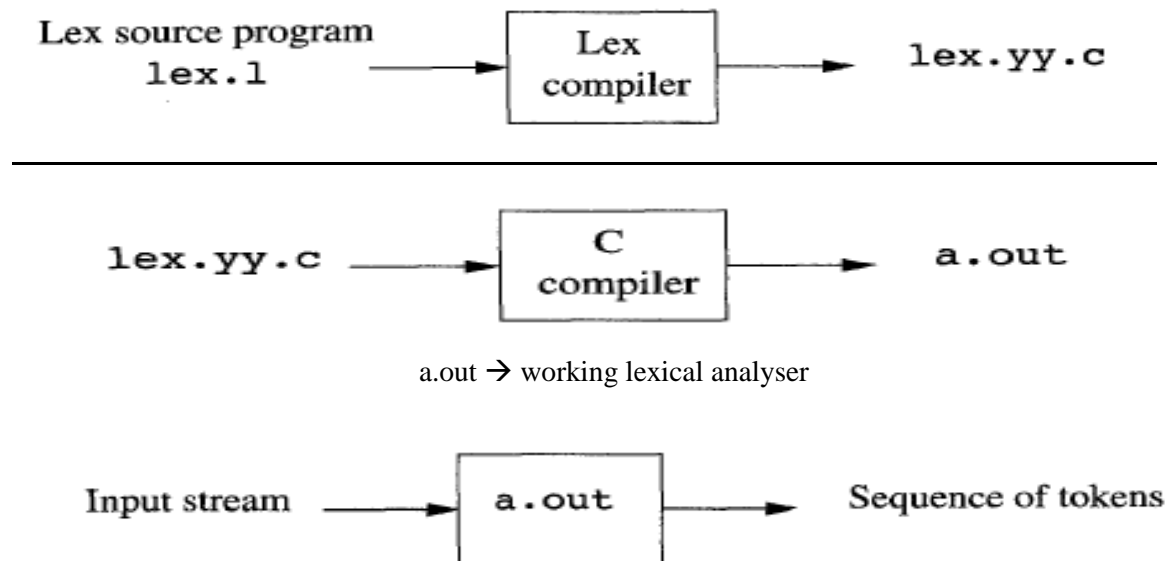


vi. $(a|b)^*(abb)^*$



The Lexical-Analyzer Generator (Lex)

Lex Source Program: C code + regular definitions/regular expressions and the actions to be taken based on REs.



Structure of Lex Source Program:

- a. Declaration
- b. Translation rules
- c. Auxiliary Functions
- a. **Declarations:** includes:
 - i. C variables, constants
 - ii. C header file Inclusion
 - iii. Regular definitions
 - iv. Written as

```
%{  
    C language source program  
%} regular definitions
```

Eg.,

```
%{  
    #include<stdio.h>  
    int a,b;  
    const float count = 0;  
%}  
Letter  [A-Za-z]  
Digit   [0-9]
```

Note: Any code written within %{ %} is directly copied into the lex.yy.c file and it is not treated as regular definition.

- b. **Translation Rules:** It has the following form:

```
%%  
Pattern {action}  
Pattern1 {action1}  
Pattern2 {action2}  
Pattern3 {actions3}  
%%
```

Pattern: regular expression/set of rules

Action: C language statement

Eg. [0-9] { printf("Digits found); }

- c. **Auxiliary Functions:** additional functions and are compiled separately and loaded with lexical analyzer.

```
% { % }  
%% %%  
Auxiliary functions
```

Q1. Write a lex program for recognition of lowercase and uppercase letters in given word.

```
% {  
#include<stdio.h>  
% }  
  
%%  
[A-Z] {printf("Uppercase letters found");}  
[a-z] {printf("Lowercase letters found");}  
%%  
int main(){  
    printf("Enter the String");  
    yylex(); // takes the input from the console and converts strings to token or a  
             //sequence of tokens  
}
```


Q2. Write a lex program to count vowels and consonants.

```
% {
    #include<stdio.h>
    int v=0, c=0;
    % }
    vow [aeiouAEIOU]
    con [a-zA-Z^a|e|i|o|u|A|E|I|O|U]

    %%

    {vow} {v++;}

    {con} {c++;}

    %%

    int main(){
        printf("Enter the String");

        yylex();

        printf("%dTotal Vowels: ",v);

        printf("%dTotal Consonants: ",c);

    }
```

Converting ϵ -NFA to DFA Conversion

A Deterministic Finite Automaton (DFA) has at most one edge from each state for a given symbol and is a suitable basis for a transition table. We need to eliminate the ϵ -transitions by subset construction.

Definitions

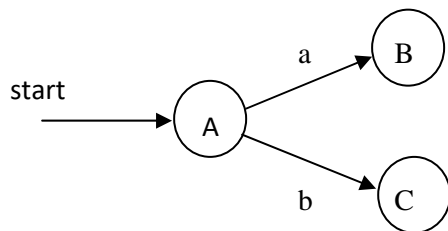
Consider a single state s . Consider a set of states T

Operation	Description
ϵ -closure(s)	Set of NFA states reachable from NFA state s on ϵ -transitions alone
ϵ -closure(T)	Set of NFA states reachable from set of states T on ϵ -transitions alone
move(T,a)	Set of states to which there is a transition on input symbol a from some NFA state in T

We have as input the set of N states. We generate as output a set of D states in a DFA. Theoretically an NFA with n states can generate a DFA with 2^n states.

Start the Conversion

1. Begin with the start state 0 and calculate ϵ -closure(0).
 - a. the set of states reachable by ϵ -transitions which includes 0 itself is $\{0,1,2,4,7\}$. This defines a new state A in the DFA $A = \{0,1,2,4,7\}$
2. We must now find the states that A connects to. There are two symbols in the language (**a**, **b**) so in the DFA we expect only two edges: from A on **a** and from A on **b**. Call these states B and C:



We find B and C in the following way:

Find the state B that has an edge on a from A

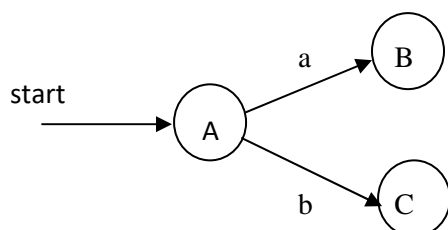
- a. start with $A\{0,1,2,4,7\}$. Find which states in A have states reachable by **a** transitions. This set is called $\text{move}(A,a)$ The set is $\{3,8\}$:
 $\text{move}(A,a) = \{3,8\}$
- b. now do an ϵ -closure on $\text{move}(A,a)$. Find all the states in $\text{move}(A,a)$ which are reachable with ϵ -transitions. We have 3 and 8 to consider. Starting with 3 we can get to 3 and 6 and from 6 to 1 and 7, and from 1 to 2 and 4. Starting with 8 we can get to 8 only. So the complete set is $\{1,2,3,4,6,7,8\}$. So ϵ -closure($\text{move}(A,a)$) = $B = \{1,2,3,4,6,7,8\}$

This defines the new state B that has an edge on **a** from A

Find the state C that has an edge on b from A

- c. start with $A\{0,1,2,4,7\}$. Find which states in A have states reachable by **b** transitions. This set is called $\text{move}(A,b)$ The set is $\{5\}$:
 $\text{move}(A,b) = \{5\}$
- d. now do an ϵ -closure on $\text{move}(A,b)$. Find all the states in $\text{move}(A,b)$ which are reachable with ϵ -transitions. We have only state 5 to consider. From 5 we can get to 5, 6, 7, 1, 2, 4. So the complete set is $\{1,2,4,5,6,7\}$. So ϵ -closure($\text{move}(A,b)$) = $C = \{1,2,4,5,6,7\}$

This defines the new state C that has an edge on **b** from A



$A = \{0,1,2,4,7\}$

$B = \{1,2,3,4,6,7,8\}$

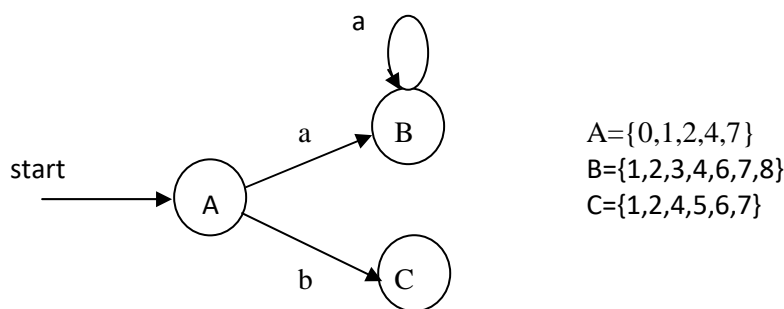
$C = \{1,2,4,5,6,7\}$

Now that we have B and C we can move on to find the states that have **a** and **b** transitions from B and C.

Find the state that has an edge on a from B

- e. start with B{1,2,3,4,6,7,8}. Find which states in B have states reachable by **a** transitions. This set is called $\text{move}(B,a)$ The set is {3,8}: $\text{move}(B,a) = \{3,8\}$
- f. now do an ϵ -closure on $\text{move}(B,a)$. Find all the states in $\text{move}(B,a)$ which are reachable with ϵ -transitions. We have 3 and 8 to consider. Starting with 3 we can get to 3 and 6 and from 6 to 1 and 7, and from 1 to 2 and 4. Starting with 8 we can get to 8 only. So the complete set is {1,2,3,4,6,7,8}. So $\epsilon\text{-closure}(\text{move}(A,a)) = \{1,2,3,4,6,7,8\}$

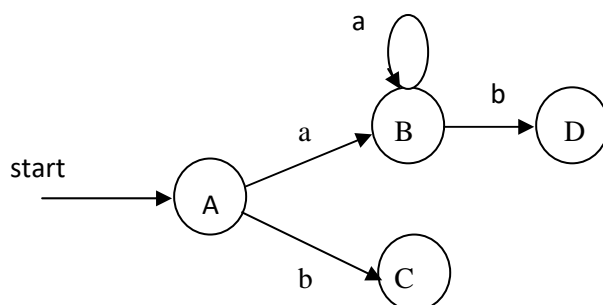
which is the same as the state B itself. In other words, we have a repeating edge to B:



Find the state D that has an edge on b from B

- g. start with B{1,2,3,4,6,7,8}. Find which states in B have states reachable by **b** transitions. This set is called $\text{move}(B,b)$ The set is {5,9}: $\text{move}(B,b) = \{5,9\}$
- h. now do an ϵ -closure on $\text{move}(B,b)$. Find all the states in $\text{move}(B,b)$ which are reachable with ϵ -transitions. From 5 we can get to 5, 6, 7, 1, 2, 4. From 9 we get to 9 itself. So the complete set is {1,2,4,5,6,7,9}. So $\epsilon\text{-closure}(\text{move}(B,a)) = D = \{1,2,4,5,6,7,9\}$
This defines the new state D that has an edge on **b** from B

$A=\{0,1,2,4,7\}$, $B=\{1,2,3,4,6,7,8\}$, $C=\{1,2,4,5,6,7\}$, $D=\{1,2,4,5,6,7,9\}$

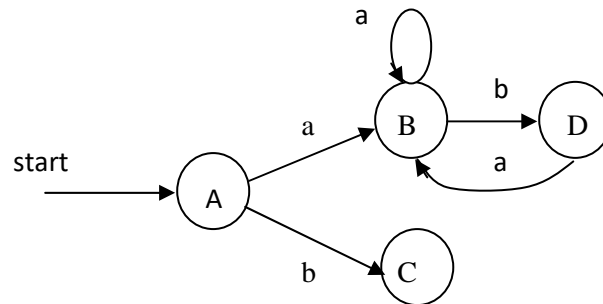


Find the state that has an edge on a from D

- i. start with D{1,2,4,5,6,7,9}. Find which states in D have states reachable by **a** transitions. This set is called $\text{move}(D,a)$ The set is {3,8}: $\text{move}(D,a) = \{3,8\}$

- j. now do an ϵ -closure on **move(D,a)**. Find all the states in **move(B,a)** which are reachable with ϵ -transitions. We have 3 and 8 to consider. Starting with 3 we can get to 3 and 6 and from 6 to 1 and 7, and from 1 to 2 and 4. Starting with 8 we can get to 8 only. So the complete set is $\{1,2,3,4,6,7,8\}$.
 So ϵ -closure(move(D,a)) = **$\{1,2,3,4,6,7,8\} = B$**
 This is a return edge to B:

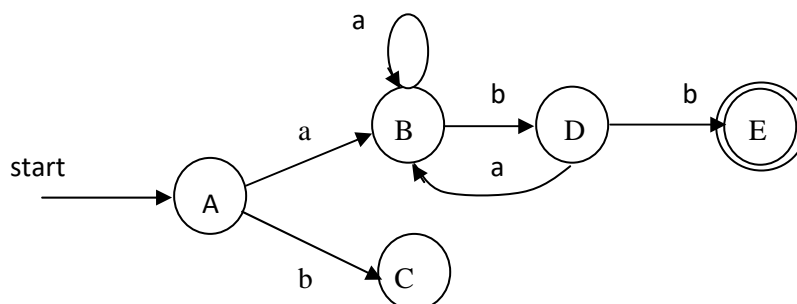
$A=\{0,1,2,4,7\}$, $B=\{1,2,3,4,6,7,8\}$, $C=\{1,2,4,5,6,7\}$, $D=\{1,2,4,5,6,7,9\}$



Find the state E that has an edge on b from D

- k. start with $D=\{1,2,4,5,6,7,9\}$. Find which states in D have states reachable by **b** transitions. This set is called **move(B,b)**. The set is $\{5,10\}$: **move(D,b) = $\{5,10\}$**
 l. now do an ϵ -closure on **move(D,b)**. Find all the states in **move(D,b)** which are reachable with ϵ -transitions. From 5 we can get to 5, 6, 7, 1, 2, 4. From 10 we get to 10 itself. So the complete set is $\{1,2,4,5,6,7,10\}$. So
 ϵ -closure(move(D,b)) = **$E = \{1,2,4,5,6,7,10\}$** This defines
 the new state E that has an edge on **b** from D. **Since it contains an accepting state, it is also an accepting state.**

$A=\{0,1,2,4,7\}$, $B=\{1,2,3,4,6,7,8\}$, $C=\{1,2,4,5,6,7\}$, $D=\{1,2,4,5,6,7,9\}$,
 $E=\{1,2,4,5,6,7,10\}$



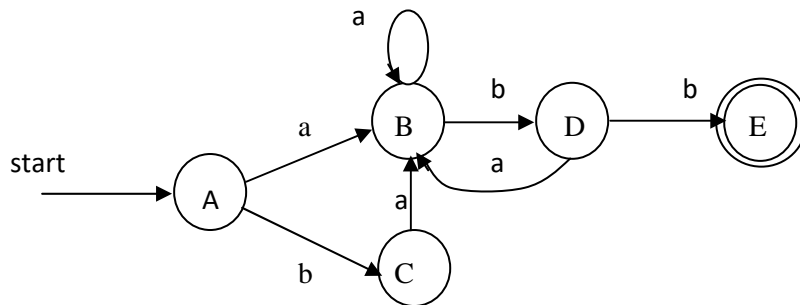
We should now examine state C

Find the state that has an edge on a from C

- m. start with $C\{1,2,4,5,6,7\}$. Find which states in C have states reachable by **a** transition. This set is called $\text{move}(C,a)$ The set is $\{3,8\}$:
 $\text{move}(C,a) = \{3,8\}$

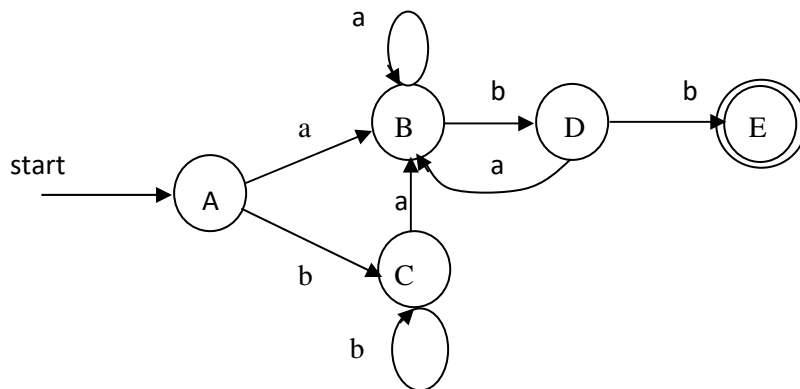
we have seen this before. It's the state B

$$A=\{0,1,2,4,7\}, B=\{1,2,3,4,6,7,8\}, C=\{1,2,4,5,6,7\}, D=\{1,2,4,5,6,7,9\}, E=\{1,2,4,5,6,7,10\}$$



Find the state that has an edge on b from C

- n. start with $C\{1,2,4,5,6,7\}$. Find which states in C have states reachable by **b** transitions. This set is called $\text{move}(C,b)$ The set is $\{5\}$: $\text{move}(C,b) = \{5\}$
 o. now do an ϵ -closure on $\text{move}(C,b)$. Find all the states in $\text{move}(C,b)$ which are reachable with ϵ -transitions. From 5 we can get to 5,6,7,1,2,4. which is C itself So $\epsilon\text{-closure}(\text{move}(C,b)) = C$
 This defines a loop on C



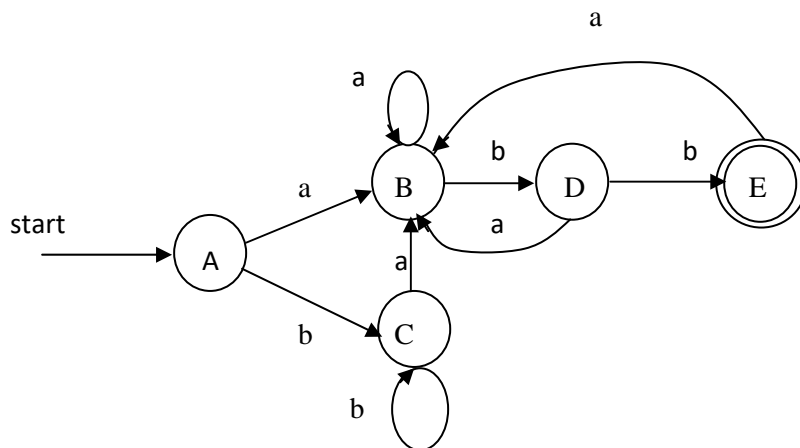
Finally, we need to look at E. Although this is an accepting state, the regular expression allows us to repeat adding in more a's and b's as long as we return to the accepting E state finally. So

Find the state that has an edge on a from E

- p. start with $E\{1,2,4,5,6,7,10\}$. Find which states in E have states reachable by **a** transitions. This set is called $\text{move}(E,a)$ The set is $\{3,8\}$:

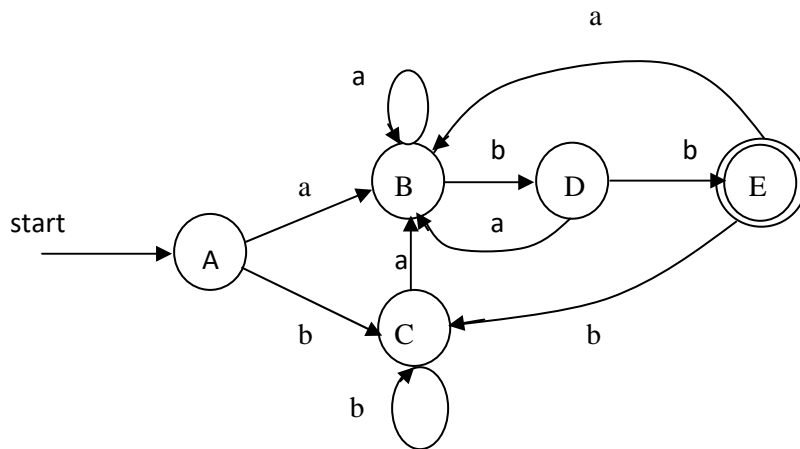
$$\text{move}(E,a) = \{3,8\}$$

We saw this before, it's B. So



Find the state that has an edge on b from E

q. start with $E\{1,2,4,5,6,7,10\}$. Find which states in E have states reachable by **b** transitions. This set is called $\text{move}(E,b)$. The set is $\{5\}$: $\text{move}(A,b) = \{5\}$ We've seen this before. It's C. Finally



That's it ! There is only one edge from each state for a given input character. It's a DFA. Disregard the fact that each of these states is actually a group of NFA states. We can regard them as single states in the DFA. In fact it also requires **other** as an edge beyond E leading to the ultimate accepting state. Also the DFA is not yet optimized (there can be less states).

However, we can make the transition table so far. Here it is:

State	Input a	Input b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

Minimization of DFA (Partitioning Method)

Suppose p, q are two different states and are said to be equivalent iff,

- $\text{move}(p, w) = s$ and s belongs to F and $\text{move}(q, w) = t$ and t also belongs to F , where F is a set of final states
- $\text{move}(p, w) = a$ and a does not belong to F and $\text{move}(q, w) = b$ and b also does not belong to F , where F is a set of final states.

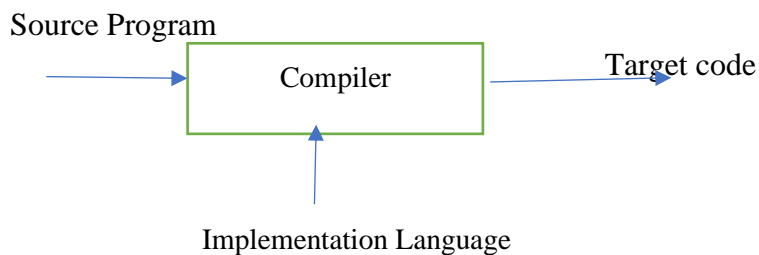
Suppose p, q are two different states and are said to be non-equivalent or distinguishable iff,

- $\text{move}(p, w) = n$ and n belongs to F and $\text{move}(q, w) = m$ and m does not belong to F , where F is a set of final states

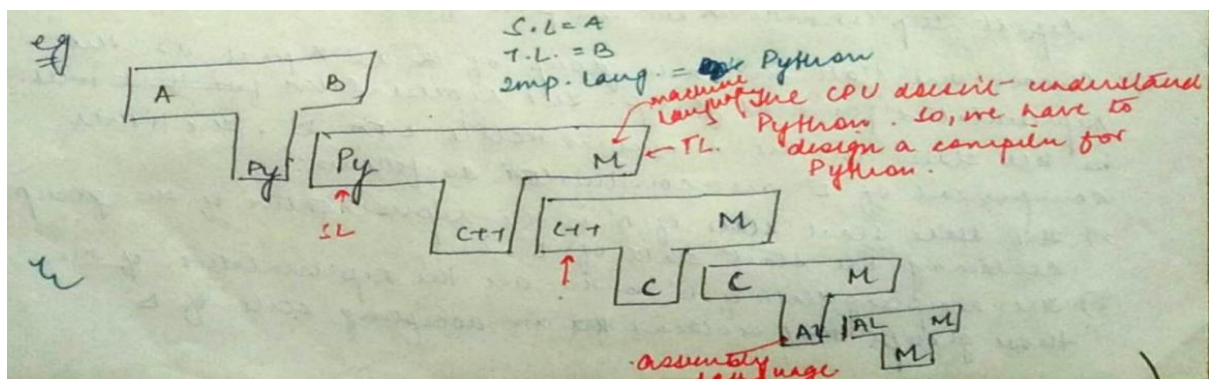
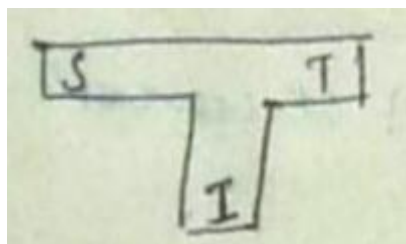
Examples in separate PDF file

Bootstrapping

- It is a process for designing the compiler by which simple language is used to translate more complicated program which may then handle more complicated programs.
- Any compiler can be represented by three languages:
 - Source language
 - Target language
 - Implementation language

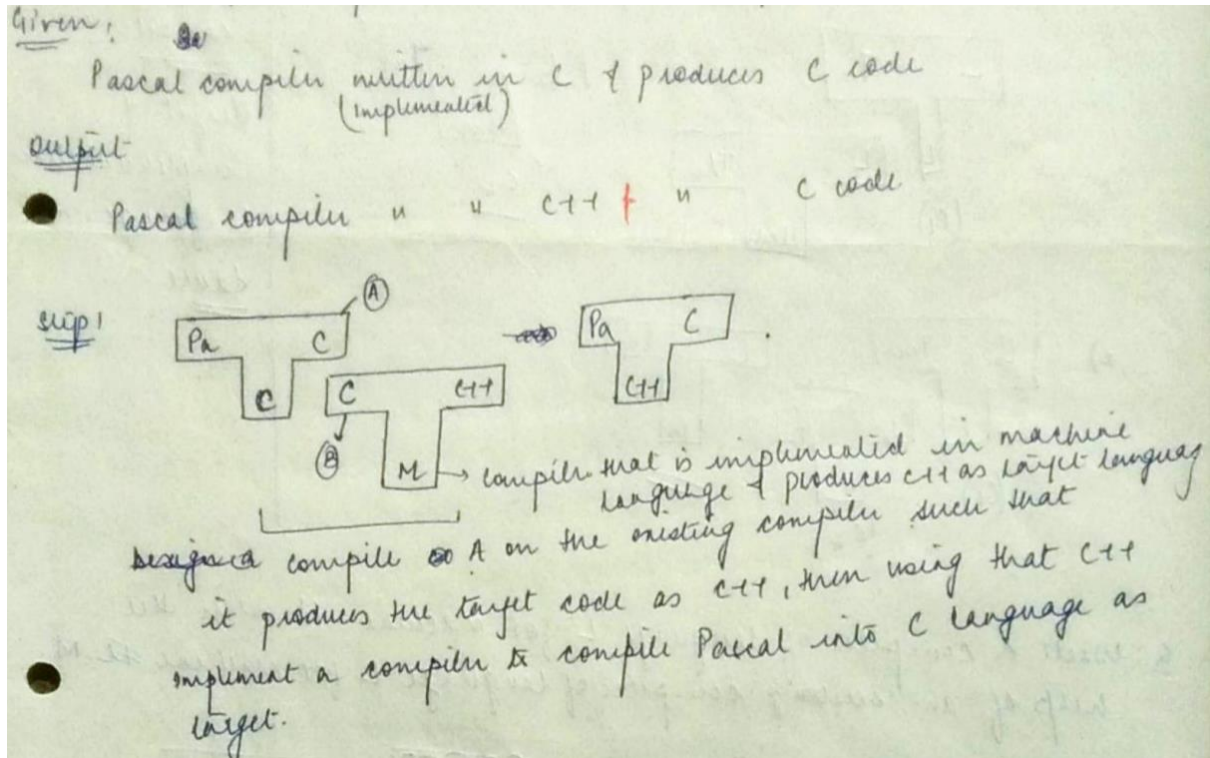


Compilation process can be represented using T-diagram:



Types of Compilers

- **Self-hosted compilers:** Source language and implementation language are same. Eg. C language
- **Native compilers:** target language and implementation language are same.
- **Cross compilers:** Run on one machine and produces target code of another machine.
- **Transcompiler:** A transcompiler, also called a source-to-source compiler or transpiler, is a special type of compiler that converts a program's source code into another language. It can also process a program written in an older version of a programming language, converting it to a newer version of the same language.



Chomsky's Classification of Grammars

$$G = (V, T, P, S)$$

V = Set of variables or non-terminal

T = Set of Terminals

P = Set of production

S = Start symbol

Type 3 Grammar

- a. Regular grammar.
- b. Generates regular language.
- c. Accepted by finite automata.
- d. It can be either right linear or left linear but not both.

$$A \rightarrow \alpha B \mid \beta$$

$$A, B \in V$$

$$\alpha, \beta \in T$$

$$A \rightarrow \alpha B \mid \beta \quad [\text{Right linear}]$$

$$|\alpha| = |\beta| = 1$$

$$A \rightarrow B\alpha \mid \beta \quad [\text{Left Linear}]$$

Eg.

- i. $A \rightarrow aB \mid a$
 $B \rightarrow aB \mid bB \mid a \mid b$
Right Linear

- ii. $A \rightarrow Ba \mid a$
 $B \rightarrow Ba$
 $B \rightarrow bB \mid a \mid b$
Not a regular grammar because it is left as well as right linear.

Type 2 Grammar:

- a. Context Free Grammar
- b. Generates CFL
- c. Accepted by PDA
- d. $A \rightarrow \alpha, |\alpha| = 1$ and $A \in V, \alpha \in (V \cup T)^*$

Eg.

$$A \rightarrow aAb \mid ab \mid \varepsilon$$

$$A \Rightarrow aabb$$

$$A \Rightarrow ab$$

- e. CFG can be:
 - i. Ambiguous & Non-Ambiguous
 - ii. Left recursive & Right recursive
 - iii. Deterministic & Non-deterministic

Type 1 Grammar:

- a. Context-Sensitive Grammar
- b. Generates CSL
- c. Accepted by Linear-Bound Automata
- d. $\alpha \rightarrow \beta, \alpha \in (VUT)^*V(VUT)^*, \beta \in (VUT)^+$
- e. RHS cannot have ε
- f. $|\alpha| \leq |\beta|$
e.g. $A \rightarrow \varepsilon$

Type 0 Grammar:

- a. Unrestricted Grammar/Recursively Enumerable Grammar
- b. Generates REL
- c. Accepted by Turing machine
- d. $\alpha \rightarrow \beta, \alpha \in (VUT)^*V(VUT)^*, \beta \in (VUT)^*$
- e. RHS can have ε

Context-Free Grammar

1. **Ambiguous & Non-Ambiguous:** For a given string, if there is left-derivation tree as well as right-derivation tree then the grammar is ambiguous else non-ambiguous.

Or

For a given string, if there is left most derivation as well as right most derivation possible then the grammar is ambiguous else non-ambiguous.

$$E \rightarrow E+E \mid E * E \mid \text{id}, w = \text{id} + \text{id} * \text{id}$$
Left Most Derivation

$$E \Rightarrow E + E$$

$$E \Rightarrow \text{id} + E$$

$$E \Rightarrow \text{id} + E * E$$

$$E \Rightarrow \text{id} + \text{id} * E$$

$$E \Rightarrow \text{id} + \text{id} * \text{id}$$
Right Most Derivation

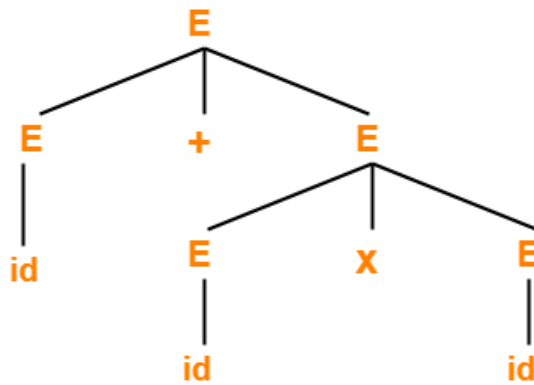
$$E \Rightarrow E + E$$

$$E \Rightarrow E + E * E$$

$$E \Rightarrow E + E * \text{id}$$

$$E \Rightarrow E + \text{id} * \text{id}$$

$$E \Rightarrow \text{id} + \text{id} * \text{id}$$



Parse Tree-01

id + id x id
 $2 + (3 \times 4) = 14$
 20

id + id + id
 $2 + (3 + 4)$

id + id - id
 $2 + (3 - 4)$

id + id x id → Precedence not taken under consideration
 id + id + id → Associativity not taken under consideration

Note:

- i. To construct a parser, we should first check if the grammar is ambiguous or not.
- ii. If the grammar is ambiguous, then the parser gets confused out of the two, which derivation tree should be used. So, we need to remove the ambiguity.
- iii. There is no algorithm for finding whether the grammar is ambiguous nor to convert the ambiguous grammar to non-ambiguous grammar. The only way is to **use hit and trial approach**.

Q. Find whether the following grammar is ambiguous or not.

- i. $S \rightarrow aS|Sa|a$, $w = aa$
- ii. $S \rightarrow aSbS|bSaS|\epsilon$, $w = abab$
- iii. $R \rightarrow R+R|RR|R^*|a|b|c$, $w = a + bc$

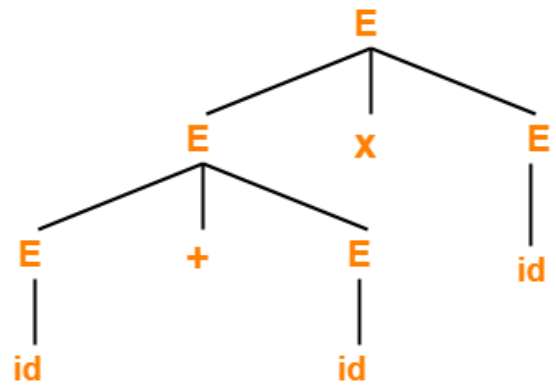
Disambiguity rules:

Disambiguity rules are required because precedence and associativity of the operator is to be maintained.

$E \rightarrow E+E$
 $E \rightarrow E^*E$
 $E \rightarrow id$

To make the grammar unambiguous, we must take care of the associativity and precedence.

$E \rightarrow E + id$
 $E \rightarrow id$



Parse Tree-02

id + id x id
 $(2 + 3) \times 4 =$

id + id + id
 $(2 + 3) + 4$

id + id - id
 $(2 + 3) - 4$

$w = id + id + id + id$

$((id + id) + id) + id$

For left associativity, the left most symbol of the RHS must be same as the LHS and the grammar is left recursive.

$E \rightarrow E + id$

$E \rightarrow id$

$E \rightarrow id + E$

$E \rightarrow id$

$w = id + id + id + id$

$id + ((id + id) + id)$

For right associativity, the right-most symbol of the RHS must be same as the LHS and the grammar is right recursive.

$E \rightarrow id + E$

$E \rightarrow id$

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow id$

$w = id + id * id \text{ or } id * id + id$

$E \Rightarrow E + T$

$E \Rightarrow T + T$

$E \Rightarrow F + T$

$E \Rightarrow id + T$

$E \Rightarrow id + T * F$

$E \Rightarrow id + F * F$

$E \Rightarrow id + id * F$

$E \Rightarrow id + id * id$

The highest precedence operator must be farthest from the start symbol and the operator must be at last level

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow G ** F \mid G$

$G \rightarrow id$

$E \rightarrow T + E \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow F ** G \mid G$

$G \rightarrow id$

2. Left-recursive & Right-recursive Grammar

If the grammar is left recursive, it takes the form $A \rightarrow A\alpha \mid \beta$

If the grammar is right recursive, it takes the form $A \rightarrow \alpha A \mid \beta$

Problem with left recursive grammar,

$$\begin{aligned}
 A &\rightarrow A\alpha \mid \beta \\
 A &\Rightarrow A\alpha \\
 A &\Rightarrow A\alpha\alpha \\
 A &\Rightarrow A\alpha\alpha\alpha \\
 &\vdots \\
 A &\Rightarrow \beta\alpha^*
 \end{aligned}$$

$A() \{$
 $\quad \beta;$
 $\quad \alpha \rightarrow \text{Infinite Loop}$
 $\}$

$$\begin{aligned}
 A &\rightarrow \alpha A \mid \beta \\
 A &\Rightarrow \alpha^* \beta
 \end{aligned}$$

Removing Left Recursion

Top-down parsers do not work on the grammar that is left recursive. It is important to remove left recursion without changing the grammar.

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ -productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G . Note that the resulting non-left-recursive grammar may have ϵ -productions. \square

- 1) arrange the nonterminals in some order A_1, A_2, \dots, A_n .
- 2) **for** (each i from 1 to n) {
- 3) **for** (each j from 1 to $i - 1$) {
- 4) replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \dots \mid \delta_k\gamma$, where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \dots \mid \delta_k$ are all current A_j -productions
- 5) }
- 6) eliminate the immediate left recursion among the A_i -productions
- 7) }

$$\begin{aligned}
 A &\rightarrow A\alpha \mid \beta \\
 A &\rightarrow \beta A' \\
 A' &\rightarrow \epsilon \mid \alpha A'
 \end{aligned}$$

$$Q1. E \rightarrow E + T \mid T$$

$$A: E$$

$$\alpha: +T$$

$$\beta: T$$

$$E \rightarrow TE'$$

$$E' \rightarrow \varepsilon \mid +TE'$$

$$Q2. S \rightarrow S0S1S$$

$$S \rightarrow 01$$

$$S \rightarrow 01S'$$

$$S' \rightarrow \varepsilon \mid 0S1SS'$$

$$Q3. S \rightarrow (L) \mid S$$

$$L \rightarrow L,S \mid S$$

$$Q4. S \rightarrow AaB$$

$$A \rightarrow Aa \mid bB \mid a$$

$$B \rightarrow bB \mid c$$

$$Q5. S \rightarrow Aa \mid b$$

$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

$$Q6. S \rightarrow AaB$$

$$A \rightarrow SAc \mid a$$

$$B \rightarrow Ba \mid b$$

3. Deterministic and Non-Deterministic Grammar

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3 \dots$$

$$w = \alpha\beta_3$$

When α is encountered, the compiler or the parser, may think that $\alpha\beta_1$ is the right production. But, when β_1 is encountered, it realises that $\alpha\beta_1$ not the correct production. Hence, it backtracks. Same is the case with $\alpha\beta_2$. The main problem is **backtracking**. This is due to the **common prefixes**. Hence, it is also called **common prefixes problem**. That is, the decision of deriving $\alpha\beta_3$ must be taken by β_3 . This problem is resolved by **left factoring**, i.e., postponing the decision-making process until β is encountered.

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$$

$$A \rightarrow \alpha A'$$

$$A' \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$$

$$Q1. S \rightarrow \mathbf{iEtS} \mid \mathbf{iEtSeS} \mid a$$

$$E \rightarrow b$$

$$S \rightarrow \mathbf{iEtS'} \mid a$$

$$S' \rightarrow S \mid SeS$$

$$E \rightarrow b$$

Q2. $S \rightarrow \mathbf{aSSbS} \mid \mathbf{aSaSb} \mid abb \mid b$

Taking aS as common prefix:

$S \rightarrow aSS' \mid abb \mid b$

$S' \rightarrow SbS \mid aSb$

Taking a as common prefix

$S \rightarrow \mathbf{aSSbS} \mid \mathbf{aSaSb} \mid abb \mid b$

$S \rightarrow aS' \mid b$

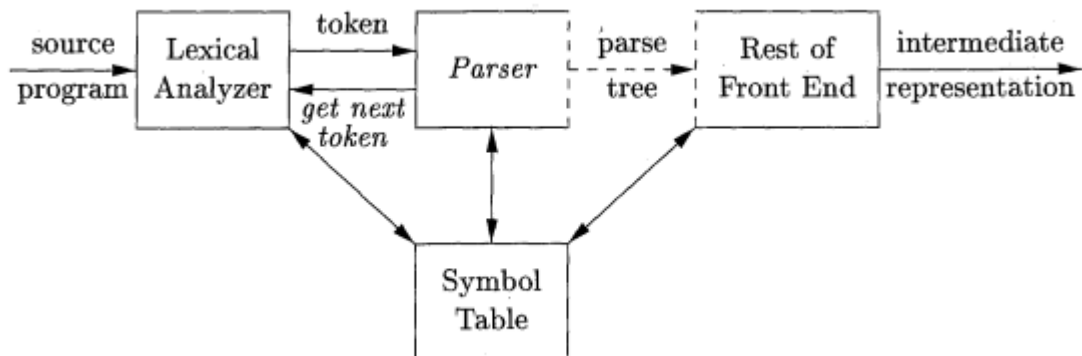
$S' \rightarrow \mathbf{SSbS} \mid \mathbf{SaSb} \mid bb$

$S' \rightarrow SS'' \mid bb$

$S'' \rightarrow SbS \mid aSb$

Syntax Analysis

Role of the Parser



- the parser obtains a string of tokens from the lexical analyzer.
- verifies that the string of token names can be generated by the grammar for the source language.

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow \text{id}$

$w = \text{id} + \text{id} * \text{id} \text{ or } \text{id} * \text{id} + \text{id} \text{ or } * \text{id} \text{id} ++$

- report any syntax errors in an intelligible fashion
- recover from commonly occurring errors to continue processing the remainder of the program.

Parse Tree

Pictorial representation of how a **start symbol** of a grammar derives a string in the language.

$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \text{id}$

$w = -(\text{id} + \text{id})$

Derivation

$E \Rightarrow -E$

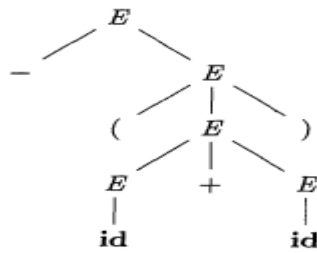
$E \Rightarrow -(E)$

$E \Rightarrow -(E+E)$

$E \Rightarrow -(\text{id} + E)$

$E \Rightarrow -(\text{id} + \text{id})$

Parse Tree



Properties of a Parse Tree

- i. Root is always labelled with the start symbol
- ii. Each leaf node is labelled with a terminal or token
- iii. Each interior node must be labelled with non-terminals

Yield of the Parse Tree

The leaves of a parse tree when they are read from left to right form the yield.

Types of Parsers

Universal Parser: can parse any grammar. They are not used because they are very efficient.

Algorithms used are

Cocke-Younger-Kasami (CYK) algorithm and Earley's algorithm

Top-down Parser: Build a parse tree from top(root) to the bottom(leave)

Bottom-up Parser: build the parse tree by starting from the bottom, i.e., leave and ends on top, i.e., root.

Top-down parser:

- a. *Top-down parser with full backtracking:* Brute force method, i.e., it checks all the combination
- b. *Top-down parser without backtracking:* do not support non-deterministic grammar and left-recursive grammar
 - a. Recursive descent parser
 - b. Non-recursive Descent Parser/Predictive Parser: LL(1)

Bottom-up Parsers

a. Operator precedence parsers: allows ambiguous grammar

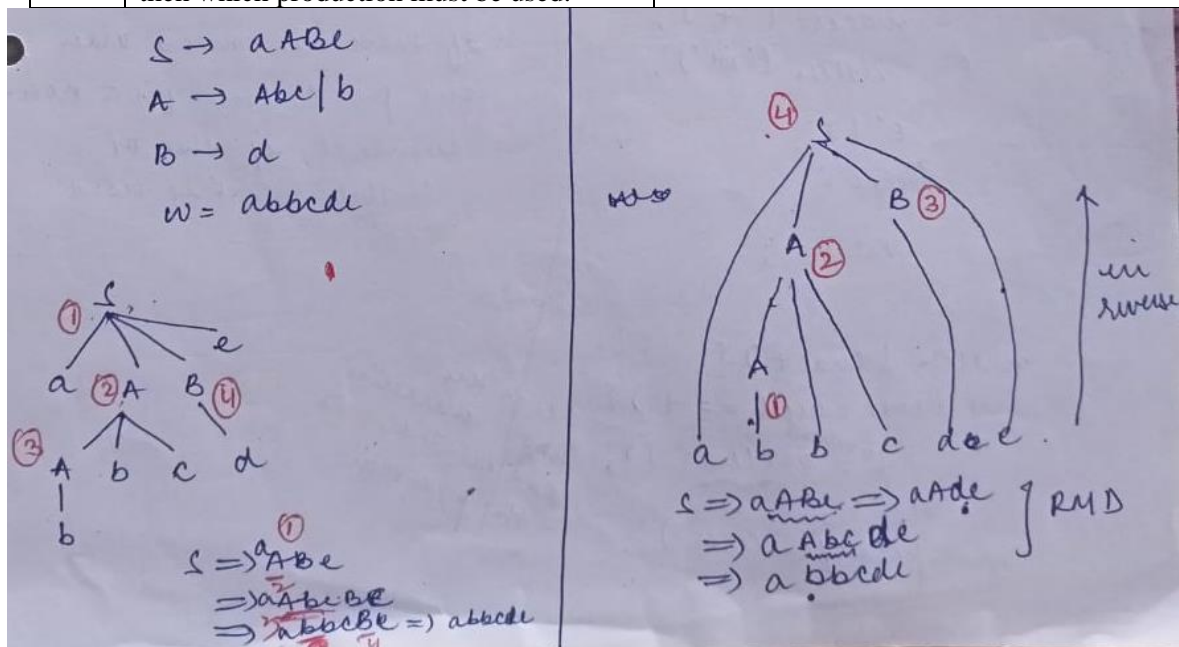
b. LR Parsers:

- i. LR(0) parser
- ii. SLR(1) Parser
- iii. CLR(1) Parser
- iv. LALR(1) parser

Difference between Top-down Parser and Bottom-up Parsers

S.No	Top-down	Bottom-up
1	Constructs the parse tree for the input string, starting from the root & creating the nodes of the tree in pre-order (root →	Constructs the parse tree for the input string beginning at the leaves (terminals) and working up towards the root(start symbol).

	left \rightarrow right) manner.	
2	Equivalent to finding the left most derivation for the input string.	Equivalent to finding the right most derivation for the input string in reverse.
3	$S \rightarrow aABe$ $A \rightarrow Abc b$ $B \rightarrow d$ $w = abbcd$ $S \Rightarrow aABe$ $S \Rightarrow aAbcBe$ $S \Rightarrow abbcBe$ $S \Rightarrow abbcd$	$S \rightarrow aABe$ $A \rightarrow Abc b$ $B \rightarrow d$ $w = abbcd$ $S \Rightarrow aABe$ $S \Rightarrow aAde$ $S \Rightarrow aAbcde$ $S \Rightarrow aabcde$
4	Root to terminals	Terminals to root
5	What production to use while making the decision to generate the string, i.e., if we have more than one alternative production then which production must be used.	When to reduce, i.e., which terminal can be used to reduce



Top-down parser with Full Backtracking

$A \rightarrow abC \mid aBd \mid aAD$

$B \rightarrow bB \mid \epsilon$

$C \rightarrow d \mid \epsilon$

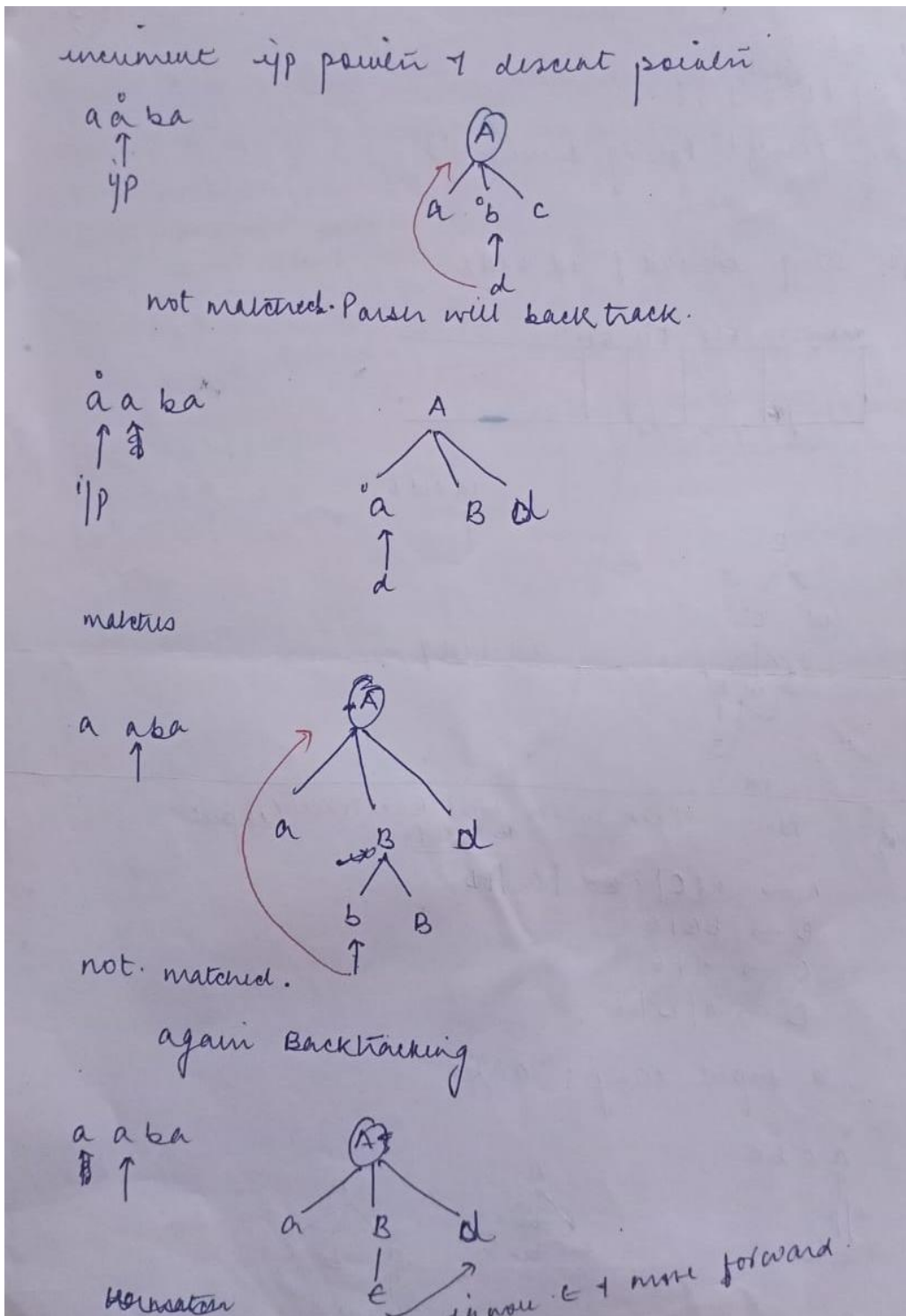
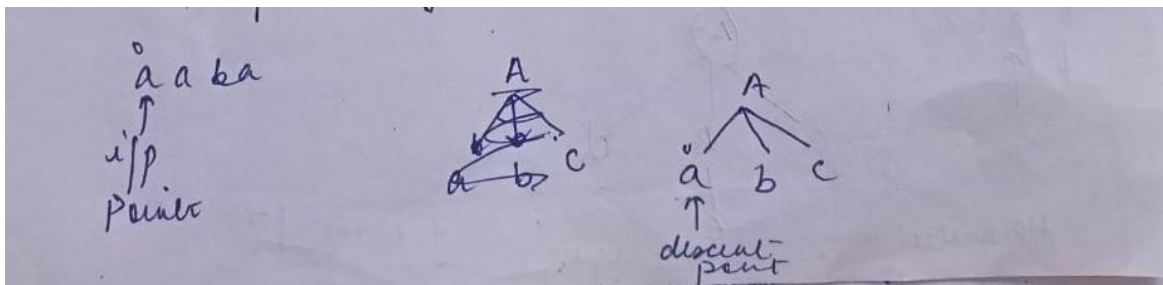
$D \rightarrow a \mid b \mid \epsilon$

Input String: aaba

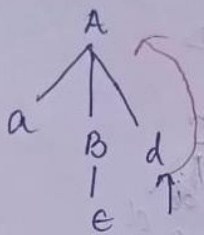
Two types of pointers are maintained:

Input pointer: points to the current symbol in the input string.

Descent pointer: points to the current symbol in the parse tree.

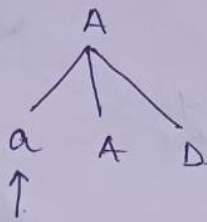


a a ba
↑



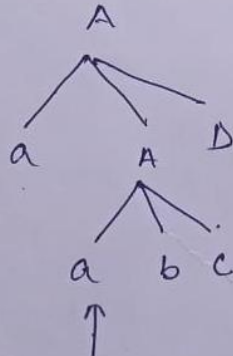
no match. Again backtrack.

a a ba
↑



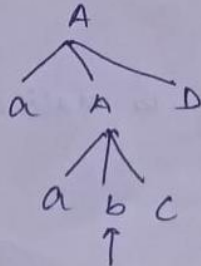
matches. Advance forward.

a a ba
↑



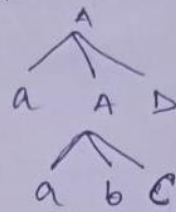
matches. Advance forward.

a a ba
↑



matches

a a b a
 * ↑

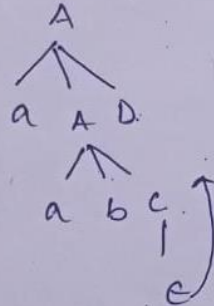


not matched. Backtrack: ↑ ↑

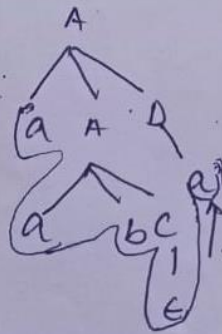
a a b a
 ↑



no match. backtrack & use another



a a b a
 ↑



match found. The string is matched. Hence parser is successful.

Recursive Descent Parser

Non-terminal/Variable: define and call the corresponding procedures

Terminals: compared with the input symbol. If the match is found, input pointer is incremented.

If a non-terminal produces more than one production, then all the production code must be written in corresponding procedure.

Algorithm

```
void A() {  
1)    Choose an A-production,  $A \rightarrow X_1X_2 \cdots X_k$ ;  
2)    for (  $i = 1$  to  $k$  ) {  
3)        if (  $X_i$  is a nonterminal )  
4)            call procedure  $X_i()$ ;  
5)        else if (  $X_i$  equals the current input symbol  $a$  )  
6)            advance the input to the next symbol;  
7)        else /* an error has occurred */;  
    }  
}
```

Step 1: Check for left recursion & non-determinism and eliminate them.

Step 2: For every production, write the corresponding procedure(function)

```
E → TE'  
E(){  
    T();  
    E'();  
}
```

Step 3: If non-terminal is encountered, call the corresponding procedure. If the terminal is encountered, compare it with the current input symbol. If the match is found, input pointer is incremented.

```
E → E+T | T  
T → T*F | F  
F → (E) | id
```

Step 1: Eliminate left recursion

```
E → TE'  
E' → +TE' | ε  
T → FT'  
T' → *FT' | ε  
F → (E) | id
```

```
E → TE'
```

```
E(){  
    T();  
    E'();
```

```
}
```

```
T(){  
    F();  
    T'();  
}
```

```
E'(){  
    if( input == '+' ){  
        input++;  
        T();  
        E'();  
    }else{  
        return; //  $E \rightarrow \epsilon$   
    }  
}
```

```
T'(){  
    If( input == '*' ){  
        input ++ ;  
        F();  
        T'();  
    }else{  
        return ; //  $T' \rightarrow \epsilon$   
    }  
}
```

```
F(){  
    if(input == '(' ) {  
        input ++ ;  
        E();  
        if (input == ')' ) {  
            input++ ;  
        }  
    }else if(input == 'id' ){  
        input++ ;  
    }  
}
```

$\$ \rightarrow$ input end marker ; $\text{id} + \text{id} * \text{id}\$$

```
main(){  
    E();  
    if (input == '$'){  
        printf("Parsing Successful");  
    }else{
```



```
    printf("Parsing Unsuccessful");  
}  
}
```

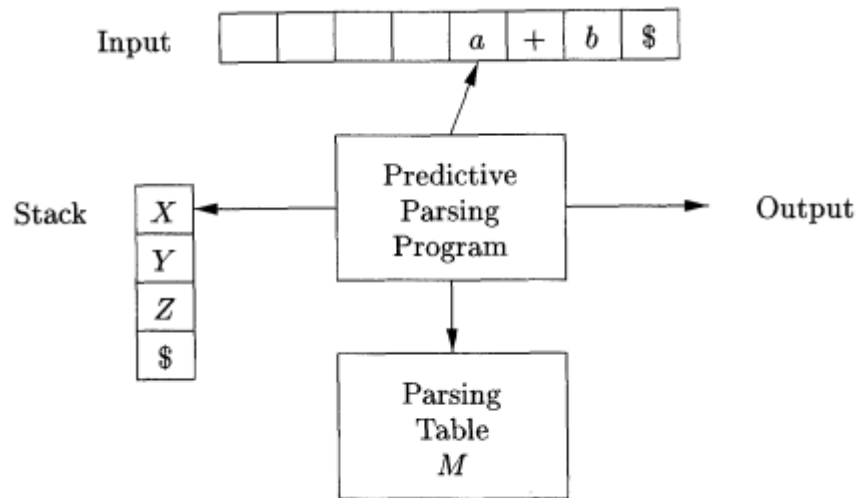
Q. Construct the recursive descent parser for

$E \rightarrow iE'$
 $E' \rightarrow +iE' \mid \varepsilon$
 $w = i + i\$$

$S \rightarrow cAd$
 $A \rightarrow aA \mid D$
 $D \rightarrow d$
 $w = caadd\$$

$A \rightarrow abC \mid Abd \mid aAD$
 $B \rightarrow bB \mid \varepsilon$
 $C \rightarrow d \mid \varepsilon$
 $D \rightarrow a \mid b \mid \varepsilon$
 $w = ababbd\$$

Predictive Parser(LL(1) Parser)



Model of Table-driven Predictive Parser

LL(1)

L: Left-to-right

L: Left most derivation

1: Number of symbols to be seen while making the decision

Input: buffer that contains the input string

Stack: data structure used for the procedure of parsing

LL(1) Parsing table: data structure constructed using the given grammar

\$: is used to get to the decision, i.e., when to stop

FIRST() & FOLLOW()

FIRST() & FOLLOW()

FIRST(X)

Eg 1:

$S \rightarrow aABCD$

$A \rightarrow b$

$B \rightarrow c$

$C \rightarrow d$

$D \rightarrow e$

FIRST(X)	Set of terminals
$\text{FIRST}(S) = \text{FIRST}(aABCD) = \text{FIRST}(a)$	{a}
$\text{FIRST}(A)$	{b}
$\text{FIRST}(B)$	{c}
$\text{FIRST}(C)$	{d}
$\text{FIRST}(D)$	{e}

Eg 2:

$S \rightarrow ABCD$

$A \rightarrow b$

$B \rightarrow c$

$C \rightarrow d$

$D \rightarrow e$

FIRST(X)	Set of terminals
$\text{FIRST}(S) = \text{FIRST}(ABCD) = \text{FIRST}(A)$	{b}
$\text{FIRST}(A) = \text{FIRST}(b)$	{b}
$\text{FIRST}(B) = \text{FIRST}(c)$	{c}
$\text{FIRST}(C) = \text{FIRST}(d)$	{d}
$\text{FIRST}(D) = \text{FIRST}(e)$	{e}

Eg 3:

$$S \rightarrow ABCD$$

$$A \rightarrow b \mid f$$

$$B \rightarrow c$$

$$C \rightarrow d$$

$$D \rightarrow e$$

FIRST(X)	Set of terminals
$\text{FIRST}(S) = \text{FIRST}(ABCD) = \text{FIRST}(A) = \text{FIRST}(b) \cup \text{FIRST}(f)$	$\{b, f\}$
$\text{FIRST}(A) = \text{FIRST}(b) \cup \text{FIRST}(f)$	$\{b, f\}$
$\text{FIRST}(B) = \text{FIRST}(c)$	$\{c\}$
$\text{FIRST}(C) = \text{FIRST}(d)$	$\{d\}$
$\text{FIRST}(D) = \text{FIRST}(e)$	$\{e\}$

Eg 4:

$$S \rightarrow ABCD$$

$$A \rightarrow b \mid f \mid \varepsilon$$

$$B \rightarrow c$$

$$C \rightarrow d$$

$$D \rightarrow e$$

$$w = cde\$$$

$$S \Rightarrow ABCD \Rightarrow BCD$$

FIRST(X)	Set of terminals
$\text{FIRST}(S) = \text{FIRST}(ABCD) = \text{FIRST}(A) = \text{FIRST}(b) \cup \text{FIRST}(f) \cup \text{FIRST}(B)$	$\{b, f, c\}$
$\text{FIRST}(A) = \text{FIRST}(b) \cup \text{FIRST}(f) \cup \text{FIRST}(\varepsilon)$	$\{b, f, \varepsilon\}$
$\text{FIRST}(B) = \text{FIRST}(c)$	$\{c\}$
$\text{FIRST}(C) = \text{FIRST}(d)$	$\{d\}$
$\text{FIRST}(D) = \text{FIRST}(e)$	$\{e\}$

Eg 5:

$$S \rightarrow ABCD$$

$$A \rightarrow b \mid \varepsilon$$

$$B \rightarrow c \mid \varepsilon$$

$$C \rightarrow d$$

$$D \rightarrow e$$

FIRST(X)	Set of terminals
$\text{FIRST}(S) = \text{FIRST}(ABCD) = \text{FIRST}(A) = \text{FIRST}(b) \cup \text{FIRST}(B)$	$\{b, c, d\}$
$\text{FIRST}(A) = \text{FIRST}(b) \cup \text{FIRST}(\varepsilon)$	$\{b, \varepsilon\}$
$\text{FIRST}(B) = \text{FIRST}(c) \cup \text{FIRST}(\varepsilon)$	$\{c, \varepsilon\}$
$\text{FIRST}(C) = \text{FIRST}(d)$	$\{d\}$
$\text{FIRST}(D) = \text{FIRST}(e)$	$\{e\}$

Eg 6:

$S \rightarrow ABCD$

$A \rightarrow b \mid \epsilon$

$B \rightarrow c \mid \epsilon$

$C \rightarrow d \mid \epsilon$

$D \rightarrow e \mid \epsilon$

FIRST(X)	Set of terminals
$\text{FIRST}(S) = \text{FIRST}(ABCD) = \text{FIRST}(A) = \text{FIRST}(b) \cup \text{FIRST}(B) \cup \text{FIRST}(C) \cup \text{FIRST}(D)$	$\{b, c, d, e, \epsilon\}$
$\text{FIRST}(A) = \text{FIRST}(b) \cup \text{FIRST}(\epsilon)$	$\{b, \epsilon\}$
$\text{FIRST}(B) = \text{FIRST}(c) \cup \text{FIRST}(\epsilon)$	$\{c, \epsilon\}$
$\text{FIRST}(C) = \text{FIRST}(d) \cup \text{FIRST}(\epsilon)$	$\{d, \epsilon\}$
$\text{FIRST}(D) = \text{FIRST}(e) \cup \text{FIRST}(\epsilon)$	$\{e, \epsilon\}$

Rules for finding FIRST(X)

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\text{FIRST}(X)$ if for some i , a is in $\text{FIRST}(Y_i)$, and ϵ is in all of $\text{FIRST}(Y_1), \dots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \xRightarrow{*} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \dots, k$, then add ϵ to $\text{FIRST}(X)$. For example, everything in $\text{FIRST}(Y_1)$ is surely in $\text{FIRST}(X)$. If Y_1 does not derive ϵ , then we add nothing more to $\text{FIRST}(X)$, but if $Y_1 \xRightarrow{*} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.
3. If $X \rightarrow \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

$S \rightarrow ABCD$

$A \rightarrow b$

$B \rightarrow c$

$C \rightarrow d$

$D \rightarrow e$

FIRST(X)	Set of Terminals
$\text{FIRST}(S) = \text{FIRST}(A)$	b
$\text{FIRST}(A) = \text{FIRST}(b)$	b
$\text{FIRST}(B)$	c
$\text{FIRST}(C)$	d
$\text{FIRST}(D)$	e

$S \rightarrow ABCD$

$A \rightarrow b \mid \epsilon$

$B \rightarrow c \mid \epsilon$

$C \rightarrow d \mid \epsilon$

$D \rightarrow e \mid \epsilon$

FIRST(X)	Set of Terminals
$\text{FIRST}(S) = \text{FIRST}(A) \cup \text{FIRST}(B) \cup \text{FIRST}(C) \cup \text{FIRST}(D)$	b, c, d, e, ϵ
$\text{FIRST}(A)$	b, ϵ
$\text{FIRST}(B)$	c, ϵ
$\text{FIRST}(C)$	d, ϵ
$\text{FIRST}(D)$	e, ϵ

FOLLOW(X): The terminal that follows a variable in the process of derivation. Every string is formed by the RHS of the production, so we need to take the RHS of the production & find out the FOLLOW(X).

Check whether the start symbol or any other variable is in the RHS of any production or not. If it is present, proceed with that variable else proceed with the next variable.

Eg1:

$S \rightarrow ABCD$
 $A \rightarrow a$
 $B \rightarrow b$
 $C \rightarrow c$
 $D \rightarrow d$

w = abcd\$

$S \Rightarrow ABCD\$$
 $S \Rightarrow aBCD\$$
 $S \Rightarrow abCD\$$
 $S \Rightarrow abcD\$$
 $S \Rightarrow abcd\$$

FOLLOW(X)	Set of terminals
FOLLOW(S)	{ \$ }
FOLLOW(A) = FIRST(BCD) = FIRST(B) = FIRST(b)	{ b }
FOLLOW(B) = FIRST(CD) = FIRST(C)	{ c }
FOLLOW(C) = FIRST(D)	{ d }
FOLLOW(D) = FOLLOW(S)	{ \$ }

Eg 2:

$S \rightarrow ABCD$
 $A \rightarrow b \mid \epsilon$
 $B \rightarrow c$
 $C \rightarrow d$
 $D \rightarrow e$

FOLLOW(X)	Set of terminals
FOLLOW(S)	{ \$ }
FOLLOW(A) = FIRST(BCD) = FOLLOW(B)	{ c }
FOLLOW(B) = FIRST(CD) = FIRST(C)	{ d }
FOLLOW(C) = FIRST(D)	{ e }
FOLLOW(D) = FOLLOW(S)	{ \$ }

Eg 3:

$S \rightarrow A$
 $A \rightarrow BC$
 $B \rightarrow c$
 $C \rightarrow a$

FOLLOW(X)	Set of terminals
FOLLOW(S)	{ \$ }
FOLLOW(A) = FOLLOW(S)	{ \$ }
FOLLOW(B) = FIRST(C)	{ a }
FOLLOW(C) = FOLLOW(A)	{ \$ }

Rules for Finding FOLLOW(X)

To compute FOLLOW(A) for all nonterminals A , apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST(β) contains ϵ , then everything in FOLLOW(A) is in FOLLOW(B).

$S \rightarrow \epsilon.XYZ$

$A \rightarrow \alpha B \beta$

Rule 2: $A \rightarrow \alpha B \beta$

FOLLOW(X)

A: S

B: X

α : ϵ

β : YZ

FIRST(β)

Rule 3: $A \rightarrow \alpha B$

FOLLOW(Z)

A: S

B: Z

or

$A \rightarrow \alpha B \beta$

$S \rightarrow ABCD$

$A \rightarrow b \mid \epsilon$

$B \rightarrow c \mid \epsilon$

$C \rightarrow d \mid \epsilon$

$D \rightarrow e \mid \epsilon$

More Examples on FIRST(X) & FOLLOW(X)

Eg 4:

$S \rightarrow ABCD$

$A \rightarrow a \mid \varepsilon$

$B \rightarrow CD \mid b$

$C \rightarrow c \mid \varepsilon$

$D \rightarrow Aa \mid d \mid \varepsilon$

FIRST(X)	Set of terminals
$FIRST(S) = FIRST(ABCD) = FIRST(A) \cup FIRST(B) \cup FIRST(C) \cup FIRST(D)$	$\{a, b, c, d, \varepsilon\}$
$FIRST(A) = FIRST(a) \cup FIRST(\varepsilon)$	$\{a, \varepsilon\}$
$FIRST(B) = FIRST(CD) \cup FIRST(b)$	$\{a, b, c, d, \varepsilon\}$
$FIRST(C) = FIRST(c) \cup FIRST(\varepsilon)$	$\{c, \varepsilon\}$
$FIRST(D) = FIRST(Aa) \cup FIRST(d) \cup FIRST(\varepsilon)$	$\{a, d, \varepsilon\}$

FOLLOW(X)	Set of terminals
$FOLLOW(S)$	$\{\$ \}$
$FOLLOW(A) = FIRST(BCD) \cup FIRST(CD) \cup FIRST(D) \cup FOLLOW(S) \cup FIRST(a)$	$\{a, b, c, d, \$ \}$
$FOLLOW(B) = FIRST(CD) \cup FIRST(D) \cup FOLLOW(S)$	$\{c, a, d, \$ \}$
$FOLLOW(C) = FIRST(D) \cup FOLLOW(S) \cup FOLLOW(B)$	$\{a, d, c, \$ \}$
$FOLLOW(D) = FOLLOW(S) \cup FOLLOW(B)$	$\{c, a, d, \$ \}$

Eg. 5

$S \rightarrow ABCDE$

$A \rightarrow a \mid \varepsilon$

$B \rightarrow b \mid \varepsilon$

$C \rightarrow c \mid \varepsilon$

$D \rightarrow d \mid \varepsilon$

$E \rightarrow e \mid \varepsilon$

Eg. 6

$S \rightarrow Bb \mid Cd$

$B \rightarrow aB \mid \varepsilon$

$C \rightarrow cC \mid \varepsilon$

Eg. 7

$S \rightarrow ABCDE$

$A \rightarrow a \mid \varepsilon$

$B \rightarrow b \mid \varepsilon$

$C \rightarrow c$

$D \rightarrow d \mid \varepsilon$

$E \rightarrow e \mid \varepsilon$

Eg. 8:

$S \rightarrow ACB \mid CbB \mid Ba$
 $A \rightarrow da \mid BC$
 $B \rightarrow g \mid \epsilon$
 $C \rightarrow h \mid \epsilon$

Eg 9:

$S \rightarrow aABb$
 $A \rightarrow C \mid \epsilon$
 $B \rightarrow d \mid \epsilon$

Eg 10:

$S \rightarrow aBDh$
 $B \rightarrow cC$
 $C \rightarrow bC \mid \epsilon$
 $D \rightarrow EF$
 $E \rightarrow g \mid \epsilon$
 $F \rightarrow f \mid \epsilon$

Construction of LL(1) / Predictive Parsing Table

Algorithm 4.31: Construction of a predictive parsing table.

INPUT: Grammar G .

OUTPUT: Parsing table M .

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in $\text{FIRST}(A)$, add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in $\text{FIRST}(\alpha)$, then for each terminal b in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in $\text{FIRST}(\alpha)$ and $\$$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). \square

Note: Always check for left recursion and non-determinism.

Eg.

$E \rightarrow E+T \mid T$
 $T \rightarrow T * F \mid F$
 $F \rightarrow \text{id}$

$E \rightarrow TE'$
 $E' \rightarrow +TE' \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT' \mid \epsilon$
 $F \rightarrow id \mid (E)$

FIRST(X)	Set of terminals
$FIRST(E) = FIRST(TE')$	$\{id, (\}$
$FIRST(E') = FIRST(+TE') \cup FIRST(\epsilon)$	$\{+, \epsilon\}$
$FIRST(T) = FIRST(FT')$	$\{id, (\}$
$FIRST(T') = FIRST(*FT') \cup FIRST(\epsilon)$	$\{*, \epsilon\}$
$FIRST(F) = FIRST(id) \cup FIRST((E))$	$\{id, (\}$

FOLLOW(X)	Set of terminals
$FOLLOW(E) = FIRST()$	$\{), \$\}$
$FOLLOW(E') = FOLLOW(E) \cup FOLLOW(E')$	$\{), \$\}$
$FOLLOW(T) = FIRST(E') \cup FOLLOW(E) \cup FOLLOW(E')$	$\{+,), \$\}$
$FOLLOW(T') = FOLLOW(T) \cup FOLLOW(T')$	$\{+,), \$\}$
$FOLLOW(F) = FIRST(T') \cup FOLLOW(T) \cup FOLLOW(T')$	$\{*, +,), \$\}$

a) $E \rightarrow TE'$

$FIRST(E) = FIRST(TE') = \{id, (\}$

$M[E, id] = E \rightarrow TE'$

$M[E, (] = E \rightarrow TE'$

b) $E' \rightarrow +TE' \mid \epsilon$

i) $E' \rightarrow +TE'$

$FIRST(E) = FIRST(+TE') = \{+\}$

$M[E', +] = E' \rightarrow +TE'$

ii) $E' \rightarrow \epsilon$

$FIRST(E') = FIRST(\epsilon) = \{\epsilon\}$

Since $FIRST(RHS)$ contains ϵ , therefore we must find $FOLLOW(LHS)$,

$FOLLOW(E') = \{), \$\}$

$M[E',)] = E' \rightarrow \epsilon$

$M[E', \$] = E' \rightarrow \epsilon$

c) $T \rightarrow FT'$

$FIRST(T) = FIRST(FT') = \{id, (\}$

$M[T, id] = T \rightarrow FT'$

$M[T, (] = T \rightarrow FT'$

d) $T' \rightarrow *FT' \mid \epsilon$

i) $T' \rightarrow *FT'$

$FIRST(T') = FIRST(*FT') = \{*\}$

$M[T', *] = T' \rightarrow *FT'$

- ii) $T' \rightarrow \varepsilon$
 $\text{FIRST}(T') = \text{FIRST}(\varepsilon) = \{\varepsilon\}$
 $\text{FOLLOW}(T') = \{+,), \$\}$
 $M[T', +] = T' \rightarrow \varepsilon$
 $M[T',)] = T' \rightarrow \varepsilon$
 $M[T', \$] = T' \rightarrow \varepsilon$
- e) $F \rightarrow \text{id} \mid (E)$
 i) $F \rightarrow \text{id}$
 $\text{FIRST}(F) = \text{FIRST}(\text{id}) = \{\text{id}\}$
 $M[F, \text{id}] = F \rightarrow \text{id}$
- ii) $F \rightarrow (E)$
 $\text{FIRST}(F) = \text{FIRST}((E)) = \{(\}$
 $M[F, (] = F \rightarrow (E)$

Non-Terminals/ Input Symbols	id	+	*	()	\$
E	$E \rightarrow TE'$	<i>error</i>	<i>error</i>	$E \rightarrow TE'$	<i>error</i>	<i>error</i>
E'	<i>error</i>	$E' \rightarrow +TE'$	<i>error</i>	<i>error</i>	$E' \rightarrow \varepsilon$	$E' \rightarrow \varepsilon$
T	$T \rightarrow FT'$	<i>error</i>	<i>error</i>	$T \rightarrow FT'$	<i>error</i>	<i>error</i>
T'	<i>error</i>	$T' \rightarrow \varepsilon$	$T' \rightarrow *FT'$	<i>error</i>	$T' \rightarrow \varepsilon$	$T' \rightarrow \varepsilon$
F	$F \rightarrow \text{id}$ $F \rightarrow \text{id}E$	<i>error</i>	<i>error</i>	$F \rightarrow (E)$	<i>error</i>	<i>error</i>

Algorithm 4.34: Table-driven predictive parsing.

INPUT: A string w and a parsing table M for grammar G .

OUTPUT: If w is in $L(G)$, a leftmost derivation of w ; otherwise, an error indication.

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above $\$$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. \square

```

set  $ip$  to point to the first symbol of  $w$ ;
set  $X$  to the top stack symbol;
while (  $X \neq \$$  ) { /* stack is not empty */
    if (  $X$  is a ) pop the stack and advance  $ip$ ;
    else if (  $X$  is a terminal ) error();
    else if (  $M[X, a]$  is an error entry ) error();
    else if (  $M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_k$  ) {
        output the production  $X \rightarrow Y_1 Y_2 \cdots Y_k$ ;
        pop the stack;
        push  $Y_k, Y_{k-1}, \dots, Y_1$  onto the stack, with  $Y_1$  on top;
    }
    set  $X$  to the top stack symbol;
}

```

Figure 4.20: Predictive parsing algorithm

$w = id + id\$$

MATCHED	STACK	INPUT	ACTION
	E\$	id+id\$	
	TE'\$	id+id\$	O/P: E \rightarrow TE'
	FT'E'\$	id+id\$	O/P: T \rightarrow FT'
	idT'E'\$	id+id\$	O/P: F \rightarrow id
id	T'E'\$	+id\$	Matched: id
id	E'\$	+id\$	O/P: T $\rightarrow \epsilon$
id	+TE'\$	+id\$	O/P: E' \rightarrow +TE'
id+	TE'\$	id\$	Matched: +
id+	FT'E'\$	id\$	O/P: T \rightarrow FT'
id+	idT'E'\$	id\$	O/P: F \rightarrow id
id+id	T'E'\$	\$	Matched: id
id+id	E'\$	\$	O/P: T' $\rightarrow \epsilon$
id+id	\$	\$	Matched; Parsing completed successfully

MATCHED	STACK	INPUT	ACTION
	E\$	id+id*id\$	
	TE'\$	id+id*id\$	O/P: E \rightarrow TE'
	FT'E'\$	id+id*id\$	O/P: T \rightarrow FT'
	idT'E'\$	id+id*id\$	O/P: F \rightarrow id
id	T'E'\$	+id*id\$	Matched id
id	E'\$	+id*id\$	O/P: T' $\rightarrow \epsilon$
id	+TE'\$	+id*id\$	O/P: E' \rightarrow +TE'
id+	TE'\$	id*id\$	Matched +
id+	FT'E'\$	id*id\$	O/P: T \rightarrow FT'
id+	idT'E'\$	id*id\$	O/P: F \rightarrow id
id+id	T'E'\$	*id\$	Matched id
id+id	*FT'E'\$	*id\$	O/P: T' \rightarrow *FT'
id+id*	FT'E'\$	id\$	Matched *
id+id*	idT'E'\$	id\$	O/P: F \rightarrow id
id+id*id	T'E'\$	\$	Matched id
id+id*id	E'\$	\$	O/P: T' $\rightarrow \epsilon$
id+id*id	\$	\$	O/P: E' $\rightarrow \epsilon$ Parsing completed successfully

More questions on Predictive Parsing

1. $S \rightarrow aBDh$

$B \rightarrow Bb \mid C$

$D \rightarrow EF$

$E \rightarrow g \mid \epsilon$

$F \rightarrow f \mid \epsilon$ $w = acgf$

2. $S \rightarrow iLuET \mid a$

$L \rightarrow LS \mid \epsilon$

$E \rightarrow b$

$T \rightarrow d \mid e \mid \epsilon$ $w = iubde$

3. $A \rightarrow BCc \mid fDB$
 $B \rightarrow bCDE \mid \epsilon$
 $C \rightarrow DaB \mid ca$
 $D \rightarrow dB \mid \epsilon$
 $E \rightarrow Eaf \mid c \quad w = gdbca$

LL(1) Grammar

A grammar is said to be LL(1) iff

- $A \rightarrow \alpha_1 \mid \alpha_2, \text{FIRST}(\alpha_1) \cap \text{FIRST}(\alpha_2) = \Phi$
- $A \rightarrow \alpha \mid \epsilon, \text{FIRST}(\alpha) \cap \text{FOLLOW}(A) = \Phi$

If a grammar is not LL(1), then it is impossible to construct LL(1)/Predictive Parser/ Non-recursive Descent Parser.

Check for left recursion and non-determinism

Eg.1

$S \rightarrow AB$
 $A \rightarrow a \mid \epsilon$
 $B \rightarrow b \mid \epsilon$
 $\text{FIRST}(a) \cap \text{FOLLOW}(A)$
 $\{a\} \cap \{b, \$\} = \Phi$
 Grammar is LL(1)

Eg. 2

$S \rightarrow A \mid a$
 $A \rightarrow a$
 $\text{FIRST}(\alpha_1) \cap \text{FIRST}(\alpha_2) = \Phi$
 $\{a\} \cap \{a\} \neq \Phi$
 Grammar is not LL(1)

Eg. 3

$S \rightarrow iEtS \mid iEtSeS \mid a$
 $E \rightarrow b$

After performing left-factoring,

$S \rightarrow iEtSS' \mid a$
 $S' \rightarrow eS \mid \epsilon$
 $E \rightarrow b$

Non-terminal	FIRST	FOLLOW
S	{i, a}	{e, \$}
S'	{e, ε}	{e, \$}
E	{b}	{t}

Non-Terminals/ Input Symbols	a	b	e	i	t	\$
S	$S \rightarrow a$			$S \rightarrow iEtSS'$		
S'			$S' \rightarrow eS$ $S' \rightarrow \epsilon$			$S' \rightarrow \epsilon$
E		$E \rightarrow b$				

Grammar is not LL(1)

Q. Check whether the given grammars are LL(1) or not

1. $S \rightarrow aAB \mid bA \mid \epsilon$
2. $A \rightarrow aAb \mid \epsilon$
3. $B \rightarrow bB \mid c$

Bottom-up Parser

- A bottom-up parser corresponds to the construction of a parse tree for an input string beginning at the leaves(bottom/terminals) and working up towards the root (top/start symbol).
- The parse tree is constructed in the reverse order of RMD, i.e., the last step of RMD is the first step of construction of parse tree in bottom-up parser.

Eg. 1

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

$w = abbcde$

\rightarrow : used in productions/grammar

\Rightarrow : Derivation

$S \Rightarrow aABe \Rightarrow aAde \Rightarrow aAbcde \Rightarrow abbcde$

LHS to RHS: expansion

RHS to LHS: reduction

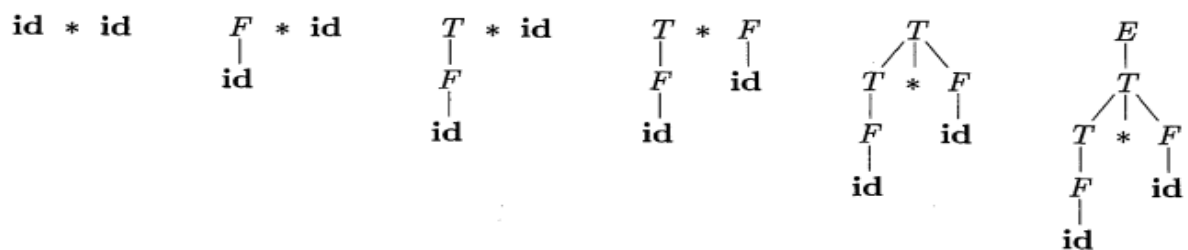
Eg. 2

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id$

$E \Rightarrow T \Rightarrow T * F \Rightarrow T * id \Rightarrow F * id \Rightarrow id * id$



Handle: substring which matches the RHS of the production. Handles in the order of reduction are $b, Abc, d, aABe$

Handle Pruning

Handle is substring which matches the RHS of the production and whose reduction represents one step along the reverse of a rightmost derivation.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$\mathbf{id_1 * id_2}$	$\mathbf{id_1}$	$F \rightarrow \mathbf{id}$
$F * \mathbf{id_2}$	F	$T \rightarrow F$
$T * \mathbf{id_2}$	$\mathbf{id_2}$	$F \rightarrow \mathbf{id}$
$T * F$	$T * F$	$E \rightarrow T * F$

Handles during a parse of $\mathbf{id_1 * id_2}$

A rightmost derivation in reverse can be obtained by "handle pruning".

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow \mathbf{id}$

Right Sentential Form	Handle	Reducing production
$\mathbf{id + id * id}$	\mathbf{id}	$F \rightarrow \mathbf{id}$
$F + \mathbf{id * id}$	F	$T \rightarrow F$
$T + \mathbf{id * id}$	T	$E \rightarrow T$
$E + \mathbf{id * id}$	\mathbf{id}	$F \rightarrow \mathbf{id}$
$E + F * \mathbf{id}$	F	$T \rightarrow F$
$E + T * \mathbf{id}$	\mathbf{id}	$F \rightarrow \mathbf{id}$
$E + T * F$	$T * F$	$T \rightarrow T * F$
$E + T$	$E + T$	$E \rightarrow E + T$