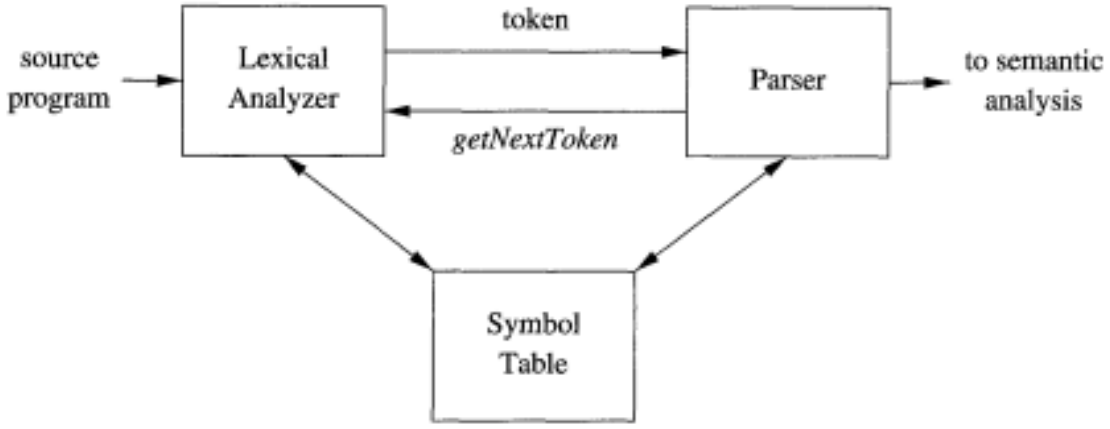
Compiler Design

Lexical Analysis



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

a. read the input characters of the source program, for eg. 'i' , 'n' , 't', ' ' , 'm' , ‘a’, ‘i’ , ‘n’, ‘(‘ , ‘)’ , ...
b. group them into lexemes. For eg. int main() …
c. produce as output a sequence of tokens for each lexeme in the source program. For eg. <int>, <id,1> , …
d. The stream of tokens is sent to the parser for syntax analysis.
e. the lexical analyzer may keep track of the number of newline characters seen, so it can associate a line number with each error message.
f. If the source program uses a macro-preprocessor, the expansion of macros may also be performed by the lexical analyser. 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

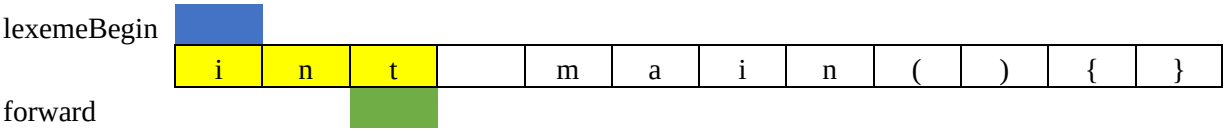
c. char *(a+5) = "abcd"; → 10 Tokens
d. char a[5] = "abcd"; → char *(a+5) = "abcd";
e. int a[5][4]; ➔ int *(*(a+5)+4);

Input Buffering

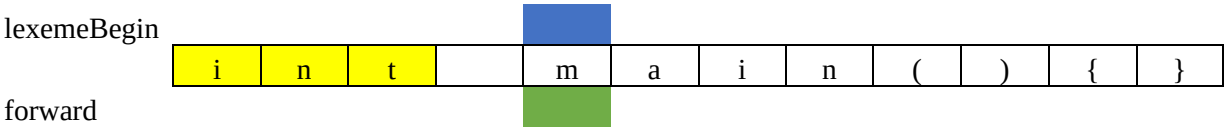
a. **lexemeBegin pointer:** points to the beginning character of the current lexeme
b. **forward pointer:** Initially points to the beginning of the current lexeme. Advances forward as the LA scans the source program.
For eg

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.

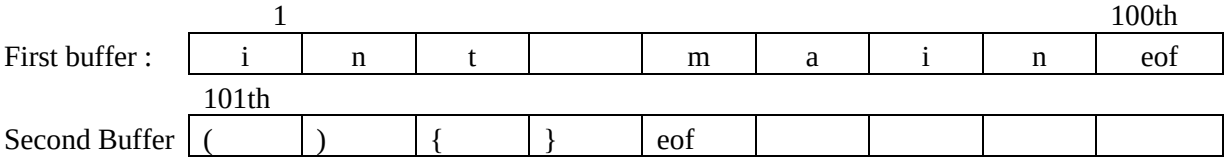
Buffering

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:

a. One buffer system: only one buffer is used.
Disadvantage: Overwriting

b. 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 = { … }
   for eg.

   Empty set, {ϵ}

   L = {a,aa,ab,bb} ➔ a+b*

   Language with empty string, L= {E} , E → 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, E.s = s.E = s

7. **Exponentiation**: if S^0 = E & 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 → String

9. Length of a String: |s| → Length of String

Length of empty string is 0

==========

Operations on Languages

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

a. Union:
   L = {a,b,c} M = {d,e,f}
   L U M = {a,b,c,d,e,f}

   L = {a,b,c} M = {a,d,e,f}
   L U M = {a,b,c,d,e,f}

b. Concatenation:
   L = { a, b,ab,ba } M = {a,b}

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

c. Kleene Closure:
   L^0 = Set of strings of length 0
   L^1 = Set of strings of length 1
   L^2 = Set of strings of length 2

   Kleene Closure = L^0 U L^1 U L^2 U L^3 U L^4 …

   L = {a,b}

   L^* = {E, a,b,aa,bb,ab,ba …}

d. Positive Closure:
   L^1 = Set of strings of length 1
   L^2 = Set of strings of length 2

   Positive Closure = L^1 U L^2 U L^3 U L^4 …

   L = {a,b}

   L^+ = {a,b,aa,bb,ab,ba …}

Eg.

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

a. L U D is the set of letters and digits
b. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit
c. L^4 is the set of all 4-letter strings.
d. L* is the set of all strings of letters, including E, the empty string.
e. L (L U D) * : identifers(abc, abc123,..), int

Regular Expression

a. Ways to represent the regular language.
b. It is an expression of string and operators
   Operations
   i. * : Kleene Closure
   ii. $^+$: Positive Closure
   iii. . : Concatenation
   iv. + or | : Union

BASIS: There are two rules that form the basis:

a. ε is RE, L(ε) = { ε }
b. 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.

1. (r)|(s) is a regular expression denoting the language L(r) U L(s)
2. (r)(s) is a regular expression denoting the language L(r).L(s)
3. (r)* is a regular expression denoting (L(r)) *
4. (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. (a|b) : L(a) = {a}, L(b) = {b}; L(a) U L(b); L = {a,b}
b. (a+b)(a+b) : : L = {aa,ab,bb,ba}
c. (a|b)(a|b)(a|b)(a|b) : L = {aaaa, aaab,aabb,…}
d. (a|b)* : L = {ε, a,b,aa,bb,ab,ba,…}
e. (a*b*)* : L = {ε,a,b,aa,aab,…}
f. a|a*b : L = {a,b,ab,aab,aaab,…}

Regular Definitions

a. It is the name given to a regular expression.
b. regular definition is a sequence of definitions of the form:

$\mathbf{d_i \rightarrow r_i}$

roll_number → (Year_of_study).(college_code).(branch_code).(serial_number)

phone_number → country_code.area_code.unique_number

i.
letter_ → _|A|B|C|D...|Z|a|b|c|d|...|z
digit → 0|1|2|3|4|...|9
*id → letter_(letter_ | digit)**

ii.
digit → 0|1|2|3|4|...|9
*digits → digit digit**
optionalFraction → .digits | ε
optionalExponent → (E(+|-|ε)digits) | ε
unsignednumber → digits optionalFraction optionalExponent

iii.
sign → +|-| ε
digit → 0|1|2|3|4|...|9
*digits → digit digit**
optionalFraction → .digits | ε
optionalExponent → (E sign digits) | ε
signednumber → 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 $(L(r))^+$
$r^+ = r.r^*$;
$r^* = r^+ \mid \varepsilon$
b. *Zero or one instance: ?*
$r? = r \mid \varepsilon$
c. *Character classes: A regular expression* $a_1|a_2| \ldots |a_n$, where a_i's are each symbols of the alphabet, can be written as the shorthand $[a_1a_2 \ldots a_n]$

sign → [+-]?

digit → [0-9]
letter_ → [A-Za-z_]

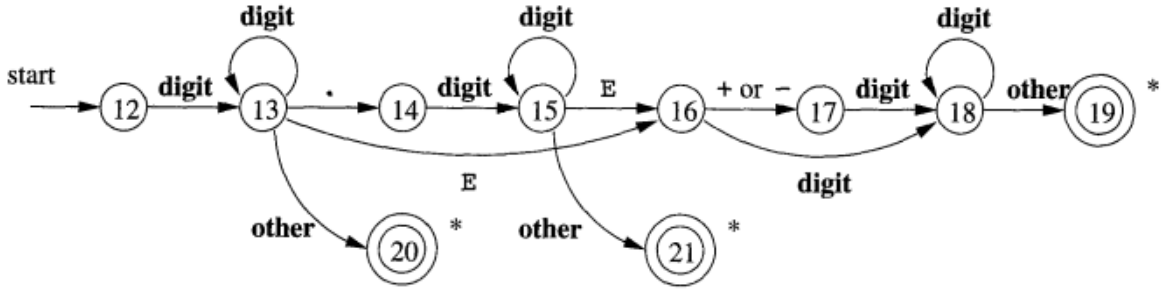
i.
letter_ → [_A-Za-z]
digit → [0-9]
*id → letter (letter_ | digit)**

ii.

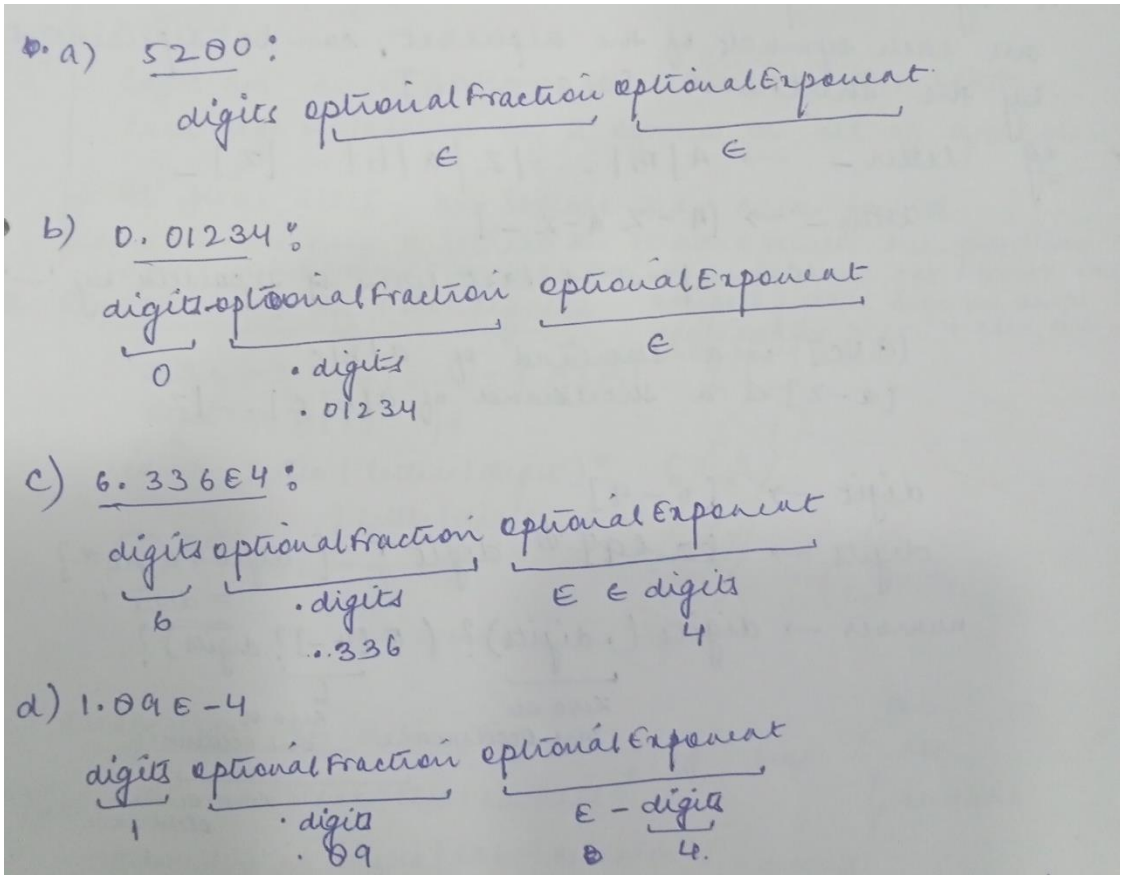
digit → *[0-9]*
digits → *digit*$^{+}$
optionalFraction → *.digits?*
optionalExponent → *(E [+-]?)digits)?*
unsignednumber → ***digits optionalFraction optionalExponent***

iii.

sign → *[+-]?*
digit → *[0-9]*
digits → *digit*$^{+}$
optionalFraction → *(.digits)?*
optionalExponent → *(E sign digits)?*
signednumber → ***digits optionalFraction optionalExponent***

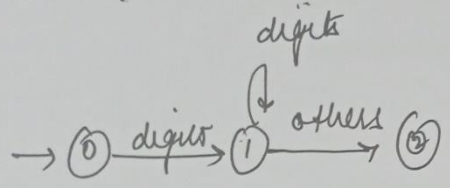


A transition diagram for unsigned numbers

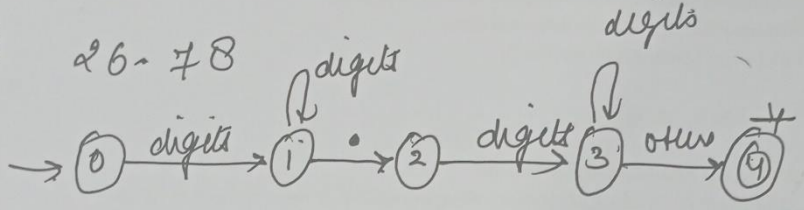


44

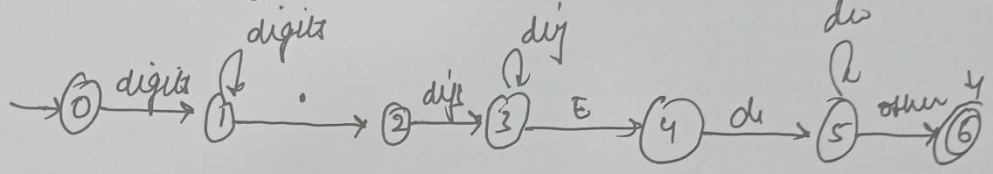
(1)



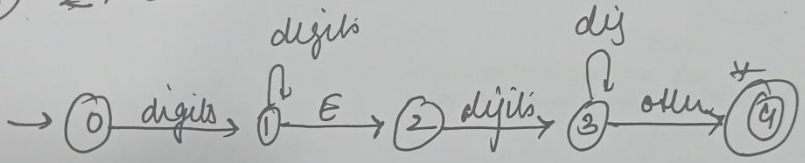
(2) 26.78



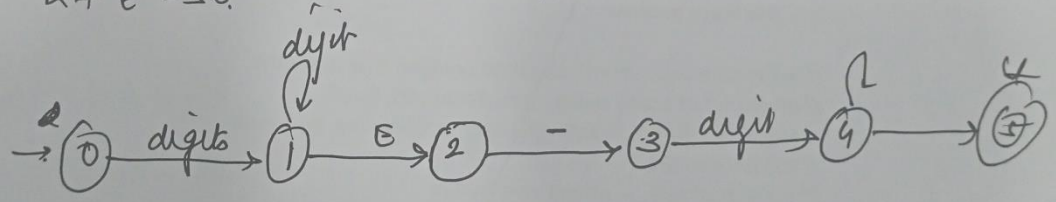
(3) 27.0E56



(4) 27E56



(5) 27E-56



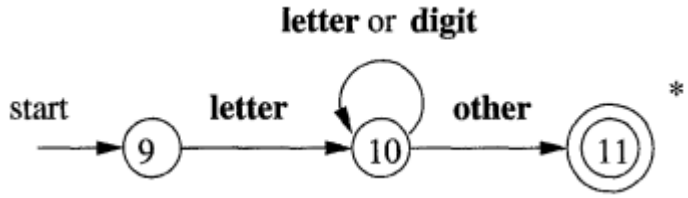
Recognition of Tokens

Transition diagrams: state, transitions

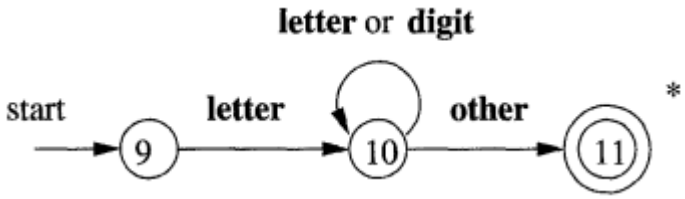
a. Recognition of identifiers:

letter → *[_A-Za-z]*
digit → *[0-9]*
id → *letter (letter | digits)* *

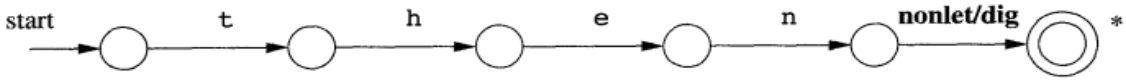
int no_Of_Books= 9



b. Recognition of an keywords/reserved words:

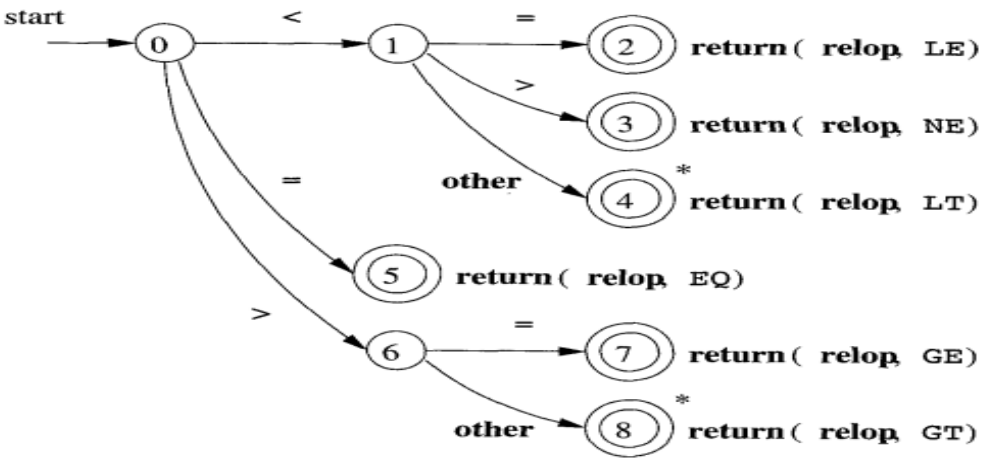


int, float



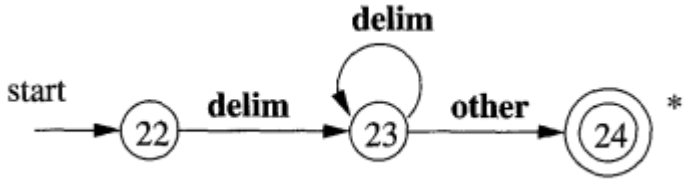
then
else
int

c. **Recognition of relational operators:**
<= , = , >=, !=, <>,<



d. **Recognition of Whitespaces:**

delim → *blank|tab|newline*
ws → *(blank|tab|newline)*$^+$
ws → *delim*$^+$
ws → *delim(delim)**



Construction of NFA from Regular expression

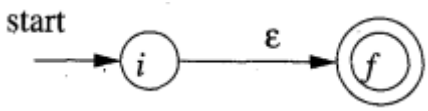
Algorithm: The McNaughton-Yamada-Thompson algorithm to convert a regular expression to an NFA

INPUT: A regular expression r over alphabet ∑.

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

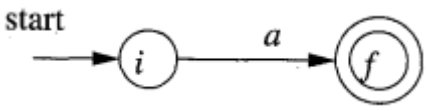
BASIS: For regular expression ε, construct the NFA

r = ε; L(r) = { ε }



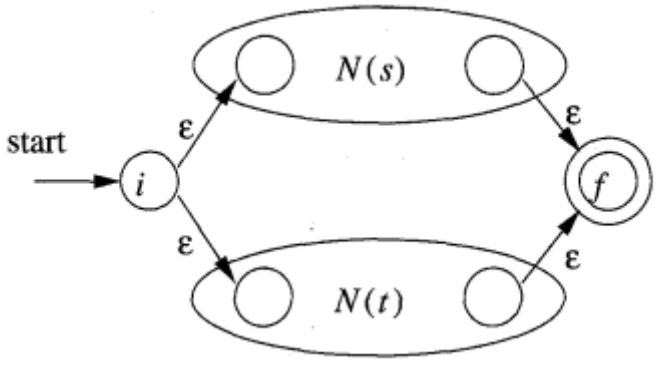
For any regular expression a in ∑, construct the NFA

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

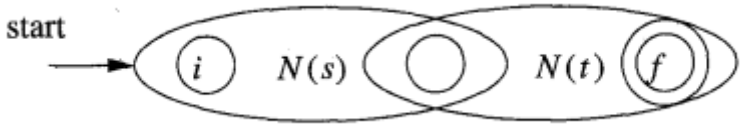


INDUCTION: Suppose **N(s)** and **N(t)** are NFA's for regular expressions **s** and **t**, respectively.

a. Suppose *r* = *s* | *t*



b. Suppose $r = st$



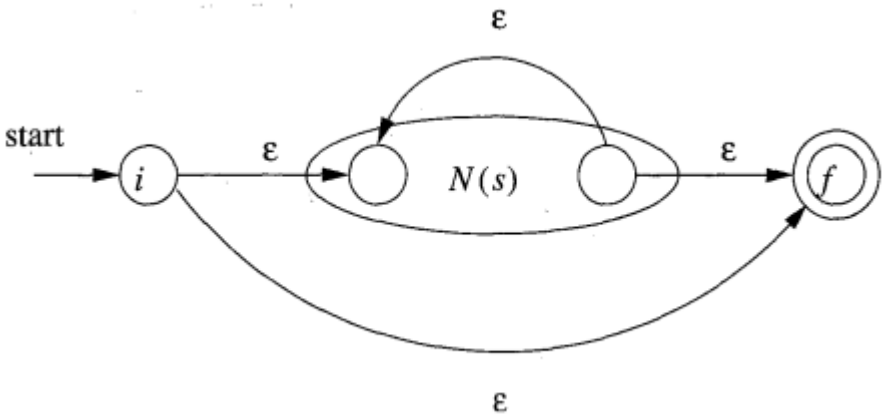
c. Suppose r = s^*

ε = ε

x = x

x. ε. x = xx

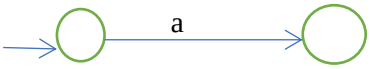
x. ε .x. ε. x = xxx

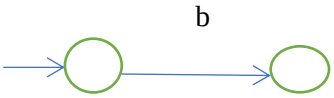


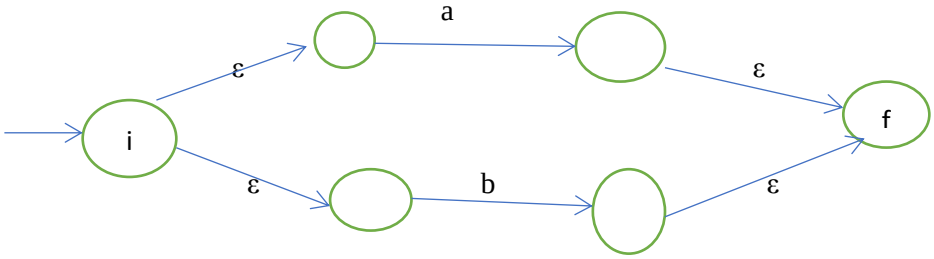
r = s*; L(r) = { ε, s,ss,sss,…}

For eg.

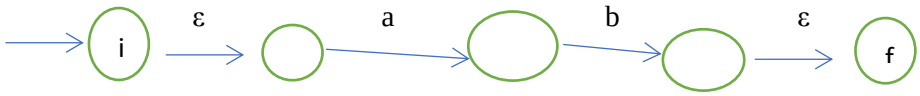
i. (a|b); L(r)={a,b}



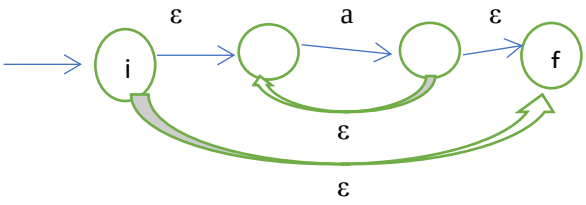




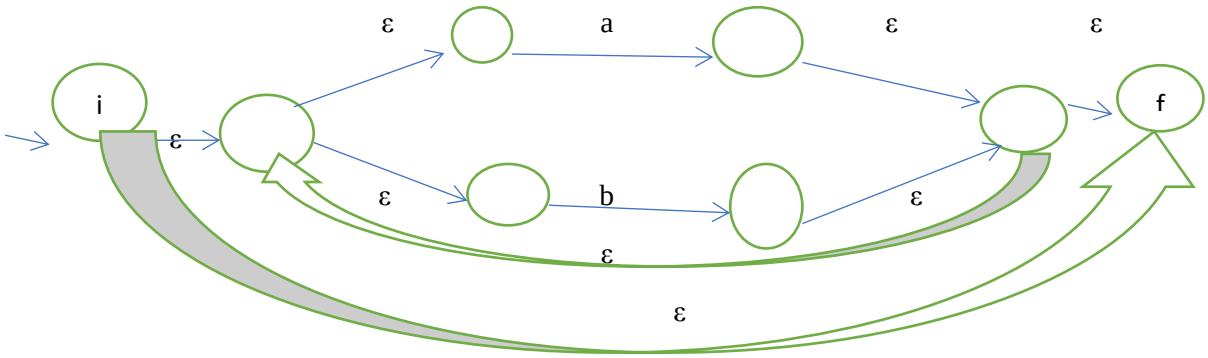
ii. ab



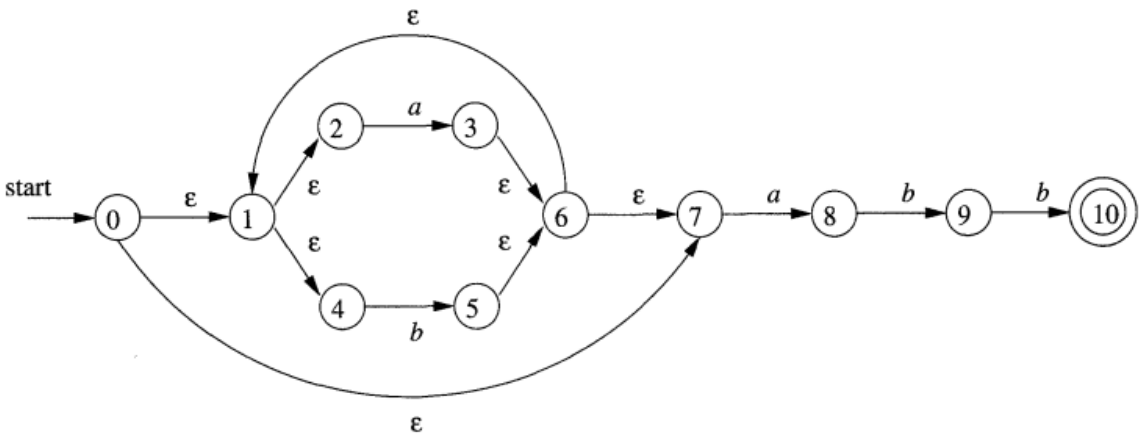
iii. a* : L(r) = { ε,a,aa,…}



iv. (a|b)*; L(r) = { ε ,a,b,aa,ab,bb,ba,…}



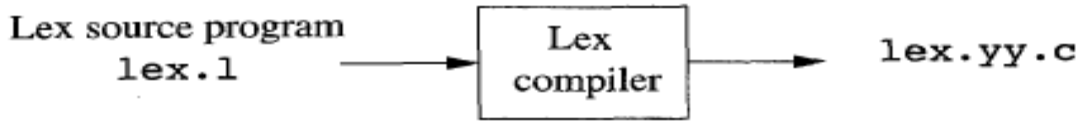
v. (a|b)*abb

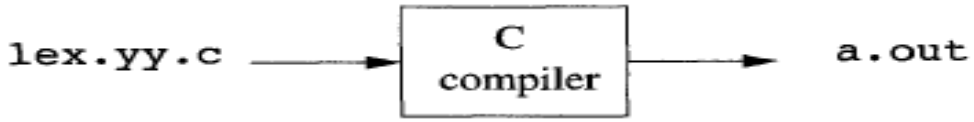


vi.(a|b) *(abb)*

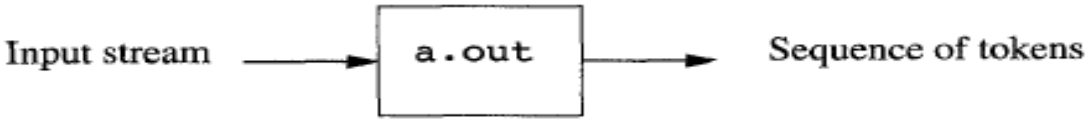
The Lexical-Analyzer Generator (*Lex)*

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





a.out → working lexical analyser



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

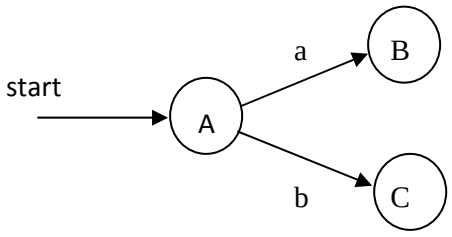
Operation	Description
ε-closure(s)	Set of NFA states reachable from NFA state s on ε-transitions alone
ε-closure(T)	Set of NFA states reachable from set of states **T** on ε-transitions alone
move(T,a)	Set of states to which there is a transition on input symbol **a** from some NFA state in **T**

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 $\mathbf{2^n}$ 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 move(A,a) The set is {3,8}:
   move(A,a) = {3,8}

b. now do an ε-closure on **move(A,a)**. Find all the states in **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(move(A,a)) = **B = {1,2,3,4,6,7,8}**

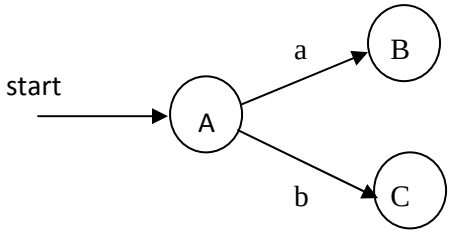
   This defines the new state B that has an edge on **a** from A

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 move(A,b) The set is {5}:
   move(A,b) = {5}

d. now do an ε-closure on **move(A,b)**. Find all the states in **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(move(A,a)) = **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}

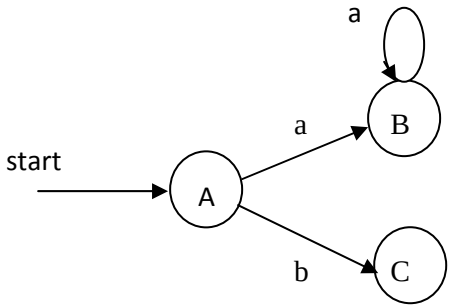
Now that we have B and C we can move on to find the states that have **a** and **b** transitions from B and C.

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 move(B,a) The set is {3,8}: **move(B,a)** = {3,8}

f. now do an ε-closure on **move(B,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(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:



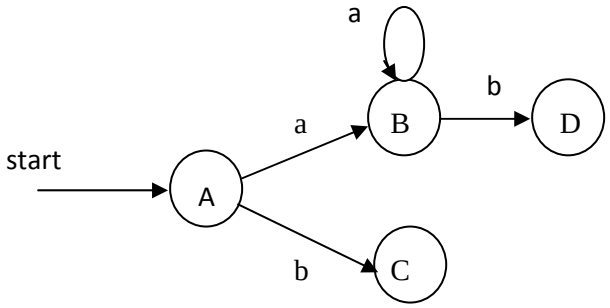
A={0,1,2,4,7}
B={1,2,3,4,6,7,8}
C={1,2,4,5,6,7}

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 move(B,b) The set is {5,9}: **move(B,b)** = {5,9}

h. now do an ε-closure on **move(B,b)**. Find all the states in **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 ε-closure(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}

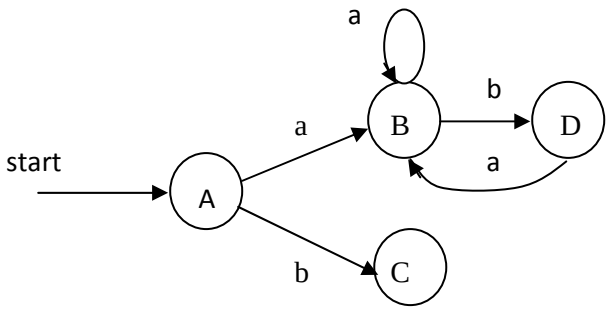


Find the state that has an edge on a from D

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 move(D,a) The set is {3,8}: **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}

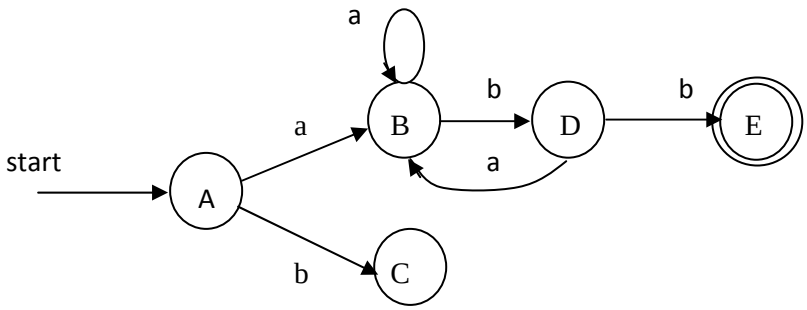


Find the state E that has an edge on b from D

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}



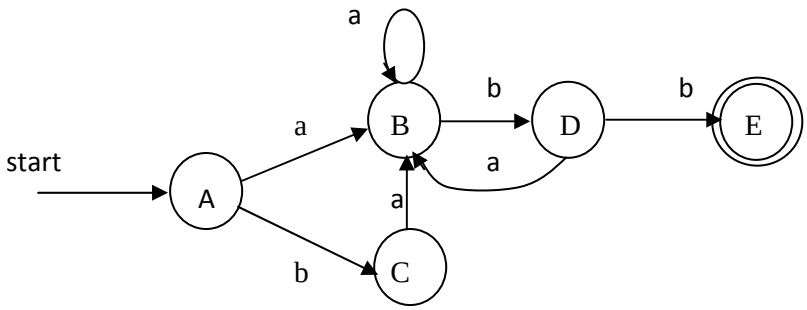
We should now examine state C

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 move(C,a) The set is {3,8}:
move(C,a) = {3,8}

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

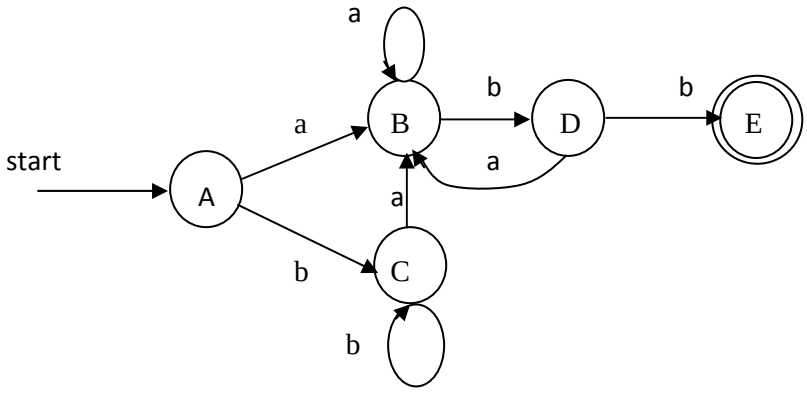
A={0,1,2,4,7}, B={1,2,3,4,6,7,8}, C={1,2,4,5,6,7}, D={1,2,4,5,6,7,9}, E={1,2,4,5,6,7,10}



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 move(C,b) The set is {5}: **move(C,b)** = {5}

o. now do an ε-closure on **move(C,b)**. Find all the states in **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 ε-closure(move(C,b)) = **C**
This defines a loop on C



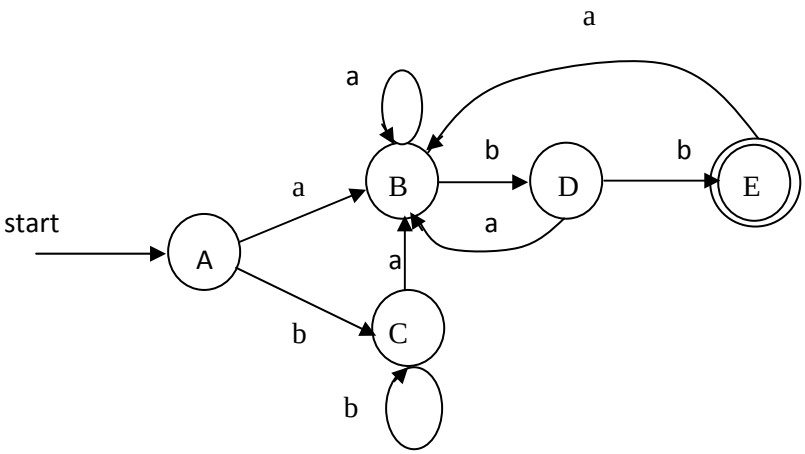
Finally, we need to look at E. Although this is an accepting state, the regular expression allows us to repeat adding in more a's and b's as long as we return to the accepting E state finally. So

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 move(E,a) The set is {3,8}:

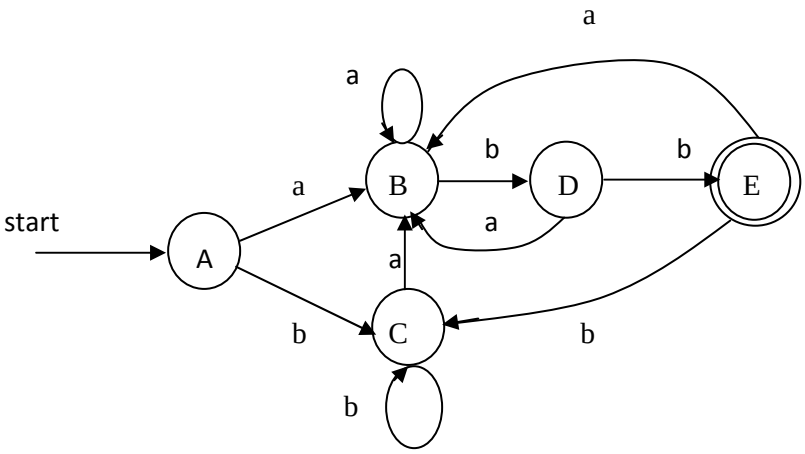
move(E,a) = {3,8}

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 move(E,b) The set is {5}: **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,

a. move(p,w) = s and s belongs to F and move(q,w) = t and t also belongs to F, where F is a set of final states
b. move(p,w) = a and a does not belongs to F and move(q,w) = b and b also does not belongs to F, where F is a set of final states.

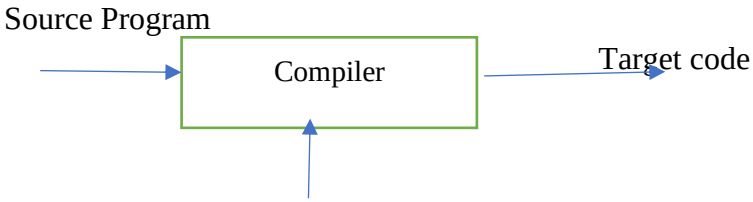
Suppose p, q are two different states and are said to be non-equivalent or distinguishable iff,

a. move(p,w) = n and n belongs to F and move(q,w) = m and m does not belongs to F, where F is a set of final states

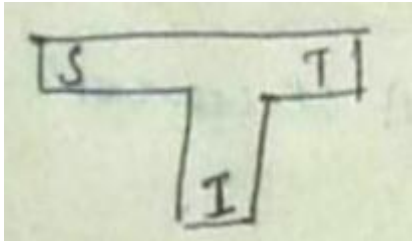
Examples in separate PDF file

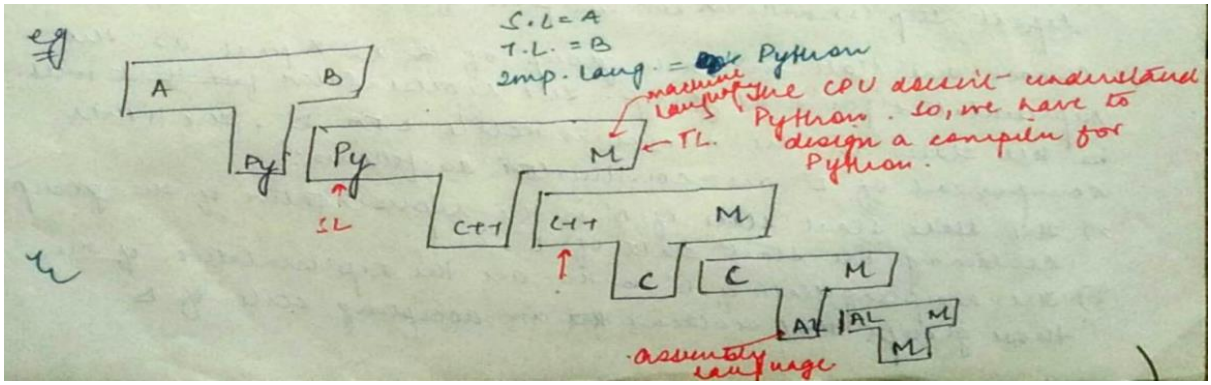
Bootstraping

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



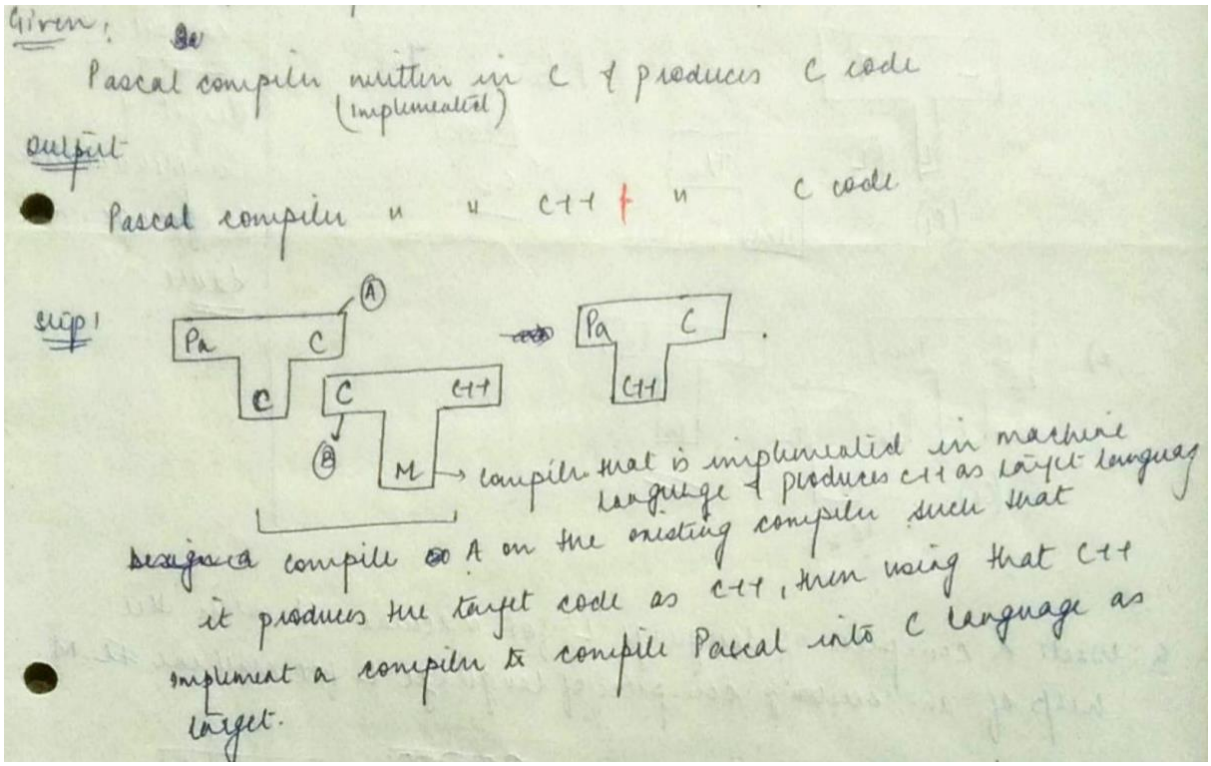
Compilation process can be represented using T-diagram:





Types of Compilers

- **Self-hosted compiler**s: 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 → αB|β

A,B € V

α, β € T

A → αB|β [Right linear]

|A| = |B| = 1

A → Bα |β [Left Linear]

Eg.

i. A→ aB|a
B→ aB|bB|a|b
Right Linear

ii. A→ Ba|a
B→ Ba
B → bB |a|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→ α, |A| = 1 and A € V , α € (V U T)*
Eg.
A → aAb | ab | ε
A == > aabb
A ==> 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. $|\alpha| <= |\beta|$
   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

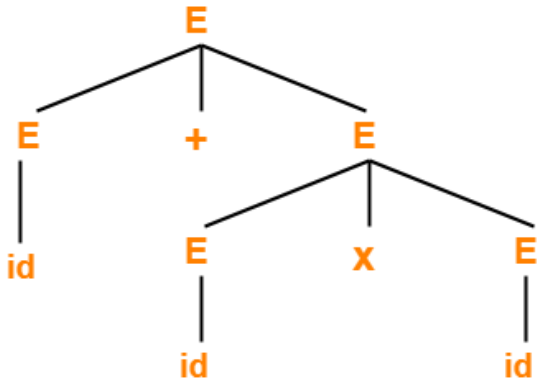
Context-Free Grammar

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.

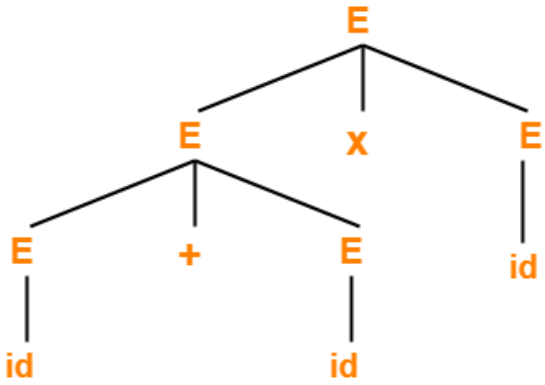
   E →E+E | E*E | id, w = id + id * id

   Left **M**ost **D**erivation
   E ==> E + E
   E ==> id + E
   E ==> id + E * E
   E ==> id + id * E
   E ==> id + id * id

   Right **M**ost **D**erivation
   E==> E + E
   E==> E + E * E
   E ==> E + E * id
   E ==> E + id * id
   E ==> id + id * id



Parse Tree-01



Parse Tree-02

id + id x id
2 + (3 x 4) = 14
20

id + id x id
(2 + 3) x 4 =

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

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

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 → aS|Sa|a, w = aa

ii. S → aSbS|bSaS|ε, w= abab

iii. R → R+R|RR|R*|a|b|c, w= a + bc

<u>Disambiguity rules:</u>

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

E →E+E
E→ E*E
E→ id

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

E → E + id
E→ id

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 → **E** + id
E → id

E → id + E
E→ 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 → id + **E**
E → id

E → E + T | T
T→ T * F | F
F→ id

w = id + id * id or id * id + id

E ==> E + T
E ==> T + T
E ==> F + T
E ==> id + T
E ==> id + T * F
E ==> id + F * F
E ==> id + id * F
E ==> id + id * id

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

E → E + T | T
T → T * F | F
F → G ** F | G
G → id

E → T + E | T
T → T * F | F
F → F ** G | G
G → id

2. **Left-recursive & Right-recursive Grammar**
   If the grammar is left recursive, it takes the form A → Aα | β
   If the grammar is right recursive, it takes the form A → αA | β

Problem with left recursive grammar,

A → Aα | β
A==> Aα
A==>Aαα
A==> Aααα
.
.
.
A ==> βα*

A(){

β;

α ➔ Infinite Loop

}

A → αA | β
A==> α* β

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. □

```
1)  arrange the nonterminals in some order A1, A2, ..., An.
2)  for ( each i from 1 to n ) {
3)      for ( each j from 1 to i - 1 ) {
4)          replace each production of the form Ai → Aj γ by the
              productions Ai → δ1 γ | δ2 γ | ··· | δk γ, where
              Aj → δ1 | δ2 | ··· | δk are all current Aj-productions
5)      }
6)      eliminate the immediate left recursion among the Ai-productions
7)  }
```

A → Aα | β
A → βA'
A' → ε | α A'

Q1. E → E + T | T
A: E
α: +T
β: T

E → TE'
E'→ ε | +TE'

Q2. S→ S0S1S
S → 01

S → 01S'
S' → ε | 0S1SS'

Q3. S→ (L) | S
L→ L,S|S

Q4. S→AaB
A→ Aa|bB|a
B→ bB|c

Q5.S → Aa|b
A→Ac|Sd| ε

Q6. S→ AaB
A→ SAc|a
B→Ba|b

3. **Deterministic and Non-Deterministic Grammar**

A → $\alpha\beta_1$ | $\alpha\beta_2$ | $\alpha\beta_3$ …
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 → $\alpha\beta_1$ | $\alpha\beta_2$ | $\alpha\beta_3$

A → αA'
A' → β_1 | β_2 | β_3

Q1. S→ **iEt**S | **iEt**SeS |a
E→ b

S→ **iEt**S'| a
S'→ S | SeS
E → b

Q2. S→ **aS**SbS | **aS**aSb | abb | b

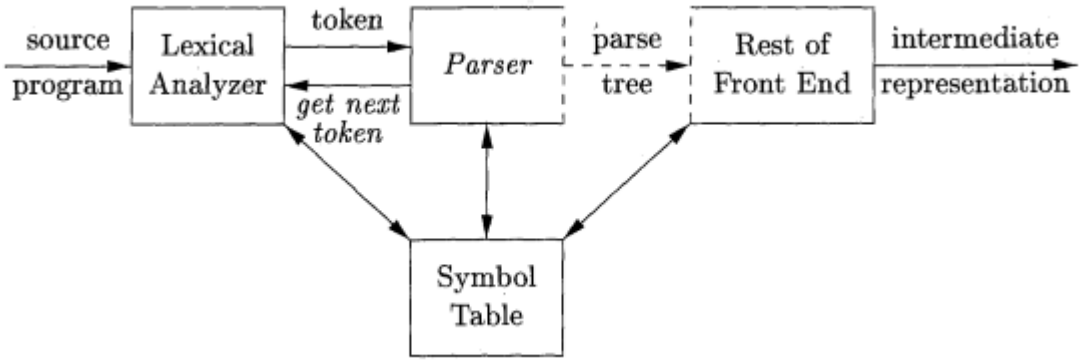
Taking aS as common prefix:
S → aSS' | abb | b
S'→ SbS | aSb

Taking a as common prefix
S→ **a**SSbS | **a**SaSb | abb | b
S → aS' | b
S' → **S**SbS | **S**aSb | bb

S' → SS'' | bb
S'' → SbS |aSb

Syntax Analysis

Role of the Parser



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

   E → E + T | T
   T→ T * F | F
   F→ id

   w = id + id * id or id * id + id or *id id ++

c. report any syntax errors in an intelligible fashion
d. 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.

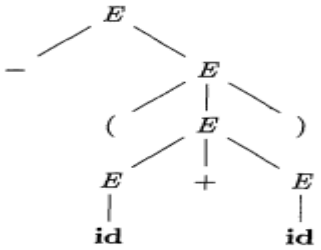
$$E \rightarrow E + E \mid E * E \mid - E \mid (E) \mid \mathbf{id}$$

w = -(id+id)

Derivation

E ==> -E
E ==> -(E)
E ==> -(E+E)
E ==> -(id + E)
E==> -(id + 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)

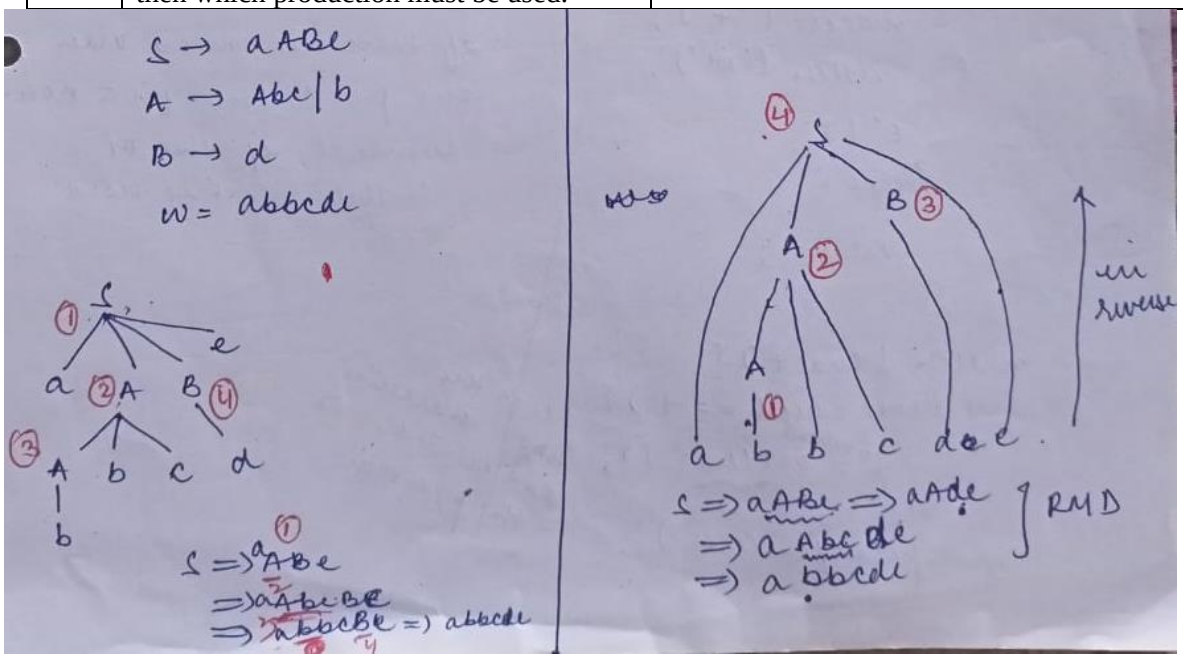
Bottom-up Parsers

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 → 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→ aABe A→Abc|b B→d w = abbcde S ==> aABe S ==> aAbcBe S ==> abbcBe S ==> abbcde	S→ aABe A→Abc|b B→d w = abbcde S ==> aABe S ==> aAde S ==> aAbcde S ==> 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→ abC | aBd | aAD

B → bB |ε

C → d | ε

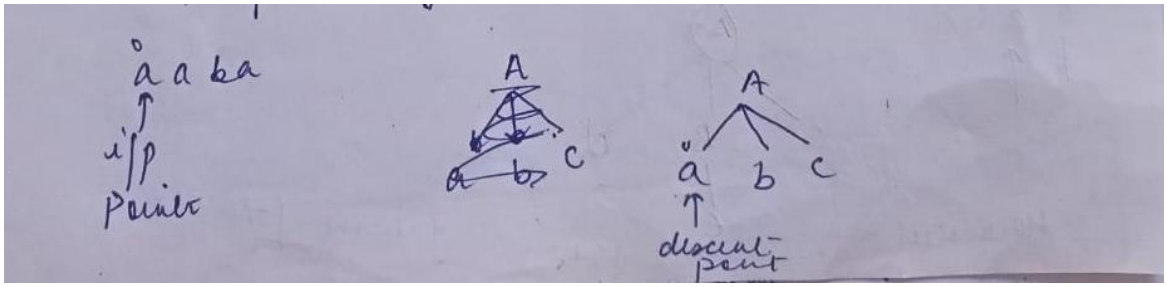
D → a | b | ε

Input String: aaba

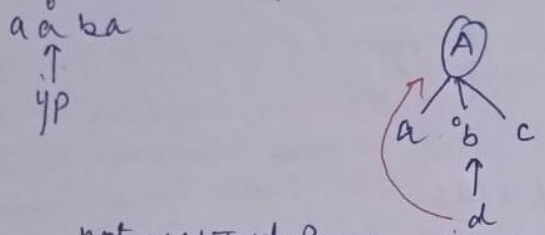
Two types of pointers are maintained:

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

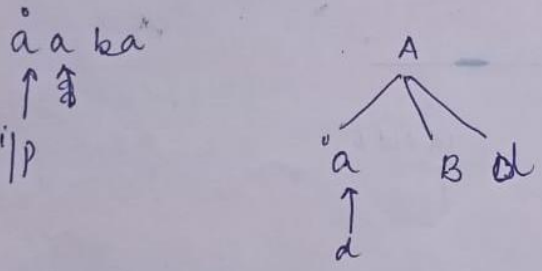
Descent pointer: points to the current symbol in the parse tree.



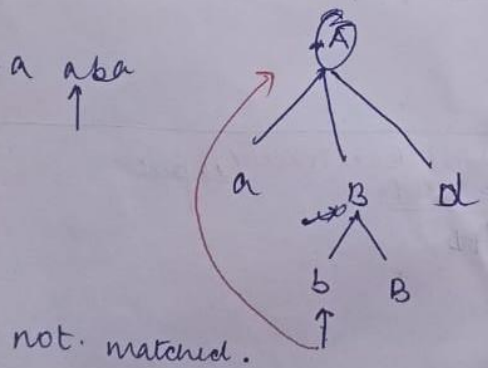
increment i/p pointer & descent pointer



not matched. Parser will back track.

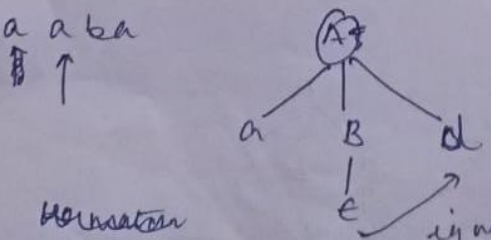


matches



not matched.

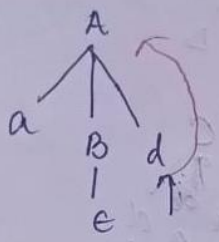
again Backtracking



Derivation

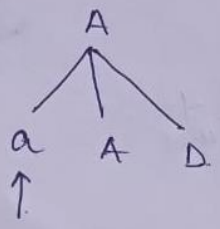
move ε & move forward

a a ba



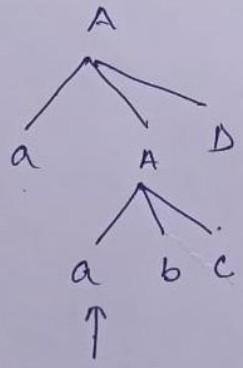
no match. Again backtrack.

a a ba



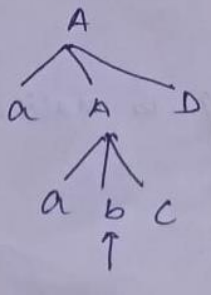
matches. Advance forward.

a a ba



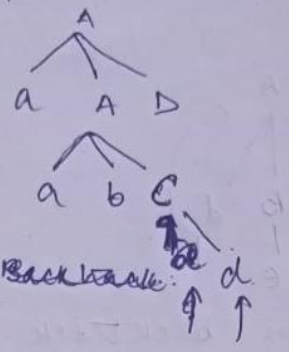
matches. Advance forward.

a a b a



matches

a a b a

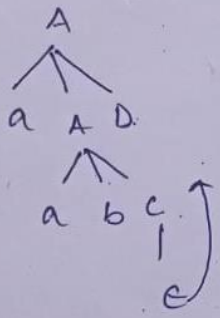


~~not matched~~ · ~~Backtrack~~

a a ba

Backtrack ↑

no match. ∴ not 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 → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi 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→ ε
 }
}

T'(){
  If( input == '*'){
      input ++ ;
     F();
     T'();
  }else{
 return ; // T' → ε
  }
}

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

$ ➔ input end marker ; id + id * id$

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

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

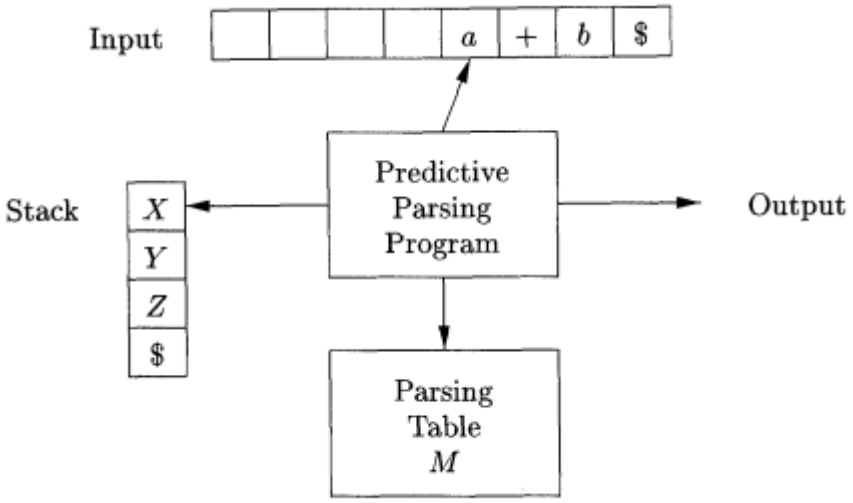
Q. Construct the recursive descent parser for

E → iE'
E' → +iE' | ε
w = i +i$

S → cAd
A→ aA|D
D→d
w = caadd$

A → abC | Abd |aAD
B→ bB | ε
C → d | ε
D → a | b | ε
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 → aABCD
A → b
B → c
C → d
D → e

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

Eg 2:

S → ABCD
A → b
B → c
C → d
D → e

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

Eg 3:

S → ABCD
A → b | f
B → c
C → d
D → e

FIRST(X)	**Set of terminals**
FIRST(S) = FIRST(ABCD) = FIRST(A) = FIRST(b) U FIRST(f)	{b, f}
FIRST(A) = FIRST(b) U FIRST(f)	{b, f}
FIRST(B) = FIRST(c)	{c}
FIRST(C) = FIRST(d)	{d}
FIRST(D) = FIRST(e)	{e}

Eg 4:

S → ABCD
A → b | f | ε
B → c
C → d
D → e

w = cde$
S ==> ABCD ==> BCD

FIRST(X)	**Set of terminals**
FIRST(S) = FIRST(ABCD) = FIRST(A) = FIRST(b) U FIRST(f) U FIRST(B)	{b, f, c}
FIRST(A) = FIRST(b) U FIRST(f) U FIRST(ε)	{b, f, ε}
FIRST(B) = FIRST(c)	{c}
FIRST(C) = FIRST(d)	{d}
FIRST(D) = FIRST(e)	{e}

Eg 5:

S → ABCD
A → b | ε
B → c | ε
C → d
D → e

FIRST(X)	**Set of terminals**
FIRST(S) = FIRST(ABCD) = FIRST(A) = FIRST(b) U FIRST(B)	{b, c, d}
FIRST(A) = FIRST(b) U FIRST(ε)	{b, ε}
FIRST(B) = FIRST(c) U FIRST(ε)	{c, ε}
FIRST(C) = FIRST(d)	{d}
FIRST(D) = FIRST(e)	{e}

Eg 6:

S → ABCD
A → b | ε
B → c | ε
C → d | ε
D → e | ε

FIRST(X)	**Set of terminals**
FIRST(S) = FIRST(ABCD) = FIRST(A) = FIRST(b) U FIRST(B) U FIRST(C) U FIRST(D)	{b, c, d, e, ε}
FIRST(A) = FIRST(b) U FIRST(ε)	{b, ε}
FIRST(B) = FIRST(c) U FIRST(ε)	{c, ε}
FIRST(C) = FIRST(d) U FIRST(ε)	{d, ε}
FIRST(D) = FIRST(e) U FIRST(ε)	{e, ε}

Rules for finding FIRST(X)

1. If X is a terminal, then $\text{FIRST}(X) = \{X\}$.
2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ 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 $\text{FIRST}(Y_1), \ldots, \text{FIRST}(Y_{i-1})$; that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in $\text{FIRST}(Y_j)$ for all $j = 1, 2, \ldots, k$, 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 $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add $\text{FIRST}(Y_2)$, and so on.
3. If $X \to \epsilon$ is a production, then add ϵ to $\text{FIRST}(X)$.

S → ABCD
A → b
B → c
C → d
D → e

FIRST(X)	**Set of Terminals**
FIRST(S) = FIRST(A)	b
FIRST(A) = FIRST(b)	b
FIRST(B)	c
FIRST(C)	d
FIRST(D)	e

S → ABCD
A → b | ε
B → c | ε
C → d | ε
D → e | ε

FIRST(X)	**Set of Terminals**
FIRST(S) = FIRST(A) U FIRST(B) U FIRST(C) U FIRST(D)	b, c, d, e, ε
FIRST(A)	b, ε
FIRST(B)	c, ε
FIRST(C)	d, ε
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 → ABCD
A → a
B → b
C → c
D → d

w = abcd$

S==> ABCD$
S==> aBCD$
S==> abCD$
S==> abcD$
S==> 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 → ABCD
A → b | ε
B → c
C → d
D → 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 → A
A → BC
B → c
C → 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 $A \to \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where FIRST(β) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

S → ε.XYZ

A → αBβ

Rule 2: A → αBβ
FOLLOW(X)
A: S
B:X
α: ε
β: YZ
FIRST(β)

Rule 3: A → αB
FOLLOW(Z)
A: S
B: Z

or

A → αBβ

S → ABCD
A → b | ε
B → c | ε
C → d | ε
D → e | ε

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

Eg 4:

S → ABCD
A → a | ε
B → CD | b
C → c | ε
D → Aa | d | ε

FIRST(X)	**Set of terminals**
FIRST(S) = FIRST(ABCD) =FIRST(A) U FIRST(B) U FIRST(C) U FIRST(D)	{a, b, c, d, ε}
FIRST(A) = FIRST(a) U FIRST(ε)	{a, ε}
FIRST(B) = FIRST(CD) U FIRST(b)	{a, b, c, d, ε}
FIRST(C) = FIRST(c) U FIRST(ε)	{c, ε}
FIRST(D) = FIRST(Aa) U FIRST(d) U FIRST(ε)	{a, d, ε}

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

Eg. 5

S → ABCDE
A → a | ε
B → b | ε
C → c | ε
D → d | ε
E → e | ε

Eg. 6

S → Bb | Cd
B → aB | ε
C → cC | ε

Eg. 7

S → ABCDE
A → a | ε
B → b | ε
C → c
D → d | ε
E → e | ε

Eg. 8:

S → ACB | CbB | Ba
A → da | BC
B → g | ε
C → h | ε

Eg 9:

S → aABb
A → C | ε
B → d | ε

Eg 10:

S → aBDh
B→ cC
C → bC | ε
D → EF
E → g | ε
F → f | ε

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 $A \to \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \to \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \to \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \to \alpha$ 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). □

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

Eg.

E → E+T | T
T → T*F| F
F → id

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

FIRST(X)	**Set of terminals**
FIRST(E) = FIRST(TE’)	{id, (}
FIRST(E’) = FIRST(+TE’) U FIRST(ε)	{+, ε}
FIRST(T) = FIRST(FT’)	{id, (}
FIRST(T’) = FIRST(*FT’) U FIRST(ε)	{*, ε}
FIRST(F) = FIRST (id) U FIRST((E))	{id, (}

FOLLOW(X)	**Set of terminals**
FOLLOW(E) = FIRST ())	{), $}
FOLLOW(E’) = FOLLOW(E) U FOLLOW(E’)	{), $}
FOLLOW(T) = FIRST(E’) U FOLLOW(E) U FOLLOW(E’)	{+,), $}
FOLLOW(T’) = FOLLOW(T) U FOLLOW(T’)	{+,), $}
FOLLOW(F) = FIRST(T’) U FOLLOW(T) U FOLLOW(T’)	{*, +,), $}

a) E → TE’
FIRST(E) =FIRST(TE’) = {id, (}
M[E,id] = E → TE’
M[E, (] = E→ TE’

b) E’ → +TE’ | ε
i) E’ → +TE’

FIRST(E) = FIRST(+TE’) = {+}
M[E’, +] = E’ → +TE’

ii) E’ → ε
FIRST(E’) = FIRST(ε) = {ε}
Since FIRST(RHS) contains ε, therefore we must find FOLLOW(LHS),
FOLLOW(E’) = {), $}

M[E’,)] = E’ → ε
M[E’,$] = E’ → ε

c) T → FT’
FIRST(T) = FIRST(FT’) = {id, (}
M[T,id] = T → FT’
M[T,(] = T → FT’

d) T’ → *FT’ | ε
i) T’ → *FT’
FIRST(T’) = FIRST(*FT’) = {*}
M[T’,*] = T’ → *FT’

ii) T' → ε

FIRST(T') = FIRST(ε) = {ε}

FOLLOW(T') = {+,), $}

M[T', +] = T' → ε

M[T',)] = T' → ε

M[T', $] = T' → ε

e) F → id | (E)

i) F → id

FIRST(F) = FIRST(id) = {id}

M[F,id] = F → id

ii) F → (E)

FIRST(F) = FIRST((E)) = {(}

M[F,(] = F → (E)

Non-Terminals/ Input Symbols	id	+	*	(	)	$
E	E → TE'	***error***	***error***	E → TE'	***error***	***error***
E'	***error***	E' → +TE'	***error***	***error***	E' → ε	E' → ε
T	T → FT'	***error***	***error***	T → FT'	***error***	***error***
T'	***error***	T' → ε	T' → *FT'	***error***	T' → ε	T' → ε
F	F → id F→idE	***error***	***error***	F→ (E)	***error***	***error***

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. □

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* 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 → Y1 Y2 ··· Yk ) {
                output the production X → Y1 Y2 ··· Yk;
                pop the stack;
                push Yk, Yk−1, ... , Y1 onto the stack, with Y1 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 → TE'
	FT'E$	id+id$	O/P: T → FT'
	idT'E'$	id+id$	O/P: F → id
id	T'E'$	+id$	Matched: id
id	E'$	+id$	O/P: T → ε
id	+TE'$	+id$	O/P: E' → +TE'
id+	TE'$	id$	Matched: +
id+	FT'E'$	id$	O/P: T→FT'
id+	idT'E'$	id$	O/P: F → id
id+id	T'E'$	$	Matched: id
id+id	E'$	$	O/P: T' → ε
id+id	$	$	Matched; Parsing completed successfully

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

<u>More questions on Predictive Parsing</u>

1. S→ aBDh
   B→Bb | C
   D →EF
   E → g | ε
   F → f | ε w = acgf

2. S → iLuET | a
   L → LS | ε
   E → b
   T → d | e| ε w = iubde

3. A → BCc | fDB
B → bCDE | ε
C → DaB | ca
D → dB | ε
E → Eaf | c w = gdbca

LL(1) Grammar

A grammar is said to be LL(1) iff

a. $A \rightarrow \alpha_1 \mid \alpha_2$, $FIRST(\alpha_1) \Lambda FIRST(\alpha_2) = \Phi$
b. $A \rightarrow \alpha \mid \varepsilon$, $FIRST(\alpha) \Lambda FOLLOW(A) = \Phi$

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 → AB
A → a | ε
B → b | ε

FIRST(a) Λ FOLLOW(A)
{a} Λ {b, $} = Φ
Grammar is LL(1)

Eg. 2

S → A | a
A → a

$FIRST(\alpha_1) \Lambda FIRST(\alpha_2) = \Phi$
{a} Λ {a} ≠ Φ
Grammar is not LL(1)

Eg. 3

S → iEtS | iEtSeS | a
E → b

After performing left-factoring,
S → iEtSS' | a
S' → eS | ε
E → 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→a			S→ iEtSS’		
S’			S’ → eS S’ → ε			S’ → ε
E		E → b				

Grammar is not LL(1)

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

1. S → aAB | bA | ε
2. A → aAb | ε
3. B → bB | c

Bottom-up Parser

a. 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).
b. 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 → aABe
A → Abc | b
B → d

w = abbcde

→ : used in productions/grammar
➔ : Derivation

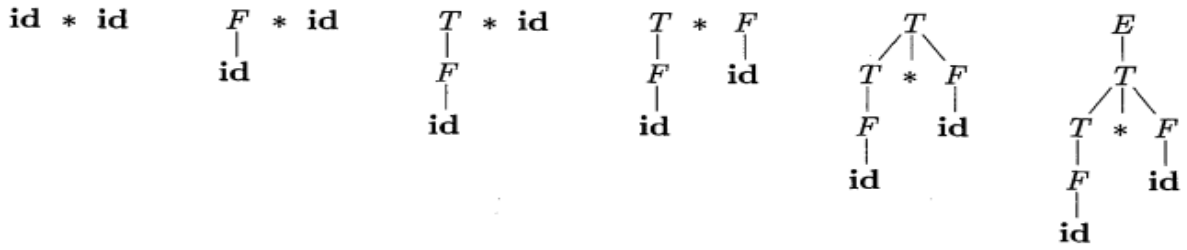
S ➔ aABe ➔ aAde ➔ aAbcde ➔ abbcde

LHS to RHS: expansion
RHS to LHS: reduction

Eg. 2

E→ E+T | T
T → T*F | F
F → id

E ➔ T ➔ T * F ➔ T * id ➔ F * id ➔ 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
$\mathbf{id}_1 * \mathbf{id}_2$	$\mathbf{id}_1$	$F \to \mathbf{id}$
$F * \mathbf{id}_2$	F	$T \to F$
$T * \mathbf{id}_2$	$\mathbf{id}_2$	$F \to \mathbf{id}$
$T * F$	$T * F$	$E \to T * F$

Handles during a parse of $\mathbf{id}_1 * \mathbf{id}_2$

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

E→ E+T | T
T → T*F | F
F → id

Right Sentential Form	Handle	Reducing production
id + id * id	id	F→id
F + id * id	F	T → F
T + id * id	T	E → T
E + id * id	id	F → id
E + F * id	F	T → F
E + T * id	id	F → id
E + T * F	T * F	T → T * F
E + T	E + T	E → E + T

Shift Reduce Parsing

a. Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols, and an input buffer holds the rest of the string to be parsed.
b. Handle is always at the top of the stack just before it is identified as handle.
c. $: marks the bottom of the stack and right end of the string.
d. Possible action performed by SR Parser:
    i. **Shift**: Next the symbol is shifted/pushed on top of the stack.
    ii. **Reduce**: handle that appear on the top of the stack is replaced with the appropriate non-terminal.
    iii. **Accept**: denotes the successful completion of parsing.
    iv. **Error**: a situation in which parser cannot either shift or reduce the symbol and cannot perform accept operation.

Eg. 1. Let us consider **desk calculator** grammar:

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

w = id * id$

Stack	Input Buffer	Action
$	id * id$	Shift: id
$id	* id$	Reduce: F → id
$F	* id$	Reduce: T → F
$T	* id$	Shift: *
$T *	id$	Shift: id
$ T * id	$	Reduce: F → id
$T * F	$	Reduce: T → T * F
$T	$	Reduce: E → T
$E	$	Accept

Shift-Reduce Conflict: If the parser has the choice to perform shift as well as reduce action.

Reduce-Reduce Conflict: A handle can be replaced by more than one production then the conflict arises which production can be used for reduction.

Note: Grammars in which such conflicts do not arise are known as LR Grammar else non-LR Grammar.

LR
L: Left to right
R: RMD

Eg. 2

S → (L) | a
L → L,S | S

w = (a, (a, a))

Stack	Input Buffer	Action
$	(a, (a, a)) $	Shift: (
$(	a,(a,a))$	Shift: a
$(a	,(a,a))$	Reduce: S → a
$(S	,(a,a))$	Reduce: L → S
$(L	,(a,a))$	Shift:,
$(L,	(a,a))$	Shift:(
$(L,(	a,a))$	Shift: a
$(L,(a	,a))$	Reduce: S → a
$(L,(S	,a))$	Reduce L → S
$(L,(L	,a))$	Shift:,
$(L,(L,	a))$	Shift: a
$(L,(L,a	))$	Reduce: S → a
$(L,(L,S	))$	Reduce: L → L,S
$(L,(L	))$	Shift:)
$(L,(L)	)$	Reduce: S→ (L)
$(L,S	)$	Reduce: L→ L,S
$(L	)$	Shift:)
$(L)	$	Reduce: S→ (L)
$S	$	Accept

More questions on SR Parser

1. S → TL
   T → int | float
   L → L, id, id

   w = int id, id

2. E → E - E
   E → E + E
   E → id
   w = id -id + id

LR Parsers/LR(k) Parser

L: left to right scanning
R: RMD in reverse
k: number of input symbols of lookahead that are used in parsing decision.

k<=1 is only in syllabus
if k =1, then we can omit k. for eg, LR(0), SLR(1) parser can be called as SLR parser

LR(k) parsers:

a. LR (0) Parser
b. SLR(1) Parser: Simple LR(1) Parser
c. CLR(1) Parser: Canonical LR(1) Parser
d. LALR(1) Parser: Look Ahead LR(1) Parser

LR parsers are table-driven.
The grammar for which parsing table for LR parsing can be constructed is said to be LR Grammar.

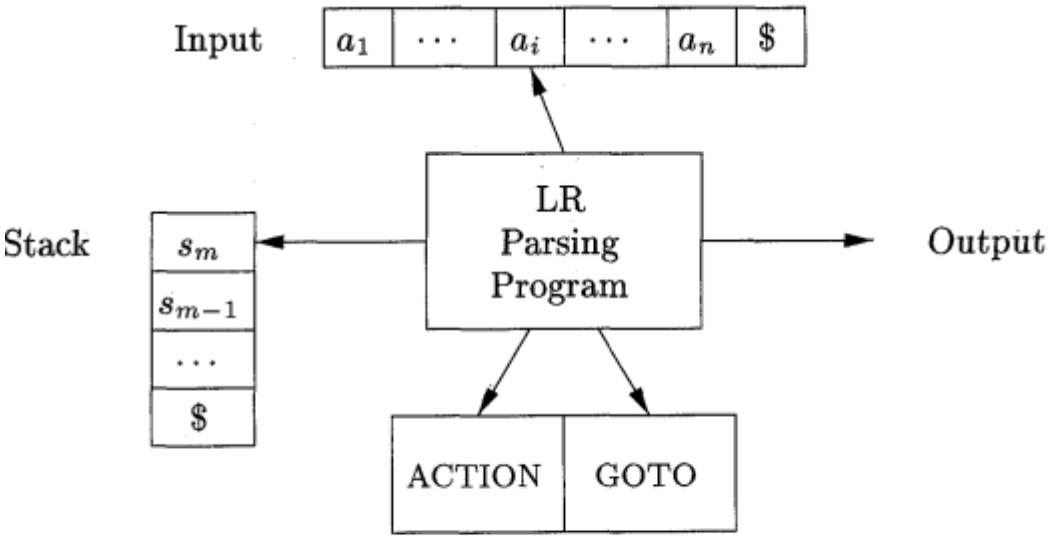


Figure 4.35: Model of an LR parser

Parsing actions can be

a. Shift
b. Action
c. Error
d. Accept

Items and the LR(0) Automaton

a. An LR parser makes shift-reduce decisions by maintaining states to keep track of where we are in a parse.
b. States represents the set of items.
c. An *LR(0)* item (item for short) of a grammar G is a production of G with a dot at some position of the body. Thus, production $A \rightarrow XYZ$ yields the four items

   A → .XYZ, means the string generated by XYZ is still to be parsed
   A → X.YZ, means the string generated by X is parsed and YZ is still to be parsed
   A → XY.Z, means the string generated by XY is parsed and Z is still to be parsed

A → XYZ., means the string has been parsed and now XYZ can be reduced to A

A → ε. will generate only one item, A → .

LR(0) collection of items and items means the same

Construction of LR(0) Collection of items

1. Define the augmented grammar.
2. Define the functions:CLOSURE & GOTO

Augmented Grammar

If G is grammar with start symbol S, then G' is the augmented grammar for G such that G' with the start symbol S' and the production S' → S[S' → S or E' → S or R' → S]

G: Grammar which has the S as start symbol
G': Augmented Grammar having the start symbol S'/E'/A' etc
G:
S → A
A → Abc

G':
S' → S: Augmented Production
S→A
A → Abc

Closure of Item Sets

If I is a set of items for a grammar G, then CLOSURE(I) is the set of items constructed from I by the two rules:

1. Initially, add every item in I to CLOSURE(I).
2. If $A \to \alpha \cdot B\beta$ is in CLOSURE(I) and $B \to \gamma$ is a production, then add the item $B \to \cdot\gamma$ to CLOSURE(I), if it is not already there. Apply this rule until no more new items can be added to CLOSURE(I).

```
SetOfItems CLOSURE(I) {
        J = I;
        repeat
                for ( each item A → α·Bβ in J )
                        for ( each production B → γ of G )
                                if ( B → ·γ is not in J )
                                        add B → ·γ to J;
        until no more items are added to J on one round;
        return J;
}
```

For eg.
Eg. 1

Set of Items, I:
S' → S
S→A
A → Abc | a

Closure(I):

S' → .S
S → .A
A → .Abc
A → .a

Eg. 2:

G:
E→ E +T | T
T → T * F | F
F → (E) | id

G':

S' → E
E→ E +T | T
T → T * F | F
F → (E) | id

Closure(I):
S' → .E
E → .E+T
E → .T
T → .T * F
T → .F
F → .(E)
F → .id

GOTO Function:

GOTO(I,X):

I: Set of items
X: Grammar symbol

GOTO[I,X]: Closure of the set of all [A→ α.Xβ] such that [A→ α.Xβ] is in I. GOTO function is used to define the transition in the LR(0) automaton for a grammar. The states of the automaton correspond to sets of items, and GOTO(I,X) specifies the transition from the state for I under input X.

I_0: Closure(I) S' → .E E → .E + T E → .T T→ .T*F T → .F F → .(E) F → .id	I_1:GOTO(I_0,E) S' → E. E → E.+T	I_2:GOTO(I_0,T) E→ T. T → T.*F	I_3:GOTO(I_0,F) T → F.	I_4:GOTO(I_0,() F → (.E) E → .E+T E →.T T→ .T*F T→ .F F → .(E) F→ .id
I_5:GOTO(I_0,id) F → id.	I_6:GOTO(I_1,+) E → E+.T T →.T*F T → .F F → .(E) F → .id	I_7:GOTO(I_2,*) T → T*.F F → .(E) F → id	I_8:GOTO(I_4,E) F → (E.) F → E.+T	GOTO(I_4,T) = I_2 E → T. T → T.*F