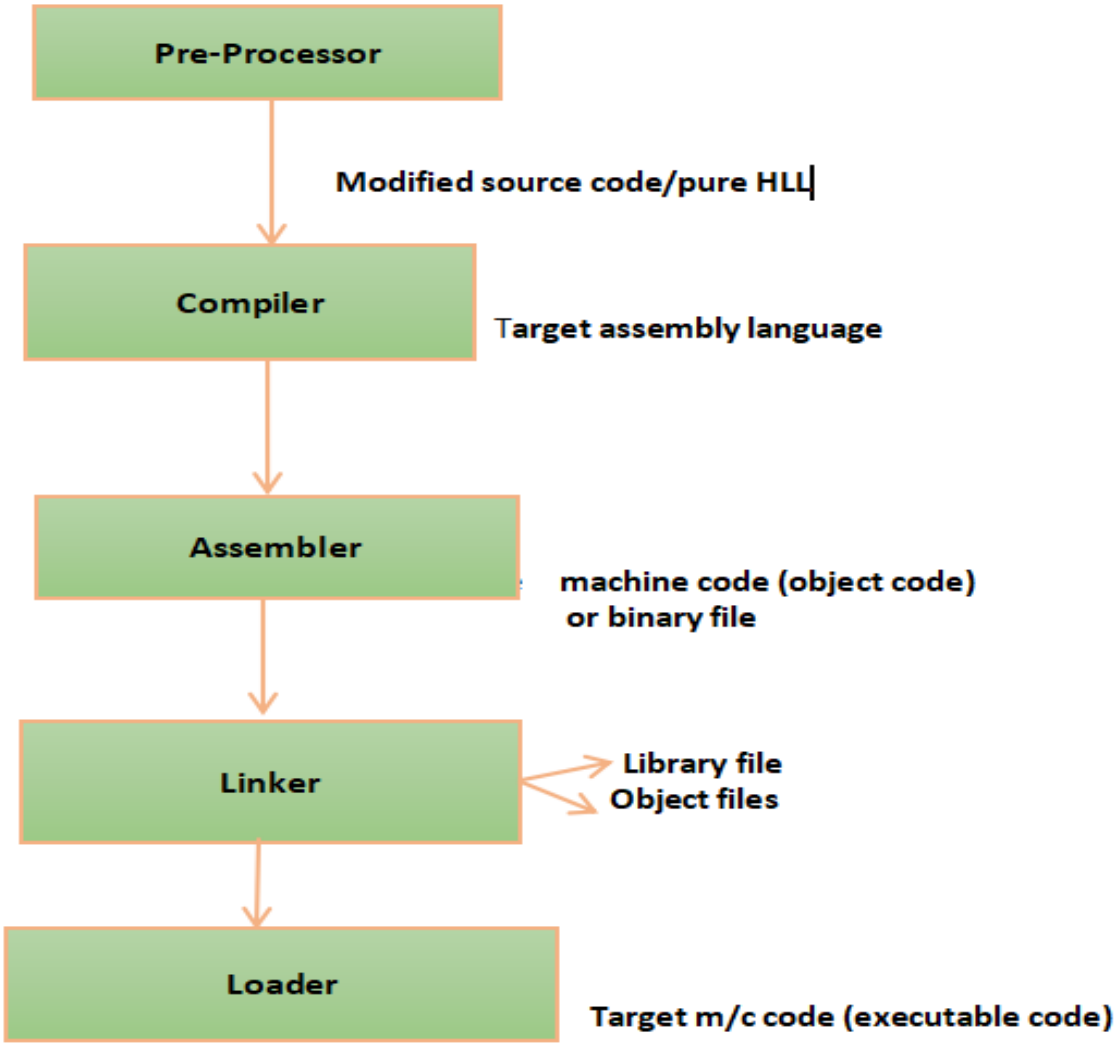
Chapter 5
Compilers: Analysis Phase

Syllabus

	Compilers: Analysis Phase
1	Introduction to compilers
2	Phases of compilers: Lexical Analysis- Role of Finite State Automata in Lexical Analysis
3	Design of Lexical analyser, data structures
4	Syntax Analysis- Role of Context Free Grammar in Syntax analysis,
5	Types of Parsers:
6	Top down parser- LL(1)
7	Bottom up parser- Operator precedence parser,
8	SLR
9	Semantic Analysis
9	Syntax directed definitions

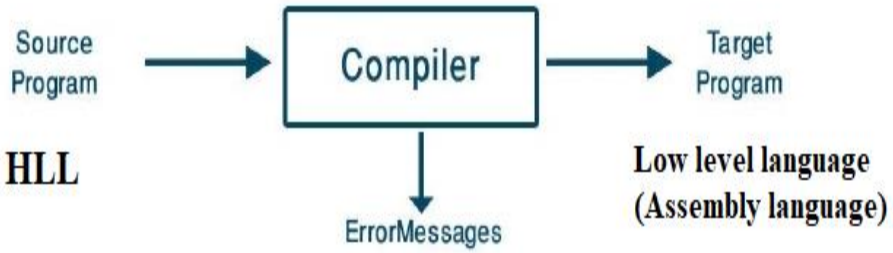
Background


Pre-Processor
Modified source code/pure HLL
Compiler
Target assembly language
Assembler
machine code (object code)
or binary file
Linker
Library file
Object files
Loader
Target m/c code (executable code)

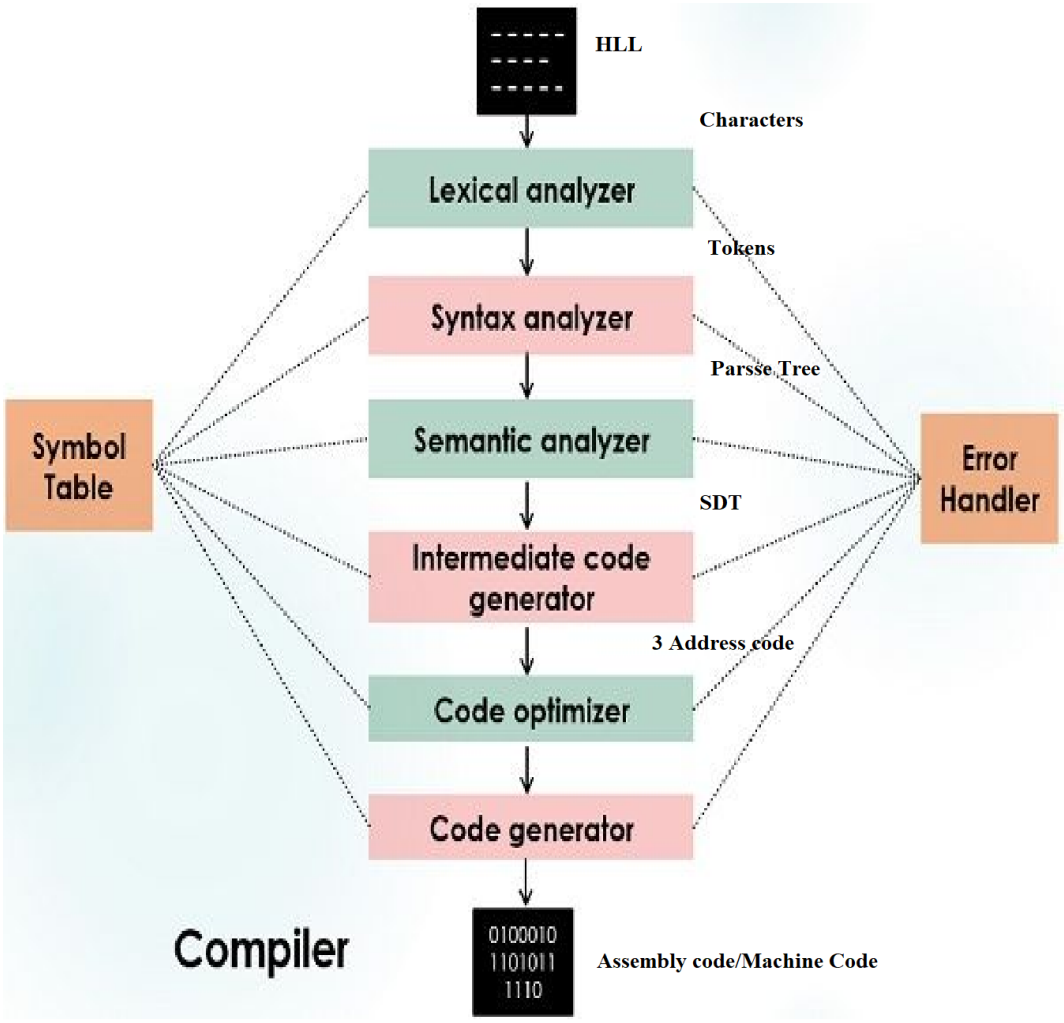
Introduction to Compilers

A compiler translates source code written in HLL to target code written in assembly language or low level language.

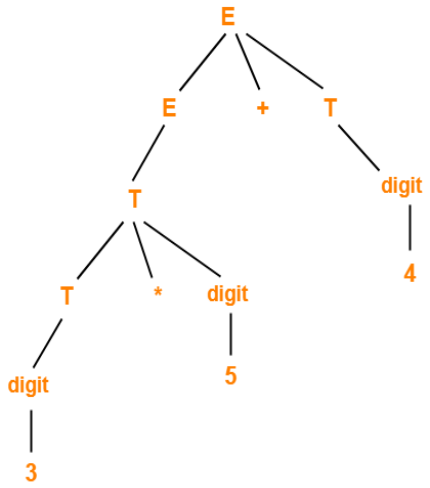
It reports the user about error(s) in source program if any.



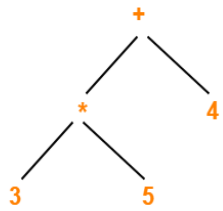
Phases of Compiler



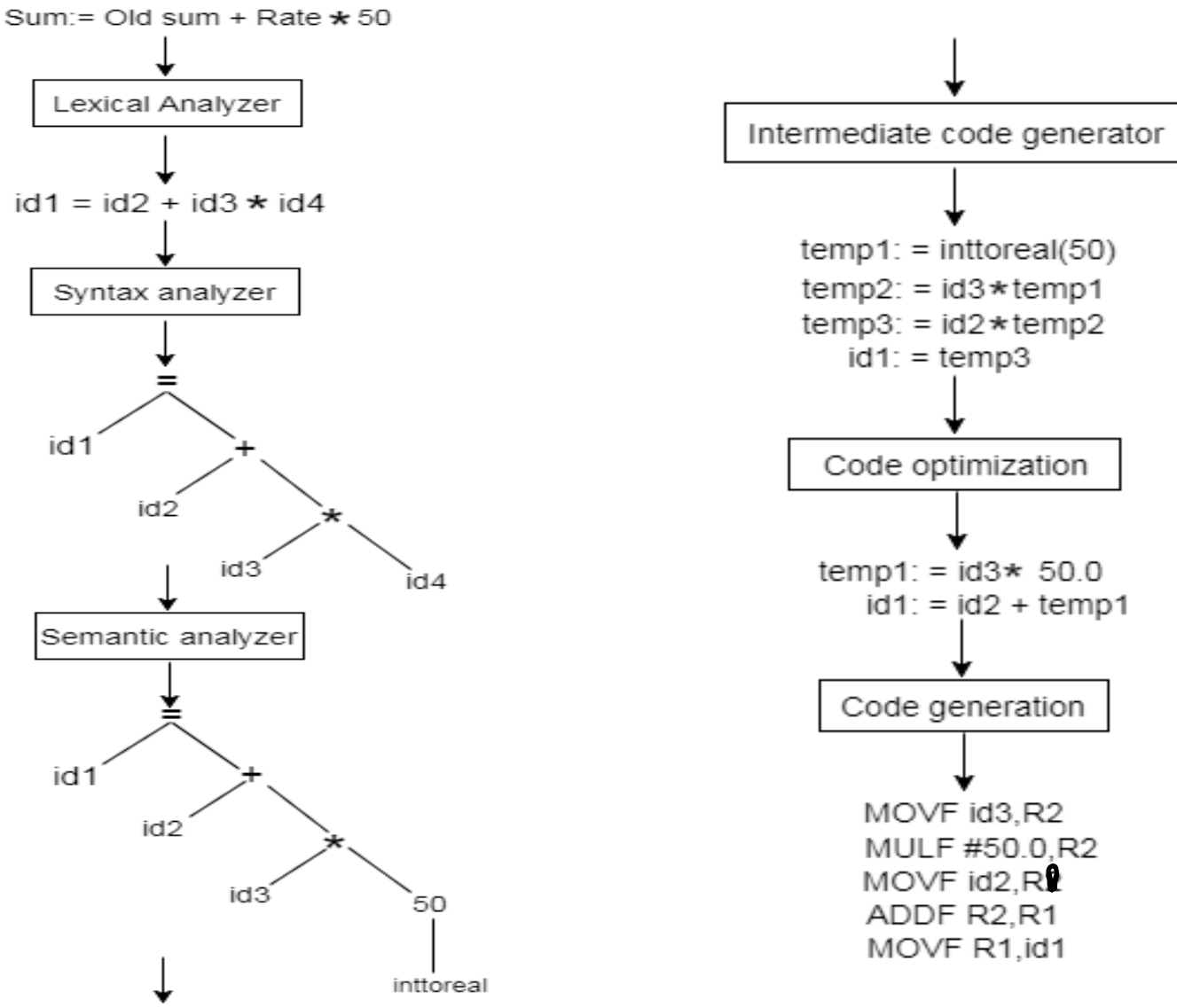
y=m*x+c



Parse Tree



Syntax Tree


Sum:= Old sum + Rate * 50
Lexical Analyzer
id1 = id2 + id3 * id4
Syntax analyzer
=
id1
+
id2
*
id3
id4
Semantic analyzer
=
id1
+
id2
*
id3
50
inttoreal
Intermediate code generator
temp1: = inttoreal(50)
temp2: = id3*temp1
temp3: = id2*temp2
id1: = temp3
Code optimization
temp1: = id3* 50.0
id1: = id2 + temp1
Code generation
MOVF id3,R2
MULF #50.0,R2
MOVF id2,R1
ADDF R2,R1
MOVF R1,id1

Lexical Analysis

LEXICAL ANALYSIS is the very first phase in the compiler designing.

It is also known as scanner

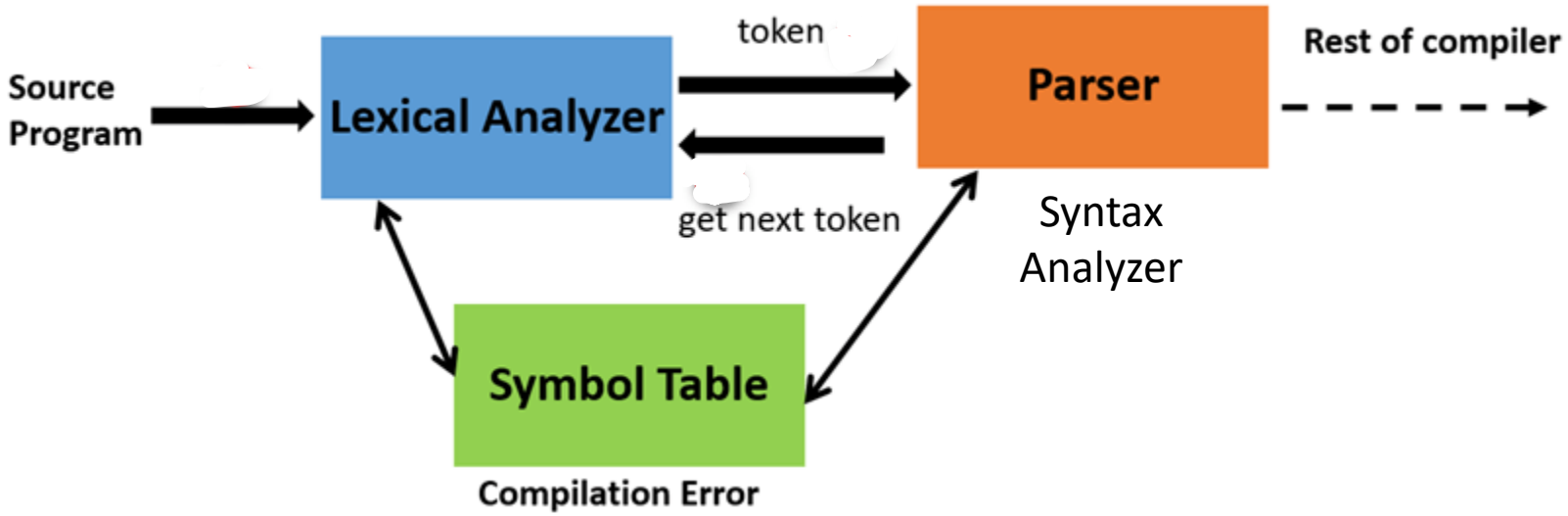
- keywords- lexer, tokenizer, scanner

➢ **Functions:**

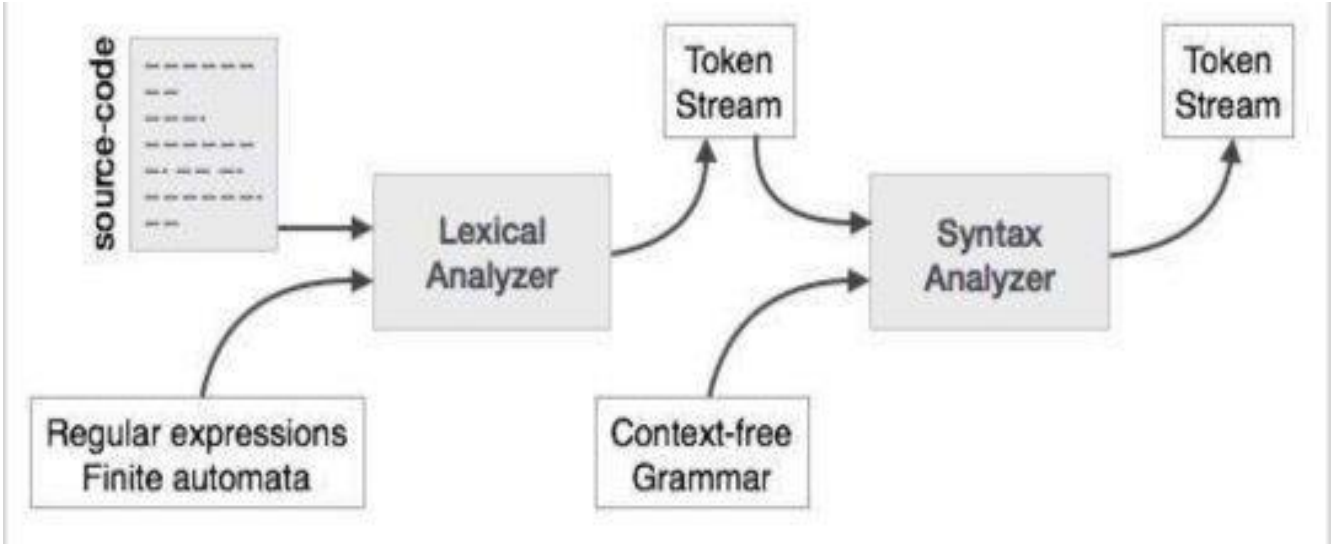
- A Lexer reads the source code which is written in the form of sentences
- It converts a sequence of characters into a sequence of tokens (**tokenization**)
- It removes any extra space or comment written in the source code (//, #)
- If the lexical analyzer detects that the token is invalid, it generates an error- ***lexical errors*** – *spelling error, exceeding length of identifier, appearance of illegal characters. It is found during execution of program*
- It reads character streams from the source code, checks for legal tokens, and pass the data to the syntax analyzer when it demands.
- Helps to identify tokens into the symbol table

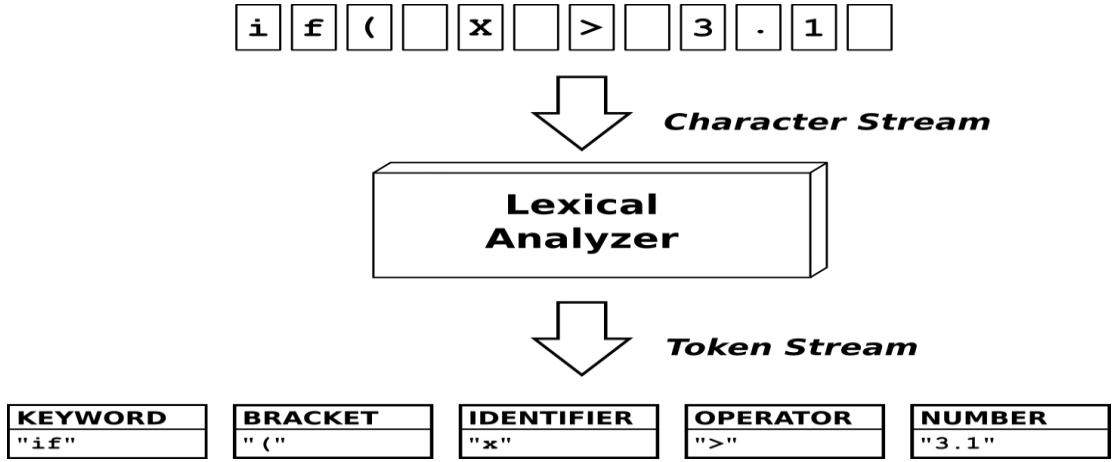
E.g.: ***Can be used by web browser to format a webpage with parse data from javascript, html and css.***

Lexical Analysis



- Lexical analyzer reads the input source program, scans the characters and produce a sequence of tokens that the parser can use for syntactic analysis.
- "get next token" is a command sent from the parser to the lexical analyzer


source-code
Token Stream
Token Stream
Lexical Analyzer
Syntax Analyzer
Regular expressions
Finite automata
Context-free Grammar


i f (x > 3 . 1
Character Stream
Lexical Analyzer
Token Stream
KEYWORD
"if"
BRACKET
"("
IDENTIFIER
"x"
OPERATOR
">"
NUMBER
"3.1"

Language

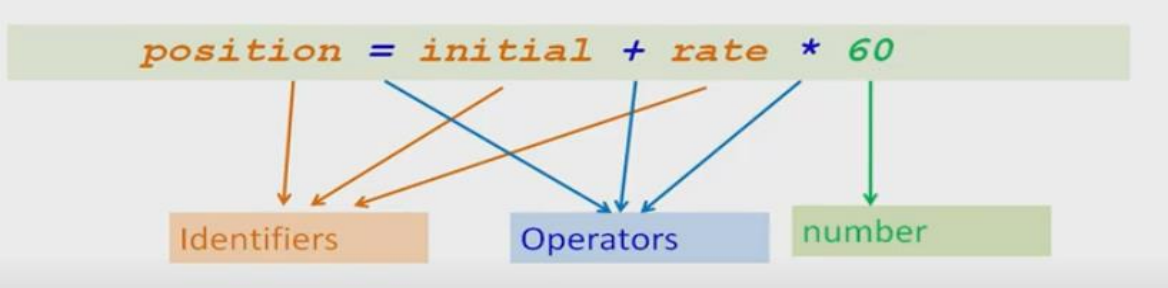
A language is considered as a finite set of strings over some finite set of alphabets. Computer languages are considered as finite sets, and mathematically set operations can be performed on them. Finite languages can be described by means of regular expressions.

Longest Match Rule

When the lexical analyzer read the source-code, it scans the code letter by letter; and when it encounters a whitespace, operator symbol, or special symbols, it decides that a word is completed.

```
int intvalue;
```

While scanning both lexemes till ‘int’, the lexical analyzer cannot determine whether it is a keyword int or the initials of identifier int value.



position = initial + rate * 60

gets translated as:

- The IDENTIFIER "position"
- The ASSIGNMENT SYMBOL "="
- The IDENTIFIER "initial"
- The PLUS OPERATOR "+"
- The IDENTIFIER "rate"
- The MULTIPLICATION OPERATOR "*"
- The NUMERIC LITERAL 60

Recognition of Tokens

Complete set of tokens form the set of terminal symbols used in the grammar for the parser.

In most of the languages, the tokens fall into these categories:

- Keywords,
- operators,
- identifiers,
- constants,
- literal strings,
- puntuation

The regular definitions for tokens are as follows:

if → if

then → then

else → else

relop → < | <= | = | <> | > | >=

id → letter (letter|digit)*

num → digit$^+$ (.digit+)? (E(+|-)?digit$^+$)?

delim → blank | tab | newline

ws → delim$^+$

FA and RE in lexical analyzer

Lexical analysis is process of recognizing tokens from the input.

Following are the steps:

1. Store the input in the input buffer
2. The token is read and regular expressions are built for corresponding token.
3. Regular expression is converted into a finite automata.
4. For each state of FA, a function is designed and each input along with transitional edges correspond to the input parameters of these functions.
5. The set of such functions ultimately creates lexical analyser program.

Steps of lexical analysis

- Tokenization
- Give only lexeme related Error messages: Exceeding length, unmatched string, illegal characters. No msgs related to syntax, semantics
- Eliminate comments, white spaces (Tab, blank space, newline)

E.g. Find no. of tokens in the foll:

int a=20, b=30;

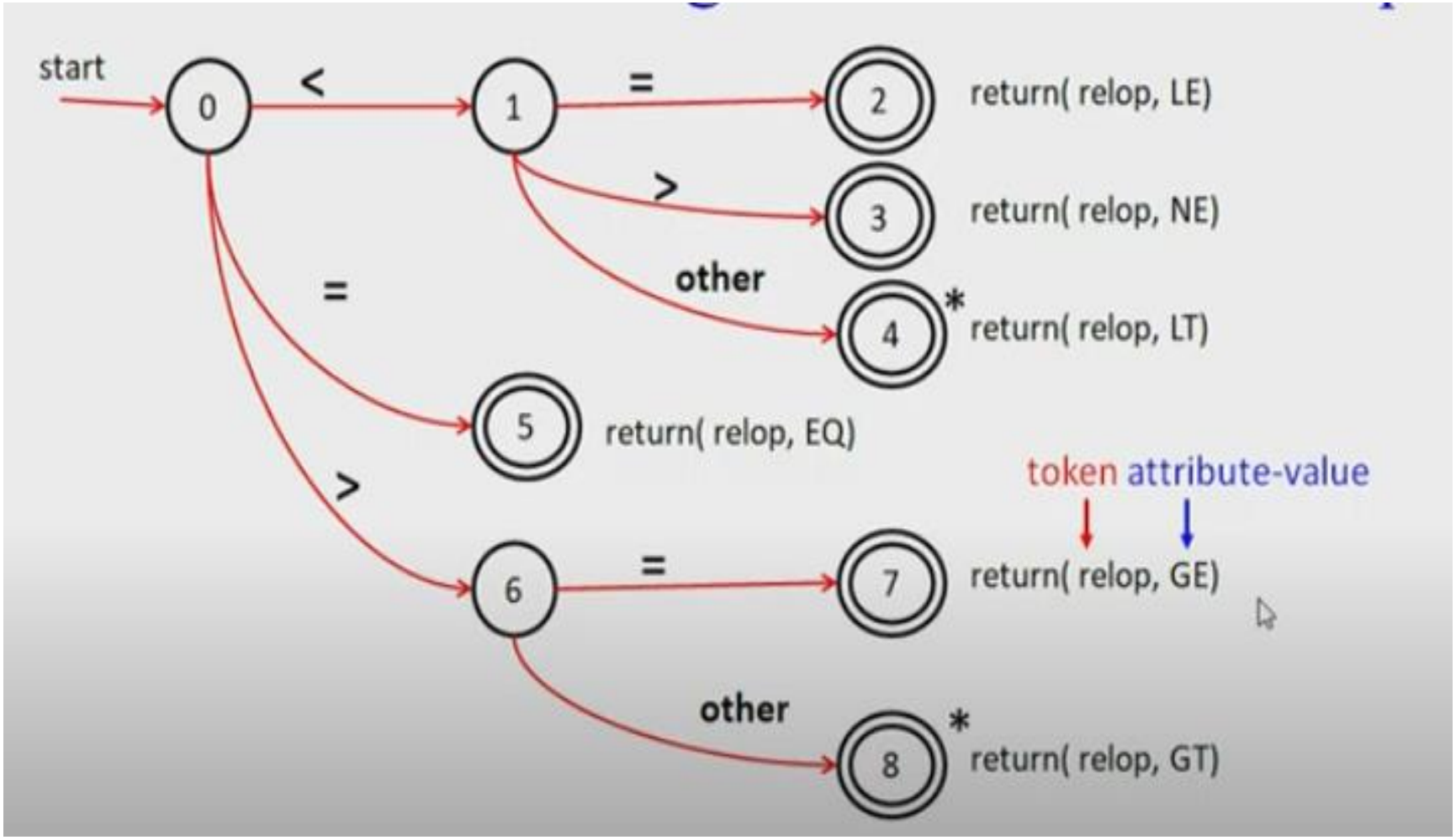
No. of tokens= 9

Printf(“i=%d”,i);

No. of tokens = 7

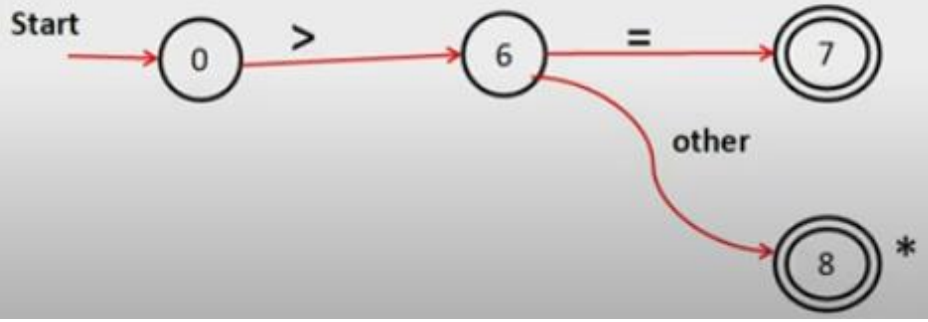
LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
`if`	**if**	–
`then`	**then**	–
`else`	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
`<`	**relop**	`LT`
`<=`	**relop**	`LE`
`=`	**relop**	`EQ`
`<>`	**relop**	`NE`
`>`	**relop**	`GT`
`>=`	**relop**	`GE`

Transition diagram for relational operators



Transition Diagram

- Lexical analysis uses transition diagram to keep track of information about characters that are seen as the forward pointer scans the input
- Transition diagrams are also called finite automata



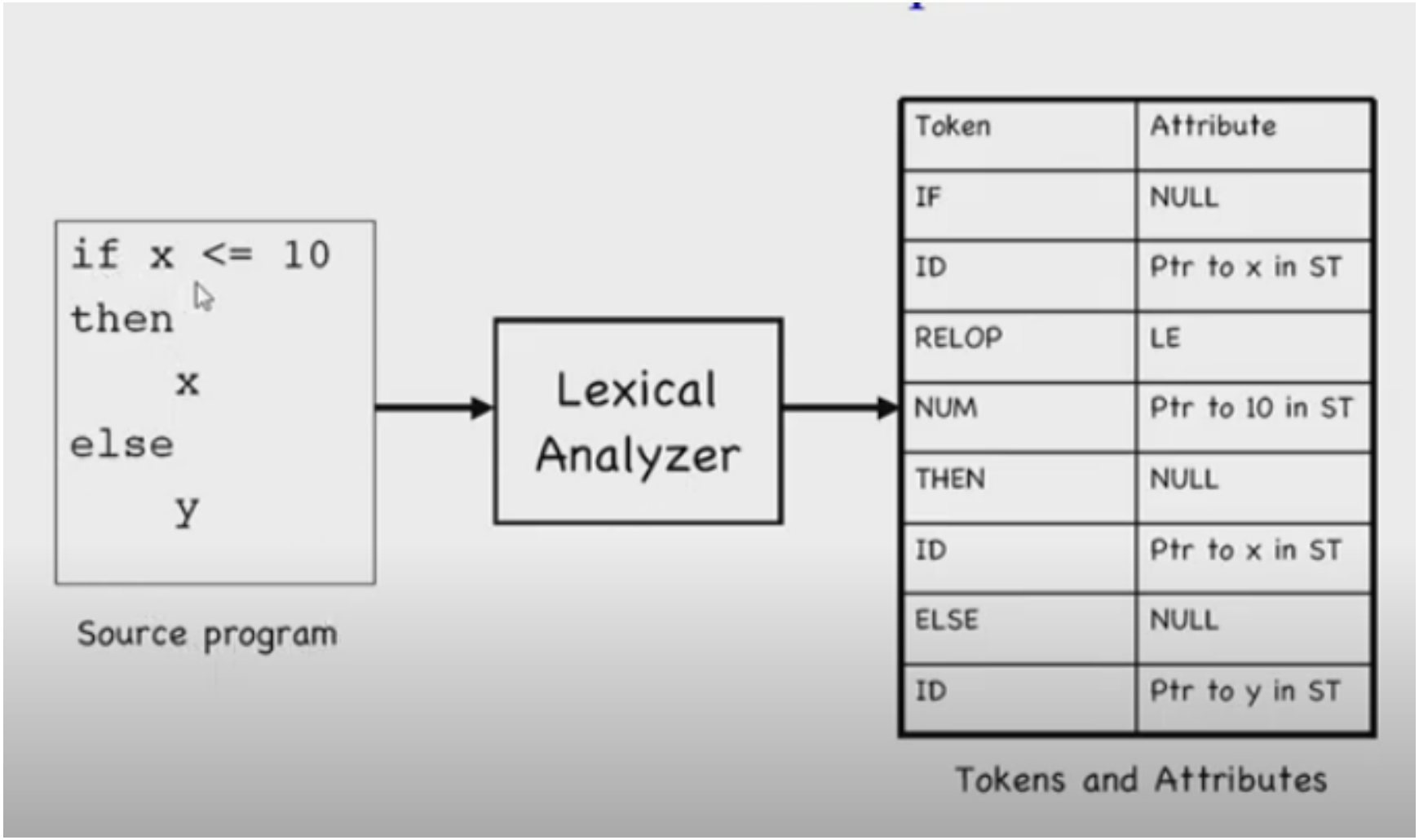
Transition diagram for identifiers and keywords

gettoken( ): returns token (**id, if, then**,…) if it looks the symbol table

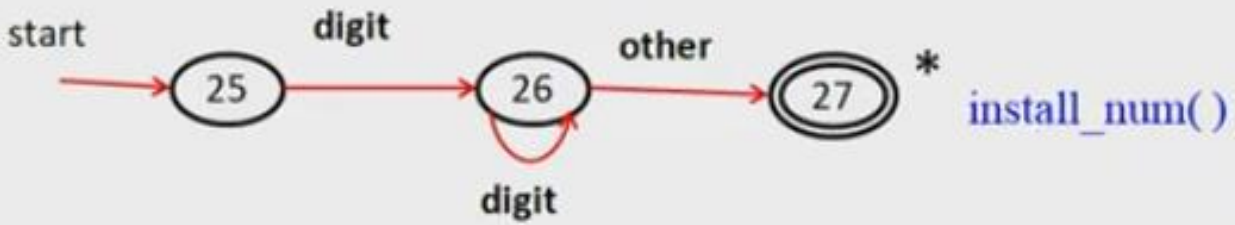
install_id( ): return 0 if keyword or a pointer to the symbol table entry if **id**

Input String	gettoken()	install_id()
for	for	0
if	if	0
else	else	0
count	id	ptr to sym table entry for count
num4	id	ptr to sym table entry for num4

Example

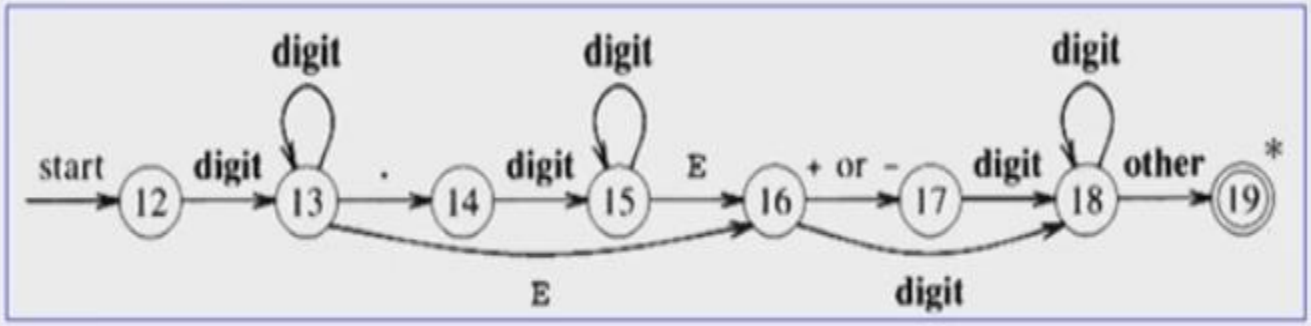

if x <= 10
then
x
else
y
Source program
Lexical Analyzer
Token
Attribute
IF
NULL
ID
Ptr to x in ST
RELOP
LE
NUM
Ptr to 10 in ST
THEN
NULL
ID
Ptr to x in ST
ELSE
NULL
ID
Ptr to y in ST
Tokens and Attributes

Unsigned integer numeric constant



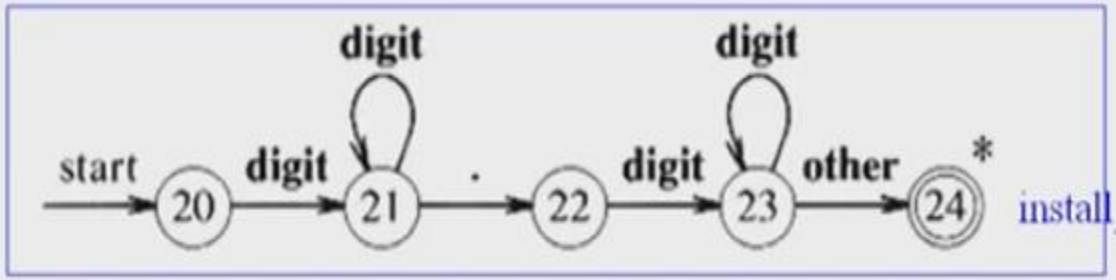
- It accepts the unsigned integer numeric constants
- Example- 314, 9, 41267 ……
- Install_num(): Enters the number in a literal table and returns a pointer to that entry

Unsigned Numbers



install_num( )

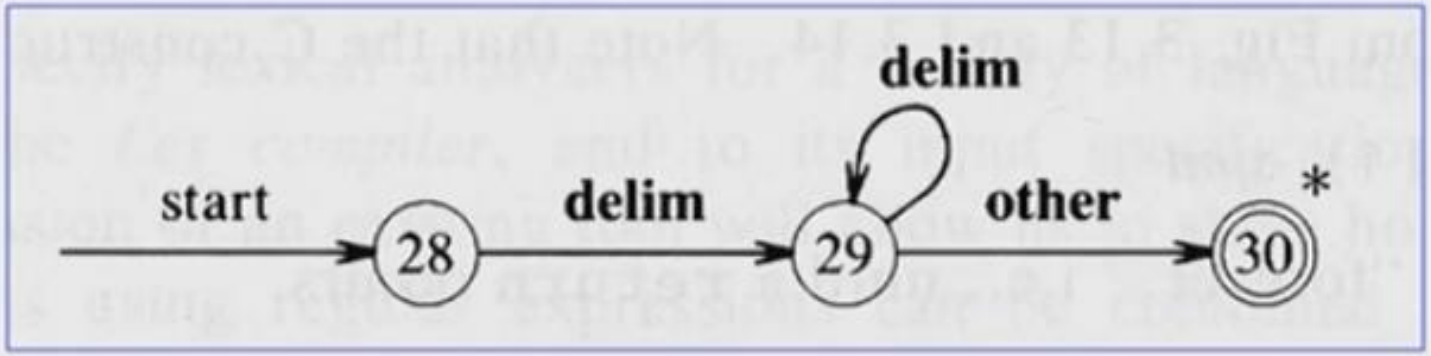
e.g. 12.3E4, 10E15



install_num( )

e.g. 12.3, 10.15

Whitespace



- It accepts the whitespaces (newline, tab space and space)
- Returns the token as ws

Tokens, Lexemes and patterns

A **lexeme** is a sequence of characters that are included in the source program according to the matching pattern of a token. It is nothing but an instance of a token.

The **token** is a sequence of characters which represents a unit of information in the source program.

E.g. Identifier, keyword, constants (literals), operators, separators, special characters.

A **pattern** is a description which is used by the token. In the case of a keyword which uses as a token, the pattern is a sequence of characters.

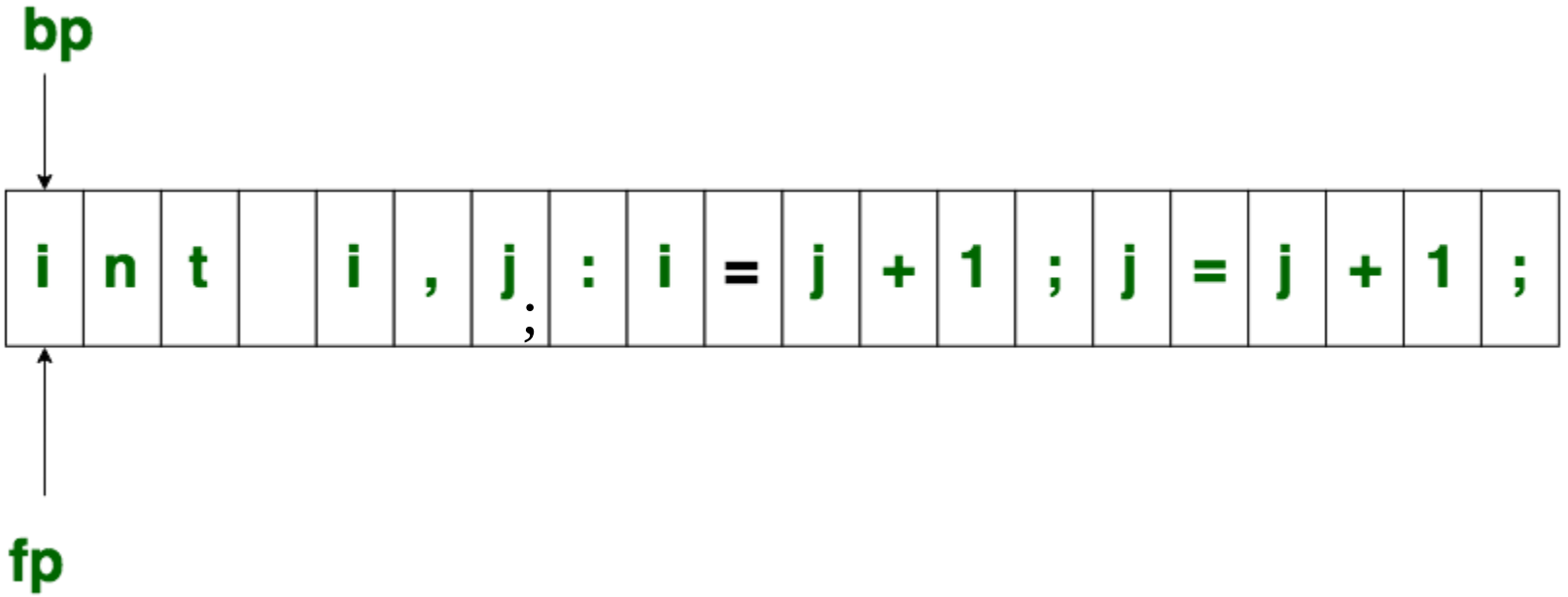
Token	Sample Lexemes	Informal description of pattern
if	if	if
while	while	while
Relation	<, <=, = , <>, > >=	< or <= or = or <> or > or >=
Id	count, sun, i, j, pi, D2	Letter followed by letters and digits
Num	0, 12, 3.1416, 6.02E23	Any numeric constant

Tokens, Lexemes and patterns

Token Type	Token Value	Informal Description
Integer constant	Numeric value	Numbers like 3, -5, 12 without decimal pts.
Floating constant	Numeric value	Numbers like 3.0, -5.1, 12.2456789
Reserved word	Word string	Words like if, then, class...(Keywords)
Identifiers	Symbol table index	Words not reserved starting with letter or _ and containing only letters, _, and digits
Relations	Operator string	<, <=, ==, ...
Operators	Operator string	=, +, - , ++, ...
Char constant	Char value	'A', ...
String	String	"this is a string", ...

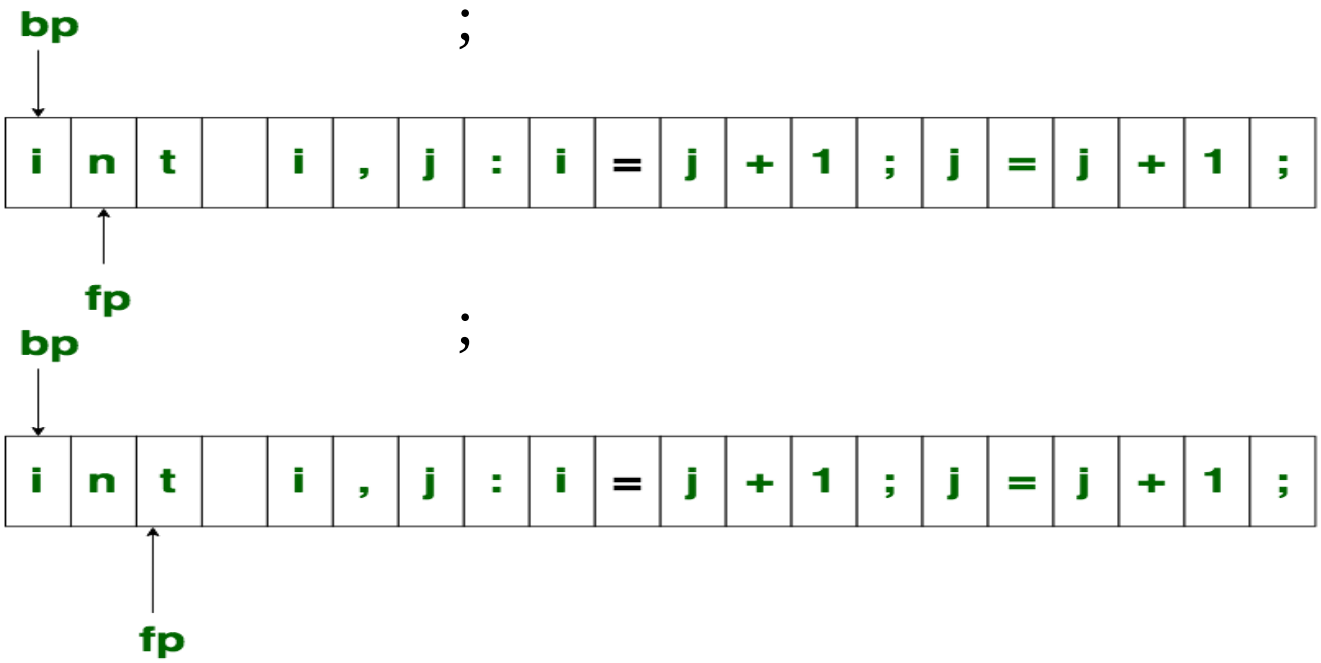
Example

Ex. int i,j; i=j+1; j=j+1;



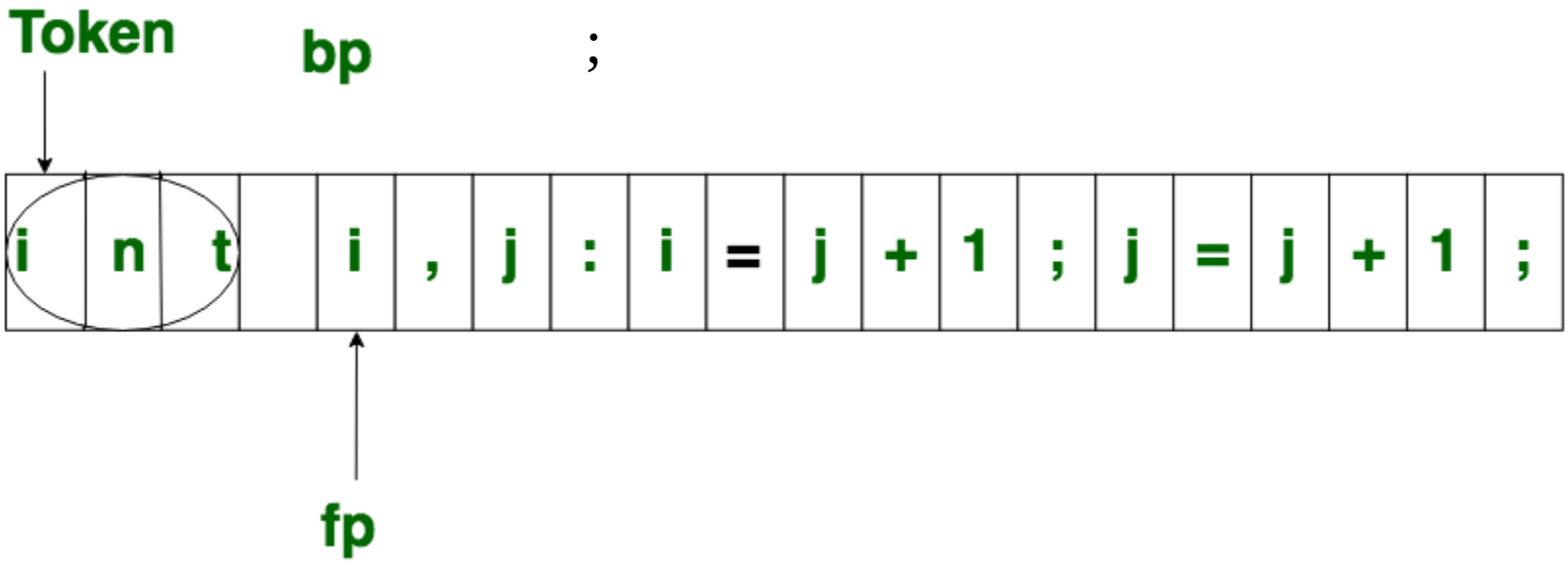
Initial Configuration

Initially both the pointers point to the first character of the input string as shown below



Input Buffering

- The forward ptr moves ahead to search for end of lexeme. As soon as the blank space is encountered, it indicates end of lexeme.
- In above example as soon as ptr (fp) encounters a blank space the lexeme “int” is identified.



Input buffering

Practice problems

```
int main )(
{
   x= y+z;
   int x,y,z;
   Printf ( " Sum %d %d", x);
{
```

No. of tokens= 26

```
main () {
int a= 10;
char b = "abc";
in t c= 30;
Ch ar d= " xyz";
in /* Comment */ t  m= 40.5;
}
```

No. of tokens= 33

```
main () {
a= b+++ --- --- +++==;
Printf ( "%d %d,a,b);
}
```

No. of tokens= 25

Data Structure-Symbol Table

The symbol table will contain the following types of information for the input strings in a source program:

- The lexeme (*input string*) itself
- Corresponding token
- Its semantic component (e.g., variable, operator, constant, functions, procedure, etc.)
- Data type
- Pointers to other entries (when necessary)

Data stored in the Symbol Table

Let's take a look at the kinds of information stored in the symbol table for different types of elements within a program.

Lexeme	Token Class	Symbol Type	Data Type	Value	Scope
while	TokWhile	Keyword	None	0	Global
+	TokPlus	Operator	None	0	Global
x	TokIdentifier	Variable	Integer	0	Sample
8	TokConstant	Literal	Integer	8	Global

The basic operations of the Symbol Table

- There are several operations that are performed on the Symbol Table, the most important being:
- *Adding symbols* - The reserved words, standard identifiers and operators are placed in the Symbol Table during its initialization.
  - New lexemes are added when the scanner encounters them, and they are assigned a token class.
  - Similarly, the semantic analyzer adds the appropriate properties and attributes that belong to the lexeme.

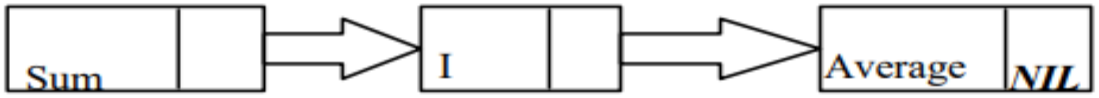
There are several different ways to organize the symbol table:

The most obvious is to organize it as an ***array of records***, but it would either require a linear search or constant sorting.

begin	tokbegin	stkeyword	dtnone	...	global
call	tokcall	stkeyword	dtnone	...	global
called	tokidentifier	stvariable	dtreal	...	sample
procedure	tokprocedure	stkeyword	dtnone	...	global
xmin	tokidentifier	tokconstant	dtinteger	...	sample

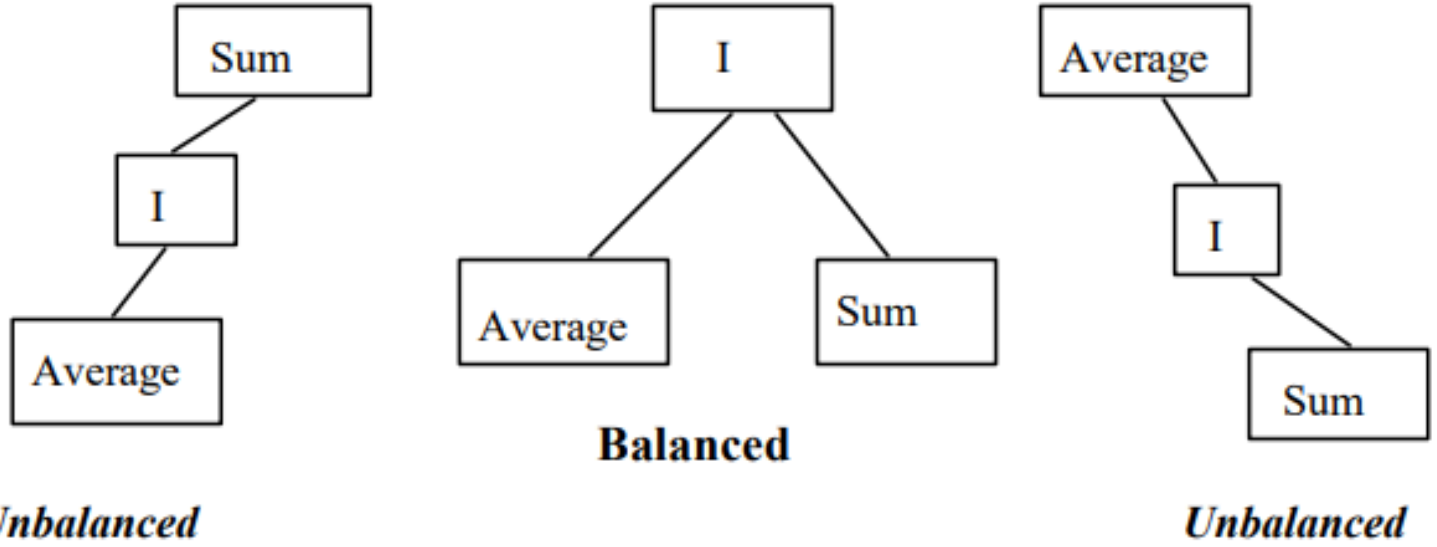
Possible organization methods (continued)

We could use an ***unordered list***. This would make efficient use of memory and it would be quick and easy to install names, but retrieval would be unacceptably slow.



An ordered list would make storing and retrieving names faster, but not by that much.

Binary search trees would provide some compromises on these structures above, but only when the tree is balanced. This rarely happens on its own and takes time for balancing to be done by the insertion routine.



Organization of the Symbol Table

We need a method for storing the lexemes in an efficient manner. Using an array of strings is not efficient and would limit the length of key words and identifiers.

1	c	a	l	l				
2	d	e	c	l	a	r	e	
3	a	v	e	r	a	g	e	s
4	x							

FORTRAN and other earlier programming languages restricted identifiers to a specific length for exactly this reason. (FORTRAN limited identifiers to 6 letters; C originally ignored anything after the first 8 letters.)

The string table and name table

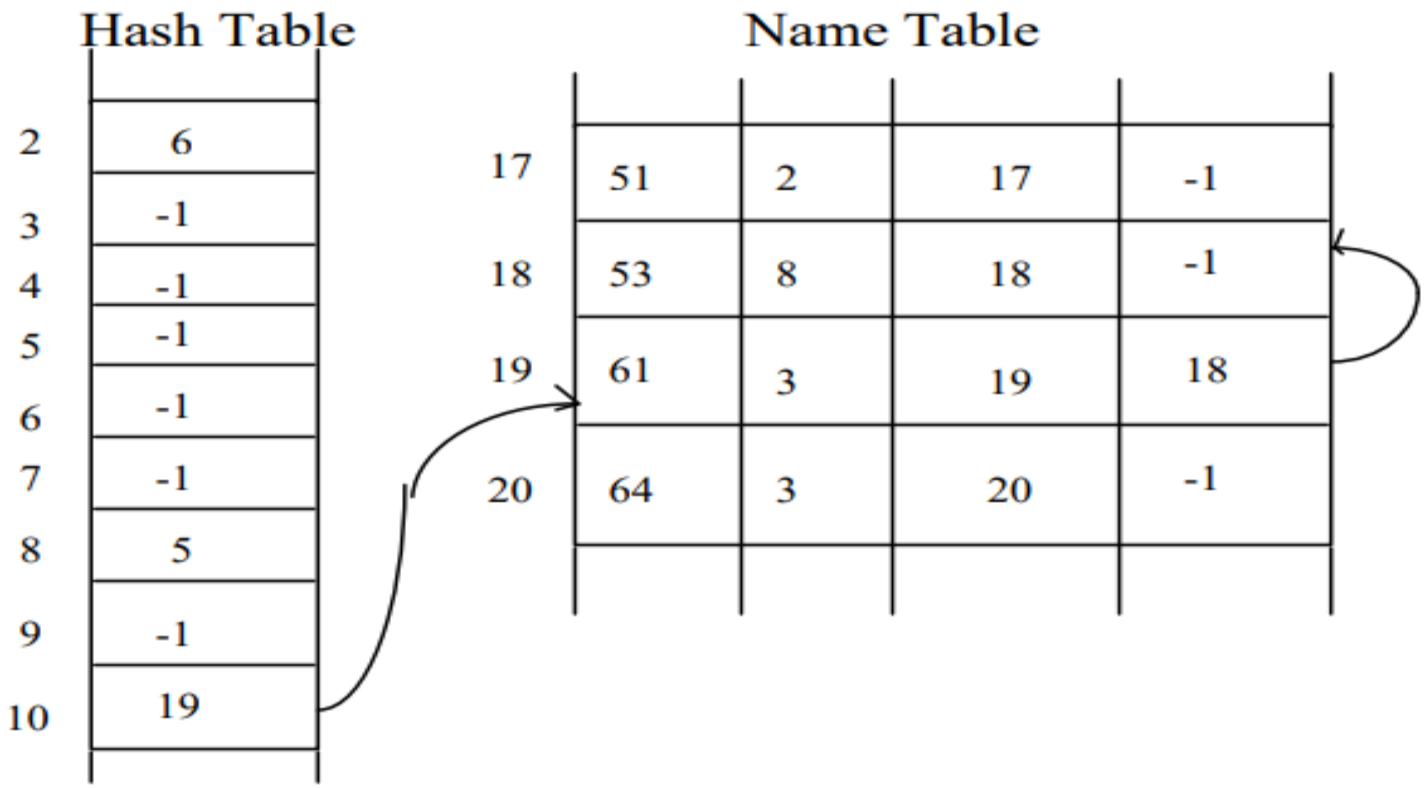
String table

b	e	g	i	n	c	a	l	l	d	e	c	l	a	r	e	

Name table

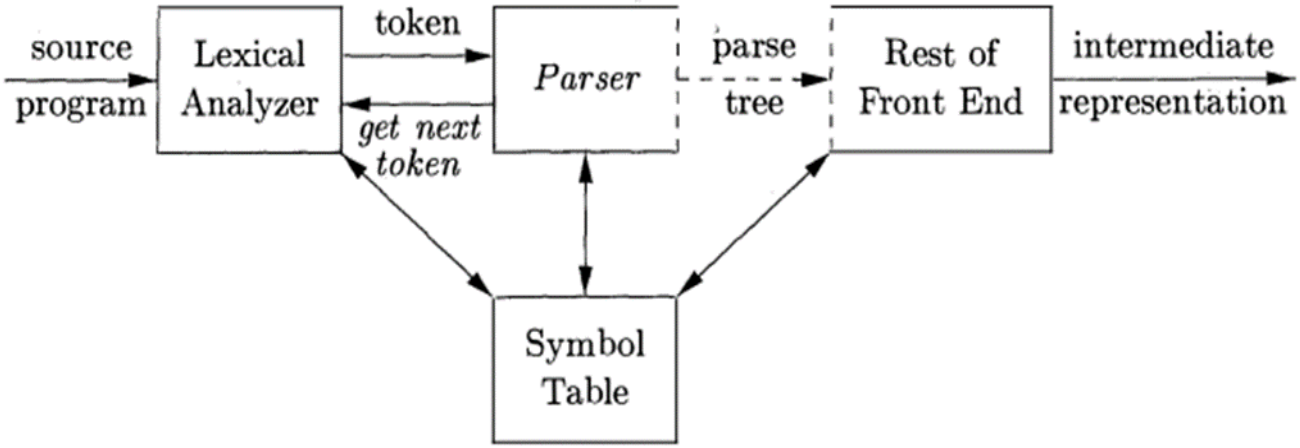
	Start position	Length	Pointer to attribute table entry	Token
0	0	5	0	-1
1	5	4	1	-1
2	9	7	2	-1

Hash Table and the Name Table



Syntax Analysis

Syntax Analyzer follows the production rules defined by the Context free grammar.



Context Free Grammar is defined as : G (V, T, P, S)

V is a set of non-terminal symbols.

T is a set of terminals where $V \cap T = NULL$.

P is a set of rules, P: $V \rightarrow (V \cup T)^*$, i.e., the left-hand side of the production rule P does have any right context or left context.

S is the start symbol.

Syntax Analysis

A derivation tree or parse tree is an ordered rooted tree that graphically represents the semantic information a string derived from a context-free grammar.

Leftmost and Rightmost Derivation of a String

Leftmost derivation − A leftmost derivation is obtained by applying production to the leftmost variable in each step.

Rightmost derivation − A rightmost derivation is obtained by applying production to the rightmost variable in each step.

Let any set of production rules in a CFG be

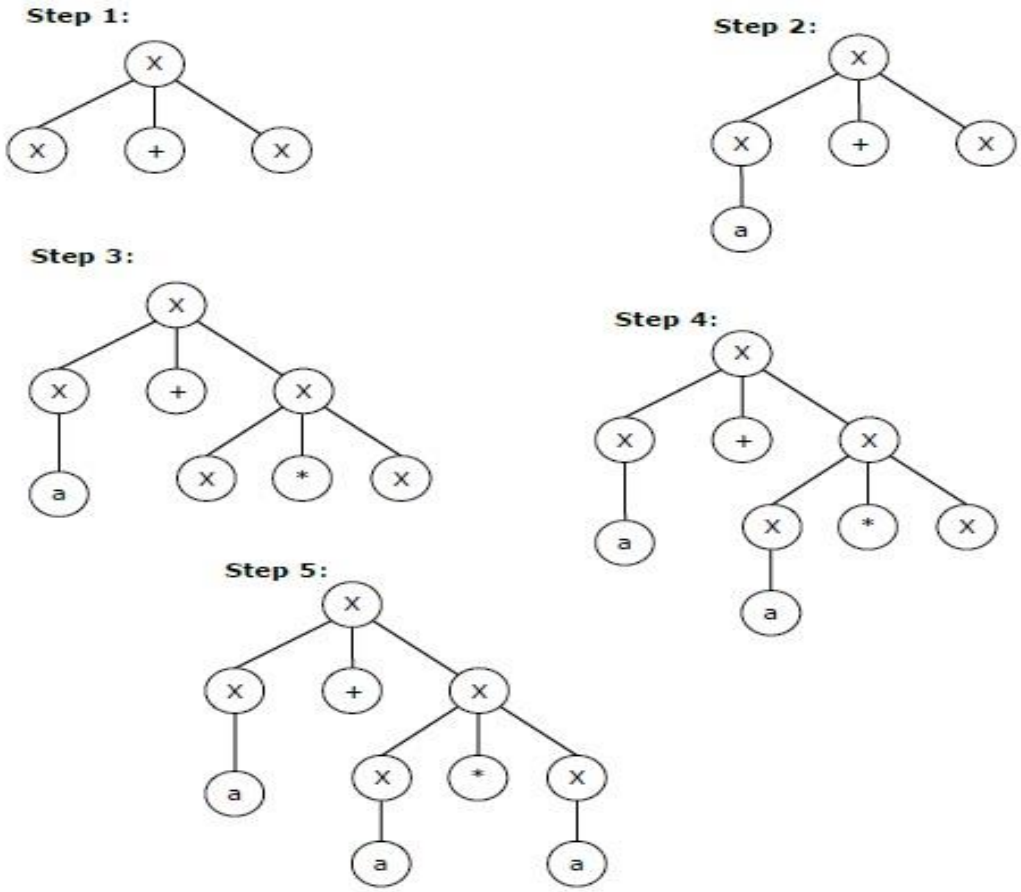
$X \rightarrow X+X \mid X*X \mid |X| \mid a$

over an alphabet {a}.

$X \rightarrow X+X \mid X*X \mid X \mid a$

The leftmost derivation for the string "a+a*a" may be −

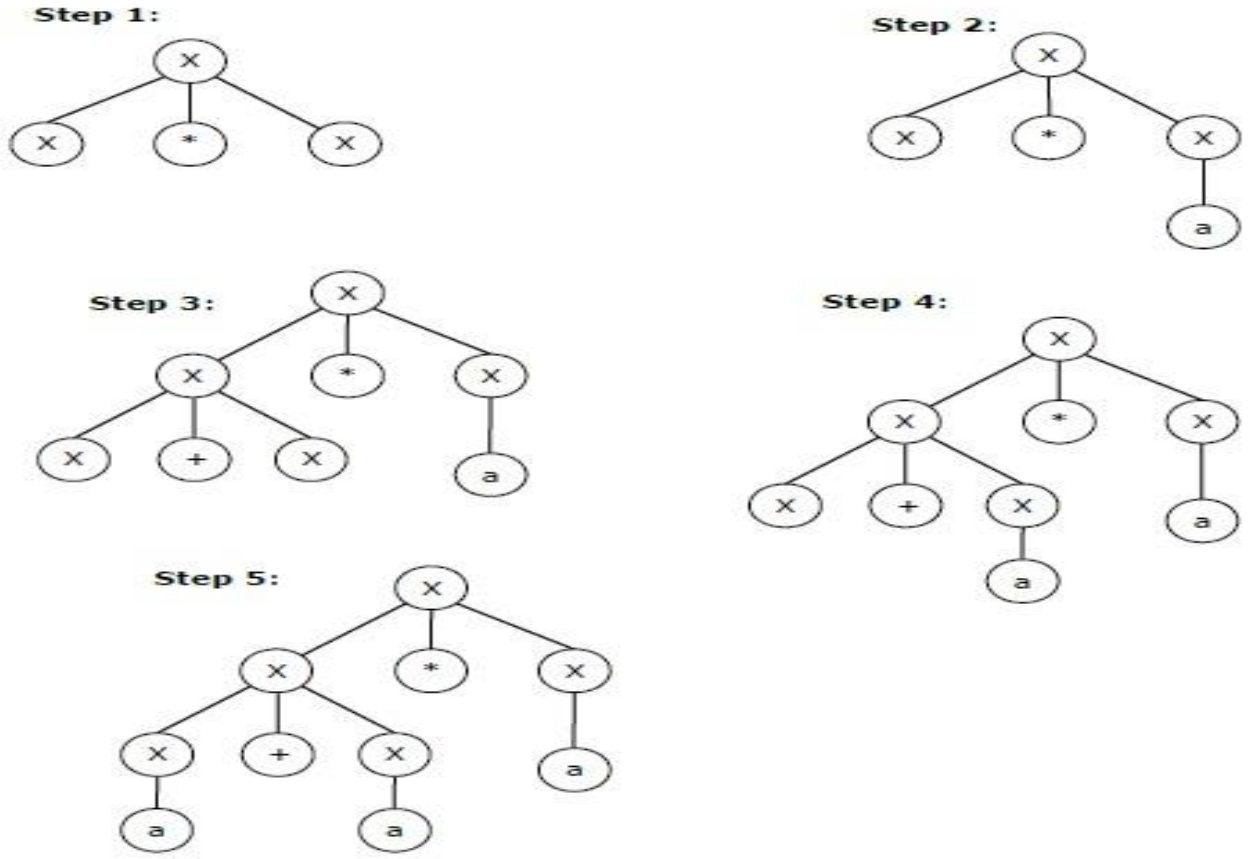
$X \rightarrow X+X \rightarrow a+X \rightarrow a + X*X \rightarrow a+a*X \rightarrow a+a*a$



X → X+X | X*X |X| a

The rightmost derivation for the above string "a+a*a" may be −

X → X*X → X*a → X+X*a → X+a*a → a+a*a



Ambiguous grammar

A grammar is said to ambiguous if for any string generated by it, it produces more than one-

- Parse tree
- Or derivation tree
- Or syntax tree
- Or leftmost derivation
- Or rightmost derivation

Ambiguous Grammar creates confusion for parser

Check the Grammar : $E \rightarrow E + E / E \times E / id$

E → E + E / E x E / id ambiguous grammar

E → **E** + E
→ id + **E**
→ id + **E** x E
→ id + id x **E**
→ id + id x id

Leftmost Derivation-01

E → **E** x E
→ **E** + E x E
→ id + **E** x E
→ id + id x **E**
→ id + id x id

Leftmost Derivation-02

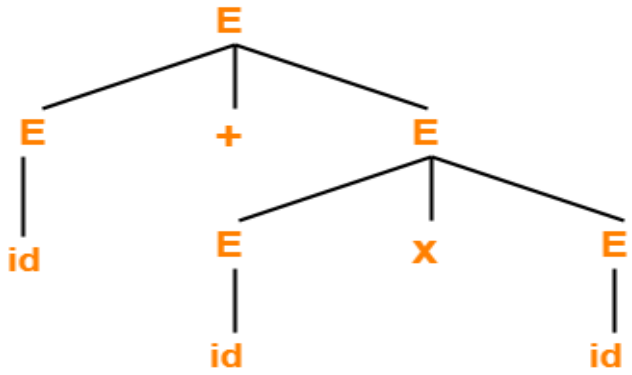
E → E + **E**
→ E + E x **E**
→ E + **E** x id
→ **E** + id x id
→ id + id x id

Rightmost Derivation-01

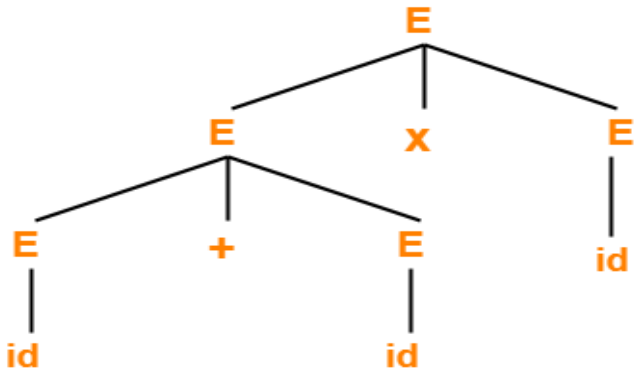
E → E x **E**
→ **E** x id
→ E + **E** x id
→ **E** + id x id
→ id + id x id

Rightmost Derivation-02

E → E + E / E x E / id ambiguous grammar

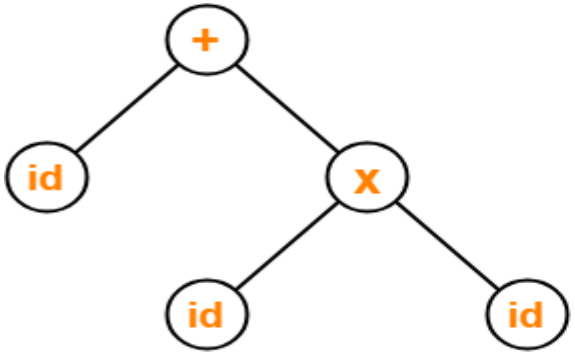


Parse Tree-01

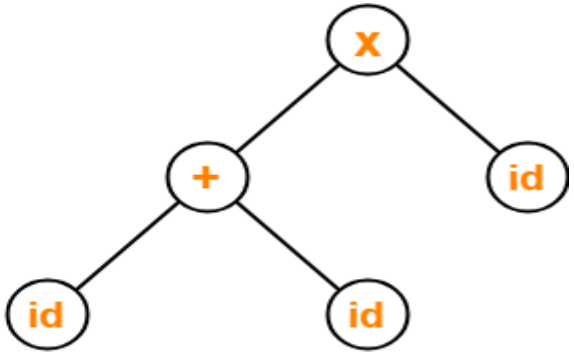


Parse Tree-02

Since two parse trees exist for string w, therefore the grammar is ambiguous.



Syntax Tree-01



Syntax Tree-02

Consider a string w generated by the grammar-

w = id + id x id

Since two syntax trees exist for string w, therefore the grammar is ambiguous.

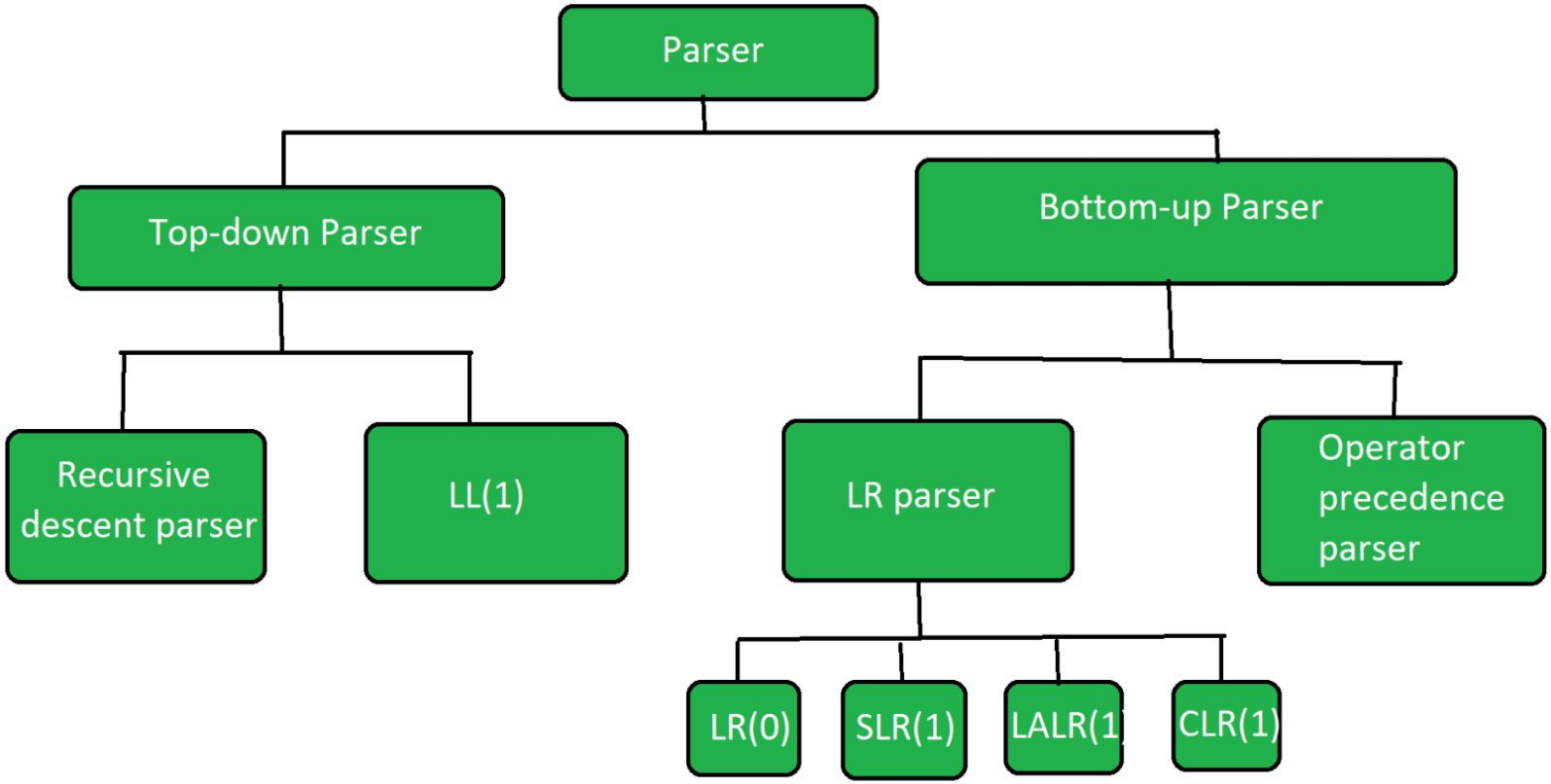
Left and Right Recursive Grammars

In a context-free grammar G, if there is a production in the form X → Xa where X is a non-terminal and ‘a’ is a string of terminals, it is called a left recursive production. The grammar having a left recursive production is called a left recursive grammar.

And if in a context-free grammar G, if there is a production is in the form X → aX where X is a non-terminal and ‘a’ is a string of terminals, it is called a right recursive production. The grammar having a right recursive production is called a right recursive grammar.

PARSING

It is a process of deriving string from a given grammar.



Rules for calculating first()

First and Follow sets are needed so that the parser can properly apply the needed production rule at the correct position.

First(α) is a set of terminal symbols that begin in strings derived from α.

Consider the production rule-

A → abc / def / ghi

Then, we have-

First(A) = { a , d , g }

Rules For Calculating First Function-

Rule-01:

For a production rule X → ∈,

First(X) = { ∈ }

Rule-02:

For any terminal symbol ‘a’,

First(a) = { a }

Rule-03:

For a production rule $X \rightarrow Y1Y2Y3$,

Calculating First(X)

If $\in \notin$ First(Y1), then First(X) = First(Y1)

If $\in \in$ First(Y1), then First(X) = { First(Y1) – $\in$ } $\cup$ First(Y2Y3)

Calculating First(Y2Y3)

If $\in \notin$ First(Y2), then First(Y2Y3) = First(Y2)

If $\in \in$ First(Y2), then First(Y2Y3) = { First(Y2) – $\in$ } $\cup$ First(Y3)

Similarly, we can make expansion for any production rule $X \rightarrow Y1Y2Y3\ldots..Yn$.

Consider the production rule-

$$A \rightarrow abc / def / ghi$$

Then, we have-

$$First(A) = \{ a , d , g \}$$

Calculate the first and follow functions for the given grammar-

S → aBDh

B → cC

C → bC / ∈

D → EF

E → g / ∈

F → f / ∈

First Functions-

- First(S) = { a }
- First(B) = { c }
- First(C) = { b , ∈ }
- First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }
- First(E) = { g , ∈ }
- First(F) = { f , ∈ }

Follow Functions-

- Follow(S) = { $ }
- Follow(B) = { First(D) – ∈ } ∪ First(h) = { g , f , h }
- Follow(C) = Follow(B) = { g , f , h }
- Follow(D) = First(h) = { h }
- Follow(E) = { First(F) – ∈ } ∪ Follow(D) = { f , h }
- Follow(F) = Follow(D) = { h }

Rules for calculating follow()

Follow(α) is a set of terminal symbols that appear immediately to the right of α.

Rule-01:

For the start symbol S, place $ in Follow(S).

Rule-02:

For any production rule $A \rightarrow \alpha B$,

Follow(B) = Follow(A)

Rule-03:

For any production rule $A \rightarrow \alpha B\beta$,

If $\in \notin$ First(β), then Follow(B) = First(β)

If $\in \in$ First(β), then Follow(B) = { First(β) – $\in$ } ∪ Follow(A)

S-> AaAb | BbBa

A-> ∈

B-> ∈

Follow(A)={a,b}

Follow(B)={b,a}

S-> ABC Follow (A)=First(B) if it is ∈, So, First (C) then follow (S) = $

A->DEF

B-> ∈

C-> ∈

D-> ∈

E-> ∈

F-> ∈

Calculate the first and follow functions for the given grammar-

S → aBDh

B → cC

C → bC / ∈

D → EF

E → g / ∈

F → f / ∈

First Functions-

- First(S) = { a }
- First(B) = { c }
- First(C) = { b , ∈ }
- First(D) = { First(E) – ∈ } ∪ First(F) = { g , f , ∈ }
- First(E) = { g , ∈ }
- First(F) = { f , ∈ }

Follow Functions-

- Follow(S) = { $ }
- Follow(B) = { First(D) – ∈ } ∪ First(h) = { g , f , h }
- Follow(C) = Follow(B) = { g , f , h }
- Follow(D) = First(h) = { h }
- Follow(E) = { First(F) – ∈ } ∪ Follow(D) = { f , h }
- Follow(F) = Follow(D) = { h }

	First()	Follow()
S->ABCDE	{a, b, c}	{$}
A->a/ϵ	{a, ϵ}	{b, c}
B->b/ϵ	{b, ϵ}	{c}
C->c	{c}	{d, e, $ }
D->d/ϵ	{d, ϵ}	{e, $}
E->e/ϵ	{e, ϵ}	{$}

	First()	Follow()
S->ACB/CbB/Ba	{d, g, h, ϵ, b, a}	{$}
A->da/BC	{d, g, h, ϵ }	{h, g, $}
B->g/ϵ	{g, ϵ}	{$, a, h, g}
C->h/ϵ	{h, ϵ}	{g, $, b, h}

	First()	Follow()
S->Bb/Cd	{a,b,c,d}	{$}
B->aB/ϵ	{a, ϵ}	{b}
C->cC/ϵ	{c, ϵ}	{d}

LL1 parser

It is a non recursive predictive parsing

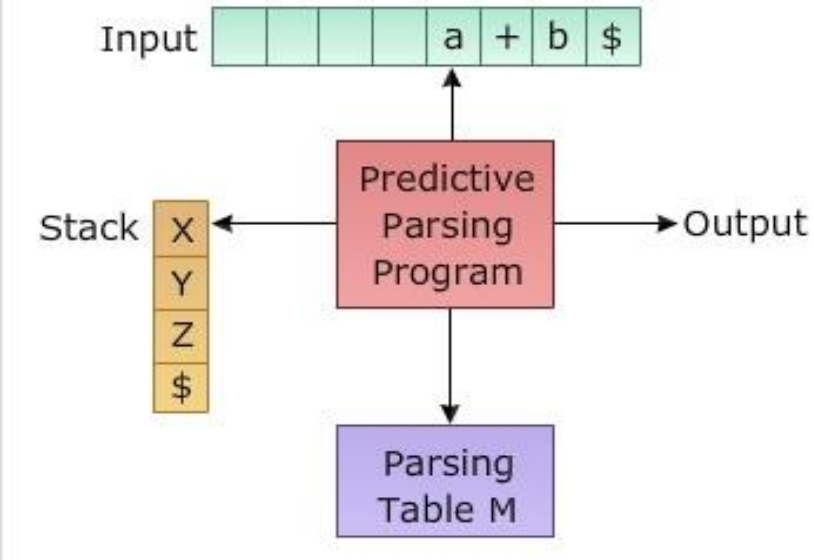
It is a top-down parser.

In this, L - Scanning input from left to right,

L - Producing a leftmost derivation and

1- One input symbol of lookahead at each step

- A grammar G is LL(1) iff whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:
  - ✓ **Condition 1**: For no terminal a, do both α and β derive strings beginning with a. ($FIRST(\alpha) \cap FIRST(\beta) = \phi$)
  - ✓ **Condition 2**: At most one of α and β can derive empty string.
  - ✓ **Condition 3**: If $\beta \xrightarrow{*} \varepsilon$ then α does not derive any string beginning with a terminal in FOLLOW(A). ($FIRST(\alpha) \cap FOLLOW(A) = \phi$)



INPUT: Contains string to be parsed with $ as it's end marker

STACK: Contains sequence of grammar symbols with $ as it's bottom marker. Initially stack contains only $

PARSING TABLE: A two dimensional array M[A,a], where A is a non-terminal and a is a Terminal

S->AA --- (1) first() - S-{a, b}

A->aA --- (2) A-{a, b}

A->b --- (3) follow() S-{$}

A-{$, a, b }

a	b	a	b	$

Input Buffer

Lookahead

	a	b	$
S	S->AA	S->AA	
A	A->aA	A->b	

Parsing Table

b-match (3)
a- match (1)
A
b -match (3)
a- match (2)
A
A - (1)
A
S (1)
$

Stack

S->AA->aAA->abA->abaA->abab

Check following grammar for LL(1):

This grammar is LL(1).

$S \rightarrow CC$

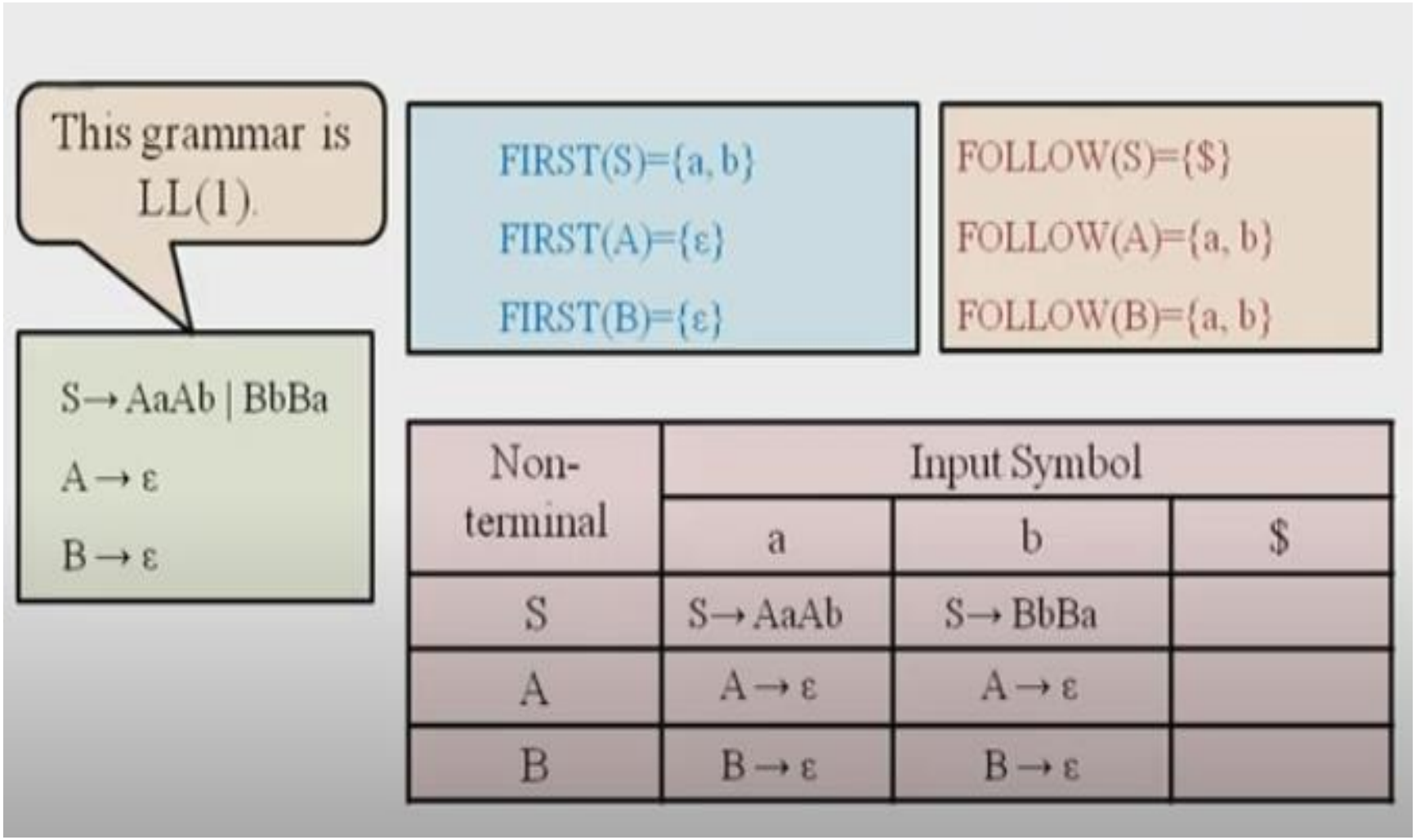
$C \rightarrow cC \mid d$

$S \rightarrow CC$

$C \rightarrow cC \mid d$

FIRST(S)={c, d}	FOLLOW(S)={$}
FIRST(C)={c, d}	FOLLOW(C)={c, d, $}

Non-terminal	Input Symbol		
	c	d	$
S	S→CC	S→CC	
C	C→cC	C→d	



Non-terminal	Input Symbol		
	a	b	$
S	S→AaAb	S→BbBa	
A	A→ε	A→ε	
B	B→ε	B→ε	

Find if the given grammar is LL1

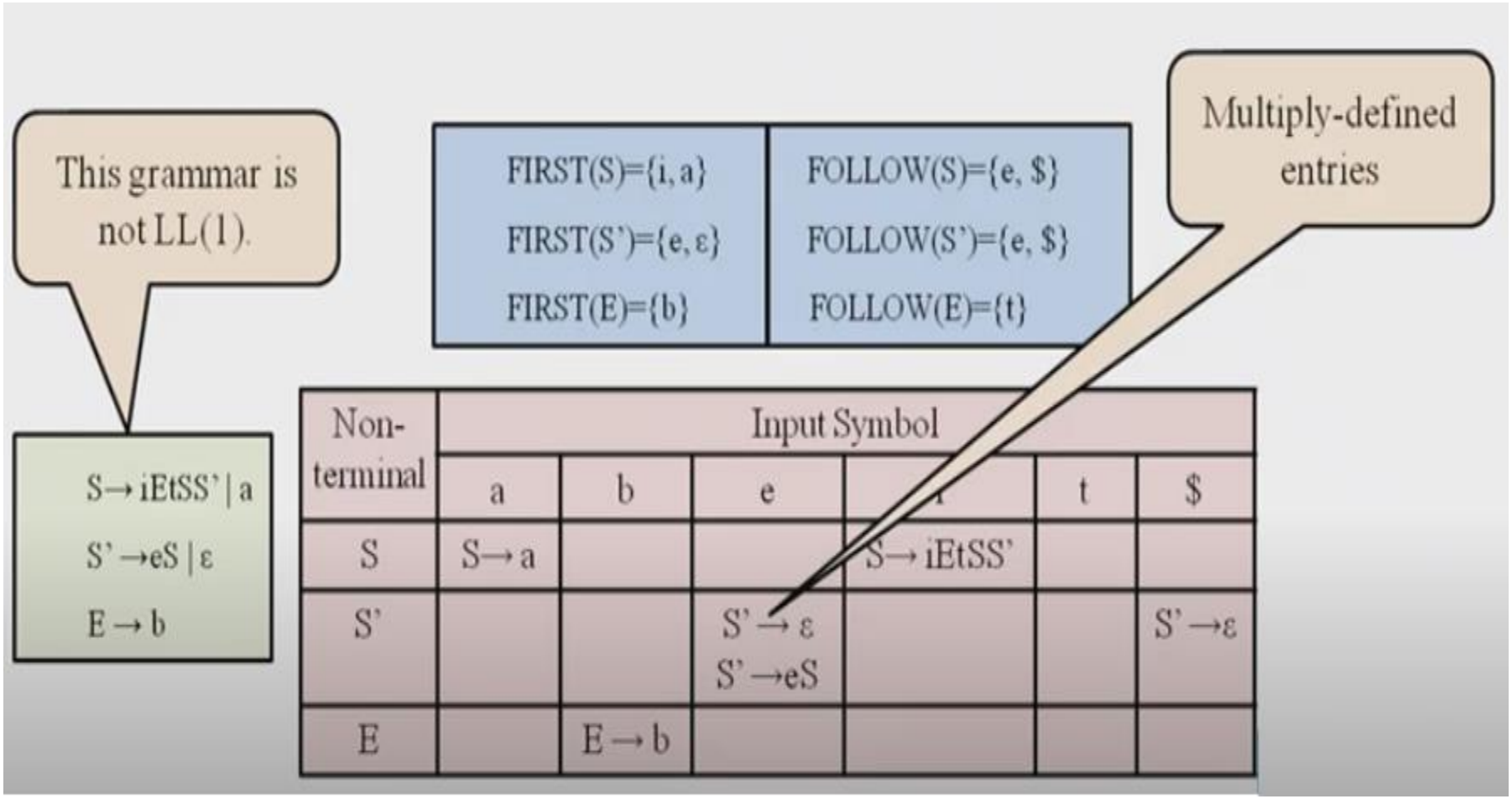
S-> aSbS/bSaS/ϵ

First()- {a, b, ϵ }

Follow()- {b, a, $}

	a	**b**	$
S	S-> aSbS S-> ϵ	S-> bSaS S-> ϵ	S-> ϵ

It shows multiple entries hence it is not LL1 grammar


This grammar is not LL(1).
S→ iEtSS' | a
S' →eS | ε
E → b
FIRST(S)={i, a}
FIRST(S')={e, ε}
FIRST(E)={b}
FOLLOW(S)={e, $}
FOLLOW(S')={e, $}
FOLLOW(E)={t}
Multiply-defined entries
Non-terminal
Input Symbol
a
b
e
i
t
$
S
S→ a
S→ iEtSS'
S'
S' → ε
S' →eS
S' →ε
E
E → b

Left recursive grammar

A production of grammar is said to have left recursion if the leftmost variable of its RHS is same as variable of its LHS.

A grammar containing a production having left recursion is called as Left Recursive Grammar. Elimination of Left Recursion

Left recursion is eliminated by converting the grammar into a right recursive grammar.

If we have the left-recursive pair of productions-

$A \rightarrow A\alpha / \beta$

(Left Recursive Grammar)

where β does not begin with an A.

Then, we can eliminate left recursion by replacing the pair of productions with-

For $A \rightarrow A\alpha / \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \in$

(Right Recursive Grammar)

This right recursive grammar functions same as left recursive grammar.

Right recursive Grammar

A production of grammar is said to have right recursion if the rightmost variable of its RHS is same as variable of its LHS.

A grammar containing a production having right recursion is called as Right Recursive Grammar.

Example-

$S \rightarrow aS / \in$

(Right Recursive Grammar)

Right recursion does not create any problem for the Top down parsers.

Therefore, there is no need of eliminating right recursion from the grammar.

Eliminate Left recursion

Consider the following grammar and eliminate left recursion-

$E \rightarrow E + T / T$

$T \rightarrow T \text{ x } F / F$

$F \rightarrow id$

For $A \rightarrow A\alpha / \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \in$

Solution-

The grammar after eliminating left recursion is-

$E \rightarrow TE'$

$E' \rightarrow +TE' / \in$

$T \rightarrow FT'$

$T' \rightarrow xFT' / \in$

$F \rightarrow id$

Eliminate Left recursion

- Consider the following grammar and eliminate left recursion-

$A \rightarrow ABd / Aa / a$

$B \rightarrow Be / b$

Solution:

For $A \rightarrow A\alpha / \beta$

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \in$

$A \rightarrow aA'$

$A' \rightarrow BdA' / aA' / \in$

$B \rightarrow bB'$

$B' \rightarrow eB' / \in$

- Consider the following grammar and eliminate left recursion-

$S \rightarrow S0S1S / 01$

Solution:

$S \rightarrow 01A$

$A \rightarrow 0S1SA / \in$

If RHS of more than one production starts with the same symbol,
then such a grammar is called as Grammar With Common Prefixes.

$A \rightarrow \alpha\beta1 / \alpha\beta2 / \alpha\beta3$

This kind of grammar creates a problematic situation for Top down parsers.

Top down parsers can not decide which production must be chosen to parse the string in right hand.

To remove this confusion we use left factoring,

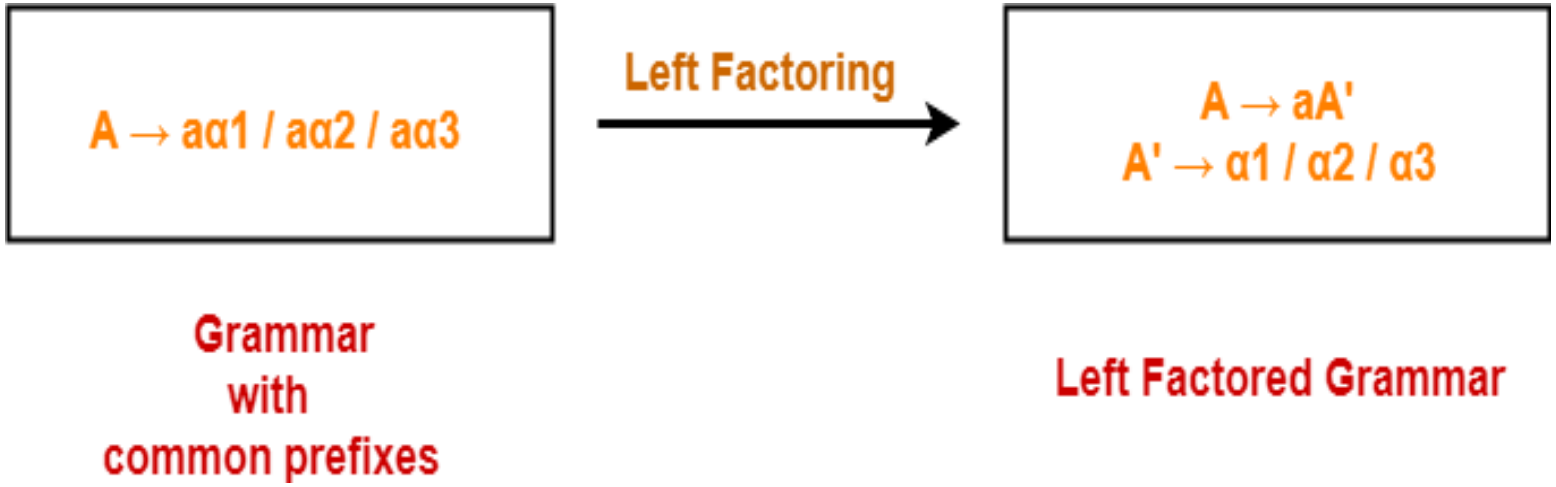
***It converts nondeterministic CFG into deterministic CFG**

We make one production for each common prefixes.

The common prefix may be a terminal or a non-terminal or a combination of both.

Rest of the derivation is added by new productions.

The grammar obtained after the process of left factoring is called as Left Factored Grammar.n, we use left factoring.



Do left factoring in the following grammar-

A-> aAB/aA

B->bB/b

Solution:

A->aA’

A’->AB/A

B->bB’

B’-> B/ ∈

$$A \rightarrow aA'$$
$$A' \rightarrow \alpha 1 / \alpha 2 / \alpha 3$$

Do left factoring in the following grammar-

$A \rightarrow aAB / aBc / aAc$

Solution-

Step1: $A \rightarrow aA'$

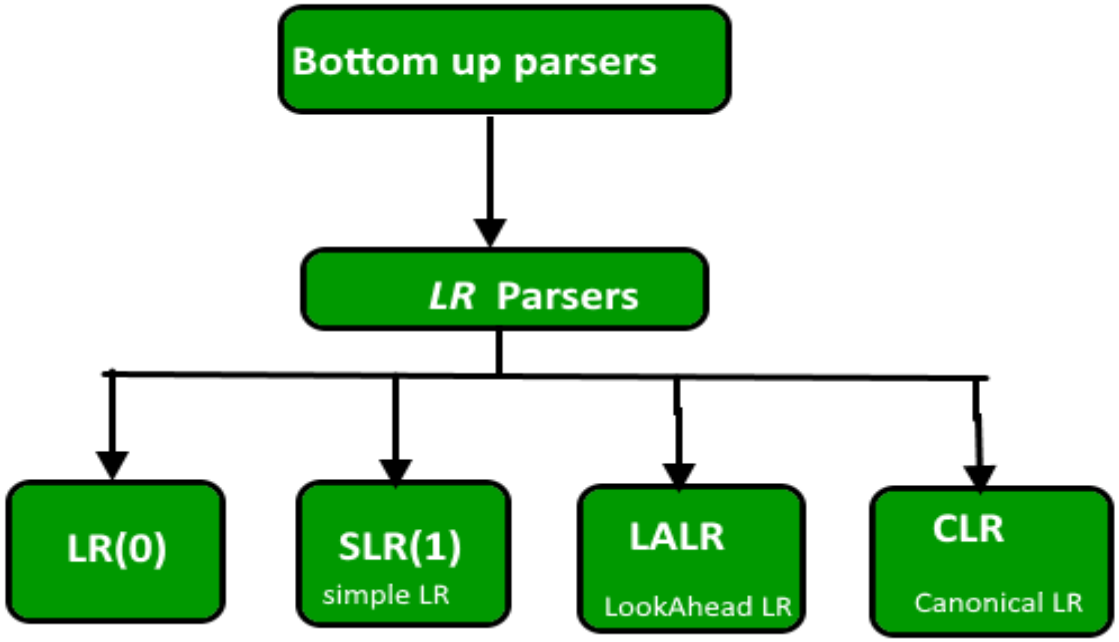
$A' \rightarrow AB / Bc / Ac$

Step2: $A \rightarrow aA'$

$A' \rightarrow AD / Bc$

$D \rightarrow B / c$

Bottom up Parsing



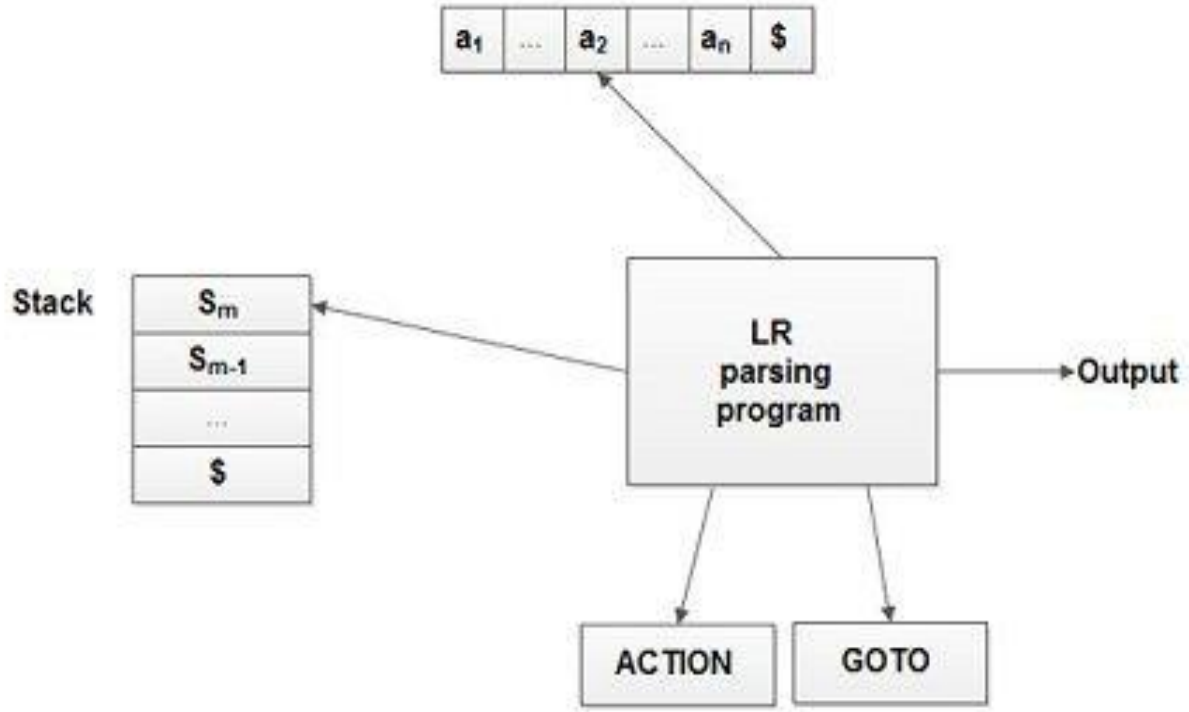
LR parser

It is a non-recursive shift reduce parser – LR(k)

L- Left to right scanning input stream

R- Construction of rightmost derivation which is in reverse manner

k- Denotes lookahead symbol to make decision



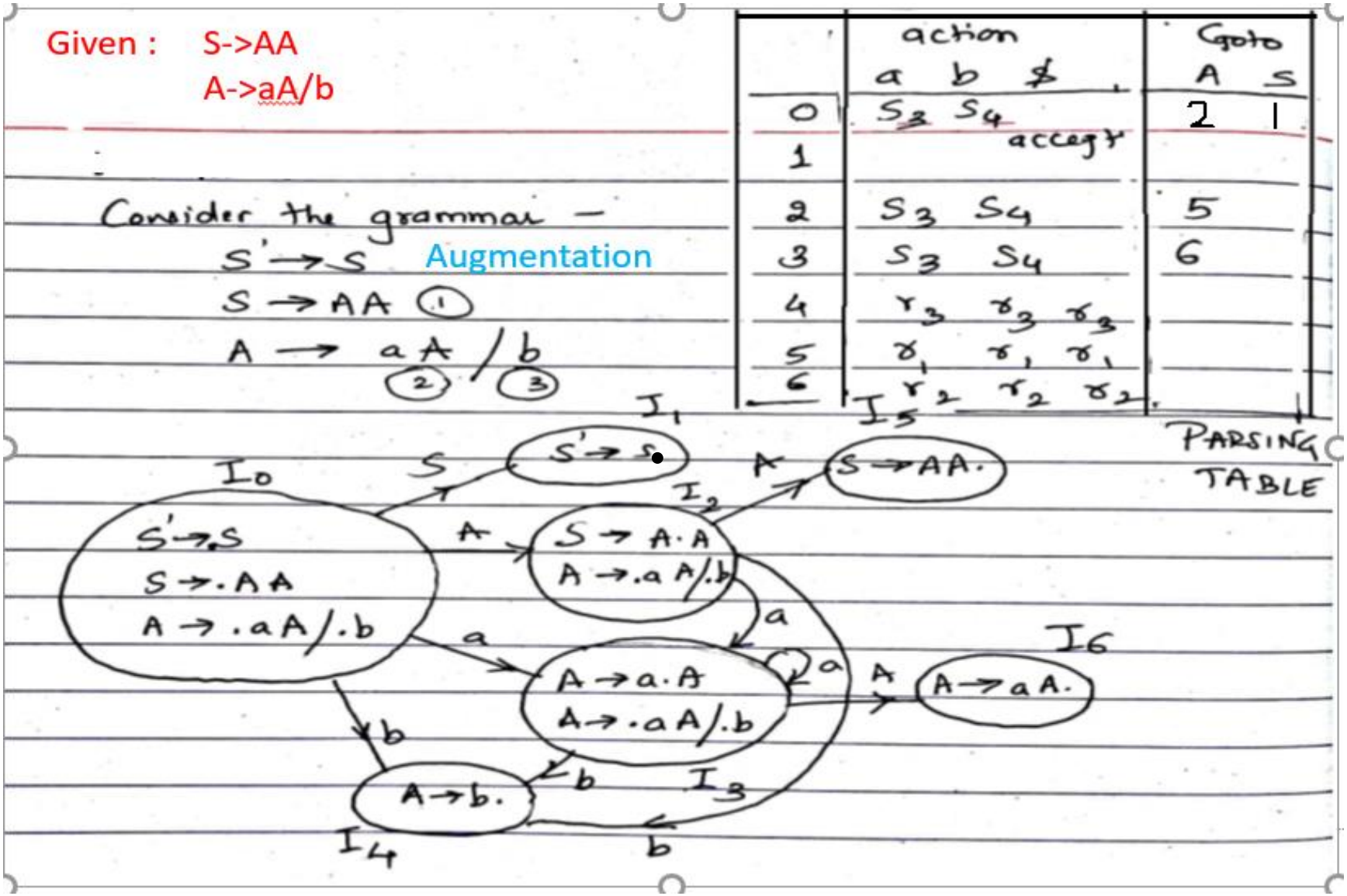
An LR-Parser uses

- states to memorize information during the parsing process,
- an action table to make decision (such as shift or reduce) and to compute states
- a goto table to compute states

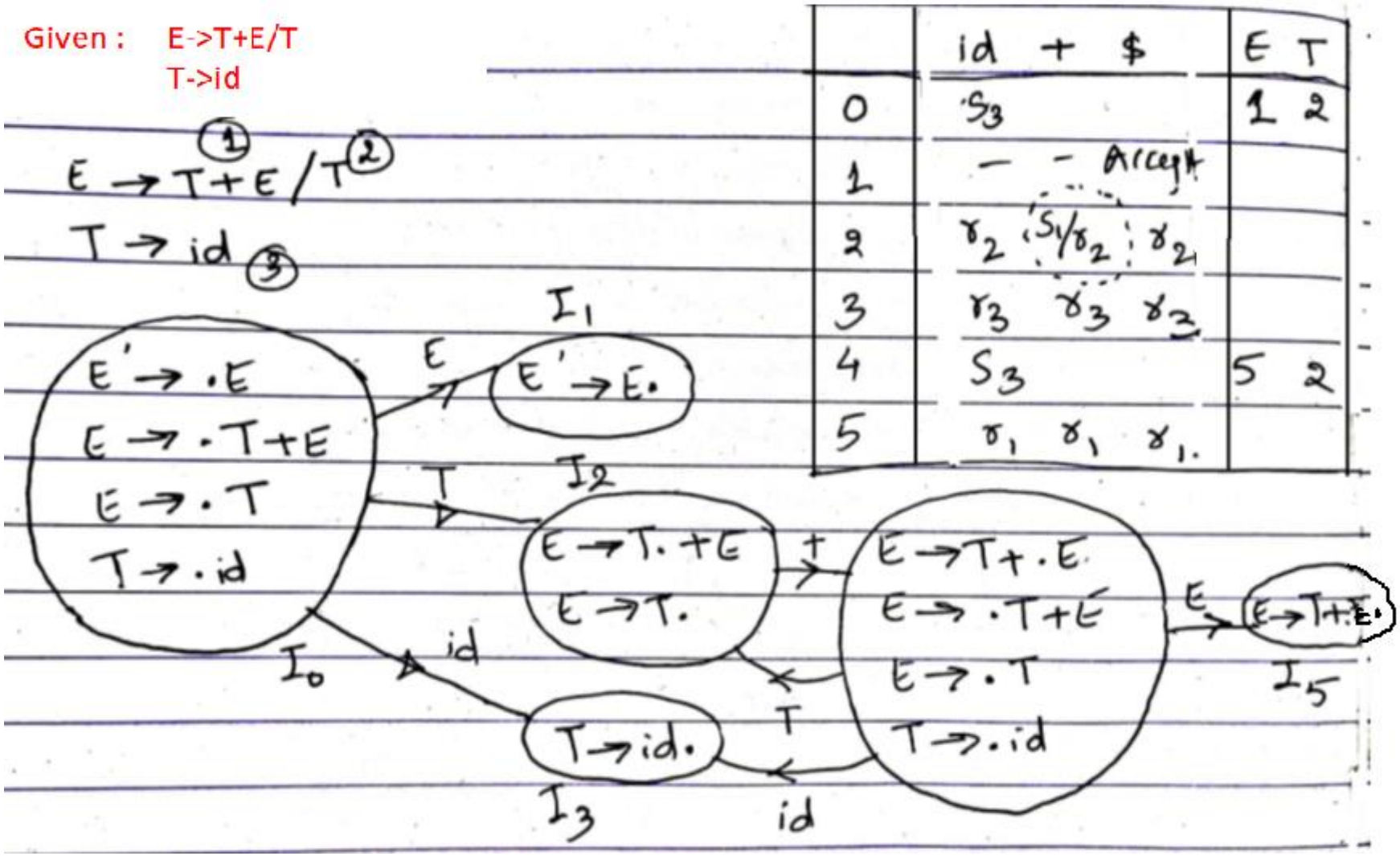
For a state s and a terminal a, the entry action[s, a] has one of the following forms

- shift s' where s' is a state,
- reduce A $ \longmapsto$ $ \beta$,
- accept,
- error.

Example : LR (0) grammar

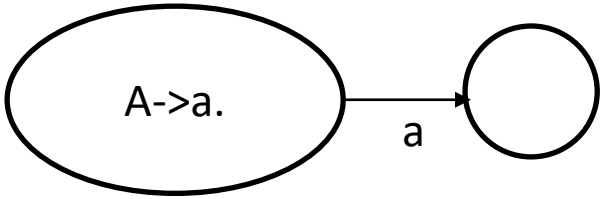


Example: Non LR(0)grammar



S-R and R-R conflicts

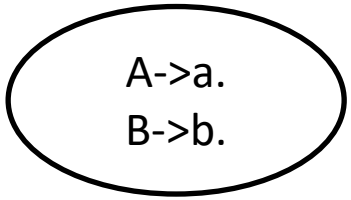
S-R conflicts



I5 I6

	a	b	
5	S6/r1	r1	

R-R conflicts



I6

	a	b	c	
5	r1/r2	r1/r2	r1/r2	

Reasons for attractiveness of LR parser

- LR parsers can handle a large class of context-free grammars.
- The LR parsing method is a most general non-back tracking shift-reduce parsing method.
- An LR parser can detect the syntax errors as soon as they can occur.
- LR grammars can describe more languages than LL grammars.

Drawbacks of LR parsers

- It is too much work to construct LR parser by hand. It needs an automated parser generator.
- If the grammar contains ambiguities or other constructs then it is difficult to parse in a left-to-right scan of the input.

LL parsing vs LR parsing

LL parsing	LR parsing
Left most derivative	Right most derivative
Starts with the root nonterminal on the stack	Ends with the root nonterminal on the stack
Ends when the stack is empty	Starts when the stack is empty
Uses the stack for designating what is still to be expected	Uses the stack for designating what is already seen
Builds the parse tree top-down	Builds the parse tree bottom-up
continuously pops a nonterminal off the stack, and pushes a corresponding right hand side	tries to recognize a right hand side on the stack, pops it, and pushes the corresponding nonterminal
expands nonterminals	reduces them
reads terminal when it pops one off the stack	reads terminals while it pushes them on the stack
uses grammar rules in an order which corresponds to pre-order traversal of the parse tree	does a post-order traversal.

Operator Grammars

Operator grammars have the property that no production on right side is empty (no null productions) or has two adjacent nonterminals. This property enables the implementation of efficient operator-precedence parsers. These parsers rely on the following three precedence relations:

Relation	Meaning
$a \lessdot b$	a yields precedence to b
$a \doteq b$	a has the same precedence as b
$a \gtrdot b$	a takes precedence over b

Ex: E-> E+E/E*E/id ---- is operator grammar (but is ambiguous)

E->EAE/id ----- is not an operator grammar

A-> +/*

Rule-01:

If precedence of b is higher than precedence of a, then we define a < b

If precedence of b is same as precedence of a, then we define a = b

If precedence of b is lower than precedence of a, then we define a > b

Rule-02:

An identifier is always given the higher precedence than any other symbol.

$ symbol is always given the lowest precedence.

Rule-03:

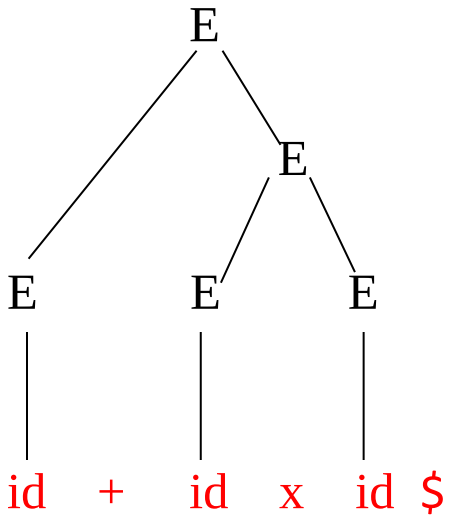
If two operators have the same precedence, then we go by checking their associativity.

E-> E+E/ExE/id

Operation relation (precedence) table is constructed as follows

	Id (Highest prec.)	+	x	$ (Lowest prec.)
Id	--	.>	.>	.>
+	<.	.> (left associative)	<.	.>
x	<.	.> (left associative)	.>	.>
$	<.	<.	<.	--

If TOS < input string char, push (shift) else pop (Reduce)



$ < id > + < id > x < id > $

$ E + < id > x < id > $

$ E + E x < id > $

$ E + E x E $

$ + x $

$ < + < x > $

$ < + > $

$ $

$	id	+	id	x	id	$

Shift –Reduce Parsing

E-> E+E/E*E/id

Operation relation (precedence) table is constructed as follows : id*id+id

	Id (Highest prec.)	+	*	$ (Lowest prec.)
Id	--	.>	.>	.>
+	<.	.> (left associative)	<.	.>
*	<.	.>	.>(left associative)	.>
$	<.	<.	<.	--

Stack	Input	Action
$	id*id+id $	Shift id
$ id	*id+id $	Reduce E -> id
$ E	*id+id $	Shift *
$ E *	id+id $	Shift id
$ E * id	+id $	Reduce E -> id
$ E * E	+id $	Reduce E-> E * E
$ E	+id $	Shift +
$ E +	id $	Shift id
$ E + id	$	Reduce E -> id
$ E + E	$	Reduce E -> E + E
$ E	$	Accept

Consider the following grammar-

$S \rightarrow (L) \mid a$

$L \rightarrow L , S \mid S$

Construct the operator precedence parser and parse the string (a , (a , a)).

	a	(	)	,	$
a		>	>	>	>
(	<	>	>	>	>
)	<	>	>	>	>
,	<	<	>	>	>
$	<	<	<	<	

$ < (<u>< a ></u> , < (< a > , < a >) >) > $

$ < (S , < (<u>< a ></u> , < a >) >) > $

$ < (S , < (S , <u>< a ></u>) >) > $

$ < (S , <u>< (S , S) ></u>) > $

$ < (S , <u>< (L , S) ></u>) > $

$ < (S , <u>< (L) ></u>) > $

$ <u>< (S , S) ></u> $

$ <u>< (L , S) ></u> $

$ <u>< (L) ></u> $

$ <u>< S ></u> $

$ $

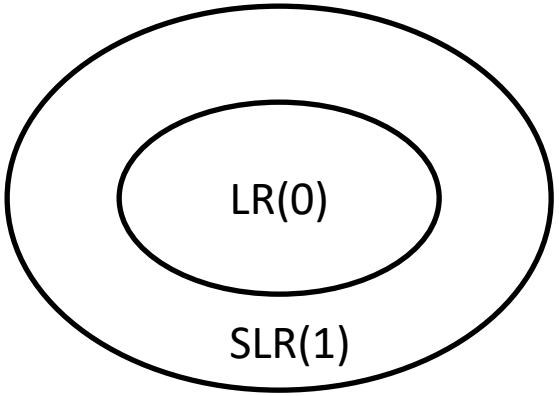
SLR Parsing

SLR (1) refers to simple LR Parsing. It is same as LR(0) parsing. The only difference is in the parsing table. To construct SLR (1) parsing table, we use canonical collection of LR (0) item.

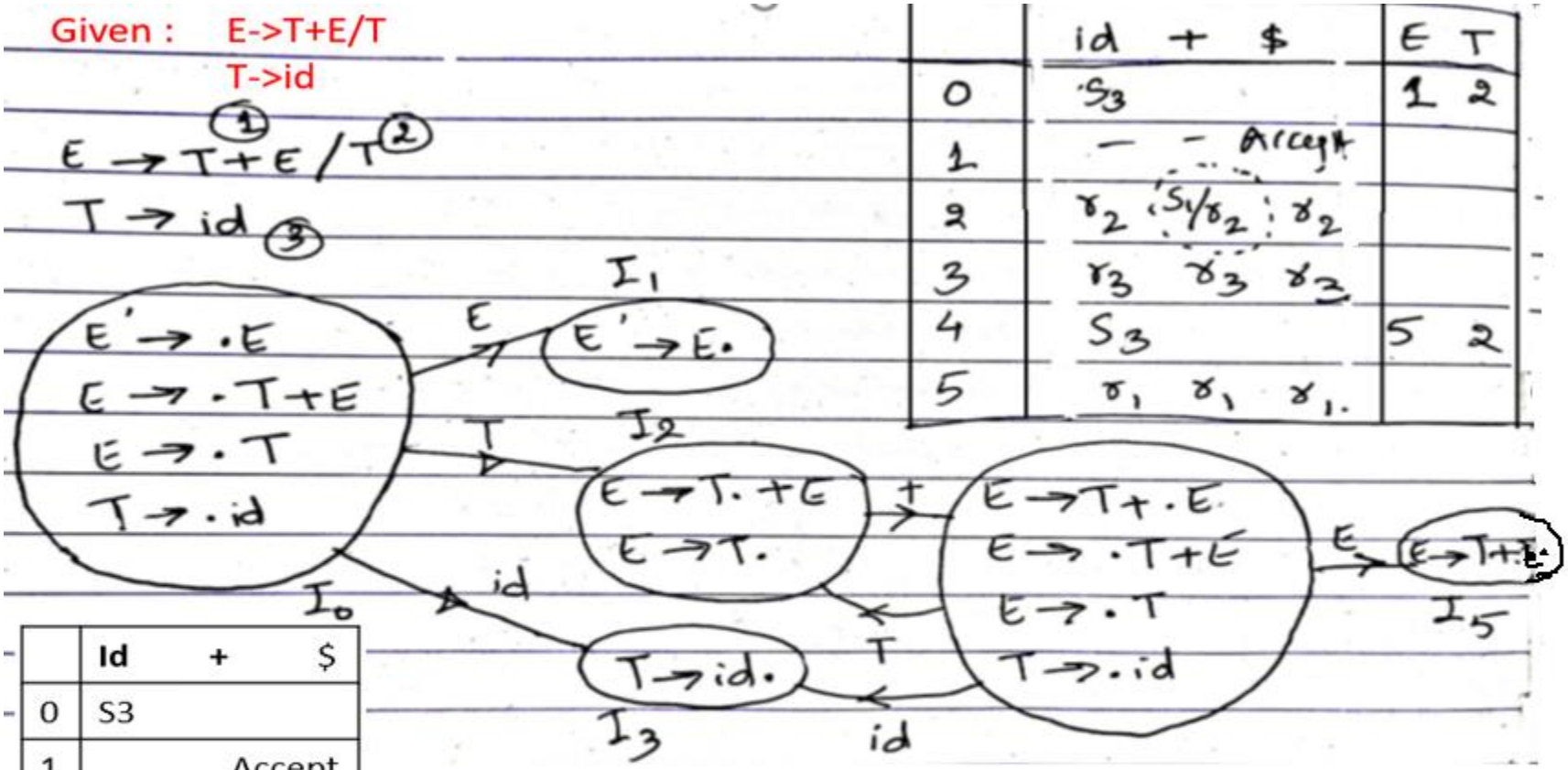
In the SLR (1) parsing, we place the reduce move only in the follow of left hand side.

Various steps involved in the SLR (1) Parsing:

- For the given input string write a context free grammar
- Check the ambiguity of the grammar
- Add Augment production in the given grammar
- Create Canonical collection of LR (0) items
- Draw a data flow diagram (DFA)
- Construct a SLR (1) parsing table



SLR (https://www.javatpoint.com/slr-1-parsing)



	id	+	$	E	T
0	S_3			1	2
1	—	—	Accept		
2	r_2	S_4/r_2	r_2		
3	r_3	r_3	r_3		
4	S_3			5	2
5	r_1	r_1	r_1		

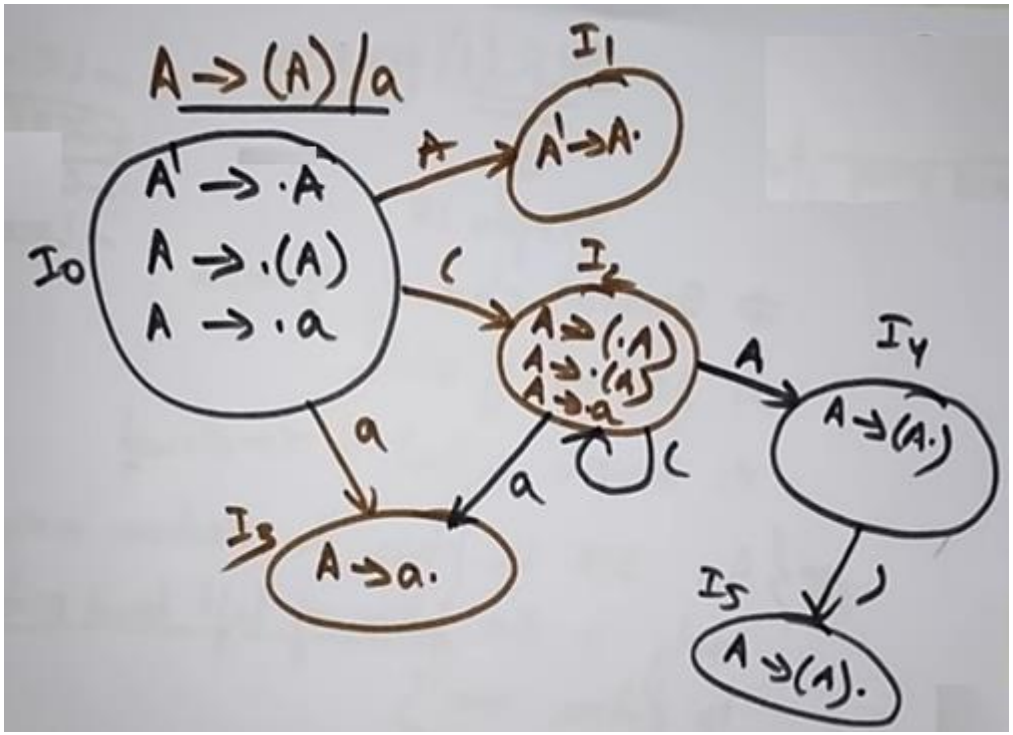
	Id	+	$
0	S3		
1			Accept
2		S4	r2
3		r3	r3
4	S3		
5			r1

Reduced move is written only in the follow of the LHS variable of corresponding production. E.g. r2 written in only follow of E, r3 written in only follow of T and r1 written in follow of E

Example-SLR

Given: A-> (A) | a …. A-> (A) –rule1 & A-> a –rule 2

Augmented rule- A'->A



	Action				Goto
	a	(	)	$	A
0	S3	S2			1
1				Acc	
2	S3	S2			4
3			r2	r2	
4		S5			
5			r1	r1	

Reduce move is written only in the follow of LHS of the corresponding rule.

Follow(A)= {), $ }

LR(0) vs SLR

- LR(0)
  - One LR(0) state mustn't have both shift and reduce items, or two reduce items.
  - So any complete item (dot at end) must be in its own state; parser will always reduce when in this state.
- SLR
  - Peek ahead at input to see if reduction is appropriate.
  - Before reducing by rule A → XYZ, see if the next token is in Follow(A). Reduce *only* in that case. Otherwise, shift.

Semantic Analysis

Semantic analysis checks the semantic consistency of the code. It uses the syntax tree of the previous phase along with the symbol table to verify that the given source code is semantically consistent. It also checks whether the code is conveying an appropriate meaning.

CFG + semantic rules = Syntax Directed Definitions

The semantic analyzer is expected to recognize:

- Type mismatch
- Undeclared variable
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Functions of Semantic analyses phase

- Helps you to store type information gathered and save it in symbol table or syntax tree
- Allows you to perform type checking
- In the case of type mismatch, where there are no exact type correction rules which satisfy the desired operation a semantic error is shown
- Collects type information and checks for type compatibility
- Checks if the source language permits the operands or not

```
float x = 20.2;
float y = x*30;
```

In the above code, the semantic analyzer will typecast the integer 30 to float 30.0 before multiplication

Syntax directed translation

In syntax directed translation, along with the grammar we associate some informal notations and these notations are called as semantic rules.

Grammar + semantic rule = SDT (syntax directed translation)

In syntax directed translation, every non-terminal can get one or more than one attribute or sometimes 0 attribute depending on the type of the attribute. The value of these attributes is evaluated by the semantic rules associated with the production rule.

In the semantic rule, attribute is VAL and an attribute may hold anything like a string, a number, a memory location and a complex record.

In Syntax directed translation, whenever a construct encounters in the programming language then it is translated according to the semantic rules define in that particular programming language.

Production	Semantic Rules
E → E + T	E.val := E.val + T.val
E → T	E.val := T.val
T → T * F	T.val := T.val + F.val
T → F	T.val := F.val
F → (F)	F.val := F.val
F → num	F.val := num.lexval

E.val is one of the attributes of E.

num.lexval is the attribute returned by the lexical analyzer.

Syntax directed translation scheme

The Syntax directed translation scheme is a context -free grammar.

The syntax directed translation scheme is used to evaluate the order of semantic rules.

In translation scheme, the semantic rules are embedded within the right side of the productions.

The position at which an action is to be executed is shown by enclosed between braces. It is written within the right side of the production.

Annotated Parse Tree – The parse tree containing the values of attributes at each node for given input string is called annotated or decorated parse tree.

Features –

High level specification

Hides implementation details

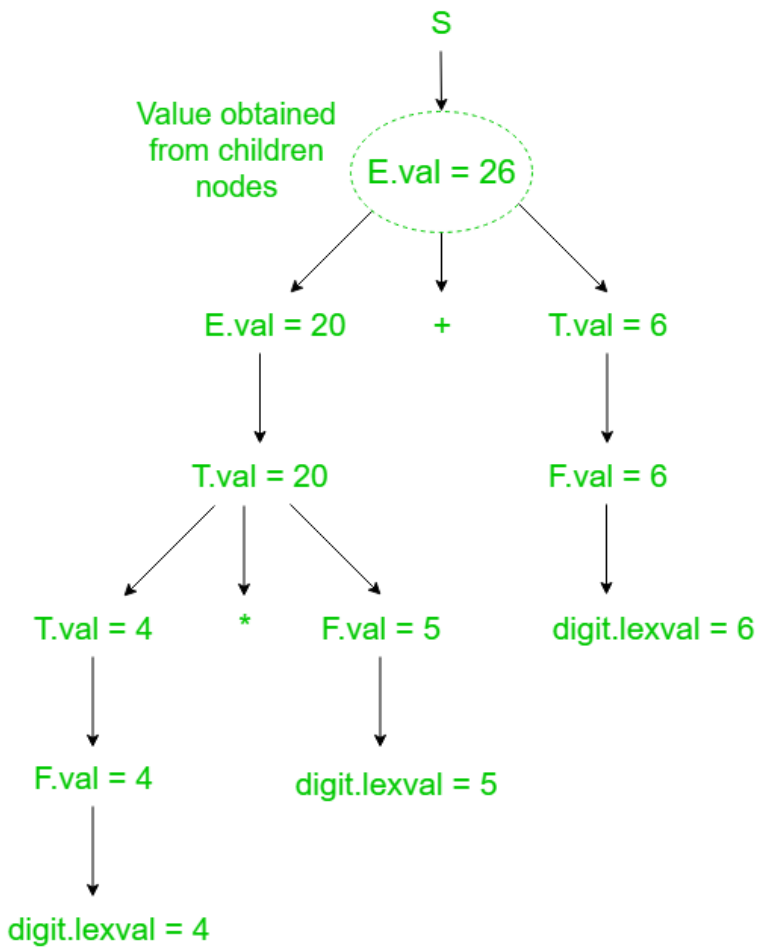
Explicit order of evaluation is not specified

Implementation of SDT

Consider a grammar: an input string 4*5+6

```
S --> E
E --> E1 + T
E --> T
T --> T1 * F
T --> F
F --> digit
```

Production	Semantic Actions
S --> E	Print(E.val)
E --> E_1 + T	E.val = E_1.val + T.val
E --> T	E.val = T.val
T --> T_1 * F	T.val = T_1.val * F.val
T --> F	T.val = F.val
F --> digit	F.val = digit.lexval



Annotated Parse Tree

Implementation of SDT

Syntax direct translation is implemented by constructing a parse tree and performing the actions in a left to right depth first order.

SDT is implementing by parsing the input and produces a parse tree as a result.

Production	Semantic Rules
S → E $	{ printE.VAL }
E → E + E	{E.VAL := E.VAL + E.VAL }
E → E * E	{E.VAL := E.VAL * E.VAL }
E → (E)	{E.VAL := E.VAL }
E → I	{E.VAL := I.VAL }
I → I digit	{I.VAL := 10 * I.VAL + LEXVAL }
I → digit	{ I.VAL:= LEXVAL}

23*5+4