Unit-2- Lexical Analyzer.

* Role of Lexical Analyzer:

Basic role of lexical analyzer is to take the input as high-level language divides it into smallest units called as lexical units, group them into meaningful sequence called as lexemes. If lexeme matches any of the pattern then tokens are generated. If it identifies the lexeme as identifier, then it enters the id into symbol table.

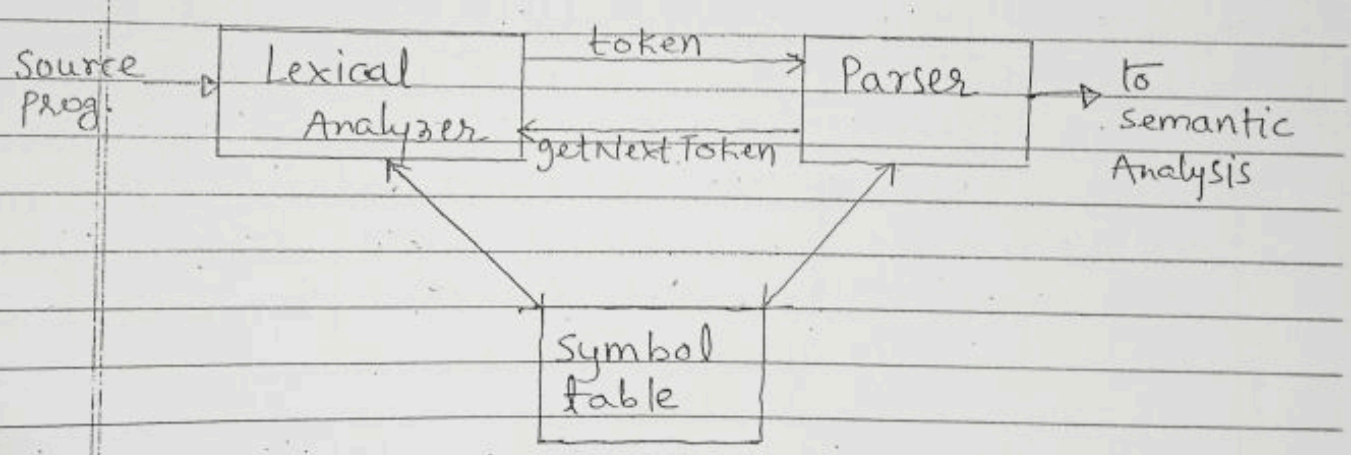


Fig: Illustrates interaction between lexical Analyzer and Parser.

The call getNextToken in the above figure, requests lexical analyzer to send next token to the parser. The process will continue until lexical analyzer sends all the tokens.

Apart from sending tokens to the parser, lexical analyzer also performs three important task

(i) Stripping out whitespaces and comments

(ii) correlating error messages along with the line number. It keep tracks of number of new lines, whenever an error occurs, at that particular moments it sends that error message along with the line number.

(iii) If the input contains macros, expansion of macros is also done by lexical Analyzer.

* Tokens, Lexeme and Patterns

Tokens: Token is a pair containing token name and optional attribute value. The token name is an abstract symbol representing the name of the token. Eg for few of the token names are Id, keyword, num etc.

Lexeme: A lexeme is sequence of characters in the source program when it matches pattern for token ~~that~~ are generated & is identified by the lexical Analyzer as an instance of that token.

Pattern: Pattern is a description of the form that the lexeme of the token may take. In case of identifier the pattern can be sequence of letters followed by any number of letters and digits

Eg: Letter (letter/digit)*

Page No.
Date

* Lexical Errors:

There are two cases lexical errors can be studied.

1st case: Sometimes it is hard for a lexical analyzer to tell, without the help of other components, that there is error in the source code.

For instance, if a string 'fi' occurs which is infact reads as 'if' in a statement

```
if(a==b)
```

So in the above eg eventhough 'fi' is a misspelling of keyword 'if', the lexical analyzer not able to identify it, as it is still matches to the pattern called as identifier. In this situation other phases of the compiler identifies such kinds of errors.

2nd case: Sometimes it is not possible for lexical analyzer to generate token only, when input lexeme does not matches with any of the pattern in the system. The simplest recovery strategy in this situation is "Panic mode recovery" technique.

In this technique, we delete successive characters from the input until the remaining input matches any of the pattern so that tokens can be generated.

other possible error recovery techniques are

(i) Delete one character from the remaining input.
(ii) Insert a missing character into the remaining input
(iii) Replace one character by another character
(iv) Transponse two adjacent characters.

Page Nr.
Date

* Specification of Tokens:

Regular expressions are important way to represent regular expression patterns. So, in this section we are going to study some of the notations of regular expression.

Alphabets, Strings and Languages:

Alphabets: is a finite set of symbols. Typical example of symbols are letters, digits and punctuation. The set {0,1} is a binary alphabet.

Strings: is a finite sequence of symbols which are drawn from alphabet. Strings are normally represented using a greek symbols α, β & γ
The length of the string is nothing but number of occurrences in that string.

For eg: α = TKIET
the length of α which is represented by $|\alpha|$ is 5

Concatenation of two string is nothing but apperding second string to the end of first string

For eg: α = TKIET & β = warana
is represented as
$\alpha \cdot \beta = \alpha\beta$ = TKIETwarana.

Language: A language is any countable set of

Strings over some fixed alphabet.

* Operations on languages:

There are three important operations on languages
(i) union
(ii) Concatenation
(iii) closure
    (a) Kleene closure
    (b) Positive closure

(i) Union: Union is a familiar is familiar operation on set, and is nothing but taking all the components from two sets.

(ii) Concatenation: Concatenation of two languages are nothing but, taking string from one language and taking string from another language and concatenating them.

(iii) closure
    (a) Kleene closure: Kleene closure of a language 'L' is nothing but concatenating 'L' only '0' or more times. It is represented as L^*.

    (b) Positive closure: Positive closure of a language 'L' is nothing but concatenating 'L' only 1 or more times. It is represented as L^+.

Table in the next page represents definitions of operations on languages.

operations	Definition and Notation
Union of L & M	$L \cup M = \{s ; s \text{ is in } L \text{ or } s \text{ is in } M\}$
Concatenation of L & M	$LM = \{st : s \text{ is in } L \text{ and } t \text{ is in } M\}$
Kleene closure of L	$L^* = \bigcup_{i=0}^{\infty} L^i$
Positive closure of L	$L^+ = \bigcup_{i=1}^{\infty} L^i$

Table: Definition of operations on Languages.

E.g: Let L be the set of Letters $L = \{A-Z a-z\}$ and D be the set of digits $D = \{0, 1 \cdots 9\}$

1. LUD is the set of letters and digits - It produces the language with 62-strings of length one, each of which strings is either one letter or one digit.
2. LD is the set of 520 strings of length two, each consisting of one letter followed by one digit.
3. L^4 is set of all 4-letter strings.
4. L^* is the set of all strings of letters, including ε the empty string.
5. $L(L \cup D)^*$ is the set of all strings of letters and digits beginning with a letter.

Regular expression:

We have seen some of the notations in the previous section. using those notations we can represent regular expression pattern.

for Eg: Regular expression pattern for identifies can be written as

Letter_ (Letter | digit)*

in above pattern Letter & digit are alphabets. juxtaposition (_) represents concatenation, '|' indicates union and * represents kleene closure.

Let us see some of the regular expression which produces different languages.

Let $\Sigma = \{a, b\}$

1. The regular expression a|b denotes the language {a, b}.

2. (a|b)(a|b) denotes the language {aa, ab, ba, bb}

3. a* denotes the language consisting of all strings of zero or more a's that is $\{\epsilon, a, aa, aaa \ldots\}$

4. (a|b)* denotes the language $\{\epsilon, a, b, aa, ab, ba, bb, aaa \ldots\}$

5. a|a*b denotes the language {a, b, ab, aab, aaab}

Page No.
Date

Recognition of Tokens:

In this section we study some of the tokens and its patterns.

digit → [0-9]
digits → [0-9]+
number → digits (. digits)? (E [+-]? digits)
letter → [A-Za-z]
id → letter (letter | digit)*
if → if
else → else
then → then
relop → < | > | <= | >= | = | <>

In addition lexical analyzer strip out whitespace and comments by recognizing a token "ws" which defined as

ws → (blank | tab | newline)+

Token 'ws' is different from other tokens in that when we recognize it, we do not return it to the parser, but rather restart the process of lexical analysis from the character followed by the whitespace.

Transition diagrams:

Regular expression patterns are converted into stylized flowcharts called "transition diagram"

Transition diagrams have a collection of nodes or circles, called states.

Edges are directed from one state of transition diagram to other state with a given input symbol.

All the transition diagram in our examples are deterministic, meaning that there is exactly one edge from one state to other state with given input symbol

Three important conventions about transition diagrams are.

(i) Certain states are said to be accepting, or final. These states indicate that a lexeme has been found. We always indicate an accepting state by a double circle, and if there is an action to be taken- typically returning a token to the parser - we shall attach that action to the accepting state.

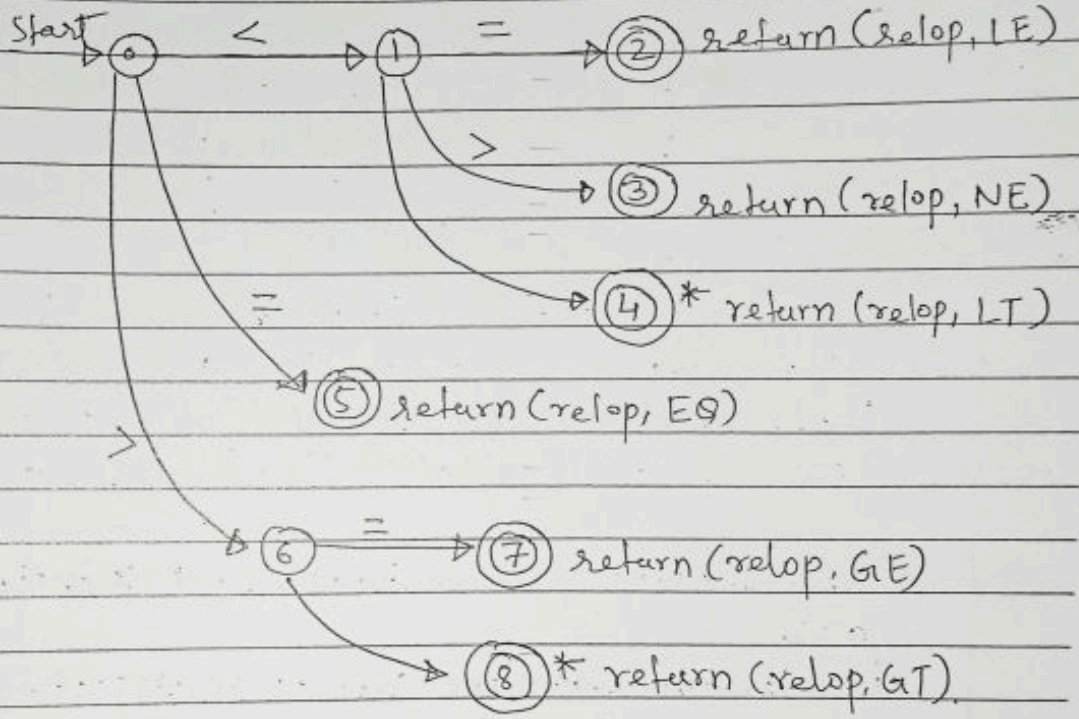
(ii) If forward pointer wants to retract one position back, that can be indicated by putting a single '*' at the accepting state.

(iii) One state is designated as 'start state' or 'initial state', it is indicated by an incoming edge, labeled "start", entering from nowhere.

Page No.
Date

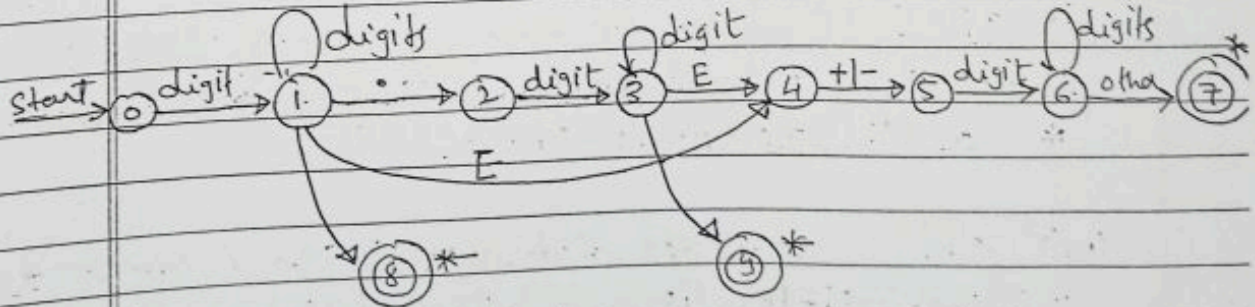
* Transition diagram for relop

relop → < | <= | > | >= | = | <>



* Transition diagram for number

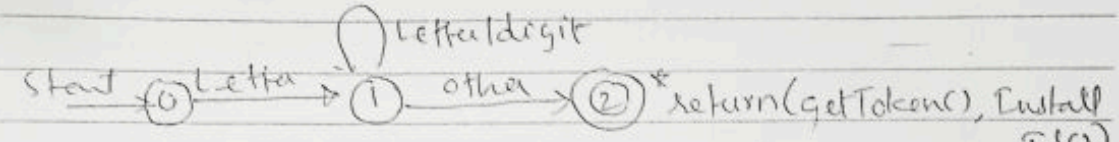
number → digits (.digits)? (E [+-]? digits)?



* Transition diagram for identifiers and keywords (Reserved words).

Same regular expression of transition diagram is used for recognizing both identifiers and keywords.

Id → letter (letter | digit)*

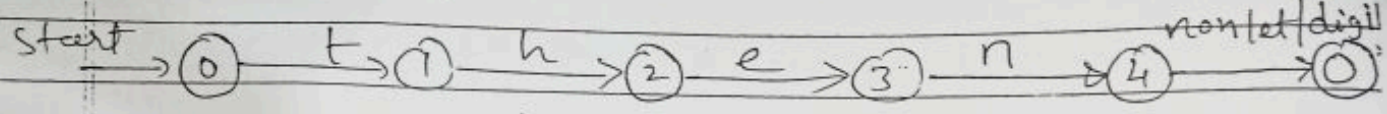


There are two ways to differentiate between reserved words (keywords) and identifiers.

1. In the beginning, enter all the keywords of the language into the symbol table. one of the field of the symbol table identify these as a keywords. When we find an identifier, a call to InstallId() places it in the symbol table. If we found keyword we call a function getToken() which compares the lexeme keyword with all the entries of the symbol table, if match found it means it is keyword and generate token as keyword. If the input lexeme matches the entry which placed after all the keywords then it generates token as identifier.

2. Second method is to generate separate transition diagram for all the keywords:

For eg. the transition diagram for 'then' is

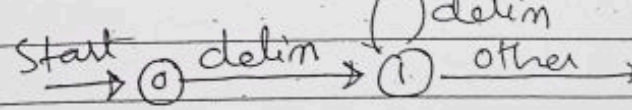


If the input matches both identifier transition diagram and keyword transition diagram, we must prioritize the tokens so that the reserved-words are recognized in preference to id. We do not adopt this approach practically as it is costly and take lot of space.

* Transition diagram for whitespace:

ws → (blank | newline | tab)+

If we consider blank, newline & tab altogether as 'delim' then we can construct transition diagram for ws as follows.



The Lexical-Analyzer Generator LEX:

LEX is a software, which automatically generates lexical analyzer by taking input as a lexical specification.

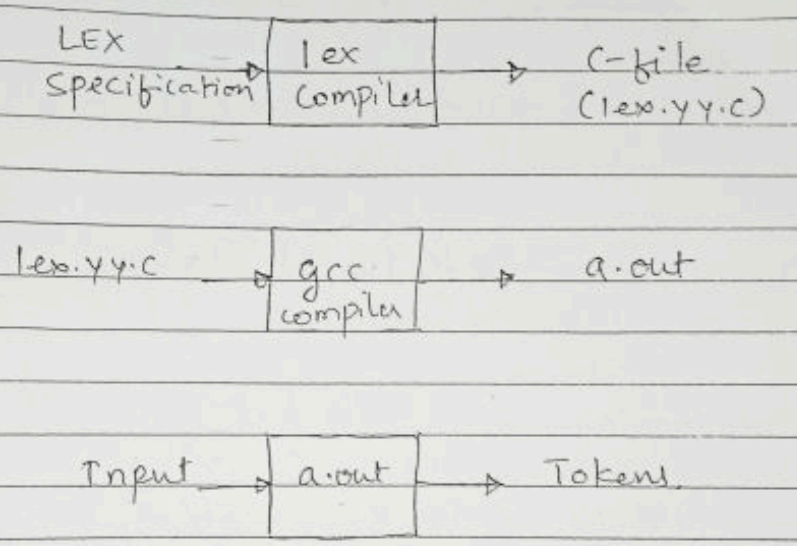


Fig: Illustrates How lex generates Lexical analyzer.

Lex compiler takes input as a lex specification and produces output as C-file.

gcc-compiler takes input as C-file (lex.yy.c) which is generated by lex compiler and produces output as a.out, which works as a lexical analyzer. Lexical analyzer in-turn takes any high-level language input and produces output as a token.

Commands to compile and run lex program:

To compile lex program by lex compiler we use command as

lex filename.l

which produces output as 'lex.yy.c' which is a C-file

To compile C-file by gcc-compiler we use following command.

gcc lex.yy.c -ll

which produces output as 'a.out' which works as a lexical analyzer

To execute lexical analyzer we use following command.

./a.out

syntax of lex specification:

```
%{
    Declaration / Definition
%}
%%
    Translation Rules
    (Regular expression)
    [Patterns] {Action}
%%
    user defined function.
```

Regular expression further can be written as

```
%%
[Pattern 1]  {Action 1}
[Pattern 2]  {Action 2}
    .            .
[Pattern n]  {Action n}
%%
```

whenever a particular input matches any pattern, its corresponding action will execute.

Eg: LEX program for identifying and generating tokens for vowels and consonants.

```
%{
    int v, c;
%}
%%
[AEIOUaeiou] {Printf("%s is a vowel\n",
                yytext); v++;}
[A-Za-z] {Printf("%s is a consonant\n",
                yytext); c++;}
%%
main()
{
 Printf(" enter the input string\n");
 yylex();
}
```

* Finite automata:

Finite automata simply say 'yes' or 'no' for the possible input string.

There are two types of finite automa
1. NFA (Non-deterministic finite automa)
2. DFA (Deterministic finite automata)

Page No.
Date

Unit-3 - Syntax Analysis.

Syntax Analyzer takes input as tokens generated by lexical Analyzer, checks tokens are valid or not by passing tokens into the grammar. If grammar accepts the tokens, then it considers that tokens are valid and generate parse tree else generates syntax error.

Role of Syntax Analyzer: (Role of Parser):

In the compiler model, normally parser obtains a string of tokens from the lexical analyzer and verifies tokens are valid or not. Below diagram depicts the position of Parser in the compiler model.

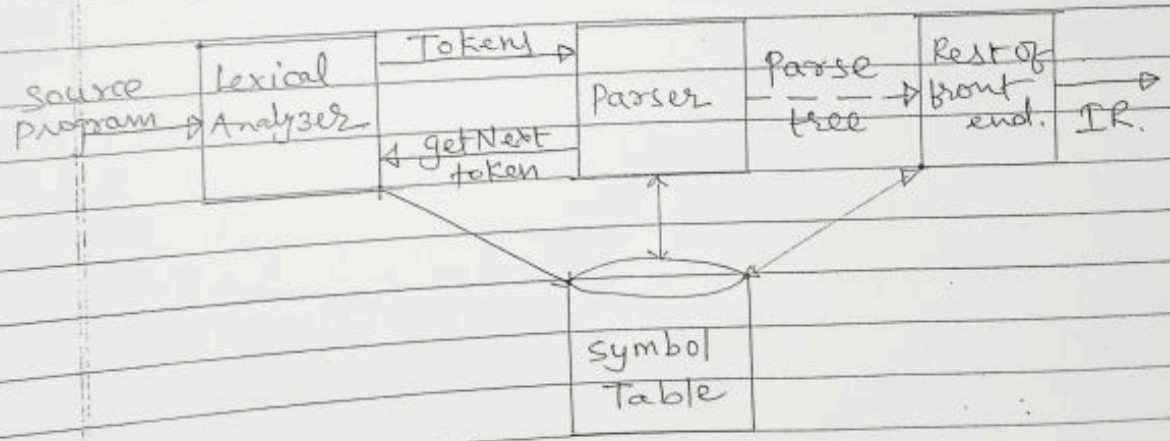


Fig:- illustrates position of Parser in the compiler model.

If tokens are accepted in the grammar, syntax Analyzer/parser generates parse tree.

To construct the parse tree, we have mainly two types of Parsing techniques.

Context free grammer : consists of

Page No.
Date

1. Terminal symbols: are the basic symbols from which strings are formed. Terminal symbols are normally considered as token name. Terminal symbols are last element in the language meaning that terminal symbols cannot be replaced by any other terminal or non-terminal symbols.

Some of the possible terminal symbols are:

(i) All lower case letters
(ii) All punctuation symbols
(iii) All arithmatic and logical operators
(iv) All digits
(v) and anything written in boldface letter is also considered as terminal symbols.

2. Non-terminal symbols :→ Non-terminal symbols are normally represented by capital letters. Non-terminal symbols can be further replaced by any other terminal or non-terminal symbols.

3. In a grammar, one non-terminal is distinguished as the start symbol. Normally left side non-terminal symbol of first production is considered as start symbol.

4. Production: Production is considered as rewriting rule. is a rule of a grammar. syntax of the production is

NT → Strings of terminal and non-terminal.

Left side of the production always contains single non-terminal and right side of the production contains either single terminal or non-terminal or combination of both terminal and non-terminal.

Page No.
Date

Eg: $E \rightarrow E+T/T$
$T \rightarrow T*f/f$
$F \rightarrow (E)/id.$

Above grammar is example for asithmatic operati + and *.

Here E, T & F are Considered as non-termical symbols.

+, *, (,) & id are considered as termical symbols.

E' start symbol of the grammar and,

$E \rightarrow E+T$
$E \rightarrow T$
$T \rightarrow T*F$
$T \rightarrow F$
$F \rightarrow (E)$
$F \rightarrow id$

are productions.

Ambigous grammar:

A grammar is said to be ambigous grammar, when it produces more than one parse tree for given input string.

How to eliminate ambiguity of the grammar?

Ambiguity of the grammar can be eliminated by rewriting / modifying the grammar.

for Eg:

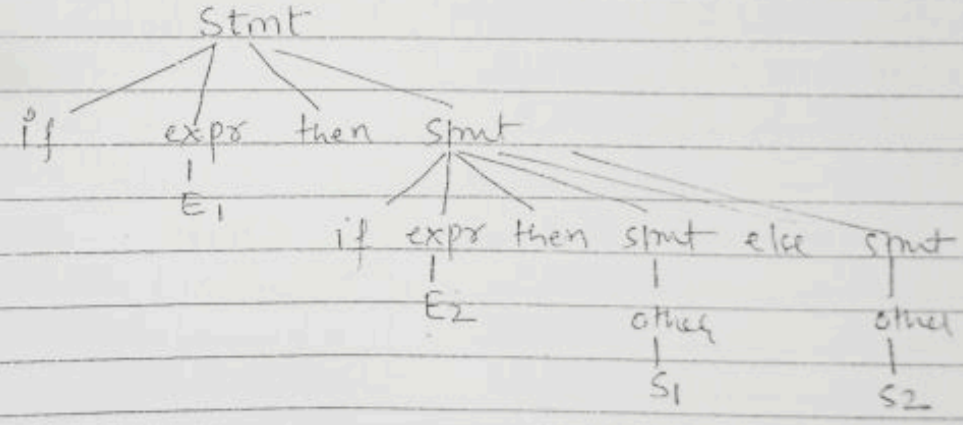
Let us consider a grammar

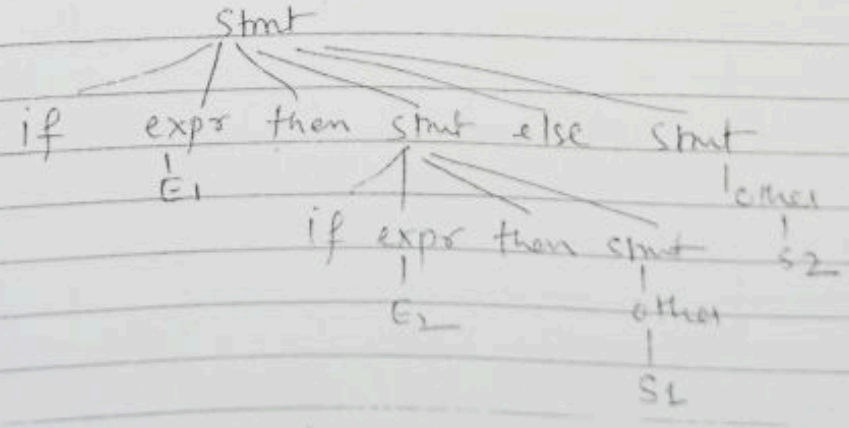
Stmt → if expr then stmt | if expr then stmt else stmt | other

and input string is

if E_1 then if E_2 then S_1 else S_2

Let us check how many Parse above grammar produces for the given Input string



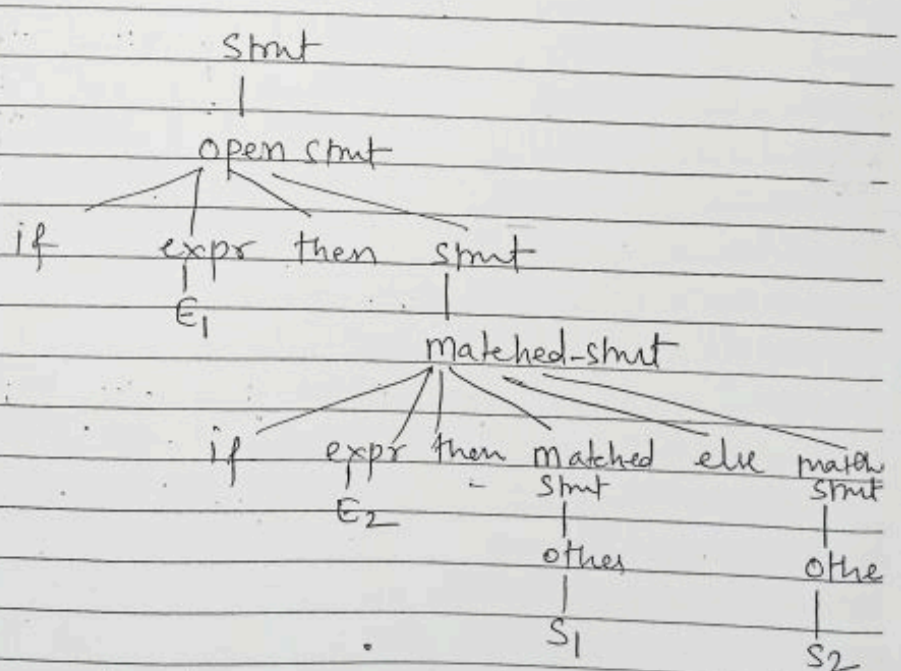


above grammar produces more than one parse tree for given Input string so, grammar is ambiguous.

By rewriting the grammar, we can eliminate ambiguity of the grammar.
rewriting the grammar is completely depend upon compiler developer, what logic they have used based on that grammar is modified.

Let us modify the grammar and we will check how many parse tree it generates.

stmt → matched_stmt | open_stmt
matched-stmt → if expr then matched_stmt else matched-stmt | others
open stmt → if expr then stmt |
if expr then matched-stmt else open-stmt



So, above modified grammar produces only one parse-tree hence, grammar is unambiguous grammar.

Left Recursion grammar:-

A grammar is left recursive if it has a non-terminal A such that there is a derivation $A \Rightarrow A\alpha$ for some string α.

(or)

If left side non-terminal symbol is equals to or same as that of left most non-terminal symbol of RHS side than the grammar is said to be left recursive grammar.

Note:- * Top-down passing technique does not supports grammar, if it contains left recursion in it. So, we need to left recursion from the grammar if at all if we want to use it for top-down passing technique.

if the production is

$$A \rightarrow A\alpha | \beta$$

then left recursion can be eliminated by rewriting the above grammar as

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' | \varepsilon$$

} Free from left recursion.

For Eg:

Let us consider a grammar

$$E \rightarrow E+T | T$$
$$T \rightarrow T*F | F$$
$$F \rightarrow (E) | id$$

Page No.
Date

above grammar is left recursion since, first Production, where left side Non-terminal symbol is equals to left most non-terminal symbol of RHS side. So left recursion can be eliminated using above rule as.

$E \rightarrow TE'$
$E' \rightarrow +TE' | \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' | \epsilon$
$f \rightarrow (E) | id.$

Algorithm for eliminating left recursion:

Input: Grammar G with no cycles of ϵ-production

output: An equivalent grammar with no left recursion.

Method: Apply below algo. to eliminate left recursion.

1 arrange the non-terminals in some order $A_1, A_2, \ldots A_n$
2 for (each i from 1 to n) {
3 for (each j from 1 to i-1) {
4 replace each production of the form $A_i \rightarrow A_j\gamma$ by the productions $A_i \rightarrow \delta_1\gamma | \delta_2\gamma | \ldots | \delta_k\gamma$, where $A_j \rightarrow \delta_1 | \delta_2 | \ldots | \delta_k$ are all current A_j productions.
}
}

Left Factoring :-

Left factoring is a grammar transformation that is useful for producing grammar suitable for predictive or top-down passing technique.

When the choice between two alternative A-productions is not clear, we may rewrite the productions in such way that there is no confusion in choosing A-production.

For eg. if we have the production in general as

$$A \to \alpha\beta_1 \mid \alpha\beta_2$$

are two A-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$. So to overcome this problem we rewrite the above grammar as

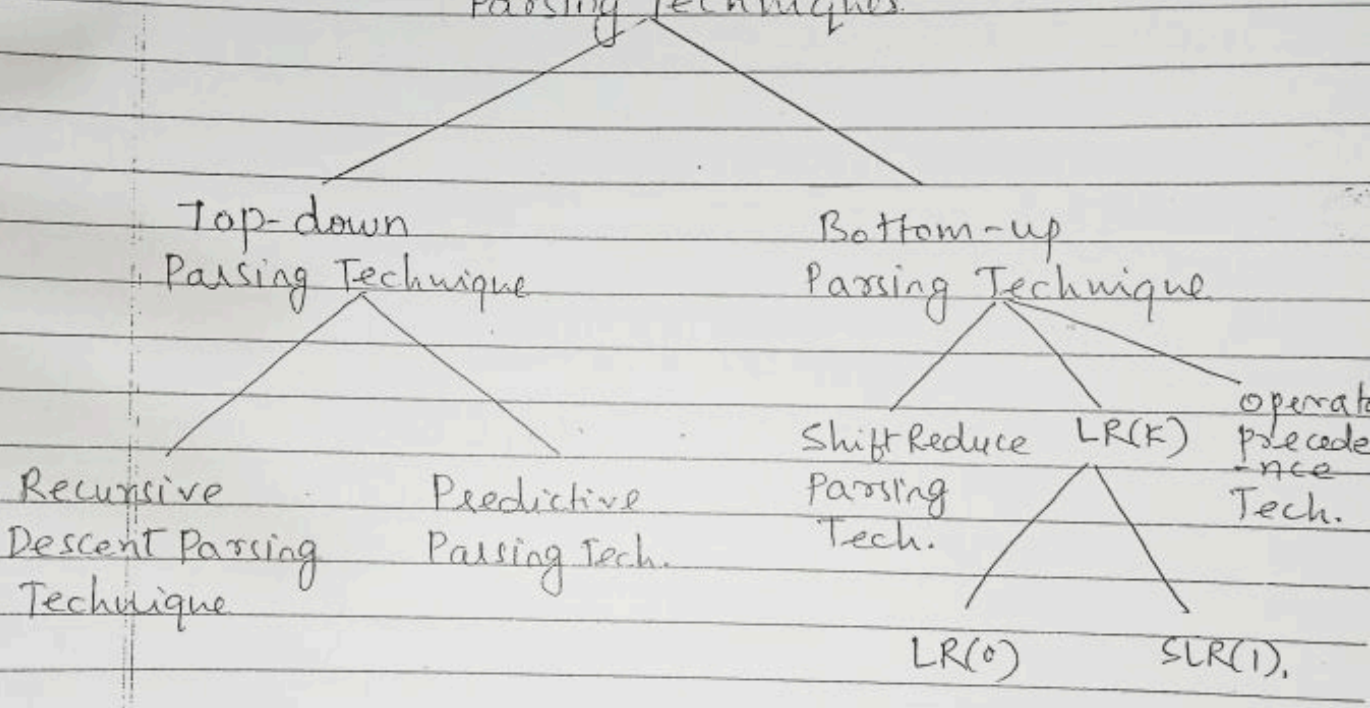
$$\left.\begin{aligned} A &\to \alpha A' \\ A' &\to \beta_1 \mid \beta_2 \end{aligned}\right\} \text{ Left Factored grammar.}$$

In the above modified grammar A can be expand to $\alpha A'$. Then, after seeing the input derived from α, we expand A' to β_1 or β_2.

For eg: $S \to iEtS \mid iEtSeS \mid a$
$E \to b$

above grammar can be left factored as

$S \rightarrow iEtSS' | a$
$S' \rightarrow eS | \epsilon$
$E \rightarrow b.$

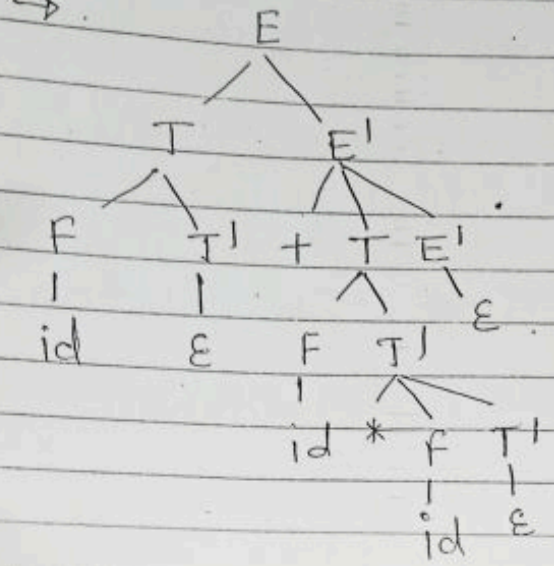


Top-down parsing Technique:

In top down parsing technique, Parse tree will be constructed from top (root) to bottom (leaves) top down parsing technique is also called as 'derivation'.

Consider a grammar

$E \rightarrow TE'$
$E' \rightarrow +TE' | \varepsilon$
$T \rightarrow fT'$
$T' \rightarrow *fT' | \varepsilon$
$F \rightarrow (E) | id$



Derivation: In the derivation process we, initially take start symbol of the grammar and at every step we replace by its right side production. At the end if we get input string then consider that input is valid else invalid.

Types of derivation:

There are two types of derivation

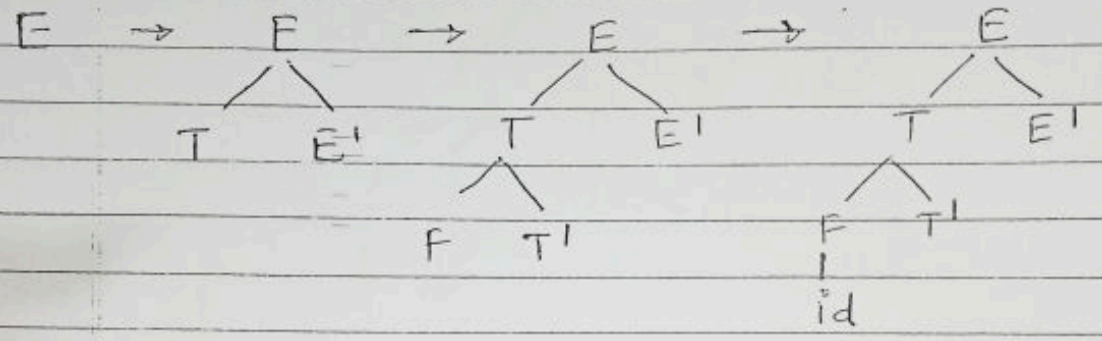
(i) Left most derivation
(ii) Right most derivation.

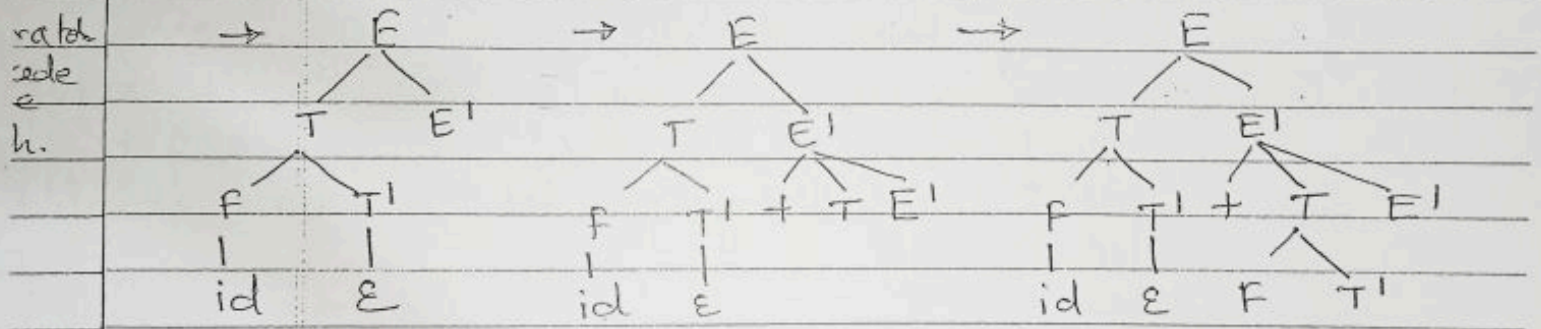
Left most derivation: In the left-most derivation, the left most non-terminal symbols of RHS is replaced by its production.

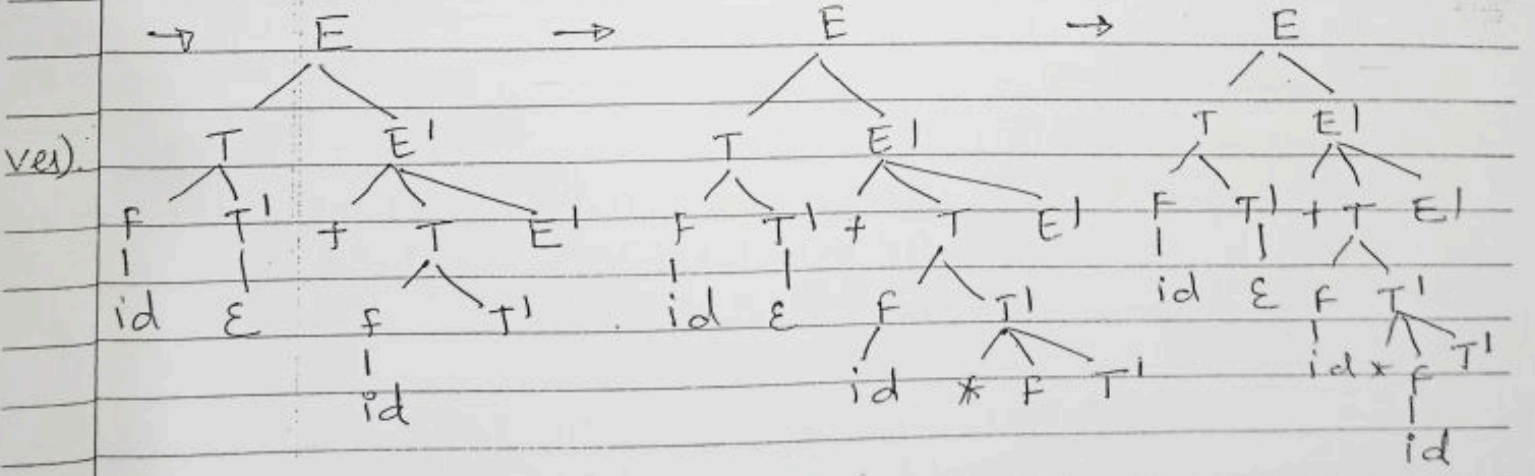
Right most derivation: In the Right most-derivation the right most non-terminal symbol of RHS is replaced by its Production.

and input string as id+id*id

Construction of Parse tree using top-down parsing technique is as follows.







Let us consider an eg grammar

$E \to E+T \mid T$
$T \to T*F \mid F$
$F \to (E) \mid id$

and input string is id+id*id.

Eg. Left most derivation:

$$E \xrightarrow{lm} E+T \xrightarrow{lm} T+T \xrightarrow{lm} f+T \xrightarrow{lm} id+T \xrightarrow{lm} id+T*f \xrightarrow{lm} id+f*f \xrightarrow{lm} id+id*f \xrightarrow{lm} id+id*id.$$

Eg. Right most derivation:

$$E \xrightarrow{rm} E+T \xrightarrow{rm} E+T*f \xrightarrow{rm} E+T*id \xrightarrow{rm} E+f*id \xrightarrow{rm} E+id*id \xrightarrow{rm} T+id*id \xrightarrow{rm} f+id*id \xrightarrow{rm} id+id*id.$$

* Note: Top-down-parsing technique supports left most derivation and bottom-up parsing technique supports right most derivation.

Recursive Descent Parsing Technique:

Recursive Descent Parsing technique some-time requires backtracking process to find correct A-production.

As backtracking process increases the time and reduces the efficiency, normally recursive descent parsing technique not used practically.

Let us see the below eg. to understand recursive descent parsing technique.

Consider the grammar

$$S \rightarrow cAd$$
$$A \rightarrow ab \mid a$$

to construct a parse tree top-down for the i/p string $w = cad$, begin with a tree consisting of a single node labeled S, and the input pointer pointing to c, the first symbol of w. S has only one production, so we use it to expand S & obtain the tree of figure (a). the left most leaf, labeled c, matches the first symbol of input w, so we advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A.

Now, we expand A using the first alternative $A \rightarrow a$ to obtain the tree of fig(b). We have a match for the second input symbol, a. So, we advance the input pointer to d, the third input

symbol 'a' and compare 'd' against the next leaf labeled b. Since b does not d, we report failure and go back to A-production to see whether there is another alternative for A that has not been tried, but that might produce a match.

In going back to A, we must reset the input pointer to position 2, the position it had we first came to A.

The second alternative for A produces tree of figure (c). The leaf a matches the second symbol of w and the leaf d matches the third symbol.

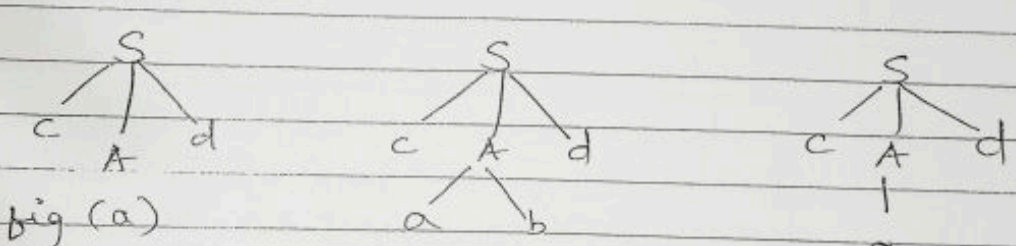


fig (a) fig (b) fig (c).

Fig: Illustrates steps in Recursive descent top-down parsing technique.

Algorithm (Recursive Descent Parsing)

```
void A(){
1.  Choose an A-production A → X1, X2 -- Xk ;
2.  for (i = 1 to k) {
3.      if (Xi is a terminal)
4.          call procedure Xi()
5.      else if (Xi equals to current i/p symbol a)
6.          advance the i/p to next symbol ;
7.      else an error.
    }
```

To allow backtracking in the above algorithm, first we cannot choose a unique A-production at line (1), so we must try each of several productions in some order. Then failure at line (7) is not ultimate failure, but suggests only that we need to return at line (1) and try another A-production.

As it requires more time to parse the input string we go to the another technique called as Predictive Parsing technique (Non-recursive parsing technique).

Predictive Parsing Technique

Predictive parsing, a special case of recursive-descent parsing, where no backtracking is required. Predictive parsing chooses the correct A-production by looking ahead at the input a fixed number of symbols, typically we may look only at one (that is, the next input symbol).

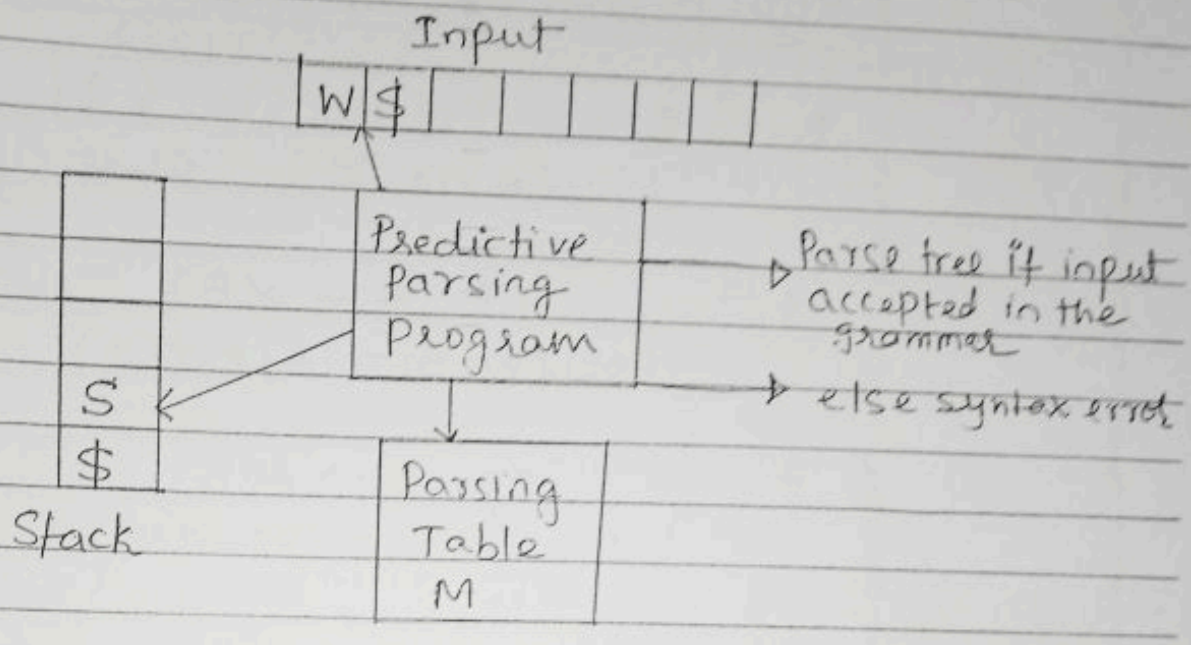
Predictive parsers, that is, recursive-descent parsers needing no backtracking can be constructed for a class of grammar called LL(1).

* LL(1) Grammars:-

The first "L" in LL(1) stands for scanning the input from Left to Right, the second 'L' for producing left most derivation, and the '1' for using one input symbol of lookahead at

each step to make parsing action decisions.

Model of Predictive Parser of LL(1) grammars:-



In above Predictive parser model, we will be given Input 'w', a grammar G and Predictive parsing program. Now we need to construct predictive parsing table M.

To construct predictive parsing table M, we need to find out FIRST set and FOLLOW set of the given grammar.

Rules to find out FIRST set of the grammar.

Basic Rule: FIRST(terminal) = {terminal}

1. If $A \to a\alpha$
   FIRST(A) = {a}

2. If $A \to B\alpha$ where $B \neq \varepsilon$
   FIRST(A) = FIRST(B)

Page No.
Date

3. If $A \rightarrow B\alpha$ where $B = \varepsilon$

$FIRST(A) = \{FIRST(B) - \varepsilon\} \cup \{FIRST(\alpha)\}$

Rules to find out FOLLOW set of the grammar

Basic Rule: $FOLLOW(S) = \{\$\}$

where 'S' is start symbol of the grammar

1. If $A \rightarrow \alpha B\beta$ where $\beta \neq \varepsilon$

$FOLLOW(B) = FIRST(\beta)$

2. If $A \rightarrow \alpha B$

$FOLLOW(B) = FOLLOW(A)$

3. If $A \rightarrow \alpha B\beta$ where $\beta = \varepsilon$

$FOLLOW(B) = \{FIRST(\beta) - \varepsilon\} \cup \{FOLLOW(A)\}$

Example:

1. $S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

Find out FIRST set and FOLLOW set of the grammar.

$FIRST(S) = FIRST(A) = \{a\}$

$FIRST(B) = \{b\}$

$FOLLOW(S) = \{\$\}$

$FOLLOW(A) = FIRST(B) = \{b\}$

$FOLLOW(B) = FOLLOW(S) = \{\$\}$

x)}

2. $S \to AB$
$A \to a/\varepsilon$
$B \to b/\varepsilon$

$FIRST(S) = \{FIRST(A) - \varepsilon\} \cup \{FIRST(B)\}$
$= \{(a, \varepsilon) - \varepsilon\} \cup \{b\}$
$= \{a\} \cup \{b\}$
$= \{a, b\}$

$FIRST(A) = \{a, \varepsilon\}$
$FIRST(B) = \{b, \varepsilon\}$

$FOLLOW(S) = \{\$\}$
$FOLLOW(A) = \{FIRST(B) - \varepsilon\} \cup \{FOLLOW(S)\}$
$= \{(b, \varepsilon) - \varepsilon\} \cup \{\$\}$
$= \{b\} \cup \{\$\}$
$= \{b, \$\}$

3. $S \to iEtS / iEtSeS / b$
$E \to a$

$FIRST(S) = \{i, b\}$
$FIRST(E) = \{a\}$

$FOLLOW(S) = \{\$, e\}$
$FOLLOW(E) = \{t\}$

4. $E \to TE'$
$E' \to +TE' / \varepsilon$
$T \to FT'$
$T' \to *FT' / \varepsilon$
$F \to (E) / id$

Page No.
Date

FIRST(E) = FIRST(T) = FIRST(F) = { (, id }
FIRST(E') = { +, ε }
FIRST(T') = { *, ε }

FOLLOW(E) = { $,) }
FOLLOW(E') = FOLLOW(E) = { $,) }
FOLLOW(T) = { FIRST(E') - ε } ∪ { FOLLOW(E) }
= { (+, ε) - ε } ∪ { ($,) }
= { + } ∪ { $,) }
= { +, $,) }
FOLLOW(T') = FOLLOW(T) = { +, $,) }
FOLLOW(F) = { FIRST(T') - ε } ∪ { FOLLOW(T) }
= { (*, ε) - ε } ∪ { +, $,) }
= { * } ∪ { +, $,) }
= { +, *, $,) }

Once we find out FIRST set and FOLLOW set of the grammar, we can construct Predictive Parsing table M.

Let us consider below grammar

1. S → AB
A → a
B → b

	FIRST	FOLLOW
S	{ a }	{ $ }
A	{ a }	{ b }
B	{ b }	{ $ }

M	a	b	$
S	S → AB		
A	A → a		
B		B → b	

As table does not contain multiple entries, we can say grammar is LL(1).

* [[Question to be asked in the exam:
Check whether the given grammar is in LL(1) or not]]

** Note - 1. If table contains multiple entries then, we say that the given grammar is not in LL(1) and hence it is ambigous grammar. If the grammar is ambigous then it cannot be used to parse any input string.

2. If table does not contain any multiple entries as in the above example, then we say that the given grammar is in LL(1) and hence it is unambigous grammar and can be used to parse any valid input string.

Example - 2:

$E \to TE'$

$E' \to +TE' | \varepsilon$

$T \to FT'$

$T' \to *FT' | \varepsilon$

$F \to (E) | id$

	FIRST	FOLLOW
E	{(, id}	{$,)}
E'	{+, ε}	{$,)}
T	{(, id}	{+, $,)}
T'	{*, ε}	{+, $,)}
F	{(, id}	{+, *, $,)}

M	+	*	(	)	id	$
E			$E \to TE'$		$E \to TE'$	
E'	$+TE'$			$E' \to \epsilon$		$E' \to \epsilon$
T			$T \to FT'$		$T \to FT'$	
T'	$T' \to \varepsilon$	$T' \to *FT'$		$T' \to \epsilon$		$T' \to \epsilon$
F			$F \to (E)$		$F \to id$	

As above table does not contain multiple entries, hence the grammar is in LL(1).

Example-3

S→iEtSS'|a S'→eS|ϵ E→b		FIRST	FOLLOW
	S	{i}	{$, e}
	S'	{e, ε}	{$, e}
	E	{b}	{t}

M	i	t	e	b	$
S	S→iEtSS'				
S'			S'→eS S'→ϵ		S'→ϵ
E				E→b	

As above table contains multiple entries at M[S', e], the grammar is not in LL(1) and hence it is ambigous grammar

* Note:- while constructing Predictive Parsing table, if FIRST set of any non-terminal contains 'ϵ' in it then only we look for FOLLOW set of that Particular non-terminal, to fill the table.

If the table does not contain any multiple entries then grammar is in LL(1) and hence it is unambigous grammar and consequently it can be used to parse any valid input string.

Let us consider example-2 to parse the input string id+id*id$.

As predictive parsing table for example-2 is already constructed, we directly parse the input string as follows.

Matched	Stack	Input	Action
	E$	id+id*id$	
	TE'$	id+id*id$	output E→TE'
	FT'E'$	id+id*id$	output T→FT'
	idT'E'$	id+id*id$	output F→id
id	T'E'$	+id*id$	match id
id	E'$	+id*id$	output T'→ε
id	+TE'$	+id*id$	output E'→+TE'
id+	TE'$	id*id$	match +
id+	FT'E'$	id*id$	output T→FT'
id+	idT'E'$	id*id$	output F→id
id+id	T'E'$	*id$	match id
id+id	*FT'E'$	*id$	output T'→*FT'
id+id*	FT'E'$	id$	match *
id+id*	idT'E'$	id$	output F→id
id+id*id	T'E'$	$	match id
id+id*id	E'$	$	output T'→ε
id+id*id	$	$	output E'→ε

* Predictive parsing algorithm

Input: A string 'w', a parsing table M and a 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 $.

Page No.
Date

```
let a be the first symbol of w;
let X be the top of the stack symbol;
while (X ≠ $){ /* stack not empty */
   if (X=a) pop the stack and let a be the
   next symbol of w;
   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;
   }
}
```

Bottom-up parsing Technique:

Bottom up parse corresponds to the construction of a parse tree for an input string beginning at the leaves (bottom) and working up towards the root (the top).

For Eg:

Consider a grammar

E → E+T | T
T → T*F | F
F → (E) | id

and Input string id * id.

Bottom up parse tree generated for above input using above grammar is as follows.

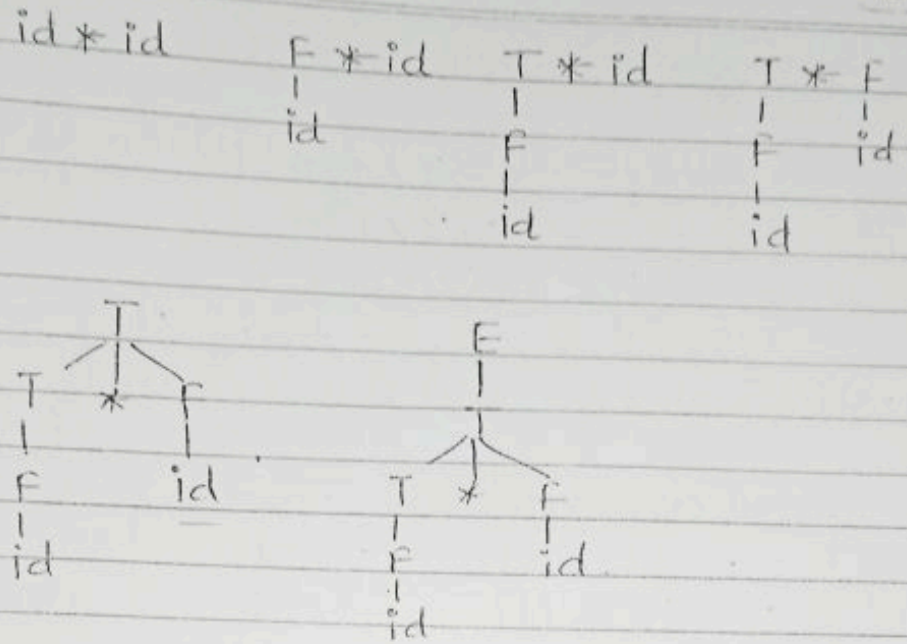


Fig: illustrates A bottom up parse tree for id * id.

Bottom up passing is also known as process of "reduction" a string 'w' to the start symbol of the grammar. At each reduction step, a specific substring matching the body of a production is replaced by the non-terminal at the left side of the production.

For eg: id * id
F * id
T * id
T * F
T
E

Handle: Handle is a substring of the input string, when it matches RHS of the production, whose reduction will gives us start symbol of the grammar.

For Eg: For the grammar

$E \to E+T/T$
$T \to T*f/f$
$f \to (E)/id$

and the input string $id_1 * id_2$ the handle is as shown in the below table.

RIGHT SENTENTIAL FORM	HANDLE	REDUCING PRODUCTION
$id_1 * id_2$	id_1	$f \to id$
$f * id_2$	f	$T \to f$
$T * id_2$	id_2	$f \to id$
$T * f$	$T * f$	$T \to T * f$
T	T	$E \to T$

Shift-Reduce Parsing:-

Shift-Reduce Parsing is a form of bottom-up parsing in which a stack holds grammar symbols and an input buffer holds input string to be parsed.

We use $ to mark the bottom of the stack and also to represent end of the input string.

Initially, the stack is empty, and the string w is on the input, as follows

STACK	Input
$	w$

Final configuration of stack and input after

Page No.
Date

Successful completion of Parsing the input string is

STACK	INPUT
$S	$

where S' is Start symbol of the grammar.

Shift-Reduce parsing technique uses 4-actions
(i) Shift
(ii) Reduce
(iii) Accept
(iv) Error

(i) Shift: Shift the i/p symbol onto the top of the Stack.
(ii) Reduce: When top of the stack matches RHS of the production, then it can be reduced to left side of the production.
(iii) Accept: If input is successfully parsed, then it is an accept action.
(iv) Error: else an error and call an error recovery routine.

For Eg: Consider a grammar

1. S → AB
A → a
B → b

and input string ab.

Stack	Input	Action
$	ab$	Shift a
$a	b$	Reduce A→a
$A	b$	Shift b
$Ab	$	Reduce B→b
$AB	$	Reduce S→AB
$S	$	Accept

total actions used to parse the input is '6'.

Page No.
Date

2. E → T+E
E → T
T → i
and input string is i+i

	Stack	Input	Action.
1	$	i+i$	Shift i.
2	$i	+i$	Reduce T→i
3	$T	+i$	—

At line (3) shift-Reduce parsing technique fall into dilemma that whether to reduce 'T' by E or shift 't' onto the stack. If 'T' is reduced to E, we never get start symbol of the grammar on top of the stack and $ on input.

This situation is called S-R conflict. Similarly it may faces another conflict called as R-R conflict. Whenever shift-Reduce Parsing technique faces either of these conflict, it not able to handle it properly. So, to overcome this problem, we go to another technique called as LR(K).

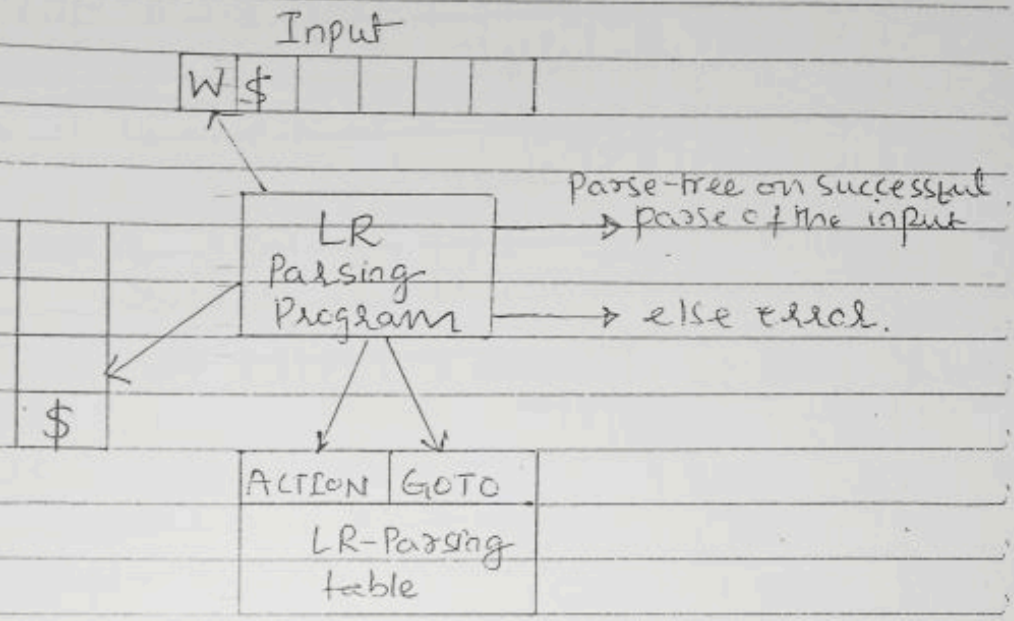
LR(K):

The "L" is for left-to-right scanning of the input, the "R" for constructing a right most derivation and "K" stands for number of symbols lookahead.

Practically K value either '0' or '1' is appreciable. Based on that the first technique is LR(0)

LR(0):

Model of an LR-parser is as below.



Given input string 'w', a grammar 'G' and LR-parsing program, we need to construct LR-parsing table.

Construction of LR-parsing table.

1. Generate Augumented grammar by adding $S' \to S$ to the given grammar to identify accept action.
2. Compute cannonical collection by putting '.' before left-most non-terminal of RHS side of the first production. For instance.

$$A \to .XYZ$$

which means that X is ready to shift as '.' is before X.

$$A \to X.YZ$$

As '.' is before 'Y', now Y is ready to shift.

$$A \to XYZ.$$

once we shift all the symbol as in the above case now it is ready to reduce.

Page No.
Date

3. Construct finite automata.

Note: Step-2 & 3 must be performed simultaneously.

Example-1:

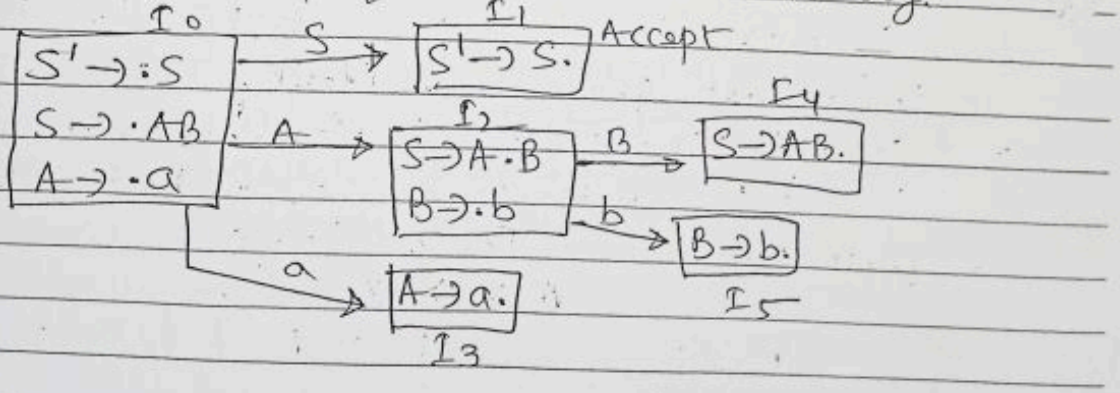
Consider grammar

$S \to AB$
$A \to a$
$B \to b$

Step: 1. Generate augmented grammar by adding $S' \to S$ to the given grammar

$S' \to S$
$S \to AB$
$A \to a$
$B \to b$
} Augumented grammar.

Step: 2 & 3 will be performed simultaneously.



Now we can construct LR-parsing table as follows.

	ACTION			GOTO		
	a	b	$	S	A	B
I_0	S_3			1	2	
I_1			Accept			
I_2		S_5				4
I_3	r_2	r_2	r_2			
I_4	r_1	r_1	r_1			
I_5	r_3	r_3	r_3			

To enter reduce entries, we need to serialize the given grammar like as below

1 $S \rightarrow AB$
2 $A \rightarrow a$
3 $B \rightarrow b$

and look where above productions are reduced in the finite automata, on that particular state need to enter 'r' along with production number as suffix with 'r' on all the input symbol (ACTION).

Since the technique is LR(0), it does not decide where exactly input is reduced, so needs to enter 'r' on all the input symbol in ACTION part of the table.

As the above parsing table does not contain multiple entries, we can say grammar in LR(0).

If the table does not contain any multiple entries then the grammar is in LR(0) and hence the grammar is unambigous grammar and consequently it can be used for parsing any valid input string.

Page No.
Date

Let us consider example-1 to parse the input string "ab".

Stack	Symbol	Input	Action
0	$	ab$	Shift 3
0,3	$a	b$	Reduce A→a
0,2	$A	b$	Shift 5
0,2,5	$Ab	$	Reduce B→b
0,2,4	$AB	$	Reduce S→AB
0,1	$S	$	Accept

Algorithm for LR(0):— (LR-parsing algorithm)

Input: An input string 'w' and LR-parsing table 'M' with ACTION and GOTO function, and a grammar G.

Output: If 'w' is in L(G), the reduction steps of bottom-up parse for w; otherwise an error indication.

Method:- Initially, the parser has 0 on its stack, where '0' is the initial state, and w$ in the input buffer.

```
let a be the first symbol of w$;
while (1) { (* repeat forever *)
    let s be the state on top of the stack;
    if (ACTION[s,a] = shift t) {
        push 't' onto the stack;
        let a be the next input symbol;
    } else if (ACTION[s,a] = reduce A→β) {
        pop |β| symbols off the stack;
        let state 't' now be on top of the stack;
```

push GOTO[t,A] onto the stack;
} else if (ACTION[s,a] = Accept) break;
else call error recovery technique;
}

Drawbacks of LR(0) parsers:

1. LR(0) accepts only small class of LR(0) grammar because of SR & RR - conflicts.
2. The fundamental limitation of LR(0) is that no look ahead symbols are used.

Note: In LR(0), whenever we get a state like

$$\begin{bmatrix} A \to \alpha . a\beta \\ B \to \gamma . \end{bmatrix}$$

we say state contain both shift and reduce action. So, we say this situation as SR-conflict and hence can conclude the grammar is not in LR(0).

Similarly if we get a state like

$$\begin{bmatrix} A \to \alpha . \\ B \to \gamma . \end{bmatrix}$$

We say this situation as RR-conflict and hence can conclude the grammar is not in LR(0)

Page No.
Date

Example for Practice:

1. $E \rightarrow E+T$
   $E \rightarrow T$
   $T \rightarrow T*f$
   $T \rightarrow f$
   $F \rightarrow (E)$
   $F \rightarrow id$

   Check the given grammar is in LR(0) or not

2. $E \rightarrow T+E$
   $E \rightarrow T$
   $T \rightarrow i$

   check the given grammar is LR(0) or not.

To overcome the problem of LR(0), we goto other technique called as SLR(1).

SLR(1)

'S' stands for simple, 'L' stands for left to right and 'R' stands for right most derivation, '1' indicates one symbol lookahead at every reduction step.

SLR(1) is same as that of LR(1) except entering reduce entries in the parsing table.

Also whenever we get SR-conflict and RR-conflict kind of states, in SLR(1) we cannot say directly that the state contains SR-conflict and RR conflict

instead we need to find out following.

* If SLR(1) contains a state like

$A \to \alpha . a\beta$
$B \to \gamma .$

We cannot directly say, the state is SR-conflict
instead we need to find

FOLLOW(B)

if FOLLOW(B) = {a} (symbol after '.')
then we say that state contains SR
conflict and hence grammar is not in SLR(1)

* If SLR(1) contains a state like

$A \to \alpha .$
$B \to \gamma .$

We cannot directly say, the state is RR-conflict
instead We need to find

FOLLOW(A) & FOLLOW(B)

if FOLLOW(A) ∩ FOLLOW(B) ≠ {ϕ}
then we say that state contains RR-conflict
and hence grammar is not in SLR(1).