UNIT – II

TOPDOWN PARSING

1. Context-free Grammars: Definition:

Formally, a context-free grammar G is a 4-tuple G = (V, T, P, S), where:

1. V is a finite set of variables (or nonterminals). These describe sets of "related" strings.
2. T is a finite set of terminals (i.e., tokens).
3. P is a finite set of productions, each of the form

$A \rightarrow \alpha$

where $A \in V$ is a variable, and $\alpha \in (V \cup T)^*$ is a sequence of terminals and nonterminals. $S \in V$ is the start symbol.

Example of CFG:

E ==>EAE | (E) | -E | id A==> + | - | * | / |

Where E, A are the non-terminals while id, +, *, -, /,(,) are the terminals. 2. Syntax analysis:

In syntax analysis phase the source program is analyzed to check whether if conforms to the source language's syntax, and to determine its phase structure. This phase is often separated into two phases:

- Lexical analysis: which produces a stream of tokens?
- Parser: which determines the phrase structure of the program based on the context-free grammar for the language?

PARSING:

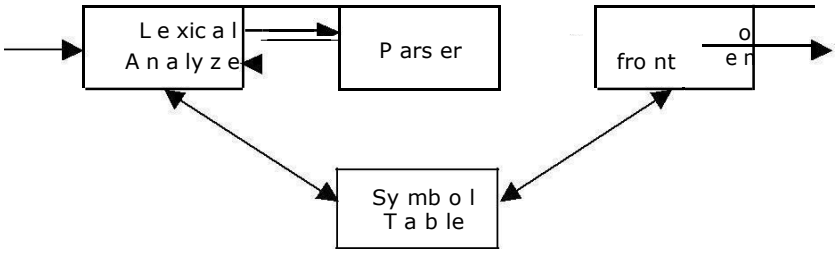
Parsing is the activity of checking whether a string of symbols is in the language of some grammar, where this string is usually the stream of tokens produced by the lexical analyzer. If the string is in the grammar, we want a parse tree, and if it is not, we hope for some kind of error message explaining why not.

There are two main kinds of parsers in use, named for the way they build the parse trees:

- Top-down: A top-down parser attempts to construct a tree from the root, applying productions forward to expand non-terminals into strings of symbols.
- Bottom-up: A Bottom-up parser builds the tree starting with the leaves, using productions in reverse to identify strings of symbols that can be grouped together.

In both cases the construction of derivation is directed by scanning the input sequence from left to right, one symbol at a time.

Parse Tree:



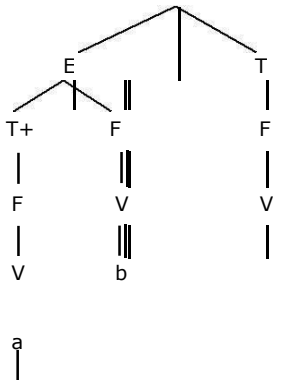
A parse tree is the graphical representation of the structure of a sentence according to its grammar.

Example:
Let the production P is:

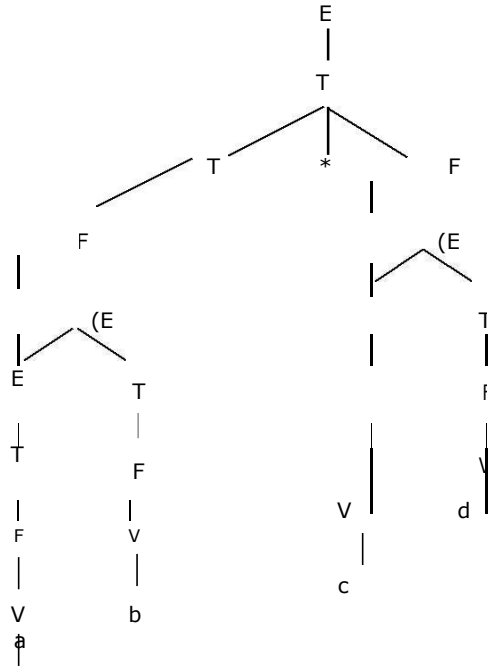
E→ T | E+T
T→ F | T*F
F→ V | (E)
V→ a | b | c |d

The parse tree may be viewed as a representation for a derivation that filters out the choice regarding the order of replacement.

Parse tree for a * b + c



Parse tree for (a * b) * (c + d)

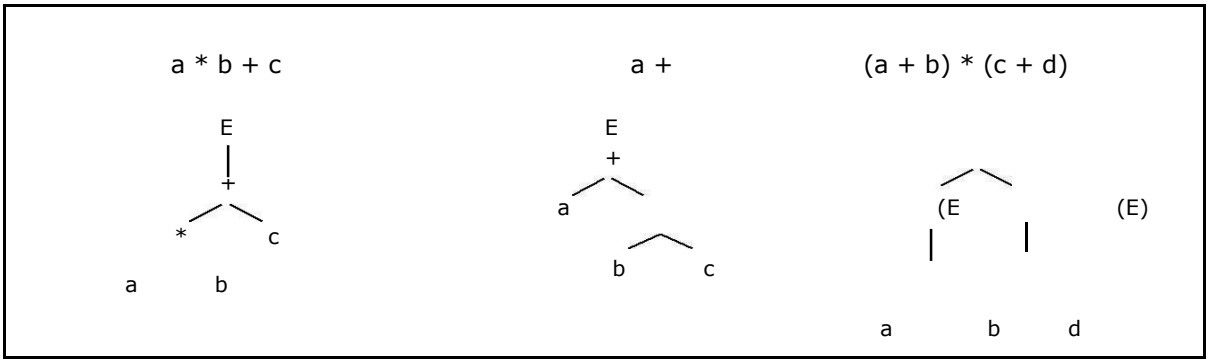


SYNTAX TREES:

Parse tree can be presented in a simplified form with only the relevant structure information by:

- Leaving out chains of derivations (whose sole purpose is to give operators difference precedence).
- Labeling the nodes with the operators in question rather than a non-terminal.

The simplified Parse tree is sometimes called as structural tree or syntax tree.



Synt a x T re e s

Syntax Error Handling:

If a compiler had to process only correct programs, its design & implementation would be greatly simplified. But programmers frequently write incorrect programs, and a good compiler should assist the programmer in identifying and locating errors.The programs contain errors at many different levels.
For example, errors can be:

1) Lexical – such as misspelling an identifier, keyword or operator
2) Syntactic – such as an arithmetic expression with un-balanced parentheses.
3) Semantic – such as an operator applied to an incompatible operand.
4) Logical – such as an infinitely recursive call.

Much of error detection and recovery in a compiler is centered around the syntax analysis phase. The goals of error handler in a parser are:

- It should report the presence of errors clearly and accurately.
- It should recover from each error quickly enough to be able to detect subsequent errors.
- It should not significantly slow down the processing of correct programs.

Ambiguity:

Several derivations will generate the same sentence, perhaps by applying the same productions in a different order. This alone is fine, but a problem arises if the same sentence has two distinct parse trees. A grammar is ambiguous if there is any sentence with more than one parse tree.

Any parses for an ambiguous grammar has to choose somehow which tree to return. There are a number of solutions to this; the parser could pick one arbitrarily, or we can provide some hints about which to choose. Best of all is to rewrite the grammar so that it is not ambiguous.

There is no general method for removing ambiguity. Ambiguity is acceptable in spoken languages. Ambiguous programming languages are useless unless the ambiguity can be resolved.

Fixing some simple ambiguities in a grammar:

	Ambiguous	language	unambiguous
(i)	$A \rightarrow B \mid AA$	Lists of one or more B's $C \rightarrow A \mid E$	$A \rightarrow BC$
(ii)	$A \rightarrow B \mid A;A$	Lists of one or more B's with punctuation $C \rightarrow ;A \mid E$	$A \rightarrow BC$
(iii)	$A \rightarrow B \mid AA \mid E$	lists of zero or more B's	$A \rightarrow BA \mid E$

Any sentence with more than two variables, such as (arg, arg, arg) will have multiple parse trees.

Left Recursion:

If there is any non terminal A, such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α, then

grammar is left recursive.
Algorithm for eliminating left Recursion:

Group all the A productions together like this: $A \Rightarrow A\ \alpha_1 \mid A\ \alpha_2 \mid - - - \mid A\ \alpha_m \mid \beta_1 \mid \beta_2 \mid - -- \mid \beta_n$ where
A is the left recursive non-terminal,
α is any string of terminals and
β is any string of terminals and non terminals that does not begin with A.

1. Replace the above A productions by the following: $A \Rightarrow \beta_1 A^I \mid \beta_2 A^I \mid - - - \mid \beta_n A^I$

$A^I \Rightarrow \alpha_1 A^I \mid \alpha_2 A^I \mid - - - \mid \alpha_m A^I \mid \in$ Where, A^I is a new non terminal.

Top down parsers cannot handle left recursive grammars.

If our expression grammar is left recursive:

- This can lead to non termination in a top-down parser.
- for a top-down parser, any recursion must be right recursion.
- we would like to convert the left recursion to right recursion.

Example 1:
Remove the left recursion from the production: $A \rightarrow A\ \alpha \mid \beta$

Left Recursive.
Eliminate

Applying the transformation yields:

$A \rightarrow \beta\ A^I$
$A^I \rightarrow \alpha\ A^I \mid \in$
↑
Remaining part after A.

Example 1:
Remove the left recursion from the productions:

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

Applying the transformation yields:

$E \rightarrow T\ E^I$ $\qquad$ $T \rightarrow F\ T^I$

$E^I \rightarrow T\ E^I \mid \in$ $\qquad$ $T^I \rightarrow * F\ T^I \mid \in$

Example 2:

Remove the left recursion from the productions:

$E \rightarrow E + T \mid E - T \mid T$

$T \rightarrow T * F \mid T/F \mid F$

Applying the transformation yields:

$E \rightarrow T\ E^I$ $\qquad$ $T \rightarrow F\ T^I$

$E \rightarrow + T\ E^I \mid - T\ E^I \mid \in$ $\qquad$ $T^I \rightarrow * F\ T^I \mid /F\ T^I \mid \in$

1. The non terminal S is left recursive because $S \rightarrow A\ a \rightarrow S\ d\ a$ But it is not immediate left recursive.
2. Substitute S-productions in $A \rightarrow S\ d$ to obtain:
   $A \rightarrow A\ c \mid A\ a\ d \mid b\ d \mid \in$
3. Eliminating the immediate left recursion:

Left Factoring:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

When it is not clear which of two alternative productions to use to expand a non-terminal A, we may be able to rewrite the productions to defer the decision until we have some enough of the input to make the right choice.

Algorithm:

For all A ∈ non-terminal, find the longest prefix α that occurs in two or more right-hand sides of A.

If $\alpha \neq \in$ then replace all of the A productions, $A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2 \mid - - - \mid \alpha\ \beta_n \mid r$ With

$A \rightarrow \alpha\ A^I \mid r$

$A^I \rightarrow \beta_1 \mid \beta_2 \mid - - - \mid \beta_n \mid \in$

Where, A^I is a new element of non-terminal. Repeat until no common prefixes remain.

It is easy to remove common prefixes by left factoring, creating new non-terminal.

For example consider:

$V \rightarrow \alpha\ \beta \mid \alpha\ r$ Change to:

$V \rightarrow \alpha\ V^I$ $V^I \rightarrow \beta \mid r$

Example 1:

Eliminate Left factoring in the grammar: $S \rightarrow V := int$

$V \rightarrow$ alpha '[' int ']' | alpha

Becomes:

$S \rightarrow V := int$

$V \rightarrow alpha\ V^I$

$V^I \rightarrow$ '[' int '] | $\in$

TOP DOWN PARSING:

Top down parsing is the construction of a Parse tree by starting at start symbol and "guessing" each derivation until we reach a string that matches input. That is, construct tree from root to leaves.The advantage of top down parsing in that a parser can directly be written as a program. Table-driven top-down parsers are of minor practical relevance. Since bottom-up parsers are more powerful than top-down parsers, bottom-up parsing is practically relevant.
For example, let us consider the grammar to see how top-down parser works:

S → if E then S else S | while E do S | print
E → true | False | id

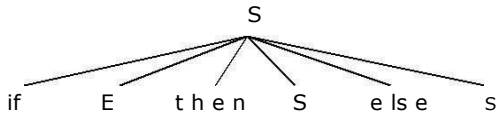
The input token string is: If id then while true do print else print.

1. Tree:

S

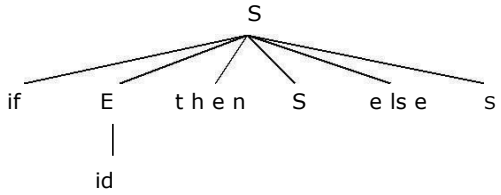
Input: if id then while true do print else print.
Action: Guess for S.

2. Tree:



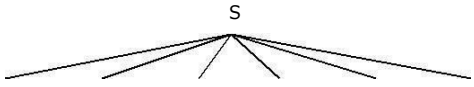
Input: if id then while true do print else print.
Action: if matches; guess for E.

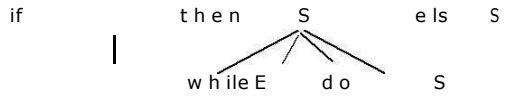
3. Tree:



Input: id then while true do print else print. Action: id matches; then matches; guess for S.

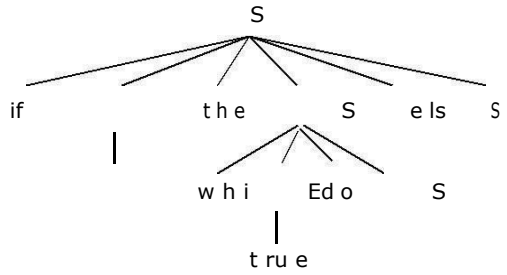
4. Tree:





Input: while true do print else print.
Action: while matches; guess for E.

5. Tree:



Input: true do print else print
Action:true matches; do matches; guess S.

Recursive Descent Parsing:

Top-down parsing can be viewed as an attempt to find a left most derivation for an input string. Equivalently, it can be viewd as a attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

The special case of recursive –decent parsing, called predictive parsing, where no backtracking is required. The general form of top-down parsing, called recursive descent, that may involve backtracking, that is, making repeated scans of the input.

Recursive descent or predictive parsing works only on grammars where the first terminal symbol of each sub expression provides enough information to choose which production to use.

Recursive descent parser is a top down parser involving backtracking. It makes a repeated scans of the input. Backtracking parsers are not seen frequently, as backtracking is very needed to parse programming language constructs.

Example: consider the grammar

S→cAd

A→ab|a

And the input string w=cad. To construct a parse tree for this string top-down, we initially create a tree

consisting of a single node labeled scan input pointer points to c, the first symbol of w. we then use the first production for S to expand tree and obtain the tree of Fig(a).

The left most leaf, labeled c, matches the first symbol of w, so we now advance the input pointer to a ,the second symbol of w, and consider the next leaf, labeled A. We can then expand A using the first alternative for A to obtain the tree in Fig (b). we now have a match for the second input symbol so we advance the input pointer to d, the third, input symbol, and compare d against the next leaf, labeled b. since b does not match the d ,we report failure and go back to A to see where there is any alternative for Ac that we have not tried but that might produce a match.

In going back to A, we must reset the input pointer to position2,we now try second alternative for A to obtain the tree of Fig(c).The leaf matches second symbol of w and the leaf d matches the third symbol .

The left recursive grammar can cause a recursive- descent parser, even one with backtracking, to go into an infinite loop.That is ,when we try to expand A, we may eventually find ourselves again trying to ecpand A without Having consumed any input.

Predictive Parsing:

Predictive parsing is top-down parsing without backtracking or look a head. For many languages, make perfect guesses (avoid backtracking) by using 1-symbol look-a-head. i.e., if:

$A \rightarrow \alpha_1 | \alpha_2 | - - - | \alpha_n$.

Choose correct α_i by looking at first symbol it derive. If $\in$ is an alternative, choose it last.

This approach is also called as predictive parsing. There must be at most one production in order to avoid backtracking. If there is no such production then no parse tree exists and an error is returned.

The crucial property is that, the grammar must not be left-recursive.

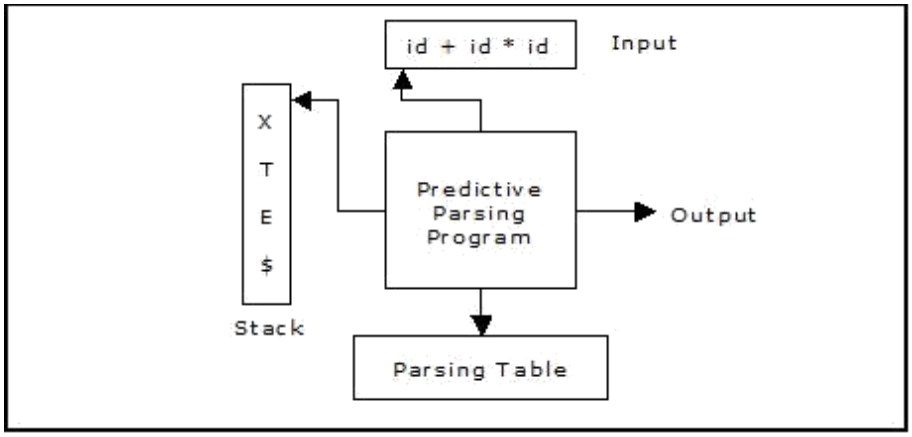
Predictive parsing works well on those fragments of programming languages in which keywords occurs frequently.

For example:

stmt $\rightarrow$ **if** exp **then** stmt **else** stmt | **while** expr **do** stmt
| **begin** stmt-list **end**.

then the keywords if, while and begin tell, which alternative is the only one that could possibly succeed if we are to find a statement.

The model of predictive parser is as follows:



A predictive parser has:

- Stack
- Input
- Parsing Table
- Output

The input buffer consists the string to be parsed, followed by $, a symbol used as a right end marker to indicate the end of the input string.

The stack consists of a sequence of grammar symbols with $ on the bottom, indicating the bottom of the stack. Initially the stack consists of the start symbol of the grammar on the top of $.

Recursive descent and LL parsers are often called predictive parsers, because they operate by predicting the next step in a derivation.

The algorithm for the Predictive Parser Program is as follows: Input: A string w and a parsing table M for grammar G

Output: if w is in L(g),a leftmost derivation of w; otherwise, an error indication.

Method: Initially, the parser has $S on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for the input is:

Set ip to point to the first symbol of w$; **repeat**

let x be the top stack symbol and a the symbol pointed to by ip; **if** X is a terminal or $ **then**

if X = a **then**

pop X from the stack and advance ip **else** error()

else /* X is a non-terminal */

if M[X, a] = X → $Y_1 Y_2 \ldots Y_k$ **thenbegin**

pop X from the stack;

push $Y_k, Y_{k-1}, \ldots Y_1$ onto the stack, with Y_1 on top; output the production X → $Y_1 Y_2 \ldots Y_k$

end

else error()

until X = $ /*stack is empty*/

FIRST and FOLLOW:

The construction of a predictive parser is aided by two functions with a grammar G. these functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible. Sets of tokens yielded by the **FOLLOW** function can also be used as synchronizing tokens during pannic-mode error recovery.

If α is any string of grammar symbols, let FIRST (α) be the set of terminals that begin the strings derived from α. If α=>€,then € is also in FIRST(α).

Define FOLLOW (A), for nonterminals A, to be the set of terminals a that can appear immediately to the right of A in some sentential form, that is, the set of terminals a such that there exist a derivation of the form S=>αAaβ for some α and β. If A can be the rightmost symbol in some sentential form, then $ is in FOLLOW(A).

Computation of FIRST ():

To compute **FIRST(X)** for all grammar symbols X, apply the following rules until no more terminals or € can be added to any FIRST set.

- If X is terminal, then FIRST(X) is {X}.
- If X→€ is production, then add € to FIRST(X).
- If X is nonterminal and X→Y_1 Y_2……Y_k is a production, then place a in FIRST(X) if for some i,a is in FIRST(Y_i),and € is in all of FIRST(Y_i),and € is in all of FIRST(Y_1),….. FIRST(Y_{i-1});that is Y_1 …… Y_{i-1}==>€.if € is in FIRST(Y_j), for all j=,2,3…….k, then add € to FIRST(X).for example, everything in FIRST(Y_1) is surely in FIRST(X).if Y_1 does not derive €,then we add nothing more to FIRST(X),but if Y_1=>€,then we add FIRST(Y_2) and so on.

FIRST (A) = FIRST (α_1) U FIRST (α_2) U - - - U FIRST (α_n) Where, A → α_1 | α_2 |------|α_n, are all the productions for A. FIRST (Aα) = if ∈ ∉ FIRST (A) then FIRST (A)
else (FIRST (A) - {∈}) U FIRST (α)

Computation of FOLLOW ():

To compute **FOLLOW (A)** for all nonterminals A, apply the following rules until nothing can be

added to any FOLLOW set.

- Place $ in FOLLOW(s), where S is the start symbol and $ is input right end marker .
- If there is a production A→αBβ,then everything in FIRST(β) except for € is placed in FOLLOW(B).
- If there is production A→αB, or a production A→αBβ where FIRST (β) contains € (i.e.,β→€),then everything in FOLLOW(A)is in FOLLOW(B).

Example:

Construct the FIRST and FOLLOW for the grammar:

$A \rightarrow BC \mid EFGH \mid H$
$B \rightarrow b$
$C \rightarrow c \mid \in$
$E \rightarrow e \mid \in$
$F \rightarrow CE$
$G \rightarrow g$
$H \rightarrow h \mid \in$

Solution:

1. Finding first () set:
   1. first (H) = first (h) ∪ first (ε) = {h, ε}
   2. first (G) = first (g) = {g}
   3. first (C) = first (c) ∪ first (ε) = c, ε}
   4. first (E) = first (e) ∪ first (ε) = {e, ε}
   5. first (F) = first (CE) = (first (c) - {ε}) ∪ first (E)
      = (c, ε} {ε}) ∪ {e, ε} = {c, e, ε}
   6. first (B) = first (b)={b}
   7. first (A) = first (BC) ∪ first (EFGH) ∪ first (H)
      = first (B) ∪ (first (E) – { ε}) ∪ first (FGH) ∪ {h, ε}
      = {b, h, ε} ∪ {e} ∪ (first (F) – {ε}) ∪ first (GH)
      = {b, e, h, ε} ∪ {C, e} ∪ first (G)
      = {b, c, e, h, ε} ∪ {g} = {b, c, e, g, h, ε}

2. Finding follow() sets:
   1. follow(A) = {$}

2. follow(B) = first(C) – {ε} ∪ follow(A) = {C, $}
3. follow(G) = first(H) – {ε} ∪ follow(A)
   ={h, ε} – {ε} ∪ {$} = {h, $}
4. follow(H) = follow(A) = {$}
5. follow(F) = first(GH) – {ε} = {g}
6. follow(E) = first(FGH) m- {ε} ∪ follow(F)
   = ((first(F) – {ε}) ∪ first(GH)) – {ε} ∪ follow(F)
   = {c, e} ∪ {g} ∪ {g} = {c, e, g}
7. follow(C) = follow(A) ∪ first (E) – {ε} ∪ follow (F)
   ={$} ∪ {e, ε} ∪ {g} = {e, g, $}

Example 1:

Construct a predictive parsing table for the given grammar or Check whether the given grammar is LL(1) or not.

$E \rightarrow E + T \mid T$

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

Step 1:

Suppose if the given grammar is left Recursive then convert the given grammar (and ∈) into non-left Recursive grammar (as it goes to infinite loop).

$E \rightarrow T\ E^I$

$E^I \rightarrow + T\ E^I \mid \in$ $T^I \rightarrow F\ T^I$

$T^I \rightarrow * F\ T^I \mid \in$ $F \rightarrow (E) \mid id$

Step 2:

Find the FIRST(X) and FOLLOW(X) for all the variables.

The variables are: {E, E^I, T, T^I, F}

Terminals are: {+, *, (,), id} and $

Computation of FIRST() sets:

FIRST (F) = FIRST ((E)) U FIRST (id) = {(, id}
FIRST (T^I) = FIRST (*FT^I) U FIRST (∈) = {*, ∈}
FIRST (T) = FIRST (FT^I) = FIRST (F) = {(, id}
FIRST (E^I) = FIRST (+TE^I) U FIRST (∈) = {+, ∈}
FIRST (E) = FIRST (TE^I) = FIRST (T) = {(, id}

Computation of FOLLOW () sets:

	Relevant production
FOLLOW (E) = {$} U FIRST ( )) = {$,)}	F → (E)

FOLLOW (E^I) = FOLLOW (E) = {$,)} $E \rightarrow TE^I$

FOLLOW (T) = (FIRST (E^I) - {∈}) U FOLLOW (E) U FOLLOW (E^I) $E \rightarrow TE^I$
= {+, $E^I \rightarrow +TE^I$

FOLLOW (T^I) = FOLLOW (T) = {+,), $} $T \rightarrow FT^I$

FOLLOW (F) = (FIRST (T^I) - {∈}) U FOLLOW (T) U FOLLOW (T^I) $T \rightarrow T^I$
= {*, +,

Step 3:
Construction of parsing table:

Termina / Variables	+		(	)	id	$
E			$E \rightarrow TE$		$E \rightarrow TE^I$	
E^I	$E^I \rightarrow +TE^I$			$E^I \rightarrow \in$		$E^I \rightarrow \in$
T			$T \rightarrow FT$		$T \rightarrow FT^I$	
T^I	$T^I \rightarrow \in$	$T^I \rightarrow *F$		$T^I \rightarrow \in$		$T^I \rightarrow \in$
F			$F \rightarrow (E)$		$F \rightarrow id$	

Table 3.1. Parsing Table

Fill the table with the production on the basis of the FIRST(α). If the input symbol is an ∈ in FIRST(α), then goto FOLLOW(α) and fill α → ∈, in all those input symbols.

Let us start with the non-terminal E, FIRST(E) = {(, id}. So, place the production E → TE^I at (and id.

For the non-terminal E^I, FIRST (E^I) = {+, ∈}.

So, place the production $E^I \rightarrow +TE^I$ at + and also as there is a ∈ in FIRST(E^I), see FOLLOW(E^I) = {$,)}. So write the production $E^I \rightarrow \in$ at the place $ and).

Similarly:

For the non-terminal T, FIRST(T) = {(, id}. So place the production T → FT^I at (and id.

For the non-terminal T^I, FIRST (T^I) = {*, ∈}

So place the production $T^I \rightarrow *FT^I$ at * and also as there is a ∈ in FIRST (T^I), see FOLLOW (T^I) = {+, $,)}, so write the production $T^I \rightarrow \in$ at +, $ and).

For the non-terminal F, FIRST (F) = {(, id}.
So place the production F → id at id location and F → (E) at (as it has two productions.

Finally, make all undefined entries as error.

As these were no multiple entries in the table, hence the given grammar is LL(1).

Step 4:

Moves made by predictive parser on the input id + id * id is:

STACK	INPUT	REMARKS
$ **E**	**id** + id * id $	E and id are not identical; so see E on id in parse table, the production is $E \rightarrow TE^I$; pop E, push E^I and T i.e., move in reverse order.
$ E^I **T**	**id** + id * id $	See T on id the production is $T \rightarrow F\,T^I$; Pop T, push T^I and F; Proceed until both are identical.
$ $E^I\,T^I$ **F**	**id** + id * id $	$F \rightarrow$ id
$ $E^I\,T^I$ **id**	**id** + id * id $	Identical; pop id and remove id from input symbol.
$ E^I **T^I**	**+** id *	See T^I on +; $T^I \rightarrow \in$ so, pop T^I
$ **E^I**	**+** id *	See E^I on +; $E^I \rightarrow +T\,E^I$; push E^I , + and T
$ E^I T **+**	**+** id *	Identical; pop + and remove + from input symbol.
$ E^I **T**	**id** *	
$ $E^I\,T^I$ **F**	**id** *	$T \rightarrow F\,T^I$
$ $E^I\,T^I$ **id**	**id** *	$F \rightarrow$ id
$ E^I **T^I**	*****	
$ $E^I\,T^I$ F *****	*****	$T^I \rightarrow$ * F T^I
$ $E^I\,T^I$ **F**		
$ $E^I\,T^I$ **id**		$F \rightarrow$ id
$ E^I **T^I**		$T^I \rightarrow \in$
$ **E^I**		$E^I \rightarrow \in$
$		Accept.

Table 3.2 Moves made by the parser on input id + id * id

Predictive parser accepts the given input string. We can notice that $ in input and stuck, i.e., both are empty, hence accepted.

LL (1) Grammar:

The first L stands for "Left-to-right scan of input". The second L stands for "Left-most derivation". The '1' stands for "1 token of look ahead".
No LL (1) grammar can be ambiguous or left recursive.

If there were no multiple entries in the Recursive decent parser table, the given grammar is LL (1).

If the grammar G is ambiguous, left recursive then the recursive decent table will have atleast one multiply defined entry.

The weakness of LL(1) (Top-down, predictive) parsing is that, must predict which production to use.

Error Recovery in Predictive Parser:

Error recovery is based on the idea of skipping symbols on the input until a token in a selected set of synchronizing tokens appear. Its effectiveness depends on the choice of synchronizing set. The Usage of FOLLOW and FIRST symbols as synchronizing tokens works reasonably well when expressions are parsed.

For the constructed table., fill with **synch** for rest of the input symbols of FOLLOW set and then fill the rest of the columns with **error** term.

Terminal / Variables	+	*	(	)	id	$
E	error	error	$E \rightarrow TE$	synch	$E \rightarrow TE^I$	synch
E_I	$E^I \rightarrow +TE^I$	error	error	$E^I \rightarrow \in$	error	$E^I \rightarrow \in$
T	synch	error	$T \rightarrow FT$	synch	$T \rightarrow FT^I$	synch
T_I	$T^I \rightarrow \in$	$T^I \rightarrow *F$	error	$T^I \rightarrow \in$	error	$T^I \rightarrow \in$
F	synch	synch	$F \rightarrow (E)$	synch	$F \rightarrow id$	synch

Table3.3 :Synchronizing tokens added to parsing table for table 3.1.

If the parser looks up entry in the table as synch, then the non terminal on top of the stack is popped in an attempt to resume parsing. If the token on top of the stack does not match the input symbol, then pop the token from the stack.

The moves of a parser and error recovery on the erroneous input) id*+id is as follows:

STACK	IN	REMARKS
$ **E**	) id * +	Error, skip)
$ **E**	**id** * +	
$ E^I **T**	**id** * +	
$ $E^I T^I$ **F**	**id** * +	
$ $E^I T^I$ **id**	**id** * +	
$ E^I $\mathbf{T^I}$	* +	
$ $E^I T^I$ F *	* +	

$ E^{I} T^{I} **F**	+	Error; F on + is synch; F has been popped.
$ E^{I} **T**^{I}	+	
$ **E**^{I}	+	
$ E^{I} T +	+	
$ E^{I} **T**		
$ E^{I} T^{I} **F**		
$ E^{I} T^{I} **id**		
$ E^{I} **T**^{I}		
$ **E**^{I}		
$		Accept.

Example 2:

Table 3.4. Parsing and error recovery moves made by predictive parser

Construct a predictive parsing table for the given grammar or Check whether the given grammar is LL(1) or not.

$S \rightarrow iEtSS^{I} \mid a$
$S^{I} \rightarrow eS \mid \in$
$E \rightarrow b$

Solution:

1. Computation of First () set:

   1. First (E) = first (b) = {b}
   2. First (S^{I}) = first (eS) $\cup$ first (ε) = {e, ε}
   3. first (S) = first ($iEtSS^{I}$) $\cup$ first (a) = {i, a}

2. Computation of follow() set:

   1. follow (S) = {$} $\cup$ first (S^{I}) – {ε} $\cup$ follow (S) $\cup$ follow (S^{I})
      = {$} $\cup$ {e} = {e, $}
   2. follow (S^{I}) = follow (S) = {e, $}
   3. follow (E) = first (tSS^{I}) = {t}

3. The parsing table for this grammar is:

	a	b	e	i	t	
S	$S \rightarrow a$			$S \rightarrow iEtSS^{I}$		
S^{I}			$S^{I} \rightarrow I$			$S^{I} \rightarrow$

∈

As the table multiply defined entry. The given grammar is not LL(1).

Example 3:

Construct the FIRST and FOLLOW and predictive parse table for the grammar:

S → AC$
C → c | ∈
A → aBCd | BQ | ∈
B → bB | d
Q → q

Solution:

1. Finding the first () sets: First (Q) = {q}

First (B) = {b, d}

First (C) = {c, ε}

First (A) = First (aBCd) ∪ First (BQ) ∪ First (ε)

= {a} ∪ First (B) ∪ First (d) ∪{ε}

= {a} ∪ First (bB) ∪ First (d) ∪ {ε}

= {a} ∪ {b} ∪ {d} ∪ {ε}

= {a, b, d, ε} First (S) = First (AC$)

= (First (A) – {ε}) ∪ (First (C) – {ε}) ∪ First (ε)

= ({a, b, d, ε} – {ε}) ∪ ({c, ε} – {ε}) ∪ {ε}

= {a, b, d, c, ε}

2. Finding Follow () sets: Follow (S) = {#}

Follow (A) = (First (C) – {ε}) ∪ First ($) = ({c, ε} – {ε}) ∪ {$} Follow (A) = {c, $}

Follow (B) = (First (C) – {ε}) ∪ First (d) ∪ First (Q)

= {c} ∪ {d} ∪ {q} = {c, d, q} Follow (C) = (First ($) ∪ First (d) = {d, $}

Follow (Q) = (First (A) = {c, $}

3. The parsing table for this grammar is:

	a	b	c	D	q	$	
S	S → AC$	S → AC $	S → AC $	S → AC $		S → AC $	
A	A → aBCd	A → BQ	A → ε	A → BQ		A → ε	
B		B → bB		B → d			
C			C → c	C → ε		C → ε	
Q					Q → q		

4. Moves made by predictive parser on the input abdcdc$ is:

Stack symbol	Input	Remarks
#S	abdcdc$#	S → AC$
#$CA	abdcdc$#	A → aBCd
#$CdCBa	abdcdc$#	Pop a
#$CdCB	bdcdc$#	B → bB
#$CdCBb	bdcdc$#	Pop b
#$CdCB	dcdc$#	B → d
#$CdCd	dcdc$#	Pop d
#$CdC	cdc$#	C → c
#$Cdc	cdc$#	Pop C
#$Cd	dc$#	Pop d
#$C	c$#	C → c
#$c	c$#	Pop c
#$	$#	Pop $
#	#	Accepted

BOTTOM UP PARSING

1. BOTTOM UP PARSING:

Bottom-up parser builds a derivation by working from the input sentence back towards the start symbol S. Right most derivation in reverse order is done in bottom-up parsing.

(The point of parsing is to construct a derivation. A derivation consists of a series of rewrite steps)

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 \Rightarrow - - - \Rightarrow r_{n-1} \Rightarrow r_n \Rightarrow$ sentence

Bottom-up

Assuming the production $A \rightarrow \beta$, to reduce r_i r_{i-1} match some RHS β against r_i then replace β with its corresponding LHS, A.

In terms of the parse tree, this is working from leaves to root.

Example – 1:

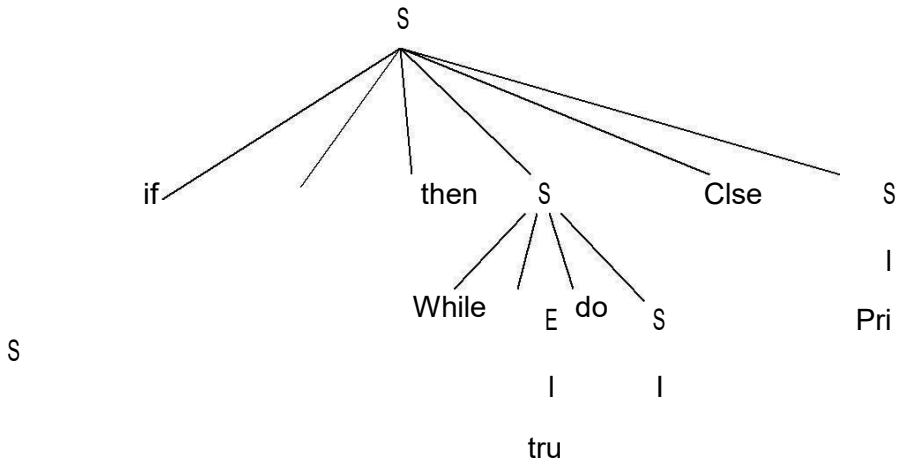
S→if E then S else S/while E do S/ print

E→ true/ False/id

Input: if id then while true do print else print.

Parse tree:

Basic idea: Given input string a, "reduce" it to the goal (start) symbol, by looking for substring that match production RHS.



$\Rightarrow_{lm}$ if E then S else S

$\Rightarrow_{lm}$ if id then S else S

$\Rightarrow_{lm}$ if id then while E do S else S

$\Rightarrow_{lm}$ if id then while true do S else S

$\Rightarrow_{lm}$ if id then while true do print else S

$\Rightarrow_{lm}$ if id then while true do print else print

$\Leftarrow_{rm}$ if E then while true do print else print

$\Leftarrow_{rm}$ if E then while E do print else print

$\Leftarrow_{rm}$ if E then while E do S else print

$\Leftarrow_{rm}$ if E then S else print

$\Leftarrow_{rm}$ if E then S else S

$\Leftarrow_{rm}$ S

Topdown Vs Bottom-up parsing:

Top-down	Bottom-up
1. Construct tree from root to leaves 2. "Guers" which RHS to substitute for nonterminal 3. Produces left-most derivation 4. Recursive descent, LL parsers 5. Recursive descent, LL parsers 6. Easy for humans	1. Construct tree from leaves to root 2. "Guers" which rule to "reduce" terminals 3. Produces reverse right-most derivation. 4. Shift-reduce, LR, LALR, etc. 5. "Harder" for humans.

→ Bottom-up can parse a larger set of languages than topdown.

→ Both work for most (but not all) features of most computer languages.

Example – 2:

S→aAcBe

A→Ab/b

B→d

llp: abbcde/

Right-most derivation

S→aAcBe

→ aAcde

→aAbcde

→ abbcde

Bottom-up approach

"Right sentential form"	Reduction
abbcde	
aAbcde	A→b
Aacde	A→Ab
AacBe	B→d
S	S→aAcBe

Steps correspond to a right-most derivation in reverse.

(must choose RHS wisely)

Example – 3:

S→aABe

A→Abc/b

B→d

1/p: abbcde

Right most derivation:

aABe

aAde Since ( ) B→d

aAbcde Since ( ) A→Abc

abbcde Since ( ) A→b

Parsing using Bottom-up approach:

Input	Production used
abbcde	
aAbcde	A→b
AAde	A→Abc
AABe	B→d

S parsing is completed as we got a start symbol

Hence the 1/p string is acceptable.

Example – 4

E→E+E

E→E*E

E→(E)

E→id

1/p: $id_1+id_2+id_3$

Right most derivation

E →E+E

→E+E*E

→E+E*id_3 →E+id_2*id_3

→id_1+id_2*id_3

Parsing using Bottom-up approach:

Go from left to right

id_1+id_2*id_3	
E+id_2*id_3	E→id
E+E*id_3	E→id
E*id_3	E→E+E
E*E	E→id
E	

= start symbol, Hence acceptable.

2. HANDLES:

Always making progress by replacing a substring with LHS of a matching production will not lead to the goal/start symbol.

For example:

abbcde

aAbcde A→b

aAAcde A→b

struck

Informally, A Handle of a string is a substring that matches the right side of a production, and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a right most derivation.

If the grammar is unambiguous, every right sentential form has exactly one handle.

More formally, A handle is a production A→β and a position in the current right-sentential form αβω such that:

S⇒αAω⇒α/βω

For example grammar, if current right-sentential form is

a/Abcde

Then the handle is A→Ab at the marked position. 'a' never contains non-terminals.

HANDLE PRUNING:

Keep removing handles, replacing them with corresponding LHS of production, until we reach S.

Example:

E→E+E/E*E/(E)/id

Right-sentential form	Handle	Reducing production
a+b*c	a	E→id
E+b*c	b	E→id

E+E*C	C	E→id
E+E*E	E*E	E→E*E
E+E	E+E	E→E+E
E		

The grammar is ambiguous, so there are actually two handles at next-to-last step. We can use parser-generators that compute the handles for us.

3. SHIFT- REDUCE PARSING:

Shift Reduce Parsing uses a stuck to hold grammar symbols and input buffer to hold string to be parsed, because handles always appear at the top of the stack i.e., there's no need to look deeper into the state.

A shift-reduce parser has just four actions:

1. Shift-next word is shifted onto the stack (input symbols) until a handle is formed.
2. Reduce – right end of handle is at top of stack, locate left end of handle within the stack. Pop handle off stack and push appropriate LHS.
3. Accept – stop parsing on successful completion of parse and report success.
4. Error – call an error reporting/recovery routine.

Possible Conflicts:

Ambiguous grammars lead to parsing conflicts.

1. **Shift-reduce:** Both a shift action and a reduce action are possible in the same state (should we shift or reduce)

Example: dangling-else problem

2. **Reduce-reduce:** Two or more distinct reduce actions are possible in the same state. (Which production should we reduce with 2).

Example:

Stmt →id (param) (a(i) is procedure call)

Param→ id

Expr → id (expr) /id (a(i) is array subscript)

Stack	input buffer	action
$…aa (i) ….$	Reduce by ?	

Should we reduce to param or to expr? Need to know the type of a: is it an array or a function. This information must flow from declaration of a to this use, typically via a symbol table.

Shift – reduce parsing example: (Stack implementation)

Grammar: E→E+E/E*E/(E)/id Input: $id_1+id_2+id_3$

One Scheme to implement a handle-pruning, bottom-up parser is called a shift-reduce parser. Shift reduce parsers use stack and an input buffer.

The sequence of steps is as follows:

1. initialize stack with $.
2. Repeat until the top of the stack is the goal symbol and the input token is "end of life". **a. Find the handle**

If we don't have a handle on top of stack, shift an input symbol onto the stack**.**

b. Prune the handle

if we have a handle (A→β) on the stack, reduce

(i) pop /β/ symbols off the stack (ii)push A onto the stack.

Stack	input	Action
$	$id_1+id_2*id_3$$	Shift
$ id_1	$+id_2*id_3$$	Reduce by E→id
$E	$+id_2*id_3$$	Shift
$E+	$id_2*id_3$$	Shift
$E+ id_2	$*id_3$$	Reduce by E→id

$E+E	*id3$	Shift
$E+E*	id3$	Shift
$E+E* id3	$	Reduce by E→id
$E+E*E	$	Reduce by E→E*E
$E+E	$	Reduce by E→E+E
$E	$	Accept

Example 2:

Goal → Expr

Expr → Expr + term | Expr – Term | Term

Term → Tem & Factor | Term | factor | Factor

Factor → number | id | (Expr)

The expression grammar : x – z * y

Stack	Input	Action
$	Id - num * id	Shift
$ id	- num * id	Reduce factor → id
$ Factor	- num * id	Reduce Term → Factor
$ Term	- num * id	Reduce Expr → Term
$ Expr	- num * id	Shift
$ Expr -	num * id	Shift
$ Expr – num	* id	Reduce Factor → num
$ Expr – Factor	* id	Reduce Term → Factor
$ Expr – Term	* id	Shift
$ Expr – Term *	id	Shift

$ Expr – Term * id		Reduce Factor → id
$ Expr – Term & Factor		Reduce Term → Term * Factor
$ Expr – Term		Reduce Expr → Expr – Term
$ Expr		Reduce Goal → Expr
$ Goal		Accept

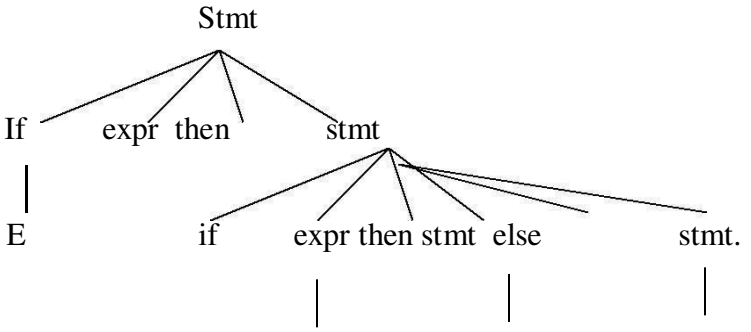
1. shift until the top of the stack is the right end of a handle
2. Find the left end of the handle & reduce.

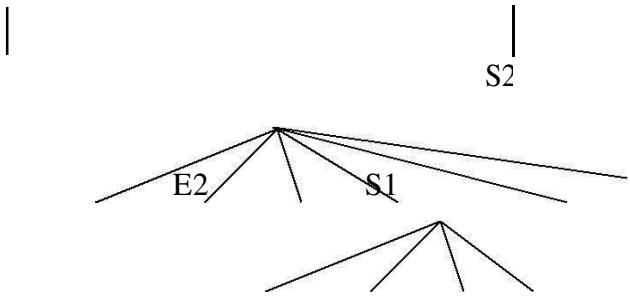
Procedure:

1. Shift until top of stack is the right end of a handle.
2. Find the left end of the handle and reduce.

* Dangling-else problem:

stmt→if expr then stmt/if expr then stmt/other then example string is: if E_1 then if E_2 then S_1 else S_2 has two parse trees (ambiguity) and so this grammar is not of LR(k) type.





3. OPERATOR – PRECEDENCE PARSING:

Precedence/ Operator grammar: The grammars having the property:

1. **No production right side is should contain ∈.**
2. **No production sight side should contain two adjacent non-terminals.**

Is called an **operator grammar.**

Operator – precedence parsing has three disjoint precedence relations, <.,=and .> between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

RELATION	MEANING
a<.b	'a' yields precedence to 'b'.
a=b	'a' has the same precedence 'b'
a.>b	'a' takes precedence over 'b'.

Operator precedence parsing has a number of disadvantages:

1. It is hard to handle tokens like the minus sign, which has two different precedences.
2. Only a small class of grammars can be parsed.
3. The relationship between a grammar for the language being parsed and the operator-precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language.

Disadvantages:

1. **L(G) ≠L(parser)**
2. **error detection**
3. **usage is limited**
4. **They are easy to analyse manually Example:**

Grammar: E→EAE|(E)|-E/id

A→+|-|*|/|↑

Input string: id+id*id

The operator – precedence relations are:

	Id	+	*	$
Id		.>	.>	.>
+	<.	.>	<.	.>
*	<.	.>	.>	.>
$	<.	<.	<.	

Solution: This is not operator grammar, so first reduce it to operator grammar form, by eliminating adjacent non-terminals.

Operator grammar is:

E→E+E|E-E|E*E|E/E|E↑E|(E)|-E|id

The input string with precedence relations interested is:

$<.id.> + <.id.> * <.id.> $

Scan the string the from left end until first .> is encounted.

$<.id.>+<.id.>*<.id.<$

This occurs between the first id and +.

Scan backwards (to the left) over any =’s until a <. Is encounted. We scan backwards to $.

$<.id.>+<.id.>*<.id.>$

↑ ↑

Everything to the left of the first .> and to the right of <. Is called handle. Here, the handle is the first id.

Then reduce id to E. At this point we have: E+id*id

By repeating the process and proceding in the same way: $+<.id.>*<.id.>$

substitute E→id,

After reducing the other id to E by the same process, we obtain the right-sentential form

E+E*E

Now, the 1/p string afte detecting the non-terminals sis:

⇒ $+*$

Inserting the precedence relations, we get: $<.+<.*.>$

↑ ↑

The left end of the handle lies between + and * and the right end between * and $. It indicates that, in the right sentential form E+E*E, the handle is E*E.

Reducing by E→E*E, we get:

E+E

Now the input string is: $<.+$

Again inserting the precedence relations, we get:

⇒$<.+.>$

↑ ↑

reducing by E→E+E, we get,

$ $

and finally we are left with:

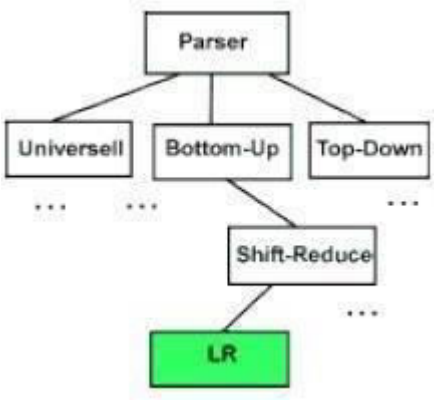
E

Hence accepted.

Input string	Precedence relations inserted	Action
id+id*id	$<.id.>+<.id.>*<.id.>$	
E+id*id	$+<.id.>*<.id.>$	E→id
E+E*id	$+*<.id.>$	E→id
E+E*E	$+*$	
E+E*E	$<.+<.*.>$	E→E*E
E+E	$<.+$	
E+E	$<.+.>$	E→E+E
E	$$	Accepted

5. LR PARSING INTRODUCTION:

The "L" is for left-to-right scanning of the input and the "R" is for constructing a rightmost derivation in reverse.



WHY LR PARSING:

1. LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.

2. The LR parsing method is the most general non-backtracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.

3. The class of grammars that can be parsed using LR methods is a proper subset of the class of grammars that can be parsed with predictive parsers.

4. An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

The disadvantage is that it takes too much work to constuct an LR parser by hand for a typical programming-language grammar. But there are lots of LR parser generators available to make this task easy.

LR PARSERS:

LR(k) parsers are most general non-backtracking shift-reduce parsers. Two cases of interest are k=0 and k=1. LR(1) is of practical relevance

'L' stands for "Left-to-right" scan of input.

'R' stands for "Rightmost derivation (in reverse)".

'K' stands for number of input symbols of look-a-head that are used in making parsing decisions.

When (K) is omitted, 'K' is assumed to be 1.

LR(1) parsers are table-driven, shift-reduce parsers that use a limited right context (1 token) for handle recognition.

LR(1) parsers recognize languages that have an LR(1) grammar. A grammar is LR(1) if, given a right-most derivation

$S \Rightarrow r_0 \Rightarrow r_1 \Rightarrow r_2 - - - r_{n-1} \Rightarrow r_n \Rightarrow$ sentence.

We can isolate the handle of each right-sentential form r_i and determine the production by which to reduce, by scanning r_i from left-to-right, going atmost 1 symbol beyond the right end of the handle of r_i.

Parser accepts input when stack contains only the start symbol and no remaining input symbol are left.

LR(0) item: (no lookahead)

Grammar rule combined with a dot that indicates a position in its RHS.

Ex– 1: $S^I \rightarrow .S\$$ $S \rightarrow .x$ $S \rightarrow .(L)$

Ex-2: $A \rightarrow XYZ$ generates 4LR(0) items –

$A \rightarrow .XYZ$

$A \rightarrow X.YZ$

$A \rightarrow XY.Z$

$A \rightarrow XYZ.$

The '.' Indicates how much of an item we have seen at a given state in the parse.

$A \rightarrow .XYZ$ indicates that the parser is looking for a string that can be derived from XYZ.

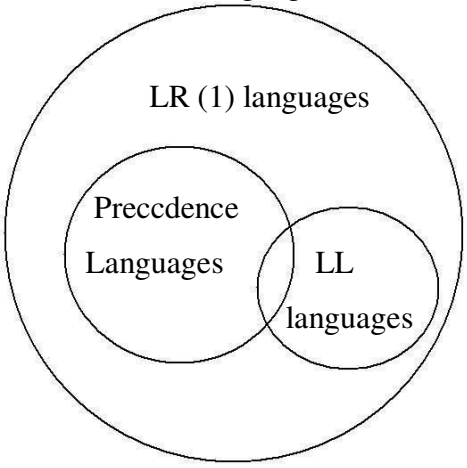
A→XY.Z indicates that the parser has seen a string derived from XY and is looking for one derivable from Z.

→ LR(0) items play a key role in the SLR(1) table construction algorithm.

→ LR(1) items play a key role in the LR(1) and LALR(1) table construction algorithms. LR parsers have more information available than LL parsers when choosing a production:

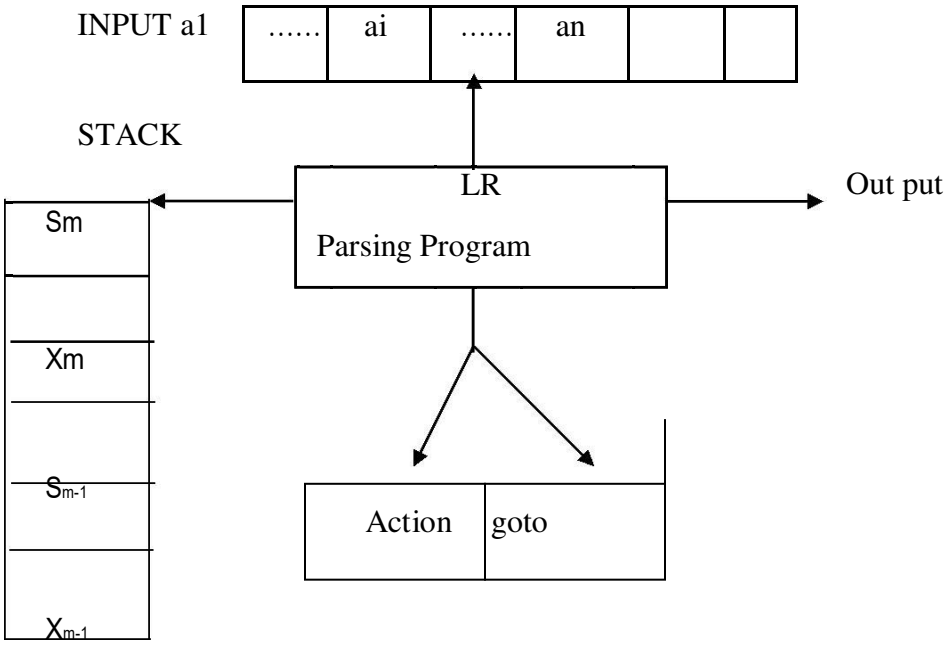
* **LR knows everything derived from RHS plus 'K' lookahead symbols.**
* **LL just knows 'K' lookahead symbols into what's derived from RHS.**

Deterministic context free languages:



LR PARSING ALGORITHM:

The schematic form of an LR parser is shown below:



It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts: action and goto.

The LR parser program determines Sm, the current state on the top of the stack, and a_i, the current input symbol. It then consults action [Sm, a_i], which can have one of four values:

1. **Shift S, where S is a state.**
2. **reduce by a grammar production A→β**
3. **accept and**
4. **error**

The function goes to takes a state and grammar symbol as arguments and produces a state. The goto function of a parsing table constructed from a grammar G using the SLR, canonical LR or LALR method is the transition function of DFA that recognizes the viable prefixes of G. (Viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shift-reduce parser, because they do not extend past the right-most handle)

5.6 AUGMENTED GRAMMAR:

If G is a grammar with start symbol S, then G^I, the augmented grammar for G with a new start symbol S^I and production $S^I \rightarrow S$.

The purpose of this new start stating production is to indicate to the parser when it should stop parsing and announce acceptance of the input i.e., acceptance occurs when and only when the parser is about to reduce by $S^I \rightarrow S$.

CONSTRUCTION OF SLR PARSING TABLE:

Example:

The given grammar is:

1. **E→E+T**
2. **E→ T**
3. **T →T*F**
4. **T→F**
5. **F→(E)**
6. **F→id Step I: The Augmented grammar is:**

$E^I \rightarrow E$

E→E+T

E→T

T→T*F

T→F

F→(E)

F→id

Step II: The collection of LR (0) items are:

I_0: $E^I \rightarrow .E$

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

Start with start symbol after since ( ) there is E, start writing all productions of E.

Start writing 'T' productions

Start writing F productions

Goto (I_0,E): States have successor states formed by advancing the marker over the symbol it preceeds. For state 1 there are successor states reached by advancing the masks over the

$E^I \rightarrow E.$ - symbols E,T,F,C or id. Consider, first, the

E→E.+T

Goto (I_0,T):

I_2: E→T. - reduced Item (RI)

T→T.*F

Goto (I_0,F):

I_2: E→T. - reduced item (RI)

T→T.*F

Goto (I_0,C):

I_4: F→(.E)

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

If '.' Precedes non-terminal start writing its corresponding production. Here first E then T after that F.

Start writing F productions.

Goto (I_0,id):

I_5: F →id. - reduced item.

E successor (I, state), it contains two items derived from state 1 and the closure operation adds no more (since neither marker precedes a non-terminal). The state I_2 is thus:

Goto (I_1,+):

I_6: E→E+.T start writing T productions

T→.T*F

T→.F start writing F productions

F→.(E)

F→.id

Goto (I_2,*):

I_7: T→T*.F start writing F productions

F→.(E)

F→.id

Goto (I_4,E):

I_8: F→(E.)

E→E.+T

Goto (I_4,T):

I_2: E→T. these are same as I_2.

T→T.*F

Goto (I_4,C):

I_4: F→(.E)

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

goto (I_4,id):

I_5: F→id. - reduced item

Goto (I_6,T):

I_9: E→E+T. - reduced item

T→T.*F

Goto (I_6,F):

I_3: T→F. - reduced item Goto (I_6,C):

I_4: F→(.E)

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

Goto (I_6,id):

I_5: F→id. reduced item.

Goto (I_7,F):

I_{10}: T→T*F reduced item

Goto (I_7,C):

I_4: F→(.E)

E→.E+T

E→.T

T→.T*F

T→.F

F→.(E)

F→.id

Goto (I_7,id):

I_5: F→id. - reduced item

Goto (I_8,)):

I_{11}: F→(E). reduced item

Goto (I_8,+):

I_{11}: F→(E). reduced item

Goto (I_8,+):

I_6: E→E+.T

T→.T*F

T→.F

F→.(E)

F→.id

Goto (I_9,+):

I_7: T→T*.f

F→.(E)

F→.id

Step IV: Construction of Parse table:

Construction must proceed according to the algorithm 4.8

S→shift items

R→reduce items

Initially E^I→E. is in I_1 so, I = 1.
Set action [I, $] to accept i.e., action [1, $] to Acc

Action									Goto
State	Id	+	*	(	)	$	E	T	F
I_0	S_5			S_4			1	2	3
1		S_6				Accept			
2		r_2	S_7		R_2	R_2			

3		R 4	R 4		R4	R4			
4	S5			S4					3
5		R 6	R 6		R6	R6			
6	S5			S4					3
7	S5			S4					10
8		S 6			S11				
9		R1	S7		r1	r1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

As there are no multiply defined entries, the grammar is SLR®.

STEP – III Finding FOLLOW ( ) set for all non-terminals.

Relevant production

FOLLOW (E) = {$} U FIRST (+T) U FIRST ( )) $E \rightarrow {}^{E}/_{B} + {}^{T}/_{B}$

= {+,), $} $F \rightarrow (E)$

$B\beta$

FOLLOW (T) = FOLLOW (E) U $E \rightarrow T$

FIRST (*F) U $T \rightarrow T*F$

FOLLOW (E) $E \rightarrow E+T$

B

= {+,*,),$}

FOLLOW (F) = FOLLOW (T)

= {*,*,),$}

Step – V:

1. Consider I_0:

1. The item F→.(E) gives rise to goto (I_0,C) = I_4, then action [0,C] = shift 4
2. The item F→.id gies rise goto (I_0,id) = I_4, then action [0,id] = shift 5

the other items in I_0 yield no actions. Goto (I_0,E) = I_1 then goto [0,E] = 1

Goto (I_0,T) = I_2 then goto [0,T] = 2

Goto (I_0,F) = I_3 then goto [0,F] = 3

2. Consider I_1:

    1. The item $E^I \rightarrow E.$ is the reduced item, so I = 1 This gives rise to action [1,$] to accept.
    2. The item E→E.+T gives rise to goto (I_1,+)=I_6, then action [1,+] = shift 6.
    3. Consider I_2:
        1. The item E→T. is the reduced item, so take FOLLOW (E),

FOLLOW (E) = {+,),$}

The first item +, makes action [Z,+] = reduce E→T. E→T is production rule no.2. So action [Z,+] = reduce 2.

The second item, makes action [Z,)] = reduce 2 The third item $, makes action [Z,$] = reduce 2

2. The item T→T.*F gives rise to goto [I_2,*]=I_7, then action [Z,*] = shift 7.
    4. Consider I_3:
        1. T→F. is the reduced item, so take FOLLOW (T).

FOLLOW (T) = {+,*,),$}

So, make action [3,+] = reduce 4

Action [3,*] = reduce 4

Action [3,)] = reduce 4

Action [3,$] = reduce 4

In forming item sets a closure operation must be performed to ensure that whenever the marker in an item of a set precedes a non-terminal, say E, then initial items must be included in the set for all productions with E on the left hand side.

The first item set is formed by taking initial item for the start state and then performing the closure operation, giving the item set;

We construct the action and goto as follows:

1. **If there is a transition from state I to state J under the terminal symbol K, then set action [I,k] to S_J.**
2. **If there is a transition under a non-terminal symbol a, say from state 'i' to state 'J', set goto [I,A] to S_J.**
3. **If state I contains a transition under $ set action [I,$] to accept.**
4. **If there is a reduce transition #p from state I, set action [I,k] to reduce #p for all terminals k belonging to FOLLOW (A) where A is the subject to production #P.**

   If any entry is multiply defined then the grammar is not SLR(1). Blank entries are represented by dash (-).

5. **Consider I_4 items:**

   The item F→id gives rise to goto [I_4,id] = I_5 so,

   Action (4,id) → shift 5

   The item F→.E action (4,c)→ shift 4

   The item goto (I_4,F) → I_3, so goto [4,F] = 3

   The item goto (I_4,T) → I_2, so goto [4,F] = 2

   The item goto (I_4,E) → I_8, so goto [4,F] = 8

6. **Consider I_5 items:**

   F→id. Is the reduced item, so take FOLLOW (F).

FOLLOW (F) = {+,*,),$}

F→id is rule no.6 so reduce 6

Action (5,+) = reduce 6

Action (5,*) = reduce 6

Action (5,)) = reduce 6

Action (5,)) = reduce 6

Action (5,$) = reduce 6

7. Consider I_6 items:

goto (I_6,T) = I_9, then goto [6,T] = 9 goto (I_6,F) = I_3, then goto [6,F] = 3 goto (I_6,C) = I_4, then goto [6,C] = 4 goto (I_6,id) = I_5, then goto [6,id] = 5

8. Consider I_7 items:

1. goto (I_7,F) = I_{10}, then goto [7,F] = 10
2. goto (I_7,C) = I_4, then action [7,C] = shift 4
3. goto (I_7,id) = I_5, then goto [7,id] = shift 5

9. Consider I_8 items:

1. goto (I_8,)) = I_{11}, then action [8,)] = shift 11
2. goto (I_8,+) = I_6, then action [8,+] = shift 6

10. Consider I_9 items:

1. E→E+T. is the reduced item, so take FOLLOW (E).

FOLLOW (E) = {+,),$}

E→E+T is the production no.1., so

Action [9,+] = reduce 1

Action [9,)] = reduce 1

Action [9,$] = reduce 1

2. goto [I_5,*] = I_7, then acgtion [9,*] = shift 7.

11. Consider I_{10} items:

1. T→T*F. is the reduced item, so take

FOLLOW (T) = {+,*,),$}

T→T*F is production no.3., so

Action [10,+] = reduce 3

Action [10,*] = reduce 3

Action [10,)] = reduce 3

Action [10,$] = reduce 3

12. Consider I_{11} items:

1. F→(E). is the reduced item, so take

FOLLOW (F) = {+,*,),$}

F→(E) is production no.5., so

Action [11,+] = reduce 5

Action [11,*] = reduce 5

Action [11,)] = reduce 5

Action [11,$] = reduce 5

VI MOVES OF LR PARSER ON id*id+id:

	STACK	INPUT	ACTION
1.	0	id*id+id$	shift by S5
2.	0id5	*id+id$	sec 5 on *
			reduce by F→id
			If A→β
			Pop 2*| β| symbols.
			=2*1=2 symbols.
			Pop 2 symbols off the stack
			State 0 is then exposed on F.

			Since goto of state 0 on F is 3, F and 3 are pushed onto the stack
3.	0F3	*id+id$	reduce by T →F pop 2 symbols push T. Since goto of state 0 on T is 2, T and 2, T and 2 are pushed onto the stack.
4.	0T2	*id+id$	shift by S7
5.	0T2*7	id+id$	shift by S5
6.	0T2*7id5	+id$	reduce by r6 i.e. F →id Pop 2 symbols, Append F, Secn 7 on F, it is 10
7.	0T2*7F10	+id$	reduce by r3, i.e., T →T*F Pop 6 symbols, push T Sec 0 on T, it is 2 Push 2 on stack.
8.	0T2	+id$	reduce by r2, i.e., E →T Pop two symbols, Push E See 0 on E. It 10 1 Push 1 on stack
9.	0E1	+id$	shift by S6.
10.	0E1+6	id$	shift by S5
11.	0E1+6id5	$	reduce by r6 i.e.,

		F →id Pop 2 symbols, push F, see 6 on F
0E1+6F3	$	It is 3, push 3 reduce by r4, i.e., T →F Pop2 symbols, Push T, see 6 on T
0E1+6T9	$	It is 9, push 9. reduce by r1, i.e., E →E+T Pop 6 symbols, push E See 0 on E, it is 1 Push 1.
0E1	$	Accept

Procedure for Step-V

The parsing algorithm used for all LR methods uses a stack that contains alternatively state numbers and symbols from the grammar and a list of input terminal symbols terminated by $. For example:

AAbBcCdDeEf/uvwxyz$

Where, a .....f are state numbers

A . . .. E are grammar symbols (either terminal or non-terminals) u .....z are the terminal symbols of the text still to be parsed. The parsing algorithm starts in state I_0 with the configuration –

0 / whole program upto $.

Repeatedly apply the following rules until either a syntactic error is found or the parse is complete.

(i) If action [f,4] = S_i then transform aAbBcCdDeEf / uvwxyz$

to aAbBcCdDeEfui / vwxyz$ This is called a SHIFT transition

(ii) If action [f,4] = #P and production # P is of length 3, say, then it will be of the form P → CDE where CDE exactly matches the top three symbols on the stack, and P is some non-terminal, then assuming goto [C,P] = g

aAbBcCdDEfui / vwxyz$ will transform to

aAbBcPg / vwxyz$

The symbols in the stack corresponding to the right hand side of the production have been replaced by the subject of the production and a new state chosen using the goto table. This is called a REDUCE transition.

(iii) If action [f,u] = accept. Parsing is completed

(iv) If action [f,u] = - then the text parsed is syntactically in-correct.

Canonical LR(O) collection for a grammar can be constructed by augmented grammar and two functions, closure and goto.

The closure operation:

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

i) **initially, every item in I is added to closure (I).**

CANONICAL LR PARSING*:*

Example:

S → CC

C →CC/d.

1. Number the grammar productions:

1. S →CC
2. C →CC
3. C →d

2. The Augmented grammar is:

S^I →S

S →CC

C →CC

C →d.

Constructing the sets of LR(1) items:

We begin with:

S^I →.S,$ begin with look-a-head (LAH) as $.

We match the item [S^I →.S,$] with the term [A →α.Bβ,a]
In the procedure closure, i.e.,

A = S^I

α = ∈

B = S

β = ∈ a = $

Function closure tells us to add [B→.r,b] for each production B→r and terminal b in FIRST (βa). Now β→r must be S→CC, and since β is ∈ and a is $, b may only be $. Thus,

S→.CC,$

We continue to compute the closure by adding all items [C→.r,b] for b in FIRST [C$] i.e., matching [S→.CC,$] against [A→α.Bβ,a] we have, A=S, α=∈, B=C and a=$. FIRST (C$) = FIRST © FIRST© = {c,d} We add items:

C→.cC,C

C→cC,d

C→.d,c

C→.d,d

None of the new items have a non-terminal immediately to the right of the dot, so we have completed our first set of LR(1) items. The initial I_0 items are:

I_0 : S^I→.S,$ S→.CC,$ C→.CC,c/d C→.d.c/d

Now we start computing goto (I_0,X) for various non-terminals i.e., Goto (I_0,S):

I_1 : S^I→S.,$ → reduced item.

Goto (I_0,C

I_2 : S→C.C, $

C→.cC,$

C→.d,$

Goto (I_0,C :

I_2 : C→c.C,c/d

C→.cC,c/d

C→.d,c/d

Goto (I_0,d)

I_4 C→d., c/d→ reduced item.

Goto (I_2,C) I_5

S→CC.,$ → reduced item.

Goto (I_2,C) I_6

	C→c.C,$	
	C→.cC,$	
	C→.d,$	
Goto (I_2,d)	I_7	
	C→d.,$	→ reduced item.
Goto (I_3,C)	I_8	
	C→cC.,c/d	→ reduced item.
Goto (I_3,C)	I_3	
	C→c.C, c/d	
	C→.cC,c/d	
	C→.d,c/d	
Goto (I_3,d)	I_4	
	C→d.,c/d.	→ reduced item.
Goto (I_6,C)	I_9	
	C→cC.,$	→ reduced item.
Goto (I_6,C)	I_6	
	C→c.C,$	
	C→,cC,$	
	C→.d,$	
Goto (I_6,d)	I_7	
	C→d.,$	→ reduced item.

All are completely reduced. So now we construct the canonical LR(1) parsing table –

Here there is no neet to find FOLLOW ( ) set, as we have already taken look-a-head for each set of productions while constructing the states.

Constructing LR(1) Parsing table:

	Action			goto	
State	C	D	$	S	C
I_0	S3	S4		1	2
1			Accept		

2	S6	S7			5
3	S3	S4			8
4	R3	R3			
5			R1		
6	S6	S7			9
7			R3		
8	R2	R2			
9			R2		

1. **Consider I_0 items:**

The item S→.S.$ gives rise to goto [I_0,S] = I_1 so goto [0,s] = 1.

The item S→.CC, $ gives rise to goto [I_0,C] = I_2 so goto [0,C] = 2.

The item C→.cC, c/d gives rise to goto [I_0,C] = I_3 so goto [0,C] = shift 3

The item C→.d, c/d gives rise to goto [I_0,d] = I_4 so goto [0,d] = shift 4

2. **Consider I_0 items:**

The item S^I→S.,$ is in I_1, then set action [1,$] = accept

3. **Consider I_2 items:**

The item S→C.C,$ gives rise to goto [I_2,C] = I_5. so goto [2,C] = 5

The item C→.cC, $ gives rise to goto [I_2,C] = I_6. so action [0,C] = shift The item C→.d,$ gives rise to goto [I_2,d] = I_7. so action [2,d] = shift 7

4. **Consider I_3 items:**

The item C→.cC, c/d gives rise to goto [I_3,C] = I_8. so goto [3,C] = 8

The item C→.cC, c/d gives rise to goto [I_3,C] = I_3. so action [3,C] = shift 3. The item C→.d, c/d gives rise to goto [I_3,d] = I_4. so action [3,d] = shift 4.

5. **Consider I_4 items:**

The item C→.d, c/d is the reduced item, it is in I_4 so set action [4,c/d] to reduce c→d. (production rule no.3)

6. **Consider I_5 items:**

The item S→CC.,$ is the reduced item, it is in I_5 so set action [5,$] to S→CC (production rule no.1)

7. **Consider I_6 items:**

The item C→c.C,$ gives rise to goto [I_6 ,C] = I_9. so goto [6,C] = 9

The item C→.cC,$ gives rise to goto [I_6 ,C] = I_6. so action [6,C] = shift 6

The item C→.d,$ gives rise to goto [I_6 ,d] = I_7. so action [6,d] = shift 7

8. **Consider I_7 items:**

   The item C→d., $ is the reduced item, it is in I_7.

   So set action [7,$] to reduce C→d (production no.3)

9. **Consider I_8 items:**

   The item C→CC.c/d in the reduced item, It is in Is, so set action[8,c/d] to reduce C→cd (production rale no .2)

10. **Consider I9 items:**

    The item C →cC, $ is the reduced item, It is in I9, so set action [9,$] to reduce C→cC (Production rale no.2)

    If the Parsing action table has no multiply –defined entries, then the given grammar is called as LR(1) grammar

LALR PARSING:

Example:

1. Construct C={I0,I1,… ..... ,In} The collection of sets of LR(1) items

2. For each core present among the set of LR (1) items, find all sets having that core, and replace there sets by their Union# (clus them into a single term)

I_0 →same as previous
I_1 →“ I_2 →“

I_{36} – Clubbing item I3 and I6 into one I36 item.

C →cC,c/d/$

C→cC,c/d/$

C→d,c/d/$

I_5 →some as previous

I_{47} →C→d,c/d/$

I_{89} →C→cC, c/d/$

LALR Parsing table construction:

State	Action			Goto	
	c	d			C
I_o	S_{36}	S_{47}			2
1			Accept		
2	S_{36}	S_{47}			5
36	S_{36}	S_{47}			89
47	r_3	r_3			
5			r_1		
89	r_2	r_2	r_2		