What is a Parse Tree?

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non terminals.

- ∑ Each interior node of a parse tree represents the application of a production.
- ∑ All the interior nodes are Non terminals and all the leaf nodes terminals.
- ∑ All the leaf nodes reading from the left to right will be the output of the parse tree.
- ∑ If a node n is labeled X and has children n1,n2,n3,…nk with labels X1,X2,…Xk respectively, then there must be a production A->X1X2…Xk in the grammar.

Example1:- Parse tree for the input string - **(id + id) using** the above Context free Grammar is

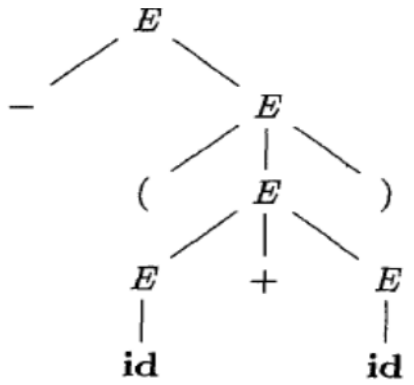


Figure 2.4 : Parse Tree for the input string - **(id + id)**

The Following figure shows step by step construction of parse tree using CFG for the parse tree for the input string - **(id + id).**

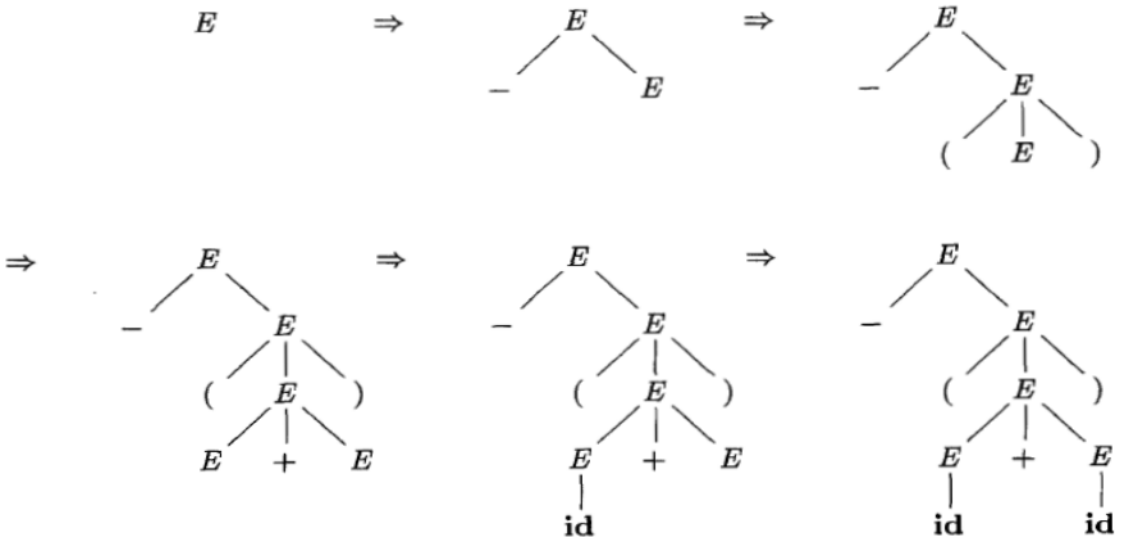


Figure 2.5 : Sequence outputs of the Parse Tree construction process for the input string **–(id+id)**

Example2:- Parse tree for the input string **id+id*id** using the above Context free Grammar is

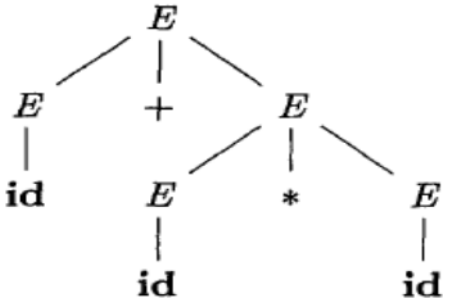


Figure 2.6: Parse tree for the input string id+ id*id

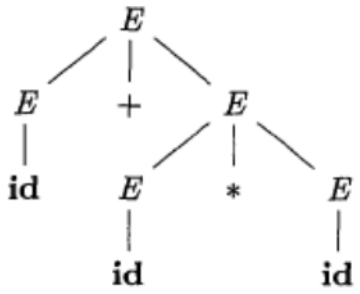
AMBIGUITY in CFGs:

Definition: A grammar that produces more than one parse tree for some sentence (input string) is said to be ambiguous.

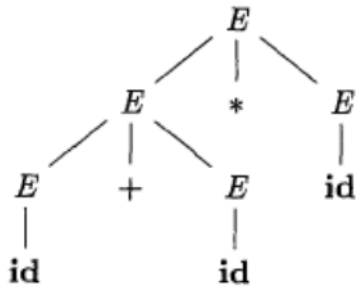
In other words, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

Or If the right hand production of the grammar is having two non terminals which are exactly same as left hand side production Non terminal then it is said to an ambiguous grammar.
Example : If the Grammar is **E-> E+E | E*E | -E| (E) | id** and the Input String is **id + id* id**

Two parse trees for given input string are



(a)



(b)

Two Left most Derivations for given input String are :

E => E +E

=> id + E

=> id + E * E

=> id + id * E

=> id + id * id

(a)

E => E * E

=> E+E*E

=> id + E * E

=> id+ id* E

=> id + id * id

(b)

The above Grammar is giving two parse trees or two derivations for the given input string so, it is an ambiguous Grammar

Note: LL (1) parser will not accept the ambiguous grammars or We cannot construct an LL(1) parser for the ambiguous grammars. Because such grammars may cause the Top Down parser to go into infinite loop or make it consume more time for parsing. If necessary we must remove all types of ambiguity from it and then construct.

ELIMINATING AMBIGUITY: Since Ambiguous grammars may cause the top down Parser go into infinite loop, consume more time during parsing.

Therefore, **s**ometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. The general form of ambiguous productions that cause ambiguity in grammars is

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

This can be written as (introduce one new non terminal in the place of second non terminal)

$$A \rightarrow \beta\, A'$$
$$A' \rightarrow \alpha\, A' \mid \varepsilon$$

Example : Let the grammar is **E⟶ E+E | E*E | -E| (E) | id .** It is shown that it is ambiguous that can be written as

E → E+E
E → E-E
E → E*E
E → -E
E → (E)
E → id

In the above grammar the 1st and 2nd productions are having ambiguity. So, they can be written as

E-> E+E | E*E this production again can be written as
E-> E+E | β , where β is E*E
The above production is same as the general form. so, that can be written as
E->E+T|T
T->β

The value of β is E*E so, above grammar can be written as

1) **E->E+T|T**
2) **T-> E*E** **The first production is free from ambiguity** and substitute E->T in the 2nd production then it can be written as
   T-> T*T | -E| (E) | **id** this production again can be written as
   T->T*T | β where β is -E| (E) | id, introduce new non terminal in the Right hand side production then it becomes
   T->T*F | F
   F-> -E | (E) | id now the entire grammar turned in to it equivalent unambiguous,

The Unambiguous grammar equivalent to the given ambiguous one is

1) **E → E + T | T**
2) **T → T * F | F**
3) **F → -E | (E) | id**

LEFT RECURSION:

Another feature of the CFGs which is not desirable to be used in top down parsers is left recursion. A grammar is left recursive if it has a non terminal A such that there is a derivation A=>Aα for some string α in (TUV)*. LL(1) or Top Down Parsers can not handle the Left Recursive grammars, so we need to remove the left recursion from the grammars before being used in Top Down Parsing.

The General form of Left Recursion is

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

The above left recursive production can be written as the non left recursive equivalent :

$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid €$$

Example : - Is the following grammar left recursive? If so, find a non left recursive grammar equivalent to it.

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

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

$$F \rightarrow -E \mid (E) \mid id$$

Yes ,the grammar is left recursive due to the first two productions which are satisfying the general form of Left recursion, so they can be rewritten after removing left recursion from $E \rightarrow E + T$, and $T \rightarrow T * F$ is

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

LEFT FACTORING:

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive or top-down parsing. A grammar in which more than one production has common prefix is to be rewritten by factoring out the prefixes.

For example, in the following grammar there are n A productions have the common prefix **α,** which should be removed or factored out without changing the language defined for A.

$$A \rightarrow \alpha A1 \mid \alpha A2 \mid \alpha A3 \mid \alpha A4 \mid \ldots \mid \alpha An$$

We can factor out the **α** from all n productions by adding a new A production $A \rightarrow \alpha A'$, and rewriting the A′ productions grammar as

$$A \rightarrow \alpha A'$$
$$A' \rightarrow A1 \mid A2 \mid A3 \mid A4 \ldots \mid An$$

FIRST and FOLLOW:

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. During top down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input (look a head) symbol.

Computation of FIRST:

FIRST function computes the set of terminal symbols with which the right hand side of the productions begin. To compute FIRST (A) **for all grammar symbols**, apply the following rules until no more terminals or **€** can be added to any FIRST set.

1. If A is a terminal, then FIRST {A} = {A}.
2. If A is a Non terminal and A->X1X2…Xi
   FIRST(A)=FIRST(X1) if X1is not null, if X1 is a non terminal and X1->**€**, add FIRST(X2) to FIRST(A), if X2-> **€** add FIRST(X3) to FIRST(A), … if Xi->**€** , i.e., all X_i's for i=1..i are null, add **€** FIRST(A).
3. If A **->€** is a production, then add € to FIRST (A).

Computation Of FOLLOW:

Follow (A) is nothing but the set of terminal symbols of the grammar that are immediately following the Non terminal A. If **a** is to the immediate right of non terminal A, then Follow(A)= {**a**}. To compute FOLLOW (A) for **all non terminals** A, apply the following rules until no more symbols can be added to any FOLLOW set.

1. Place $ in FOLLOW(S), where *S* is the start symbol, and $ is the input right end marker.
2. If there is a production A-> αBβ, then everything in FIRST (β) except € is in FOLLOW(B).
3. If there is a production A->αB or a production A-> αBβ with FIRST(β) contains **€**, then FOLLOW (B) = FOLLOW (A).

Example: - Compute the FIRST and FOLLOW values of the expression grammar

1. **E → TE′**
2. **E′→+TE′ | €**
3. **T → FT′**
4. **T′ →*FT′ | €**
5. **F → (E) | id**

Computing FIRST Values:

FIRST (E) = FIRST (T) = FIRST (F) = {(, id}
FIRST (E′) = {+, €}
FIRST (T′) = {*, €}

Computing FOLLOW Values:

FOLLOW (E) = { $,), } Because it is the start symbol of the grammar.

FOLLOW (E′) = {FOLLOW (E)} satisfying the 3rd rule of FOLLOW()

= { $,) }

FOLLOW (T) = { FIRST E′} It is Satisfying the 2nd rule.

U { FOLLOW(E′) }

= {+, FOLLOW (E′)}

= { +, $,) }

FOLLOW (T′) = { FOLLOW(T)} Satisfying the 3rd Rule

= { +, $,) }

FOLLOW (F) = { FIRST (T′) } It is Satisfying the 2nd rule.

U { FOLLOW(E′) }

= {*, FOLLOW (T)}

= { *, +, $,)}

NON TERMINAL	FIRST	FOLLOW
E	{ (, id }	{ $,) }
E′	{ +, € }	{ $,) }
T	{ (, id}	{ +, $,) }
T′	{*, €}	{ +, $,) }
F	{ (, id}	{ *, +, $,) }

Table 2.1: FIRST and FOLLOW values

Constructing Predictive Or LL (1) Parse Table:

It is the process of placing the all productions of the grammar in the parse table based on the FIRST and FOLLOW values of the Productions.

The rules to be followed to Construct the Parsing Table (M) are :

1. For Each production A-> α of the grammar, do the bellow steps.
2. For each terminal symbol _a‘ in FIRST (α), add the production A-> α to M [A, a].
3. i. If € is in FIRST (α) add production A->α to M [A, b], where b is all terminals in FOLLOW (A).
   ii. If € is in FIRST(α) and $ is in FOLLOW (A) then add production A->α to M [A, $].
4. Mark other entries in the parsing table as error .

NON-TERMINALS	INPUT SYMBOLS					
	+	*	(	)	id	$

E			E TE′		E id	
E′	E′ +TE′			E′ €		E′ €
T			T FT′		T FT′	
T′	T′ €	T′ *FT′		T′ €		T′ €
F			F (E)		F id	

Table 2.2: LL (1) Parsing Table for the Expressions Grammar

Note: if there are no multiple entries in the table for single a terminal then grammar is accepted by LL(1) Parser.

LL (1) Parsing Algorithm:

The parser acts on basis on the basis of two symbols

i. A, the symbol on the top of the stack

ii. a, the current input symbol

There are three conditions for A and _a', that are used fro the parsing program.

1. If A=a=$ then parsing is Successful.
2. If A=a≠$ then parser pops off the stack and advances the current input pointer to the next.
3. If A is a Non terminal the parser consults the entry M [A, a] in the parsing table. If M[A, a] is a Production A-> $X_1X_2..X_n$, then the program replaces the A on the top of the Stack by $X_1X_2..X_n$ in such a way that X_1 comes on the top.

STRING ACCEPTANCE BY PARSER:

If the input string for the parser is **id + id * id,** the below table shows how the parser accept the string with the help of Stack.

Stack	**Input**	**Action**	**Comments**
$E	id+ id* id $	E TE`	E on top of the stack is replaced by TE`
$E`T	id+ id*id $	T FT`	T on top of the stack is replaced by FT`
$E`T`F	id+ id*id $	F id	F on top of the stack is replaced by id
$E`T`id	id+ id*id $	pop and remove id	Condition 2 is satisfied
$E`T`	+id*id$	T` €	T` on top of the stack is replaced by €
$E`	+id*id$	E` +TE`	E` on top of the stack is replaced by +TE`
$E`T+	+id*id$	Pop and remove +	Condition 2 is satisfied
$E`T	id*id$	T FT`	T on top of the stack is replaced by FT`
$E`T`F	id*id$	F id	F on top of the stack is replaced by id
$E`T`id	id * id$	pop and remove id	Condition 2 is satisfied

$E`T`	*id$	T` *FT`	T` on top of the stack is replaced by *FT`
$E`T`F*	*id$	pop and remove *	Condition 2 is satisfied
$E`T`F	id$	F id	F on top of the stack is replaced by id
$E`T`id	id$	Pop and remove id	Condition 2 is satisfied
$E`T`	$	T` €	T` on top of the stack is replaced by €
$E`	$	E` €	E` on top of the stack is replaced by €
$	$	Parsing is successful	Condition 1 satisfied

Table2.3 : Sequence of steps taken by parser in parsing the input **token stream id+ id* id**

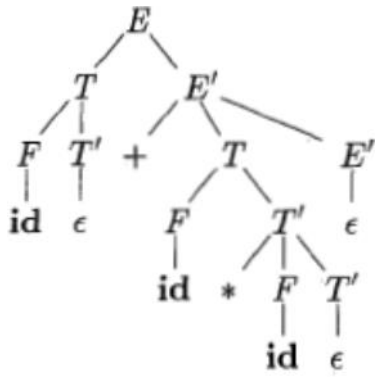


Figure 2.7: Parse tree for the input id + id* id

ERROR HANDLING (RECOVERY) IN PREDICTIVE PARSING:

In table driven predictive parsing, it is clear as to which terminal and Non terminals the parser expects from the rest of input. An error can be detected in the following situations:

1. When the terminal on top of the stack does not match the current input symbol.
2. when Non terminal A is on top of the stack, a is the current input symbol, and M[A, a] is empty or error

The parser recovers from the error and continues its process. The following error recovery schemes are use in predictive parsing:

Panic mode Error Recovery :

It is based on the idea that when an error is detected, the parser will skips the remaining input until a synchronizing token is en countered in the input. Some examples are listed below:

1. For a Non Terminal A, place all symbols in FOLLOW (A) are adde into the synchronizing set of non terminal A. For Example, consider the assignment statement ―c=;‖ Here, the expression on the right hand side is missing. So the Follow of this is considered. It is ―;‖ and is taken as synchronizing token. On encountering it, parser emits an error message ―Missing Expression‖.
2. For a Non Terminal A, place all symbols in FIRST (A) are adde into the synchronizing set of non terminal A. For Example, consider the assignment statement ―22c= a + b;‖ Here, FIRST (expr) is 22. It is ―;‖ and is taken as synchronizing token and then the reports the error as ―extraneous token‖.

Phrase Level Recovery :

It can be implemented in the predictive parsing by filling up the blank entries in the predictive parsing table with pointers to error Handling routines. These routines can insert, modify or delete symbols in the input.

RECURSIVE DESCENT PARSING :

A recursive-descent parsing program consists of a set of recursive procedures, one for each non terminal. Each procedure is responsible for parsing the constructs defined by its non terminal, Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string.

If the given grammar is

E → TE′
E′ → +TE′ | €
T → FT′
T′ → *FT′ | €
F → (E) | id

Reccursive procedures for the recursive descent parser for the given grammar are given below.

```
procedure E( )
{
        T( );
         E′( );
}
procedure T ( )
{
        F(  );
        T′( );
}
Procedure E′( )
{
        if input = _+‘
        {
                advance( );
                T ( );
                E′( );
                return true;
        }
        else error;
}
procedure T′( )
{
        if input = _*‘
        {
                advance( );
                F ( );
```

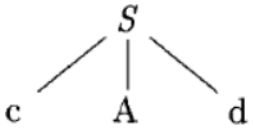
```
                T′( );
        return true;
        }
        else return error;
}
procedure F( )
{
        if input = ‗(‗
        {
                advance( );
                E ( );
                if input = ‗)‘
                advance( );
                return true;
        }
        else if input = ―id‖
        {

                advance( );
                return true;
        }
        else return error;
}
advance()
{
        input = next token;
}
```

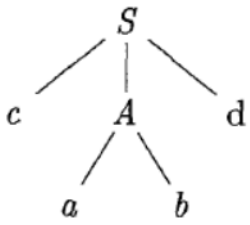
BACK TRACKING: This parsing method uses the technique called Brute Force method during the parse tree construction process. This allows the process to go back (back track) and redo the steps by undoing the work done so far in the point of processing.

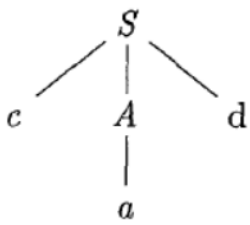
Brute force method: It is a Top down Parsing technique, occurs when there is more than one alternative in the productions to be tried while parsing the input string. It selects alternatives in the order they appear and when it realizes that something gone wrong it tries with next alternative.
For example, consider the grammar bellow.

S → cAd
A → ab | a

To generate the input string ―cad‖, initially the first parse tree given below is generated. As the string generated is not ―cad‖, input pointer is back tracked to position ―A‖, to examine the next alternate of ―A‖. Now a match to the input string occurs as shown in the 2nd parse trees given below.







(1) **(2)**

IMPORTANT AND EXPECTED QUESTIONS

1. Explain the components of working of a Predictive Parser with an example?
2. What do the FIRST and FOLLOW values represent? Give the algorithm for computing FIRST n FOLLOW of grammar symbols with an example?
3. Construct the LL (1) Parsing table for the following grammar?
   E → E+T|T

   T → T*F

   F → (E) | id
4. For the above grammar construct, and explain the Recursive Descent Parser?
5. What happens if multiple entries occurring in your LL (1) Parsing table? Justify your answer? How does the Parser

QUESTIONS

1. Eliminate the Left recursion from the below grammar?
   A-> Aab | AcB|b
   B-> Ba | d
2. Explain the procedure to remove the ambiguity from the given grammar with your own example?
3. Write the grammar for the if-else statement in the C programming and check for the left factoring?
4. Will the Predictive parser accept the ambiguous Grammar justify your answer?
5. Is the grammar G = { S->L=R, S->R, R->L, L->*R | **id** } an LL(1) grammar?

BOTTOM-UP PARSING

Bottom-up parsing corresponds to the construction of a parse tree for an input string beginning at the leaves (the bottom nodes) and working up towards the root (the top node). It involves ―reducing an input string _w' to the Start Symbol of the grammar. in each reduction step, a perticular substring matching the right side of the production is replaced by symbol on the left of that production and it is the Right most derivation. For example consider the following Grammar:

E → E+T|T

T → T*F

F → (E)|id

Bottom up parsing of the input string **"id * id "**is as follows:

INPUT STRING	**SUB STRING**	**REDUCING PRODUCTION**
id*id	Id	F->id
F*id	T	F->T
T*id	Id	F->id
T*F	*	T->T*F
T	T*F	E->T
E		Start symbol. Hence, the input String is accepted

Parse Tree representation is as follows:

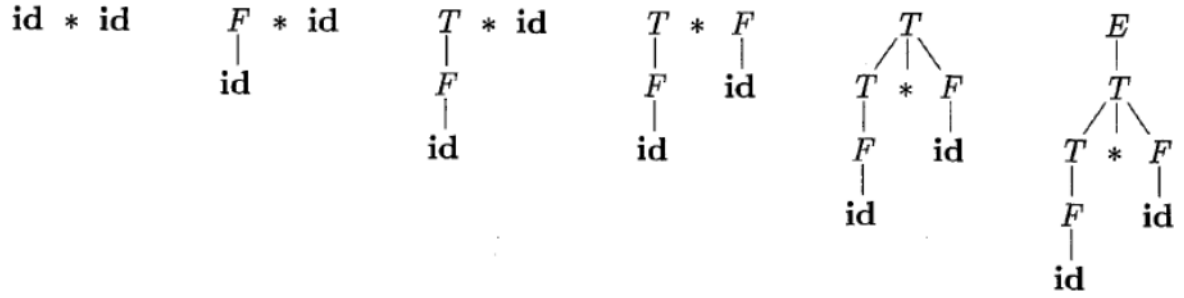


Figure 3.1 : A Bottom-up Parse tree for the input String "id*id"

Bottom up parsing is classified in to 1. Shift-Reduce Parsing, 2. Operator Precedence parsing , and 3. [Table Driven] L R Parsing

i. SLR(1)
ii. CALR (1)
iii. LALR(1)

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 the rest of the string to be parsed, We use $ to mark the bottom of the stack and also the right end of the input. And it makes use of the process of shift and reduce actions to accept the input string. Here, the parse tree is Constructed bottom up from the leaf nodes towards the root node.

When we are parsing the given input string, if the match occurs the parser takes the reduce action otherwise it will go for shift action. And it can accept ambiguous grammars also.

For example, consider the below grammar to accept the input string —id * id—, using S-R parser

E → E+T|T
T → T*F | F
F → (E)|id

Actions of the Shift-reduce parser using Stack implementation

STACK	**INPUT**	**ACTION**
$	Id*id$	Shift
$id	*id$	Reduce with F→ d
$F	*id$	Reduce with T→F
$T	*id$	Shift
$T*	id$	Shift
$T*id	$	Reduce with F→id
$T*F	$	Reduce with T→ T*F
$T	$	Reduce with E→ T
$E	$	Accept

Consider the following grammar:

S → aAcBe
A→ Ab|b
B→ d

Let the input string is ―abbcde‖. The series of shift and reductions to the start symbol are as follows.

abbcde ⇨ aAbcde ⇨ aAcde ⇨ aAcBe ⇨ S

Note: in the above example there are two actions possible in the second Step, these are as follows :

1. Shift action going to 3rd Step
2. Reduce action, that is A->b

If the parser is taking the 1st action then it can successfully accepts the given input string, if it is going for second action then it can't accept given input string. This is called shift reduce conflict. Where, S-R parser is not able take proper decision, so it not recommended for parsing.

OPERATOR PRECEDENCE PARSING:

Operator precedence grammar is kinds of shift reduce parsing method that can be applied to a small class of operator grammars. And it can process ambiguous grammars also.

- An operator grammar has two important characteristics:
  1. There are no € productions.
  2. No production would have two adjacent non terminals.
- The operator grammar to accept expressions is give below:

  E ➡ E+E / E → E-E / E→ E*E / E→ E/E / E→ E^E / E→ -E / E→ (E) / E→ id

Two main Challenges in the operator precedence parsing are:

1. Identification of Correct handles in the reduction step, such that the given input should be reduced to starting symbol of the grammar.
2. Identification of which production to use for reducing in the reduction steps, such that we should correctly reduce the given input to the starting symbol of the grammar.

Operator precedence parser consists of:

1. An input buffer that contains string to be parsed followed by a$, a symbol used to indicate the ending of input.
2. A stack containing a sequence of grammar symbols with a $ at the bottom of the stack.
3. An operator precedence relation table O, containing the precedence ralations between the pair of terminal. There are three kinds of precedence relations will exist between the pair of terminal pair _a' and _b' as follows:
4. The relation a<•b implies that he terminal _a' has lower precedence than terminal _b'.
5. The relation a•>b implies that he terminal _a' has higher precedence than terminal _b'.
6. The relation a=•b implies that he terminal _a' has lower precedence than terminal _b'.

7. An operator precedence parsing program takes an input string and determines whether it conforms to the grammar specifications. It uses an operator precedence parse table and stack to arrive at the decision.

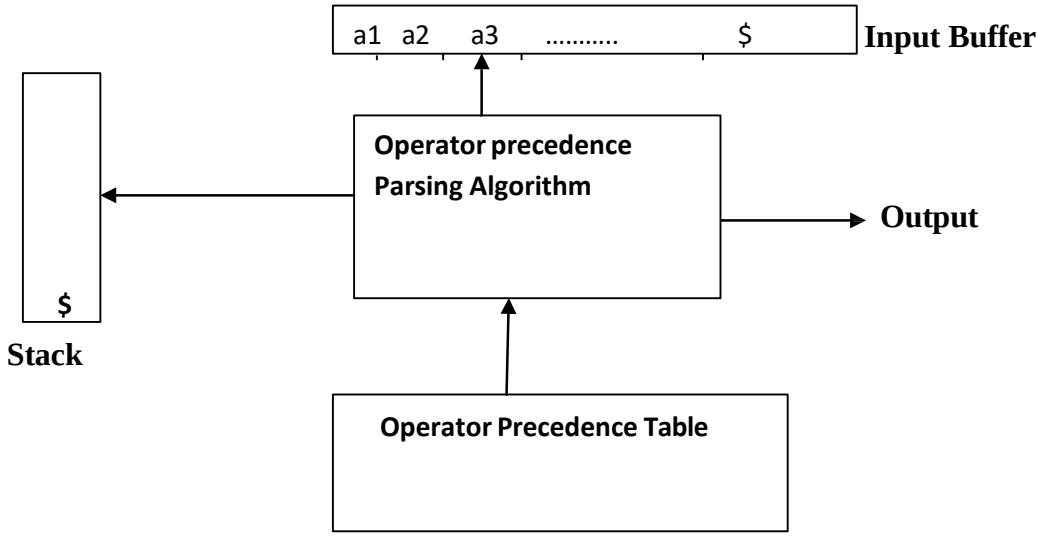


Figure3.2: Components of operator precedence parser

Example, If the grammar is

E → E+E
E → E-E
E → E*E
E → E/E
E → E^E
E → -E
E → (E)
E → id , Construct operator precedence table and accept input string **" id+id*id"**

The precedence relations between the operators are
(id) > (^) > (* /) > (+ -) > $, „^" operator is Right Associative and reaming all operators are Left Associative

	+	-	*	/	^	id	(	)	$
+	•>	•>	<•	<•	<•	<•	<•	•>	•>
-	•>	•>	<•	<•	<•	<•	<•	•>	•>
*	•>	•>	•>	•>	<•	<•	<•	•>	•>
/	•>	•>	•>	•>	<•	<•	<•	•>	•>
^	•>	•>	•>	•>	<•	<•	<•	•>	•>
Id	•>	•>	•>	•>	•>	**Err**	**Err**	•>	•>
(	<•	<•	<•	<•	<•	<•	<•	=	**Err**
)	•>	•>	•>	•>	•>	**Err**	**Err**	•>	•>
$	<•	<•	<•	<•	<•	<•	<•	**Err**	**Err**

The intention of the precedence relations is to delimit the handle of the given input String with <• marking the left end of the Handle and •> marking the right end of the handle.

Parsing Action:

To locate the handle following steps are followed:

1. Add $ symbol at the both ends of the given input string.
2. Scan the input string from left to right until the right most •> is encountered.
3. Scan towards left over all the equal precedence's until the first <• precedence is encountered.
4. Every thing between <• and •> is a handle.
5. $ on S means parsing is success.

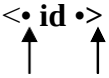
Example, Explain the parsing Actions of the OPParser for the input string is **"id*id"** and the grammar is:

E → E+E

E → E*E

E → id

1. $ <• id •> *<• id•> $



The first handle is _id' and match for the _id _in the grammar is E → id . So, id is replaced with the Non terminal E. the given input string can be written as

2. $ <• E •> *<• id•> $

The parser will not consider the Non terminal as an input. So, they are not considered in the input string. So , the string becomes

3. $ <• *<• id•> $

The next handle is _id' and match for the _id _in the grammar is E → id . So, id is replaced with the Non terminal E. the given input string can be written as

4. **$ <• *<• E•> $**

The parser will not consider the Non terminal as an input. So, they are not considered in the input string. So, the string becomes

5. **$ <• * •> $**

The next handle is _*' and match for the _ _in the grammar is E → E * E . So, id is replaced with the Non terminal E. the given input string can be written as

6. **$ E $**

The parser will not consider the Non terminal as an input. So, they are not considered in the input string. So, the string becomes

7. **$ $**

$ On $ means parsing successful.

Operator Parsing Algorithm:

The operator precedence Parser parsing program determines the action of the parser depending on

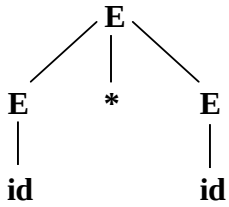
1. _a‘ is top most symbol on the Stack
2. _b‘ is the current input symbol

There are 3 conditions for _a‘ and _b‘ that are important for the parsing program

1. a=b=$, the parsing is successful
2. a <• b or a = b, the parser shifts the input symbol on to the stack and advances the input pointer to the next input symbol.
3. a •> b, parser performs the reduce action. The parser pops out elements one by one from the stack until we find the current top of the stack element has lower precedence than the most recently popped out terminal.

Example, the sequence of actions taken by the parser using the stack for the input string —id * id — and corresponding Parse Tree are as under.

STACK	INPUT	OPERATIONS
$	id * id $	$ <• id, shift _id‘ in to stack
$ id	*id $	id •> *, reduce _id‘ using E-> id
$E	*id $	$ <• *, shift _*‘ in to stack
$E*	id$	* <• id , shift _id‘ in to Stack
$E*id	$	id •> $, reduce _id‘ using E->id
$E*E	$	*•> $, reduce _*‘ using E->E*E
$E	$	$=$=$, so parsing is successful



Advantages and Disadvantages of Operator Precedence Parsing:

The following are the advantages of operator precedence parsing

1. It is simple and easy to implement parsing technique.
2. The operator precedence parser can be constructed by hand after understanding the grammar. It is simple to debug.

The following are the disadvantages of operator precedence parsing:

1. It is difficult to handle the operator like _-_which can be either unary or binary and hence different precedence‘s and associativities.
2. It can parse only a small class of grammar.

3. New addition or deletion of the rules requires the parser to be re written.
4. Too many error entries in the parsing tables.

LR Parsing:

Most prevalent type of bottom up parsing is LR (k) parsing. Where, L is left to right scan of the given input string, R is Right Most derivation in reverse and K is no of input symbols as the Look ahead.

- ∑ It is the most general non back tracking shift reduce parsing method
- ∑ The class of grammars that can be parsed using the LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
- ∑ 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.

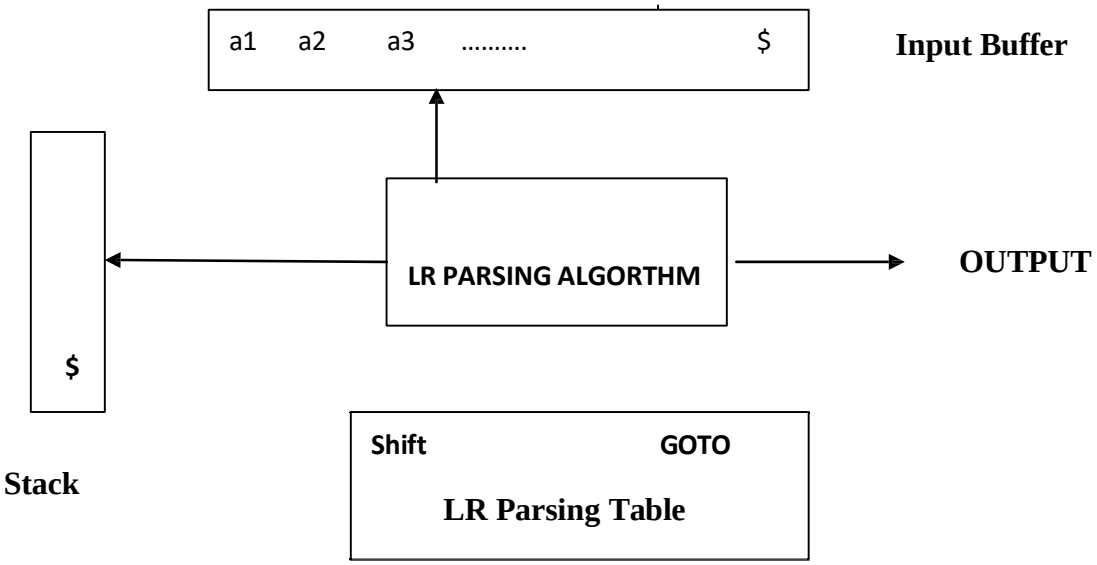


Figure 3.3: Components of LR Parsing

LR Parser Consists of

- ∑ An input buffer that contains the string to be parsed followed by a $ Symbol, used to indicate end of input.
- ∑ A stack containing a sequence of grammar symbols with a $ at the bottom of the stack, which initially contains the Initial state of the parsing table on top of $.
- ∑ A parsing table (M), it is a two dimensional array M[state, terminal or Non terminal] and it contains two parts

1. **ACTION Part**
   The ACTION part of the table is a two dimensional array indexed by state and the input symbol, i.e. **ACTION**[state][input], An action table entry can have one of following four kinds of values in it. They are:
   1. Shift X, where X is a State number.
   2. Reduce X, where X is a Production number.
   3. Accept, signifying the completion of a successful parse.
   4. Error entry.
2. **GO TO Part**
   The GO TO part of the table is a two dimensional array indexed by state and a Non terminal, i.e. GOTO[state][NonTerminal]. A GO TO entry has a state number in the table.

∑ A parsing Algorithm uses the current State X, the next input symbol ‗a' to consult the entry at action[X][a]. it makes one of the four following actions as given below:

1. If the action[X][a]=shift Y, the parser executes a shift of Y on to the top of the stack and advances the input pointer.
2. If the action[X][a]= reduce Y (Y is the production number reduced in the State X), if the production is Y->β, then the parser pops 2*β symbols from the stack and push Y on to the Stack.
3. If the action[X][a]= accept, then the parsing is successful and the input string is accepted.
4. If the action[X][a]= error, then the parser has discovered an error and calls the error routine.

The parsing is classified in to

1. LR (0)
2. Simple LR (1)
3. Canonical LR (1)
4. Look ahead LR (1)

LR (1) Parsing: Various steps involved in the LR (1) Parsing:

1. Write the Context free Grammar for the given input string
2. Check for the Ambiguity
3. Add Augment production
4. Create Canonical collection of LR (0) items
5. Draw DFA
6. Construct the LR (0) Parsing table
7. Based on the information from the Table, with help of Stack and Parsing algorithm generate the output.

Augment Grammar

The Augment Grammar G`, is G with a new starting symbol S` an additional production S` → S. this helps the parser to identify when to stop the parsing and announce the acceptance of the input. The input string is accepted if and only if the parser is about to reduce by S` S. For example let us consider the Grammar below:

E → E+T|T
T → T*F
F → (E) | id **the Augment grammar G` is Represented by**

E` → E
E → E+T|T
T → T*F
F → (E) | id

NOTE: Augment Grammar is simply adding one extra production by preserving the actual meaning of the given Grammar G.

Canonical collection of LR (0) items

LR (0) items

An LR (0) item of a Grammar is a production G with dot at some position on the right side of the production. An item indicates how much of the input has been scanned up to a given point in the process of parsing. For example, if the Production is X → YZ then, The LR (0) items are:

1. X → •AB, indicates that the parser expects a string derivable from AB.
2. X → A•B, indicates that the parser has scanned the string derivable from the A and expecting the string from Y.
3. X → AB•, indicates that he parser has scanned the string derivable from AB.

If the grammar is X → € the, the LR (0) item is

X → •, indicating that the production is reduced one.

Canonical collection of LR(0) Items:

This is the process of grouping the LR (0) items together based on the closure and Go to operations

Closure operation

If I is an initial State, then the Closure (I) is constructed as follows:

1. Initially, add Augment Production to the state and check for the • symbol in the Right hand side production, if the • is followed by a Non terminal then Add Productions which are Stating with that Non Terminal in the State I.

2. If a production X→α•Aβ is in I, then add Production which are starting with X in the State I. Rule 2 is applied until no more productions added to the State I(meaning that the • is followed by a Terminal symbol).

Example :

0.	**E`→ E**		**E`→ • E**
1.	**E→ E+T**	**LR (0) items for the Grammar is**	**E →•E+T**
2.	**T→ F**		**T→ •F**
3.	**T→ T*F**		**T→ •T*F**
4.	**F→ (E)**		**F→ • (E)**
5.	**F→ id**		**F→ • id**

Closure (I_0) State

Add **E `→ • E** in I_0 State

Since, the _•' symbol in the Right hand side production is followed by A Non terminal E. So, add productions starting with E in to Io state. So, the state becomes

E ` • E →

0. E→ •E+T

1. T →•F

The 1st and 2nd productions are satisfies the 2nd rule. So, add productions which are starting with E and T in I_0

Note: once productions are added in the state the same production should not added for the 2nd time in the same state. So, the state becomes

0. E`→ •E
1. E→ •E+T
2. T→ •F
3. T→ •T*F
4. F→ •(E)
5. F→ • id

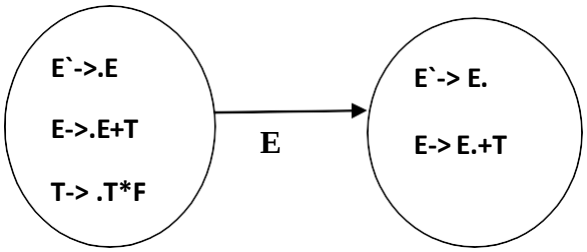
GO TO Operation

Go to (I_0, X), where I_0 is set of items and X is the grammar Symbol on which we are moving the „•" symbol. It is like finding the next state of the NFA for a give State I_0 and the input symbol is X. For example, if the production is **E →E+T**

Go to **(I_0, E)** is **E`→ •E, E→ E•+T**

Note: Once we complete the Go to operation, we need to compute closure operation for the output production

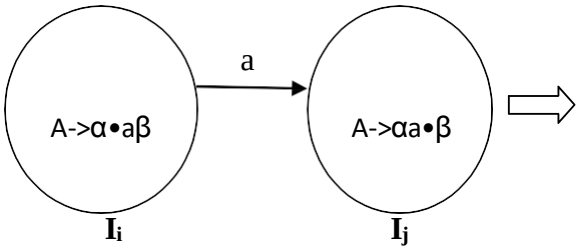
Go to **(I_0, E)** is **E→ E•+T,E`→ E. = Closure ({E`→ E•, E → E•+T})**



Construction of LR (0) parsing Table:

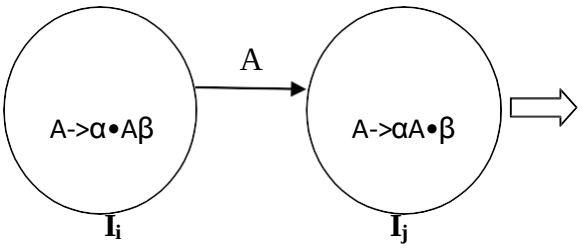
Once we have Created the canonical collection of LR (0) items, need to follow the steps mentioned below:

If there is a transaction from one state (I_i) to another state(I_j) on a terminal value then, we should write the shift entry in the action part as shown below:



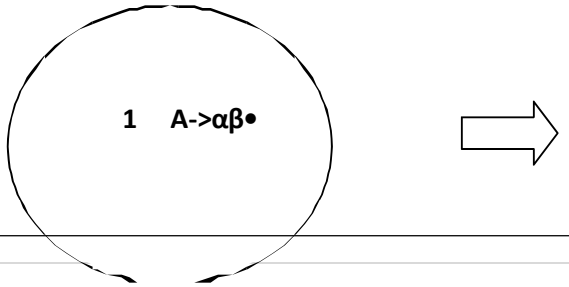
States	ACTION		GO TO
	a	$	A
I_i	S_j		
I_j			

If there is a transaction from one state (I_i) to another state (I_j) on a Non terminal value then, we should write the subscript value of I_i in the GO TO part as shown below: part as shown below:



States	ACTION		GO TO
	a	$	A
I_i			j
I_j			

If there is one state (I_i), where there is one production which has no transitions. Then, the production is said to be a reduced production. These productions should have reduced entry in the Action part along with their production numbers. If the Augment production is reducing then, write accept in the Action part.



States	ACTION		GO TO
	a	$	A
I_i	r_1	r_1	

n

I_i

I_i

For Example, Construct the LR (0) parsing Table for the given Grammar (G)

S → aB

B → bB | b

Sol: 1. Add Augment Production and insert „•" symbol at the first position for every production in G

0. S′ → •S

1. S → •aB

2. B → •bB

3. B →•b

I_0 State:

1. Add Augment production to the I_0 State and Compute the Closure

I_0 = Closure (S′→ •S)

Since _•' is followed by the Non terminal, add all productions starting with S in to I_0 State. So, the I_0 State becomes

I_0 = S′ → •S

S → •aB Here, in the S production _.' Symbol is followed by a terminal value so close the state.

I_1= Go to (I_0, S)

S`→ S•

Closure(S`→ S•) = S′→ S• Here, The Production is reduced so close the State.

I_1= S′ →S•

I_2= Go to (I_0, a) = closure (S → a•B)

Here, the _•' symbol is followed by The Non terminal B. So, add the productions which are Starting B.

I_2= B →•bB

B → •b Here, the _•' symbol in the B production is followed by the terminal value. So, Close the State.

I_2= S → a•B

B → •bB

B → •b

I_3= Go to (I_2 ,B) = Closure (S → aB•) = S → aB•

I_4= Go to (I_2 , b) = closure ({B → b•B, B→ b•})

Add productions starting with B in I_4.

B → •bB
B → •b The Dot Symbol is followed by the terminal value. So, close the State.

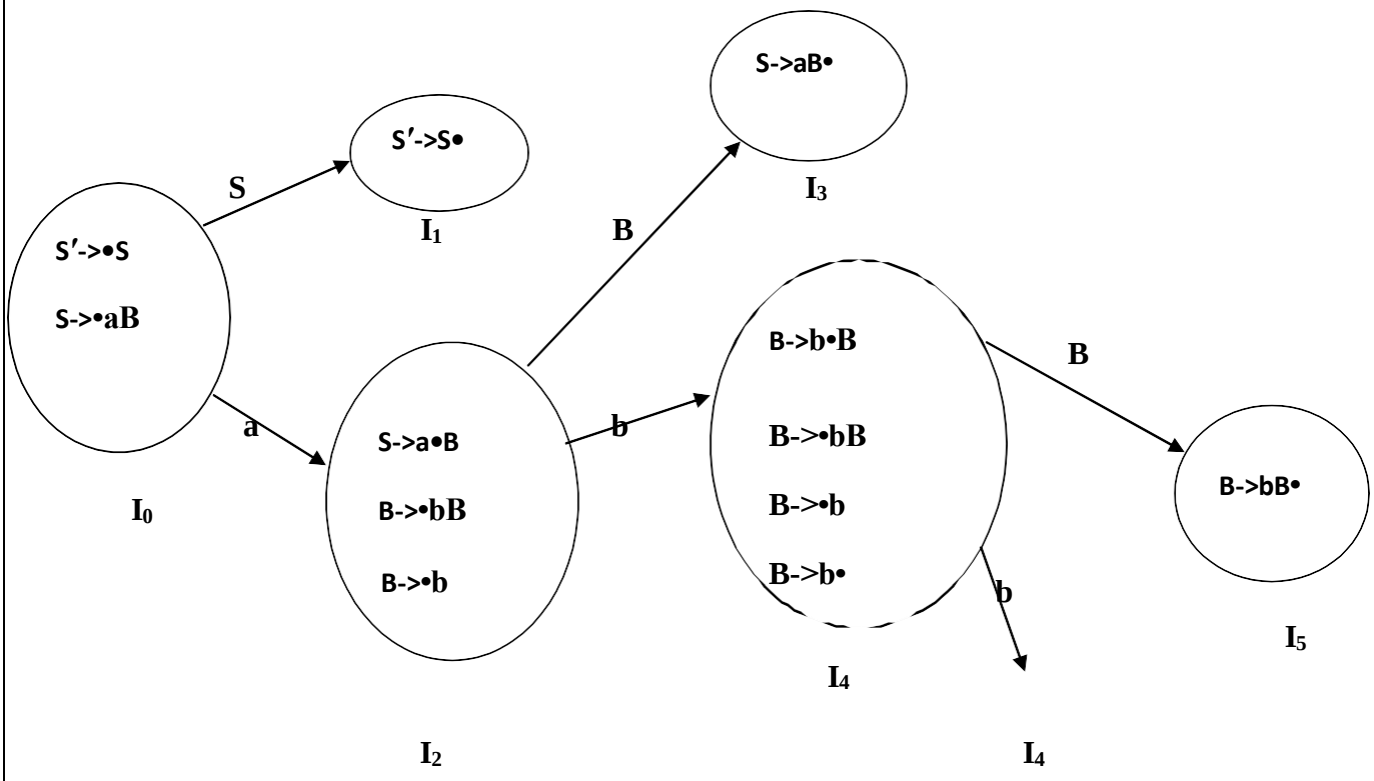
I_4= B → b•B
B → •bB
B → •b
B → b•

I_5= Go to (I_2, b) = Closure (B → b•) = B → b•

I_6= Go to (I_4, B) = Closure (B → bB•) = B → bB•

I_7 = Go to (I_4 , b) = I_4

Drawing Finite State diagram DFA: Following DFA gives the state transitions of the parser and is useful in constructing the LR parsing table.



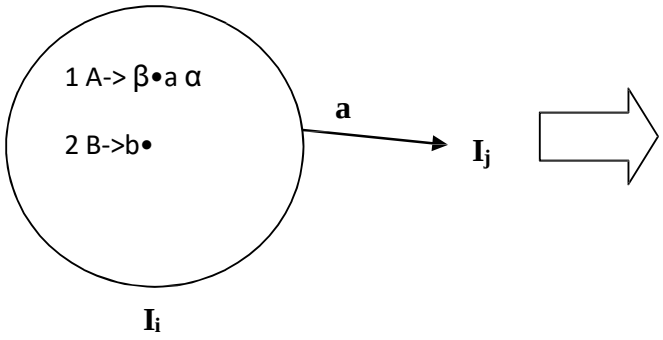
LR Parsing Table:

States	ACTION			GOTO	
	a	B	$	S	B
I_0	S_2			1	
I_1			ACC		
I_2		S_4			3
I_3	R_1	R_1	R_1		
I_4	R_3	S_4/R_3	R_3		5
I_5	R_2	R_2	R_2		

Note: if there are multiple entries in the LR (1) parsing table, then it will not accepted by the LR(1) parser. In the above table I_3 row is giving two entries for the single terminal value _b' and it is called as Shift- Reduce conflict.

Shift-Reduce Conflict in LR (0) Parsing: Shift Reduce Conflict in the LR (0) parsing occurs when a state has

1. A Reduced item of the form A → α• and
2. An incomplete item of the form A → β•aα as shown below:

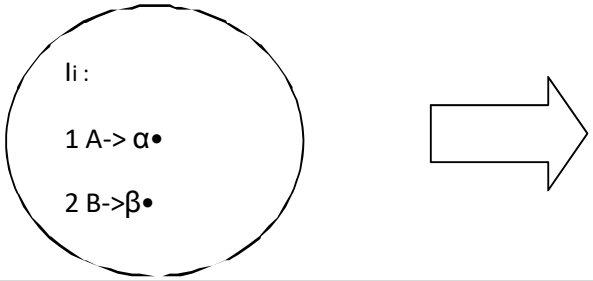


States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2	r_2		
I_j				

Reduce - Reduce Conflict in LR (0) Parsing:

Reduce- Reduce Conflict in the LR (1) parsing occurs when a state has two or more reduced items of the form

1. **A → α•**
2. **B → β•** as shown below:



States	Action		GOTO	
	a	$	A	B
I_i	$r_1/r2$	r_1/ r_2		

SLR PARSER CONSTRUCTION: What is SLR (1) Parsing

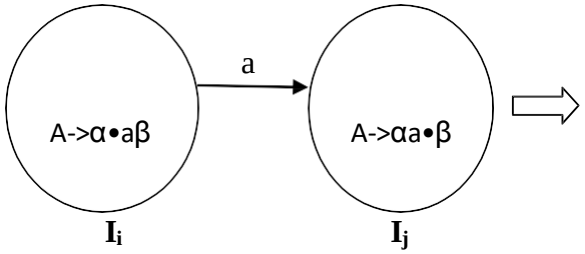
Various steps involved in the SLR (1) Parsing are:

1. Write the Context free Grammar for the given input string
2. Check for the Ambiguity
3. Add Augment production
4. Create Canonical collection of LR (0) items
5. Draw DFA
6. Construct the SLR (1) Parsing table
7. Based on the information from the Table, with help of Stack and Parsing algorithm generate the output.

SLR (1) Parsing Table Construction

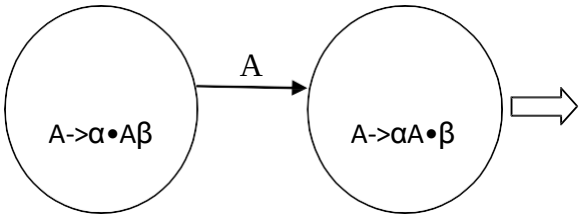
Once we have Created the canonical collection of LR (0) items, need to follow the steps mentioned below:

If there is a transaction from one state ($\mathbf{I_i}$) to another state($\mathbf{I_j}$) on a terminal value then, we should write the shift entry in the action part as shown below:



States	ACTION		GO TO
	a	$	A
$\mathbf{I_i}$	$\mathbf{S_j}$		
$\mathbf{I_j}$			

If there is a transaction from one state ($\mathbf{I_i}$) to another state ($\mathbf{I_j}$) on a Non terminal value then, we should write the subscript value of $\mathbf{I_i}$ in the GO TO part as shown below: part as shown below:



States	ACTION		GO TO
	a	$	A
$\mathbf{I_i}$			**j**
$\mathbf{I_j}$			

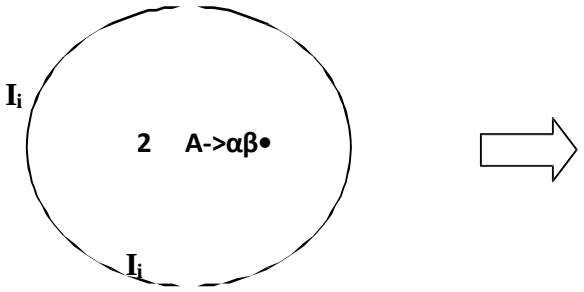
I_i I_j

1 If there is one state (**I_i**), where there is one production (**A->αβ•**) which has no transitions to the next State. Then, the production is said to be a reduced production. For all terminals X in FOLLOW (A), write the reduce entry along with their production numbers. If the Augment production is reducing then write accept.

1 S -> •aAb

2 A->αβ•

Follow(S) = {$}
Follow (A) = (b}



States	ACTION			GO TO	
	a	b	$	S	A
I_i		r_2			

SLR (1) table for the Grammar

S → aB
B → bB | b

Follow (S) = {$}, Follow (B) = {$}

States	ACTION			GOTO	
	A	**b**	**$**	**S**	**B**
I_0	S_2			1	
I_1			ACCEPT		
I_2		S_4			3
I_3			R_1		
I_4		S_4	R_3		5
I_5			R_2		

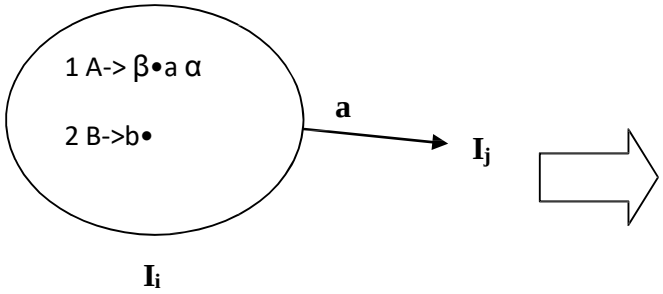
Note: When Multiple Entries occurs in the SLR table. Then, the grammar is not accepted by SLR(1) Parser.

Conflicts in the SLR (1) Parsing :

When multiple entries occur in the table. Then, the situation is said to be a Conflict.

Shift-Reduce Conflict in SLR (1) Parsing : Shift Reduce Conflict in the LR (1) parsing occurs when a state has

1. A Reduced item of the form A → α• and Follow(A) includes the terminal value ‗a'.
2. An incomplete item of the form A → β•aα as shown below:



States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2			

Reduce - Reduce Conflict in SLR (1) Parsing

Reduce- Reduce Conflict in the LR (1) parsing occurs when a state has two or more reduced items of the form

1. **A → α•**
2. **B → β• and Follow (A) ∩ Follow(B) ≠ null** as shown below:

If The Grammar is

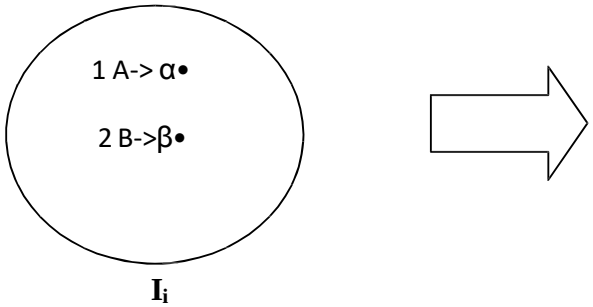
S-> αAaBa

A-> α

B-> β

Follow(S)= {$}

Follow(A)={a} and Follow(B)= {a}



States	Action		GOTO	
	a	$	A	B
I_i	r_1/r2			

Canonical LR (1) Parsing: Various steps involved in the CLR (1) Parsing:

1. Write the Context free Grammar for the given input string
2. Check for the Ambiguity
3. Add Augment production

4. Create Canonical collection of LR (1) items
5. Draw DFA
6. Construct the CLR (1) Parsing table
7. Based on the information from the Table, with help of Stack and Parsing algorithm generate the output.

LR (1) items :

The LR (1) item is defined by **production**, **position of data** and a **terminal symbol**. The terminal is called as ***Look ahead symbol***.

General form of LR (1) item is

S->α•Aβ , $
A-> •γ, FIRST(β,$)

Rules to create canonical collection:

1. Every element of I is added to closure of I
2. If an LR (1) item [**X-> A•BC, a]** exists in I, and there exists a production B->b_1b_2....., then add item **[B->• b_1b_2, z]** where z is a terminal in **FIRST(Ca),** if it is not already in Closure(I).keep applying this rule until there are no more elements adde.

For example, if the grammar is

S->CC
C->cC
C->d

The Canonical collection of LR (1) items can be created as follows:

0. S′->•S (Augment Production)
1. S->•CC
2. C->•cC
3. C->•d

I_0 State : Add Augment production and compute the Closure, the look ahead symbol for the Augment Production is $.

S′->•S, $= Closure(S′->•S, $)

The dot symbol is followed by a Non terminal S. So, add productions starting with S in I_0 State.

S->•CC, FIRST ($), using 2nd rule

S->•CC, $

The dot symbol is followed by a Non terminal C. So, add productions starting with C in I_0 State.

C->•cC, FIRST(C, $)
C->•d, FIRST(C, $)

FIRST(C) = {c, d} so, the items are

C->•cC, c/d
C->•d, c/d

The dot symbol is followed by a terminal value. So, close the I_0 State. So, the productions in the **I_0** are

S'->•S , $
S->•CC , $
C->•cC, c/d
C->•d , c/d

I_1 = Goto (I_0, S)= S'->S•,$

I_2 = Go to (I_0 , C)= Closure(S-> C•C, $)

S-> **C->•cC , $**
C->•d,$ So, the I_2 State is

S->C•C,$
C->•cC , $
C->•d,$

I_3= Goto(I_0,c)= Closure(C->c•C, c/d)
C->•cC, c/d
C->•d , c/d So, the I_3 State is

C->c•C, c/d
C->•cC, c/d
C->•d , c/d

I_4= Goto(I_0, d)= Colsure(C->d•, c/d) = C->d•, c/d

I_5 = Goto (I_2, C)= closure(S->CC•,$)= S->CC•, $

I_6= Goto (I_2, c)= closure(C->c•C , $)=
C->•cC, $
C->•d , $ S0, the I_6 State is

C->c•C , $
C->•cC , $
C->•d,$

I_7 = Go to (I_2 , d)= Closure(C->d•,$) = C->d•, $

Goto(I_3, c)= closure(C->•cC, c/d)= I_3.

I_8= Go to (I_3 , C)= Closure(C->cC•, c/d) = C->cC•, c/d

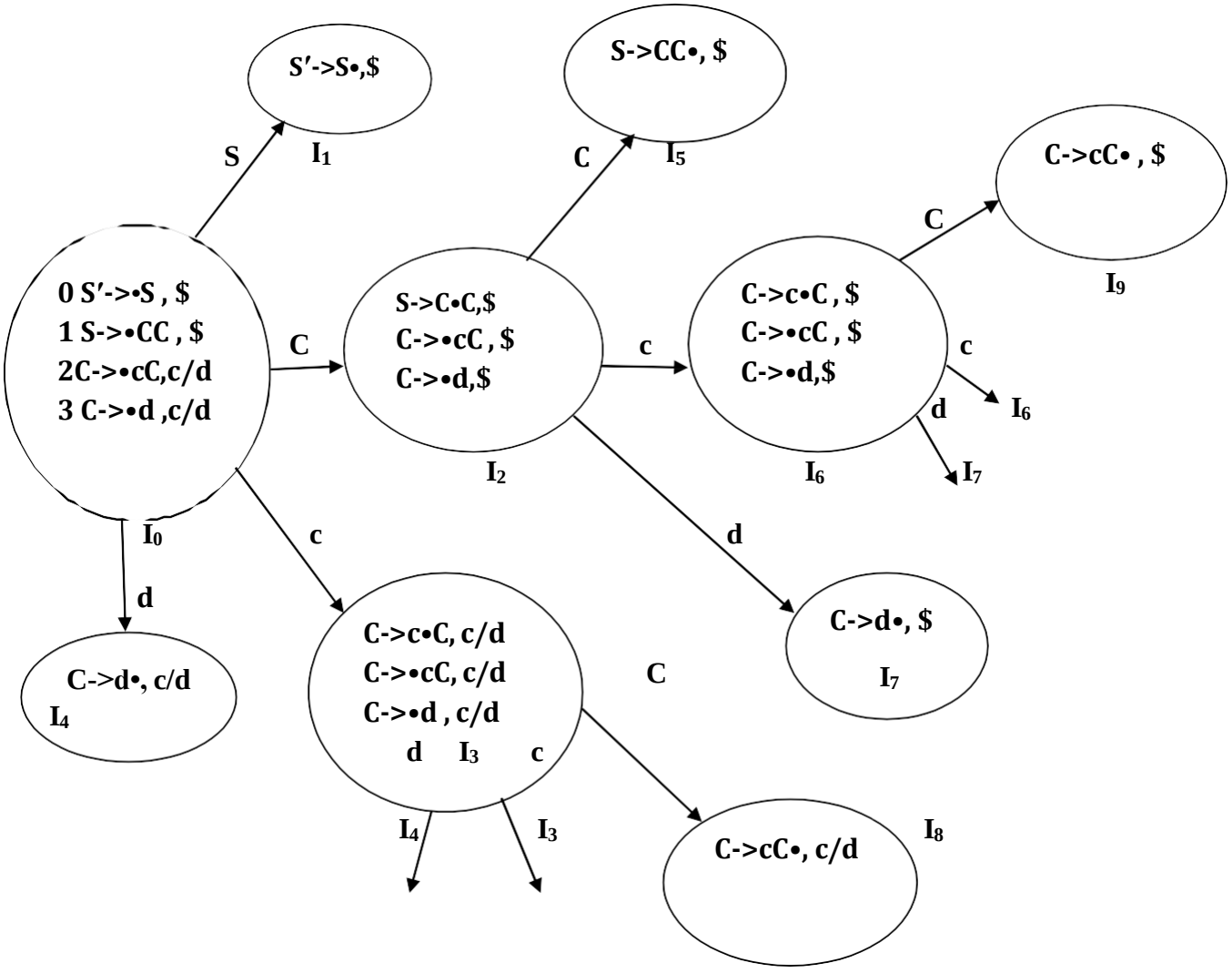
Go to (I_3 , c)= Closure(C->c•C, c/d) = I_3

Go to (I_3 , d)= Closure(C->d•, c/d) = I_4

I_9= Go to (I_6 , C)= Closure(C->cC• , $) = C->cC• , $
Go to (I_6 , c)= Closure(C->c•C , $) = I_6

Go to (I_6 , d)= Closure(C->d•,$) = I_7

Drawing the Finite State Machine DFA for the above LR (1) items



Construction of CLR (1) Table

Rule1: if there is an item [A->α•Xβ,b] in I_i and goto(I_i,X) is in I_j then action [I_i][X]= Shift j, Where X is Terminal.

Rule2: if there is an item [A->α•, b] in I_i and (A≠S`) set action [I_i][b]= reduce along with the production number.

Rule3: if there is an item [S`->S•, $] in I_i then set action [I_i][$]= Accept.

Rule4: if there is an item [A->α•Xβ,b] in I_i and go to(I_i,X) is in I_j then goto [I_i][X]= j, Where X is Non Terminal.

States	ACTION			GOTO	
	c	d	$	S	C
I_0	S_3	S_4		1	2
I_1			ACCEPT		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			5
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

Table : LR (1) Table

LALR (1) Parsing

The CLR Parser avoids the conflicts in the parse table. But it produces more number of States when compared to SLR parser. Hence more space is occupied by the table in the memory. So LALR parsing can be used. Here, the tables obtained are smaller than CLR parse table. But it also as efficient as CLR parser. Here LR (1) items that have same productions but different look-aheads are combined to form a single set of items.

For example, consider the grammar in the previous example. Consider the states I_4 and I_7 as given below:

I_4= Goto(I_0, d)= Colsure(C->d•, c/d) = C->d•, c/d

I_7 = Go to (I_2 , d)= Closure(C->d•,$) = C->d•, $

These states are differing only in the look-aheads. They have the same productions. Hence these states are combined to form a single state called as I_{47}.

Similarly the states I_3 and I_6 differing only in their look-aheads as given below:

I_3= Goto(I_0,c)=

C->c•C, c/d
C->•cC, c/d
C->•d , c/d

I_6= Goto (I_2, c)=
C->c•C , $
C->•cC , $
C->•d,$

These states are differing only in the look-aheads. They have the same productions. Hence these states are combined to form a single state called as I_{36}.

Similarly the States I_8 and I_9 differing only in look-aheads. Hence they combined to form the state I_{89}.

States	ACTION			GOTO	
	c	d	$	S	C
I_0	S_{36}	S_{47}		1	2
I_1			ACCEPT		
I_2	S_{36}	S_{47}			5
I_{36}	S_{36}	S_{47}			89
I_{47}	R_3	R_3	R_3		5
I_5			R_1		
I_{89}	R_2	R_2	R_2		

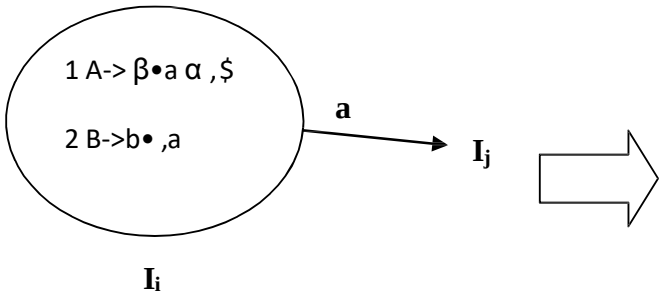
Table: LALR Table

Conflicts in the CLR (1) Parsing : When multiple entries occur in the table. Then, the situation is said to be a Conflict.

Shift-Reduce Conflict in CLR (1) Parsing

Shift Reduce Conflict in the CLR (1) parsing occurs when a state has

3. A Reduced item of the form A → α•, a and
4. An incomplete item of the form A → β•aα as shown below:

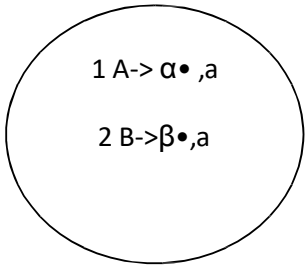


States	Action		GOTO	
	a	$	A	B
I_i	Sj/r2			

Reduce / Reduce Conflict in CLR (1) Parsing

Reduce- Reduce Conflict in the CLR (1) parsing occurs when a state has two or more reduced items of the form

3. **A→ α•**
4. **B→ β• If two productions in a state (I) reducing on same look ahead symbol as shown below:**



I_i

States	Action		GOTO	
	a	$	A	B
I_i	$r_1/r2$			

String Acceptance using LR Parsing:

Consider the above example, if the input String is **cdd**

States	ACTION			GOTO	
	c	**D**	**$**	**S**	**C**
I_0	S_3	S_4		1	2
I_1			ACCEPT		
I_2	S_6	S_7			5
I_3	S_3	S_4			8
I_4	R_3	R_3			5
I_5			R_1		
I_6	S_6	S_7			9
I_7			R_3		
I_8	R_2	R_2			
I_9			R_2		

0 S′->•S (Augment Production)

1 S->•CC

2 C->•cC

3 C->•d

STACK	INPUT	ACTION
$0	cdd$	Shift S_3
$0c3	dd$	Shift S_4
$0c3d4	d$	Reduce with R3,C->d, pop 2*β symbols from the stack
$0c3C	d$	Goto (I_3, C)=8Shift S_6

$0c3C8	d$	Reduce with R_2 ,C->cC, pop 2*β symbols from the stack
$0C	d$	Goto (I_0, C)=2
$0C2	d$	Shift S_7
$0C2d7	$	Reduce with R3,C->d, pop 2*β symbols from the stack
$0C2C	$	Goto (I_2, C)=5
$0C2C5	$	Reduce with R_1,S->CC, pop 2*β symbols from the stack
$0S	$	Goto (I_0, S)=1
$0S1	$	Accept

Handing Ambiguous grammar

Ambiguity: A Grammar can have more than one parse tree for a string . For example, consider grammar.

string ~~str~~ing + string
| string - string
| 0 | 1 | . | 9

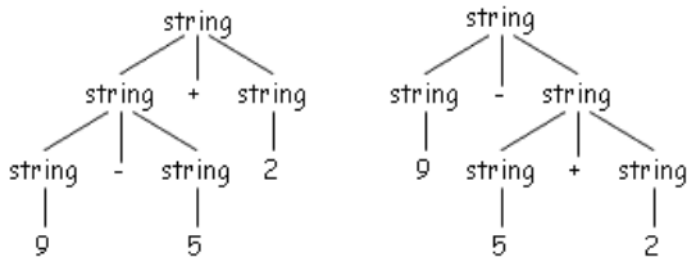
String 9-5+2 has two parse trees

A grammar is said to be an ambiguous grammar if there is some string that it can generate in more than one way (i.e., the string has more than one parse tree or more than one leftmost derivation). A language is inherently ambiguous if it can only be generated by ambiguous grammars.

For example, consider the following grammar:

string ~~str~~ing + string
| string - string
| 0 | 1 | . | 9

In this grammar, the string 9-5+2 has two possible parse trees as shown in the next slide.



Consider the parse trees for string 9-5+2, expression like this has more than one parse tree. The two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5)+2 and 9-(5+2). The second parenthesization gives the expression the value 2 instead of 6.

- ∑ Ambiguity is problematic because meaning of the programs can be incorrect
- ∑ Ambiguity can be handled in several ways

- Enforce associativity and precedence

- Rewrite the grammar (cleanest way)

There are no general techniques for handling ambiguity, but

. It is impossible to convert automatically an ambiguous grammar to an unambiguous one

Ambiguity is harmful to the intent of the program. The input might be deciphered in a way which was not really the intention of the programmer, as shown above in the 9-5+2 example. Though there is no general technique to handle ambiguity i.e., it is not possible to develop some feature which automatically identifies and removes ambiguity from any grammar. However, it can be removed, broadly speaking, in the following possible ways:-

1) Rewriting the whole grammar unambiguously.

2) Implementing precedence and associatively rules in the grammar. We shall discuss this technique in the later slides.

If an operand has operator on both the sides, the side on which operator takes this operand is the associativity of that operator

. In a+b+c b is taken by left +
. +, -, *, / are left associative
. ^, = are right associative

Grammar to generate strings with right associative operators right à letter = right | letter letter
⟶a| b |.| z

A binary operation * on a set S that does not satisfy the associative law is called non-associative. A left-associative operation is a non-associative operation that is conventionally evaluated from left to right i.e., operand is taken by the operator on the left side.
For example,
6*5*4 = (6*5)*4 and not 6*(5*4)
6/5/4 = (6/5)/4 and not 6/(5/4)

A right-associative operation is a non-associative operation that is conventionally evaluated from right to left i.e., operand is taken by the operator on the right side.

For example,

6^5^4 => 6^(5^4) and not (6^5)^4)
x=y=z=5 => x=(y=(z=5))

Following is the grammar to generate strings with left associative operators. (Note that this is left recursive and may go into infinite loop. But we will handle this problem later on by making it right recursive)

left ⟶ left + letter | letter
letter ⟶ a | b | .....| z

IMPORTANT QUESTIONS

1. Discuss the the working of Bottom up parsing and specifically the OperatorPrecedence Parsing with an exaple?
2. What do you mean by an LR parser? Explain the LR (1) Parsing technique?
3. Write the differences between canonical collection of LR (0) items and LR (1) items?
4. Write the Difference between CLR (1) and LALR(1) parsing?
5. What is YACC? Explain how do you use it in constructing the parser using it.

QUESTIONS

1. Explain the conflicts in the Shift reduce Parsing with an example?
2. E → E+T|T
   T → T*F
   F → (E)|id , construct the LR(1) Parsing table? And explain the Conflicts?
3. E → E+T|T
   T → T*F
   F → (E)|id , construct the SLR(1) Parsing table? And explain the Conflicts?
4. E → E+T|T
   T → T*F
   F → (E)|id, construct the CLR(1) Parsing table? And explain the Conflicts?
5. E → E+T|T
   T → T*F
   F → (E) |id, construct the LALR (1) Parsing table? And explain the Conflicts?