3 Parsing

3. CONTEXT FREE GRAMMAR

In context free grammar, we are using BNF (Backus Normal Form) notation to express the syntactic specification (Rules of language) of a computer language.

- A context free grammar gives a precise, yet easy to understand syntactic specification for the programs of a particular programming language.
- An efficient parser can be constructed automatically from a properly designed grammar.
- A context free grammar imparts a structure to a program that is useful for its translation into object code and for the detection of errors.
- In general, a grammar involves four sets G(V,T,P,S): a set of nonterminal or set of variables (V), a set of terminals(T), a set of production rules(P) and a start symbol(S).

Example: $E \rightarrow EAE \mid (E) \mid -E \mid id$

$A \rightarrow + \mid - \mid * \mid / \mid \uparrow$

V = {A,E}
T = {+ ,-, * , / , ↑ ,(,),id}
S = E
P = {
E → EAE,
E → (E)
E → –E
E → id ,
A → + ,
A → ↑,
A → *,
A → / ,
A → -}

Parse tree:

- A graphical form for derivations that eliminates the replacement order option.
- Tree representation of the derivation is known as the derivation tree / parse tree.
- Parse tree is the hierarchical syntactic structure of a string that is implied by the grammar.
- The leaves of the parse tree are labelled by terminals and read from left to right, they constitute a sentential form called yield or frontier of the tree.
- Each step in the derivation is one sentential form.
- If the derivation is left most, then the sentential form is left sentential form.
- If the derivation is right most, then the sentential form is the right sentential form.

Ambiguous and unambiguous grammar:

- A grammar is ambiguous if it provides multiple parse trees for the same string.
- In another way ∃(there exists) more than one parse tree for an input string.
- Ambiguous grammar is one that produces more than one left most derivation or more than one rightmost derivation for a string.
- A grammar is said to be unambiguous if every string in the language produced by that grammar has a unique parse tree.
- It is desirable for some types of parsers that the grammar be made unambiguous; otherwise, we will not be able to identify which parse tree to select for a string in a unique way.

Example:

Consider the following grammar $S \rightarrow SS / a$, for input w = 'aaa' we have more than one parse tree so, given grammar is ambiguous.

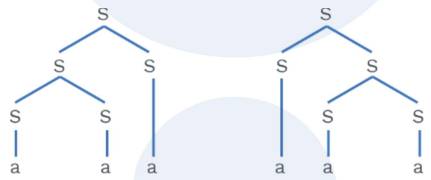


Fig. 3.1

Note

If the grammar is both left recursive and right recursive for a terminal then it is an ambiguous grammar.

Conversion from ambiguous to unambiguous grammar:

- If the grammar is ambiguous, we can describe the associativity and precedence of arithmetic operators to resolve the ambiguity.
- There is no such algorithm to convert all the ambiguous grammar into an equivalent unambiguous grammar. So, this problem is undecidable.
- There is no algorithm that can determine the ambiguity of a CFG, so it is undecidable.

Rack Your Brain

Which of the following grammar is ambiguous:

a) $S \rightarrow SaS \mid SbS \mid \in$

b) $E \rightarrow E + E \mid id$

c) $S \rightarrow SSS \mid a$

d) $E \rightarrow E + id \mid id$

- The ambiguous grammar (for which no other unambiguous grammar exists) produces the same language as the ambiguous grammar is called inherently ambiguous grammar.

SOLVED EXAMPLES

Q.1 **Consider the following grammar for arithmetic expression.**

$$E \rightarrow E + E \mid E * E \mid id$$

Check the above grammar is ambiguous or not, if ambiguous, then convert into equivalent unambiguous grammar.

Sol:

- The given grammar is ambiguous as the grammar is both left recursive and right recursive. In another way, we can say that we have more than one parse tree for id + id * id.
- In the given grammar, associativity and precedence of operators are not maintained as all operators are at the same level.
- Associativity and precedence of operators are sufficient to disambiguate both grammars.
- The input id + id * id has two distinct leftmost derivations with the corresponding parser tree.

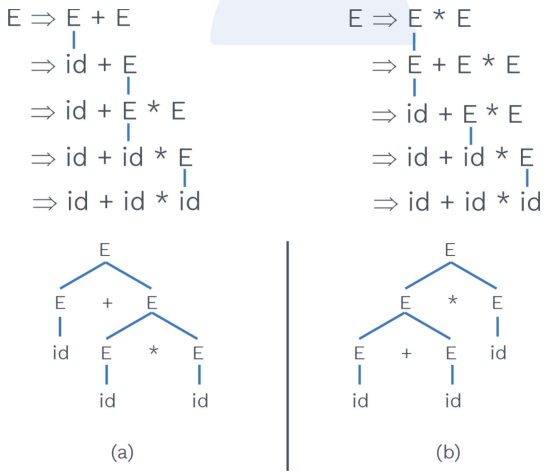


Fig. 3.2 Derivation Trees

- The parse tree of figure 'a' is correct in that it reflects the commonly assumed precedence of + and * while the figure 'b' does not.
- According to C language rules, '*' as having higher precedence than '+' so '*' is evaluated first, then '+'.

- If we take input as 2 + 3 * 4 in the first parse tree, we will get the correct output i.e. 14.
- As * and + have left associative, id * id * id means (id * id) * id and id + id + id implies (id + id) + id.
- The final unambiguous grammar is:

$$E \rightarrow E + T / T$$
$$T \rightarrow T * F / F$$
$$F \rightarrow id$$

Previous Years' Question

Consider the grammar defined by the following production rules, with two operators * and +

$$S \rightarrow T * P$$

$$T \rightarrow U \mid T * U$$

$$P \rightarrow Q + P \mid Q$$

$$Q \rightarrow Id$$

$$U \rightarrow Id$$

Which one of the following is TRUE?

a) + is left associative while * is right associative
b) + is right associative, while * is left associative
c) Both + and * right associative
d) Both + and * are left associative

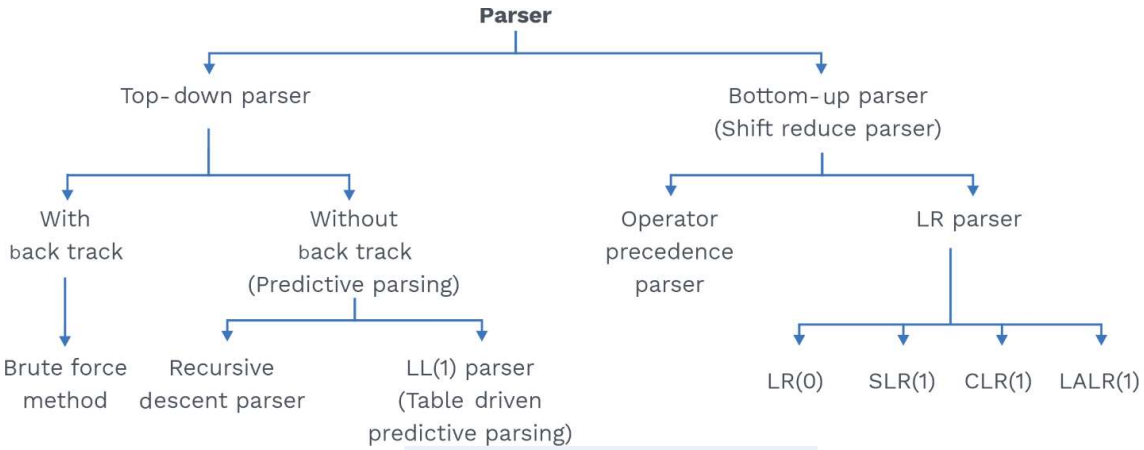
Sol: b) **[GATE: CS 2014]**

3.1 PARSER

- For a grammar G, parser is a program that accepts a string as input and produces output as a parse tree.
- Syntax analyzers are also referred to as parsers.
- There are two types of parsers: top-down and bottom-up.

Rack Your Brain

Convert the given ambiguous grammar into unambiguous grammar.
Boolean expression → not (Boolean expression) / Boolean expression and Boolean expression / Boolean expression or Boolean expression / True / False

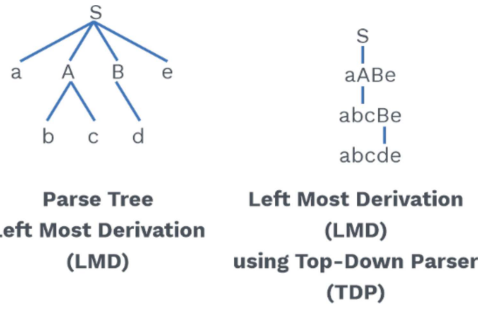


Top-down parser(TDP):

- The top-down parsers build the parse tree starting from the root node and work down to the leaf nodes.
- The input to the parser is scanned from left to right, one symbol at a time, in a top-down parser.
- For example, consider the grammar, draw the parse tree for input 'abcde' using top-down parser

$S \rightarrow aABe$
$A \rightarrow bc / b$
$B \rightarrow d$

- Top-down parser follows Left Most Derivation (LMD).



Parse Tree Left Most Derivation (LMD)

Left Most Derivation (LMD) using Top-Down Parser (TDP)

Fig. 3.3 Derivation Tree

Bottom-up parser:

- The bottom-up parsers construct a parse tree from the leaves (strings) and proceed to start symbol.

- In bottom-up parser the input to the parser is being scanned from left to right, one symbol at a time.
- A bottom-up parser builds a parse tree by reversing the rightmost derivation.
- At each stage, a string matching the right side of a production is replaced with the symbol on the left in a bottom-up parser.
- For example, consider the grammar.

$$S \rightarrow aAcBe$$
$$A \rightarrow Ab / b$$
$$B \rightarrow d$$

Draw the parse tree for input abbcde using bottom-up parser.

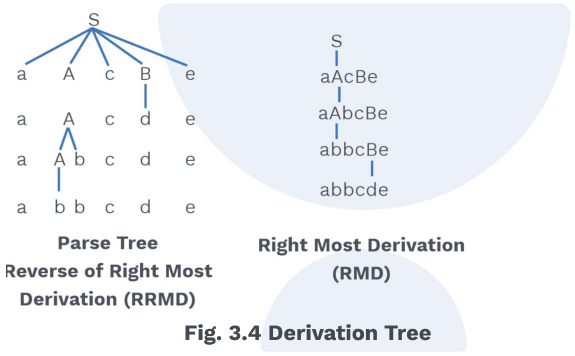


Fig. 3.4 Derivation Tree

Difference between top-down parser and bottom-up parser:

Top-down Parser	Bottom-up Parser
1) A parse tree is created from the root (top) to leaves (Bottom). **2)** The traversal of parse tree is a preorder traversal. **3)** It uses left most derivation. **4)** Expansion of tree. **5)** Less powerful.	**1)** A parse tree is created from the leaves (Bottom) to root (Top). **2)** The traversal of parse tree is a reversal of post order traversal. **3)** It is the rightmost derivation , in the reverse order. **4)** Reduction of tree. **5)** More powerful than topdown parser.

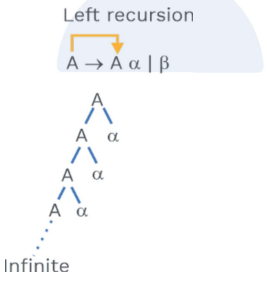
Handles:

- Reduction is the process of replacing the right side of the production with the left side of the production.
- The substring that gets replaced by its left side of the production is called a handle.
- In the previous example, “abbcde” is reduced to “abbcBe” where “d” is the handle as in the grammar, we have $B \rightarrow d$. Again “abbcBe” is reduced to “aAbcBe” where “b” is a handle as in the grammar, we have $A \rightarrow b$.
- If grammar is unambiguous, it has precisely one handle for each right sentential form.

Recursive and non-recursive grammar:

- A grammar can be classified into recursive and nonrecursive grammar.
- In recursive grammar $\exists$(there exists) atleast one production has the same variable in LHS and RHS. For e.g. $S \rightarrow Sa \mid b$.
- In non recursive grammar, no production has the same variable in LHS and RHS.
- Recursive grammar is further classified into left recursive and right recursive grammar.
- If grammar has a nonterminal A, it is left recursive, if there is a derivation $A \rightarrow A\alpha$ (Where A is at left most place) for some α.
- A left recursive grammar can cause a topdown parser to go into an infinite loop.

For example:



Elimination of left recursion:

Consider the left recursive pair of productions where the left recursion can be eliminated by replacing the pair of production with the following rules:

Rule 1: $A \rightarrow A\alpha \mid \beta$ is replaced by

$$A \rightarrow \beta A'$$

$$A' \rightarrow \in \mid \alpha A'$$

Rule 2: $A \rightarrow A\alpha \mid \beta_1 \mid \beta_2 \mid \beta_3 ..... \mid \beta_n$ is replaced by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' ..... \mid \beta_n A'$$

$$A' \rightarrow \in \mid \alpha A'$$

Rule 3: $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 ..... \mid A\alpha_n \mid \beta$ is replaced by

$$A \rightarrow \beta A'$$

$$A' \rightarrow \in \mid \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' ..... \mid \alpha_n A'$$

Rule 4: $A \rightarrow A\alpha_1 \mid A\alpha_2 .....A\alpha_n \mid \beta_1 \mid \beta_2 .....\beta_n$ is replaced by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \beta_3 A' ..... \mid \beta_n A'$$

$$A' \rightarrow \in \mid \alpha_1 A' \mid \alpha_2 A' ..... \mid \alpha_n A'$$

SOLVED EXAMPLES

Q.2 **Remove the left recursion from the grammar below.**

$A \rightarrow A + B \mid B$

$B \rightarrow B * C \mid C$

$C \rightarrow (D) \mid d$

Sol: Using rule 1

$$A \rightarrow BA'$$

$$A' \rightarrow \in \mid +BA'$$

$$B \rightarrow CB'$$

$$B' \rightarrow \in \mid *CB'$$

$$C \rightarrow (D) \mid d$$

Previous Years' Question

Which one of the following grammars is free from left recursion?

a) $S \rightarrow AB$
$A \rightarrow Aa \mid b$
$B \rightarrow c$

b) $S \rightarrow Ab \mid Bb \mid c$
$A \rightarrow Bd \mid \in$
$B \rightarrow e$

c) $S \rightarrow Aa \mid B$
$A \rightarrow Bb \mid Sc \mid \in$
$B \rightarrow d$

d) $S \rightarrow Aa \mid Bb \mid c$
$A \rightarrow Bd \mid \in$
$B \rightarrow Ae \mid \in$

a) a **b)** b **c)** c **d)** d

Sol: b)

[GATE: CS 2016 (Set-2)]

Left factoring:

- Consider a production $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ and an input that begins with a non-empty string derived from α, it is not clear whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$.
- We can solve this by left factoring

$$A \rightarrow \alpha A'$$
$$A' \rightarrow \beta_1 \mid \beta_2$$

- The grammar that is suitable for predictive parsing (after elimination of left recursion and left factoring) are called LL(K) grammars. Where first 'L' implies that we can scan from left to right, the second L stands for leftmost derivation, and K stands for the number of tokens of look ahead.

Rack Your Brain

Eliminate the left recursion from the following grammar:

$$E \rightarrow AaB$$
$$A \rightarrow aA \mid Ba$$
$$B \rightarrow AB \mid b$$

SOLVED EXAMPLES

Q.3 **Consider the following grammar**

$A \rightarrow aAb \mid aAc \mid d$

$A \rightarrow aA \mid \in$

$B \rightarrow bB \mid \in$

Remove left factoring?

Sol: Consider the following grammar

$$A \rightarrow aAb \mid aAc \mid d$$

Write down the left factored form of the above grammar.

$$A \rightarrow aAA' \mid d$$
$$A' \rightarrow b \mid c$$

Recursive-descent parser:

- A parser that uses a set of recursive procedures to recognize its input with no backtracking is called recursive-descent parser.
- Recursive descent parsers are simple and easy to implement.
- A recursive descent parser needs a larger stack due to the recursion of the procedures.

Predictive parser / LL(1) parser:

- By explicitly preserving the stack, a predictive parser is an efficient approach for achieving recursive-descent parsing.
- An input buffer, a stack, a parsing table, and an output are all included in the predictive parser.

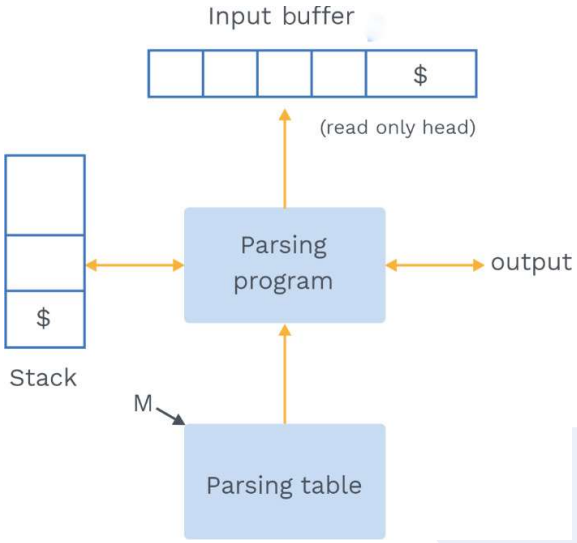


Fig. 3.5 Parser Diagram

Rack Your Brain

Consider the following grammar

1) S → iEts | iEtSes | a

E → b

2) S → assbs | asasb | asb | a

Write down the left factored form of the above grammar.

Model of predictive parsers / LL(1) parser:

- The string to be parsed is placed in an Input buffer, followed by $ (a symbol that indicates the end of the input).
- The sequence of grammar symbols in a stack, with $ at the bottom of the stack marker, is called a stack.
- The stack starts with the grammar's start symbol, which is preceded by $.
- The grammar's start symbol, which is preceded by $, is at the top of the stack.
- The parsing table is a two-dimensional array M [A, a], where A denotes non-terminal, and a denotes the terminal symbols including $.

Predictive parsing program / LL(1) algorithm:

If a is the current input symbol and X is the top of the stack.

1) If X = a = $, the parser halts and announces successful completion of parsing.

2) If X = a ≠ $, then pop-top of stack and increment input pointer.

3) If X is a nonterminal, the program consults the parsing table M's entry M[X, a]. This entry will be either a grammar production or an error entry. If M [X, a] = {X → uvw}, the parser substitutes wvu for X at the top of the stack (with u on top of stack)

4) If M[X, a] = blank then error.

First and follow:

We require two functions connected with a grammar 'G' to populate the entries of a predictive parsing table. FIRST() and FOLLOW() are these functions.

Rules for computing first:

The set of all the terminals that X can begin with is the FIRST set of non-terminal X.

Rule 1: If X is a terminal, then FIRST(X) = {X}

Rule 2: For a non-terminal X, if there exists a production $X \rightarrow \in$, then add $\in$ to FIRST(X).

Rule 3: For a production $X \rightarrow Y_1Y_2Y_3.....Y_k.....Y_n$

FIRST(X) = FIRST(Y_1), if FIRST(Y_1) does not contain $\in$

= FIRST(Y_1) U FIRST(Y_2), if FIRST(Y_1) contains $\in$ and FIRST (Y_2) does not contain $\in$

= FIRST(Y_1) U FIRST(Y_2) U FIRST(Y_3), if FIRST(Y_1), FIRST(Y_2) both contain $\in$ and FIRST(Y_3) does not contain $\in$.

Generalising:

FIRST(X) = FIRST(Y_1) U FIRST(Y_2) U FIRST(Y_3) U FIRST(Y_k) if FIRST(Y_1), FIRST(Y_2) ..... FIRST(Y_{k-1}) all contain $\in$ and FIRST(Y_k) does not contain $\in$.

FIRST(X) = FIRST(Y_1) U FIRST(Y_2) U FIRST(Y_3) U..... FIRST(Y_n) U {$\in$} if FIRST(Y_1), FIRST(Y_2) ..... FIRST(Y_n) all contain $\in$

Rules for computing follow:

A NON-TERMINAL SET OF FOLLOWING A is the collection of all terminals that can be followed by A.

Rule 1:

1) FOLLOW(S) = $, where S is the start symbol and $ is the symbol to indicate the end of input.

2) If the Grammar is

$$S \rightarrow \alpha AB \qquad B \not\rightarrow \in$$

Then FOLLOW (A) = FIRST (B)

3) If the Grammar is

$$S \rightarrow \alpha B$$

FOLLOW (B) = FOLLOW (S)

4) If the Grammar is

$S \rightarrow \alpha AB \qquad B \rightarrow \in$

FOLLOW (A) = {FIRST(B)– $\in$} $\cup$ {FOLLOW(S)}

Note

FOLLOW of a non-terminal does not contain $\in$.

SOLVED EXAMPLES

Q.4 **Find FIRST and FOLLOW set of the following grammar.**
$S \rightarrow AB$; $A \rightarrow aA \mid \in$; $B \rightarrow bB \mid \in$

Sol:

	FIRST	FOLLOW
S	a, b, $\in$	$
A	a, $\in$	b, $
B	b, $\in$	$

LL(1) Parsing table construction algorithm:

1) For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.

2) For each terminal a in FIRST (α), Add $A \rightarrow \alpha$ to M[A, a]

3) If FIRST(α) contain $\in$, add $A \rightarrow \alpha$ to M[A, x] for each terminal x in FOLLOW (A).
If $\in$ is in FIRST(α) and $ is in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, $]

4) Make each undefined entry of M as error.

Rack Your Brain

Find FIRST and follow set of the following grammar:

$S \rightarrow AbB \mid CdB \mid Ba$

$A \rightarrow ea \mid BC$

$B \rightarrow f \mid \in$

$C \rightarrow g \mid \in$

SOLVED EXAMPLES

Q.5 **Suppose the input is "id + id * id", to parse this input construct LL(1) parsing table considering the following grammar.**

E → TE'

E' → +TE' |∈

T → FT'

T' → *FT' |∈

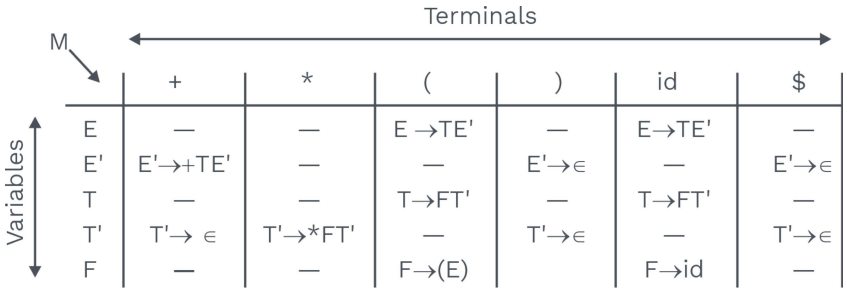
F → (E) | id

and parse the input id + id * id

Sol:

	FIRST	FOLLOW
E	(, id	), $
E'	+, ∈	), $
T	(, id	+,), $
T'	*, ∈	+,), $
F	(, id	+, *,), $

- Add E → TE', FIRST(TE') = FIRST(T) = {(, id}
- Production E → TE' causes M[E, ( ] and M[E, id] to acquire the entry E → TE'
- Production E' → +TE' causes M[E', +] to acquire E' → +TE'
- Production E' →∈ causes M[E',)] and M[E', $] to acquire E' →∈ since FOLLOW(E') = {), $}
- Production T → FT' causes M[T, ( ] and M[T, id] to acquire T → FT' since FIRST(F) = {(, id}
- Production T' → *FT' causes M[T', *] to acquire T' → *FT'
- Production T' →∈ causes M[T, +], M[T,)] and M[T, $] to acquire T' → ∈ since FOLLOW(T') = {+,), $}
- Production F → (E) causes M[F, (] to acquire F → (E)
- Production F → id causes M[F, id] to acquire F → id



All the entries in LL(1) table contain single entries so the grammar is LL(1). Now with the help of LL(1) parsing table, we can parse input id + id * id.

Moves by predictive parser:

Stack	Input	Output
$E	id+id*id$	
$E'T	id+id*id$	E → TE'
$E'T'F	id+id*id$	T → FT'
$E'T'id	id+id*id$	F → id
$E'T'	+id*id$	
$E'	+id*id$	T' → ∈
$E'T+	+id*id$	E' → +TE'
$E'T	id*id$	
$E'T'F	id*id$	T → FT'
$E'T'id	id*id$	F → id
$E'T'	*id$	
$E'T'F*	*id$	T' → * FT'
$E'T'F	id$	

Stack	Input	Output
$E'T'id	Id$	F → id
$E'T'	$	
$E'	$	T' → ∈
$	$	E' → ∈

Table 3.1

Previous Years' Question

Consider the following grammar:

$$S \rightarrow FR.$$

$$R \rightarrow *S \mid \in$$

$$F \rightarrow id$$

In the predictive parser table, M, of the grammar the entries M[S,id] and M[R, $] respectively

a) $\{S \rightarrow FR\}$ and $\{R \rightarrow \in\}$

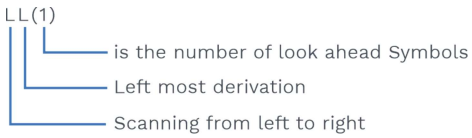
b) $\{S \rightarrow FR\}$ and $\{\ \}$

c) $\{S \rightarrow FR\}$ and $\{R \rightarrow *S\}$

d) $\{S \rightarrow id\}$ and $\{R \rightarrow \in\}$

Sol: a) **[GATE: CS 2006]**

Checking for LL(1) grammar:



- The LL(1) grammar is unambiguous, non left recursive and left factored.
- To determine whether a grammar is LL(1) or not:

1) If $A \rightarrow \alpha_1 \mid \alpha_2 \mid \alpha_3$ then

$$\left.\begin{array}{l} FIRST(\alpha_1) \cap FIRST(\alpha_2) = \phi \\ FIRST(\alpha_2) \cap FIRST(\alpha_3) = \phi \\ FIRST(\alpha_1) \cap FIRST(\alpha_3) = \phi \end{array}\right| \text{Pairwise disjoint}$$

Then the grammar is LL(1)

2) If $A \rightarrow \alpha_1 \mid \alpha_2 \mid \in$ then

$$\left.\begin{array}{l} FIRST(\alpha_1) \cap FIRST(\alpha_2) = \phi \\ FIRST(\alpha_2) \cap FOLLOW(A) = \phi \\ FIRST(\alpha_1) \cap FOLLOW(A) = \phi \end{array}\right| \text{Pairwise disjoint}$$

Then the grammar is LL(1)

- Grammar is considered to be LL(1) if it's LL(1) the parsing table is free from multiple entries.
- A left recursive grammar can not be a LL(1) grammar.
- An ambiguous grammar can not be LL(1) grammar.
- If a grammar has a common prefix, it cannot be LL(1).

Rack Your Brain

Construct the LL(1) parsing table for the following grammar

$$S \rightarrow (L) \mid a$$
$$L \rightarrow SL'$$
$$L' \rightarrow \in \mid SL'$$

And parse the input string (a, a, a).

Rack Your Brain

Which of the following is true.

1) Every ambiguous grammar is not LL(1)

2) Every unambiguous grammar need not be LL(1)

3) Every LL(1) grammar is unambiguous

4) None

Previous Years' Question

The grammar $A \rightarrow AA \mid (A) \mid \in$ is not suitable for predictive parsing because the grammar is

a) Ambiguous
b) Left-recursive
c) Right-recursive
d) An operator-grammar

Sol: a) **[GATE: CS 2005]**

Shift reduce parser:

- The shift reduce parser attempts to construct a parse tree in the same way that bottom-up parsing does.
- In other words, the parse tree is built from leaf (bottom) to root (top).
- LR parser is a more general form of shift reduce parser.
- The shift reduce parser required a data structure known as a stack.
- A stack that stores and retrieves the production rule.
- An input buffer to store the input string.

Basic operations in shift reduce parsing:

There are four actions possible in shift reduce parser:
Shift: The top of the stack will be updated with the next input symbol.
Reduce: If the handle appears on top of the stack, then it reduced by using appropriate production rule is done i.e., right hand of production rule is popped out of the stack, and the left hand of a production rule is pushed on to the stack.
Accept:Parser will consider a parsing as a successful parsing if and only if the start symbol is present in the stack and the input buffer is empty.
Error: The ERR("error recovery routine") will be triggered by the parser if there is a syntax error, and during this the parser stops shift and reduces operations.

Example:

Consider the grammar

$$A \rightarrow A + A$$
$$A \rightarrow A * A$$
$$A \rightarrow id$$

and parse the input string $id_1 + id_2 + id_3$ using shift reduce parser and find total number of shifts and reduce operation.

Stack	Input	Action
$	$id_1+id_2+id_3\$$	Shift
$\$id_1$	$+id_2+id_3\$$	Reduce by A → id
$A	$+id_2+id_3\$$	Shift
$A+	$id_2+id_3\$$	Shift
$A+id_2$	$+id_3\$$	Reduce by A → id
$A+A	$+id_3\$$	Shift
$A+A+	$Id_3\$$	Shift
$A+A+id_3$	$	Reduce by A → id
$A+A+A	$	Reduce by A → A+A
$A+A	$	Reduce by A → A+A
$A	$	Accept

Table 3.3

Total shift operation = 5
Total reduce operation = 5

Rack Your Brain

Consider the grammar

$A \rightarrow A + A$
$A \rightarrow A * A$
$A \rightarrow id$

Perform shift reduce parsing for input string $id_1+id_2*id_3$ and find total number of shift and reduce operation.

Previous Years' Question

Which one of the following derivations is used by LR parsers?

a) Leftmost
b) Leftmost in reverse
c) Rightmost
d) Rightmost in reverse

Sol: d) **[GATE: CS 2019]**

LR parser:

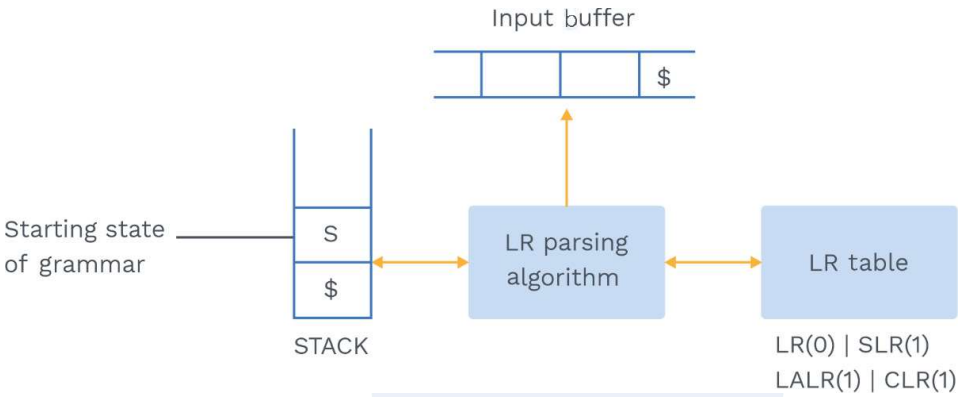
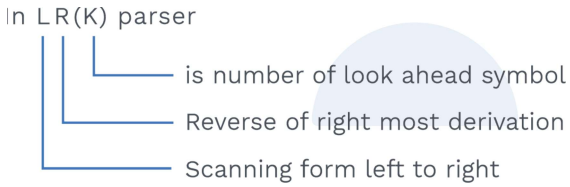


Fig. 3.6 Parser Diagram

- LR parser can be built for an unambiguous grammar.
- An operator precedence can be built for both ambiguous and unambiguous grammar.



- Bottom–up parser's average complexity is $O(n^3)$
- Handle pruning is a significant overhead for a bottom-up parser.
- Parsing algorithm is same for all LR parsers but parsing table may differ.

LR parsing algorithm:

If S is TOP of stack and a is look ahead symbol.

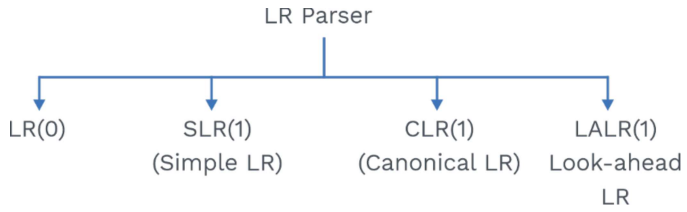
1) If action [S, a] = S_i (shift) then shift a and i and Increment input pointer.

2) If action [S, a] = r_j (reduce) and r_j is $\alpha \rightarrow \beta$, then pop $2 \times |\beta|$ from top of stack and replace by α.
S_{m-1} is the state below α, then push goto $[S_{m-1}, \alpha]$

3) If action [S, a] = ACCEPT then successful completion of parsing.

4) If action [S, a] = blank then error.

Types of LR parser:



LR(0) and SLR(1):

- LR(0) and SLR(1) work on smallest class of grammar.
- LR(0) and SLR(1) contain few number of states; hence, the table size is small.
- LR(0) and SLR(1) are simple and fast to construct.

CLR(1):

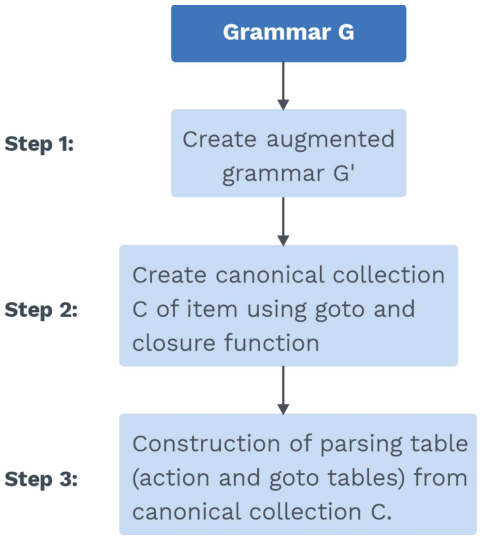
- CLR(1) works on complete set of LR(1) grammar.
- CLR(1) contain large number of states; hence, table size is large.
- CLR(1) is slow construction.

LALR(1) parser:

- LALR(1) works on intermediate size of grammar.
- Number of states are same as in SLR(1) and LR(0).

Constructing an LR parsing table:

- To generate the parsing table from the grammar, three steps are required.



- In the first step, an extra production rule is added to the original set of production to create an augmented grammar G. All LR parsers i.e., LR(0), SLR(1), CLR(1), LALR(1) follow the same procedure.
- In the second step of creating the canonical collection C of entities called items, the LR(0) item, while CLR(1) and LALR(1) use a more specialized entity called LR(1) item.
- LR(0) and SLR(1) parser has different closure and goto functions compared to CLR(1) and LALR(1).
- In step three, we convert the canonical collection of a set of items into the parsing table by applying certain Rules.

Augmented grammar:

If G is a grammar with the start symbol S, then the augmented grammar G' for G consists of all productions in G and an additional production $S' \rightarrow S$. Where S' is the start symbol in G'.

The main use of augmented grammar lies in the fact that the acceptance of input happens when the parser is about to reduce production $S' \rightarrow S$.

LR(0) item and LR(1) item:

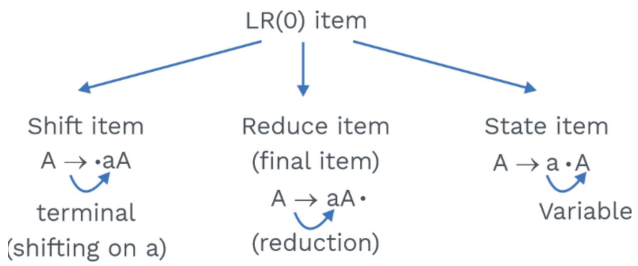
- An LR(0) item (or simply item) of a grammar G is production of G with a dot in some position of right-hand side.
- For a grammar consisting of a production $S \rightarrow ABC$, the LR(0) items are:

$$S \rightarrow \cdot ABC$$
$$S \rightarrow A \cdot BC$$
$$S \rightarrow AB \cdot C$$
$$S \rightarrow ABC \cdot$$

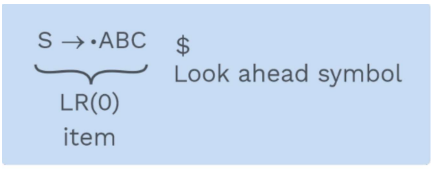
- For production $S \rightarrow \in$, there is only one LR(0) item.

$$S \rightarrow \cdot$$

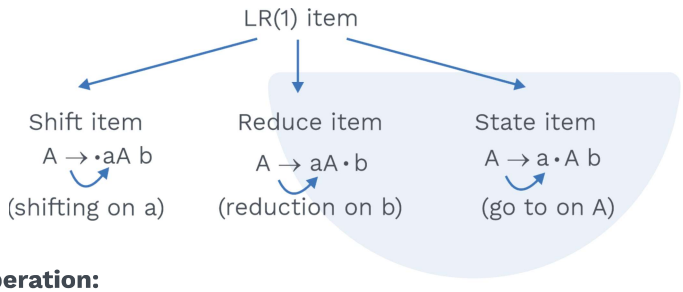
- Dot(.) can be used to show the portion of input that is already consumed. If the dot moves to the rightmost point of the production, we can use it to reduce that production.



- An LR(1) item of a grammar G is LR(0) item and look ahead symbol.
- For e.g.



LR(1) items



Closure operation:

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

Rule 1: Every element of I is added to the closure of I.

Rule 2: If $S \rightarrow A \cdot BC$, is in closure(I) and there exists a production $B \rightarrow b$, then add the item $B \rightarrow \cdot b$ to I, if it is not already in closure(I). Keep applying this rule until there are no more elements added.

Example:

$$A' \rightarrow A$$

$$A \rightarrow aA \mid b$$

$\text{Closure}(A' \rightarrow A) = A' \rightarrow A$ ('A' is non-terminal so add production of A)

$A \rightarrow \cdot aA$

$\cdot b$

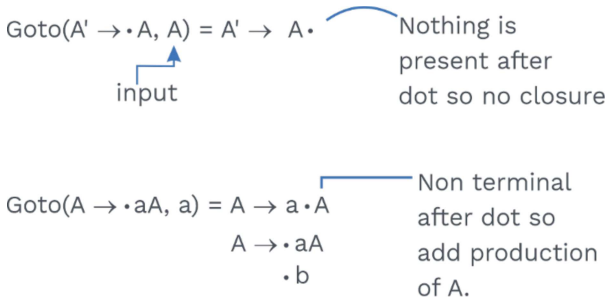
Goto operation:

Goto (I, X), I is the set of items which the goto (I, X) needs to be computed.

1) Add I by moving dot after X.

2) Apply closure to step 1.

Example:



LR(0) Parsing table construction algorithm:

Let C = $\{I_0, I_1, ...I_n\}$ the states of the parser are 0, 1, ...n state i being constructed from I_i the parsing action for state I are determined as follows.

1) If goto $(I_i, a) = I_J$, then set ACTION[i, a] = S_J (Shift entry)

2) If goto $(I_i, A) = I_J$ then set ACTION[i, A] = J (State entry)

3) If I_J contain $A \rightarrow \beta\cdot$ (reduced production) then action [i, all entries] = R_P (reduce entry) where P is a production number $(P : A \rightarrow \beta\cdot)$

4) If $[A' \rightarrow A\cdot]$ is in I_i then set action [i, $] to accept.

Note

If any conflicts are generated due to the above rules, we say grammar is not LR(0). The algorithm fails to produce a valid parser in this case. If a state contains either SR or RR conflicts, then that state is called inadequate state.

LR(0) conflicts :

There are two types of conflict in LR(0)

1) Shift-reduce conflict (SR conflict)

- If both shift and reduced productions are in same state.
  For e.g.

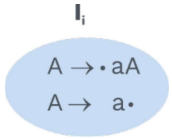


Fig. 3.7 SR Conflict

2) Reduce-reduce conflict (RR conflict):
If two reduce production (reduced items) are present in the same state.

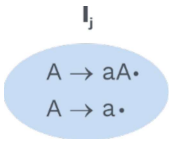


Fig. 3.8 Caption RR Conflict

Note

If any state has LR(0) conflict, then it is not LR(0) grammar.

SLR(1) parsing table construction:

- SLR(1) parsing table construction is same as LR(0) except for reduction entries.
- If $[A \rightarrow \beta \cdot]$ is in I_i, then find FOLLOW(A) and for every a in FOLLOW(A), set action [i, a] = R_P where P is a production number.
- Number of shift entries is same in LR(0) and SLR(1).
- Size of SLR(1) parser table is same as size of LR(0) parser table.
- Number of state entry is also same in LR(0) and SLR(1).
- Number of states is same in LR(0) and SLR(1).
- Every LR(0) grammar is SLR(1) but every SLR(1) need not be LR(0).

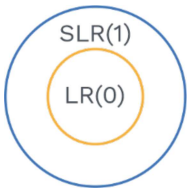


Fig. 3.9 Parsers

Conflict in SLR(1) parser:

1) Shift-reduce conflict (SR conflict): State I_i shift reduce conflict if FOLLOW $(B) \cap \{\alpha\} \neq \phi$

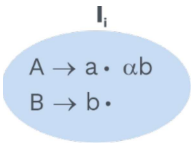


Fig. 3.10 SR Conflict

2) Reduce-reduce conflict:

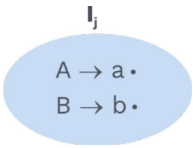


Fig. 3.11 Caption RR Conflict

State I_j is reduce-reduce conflict if FOLLOW (A) $\cap$ FOLLOW (B) $\neq \phi$

SOLVED EXAMPLES

Q.6 **Construct LR(0) and SLR(1) parsing table for the following grammar.**

$A \rightarrow aA$

$A \rightarrow b$

Sol: **Step1:** Construct augmented grammar

$A' \rightarrow A$
$A \rightarrow aA$
$A \rightarrow b$

Step2: Make LR(0) item and use closure and goto to construct a DFA.

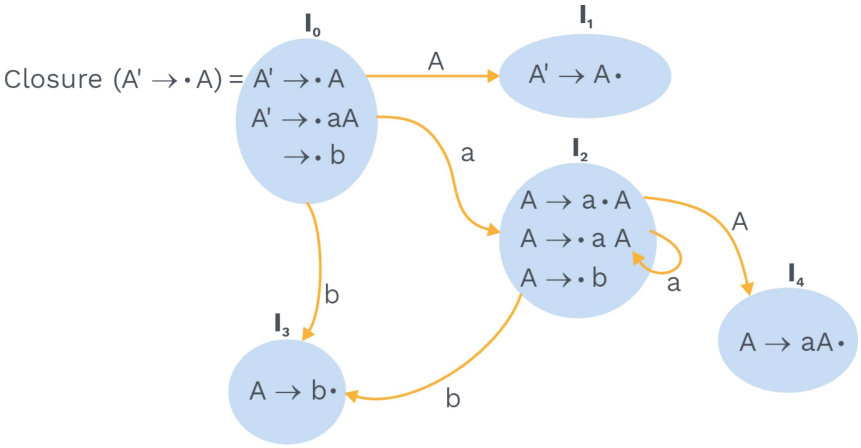


Fig. 3.12 Canonical form of LR Items

Step3: Construction of LR(0) parsing table

numbering for reduce :
$A \rightarrow aA$ ..... (1)
$A \rightarrow b$ ..... (2)

	Action			Goto
	a	b	$	A
I_0	s_2	s_3	—	1
I_1	—	—	accept	—
I_2	s_2	s_3	—	4
I_3	r_2	r_2	r_2	—
I_4	r_1	r_1	r_1	—

LR(0) / SLR(1) parsing table

As there is no conflict so, the given grammar is LR(0). (if the grammar is LR(0). It will definitely SLR(1))

Q.7 **Construct LR(0) and SLR(1) parsing table for the following grammar.**
$A \rightarrow a \mid ab \mid bB$
$B \rightarrow c$

Sol: **Step1:** Construct augmented grammar

$A' \rightarrow A$
$A \rightarrow a \mid ab \mid bB$
$B \rightarrow c$

Step2: Make LR(0) item and use closure and goto to construct a DFA.

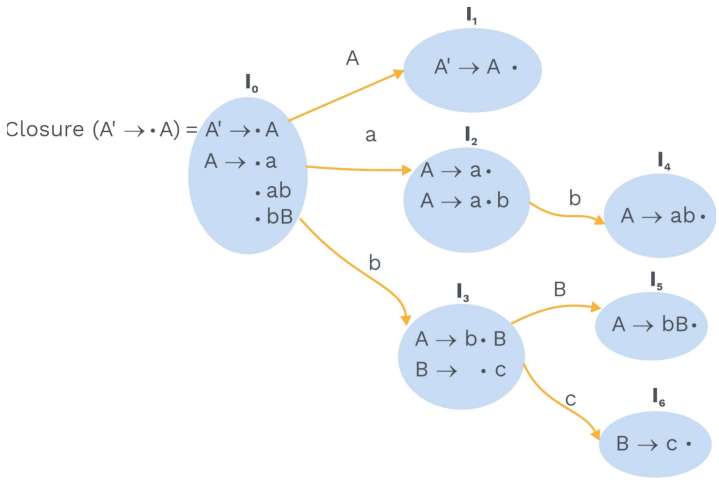


Fig. 3.13 Cononical form of LR Elements

Step 3: Construction of LR(0) parsing table

	Action				Goto	
	a	b	c	$	A	B
I_0	s_2	s_3	—		1	—
I_1	—	—	—	accept	—	—
I_2	r_1	s_4/r_1	r_1	r_1	—	—
I_3	—	—	s_6	—	—	5
I_4	r_2	r_2	r_2	r_2	—	—
I_5	r_3	r_3	r_3	r_3	—	—
I_6	r_4	r_4	r_4	r_4	—	—

numbering for reduce:
$A \rightarrow a$ .....(1)
$A \rightarrow ab$ .....(2)
$A \rightarrow bB$ .....(3)
$B \rightarrow c$ .....(4)

- As in the LR(0) parsing table, there is shift-reduce conflict so the grammar is not LR(0).
- State 2 is inadequate state (SR conflict).

SLR(1) Parsing table:
As there is no SR (shift reduce) or RR (reduce reduce) so the grammar is SLR(1).

	Action				Goto	
	a	b	c	$	A	B
I_0	s_2	s_3	—	—	1	—
I_1	—	—	—	accept	—	—
I_2	—	s_4	—	r_1	—	—
I_3	—	—	s_6	—	—	5
I_4	—	—	—	r_2	—	—
I_5	—	—	—	r_3	—	—
I_6	—	—	—	r_4	—	—

Q.8 **Construct LR(0) and SLR(1) parsing table for the following grammar**
A $\rightarrow$ AA | a | ϵ

Sol: **Step 1:** Construct augmented grammar

$$A' \rightarrow A$$

$$A \rightarrow A\,A$$

$A \rightarrow a$

$A \rightarrow \in$

Step 2: Make LR(0) item and use closure and goto to construct a DFA.

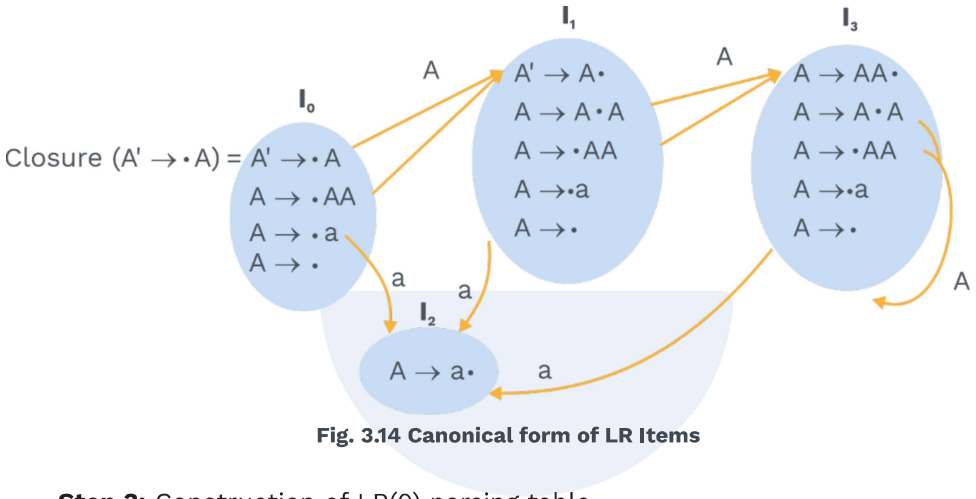


Fig. 3.14 Canonical form of LR Items

Step 3: Construction of LR(0) parsing table

	Action		Goto
	a	$	A
I_0	s_2/r_3	r_3	1
I_1	s_2/r_3	accept/r_3	3
I_2	r_2	r_2	—
I_3	$s_2/r_1/r_3$	r_1/r_3	—

numbering for reduce :
$A \rightarrow AA$.....(1)
$A \rightarrow a$.....(2)
$A \rightarrow \in$.....(3)

- There are both shift reduce and reduce conflict so the grammar is not LR(0).
- The parsing table is same for SLR(1). So, the grammar is not SLR(1) also.

Note

The given grammar is ambiguous grammar and an ambiguous grammar can never be LR(0) and SLR(1).

Q.9 **Construct LR(0) and SLR(1) parsing table for the following grammar**
$A \rightarrow B + A \mid B$
$B \rightarrow b$

Sol: **Step 1:** Construct augmented grammar

$$A' \rightarrow A$$

$$A \rightarrow B + A \mid B$$

$$B \rightarrow b$$

Step 2: Make LR(0) item and use closure and goto to construct a DFA.

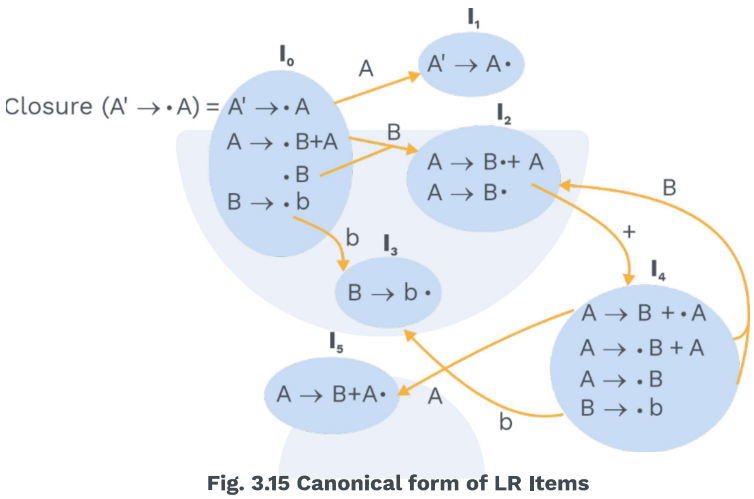


Fig. 3.15 Canonical form of LR Items

Step 3: Construction LR(0) parsing table

	Action			Goto	
	b	+	$	A	B
I_0	s_3	—	—	1	2
I_1	—	—	accept	—	—
I_2	r_2	s_4/r_2	r_2	—	—
I_3	r_3	r_3	r_3	—	—
I_4	s_3	—	—	5	2
I_5	r_1	r_1	r_1	—	—

numbering for reduce :
$A \rightarrow B+A$.....(1)
$A \rightarrow B$.....(2)
$B \rightarrow b$.....(3)

- As there is SR conflict in LR(0) paring table so, the grammar is not LR(0)
- In SLR(1) parsing table, the reduction $A \rightarrow B \cdot$ will go in FOLLOW (A) = $ so no conflict; hence the grammar is SLR(1).

Q.10 **Construct LR(0) and SLR(1) parsing table for the following grammar**
A → BB
B → cB | d

Sol: **Step 1:** Construct augmented grammar:

$$A' \rightarrow A$$

$$A \rightarrow BB$$

$$B \rightarrow cB$$

$$B \rightarrow d$$

Step 2: Make LR(0) item and use closure and goto to construct a DFA.

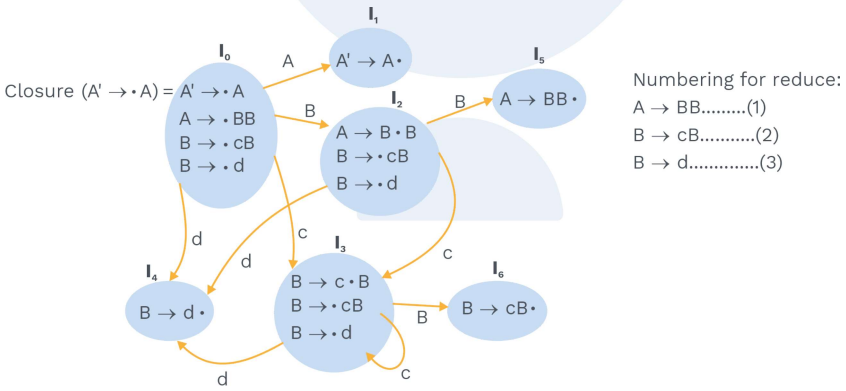


Fig. 3.16 Canonical form of LR Items

Step 3: Construct LR(0) parsing table.

	←Action→			←Goto→	
	c	d	$	A	B
I_0	s_3	s_4	—	1	2
I_1	—	—	accept	—	—
I_2	s_3	s_4	—	—	5
I_3	s_3	s_4	—	—	6
I_4	r_3	r_3	r_3	—	—
I_5	r_1	r_1	r_1	—	—
I_6	r_2	r_2	r_2	—	—

- In the above LR(0) parsing table there is no conflict so it is LR(0), if the grammar is LR(0), it is SLR(1) too.

CLR(1) Parsing table construction:

- Except reduce entries, remaining entries are same as SLR(1).
- If I_i contains $A \rightarrow a, \$$ then reduced entry (r_j) $A \rightarrow a$ in row i under the terminal given by lookhead i.e., ($).
- It takes LR(1) items. The reduce production entries are based on look ahead parsing table.

Rack Your Brain

Construct LR(0) and SLR(1) parsing table for

$S \rightarrow TuTv \mid WvWu$

$T \rightarrow \in$

$W \rightarrow \in$

LALR(1) parser:

LALR(1) parser constructed from CLR(1) parser.

- In CLR(1), if two states are having same production but contain different lookahead symbol, then those two states are combined into a single state in LALR(1).
- Every LALR(1) grammar is CLR(1), but every CLR(1) grammar need not LALR(1).
- In CLR(1) parser even if we don't have RR conflict, after merging, we might have RR conflict in LALR(1)

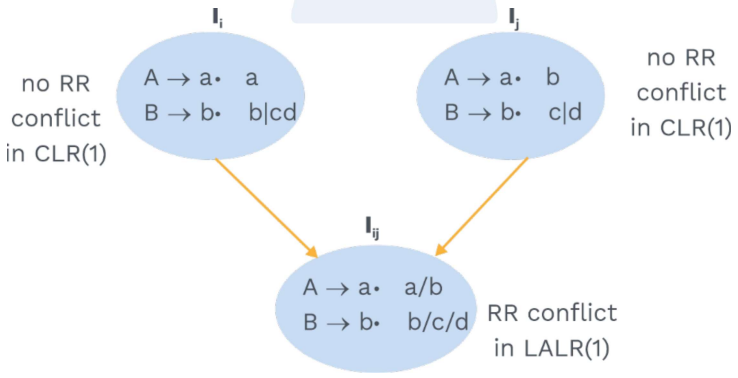


Fig. 3.17 Canonical form of LR Items

- In CLR(1) parser even if we don't have SR conflict, after merging, there will be no SR conflict in LALR(1).

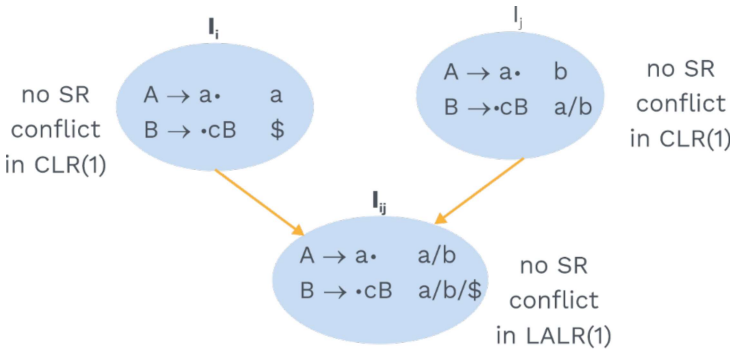


Fig. 3.18 Canonical form of LR(1) Items

- If the grammar is CLR(1) but not LALR(1) then there must be RR conflict.

Conflicts in CLR(1) and LALR(1) :

1) Shift-reduce conflict:

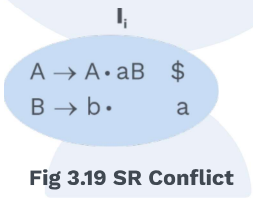


Fig 3.19 SR Conflict

State I_i is having SR conflict because both shifting and reduction on same lookahead symbol i.e., 'a'.

2) Reduce-reduce conflict:

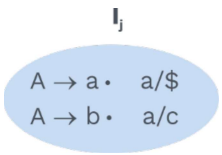


Fig. 3.20 RR Conflict

State I_j is reduce-reduce conflict because both reductions happen on same lookahead i.e., 'a'

SOLVED EXAMPLES

Q.11 **Construct CLR(1) and LALR(1) parsing tables for the following grammar**
A → a A
A → b

Sol: **Step 1:** construct augmented grammar

$A' \rightarrow A$
$A \rightarrow aA$
$A \rightarrow b$

Step2: Make LR(1) item and use closure and goto to construct a DFA.

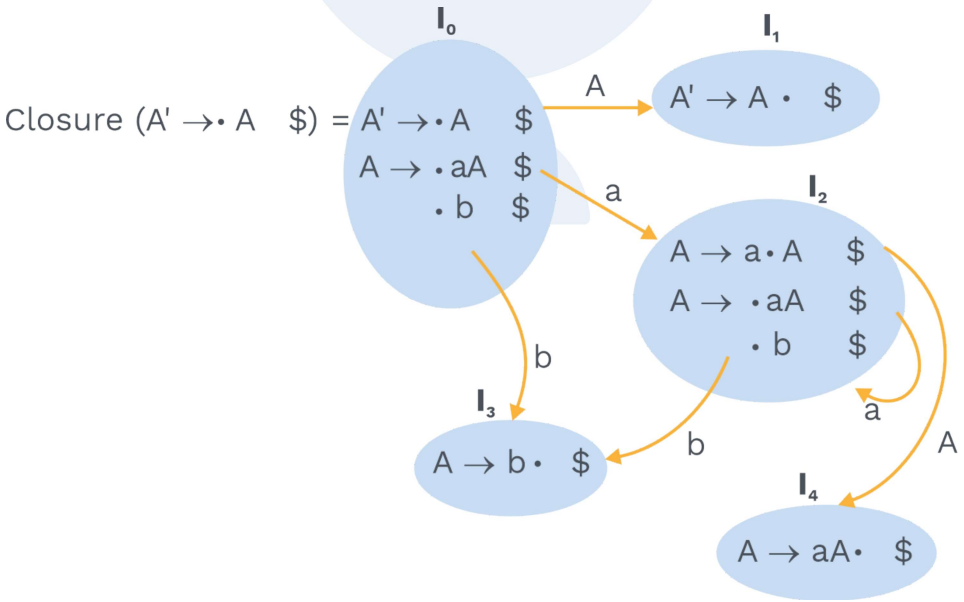


Fig. 3.21

Step 3: Construct CLR(1) parsing table.

	Action			Goto
	c	b	$	A
I_0	s_2	s_3	—	1
I_1	—	—	accept	—
I_2	s_2	s_3	—	4
I_3	—	—	r_2	—
I_4	—	—	r_1	—

numbering to reduce:
$A \rightarrow aA$.....(1)
$A \rightarrow b$.....(2)

- In the above CLR(1) parsing table, we don't have any conflict, so the grammar is CLR(1).
- Since there are no two states having the same production containing different look ahead. So it is already minimized, and it is LALR(1).
- As we see in the LR(0) example, the given grammar is also LR(0), and Every LR(0) grammar is SLR(1), CLR(1) and LALR(1).
- In the above example, we get the same number of states in LR(0), SLR(1), CLR(1) and LALR(1).

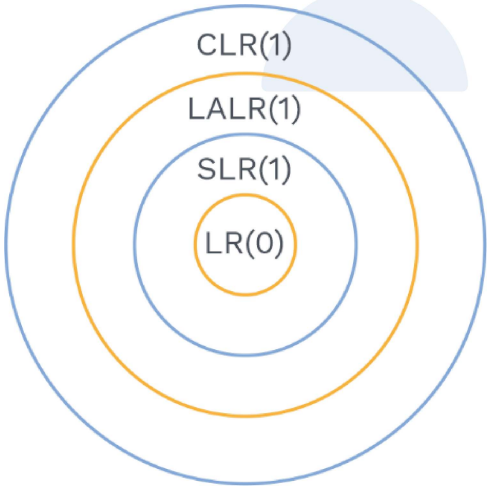


Fig. 3.22 Power of Parser

Q.12 **Construct CLR(1) and LALR(1) parsing table for the following grammar**
A → B + A | B
B → b

Sol: **Step 1:** Construct augmented grammar

$$A' \rightarrow A$$

$$A \rightarrow B + A$$
$$A \rightarrow B$$
$$B \rightarrow b$$

Step 2: Make LR(1) item and use closure and goto to construct a DFA.

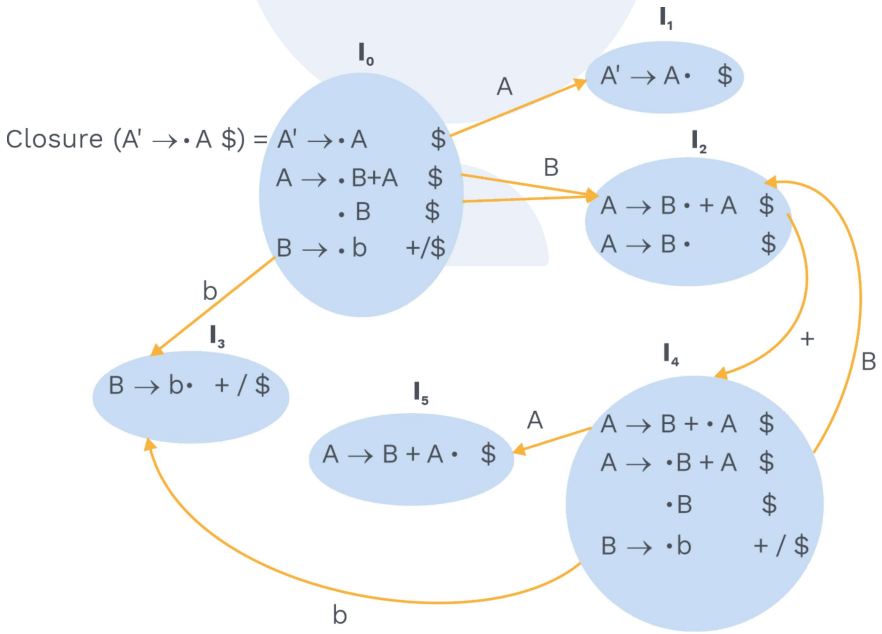


Fig. 3.23 Canonical form of LR(1) Items

Step 3: construction of CLR(1) parsing table

	Action			Goto	
	b	+	$	A	B
I_0	S_3	—	—	1	2
I_1	—	—	accept	—	—
I_2	—	S_4	r_2	—	—
I_3	—	r_3	r_3	—	—
I_4	—	—	—	5	2
I_5	—	—	r_1	—	—

numbering to reduce:
$A \rightarrow B+A$.....(1)
$A \rightarrow B$.....(2)
$B \rightarrow b$.....(3)

In the above CLR(1) parsing table, we don't have any conflict so the grammar is CLR(1) and it is minimized too so LALR(1) also (no conflict).

- The above grammar is not LR(0) as we have solved already, but it is SLR(1). And every SLR(1) grammar is also LALR(1) and CLR(1).

Q.13 **Construct augmented grammar**
$A \rightarrow AA \mid a \mid \in$

Sol: **Step 1:** construct augmented grammar.

$$A' \rightarrow \cdot A$$

$$A \rightarrow \cdot AA$$

$$A \rightarrow \cdot a$$

$$A \rightarrow \cdot \in$$

Step 2: Make LR(1) item and use closure and goto to construct a DFA.

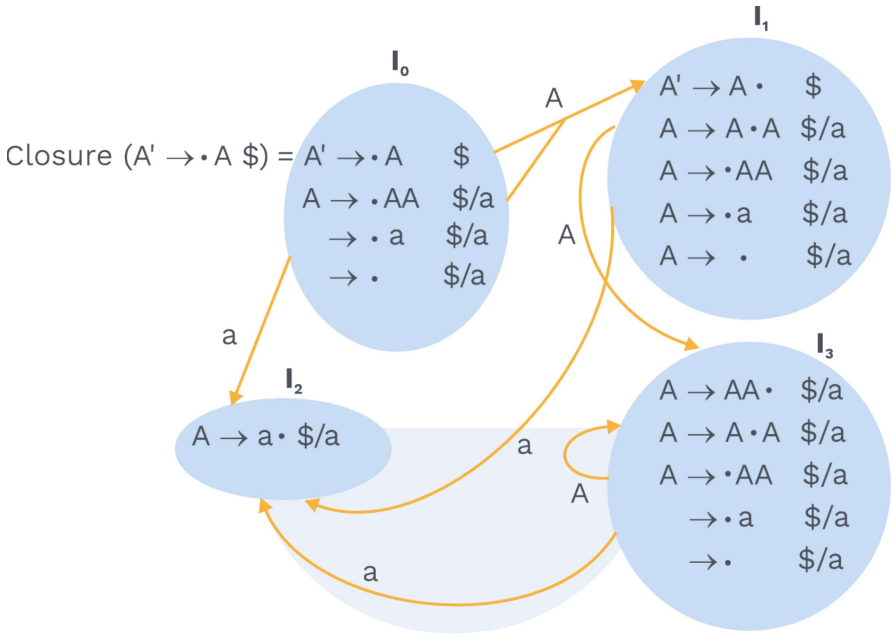


Fig. 3.24 Canonical form of LR(1) Items

Step 3:

	Action		Goto	
	a	$	A	Numbering for reduce:
I_0	s_2/r_3	r_3	1	A → AA.....(1)
I_1	s_2/r_3	accept/r_3	3	A → a.....(2)
I_2	r_2	r_2	–	A → .....(3)
I_3	$s_2/r_3/r_1$	r_3	–	

As in the above CLR(1) parsing table, there is a shift between reduce and reduce-reduce conflict, then it is not CLR(1).

- The number of states is also not further minimized so the parsing table remains the same in LALR(1).
- If the grammar is not CLR(1) then it is also not LALR(1).
- The given grammar is ambiguous, and an ambiguous grammar can never be LL(1), LR(0), SLR(1), CLR(1) and LALR(1).
- If grammar is not CLR(1) then it is not LALR(1) not SLR(1) not LR(0).

Q.14 **Construct CLR(1) and LALR(1) parsing table for the following grammar.**
A → BB
B → cB | d

Sol: **Step 1:** Construct augmented grammar

$$A' \rightarrow .A$$

$$A \rightarrow .BB$$

$$B \rightarrow .cB \mid .d$$

Step 2: Make LR(1) item and use closure and goto to construct a DFA.

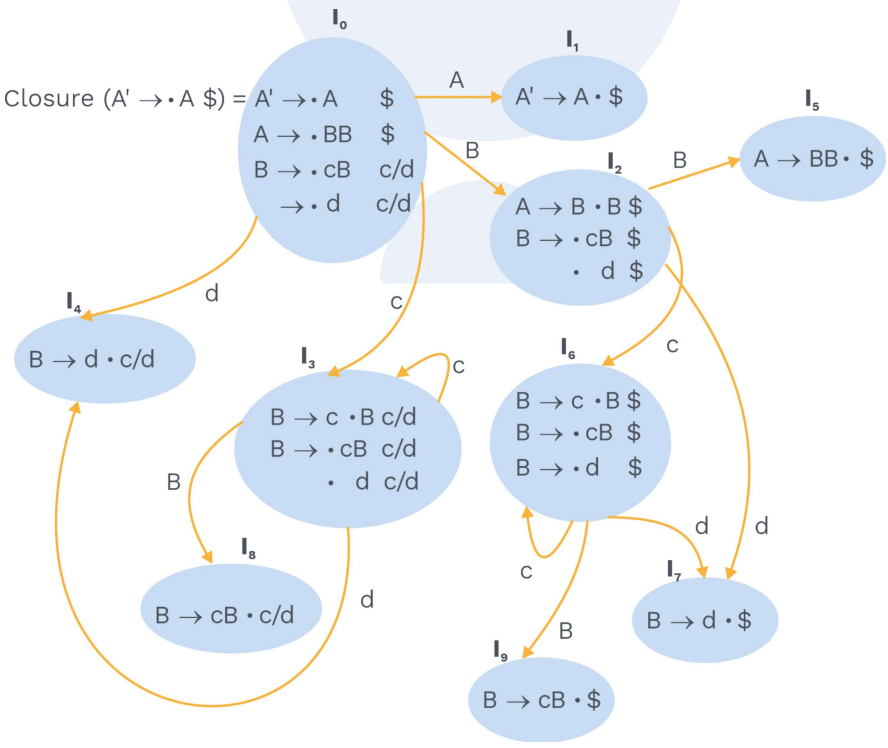


Fig. 3.25 Canonical form of LR(1) Items

	Action			Goto	
	c	d	$	A	B
I_0	s_3	s_4	accept	1	2
I_1	—	—	—	—	—
I_2	s_6	s_7	—	—	5
I_3	s_3	s_4	—	—	8
I_4	r_3	r_3	—	—	—
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	—	—

Numbering for reduce:
$A \rightarrow BB$.....(1)
$B \rightarrow cB$.....(2)
$B \rightarrow d$.....(3)

CLR(1) parsing table:

- In above CLR(1) parser there is no conflict, so it is CLR(1).
- Since state I_3, I_6. State I_4, I_7 and state I_8, I_9 having same production and different look ahead so we can combine in LALR(1).

$$I_3, I_6 \rightarrow I_{36}$$

$$I_4, I_7 \rightarrow I_{47}$$

$$I_8, I_9 \rightarrow I_{89}$$

	Action			Goto	
	c	d	$	A	B
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	—	—	—	—
I_5	—	—	—	—	—
I_{89}	r_2	r_2	r_2	—	—

- There is no conflict in LALR(1) parsing table, so it is also LALR(1).
- Number of states in LALR(1) parser is same as LR(0) and SLR(1) parser but less than CLR(1) parser.
  Updated Canonical collection of LALR(1).

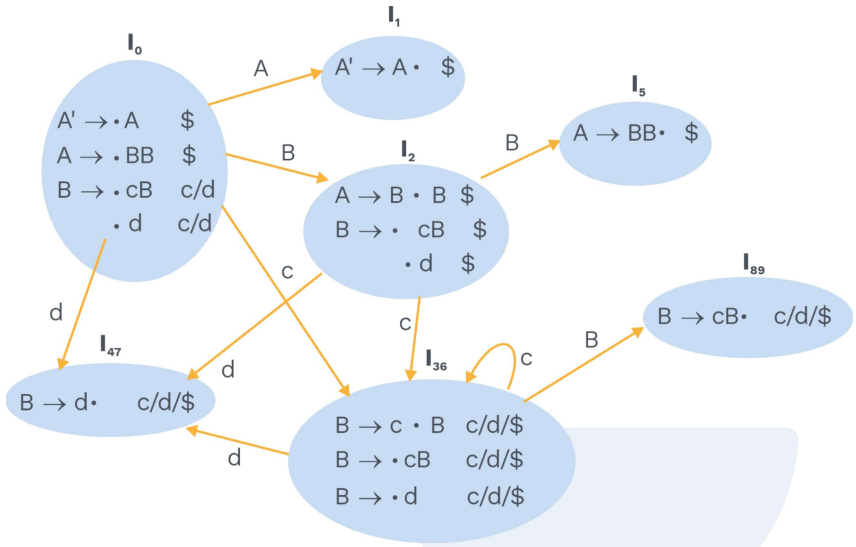


Fig. 3.26 Canonical form of LR(1) Items

Previous Years' Question

Consider the grammar

$$S \rightarrow (S) \mid a$$

Let the number of states in SLR(1), LR(1) and LALR(1) parsers for the grammar be n_1, n_2 and n_3 respectively. The following relationship holds good:

a) $n_1 < n_2 < n_3$ **b)** $n_1 = n_3 < n_2$

c) $n_1 = n_2 = n_3$ **d)** $n_1 \geq n_3 \geq n_2$

Sol: b) **[GATE: CS 2005]**

Operator precedence parsing:

- Operator precedence parser is a form of shift reduce parser, it is easy to implement.
- Operator grammar has the following properties:
  - **1)** A grammar does not contain $\in$ production.
  - **2)** A grammar does not contain two adjacent non-terminals.

  For e.g - $S \rightarrow AB$ is not allowed.

Example:
Consider the following expression

$$E \rightarrow EAE \mid id$$

$$A \rightarrow + \mid - \mid * \mid /$$

As the given grammar is not operator grammar, because the right side EAE has three consecutive non-terminals.

- However, if we substitute A with each of its productions we obtain the following operator grammar.

$$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid id$$

In operator precedence parsing, we use three disjoint precedence relations $\lessdot, \doteq$ and $\gtrdot$ between each pair of terminals.

- These precedence relations guide the selection of handles and have the following meaning.

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

- There are common ways of determining what precedence relation should hold between a pair of terminals. For e.g. if * has higher precedence than +, we make $+ \lessdot *$ and $* \gtrdot +$

Example:
Consider the following operator grammar.

$E \rightarrow E + E \mid E * E \mid id$, draw the operator precedence table according to the following rules:

- * has higher precedence than + and * is left associative.
- + has lower precedence than * and + is left associative.
- id has highest precedence and $ has the lowest precedence.

Previous Years' Question

Consider the grammar shown below:

$$S \rightarrow CC$$

$$C \rightarrow cC \mid d$$

The grammar is
a) LL(1)
b) SLR(1) but not LL(1)
c) LALR(1) but not SLR(1)
d) LR(1) but not LALR(1)
Sol: a) **[GATE: CS 2003]**

	id	+	*	$
id	—	·>	·>	·>
+	<·	·>	<·	·>
*	<·	·>	·>	·>
$	<·	<·	<·	—

Let a be topmost terminal symbol on stack, and b is the look ahead terminal symbol then.

1) If $a \lessdot b$ or $a \doteq b$, then push b / shift b onto the stack and increment input symbol.
2) If $a \cdot\!> b$, then pop stack until the top of stack terminal is related by $\lessdot$ to the terminal most recently popped.
3) Else error.

Operator precedence table:

Operator precedence parsing algorithm:

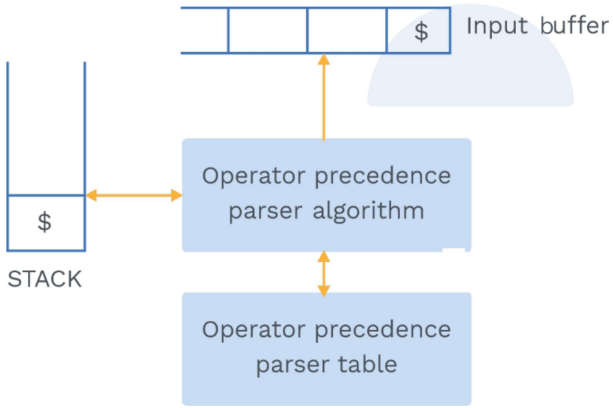


Fig. 3.27 Parser Diagram

Q. Consider the grammar $E \rightarrow E+E \mid E*E \mid id$ parser the input id+id*id using operator precedence parser.

Stack	Input	Action
$	id+id*id$	Shift
$id	+id*id$	Reduce E → id
$E	+id*id$	Shift
$E+	id*id$	Shift
$E+id	*id$	Reduce E → id
$E+E	*id$	Shift
$E+E*	id$	Shift
$E+E*id	$	Reduce E → id
$E+E*E	$	Reduce E → E*E
$E+E	$	Reduce E → E+E
$E	$	Accept

Table 3.4

Classification of bottom-up parser:

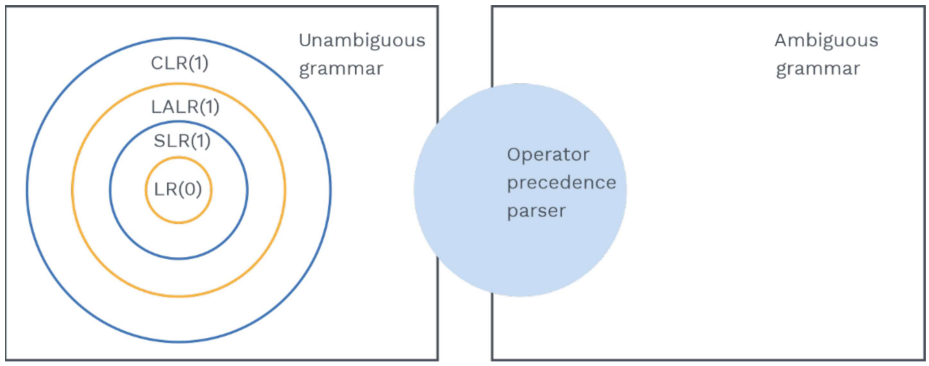


Fig. 3.28 Power of each Parser in the form of Sets

Previous Years' Question

Which of the following grammar rules violate the requirements of an operator grammar? P, Q, R are non-terminals and r, s, t are terminals.

i) $P \rightarrow QR$ **ii)** $P \rightarrow QsR$

iii) $P \rightarrow \in$ **iv)** $P \rightarrow QtRr$

a) (i) only **b)** (i) and (iii) only

c) (ii) and (iii) only **d)** (iii) and (iv) only

Sol: b) **[GATE: CS 2005]**

Previous Years' Question

Consider the following two sets of LR(1) items of LR(1) grammar.

$X \rightarrow c.X,\ c/d$ $X \rightarrow c.X,\ \$$

$X \rightarrow .cX,\ c/d$ $X \rightarrow .cX,\ \$$

$X \rightarrow .d,\ c/d$ $X \rightarrow .d,\ \$$

Which of the following statement related to margining of the two sts in the corresponding parser is/are FALSE?

1) Cannot be merged since look aheads are different.

2) Can be merged but will result in S-R conflict.

3) Can be merged but will result in R-R conflict.

4) Cannot be merged since goto on c will lead to two different sets.

a) 1 only **b)** 2 only

c) 1 and 4 only **d)** 1, 2, 3 and 4

Sol: d) **[GATE: CS 2013]**

Chapter Summary

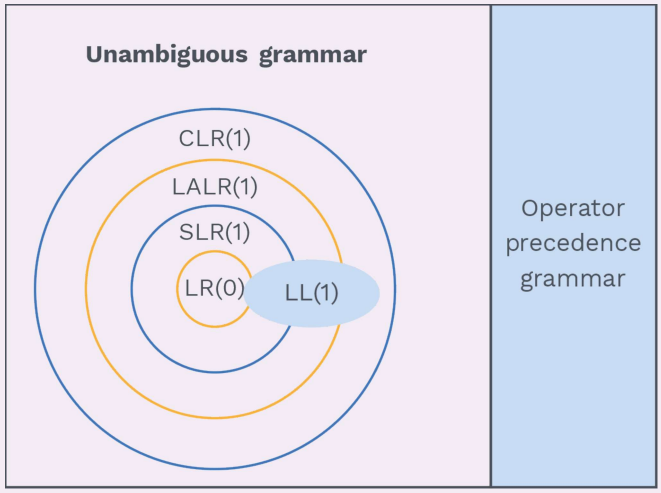


Fig. 3.29 Power of each Parser in a Bigger Picture

- Every LL(1) grammar is CLR(1).
- Every LR(0) grammar is CLR(1).
- If the grammar is not CLR(1) then it is not LALR(1), not SLR(1) and not LR(0).
- If the grammar is ambiguous it will not be LL(1), LR(0), SLR(1), LALR(1) and CLR(1) grammar.
- Number of shift entries are same in LR(0), SLR(1) and LALR(1) but less than or equal to CLR(1).
- Number of reduce entries in LR(0) $\geq$ SLR(1) $\geq$ LALR(1) $\geq$ CLR(1)
- Number of state entries (goto) are also same for LR(0), SLR(1) and LALR(1) but less than or equal to CLR(1).
- Parsing Table size is same for LR(0), SLR(1) and LALR(1) but less than or equal to CLR(1).
- Expressive power of parsers
  LL(1) < LR(0) < SLR(1) <LALR(1) < CLR(1)
- Number of states in LR parsers

$$n(\text{LR}(0)) = n(\text{SLR}(1)) = n(\text{LALR}(1)) \leq n(\text{CLR}(1))$$

- Size of LR parsing table = Number of states × (Number of symbols + 1)