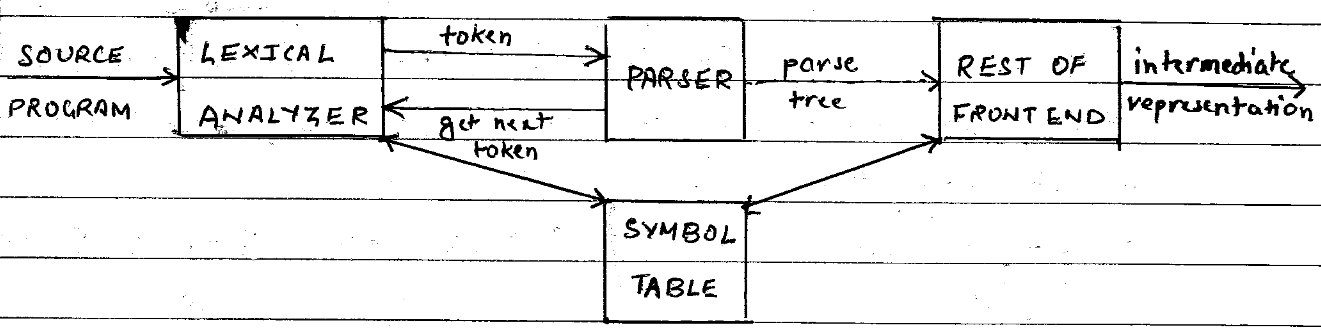
UNIT-2

SYNTAX ANALYSIS & SYNTAX DIRECTED TRANSLATION

SYNTAX ANALYSIS -

① Role of a parser -



② CFGs (CONTEXT-FREE GRAMMERS) -

A CFG consists of terminals, non-terminals, a start symbol and productions.

(token) Terminals → basic symbols from which strings are ~~f~~ formed

Non-terminals → Syntactic variables that denote sets of strings

Start symbol → A non-terminal starting symbol.

Productions → Specify the manner in which the terminals and non-terminals can be combined to form strings.

Parse Tree - Graphical Represntation for a derivation.

Leftmost derivations - Only the leftmost nonterminal in any sentential form is replaced at each step.

Rightmost derivations - Rightmost nonterminal is replaced at each step.

Ambiguity - A grammer that produces more than one parse tree for some sentence is said to be ambiguous.

③ Top Down Parsing -

A top down parsing algorithm parses an input string of tokens by tracing out the steps in a leftmost derivation. Such an algorithm is called top-down because the implied traversal of the parse tree is a pre-order traversal and, thus, occurs from the root of the leaves.

Top-down parsers come in two forms -

(1) Backtracking Parsers

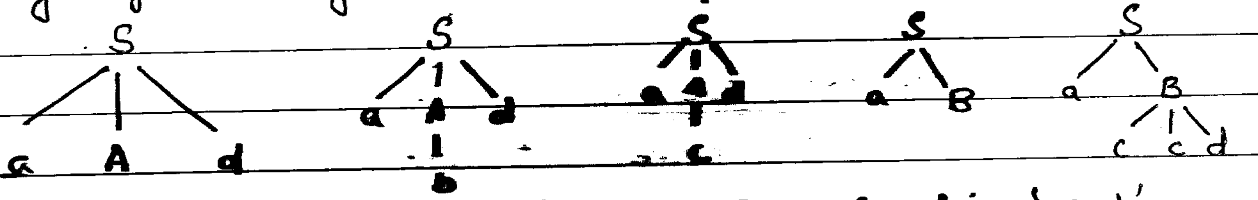
(2) Predictive Parsers

~~(3) Brute force Parsers~~.

④ <u>Brute Force Approach</u> -

Top-down parsing with full backup is a 'brute force' method of parsing. In using full backup we are willing to attempt to create a syntax tree by following branches until the correct set of terminals is reached.

Eg - Grammer is given as, S → aAd | aB   A → b | c   B → ccd | ddc



<u>Trace of a brute force top-down parse for string 'accd'.</u>

→ Due to left-recursive grammar, it causes an infinite loop for the Top-down parser. Left recursion can be removed using <u>left factoring</u> which is given as,

$A \rightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | B_1 | B_2 | \dots | B_m$

which is converted as,

$A \rightarrow \beta_1 | \beta_2 | \dots | \beta_m | \beta_1 A' | \dots | \beta_m A'$

$A' \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n | \alpha_1 A' | \dots | \alpha_n A'$

→ Error recovery is very poor, ~~and~~ very inefficient & time consuming

⑤ <u>Recursive-descent Parsing</u>

It is a top-down method of syntax analysis in which we execute a set of recursive procedures to process the input. A procedure is associated with each nonterminal of a grammer

~~Eg - S → aAd | aB   A → b | c   B → ccd | ddc~~

~~→ procedure ES;~~   Eg - if-stmt → if (exp) statement |
if (exp) statement else stateme

~~begin~~

~~match (a);~~   procedure if Stmt;

if ~~tok~~

```
procedure if Stmt;
begin
    match (if);
    match (();
    exp;
```

```
match (1);
statement;
if token = else then
    match (else);
      statement;
  endif;
end ifstmt;
```

```
procedure match (expectedtoken);
begin
  if token = expectedtoken then
    getToken;
else
   error;
endif;
end match;
```

(5) Each function returns a value of true or false depending on whether or not it recognizes a substring which is an expansion of that nonterminal.

(6) Predictive Parsing -

It is a special form of recursive-descent parsing, in which current input token unambiguously determines the production to be applied at each step.

→ No backtracking, efficient and uses LL (1) grammer.

→ We have to eliminate the left recursion which is not enough for predictive parsing.

Two types -

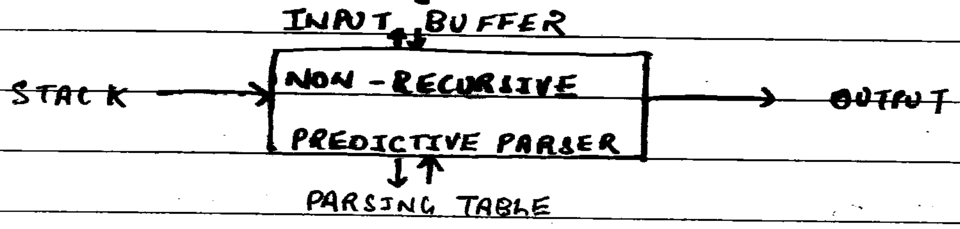
(1) Recursive Predictive Parsing -

Each non-terminal corresponds to a procedure.

Eg- $A \rightarrow aBb \mid bAB$

```
procedure A {
 case of the current token {
 'a' :- match the current token with a, and move to next token.
      - call 'B'
      - match the current token with b, and move to the next token.
 'b' :- match the current token with b, and move to the next token
      - call 'A'
      - call 'B' }  }
```

(2) <u>Non-Recursive Predictive Parsing</u> (also known as LL(1) parser) -
It is a table driven parser. It looks up the production to be applied in a parsing table.



Eg - $S \to aBa$, $B \to bB \mid \epsilon$

	a	b	\$
S	$S \to aBa$		
B	$B \to \epsilon$	$B \to bB$	

← LL(1) parsing table.

Sol -

Stack	Input	Output
\$S	abba\$	$S \to aBa$
\$aBa	abba\$	
\$aB	bba\$	$B \to bB$
\$aBb	bba\$	
\$aB	ba\$	$B \to bB$
\$aBb	ba\$	
\$aB	a\$	$B \to \Lambda$
\$a	a\$	
\$	\$	ACCEPT

➡ <u>Constructing LL(1) Parsing Tables</u> -
Two functions are used that is FIRST ~~and~~ FOLLOW.

→ FIRST(α) is a set of the terminal symbols which occur as first symbols in strings derived from ~~[illegible]~~ non-terminal α.

→ FOLLOW(α) is a set of terminals which occur immediately after the non-terminal α in the strings derived from the starting symbol.

LL(1)

Input scanned from left to right ← L

left-most derivation ← L

(1) → One input symbol ~~used to~~ as a lookahead symbol to determine parser action.

LL(1) Grammer -

- → A grammer whose parsing table has no multiple entries
- → A left recursive grammer can not be a LL(1) grammer
- → A grammer is not ~~f~~ left factored, it cannot be a LL(1) grammer
- → An ambiguous grammer cannot be a LL(1) grammer.

Computing FIRST for any string X -

(1) If X is terminal, then FIRST(X) is {X}.

(2) If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X)

(3) If X is nonterminal and $X \rightarrow Y_1 Y_2 \dots Y_k$ is a production then,

(i) place a in FIRST(X), if for some i, a is in FIRST(Y_i) and $Y_1 Y_2, \dots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$ [ϵ is in all of FIRST(Y_1)... FIRST(Y_{i-1})]

(ii) ~~If~~ place ϵ in FIRST(X), if $Y_1 . Y_2 \dots Y_K \overset{*}{\Rightarrow} \epsilon$ that means ϵ is in all of FIRST(Y_1)... FIRST(Y_k).

Computing FOLLOW for non-terminals -

(1) Place $\$$ in FOLLOW(S), where S is the start symbol and $\$$ is the input right endmarker

(2) If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B).

(3) If there is a productions $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) = {ϵ}, then everything in FOLLOW(A) is in FOLLOW(B).

Algorithm for constructing LL(1) parsing table -

For each production rule $A \rightarrow \alpha$ of a grammer G -

- → For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to M[A, a]
- → If ϵ is in FIRST(α) then for each terminal a in FOLLOW(A), add $A \rightarrow \alpha$ to M[A, a]
- → If ϵ is in FIRST(α) and $\$$ in FOLLOW(A), then add $A \rightarrow \alpha$ to M[A, $\$$]

**** All other undefined entries of the parsing table are error entries.

Eg - $E \rightarrow TE'$, $E' \rightarrow +TE' | \epsilon$, $T \rightarrow FT'$, $T' \rightarrow *FT' | \epsilon$, $F \rightarrow (E) | id$

Sol - FIRST(E) = { (, id }

FIRST(E') = { +, ε }

FIRST(T) = { (, id }

FIRST(T') = { *, ε }

FIRST(F) = { (, id }

FOLLOW(E) = { \$,) }

FOLLOW(E') = { \$,) }

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

FOLLOW(T') = { +, \$,) }

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

	id	+	*	(	)	\$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→ε	E'→ε
T	T→FT'			T→FT'		
T'		T'→ε	T'→*FT'		T'→ε	T'→ε
F	F→id			F→(E)		

⑦ Bottom up Parsing - (Shift-Reduce parsing)

It attempts to construct a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

A "handle" of a string is a substring that matches the right side of a production.

If a grammer is unambiguous, then every right-sentential form of the grammer has exactly one handle.

The approach to reduce the string by a step in the reverse of rightmost derivation is known as handle pruning.

→ Configuration of shift-reduce parser on input $id_1 * id_2$ -

Grammer is given as, $E \to E+T \mid T$, $T \to T*F \mid F$, $F \to id \mid (E)$

STACK	INPUT	ACTION
$\$$	$id_1 * id_2 \$$	shift
$\$ id_1$	$* id_2 \$$	reduce by $F \to id$
$\$ F$	$* id_2 \$$	reduce by $T \to F$
$\$ T$	$* id_2 \$$	shift
$\$ T *$	$id_2 \$$	shift
$\$ T * id_2$	$\$$	reduce by $F \to id$
$\$ T * F$	$\$$	reduce by $T \to T * F$
$\$ T$	$\$$	reduce by $E \to T$
$\$ E$	$\$$	accept

Four possible actions a shift-reduce parser can make -

(1) shift (2) reduce (3) Accept (4) Error

Conflicts during shift reduce parsing -

(1) Shift / Reduce conflict → whether make a shift operation or reduce operation

(2) Reduce / Reduce conflict → Parser cannot decide which of several reductions to make.

→ If a shift reduce parser can not be used for a grammer, that grammer is called as Non-LR(k) grammer -

→ An ambiguious grammer cannot be a LR grammer.

Types of shift reducing parsers -

(1) Operator-Precedence Parser

(2) LR parsers - Covers wide range of grammers -

(i) SLR (Simple LR) (ii) CLR (Canonical or most general LR)

(iii) LALR (Look Ahead or Intermediate LR).

⑧ Operator Precedence Parsing -

There are two rules ~~in this grammer~~ of operator grammer

(1) No production right side is ϵ.

(2) No production has two adjacent non terminals.

In operator precedence parsing, we define three disjoint precedence relation, <, = ~~and~~ > between certain pair of terminals.

Precedence relation table is given as,

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

Using precedence relations to find handles -

(1) Scan the input from left to right until first ·> is encountered

(2) Scan backwards over = until <· is encountered

(3) The handle is a string between <· and ·>.

Eg - id+id*id ⇒ $<·id·>+<·id·>*<·id·>$

Advantages -

(1) Simple to implement

Disadvantages -

(1) The operator like minus has two different precedence (unary & binary). Hence it is hard to handle tokens like minus sign

(2) This kind of parsing is applicable to only small class of program.

Application - SNOBOL

⑨ LR parsers - (LR(k) parsing)

L → for left-to-right scanning of the input

R → for constructing rightmost derivation in reverse.

k → Number of input symbols of lookahead.

It is non-~~bac~~ backtracking shift reduce parsing.

Properties of LR parser -

(1) It can be constructed to recognize most of the programming languages for which CFG can be written

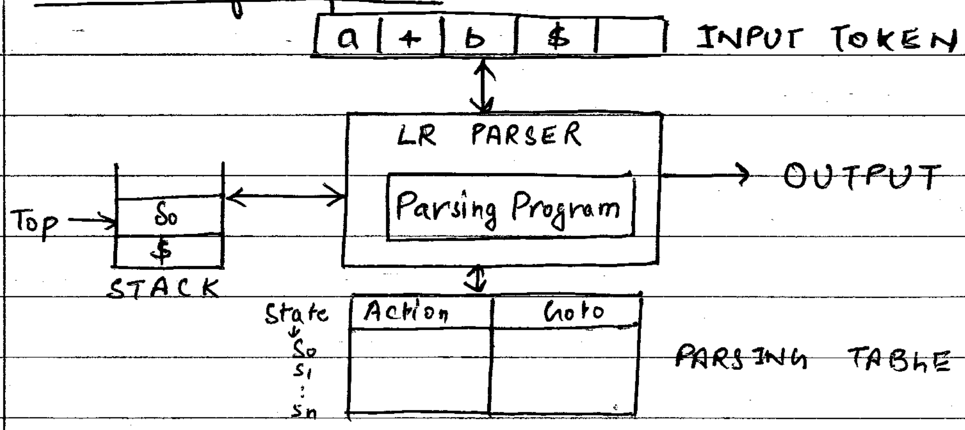
(2) The class of grammer that can be parsed by LR parser is a superset of class of grammers that can be parsed using predictive parsers.

(3) LR parsers works using non backtracking shift reduce technique yet it is efficient one

→ It detects syntactical errors very efficiently.

→ It is table driven parser

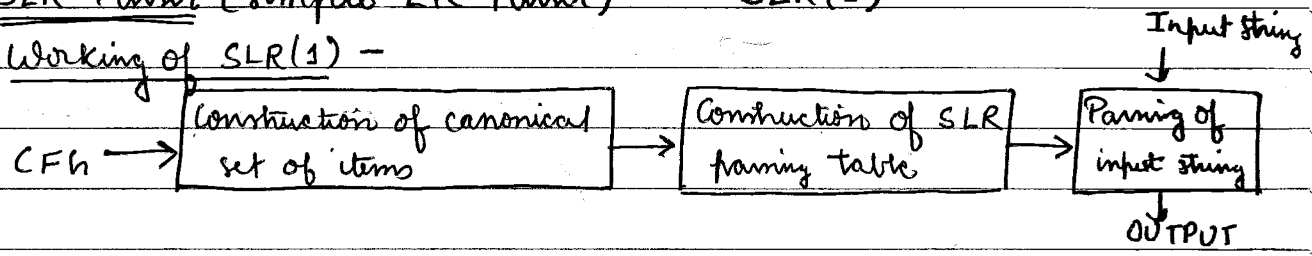
Structure of LR parsers -



***** ⇒ $$\boxed{SLR(1) \le LALR(1) \le \overset{CLR}{LR(1)}}$$

→ SLR Parser (Simple LR Parser) - SLR(1)

→ Working of SLR(1) -


CFG
Construction of canonical set of items
Construction of SLR parsing table
Input string
Parsing of input string
OUTPUT

→ Definition of LR(0) items and related items -

(1) The LR(0) item for grammer G is production rule in which symbol • is inserted at some point in RHS of the rule.
The production $S \to \epsilon$ generates only one item $S \to \bullet$

(2) Augmented Grammer - Grammer G have S as start symbol then augmented grammer G' is $\boxed{S' \to S + \text{grammer } G}$.

$$\begin{array}{ccc} \text{LHS} & & \text{RHS} \\ S & \rightarrow & \overset{}{\underset{\uparrow}{A}}BC \end{array}$$

leftmost end of RHS

Augmented grammer indicates the acceptance of input. That is when parser is about to reduce $S' \rightarrow S$ it reaches to acceptance state.

(3) Kernel items - Collection of items $S' \rightarrow \cdot S$ and all the items whose dots are not at the leftmost end of RHS of the r

Non-Kernel items - collection of all the items in which $\cdot$ are at the left~~most~~ end of RHS of the rule.

(4) Function closure and goto - Required to create collection of canonical set of items.

(5) Viable prefix - Set of prefixes in the right sentential form of productions $A \rightarrow \alpha$. This set can appear on the stack during shift/reduce action.

→ Closure Operation -

For CFG, if I → set of items then function closure(I) can be constructed using following rules -

(1) Consider I is a set of canonical items & initially every item I is added to closure (I).

(2) If rule $A \rightarrow \alpha \cdot B\beta$ is a rule in closure (I) and there is another rule for B such as $B \rightarrow \gamma$ then

closure (I): $A \rightarrow \alpha \cdot B\beta$

$B \rightarrow \cdot \gamma$

This rule is applied until no more new items can be added to closure (I).

→ Goto Operation -

If there is a production $A \rightarrow \alpha \cdot B\beta$ then

goto $(A \rightarrow \alpha \cdot B\beta, B) = A \rightarrow \alpha B \cdot \beta$

That means simply shifting of $\cdot$ one position ahead over the grammer symbol.

SLR short –

$G \xrightarrow{\text{augment } G} G'$ then find → closure (I_i) and goto (I_i, a)

second value ← First shift ← Parse table ← find FOLLOW
action actions

↳ ~~one~~ third accept → goto transitions filled → parse the input string
action ~~action~~

→ Construction of canonical collection of set of item –

(1) For the grammer G initially add $S' \rightarrow \cdot S$ in the set of item C.

(2) For each set of items I_i in C and for each grammer symbol X, add closure (I_i, X). This process should be repeated by applying goto (I_i, X) for each X in I_i such that goto (I_i, X) is not empty and not in C.

→ Construction of SLR parsing table –

Input – An Augmented grammer G'

Output – SLR parsing table.

Algorithm:

(1) Initially construct set of items $C = \{I_0, I_1, \dots I_n\}$ where C is a collection of set of LR(0) items for the input grammer G'.

(2) The parsing actions are based on each item I_i. The action are as given below –

(a) If $A \rightarrow \alpha \cdot a \beta$ is in I_i and goto $(I_i, a) = I_j$ then set action [i, a] as "shift j". Note → a is terminal.

(b) If ~~the~~ $A \rightarrow \alpha \cdot$ is in I_i then set action [i, a] to "reduce $A \rightarrow \alpha$" for all symbols a, where $a \in$ FOLLOW(A). Note → A is not augumented grammer S'.

(c) If $S' \rightarrow S$ is in I_i then the entry in the action table action [i, \$] = "accept".

(3) The goto part of the SLR table can be filled as –
The goto transitions for state i is considered for non terminals only

(4) All the entries not defined by rule 2 and 3 are considered to be error

→ Parsing the input using parsing table –

Input – The input string w that is to be parsed using parsing table

Output – Parse w if $w \in L(G)$ using bottom up. If $w \notin L(G)$ then report syntactical error.

Algorithm -

(1) Initially push 0 as inctial state onto the stack and place the input string with $ as end marker on the input tape.

(2) If S is the state on the top of the stack and a is the symbol from input buffer pointed by a lookahead pointer then

If action [S,a] is equal to -

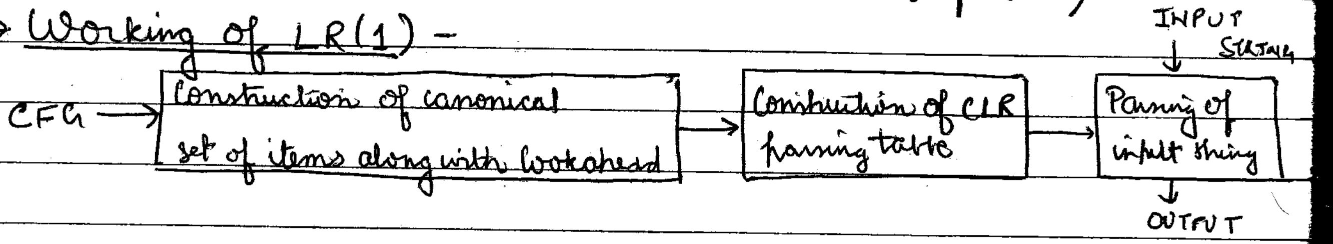
(a) shift j → then push a, then push j onto the stack. Advance the input lookahead pointer.

(b) reduce A→β → then pops 2*|β| symbols. If i is on the top of the stack then push A, then push goto[i,A] onto the stack.

(c) accept → then halt the parsing process. It indicates successful parsing.

→ CLR Parser (Canonical LR Parser or LR(1) parser) -

→ Working of LR(1) -


INPUT STRING
CFG
Construction of canonical set of items along with lookahead
Construction of CLR parsing table
Parsing of input string
OUTPUT

→ Construction of canonical set of items along with the lookahead -

(1) For the grammer G initially add $S' \to \cdot S$ in the set of item C

(2) For each set of items I_i in C and for each grammer symbol X, add closure (I_i, X). This process should be repeated by applying goto $(I_i \cdot X)$ for each X in I_i such that goto (I_i, X) is not empty and not in C.

(3) The closure function can be computed as follows -

For each item $A \to \alpha . X\beta$ and rule $X \to \gamma$ and $b \in$ FIRST such that $X \to .\gamma$ and b is not in I then add $X \to .\gamma, b$ to I.

(4) Similarly the goto function can be computed as -

For each item $[A \to \alpha . X\beta, a]$ is in I and rule $[A \to \alpha X . \beta, a]$ is not in goto items then add $[A \to \alpha X . \beta, a]$ to goto items.

→ **Construction of canonical LR parsing table -**

Input - An augmented grammer G'.

Output - The canonical LR parsing table.

Algorithm -

(1) Initially construct set of items $C = \{I_0, I_1, \ldots, I_n\}$ where C is a collection of set of LR(1) items for the input grammer G'.

(2) The parsing actions are based on each item I_i. The action are as given below -

(a) If $[A \to \alpha . a\beta, b]$ is in I_i and goto $(I_i, a) = I_j$ then set action [i, a] as "shift j". Note → a is terminal.

(b) If $[A \to \alpha . , a]$ is in I_i then set action [i, a] to "reduce $A \to \alpha$". Here A should not be S'.

(c) If $S' \to S . , \$$ is in I_i then set action [i, \$] = "accept".

(3) The goto part of the LR table can be filled as -

The goto transitions for state i is considered for non terminals only.

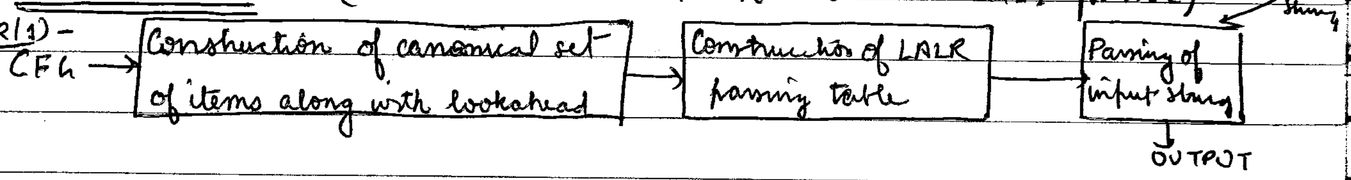
(4) All the entries not defined by rule 2 & 3 are considered to be "error".

→ **Parsing the input -**

Algorithm is same as SLR(1) parsing input algorithm.

→ **LALR Parser - (lookahead LR Parser or LALR(1) parser) -**

Working of LALR(1) -


Input string
CFG
Construction of canonical set of items along with lookahead
Construction of LALR parsing table
Parsing of input string
OUTPUT

→ **Construction set of LR(1) items with the lookahead -**

Same as CLR (LR(1)) parser. But only difference is that - In construction of LR(1) items for LR parser, we have differed the two states if the second component is different but in this case we will merge the two states by merging of first and second components from both the states.

→ Construction of LALR parsing table -

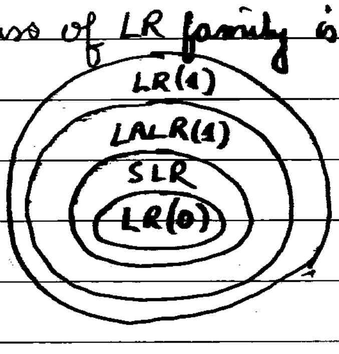
Same as CLR parsing table algorithm. But there is another step is included between step 1 and step 2 of CLR parsing table algorithm that is -

Merge the two states I_i and I_j if the first component (i.e. the production rules with dots) are matching and create a new state replacing one of the older state such as $I_{ij} = I_i \cup I_j$.

→ Comparison of LR parsers -

	SLR PARSER	LALR PARSER	CLR PARSER
(1)	SLR parser is smallest in size	LALR and SLR have same in size	CLR parser is largest in size
(2)	Easiest method based on FOLLOW function	Applicable to wider class than SLR	Most powerful than SLR and LALR
(3)	Enforces less syntactic features than that of LR parsers	Most of the syntactic features of a language are enforced in LALR	Enforces less syntactic features than that of LR parsers.
(4)	Error detection is not immediate	Error detection is not immediate	Immediate error detection is done
(5)	Requires less time and space complexity	Time and space complexity is more but efficient methods exist for constructing LALR parser directly	Time and space complexity is more

→ Graphical representation for the class of LR family is given below -

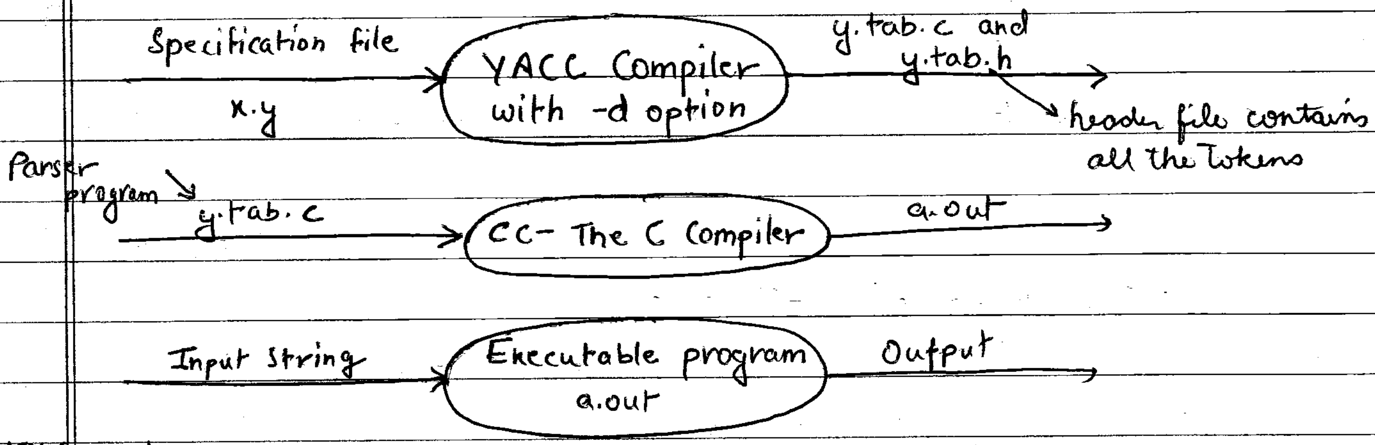


(10) Parser Generation - (YACC - Automatic Parser Generator)

YACC stands for Yet Another Compiler Compiler which is basically the utility availabel for UNIX.

Basically YACC is LALR parser generator. The YACC can report conflicts or ambiguities (if at all) in the form of error messages.

LEX and YACC work together to analyze the program syntactically.



UNIX Commands -

yacc x.y → Generate a parser ~~go~~ program (y.tab.c) using YACC specification file.

yacc -d x.y → two files generator i.e. y.tab.c and y.tab.h

YACC specification file contains the CFG and using the production rules of CFG the parsing of the input string can be done by y.tab.c

Extension to YACC program is .y

→ YACC specification -

It has three sections which can be written as -

```
%{
   /* declaration section */
%}
%%
   /* Translation rule section */
%%
   /* Required C functions */
```

(1) Declaration part - Ordinary C and grammer token declarations.

(2) Translation Rule section - All production rule of CFG

Eg :- S → Ab / Ba , A → a , B → b can be written as

```
S : Ab          // rule 1      action 1
  | Ba          //             action 2
  ;
A : a           // rule 2      action 3
  ;
B : b           // rule 3      action 4
  ;
```

(3) C function section - Consist of one main function in which the routine yyparse() will be called. Also it consists of required C functions

Compiling and running of LEX and YACC program -

```
# lex calci.l          //create lex.yy.c
# yacc -d calci.y      //create y.tab.c and y.tab.h
# cc y.tab.c lex.yy.c -ll -ly -lm
      //compile both lex.yy.c and y.tab.c by linking various library files
# ./a.out              //will run executable file of program
```

→ Shift/reduce conflict can be resolved by YACC with the help of precedence rules. If there is same precedence then YACC checks for associativity. For left associativity shift action will be performed and for right associativity reduce action will be performed

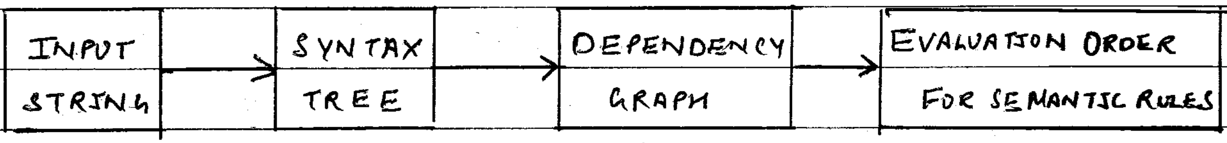
→ Rule Section - $$ is used as attributed value at LHS of the grammer

Eg → E + E ⇒ $1 + $2 , $2 = +

→ Subroutine Section - yyparse() is called which in turn calls yylex when it requires tokens. The routine yyerror is used to print the error message when an error is occurred in parsing of input.

SYNTAX DIRECTED ~~DEFINITIONS~~ TRANSLATION -

① Syntax Directed Definition (SDD) -



SYNTAX DIRECTED TRANSLATION

SDD is a kind of abstract specification which is used while doing the static analysis of the language. It means an augmented CFG is generated.

A parse tree containing the values of the attributes at each node is called an annotated or decorated ~~the~~ parse tree.

Definition - Syntax directed definition is a generalization of CFG in which each grammer production $X \rightarrow \alpha$ is associated with ~~a~~ a set of semantic rules of the form $a := f(b_1, b_2, \dots, b_K)$, where a is an attribute obtained from the function f.

Attribute - It can be a string, a number, a type, a memory location or anything else.

Consider $X \rightarrow \alpha$ be a CFG and $a := f(b_1, b_2, \dots, b_k)$ where a is the attribute. There are two types of attribute -

(1) Synthesized attribute -

The attribute 'a' is called synthesized attribute of X and $b_1, b_2, \dots, b_K$ are attributes belonging to the production symbols.

The value of synthesized attribute at a node is computed from the values of attributes at the children of that node in the parse tree.

(2) Inherited attribute -

The attribute 'a' is called inherited attribute of one of the grammer symbol on the right side of production (i.e. α) and $b_1, b_2, \dots, b_K$ are belonging to either X or α.

The inherited attributes can be computed from the values of the attributes at the siblings and parent of that node

Syntax directed definition is given as,

CFG — S → EN
E → E + T
E → E − T
E → T
T → T * F
T → T / F
T → F
F → (E)
F → digit
N → ;

PRODUCTION RULE	SEMANTIC ACTIONS
S → EN	Print (E.val)
$E \to E_1 + T$	$E.val := E_1.val + T.val$
$E \to E_1 - T$	$E.val := E_1.val - T.val$
E → T	E.val := T.val
$T \to T_1 * F$	$T.val := T_1.val \times F.val$
$T \to T_1 / F$	$T.val := T_1.val / F.val$
T → F	T.val := F.val
F → (E)	F.val := E.val
F → digit	F.val := digit.lexval
N → ;	Can be ignored by lexical analyzer (terminating symbol)

The token digit has synthesized attribute lexval whose value can be obtained from lexical analyzer.

In SDD, terminals have synthesized attributes only. No definition of terminal. The SDD that uses only synthesized attribute is called S-attributed definition.

To compute S-attributed definition -

(1) Write SDD

(2) Annotated parse tree is generated & attribute value are computed in bottom up manner.

(3) The value of obtained at root node is the final output.

To compute inherited attributes -

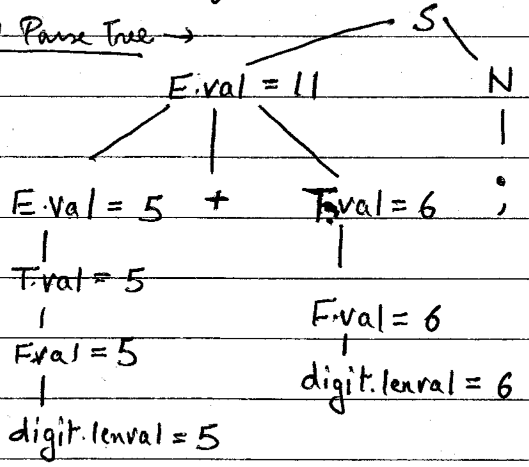
(1) Write SDD

(2) Annotate the parse tree with inherited attributes by processing in top down fashion.

Example of compilation of S-attributed definition -

Annotated Parse Tree →

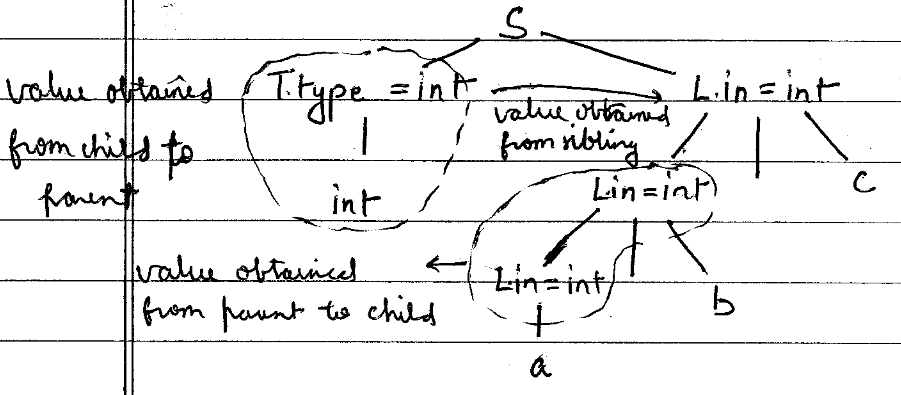
Expression String : 5+6;



Example of computation of inherited attributes -

Annotated Parse Tree →

String : int a,b,c

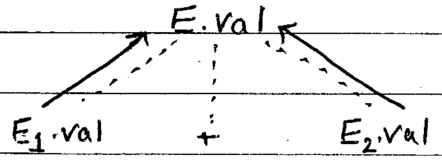

S
value obtained from child to parent
T.type = int
int
value obtained from sibling
L.in = int
L.in = int
c
value obtained from parent to child
L.in = int
b
a

PRODUCTION RULE	SEMANTIC ACTIONS
S → TL	L.in := T.type
T → int	T.type = integer
T → float	T.type := float
T → char	T.type := char
T → double	T.type := double
L → L_1, id	L_1.in := L.in
L → id	Enter.type (id.entry, L.in)

→ **Dependency Graph** -

The directed graph that represents the interdependencies between synthesized and inherited attributes at nodes in the parse tree is called dependency graph.

Eg:- $E \rightarrow E_1 + E_2$



solid arrows → dependency

dotted lines → parse tree.

→ Evaluation order –

The topological sort of the dependency graph decided the evaluation order in a parse tree. In deciding evaluation order the semantic rules in the syntax directed definitions are used. Thus the translation is decided by SDD. Therefore, precise definition of SDD is requi

(2) Construction of Syntax trees –

Syntax tree is an abstract representation of the language construct

→ For Expression –

Following functions are used in syntax tree for expression

(1) mknode (op, left, right)

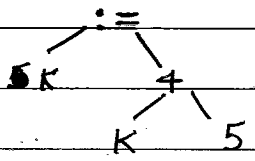
(2) mkleaf (id, entry)

(3) mkleaf (num, val)

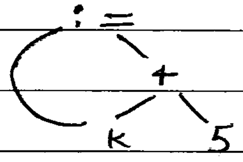
→ Direct Acyclic Graph for expression – (DAG)

DAG is directed graph drawn by identifying common subexpressions

→ Eg – Expression ⇒ k := k + 5



SYNTAX TREE



DAG

Sequence of operation –

p_1 = mkleaf (id, k)

p_2 = mkleaf (num, 5)

p_3 = mknode (+, p_1, p_2)

p_4 = mkleaf (~~num~~ id, k)

p_5 = mknode (:=, p_4, p_3)

Sequence of operation –

p_1 = mkleaf (id, k)

p_2 = mkleaf (num, 5)

p_3 = mknode (+, p_1, p_2)

p_4 = mknode (:=, p_1, p_3)

(3) Bottom-Up Evaluation of S-Attribute Definitions –

→ Synthesized Attributes on the parse stack –

(1) A translator for S-attributed definition is implemented using LR parser generator.

(2) A bottom up method is used to parse the input string.
(3) A parser stack is used to hold the values of synthesized attribute
(4) After reduction top is decremented as top-r+1 for r symbols
(5) If the symbol has no attribute then the corresponding entry in the value array will be kept undefined.

Eg - X → ABC

PRODUCTION RULE	SEMANTIC ACTION
X → ABC	X.x = f(A.a, B.b, C.c)

Before Reduction

	STATE	VALUE
	A	A.a
	B	B.b
TOP →	C	C.c

⟹

After Reduction

STATE	VALUE	
X	X.x	← TOP

PARSER STACK

→ <u>Parsing the input</u> -

Input string	State	Value	Production rule used
⋮	⋮	⋮	⋮

~~(4) L-attribute Definitions - (evaluated in depth first order).~~
~~The SDD can be defined as the L-attributed for the production rule A → $X_1 X_2 \ldots X_n$ where the inherited attribute X_k is such that $1 \leq k \leq n$. The production A → $X_1 X_2 \ldots X_n$ is such that~~
~~(1) It depends upon the attributes of the symbol $X_1, X_2, \ldots X_{j-1}$ to left of X~~
~~(2) It also depends upon the inherited attribute~~

(4) <u>L-attribute definitions</u> - (evaluated in depth first order)

A SDD is L-attributed if each inherited attribute of X_j, $1 \leq j \leq n$ on the right side of the production A → $X_1 X_2 \ldots X_n$, depends only on -
(1) the attributes of the symbols $X_1, X_2, \ldots X_{j-1}$ to left of X_j in the productio
(2) the inherited attributes of A.

***** Every S- attributed definition is L-attributed

→ Translation scheme -

The process of execution of code fragment semantic actions from the SDD is called syntax directed translation. Thus the execution of SDD can be done by syntax directed translation scheme

A translation scheme generates the output by executing the semantic actions in an ordered manner.

While designing the translation scheme —

We have to follow one restriction that is for every semantic action if it refers to some attribute then that attribute value must be computed before that attribute gets referred.

⑤ Top-down Translation -

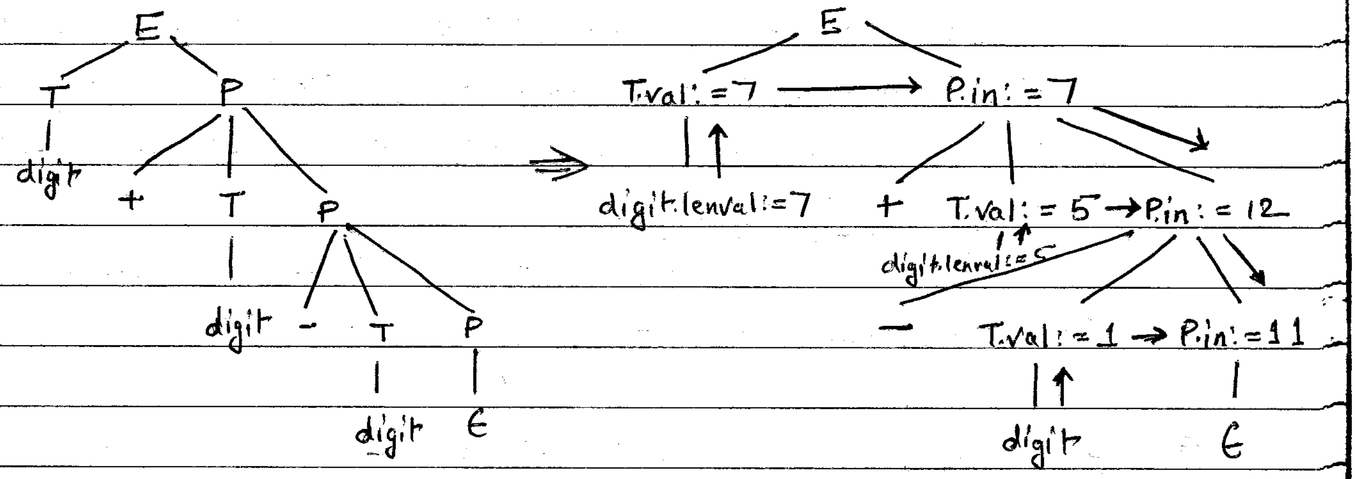
TRANSLATION SCHEME -

PRODUCTION	SEMANTIC ACTIONS
$E \rightarrow E_1 + T$	$\{E.val := E_1.val + T.val\}$
$E \rightarrow E_1 - T$	$\{E.val := E_1.val - T.val\}$
$E \rightarrow T$	$\{E.val := T.val\}$
$T \rightarrow (E)$	$\{T.val := E.val\}$
$T \rightarrow digit$	$\{T.val := digit.lexval\}$

Removing left recursion & rewrite the translation scheme for non left-recursive grammer

PRODUCTION	SEMANTIC RULE
$E \rightarrow T$	$\{P.in := T.val\}$
P	$\{E.val := P.s\}$
$P \rightarrow +T$	$\{P_1.in := P.in + T.val\}$
P_1	$\{P.s := P_1.s\}$
$P \rightarrow -T$	$\{P_1.in := P.in - T.val\}$
P_1	$\{P.s := P_1.s\}$
$P \rightarrow \epsilon$	$\{P.s := P.in\}$
$T \rightarrow ($	$\{T.val := E.val\}$
E	
$)$	
digit	$\{T.val := digit.lexval\}$

Annotated parse tree can be drawn as,



TOP DOWN TRANSLATION

In the parse tree the top down translation takes place. The blue lines show the parse tree whereas the black arrow lines shows the way of computing values of expression.

**** We can construct syntax tree for the translation scheme using mknode, ~~mkleaf~~ (op, left, right), mkleaf (id, entry) & mkleaf (num, val)

⑥ Bottom up Evaluation of Inherited Attributes -

The Bottom up parser reduces the right side of the production X → ABC by removing C, B and A from the parser stack.

Eg -

PRODUCTION RULE	CODE FRAGMENT
S → T List;	
T → int	value[top] := int
T → float	value[top] := float
List → List, id	Enter_type (value[top], value[top-3])
List → id	Enter_type (value[top], value[top-1])

⑦ Recursive Evaluation -

Recursive function that evaluate attributes as they traverse a parse tree can be constructed from a SDD using a generalization of the techniques for predictive translation.

Such functions allow us to implement SDD that cannot be implemented simultaneously with parsing.

In a translation specified by this definition, the children of a node for one production need to be visited from left to right, while the children of a node for the other productions need to be visited from right to left.

The functions need not depend on the order in which the parse tree nodes are created.

(8) **Analysis of Syntax directed definition -**

→ **Strongly non circular SDD -**

A SDD is said to be strongly noncircular if for each nonterminal A we can find a partial order RA on the attributes of A such that for each production p with left side A and nonterminals $A_1, A_2, \ldots, A_n$ occurring on the right side.

(1) $D_p[RA_1, RA_2, \ldots, RA_n]$ is acyclic, and

(2) if there is an edge from attribute A.b to A.c in $Dp[RA_1, RA_2, \ldots, RA_n]$ then RA orders A.b before A.c.

→ **Circularity Test -**

A SDD is said to be circular if the dependency graph for some parse tree has a cycle.

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

1. If X is terminal, then FIRST(X) is $\{X\}$.

2. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

3. If X is nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), . . . , FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \overset{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$) and so on.

Now, we can compute FIRST for any string $X_1 X_2 \cdots X_n$ as follows. Add to FIRST($X_1 X_2 \cdots X_n$) all the non-ϵ symbols of FIRST(X_1). Also add the non-ϵ symbols of FIRST(X_2) if ϵ is in FIRST(X_1), the non-ϵ symbols of FIRST(X_3) if ϵ is in both FIRST(X_1) and FIRST(X_2), and so on. Finally, add ϵ to FIRST($X_1 X_2 \cdots X_n$) if, for all i, FIRST(X_i) contains ϵ.

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol and \$ is the input right endmarker.

2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except for ϵ is placed in FOLLOW(B).

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ (i.e., $\beta \overset{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

➡ **Example 4.11 :** *Construct the LR(1) parsing table for the following grammar –*

(1) $S \to CC$
(2) $C \to aC$
(3) $C \to d$

Solution : First we will construct the set of LR(1) items -

I_0:
$S' \to \bullet S, \$$
$S \to \bullet CC, \$$
$C \to \bullet aC$, a/d
$C \to \bullet d$, a/d

I_1: goto (I_0,S)
$S' \to S \bullet, \$$

I_2: goto (I_0,C)
$S \to C \bullet C, \$$
$C \to \bullet aC, \$$
$C \to \bullet d, \$$

I_3: goto (I_0,a)
$C \to a \bullet C$, a/d
$C \to \bullet aC$, a/d
$C \to \bullet d$, a/d

I_4: goto (I_0,d)
$C \to d \bullet$, a/d

I_5: goto (I_2,C)
$S \to CC \bullet, \$$

I_6: goto (I_2,a)
$C' \to a \bullet C, \$$
$C \to \bullet aC, \$$
$C \to \bullet d, \$$

I_7: goto (I_2,d)
$C \to d \bullet, \$$

I_8: goto (I_3,C)
$C \to aC \bullet$, a/d

I_9: goto (I_6,C)
$C \to aC \bullet, \$$

We will initially add $S' \to \bullet S, \$$ as the first rule in I_0. Now match

$S' \to \bullet S, \$$ with

$[A \to \alpha \bullet X\beta, a]$

Hence $S' \to \bullet S, \$$

$A \to \alpha \bullet X\beta, a$

$A = S', \alpha = \varepsilon, X = S, \beta = \varepsilon, a = \$$

If there is a production $X \to \gamma$, b then add $X \to \bullet \gamma, b$

$\therefore S \to \bullet CC,$ $b \in \text{FIRST}(\beta a)$

$b \in \text{FIRST}(\varepsilon\ \$)$ as $\varepsilon\$ = \$$

$b \in \text{FIRST}(\$)$

$b = \{\$\}$

$\therefore S \to \bullet CC, \$$ will be added in I_0

The LR(1) parsing table as follows -

	action			goto	
	a	d	$	S	C
0	S3	S4		1	2
1			Accept		
2	S6	S7			5
3	S3	S4			8
4	r3	r3			
5			r1		
6	S6	S7			9
7			r3		
8	r2	r2			
9			r2		

The remaining blank entries in the table are considered as syntactical error.

Parsing the input using LR(1) parsing table

Using above parsing table we can parse the input string"aadd" as

Stack	Input buffer	action table	goto table	Parsing action
$0	aadd$	action(0,a)=S3		
$0a3	add$	action[3,a]=S3		Shift
$0a3a3	dd$	action[3,d]=S4		Shift
$0a3a3d4	d$	action[4,d]=r3	[3,C]=8	Reduce by C→ d
$0a3a3C8	d$	action[8,d]=r2	[3,C]=8	Reduce by C→ aC
$0a3C8	d$	action[8,d]=r2	[0,C]=2	Reduce by C→ aC
$0C2	d$	action[2,d]=S7		Shift
$0C2d7	$	action[7,$]=r3	[2,C]=5	Reduce by C→ d
$0C2C5	$	action[5,$]=r1	[0,S]=1	Reduce by S → CC
$0S1	$	accept		

Thus the given input string is successfully parsed using LR parser or canonical LR parser.

Example 4.13 :

S → CC

C→ aC

C→ d

Construct the parsing table for LALR(1) parser.

Solution : First the set LR(1) items can be constructed as follows with merged states.

I_0:
S'→ • S,$
S→ • CC,$
C→ • aC, a/d
C→ • d, a/d

I_1: goto (I_0,S)
S'→ S •,$

I_2: goto (I_0,C)
S→ C • C,$
C→ • aC,$
C→ • d,$

I_{36}: goto (I_0,a)
C→ a • C, a/d/$
C→ • aC, a/d/$
C→ • d, a/d/$

I_{47}: goto (I_0,d)
C→ d • , a/d/$

I_5: goto (I_2,C)
S→ CC •, $

I_{89}: goto (I_3,C)
C→ aC • a/d/$

Now consider state I_0 there is a match with the rule [A→ α • aβ, b] and goto (I_i, a) = I_j.

C→ • aC, a/d/$ and if the goto is applied on a' then we get the state $I_{36.}$ Hence we will create entry action[0,a] = shift 36. Similarly,

In I_0

C → • d   a/d

A → α • a β,·b

A=C, α = ε, a=d, β = ε, b=a/d

goto($I_{0,}$d)=I_{47}

hence action[0,d]=shift 47

For state I_{47}
C→ d • , a/d/$

A→ α •,a

A = C, α = d, a = a/d/$
action[47,a] = reduce by C → d i.e. rule 3

action[47,d] = reduce by C → d i.e. rule 3

action[47,$] = reduce by C → d i.e. rule 3

LALR(1) parsing table as follows -

States	Action			goto	
	a	d	$	S	C
0	S36	S47		1	2
1			Accept		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

Parsing the input string using LALR parser

The string having regular expression = a*da*d ∈ grammar G. We will consider input string as "aadd" for parsing by using LALR parsing table.

Stack	Input buffer	Action table	goto table	Parsing action
$0	aadd$	action[0,a]=S36		
$0a36	add$	action[36,a]=S36		Shift
$0a36a36	dd$	action[36,d]=S47		Shift
$0a36a36d47	d$	action[47,d]=r36	[36,C]=89	Reduce by C→ d
$0a36a36C89	d$	action[89,d]=r2	[36,C]=89	Reduce by C→ aC
$0a36C89	d$	action[89,d]=r2	[0,C]=2	Reduce by C→ aC
$0C2	d$	action[2,d]=S47		Shift
$0C2d47	$	action[47,$]=r36	[2,C]=5	Reduce by C→ d
$0C2C5	$	action[5,$]=r1	[0,S]=1	Reduce by S → CC
$0S1	$	accept		

Thus the LALR and LR parser will mimic one another on the same input.

➡ **Example 4.7 :** *Construct the SLR(1) parsing table for*

(1)E → E+T
(2)E → T
*(3)T → T*F*
(4)T → F
(5)F →(E)
(6)F → id

Solution : We will first construct a collection of canonical set of items for the above grammar. The set of items generated by this method are also called SLR(0) items. As there is no lookahead symbol in this set of items.

I_0 :
E'→ • E
E→ • E + T
E→ • T
T→ • T*F
T→ • F
F→ • (E)
F→ • id

goto (I_0,E)
I_1: E'→ E •
E→ E • + T

goto (I_0,T)
I_2 : E→ T •
T→ T • * F

goto (I_0,F)
I_3 : T→ F •

goto (I_0,()
I_4 : T→ (• E)
E→ • E + T
E→ • T
T→ • T*F
T→ • F
F→ • (E)
F→ • id

goto (I_0,id)
I_5 : F→ id •

goto (I_1,+)
I_6 : E→ E + • T
T→ • T*F
T→ • F
F→ • (E)
F→ • id

goto (I_2,*)
I_7 : T→ T* • F
F→ • (E)
F→ • id

goto (I_4,E)
I_8 : F→ (E •)
E→ E • +T

goto (I_6,T)
I_9:E→ E + T •
T→ T • *F

goto (I_7,F)
I_{10} : T→ T*F •

goto (I_8,))
I_{11} : F→ (E) •

Finally the SLR(1) parsing table will look as –

State	action						goto		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Accept			
2		r2	S7		r2	r2			
3		r4	r4		r4	r4			
4	S5			S4			8	2	3
5		r6	r6		r6	r6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		r1	S7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Remaining blank entries in the table are considered as syntactical errors.

Input string : id*id+id

We will consider two data structures while taking the parsing actions and those are – stack and input buffer.

Stack	Input buffer	action table	goto table	Parsing action
$0	id*id+id$	[0,id]=s5		Shift
$0id5	*id+id$	[5,*]=r6	[0,F]=3	Reduce by F → id
$0F3	*id+id$	[3,*]=r4	[0,T]=2	Reduce by T → F
$0T2	*id+id$	[2,*]=S7		Shift
$0T2*7	id+id$	[7,id]=S5		Shift
$0T2*7id5	+id$	[5,+]=r6	[7,F]=10	Reduce by F → id
$0 T2*7F10	+id$	[10,+]=r3	[0,T]=2	Reduce by T → T * F
$0T2	+id$	[2,+]=r2	[0,E]=1	Reduce by E → T
$0E1	+id$	[1,+]=S6		Shift
$0E1+6	id$	[6,id]=S5		Shift
$0E1+6id5	$	[5,$]=r6	[6,F]=3	Reduce by F → id
$0E1+6F3	$	[3,$]=r4	[6,T]=9	Reduce by T → F
$0E1+6T9	$	[9,$]=r1	[0,E]=1	E → E +T
$0E1	$	accept		Accept