UNIT-II

Syntax - Analysis.

Introduction, context-free Grammers, writing a grammar, Top-Down parsing, Bottom-up parsing, Introduction to LR parsing: simple LR, more powerful LR parsers, using ambiguous grammers, parser generators.

Introduction :-

Role of the parser:-

The second phase of compiler construction is syntax analysis phase.

* The parser accepts a string of tokens from the lexical analyzer and verifies that the string of token names can be generated by the grammar for the source language.
* The job of parser is to report any syntax errors in an intelligible fashion and to recover from commonly occuring errors to continue processing the remainder of the program.
* The output of the parser is a parse tree.
* syntax analyzer creates a syntactic structure of the given source program. This syntactic structure is called parse tree.

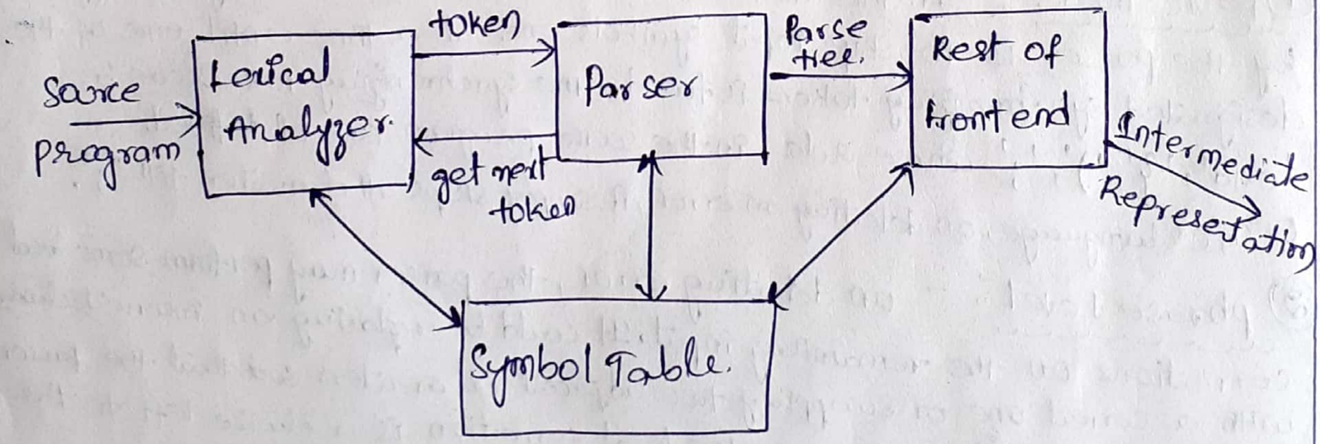


fig: Position of parser in compiler model.

* There are three general types of Parsers for grammar:

a) Universal Parser: such as cocke-Younger-kasami (CYK) algorithm and Earley's algorithms can parse any grammar.

② * Error Handling in Parsing.

Generally, errors in programs are detected at different levels.

1) At lexical analysis : unrecognized group of characters like &abc, $abc, etc., which cannot be a keyword or identifier.

(2) At syntax analysis:- Missing operator/operands in expression

3) At semantic analysis:- In compatible types of operands to an operator.

4) ~~At~~ Logical errors:- Infinite loop, Detecting logical errors at compile time is a tedious task.

In the compilation process, 90% errors are captured during the syntax and semantic analysis phase. That's why error detection and recovery in compiler are centered on parsing.

Error-recovery strategies

There are four different error-recovery strategies generally used by parsers.

→ panic mode
→ phrase Level
→ Error production
→ Global Correction :- *④ [It is preferred to have minimum changes, that is corrections in the input string]

① Panic mode :- In this method, on an error, the recovery strategy used by the parser is to skip input symbols one at a time until one of the designated synchronizing tokens is found. The synchronizing tokens can be 'end', '}', ')', ';' whose role in the source program is well defined.

Ex: In 'C' language, on detecting an error, it simply skips all characters till ";".

② phrase Level :- on detecting error, the parser may perform some local corrections on the remaining input. It could be replacing an incorrect character with a correct one or swapping two adjacent characters such that the parser can continue with the process. The local correction is a choice left to the compiler designer.

③ Error production :- If the compiler designer has a good idea about possible errors, then he can add rules with the grammar of the programming language to handle erroneous constructs. A parser is constructed for this new grammar such that it handle errors. The error productions are used by the parser to issue appropriate error diagnostics on erroneous constructs in the input.

Context free Grammars :-

A Grammar G is said to be context free if all Production in P have the form $A \to x$ where $A \in V$ and $x \in (V \cup T)^*$

V, T, P, S are the four important components in the grammatical description of a language.

V - the set of Variables, also called non-terminals. Each variables represents a set of strings, simply a language.

T - the set of terminals, which are a set of symbols that forms the strings of the language, also called terminal symbols.

P - the finite set of productions or rules that represent the recursive definition of language.

S - the start symbol. It is one of the variables that represent the language being defined.

* A ~~language~~ language generated a CFG is called a context free language (CFL).

Example 1 :-

terminal : a

nonterminal : S

productions : $S \to aS$

$S \to \epsilon$

is a simple CFG that defines $L(G) = a^*$

where $V = \{S\}$ $T = \{a\}$.

Ex: The CFG for defining palindrome over {a or b}

The productions P are

$S \to \epsilon \mid a \mid b$

$S \to aSa$

$S \to bSb$.

Grammar $G = \{ \{S\}, \{a, b\}, P, S \}$

Ex: The CFG for set of strings with equal no of a's and b's

The productions P are :-

$S \to SaSbS \mid SbSaS \mid \epsilon$

And the grammar is $G = (\{S\}, \{a, b\}, P, S\}$

Ex: The context free grammar for syntactically correct infix algebraic expressions in the variables x, y, and z.

And the grammar is $G = (\{S, T\}, \{+, *, (,), -, x, y, z\}, P, S)$

$S \to T+S \mid T-S \mid T$

$T \to T*T \mid T/T$

$T \to (S)$

$T \to x \mid y \mid z.$

This grammar can generate the string $(x+y)^{*}x-z*y/(x+x)$

Ex: A context free grammar for the language consisting of all strings over $\{a, b\}$ which contain a different number of a's ~~an~~ than b's is.

$S \to U \mid V$

$U \to TaU \mid TaT$

$V \to TbV \mid TbT$

$T \to aTbT \mid bTaT \mid \epsilon$

Here, 'T' can generate all strings with the same number of a's ~~an~~ as b's, 'U' generates all strings with more a's than b's and 'V' generates all strings with less a's than b's.

Ex:

a) Give a CFG ~~for~~ RE $(011+1)^*(01)^*$

solution: CFG for $(011+1)^*$ is $A \to CA \mid \epsilon$

$C \to 011 \mid 1$

CFG for $(01)^*$ is $B \to DB \mid \epsilon$

$D \to 01$

Hence, the final CFG is $S \to AB$

$A \to CA \mid \epsilon$

$C \to 011 \mid 1$

$B \to DB \mid \epsilon$

$D \to 01$

⑥ Give the CFG for language ~~L(G) = {a^n b^n where n≥1}~~ containing all the strings of different first and last symbols over $\Sigma = \{0,1\}$

Solution:— The string should start and end with different symbols 0, 1. But in between we can have any string on 0, 1 i.e, $(0+1)^*$. ~~[illegible]~~ Hence, the language is.

$0(0+1)^*1 \mid 1(0+1)^*0$. The grammar can be given by

$S \rightarrow 0A1 \mid 1A0$

$A \rightarrow 0A \mid 1A \mid \epsilon$.

Derivation of CFGs:—

It is a process of defining a string out of a grammar by application of the rules starting from the starting symbol.

- we can derive terminal strings, beginning with the start symbol, by repeatedly replacing a variable or non terminal symbol, by the body of the production.
- The language of CFG is the set of terminal symbols we can derive. so it's called context free language.

Ex: Derive a^4 from the grammar given below.

Terminal : a

Non Terminal : S

Productions : $S \rightarrow aS$

$S \rightarrow \epsilon$

Solution: The derivation for a^4 is.

$$\begin{aligned} S &\rightarrow aS \\ &\rightarrow aaS \\ &\rightarrow aaaS \\ &\rightarrow aaaaS \\ &\rightarrow aaaa\epsilon \Rightarrow aaaa. \end{aligned}$$

The language has strings as $\{\epsilon, a, aa, aaa, \ldots\}$

Ex:- Derive a^2 from the given grammar.

Terminal : a

Nonterminal : S

Productions : S → SS
S → a.
S → ε

Solution:-

Derivation of a^2 is

S → SS

S → SSS.

S → SSa.

→ SSSa

→ SaSa

→ εaεa

→ aa

(or)

S → SS.

→ Sa

→ aa.

Ex: find L(G) and derive "abbab"

Terminals : a, b

Non terminals : S.

productions : S → aS
S → bS
S → a
S → b.

Solution: derivation of abbab as follows.

S → aS

→ abS

→ abbS

→ abbaS

→ abbab

$L(G) = (a+b)^+$

Leftmost and Rightmost Derivation.

If a word ω is generated by a CFG by a certain derivation and at each step in the derivation, a rule of production is applied to the leftmost non-terminal in the working string, then this derivation is called a leftmost derivation (LMD)

* Practically, whenever we replace the leftmost variable first in a string, then the resulting derivation is the leftmost derivation. Similarly, replacing rightmost variable first at every step gives rightmost derivation RMD.

Ex: consider the CFG = $(\{S, X\}, \{a, b\}, P, S)$
where productions are:

$$S \to baxaS \mid ab$$
$$X \to Xab \mid aa$$

find LMD and RMD for string ω = baaaababaab.

Solution The following is an LMD

$S \to baxaS$
$\to baxabaS$
$\to baxababaS$
$\to baaaababaab$

The following is an RMD

$S \to baxaS$
$\to baxaab$
$\to baxabaab$
$\to baxababaab$
$\to baaaababaab.$

Any word that can be generated by a given CFG can have LMD | RMD.

Example:

consider the CFG:

$S \rightarrow aB \mid bA$

$A \rightarrow a \mid aS \mid bAA$

$B \rightarrow b \mid bS \mid aBB$.

find LMD and RMD for string $\omega = aabbabba$.

Solution:

The following is an LMD:

$S \rightarrow aB$
$\rightarrow aaBB$
$\rightarrow aabSB$
$\rightarrow aabbAB$
$\rightarrow aabbaB$
$\rightarrow aabbabS$
$\rightarrow aabbabbA$
$\rightarrow aabbabba$

The following is an RMD.

$S \rightarrow aB$
$\rightarrow aaBB$
$\rightarrow aaBbS$
$\rightarrow aaBbbA$
$\rightarrow aaBbba$
$\rightarrow aabSbba$
$\rightarrow aabbAbba$
$\rightarrow aabbabba$.

=.

Derivation Tree. (or) Parse tree.

The process of derivation can be shown pictorially as a tree called derivation tree.

* Derivation tree illustrate how a word is derived from a CFG.
* These trees are called syntax trees, parse trees, derivation trees.

for constructing a parse tree for a grammar $G = (V, T, P, S)$

→ The start symbol S becomes root for the derivation tree.
→ Variable or non terminal in set V is marked as interior node.

leaf node can be a terminal.

Example:-

CFG

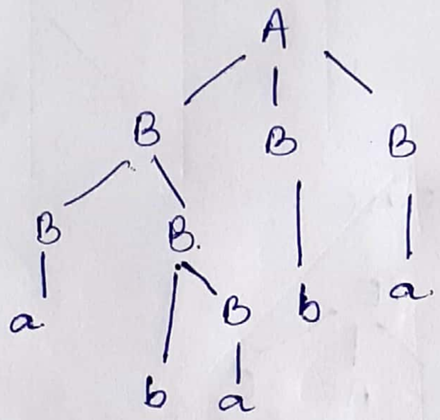
Terminals : a, b

Non-Terminals : S, B.

Productions : $A \rightarrow BBB \mid BB$

$B \rightarrow BB \mid bB \mid Ba \mid a \mid b$

String "ababa" has the following derivation tree.



By concatenating the leaves of the derivation tree from left to right, we get a string which is known as yield of the derivation tree.

Ambiguity:-

A grammar G is said tobe Ambiguous Grammar if there are two/more LMDs or RMDs for the same string and that has more than one Parse tree.

consider the production.

$E \rightarrow EAE \,/\, (E) \,/\, id$

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

check the above grammar is ambiguous or not.

Assume $w = id+id*id$

$E_{LMD} \rightarrow \underline{E}AE$

$\rightarrow id\underline{A}E$

$\rightarrow id+\underline{E}$

$\rightarrow id+EAE$

$\rightarrow id + id \cdot A E$

$\rightarrow id + id * E$

$\rightarrow id + id * id.$

Again consider LMD for $w = id + id * id$

$E_{LMD} \rightarrow \underline{E}AE$

$\rightarrow EAEAE$

$\rightarrow id\,AEAE$

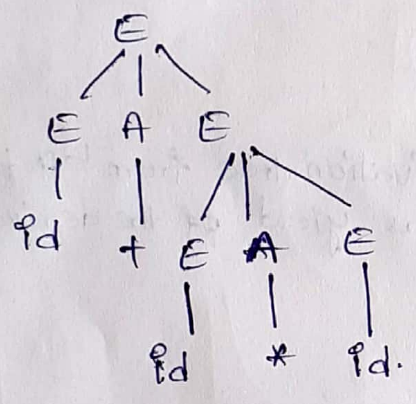
$\rightarrow id + EAE$

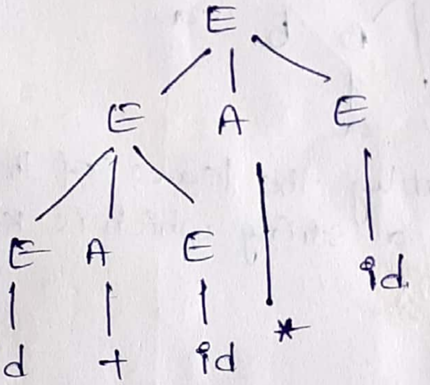
$\rightarrow id + id\,AE$

$\rightarrow id + id * E$

$\rightarrow id + id * id$

Parse tree to obtain above derivations are.





Since we get two different parse trees by applying LMD twice for the same string $w = id + id * id$.
The given grammar is ambiguous.

Example:- Show the following is ambiguous / not

$S \rightarrow aSbS \mid bSaS \mid \epsilon$

Solution:
Assume $w = abab$.

Consider LMD for $w = abab$.

$S_{lmD} \Rightarrow aSbS$

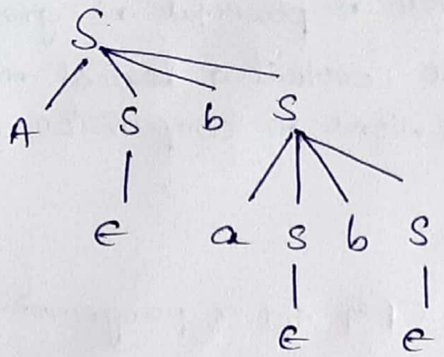
$\Rightarrow a\epsilon bS$

$\rightarrow abaSbS.$

$\rightarrow aba\epsilon bS$

$\rightarrow abab\epsilon \Rightarrow \underline{abab}$

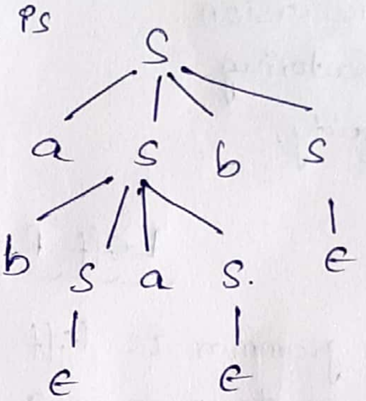
Parse tree for derivation is



Again consider LMD for ω = abab

S lmd → aSbS.
→ abSaSbS
→ abεaSbS
→ abaεbS
→ ababε
→ abab

Parse tree is



Since two different parse trees are obtained for the same string ω = abab. The given grammar is ambiguous.

Writing A Grammar.

It describes how to divide a work between a lexical analyzer and a parser, converting ambiguous grammar to unambiguous grammar, it also describes eliminating left recursion and left factoring.

Lexical VS Syntactic Analysis.

A regular expression can also be described by a grammar.

* Regular expressions are most useful in describing constructs like identifiers, constants, keywords and so on.

* Grammars are most useful in describing nested structures like matching begin-ends, balanced paranthesis and so on.

* Regular expressions are used to define the lexical syntax of a language because:

  * Notations for tokens can easily be understood by regular expressions than grammars.

* Lexical rules of a language are quite simple and to describe them we do not need a notation as powerful as grammars.

* It is easier ~~the~~ and efficient to construct a lexical analyzer automatically from regular expressions in comparision with arbitary grammars.

=

Issues to resolve when writing a CFG for a programming language.

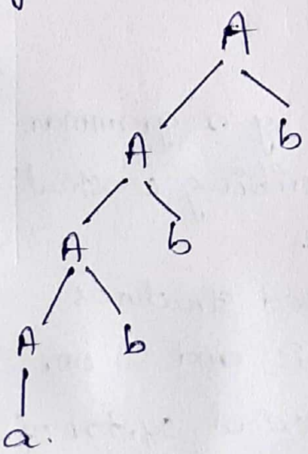
1. Left recursion
2. Left factoring.
3. Ambiguity.

Left Recursion

A grammar is left recursive if it has a non terminal "A" such that there is a derivation.

$A \Rightarrow A\alpha$ for some string α. If this grammar is used in some parsers (top-down parser), the parser may go into infinite loop.

* Consider the following left reursive grammar $A \rightarrow Ab/a$. To derive a string "abbb" there is an ambiguity as to how many times the non terminal "A" has to be expanded. As grammar is left recursive, the tree grows toward left.



* Top-down parsing technique cannot handle left recursive grammar, so we have to convert left-recursive grammar into an equivalent grammar, which is not left-recursive.

=

Eliminating left-recursion

A Grammar $G = \langle V, T, P, S \rangle$ is said to be left recursive if it has a non-terminal 'A' such that there is a derivation

$A \to A\alpha$ ~~where~~ where α is some string.

$$\Downarrow$$

$$A \to A\alpha_1 \mid A\alpha_2 \mid \dots A\alpha_m \mid \beta_1 \mid \dots \mid \beta_m.$$

immediate left recursion.

→ To eliminate immediate left recursion rewrite the grammar as

$$\boxed{\begin{array}{l} A \to \beta_1 A' \mid \beta_2 A' \mid \dots \mid \beta_m A' \\ A' \to \alpha_1 A' \mid \alpha_2 A' \mid \dots \mid \alpha_m A' \mid \epsilon \end{array}}$$

Immediate left recursion Example:-

Ex:

$E \to E+T \mid T$

$T \to T*F \mid F$

$F \to id \mid (E)$

After eliminating immediate left recursion, we get grammar as

$E \to TE'$

$E' \to +TE' \mid \epsilon$

$T \to FT'$

$T' \to *FT' \mid \epsilon$

$F \to id \mid (E)$

Example 2: Eliminate left-recursion in the following grammar

$S \to Aa \mid b$

$A \to Ac \mid Sd \mid \epsilon$.

order of non-terminals A, S.

for A: Eliminate the immediate left-recursion in A

$A \to SdA'$

$A' \to cA' \mid \epsilon$.

for S:

Replace $S \rightarrow Aa$ with $S \rightarrow SdA'a \mid A'a$.

So, we will have $S \rightarrow SdA'a \mid A'a \mid b$.

Eliminate the immediate left-recursion in S

$S \rightarrow A'aS' \mid bS'$

$S' \rightarrow dA'aS' \mid \epsilon$

So the resulting non-left-recursive is.

$S \rightarrow A'aS' \mid bS'$

$S' \rightarrow dA'aS' \mid \epsilon$

$A \rightarrow SdA' \mid A'$

$A' \rightarrow cA' \mid \epsilon$.

By eliminating left-recursion we can avoid top-down parser to go into infinite loop.

LEFT FACTORING

Sometimes we find common prefix in many productions like $A \rightarrow \alpha\beta_1, \mid \alpha\beta_2 \mid \alpha\beta_3$, where α is common prefix. While processing α we cannot decide whether to expand A by $\alpha\beta_1$ or by $\alpha\beta_2$. So this need ~~backing~~ backtracking.

* To avoid such problem, grammar can be left factoring.

If the production of the form $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \alpha\beta_3$ has α as common prefix, by left factoring, we get the equivalent grammar as

$A \rightarrow \alpha A'$

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

Example 1: $S \rightarrow ietSes \mid iets \mid f$

$\alpha = iets$ $\beta_1 = es$ $\beta_2 = \epsilon$ $A \rightarrow S$.

Left factored grammar is

$S \rightarrow ietsS' \mid f$

$S' \rightarrow es \mid \epsilon$

Example 2:- left factor the following grammar.

$A \rightarrow abB \mid aB \mid cdg \mid cdeB \mid cdfB.$

Solution:- Here common prefixes are "a" and "cd".

first takeout 'a' and rewrite the grammar as

$A \rightarrow aA' \mid cdg \mid cdeB \mid cdfB$

$A' \rightarrow bB \mid B \mid cdg \mid cdeB \mid cdfB$

Now takeout the common prefix 'cd' and rewrite the grammar as

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB.$

E13:- left factor the following grammar

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

$T \rightarrow id * T \mid id.$

$\xrightarrow{\text{Left}}$ factored grammer

$E \rightarrow TE'$

$E' \rightarrow +E \mid \epsilon$

$T \rightarrow idT'$

$T' \rightarrow *T \mid \epsilon.$

Ambiguous Grammar

A CFG is ambiguous if there exists more than one left most derivation and one rightmost derivation for at least one word in its CFL. Parse tree equivalently, more than one

Grammar

Ambiguous grammar	Unambiguous grammar
* There exists more than one LMD or RMD for a string.	* Unique LMD/RMD.
* LMD and RMD represent different parse trees.	* LMD and RMD represent the same parse tree.
* more than one parse tree for a string	* unique parse tree.

* Remember that there is no algorithm that automatically checks whether a grammar is ambiguous or not. The only way to check ambiguity is "to choose an appropriate input string and by traial and error find the number of parse trees". If more than one parse tree exists, the grammar is ambiguous. There is no algorithm that converts an ambiguous grammar to its equivalent unambiguous grammar.

Applications of CFG

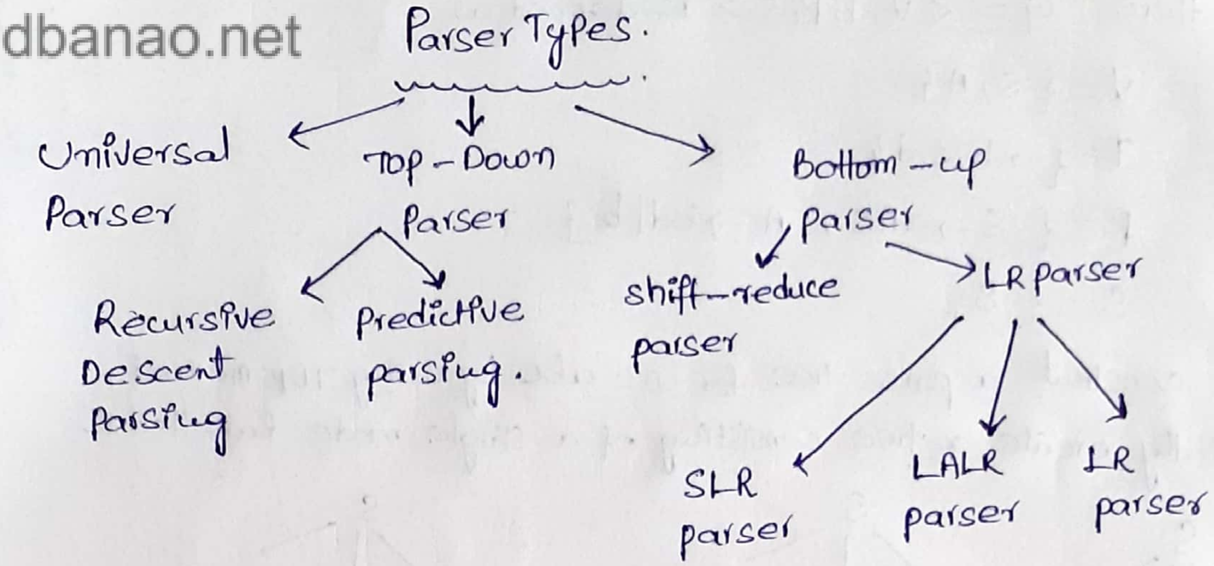
- * Grammars are useful in speifying syntax of programming language. They are mainly used in the design of programming languages.
- * They are also used in natural language processing.
- * Efficiet parsers can be automatically constructed from a CFG.
- * The expressive power of CFG is too limited to adequately capture all natural language phenomena. Therefore, extensions of CFG are of interest for computational linguistics.
- * CFGs are also used in spech recognition in processing the spoken word.

Top-Down Parsers.

For performing syntax analysis, the grammar of the language has to be speified. Context free Grammars (CFGs) are used to define standard syntax rules for the language.

- * This process of verifying whether an input string matches the grammar of the language is called parsing.
- * The process of finding a parse tree for a string of tokens is called parsing.
- * Parsing is used to check if a string of tokens can be generated by a grammar.
- * There are two-types of parsing.
  1) Top-Down parsing (TDP)
  2) Bottom-up parsing (BUP)
- * TDP is a method of Parsing where the Parse tree is constructed from the input string starting from the root and the nodes of the parse tree are created in pre order.

Parser Types.



* Parser scans the input string from left to right and idetifies that the derivation is leftmost or rightmost.

* The parser makes use of production rules for choosing the appropriate derivation.

* Different parsing techniques use different approaches in selecting the appropriate rules for derivation and finally parse tree is constructed when the parse tree is constructed from root and expanded to leaves then such type of parse tree is called Top-Down parser.

* when the parse tree is constructed from leaves to root then such type of parser is called bottom-up parser.

=

(i) Recursive Descent parsing:-

Recursive Descent parsing adopts backtracking in order to find the corret A-production to be applied. Recursive Descent parsing requires backtracking and it tries to find the LMD.

consider the following grammar.

$S \rightarrow aAc$

$A \rightarrow bd \mid b$ and input string $w = abc$.

Here $G = \langle V, T, P, S \rangle$ is defined as

$V = \{S, A\}$

$T = \{a, b, c, d\}$

$P = \{S \to aAc,\ A \to bd \mid b\}$

$S = \{S\}$.

To construct a parse tree for $w = abc$ by using TDP method, initially create a tree consisting of a single node labelled S.

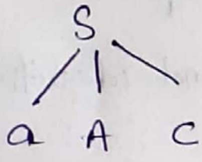
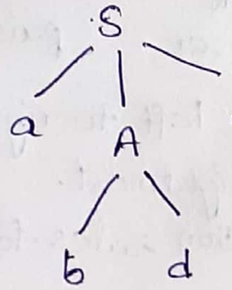
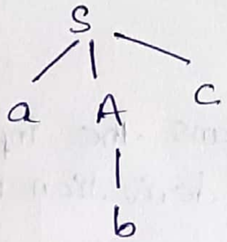


fig (a)



fails, Backtrack to A

fig (b)



successful

fig (c)

* The input pointer points to 'a' of w. Here the first production of S and the parse tree is shown in fig(a). The left most leaf labelled 'a' matches with the first symbol of w. Next we have to consider the next node 'A'. 'A' can be expanded using the production $A \to bd$ and the parse tree is shown in fig(b). Now check for second input symbol of w i.e., b. and next input symbol 'c', compare 'c' against the next leaf 'd'. Since 'd' does not match with 'c', failure is reported and go back to 'A' to see any alternative for A.

On going back to A, reset the input pointer position to second position of w and try for second alternative production of A (i.e., $A \to b$). Here leaf 'b' matches with second symbol of w. Advance the pointer to the third input symbol 'c' and compare 'c' against the next leaf 'c'. Here there is a matching. Thus, we have obtained a parse tree for w, we halt and parsing is said to be successful as shown in fig (c).

(ii) Predictive Parser:-

A predictive parser is a type of TDP obtained by writing a grammar, that is obtained after eliminating left recursion from a grammar and left factoring the resulting grammar.

* Predictive parser is an efficient way of implementing RDP.

A grammar → (Eliminate left recursion) → (Left factor) →
(A grammar suitable for PP)

* The two functions that will be used to construct the predictive parsing table are:
  1) FIRST( )
  2) FOLLOW( ).
* Both TDP and BUP can be constructed by using FIRST and FOLLOW associated with a grammar G.
* In TDP, by using FIRST and FOLLOW one can choose which production to apply based on the next input symbol.

FIRST( ):-

To complete first(X) where 'X' is some grammar symbol, apply~~ing~~ following rules until ε or no more terminals can be added to any first.

* 'X' is a grammar symbol, Hence two possibilities exist. 'X' can be a terminal or a non terminal.

Rule 1:- If 'X' is a terminal then First(X) is {X}

Ex: First(+) = {+}
first(id) = {id}.

Rule 2:- If 'X' is a non terminal, two possibilities exist

(a) X → ε is a production, then add 'ε' to First(X).

(b) If X is a non-terminal and is defined with non-null production

If the $X \to Y_1 Y_2 \cdots Y_K$ is a production, then

$First(x) = first(Y_1 Y_2 \cdots Y_K) = First(Y_1)$ if $Y_1 \Rightarrow \epsilon$ else.

$First(x) = First(Y_1) \cup First(Y_2 \cdots Y_K)$ if $Y_1 \Rightarrow \epsilon$.

Example:

$S \to aB$

$A \to c \mid \epsilon$

$B \to cbB \mid a.$

Non-terminal (LHS)	Production	First(x)
S	$S \to aB$	$first(S) = first(aB) = \{a\}$ (Here in 'aB' on RHS, 'a' is a terminal & is the first symbol. Hence add 'a' to First(S))
A	$A \to c \mid \epsilon$	$first(A) = first(c) \cup first(\epsilon)$ $= \{c, \epsilon\}$
B	$B \to cbB \mid a$	$First(B) = First(cbB) \cup First(a)$ $= \{c, a\}.$

Therefore

	First
S	$\{a\}$
A	$\{c, \epsilon\}$
B	$\{c, a\}.$

Follow ( ):– To compute the follow(A) for all non-terminals A apply the following rules until nothing can be added to any follow set.

Rule 1: place \$ in Follow(S) where 'S' is the start symbol and '\$' is the input right end marker.

$$\boxed{Follow(S) = \$}$$

Rule 2:- If there is a production $S \rightarrow aA\beta$, where β is the string of grammar symbols, then first(β) except ϵ is placed in Follow(A).

$$\boxed{\text{Follow}(A) = \text{Non-epsilon-first}(\beta)}$$

Rule 3:- If there is a production $S \rightarrow aA$ or $A \rightarrow aA\beta$ where first(β) contains ϵ, then everything in follow(S) is in follow(A).

$$\boxed{\text{follow}(A) = \text{follow}(S)}$$

Example:- Find the first and follow of all non-terminals in the Grammar.

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \epsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \mid \epsilon$
$F \rightarrow (E) \mid id$.

Solution:-

$$\text{First}(E) \Rightarrow \text{First}(TE') \Rightarrow \text{First}(T) \Rightarrow \text{First}(FT') \Rightarrow \text{First}(F)$$
$$\Rightarrow \text{First}(\text{"("}) \cup \text{First}(id)$$
$$= \{ (, id \}$$

$\text{First}(E') = \text{First}(+) \cup \text{First}(\epsilon) = \{+, \epsilon\}$

$\text{First}(T) = \text{First}(FT') \Rightarrow \text{First}(F) \Rightarrow \text{First}(() \cup \text{First}(id)$
$\Rightarrow \{ (, id \}$

$\text{First}(T') = \text{First}(*) \cup \text{First}(\epsilon) = \{*, \epsilon\}$

$\text{First}(F) = \text{First}(() \cup \text{First}(id) = \{ (, id \}$.

	First
E	$\{ (, id \}$.
E'	$\{+, \epsilon\}$
T	$\{ (, id \}$
T'	$\{*, \epsilon\}$
F	$\{ (, id \}$

Follow(E) = { \$ } by rule 1

{) } by rule 2 on production $F \rightarrow (E) \mid id$

Follow(E) = first( ))

= { \$,) }

Follow(E') = Follow(E) by rule 3 on production $E \rightarrow TE'$

= { \$,) }

Follow(T) = First(E') by rule 2 on production $E \rightarrow TE'$

= { +, ϵ } → Follow(E)

Follow(E) by rule 3 on production $E \rightarrow TE'$

= { \$,) }

= { +, \$,) }.

Follow(T') = Follow(T) by rule 3 on production $T \rightarrow FT'$

= { +, \$,) }

Follow(F) = First(T') by rule 2 on production $T' \rightarrow FT'$

= { *, ϵ → Follow(T) by rule 3 on on $T \rightarrow FT'$ }

= { *, +,), \$ }.

Non-terminal	Follow
E	{ \$,) }
E'	{ \$,) }
T	{ +, \$,) }
T'	{ +, \$,) }
F	{ +,), *, \$ }

Scanned with CamScanner

LL(1) Grammars :-

LL(1) class of grammars are used to construct predictive Parsers (PP) that does not require backtracking.

LL(1) stands for ..

L stands for scanning input from left to right

L stands for Left most Derivation.

1 stands for consisting of one input symbol of look-ahead at each step to make decisions for parsing actions.

Construction of predictive parsing Table.

for any grammar G, the following algorithm can be used to construct the predictive parsing table.

The algorithm is

Input :- Grammar G.

Output :- parsing table M

Method :- step1) for each production S→a of the grammar perform step 2 and 3.

Step 2 :- for each terminal 'a' in First(a), add S→a to M.[S, a].

Step3 :- ~~for~~ If 'ε' is in First(a), add S→a to M[S, b] for each terminal b in Follow (S). If 'ε' is in First(a) and $ is in Follow(S), add S→a to M[S, $].

The above algorithm can be applied to any grammar G to produce parsing table M. If G is left recursive or ambiguous, then there will be (at least) multiple defined entries in the parsing table M.

* All undefined entries in symbol table are error entries.

Example:— Construct LL(1) parsing table. for the given grammar.

E → TE'
E' → +TE' | ε
T → FT'
T' → *FT' | ε
F → (E) | id.

⇒

Non terminal	First( )	Follow ( )
E	{ (, id }	{), $ }
E'	{ +, ε }	{), $ }
T	{ (, id }	{ +,), $ }
T'	{ *, ε }	{ +,), $ }
F	{ (, id }	{ +, *,), $ }

LL(1) Table for the given grammar is.

Nonterminal.	(	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→(E)	F→id.				

→ To construct LL(1) Table, there has to be five raos since there are five non-terminals.

→ In the rao E, place E→TE' in the column given by First(TE') = { (, id }

→ In the rao E', place E'→+TE' in the column given by First(+TE') = { + }

→ In the rao E', place E'→ε in the column given by Follow(E') = {), $ }

→ In the rao F, place F→(E) in the column given by First((E)) = { (}

→ In the rao F, place F→id in the column given by First(id) = { id }

How to parse a string using predictive parsing.

~~Steps~~ Example

(1) Construct predictive parsing table and show the sequence of moves made by the Parser for ω = abba.

$S \rightarrow aBa$

$B \rightarrow bB | \epsilon$

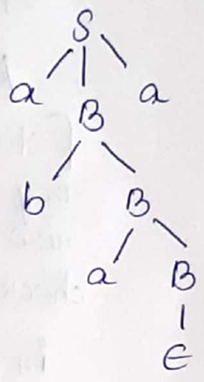
Step1:- Derive ω = abba. (Leftmost derivation)

$S \rightarrow aBa$

$\rightarrow abBa$ [∵ $B \rightarrow bB$]

$\rightarrow abbBa$ [$B \rightarrow bB$]

$\rightarrow abba$ [$B \rightarrow \epsilon$]



Step2: Find First and Follow.

	S	B
First	{a}	{b, E}
Follow	{$}	{a}

Step3: predictive parsing table construction

Non terminal	a	b	$
S	S→aBa		
B	B→E	B→bB	

Since there are no multiple entries in the table, the given grammar is LL(1) grammar

* Initial configuration of predictive parsing table is

Stack	Input
$S	ω$

Final configuration is ~~successes~~

Stack	Input
$	$ → (successful parsing)

Step4 :— Sequence of moves made by parser for string w= abba.

Stack	Input	output
① $ S start symbol	abba $ (w=abba) check for M[S,a] which is S → aBa in parsing table.	S → aBa replace S on stack by aBa.
② $ a B ~~a~~ ⇓ $ a B	~~a~~bba $ [stack = a & input = a they are same, hence pop. now stack = B input = b check M[B,b] = B → bB in parsing table.	pop 'a' B → bB replace B on stack by bB in - reverse order in step3.
③ $ a B ~~b~~ ⇓ $ a B	~~b~~ba $ [stack = b & input = b same, hence pop now and stack = B input = b M[B,b] = B → bB	pop 'b' replace B → bB
④ $ a B ~~b~~ ⇓	~~b~~a $ [stack = b & input = b] pop stack = B , input = a M[B,a] = B → ε	pop 'b' replace B → ε
⑤ $ a ε ⇓ a	a $ stack = a & input = a pop	pop 'a'
⑥ $	$	Successful parsing as stack = $ & input = $

Problems on predictive parsing

Example 1:- construct the predictive parsing table from the below grammar.

$S \rightarrow aABb$
$A \rightarrow c \mid \epsilon$
$B \rightarrow d \mid \epsilon$

$\Longrightarrow$

Non-terminal	First( )
S	$\{a\}$
A	$\{c, \epsilon\}$
B	$\{d, \epsilon\}$

Follow(S) = $\{\$\}$
Follow(A) = {First(B) by rule 2 from $S \rightarrow aABb$.}
= $\{d, \textcircled{\epsilon} \rightarrow$ first(b) by rule 2}
= $\{d, b\}$ (as First(B) ~~continues~~ contains ε continues with b in 'Bb').

Follow(B) = First(b) by rule 2
= $\{b\}$.

	Follow.
S	$\{\$\}$
A	$\{d; b\}$
B.	$\{b\}$

$\Rightarrow$

Predictive Parsing table.

	a	b	c	d	$
S	$S \rightarrow aABb$				
A		$A \rightarrow \epsilon$	$A \rightarrow c$	$A \rightarrow \epsilon$	
B		$B \rightarrow \epsilon$		$B \rightarrow d$	

~~Due to no multiple entries~~ As there are no multiple entries in the parsing table, the grammar is said to be LL(1) grammar.

Example 2:- construct LL(1) parsing table.

$S \rightarrow aBCd \mid dCBe$
$B \rightarrow bB \mid \epsilon$
$C \rightarrow ca \mid ac \mid \epsilon$

First of every Non-terminal symbol is.

First(S) = First(a) ∪ First(d) = $\{a, d\}$
First(B) = First(b) ∪ First(ε) = $\{b, \epsilon\}$
First(C) = First(c) ∪ First(a) ∪ First(ε) = $\{c, a, \epsilon\}$.

Follow of every non-terminal symbol is.

Follow (S) = $

Follow (B) = First (C) by rule 2 from production S → aBCd
= {c, a, ε → First(d)} as ε continues with d in 'aB·Cd'
= {c, a, d}
= First (e) by rule 2 from production S → dCBe
= {e}
= {c, a, d, e}

Follow (C) = First(d) by rule 2 from production S → aBCd
= {d}
= First (B) by rule 2 from production S → dCBe
= {b, ε → First (e)} = {b, e}
= {b, d, e}.

	First
S	{a, d}
B	{b, ε}
C	{c, a, ε}.

	Follow( )
S	{$}
B	{a, c, d, e}
C	{d, b, e}.

LL(1) Parsing table.

	a	b	c	d	e	$
S	S → aBCd			S → dCBe		
B	B → ε	B → bB	B → ε	B → ε	B → ε	
C	C → ac	C → ε	C → ca.	C → ε	C → ε	

The above given grammar is LL(1) grammar, as there are no multiple entries in the parsing table.

Example3: construct LL(1) parsing table for the given grammar

$S \rightarrow AaB \mid CbB \mid Ba$
$A \rightarrow da \mid BC$
$B \rightarrow g \mid \epsilon$
$C \rightarrow h \mid \epsilon$

First find first( ) all non terminals.

$First(S) = First(AaB) \cup First(CbB) \cup First(Ba)$

$\Rightarrow First(AaB) \Rightarrow First(A)$
$\Rightarrow First(da) \cup First(BC)$
$\Rightarrow First(da) \cup First(g) \cup First(\epsilon)$
$\Rightarrow \{d, g, \textcircled{\epsilon} \rightarrow First(C)\}$
$= \{d, g, First(h) \cup First(\epsilon)\}$
$= \{d, g, h, \textcircled{\epsilon} \rightarrow First(a)\}$
$= \{d, g, h, a\}.$

$\Rightarrow First(CbB) \Rightarrow First(C)$
$\Rightarrow First(h) \cup First(\epsilon)$
$\Rightarrow \{h, \textcircled{\epsilon} \rightarrow First(b)\}$
$= \{h, b\}.$

$\Rightarrow First(Ba) = First(B)$
$= First(g) \cup First(\epsilon)$
$= \{g, \textcircled{\epsilon} \rightarrow First(a)\}$
$= \{g, a\}.$

$First(S) = \{d, g, h, a\} \cup \{h, b\} \cup \{g, a\}$
$= \{a, b, d, g, h\}.$

$First(A)$ = already obtained above.
$= \{d, g, h, a\}$

$First(B) = \{g, \epsilon\}$

$First(C) = \{h, \epsilon\}.$

Follow() for all non-terminals is

Follow(S) = \$

Follow(A) = First(a) by rule 2 from production S → AaB.
= {a}

Follow(B) = Follow(S) by rule 3 from production S → AaB.
= {\$}
⇒ First(C) by rule 2 from production A → BC
⇒ {h, ⓔ → Follow(A)}
= {h, a}
= {\$, h, a}.

Follow(C) = First(b) by rule 2 from S → CbB.
= {b}
= Follow(A) by rule 3 from A → BC
= {a}.
= {a, b}.

	First( )
S	{a, b, d, g, h}
A	{a, d, g, h}
B	{g, e}
C	{h, ϵ}.

	Follow( )
S	{\$}
A	{a}
B	{\$, a, h}
C.	{b, a}.

LL(1) Parsing table.

	a	b	d	g	h	\$
S	S → AaB S → Ba	S → cbB	S → AaB.	S → AaB S → Ba	S → AaB. S → cbB	
A	A → BC.		A → da.	A → BC	A → BC	
B	B → ϵ			B → g	B → ϵ	B → e
C	C → ϵ.	C → ϵ			C → h.	

In the above table, so many places we have got more than one entry. This should not exist for a unambiguous grammar. So this grammar is ambiguous grammar and not in LL(1) grammar eg3:- $S \to TL;$ $\quad w=$ int id id ④ $S \to 0S0 \,/\, 1S1 \,/\, 2$

$T \to$ int/float

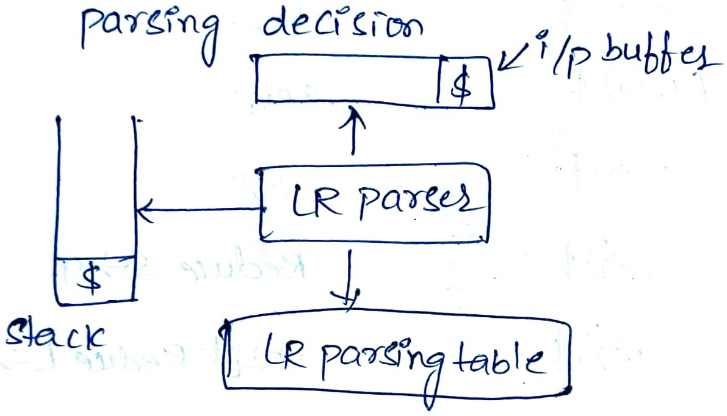
$L \to L,id \,/\, id$ $\quad w = 10201$

Introduction to LR parsing:- LR parser is a type of Bottom-up parser

→ LR(k) parser scans i/p from left to Right

→ It use Right most Derivation in Reverse orders. ~~ordes to~~ ~~deserve the~~ to reduce i/p string to start symbol of grammar

K → No. of lookahead symbols that are used to make parsing decision



Stack – Datastructure used to store grammar symbols.

i/p – " " " " " i/p string to be parsed

LR parser → uses algorithm to make decisions.

LR parsing table – is constructed by using LR(0) items.

– These table uses two functions (i)

(i) closure (ii) Action.

Benefits of LR(k) parsing:-

→ most generic Non-Backtracking shift Reduced parsing Technique.

→ These parsers can recognize all programming languages for which CFG can be written.

→ They are capable of detecting syntactic errors as soon as possible while scanning of i/p.

Simple LR Parsing:-

Eg:-
E→E+T
E→T
T→T*F
T→F
F→(E)
F→id

① Augumented Grammar
~~E'→.E~~ E'→E
E→E+T
E→T
T→T*F
T→F
F→(E)/id

Steps to construct SLR parser

① Introduce augumented grammar

② calculate canonical collection of LR(0) items.

③ construct SLR parsing table by using (i) Goto (ii) Action functions

I0:
E'→.E
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

Goto (I0, E)—I1
E'→E.
E→E.+T

(I0, T) —I2
E→T.
T→T.*F

(I0, F) →I3
T→F.

(I0, ( ) →I4
F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

(I0, id)—I5
F→id.

(I1, +) —I6
E→E+.T
T→.F
T→.T*F
F→.(E)
F→.id

(I2, *) —I7
T→T*.F
F→.(E)
F→.id

(I4, E) —I8
F→(E.)
E→E.+T

(I4, T) —I2
E→T.
~~E→E.+T~~
T→T.*F

(I4, F) —I3
T→F.

(I4, ( ) —I4
F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

(I4, id) —I5
F→id.

(I6, T) —I9
E→E+T.
T→T.*F

(I6, F) —I3
T→F.

(I6, ( ) —I4
F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→ id

(I6, id) —I5
F→id.

(I7, F) —I10
T→T*F.

(I7, ( ) —I4
F→(.E)
E→.E+T
E→.T
T→.T*F
T→.F
F→.(E)
F→.id

(I7, id) —I5
F→id.

(I8,)) —I11
F→(E).

(I8, +) —I6
E→E+.T
T→.T*F
T→.F
F→.(E)
F→.id

$(I_9, *)$ – (I_7)

$T \to T * \cdot F$

$F \to \cdot (E)$

$F \to \cdot id$

STATE	ACTION						GOTO		
	id	+	*	(	)	$	E	T	F
0	S5			S4			1	2	3
1		S6				Acceptance			
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			

1. $I_1 : E' \to E.$
2. $I_2 : E \to T.$
3. $I_3 : T \to F.$ (r_4)
4. $I_5 : F \to id.$
5. $I_9 : E \to E+T.$

$I_{10} : T \to T*F.$

$I_{11} : F \to (E).$

Follow(E) =

$E' \to \cdot E$

1 $E \to E+T$

2 $E \to T$ (r_2)

3 $T \to T*F$

4 $T \to F$ (r_3)

5 $F \to (E)$

6 $F \to id$

Follow(E') = $

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

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

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

(2, $) (2, +) (2,)) — r_2

(3, $) (3, +) (3,)) (3, *) — r_3

(5, $) (5, +) (5,)) (5, *) — r_6

$I_1 : S' \to S.$ 2 $ 4

w = id*id+id

STACK	Input	ACTION
$0	id*id+id$	shift5
$0id5	*id+id$	reduce6 F→id
$0F3	*id+id$	reduce4 T→F
$0T2	*id+id$	S7
$0T2*7	id+id$	S5
$0T2*7id5	+id$	r_6 F→id
$0T2*7F10	+id$	r_3 T→T*F
$0T2	+id$	r_2 E→T
$0E1	+id$	S6
$0E1+6	id$	S5 ~~[illegible]~~
$0E1+6id5	$	r_6 F→id
$0E1+6F3	$	r_4 T→F
$0E1+6T9	$	r_1 E→E+T
$0E1	$	Acceptance

Eg:-2 S→As/b

A→SA/a   w='abab'

→ shift → while performing shifting first we have to shift i/p symbol on to the top of the stack & then shift the state Number.

→ reduce → while performing reduce operation (if F→id we reducing), if Right hand side contains one symbol we have to pop two symbols from top of the stack.

(ii) LR(0) parser:-

Steps for construct LR(0) parser

(1) Introduce Augumented grammar

(2) calculate canonical collection of LR(0) items.

(3) construct LR(0) parsing table by using (i) Goto (ii) Action functions.

w = string = aabb$

Eg:- $S \to AA$

$A \to aA/b$

Augumented grammar

$S' \to S$

$S \to AA$

Closure:-

LR(0) items - I_0

$S' \to \cdot S$

$S \to \cdot AA$

$A \to \cdot aA / \cdot b$

Goto

(I_0, S) - I_1

$S' \to S\cdot$

Goto (I_0, A) - I_2

$S \to A\cdot A$

$A \to \cdot aA / \cdot b$

Goto (I_0, a) - I_3

$A \to a\cdot A$

$A \to \cdot aA / \cdot b$

I_4 (I_0, b)

$A \to b\cdot$

$(I_2, A) \to I_5$

$S \to AA\cdot$

(I_2, a) - I_3

$A \to a\cdot A$

$A \to \cdot aA / \cdot b$

(I_2, b) - I_4 I_3,

~~$A \to aA$~~

$A \to b\cdot$

(I_3, a) - I_3

$A \to a\cdot A$

$A \to \cdot aA / \cdot b$

(I_3, A) - I_6

$A \to aA\cdot$

(I_3, b) - I_4

$A \to b\cdot$

LR(0) parsing Table:

	Action			Goto	
	a	b	$	A	S
0	S3	S4		2	1
1			Accept		
2	S3	S4		5	
3	S3	S4		6	
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

write down the states that contains "•" at the end of the production.

$I_1 \to S' \to S.$

$I_4 \to A \to b.$

$I_6 \to A \to aA.$

$I_5 \to S \to AA.$

Given Grammar

$S \to AA$ - ①

$A \to aA$ - ②

$A \to b$ - ③

→The given string w= aa bb ⇒ w=aabb$

~~Stack~~	~~Input~~	~~Action.~~
~~$0~~	~~aabb$~~	~~shift 3~~
~~$0a3~~	~~abb$~~	~~shift 3~~
~~$0a3a3~~	~~bb$~~	~~r1 (S→AA~~ ~~shift 4~~
~~$0a3a3b4~~	~~b$~~	~~reduce r3~~ ~~A→b~~
~~$0a3a3~~	~~b$~~	~~shift 4~~
~~$0a3a3b4~~	~~$~~	

→ while performing shiftting 1st we have to shift i/p symbol on to the top of the stack & then state Numbers

Stack	Input	Action
$0	aabb$	S3
$0a3	abb$	S3
$0a3a3	bb$	S4
$0a3a3b4	b$	reduce r3 A→b
$0a3a3A6	b$	r2 A→aA
$0a3A6	b$	r2 A→aA
$0A2	b$	S4
$0A2b4	$	r3 A→b
$0A2A5	$	r1 S→AA
$0S1	$	Accepted.

String w=aabb is parsed though the grammar by using LR(0) parser.

Canonical LR parsing:-

Step1:- Augmented grammar

Eg:- $S \to CC$ $S' \to S$

$C \to aC$ $S \to CC$

$C \to d$ $C \to aC$

$C \to d$

$w = add$

Step 2: LR(1) items = LR(0) + lookahead

Ⓘ₀ : $S' \to \cdot S, \$$

$S \to \cdot CC, \$$

$C \to \cdot aC, a|d$

$C \to \cdot d, a|d$

(I_0, S) – (I_1)

$S' \to S\cdot, \$$

(I_0, C) – (I_2)

$S \to C\cdot C, \$$

$C \to \cdot aC, \$$

$C \to \cdot d, \$$

(I_0, a) – (I_3)

$C \to a\cdot C, a|d$

$C \to \cdot aC, a|d$

$C \to \cdot d, a|d$

(I_0, d) – (I_4)

$C \to d\cdot, a|d$

(I_2, C) – (I_5)

$S \to CC\cdot, \$$

(I_2, a) – (I_6)

~~I~~ $S \to a\cdot C, \$$

$C \to \cdot aC, \$$

$C \to \cdot d, \$$

(I_2, d) – (I_7)

$C \to d\cdot, \$$

(I_3, C) – (I_8)

$C \to aC\cdot, a|d$

(I_3, a) – (I_3)

$C \to a\cdot C, a|d$

$C \to \cdot aC, a|d$

$C \to \cdot d, a|d$

(I_3, d) – (I_4)

$C \to d\cdot, a|d$

(I_6, C) – (I_9)

$S \to aC\cdot, \$$

(I_6, a) – (I_6)

$C \to a\cdot C, \$$

$C \to \cdot aC, \$$

$C \to \cdot d, \$$

(I_6, d) – (I_7)

$C \to d\cdot, \$$

Step3:- construction of CLR parsing Table.

	Action			Goto ~~Goto~~	
	a	d	$	S	C
0	S3	S4		1	2
1			Accepted		
2	S6	S7			5
3	S3	S4			8
4	C→d r3	C→d r3			
5			S→CC r1		
6	S6	S7			9
7			C→d r3		
8	C→ac r2	C→ac r2			
9			r2 C→ac		

1: S→CC
2: C→ac
3: C→d

I_1: S'→S. , $
I_4: C→d. , a|d
I_5: S→CC. , $
I_7: C→d. , $
I_8: C→ac. , a|d
I_9: S→ac. , $

LALR Table :- $I_3 = I_6$
$I_4 = I_7$
$I_8 = I_9$

	Action			Goto	
	a	d	$	S	C
0	S36	S47		1	2
1	~~S36~~	~~S47~~	~~Acc~~ Acc		
2	S36	S47			5
36	S36	S47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

LALR parsing table

↓

w=add

Stack	i/p buffer	Action
$0	add$	S36
$0a36	dd$	S47
$0a36d47	d$	r3 C→d
$0a36c89	d$	r2 C→ac
$0c2	d$	S47
$0c2d47	$	r3. C→d
~~$0c2d~~	~~$~~	
$0c2c5	$	r1 S→CC
$0S1	$	Accepted

→ In LALR parser combine the ~~stat~~ same states that are having different lookahead symbols.

Canonical LR parser:- (CLR parser)

Step1:-

eg: $S \to CC$

$C \to cC$

$C \to d$

Augumented grammar

$S' \to S$

$S \to CC$

$C \to cC$

$C \to d$

Step2:- Calculating LR(1) items

$\underline{I_0} \to S' \to \cdot S, \$$

$S \to \cdot CC, \$$

$C \to \cdot cC, c/d$

$C \to \cdot d, c/d$

(I_0, S)

(I_0, S) — I_1

$S' \to S\cdot, \$$

(I_0, C) — I_2

$S \to C \cdot C, \$$

$C \to \cdot cC, \$$

$C \to \cdot d, \$$

(I_0, c) — I_3

$C \to c \cdot C, c/d$

$C \to \cdot cC, c/d$

$C \to \cdot d, c/d$

(I_0, d) — I_4

$C \to d\cdot, c/d$

(I_2, C) — I_5

$S \to CC\cdot, \$$

(I_2, c) — I_6

$C \to c \cdot C, \$$

$C \to \cdot cC, \$$

$C \to \cdot d, \$$

(I_2, d) — I_7

$C \to d\cdot, \$$

(I_3, C) — I_8

$C \to cC\cdot, c/d$

(I_3, c) — I_3

$C \to c \cdot C, c/d$

$C \to \cdot cC, c/d$

$C \to \cdot d, c/d$

(I_3, d) — I_4

$C \to d\cdot, c/d$

(I_6, C) — I_9

$C \to cC\cdot, \$$

(I_6, c) — I_6

$C \to c \cdot C, \$$

$C \to \cdot cC, \$$

$C \to \cdot d, \$$

(I_6, d) — I_7

$C \to d\cdot, \$$

construction of CLR parsing tables:-

	Action			Goto	
	c	d	\$	S	C
0	S_3	S_4		1	2
1			Accepted		
2	S_6	S_7		5	
3	S_3	S_4		8	
4	r_3	r_3			
5			r_1		
6	S_6	S_7		9	
7			r_3		
8	r_1	r_1			
9			r_2		

1 $S' \to S$

$S \to CC$ — 1

2 $C \to cC$ — 2

3 $C \to d$ — 3

I_1: $S' \to S\cdot, \$$

I_4: $C \to d\cdot, c/d$

I_5: $CC\cdot, \$$

I_7: $C \to d\cdot, \$$

I_8: $C \to cC, c/d$

I_9: $C \to cC\cdot \$$

Parse the input string w= dd$

Stack	input	String
$0	dd$	shift 4
$0d4	d$	reduce3 $c \rightarrow d$
$0c2	d$	shift 7
$0c2d7	$	reduce3 $c \rightarrow d$
$0c2c5	$	reduce r, $S \rightarrow cc$
$0S1	$	Accepted

↳ so, the given i/p string is Accepted by the grammar.

↳

LALR parser := (Look Ahead LR parser)

Eg1/2:- steps to construct LALR parser

(1) Introduce Augumented grammar

(2) calculate the LR(1) items

(3) Contruction of LALR parse table.

(4) Parsing of given i/p string.

Eg1:= construct LALR parser for
T6,

$S \rightarrow cc$

$c \rightarrow ac$

$c \rightarrow d$

and also parse the string w=add.

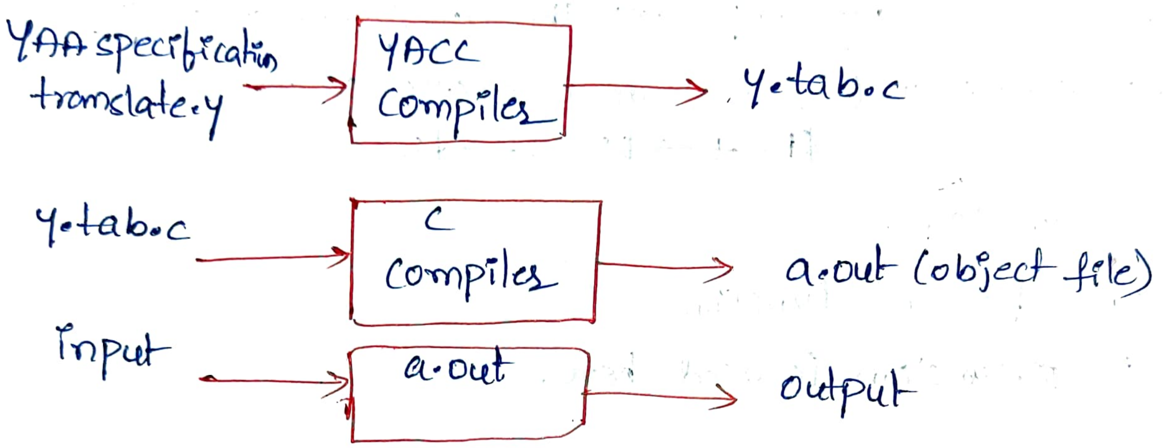
Parser generator:-

→ we use LALR parser generator YACC.

→ YACC → "yet another compiler compiler"

→ YACC is a tool to generate parser, YACC accepts an tokens as i/p and produces parse tree as o/p.

The parser generator YACC:-



→ translate.y consists of yacc specification

Structure of YACC program:-

it has three parts.

declarations
%.%
translation Rules
%.%
Auxilliary functions.

① Declarations:-

→ used to declare the c variables and constants, & header files are also specified here.

```
Syntax:- %{          eg:- %{
         %}                int a,b;
                           Const int a=20;
                           #include<stdio.h>
                          %}
```

(ii) Translation Rules:-

→ Translation Rules are enclosed between "%%" & "%%"

Syntax:- head → body1 | body 2/body. Eg:- C → aa|bb

head : body1 {semantic action}
| body2 {semantic action}
| body3 {semantic action}

$$ → represent LHS attribute value, to access LHS non-terminal value we use $$

$i →

→ In translation Rules we use two symbols $\boxed{\$\$, \$i}$

~~(ii) Auxilliary function~~:- $i → represent the ith symbol of body.

Eg:- E → E+T (if we want to access E, we have to use $1, + → $2, T → $3)

(iii) Auxilliary function:-

→ used to define "C" function.

→ yylex is ~~us~~ function is used here.

Eg:- YAAC specification (program) of simple desk calculator.

Ex:- E → E+E/T
T → T*F/F
F → (E)/id.

Declaration part:-

```
%{
    #include<ctype.h>
%}
% token DIGIT → specification of RE

%%                          it specifies I/p to desk calculator is an exp followed by \n
line: expr '\n' {printf("%d\n", $1);}
expr : expr '+' term { $$ = $1+$3 ;} $
     |term
term : term '*' factor { $$ = $1 * $3 ; }
     | factor
```

```
factor : '('expr')' {$$ =$2;}
         | DIGIT ;
%.%
yylex()
  {
    int c;
    c=getchar();
    if(isdigit(c))
  {
    yylval=c-'0';
   return DIGIT;
  }
    return c;
 }
```

yylex → is a lexical analyzer function which takes our source program as input and produces token as o/p

① ⑳ ㉒

1) Using Ambiguous Grammars

→ For language constructs like expressions, an ambiguous grammar provides a shorter, more natural specification than any equivalent unambiguous grammar.

→ Another use of ambiguous grammar is in isolating commonly occurring synetatic constructs for special-case optimization, we can add new productions to grammar.

→ Sometimes ambiguity rules allow only one parse tree, in such case it is unambiguous, It is possible to design an LR parser that allows same ambiguity resolving choices.

Precedence & Associativity to Resolve Conflicts:-

→ Consider ambiguous grammar with operators '+' & '*'

$$E \rightarrow E+E \mid E*E \mid (E) \mid id \quad —(1)$$

→ $E \rightarrow E+T$, $T \rightarrow T*F$, generates same language gives lower precedence to ~~*~~ '+' than *, makes left associative. [Use ambiguous grammar]

→ ~~Bef~~ First we change associativity and precedence of operators + & * without disturbing above grammar.

→ Second, the parser for the unambiguous grammar will spend a substantial fraction of its time reducing by the productions $E \rightarrow T$ & $T \rightarrow F$

→ $E' \rightarrow E$, (1) is ambiguous there will be parsing-action conflicts when we try to produce LR parsing table.

I_0 : E' → ·E
E → ·E+E
E → ·E*E
E → ·(E)
E → ·id

I_1 : E' → E·
E → E·+E
E → E·*E

I_2 : E → (·E)
E → ·E+E
E → ·E*E
E → ·(E)
E → ·id

I_3 : E → id·

I_4 : E → E+·E
E → ·E+E
E → ·E*E
E → ·(E)
E → ·id

I_5 : E → E*·E
E → ·E+E
E → ·E*E
E → ·(E)
E → ·id

I_6 : E → (E·)
E → E·+E
E → E·*E

I_7 : E → E+E·
E → E·+E
E → E·*E

I_8 : E → E*E·
E → E·+E
E → E·*E

I_9 : E → (E)·

LR(0) items of augmented exp grammar

Conflict occur in I_7 & I_8 E → E+E· , E → E*E·

PREFIX	STACK	INPUT
E+E	0147	*id$

If i/p is id+id.

State	ACTION						GOTO
	id	+	*	(	)	$	
0	S3			S2			1
1		S4	S5			accept	
2	S3			S2			6
3		r4	r4		r4	r4	
4	S3			S2			7
5	S3			S2			8
6		S4	S5		S9		
7		r1	S5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

Fig :- parsing table for grammar

"Dangling-Else" Ambiguity :-

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

Consider ~~The~~ grammar, an abstraction of this grammar where 'i' stands for if expr then, e stands for else and 'a' stands for "all other production".

$S' \to S$ (augmented grammar)

$S \to iSeS \,/\, iS \,/\, a$

I_0 : $S' \to \cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_1 : $S' \to S\cdot$

I_2 : $S \to i\cdot SeS$
$S \to i\cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_3 : $S \to a\cdot$

I_4 : $S \to iS\cdot eS$

I_5 : $S \to iSe\cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_6 : $S \to iSeS\cdot$

Fig:- LR(0) states for augmented grammar

if expr then stmt —(2)

Ambiguity arises in I_4 shift/reduce conflict

$S \to iS\cdot eS$ calls for a shift of e

FOLLOW(S) = {e, \$}, $S \to iS\cdot$ call for reduction by ~~S→iS~~
$S \to iS$ on i/p 'e'

In (2) should we shift else onto stack or reduce if expr then stmt, we should shift else because it is associated with previous then.

SLR parsing table is Constructed.

STATE	ACTION				GOTO
	i	e	a	$	S
0	S2		S3		1
1				accepted	
2	S2		S3		4
3		r3		r3	
4		S5		r2	
5	S2		S3		6
6		r1		r1	

input is iiaea, At line (5) state 4 selects the shift action on ~~ip~~ input e, whereas at line (9) state 4 calls for reduction by s → is on input $.

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		iiaea $	shift
(2)	0 2	i	iaea $	shift
(3)	0 2 2	ii	aea $	shift
(4)	0 2 2 3	iia	ea $	shift
(5)	0 2 2 4	iiS	ea $	reduce by s → a
(6)	0 2 2 4 5	iiSe	a $	shift
(7)	0 2 2 4 5 3	iisea	$	reduce by s → a
(8)	0 2 2 4 5 6	iises	$	reduce by s → ises
(9)	0 2 4	is	$	reduce by s → is
(10)	0 1	S	$	accept

Fig :- parsing actions on input iiaea