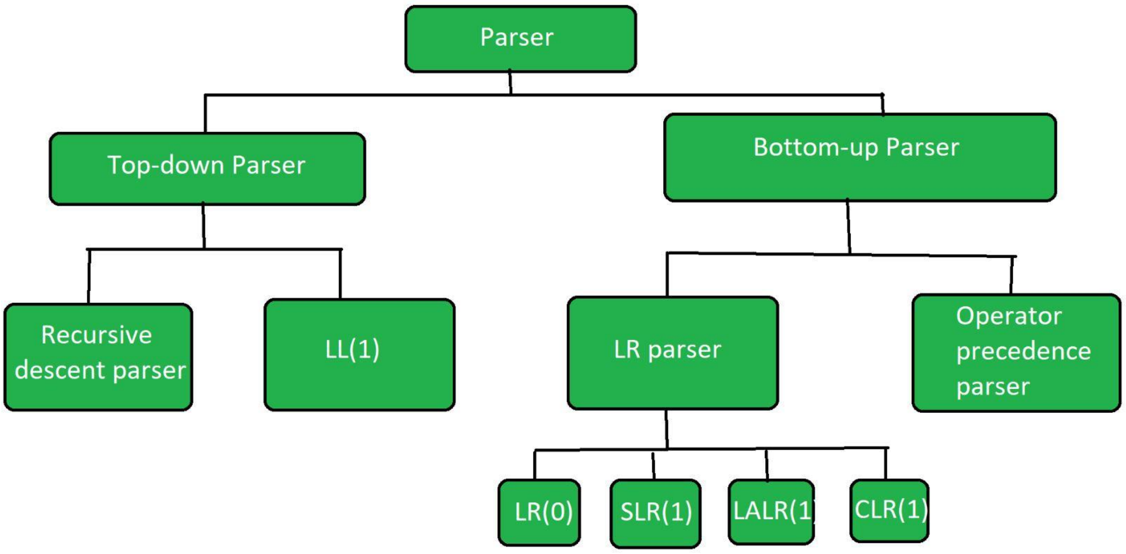
PARSERS IN COMPILER DESIGN

The **parser** is that phase of the compiler which takes a token string as input and with the help of existing grammar, converts it into the corresponding Intermediate Representation(IR). The parser is also known as *Syntax Analyzer.*



Classification of Parser

Types of Parser:

The parser is mainly classified into two categories, i.e. Top-down Parser, and Bottom-up Parser. These are explained below:

Top-Down Parser:

The top-down parser is the parser that **generates parse for the given input string** with the help of grammar productions by expanding the non-terminals i.e. it starts from the start symbol and ends on the terminals. It uses left most derivation.

Further Top-down parser is classified into 2 types: A recursive descent parser, and Non-recursive descent parser.

1. **Recursive descent parser** is also known as the Brute force parser or the backtracking parser. It basically generates the parse tree by using brute force and backtracking.
2. **Non-recursive descent parser** is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses a parsing table to generate the parse tree instead of backtracking.

Bottom-up Parser:

Bottom-up Parser is the parser that generates the parse tree for the given input string with the help of grammar productions by compressing the terminals i.e. it starts from terminals and ends on the start symbol. It uses the reverse of the rightmost derivation.

Further Bottom-up parser is classified into two types: LR parser, and Operator precedence parser.

- **LR parser** is the bottom-up parser that generates the parse tree for the given string by using unambiguous grammar. It follows the reverse of the rightmost derivation.
  LR parser is of four types:

(a)LR(0)
(b)SLR(1)
(c)LALR(1)
(d)CLR(1)

- **<u>Operator precedence parser</u>** generates the parse tree from given grammar and string but the only condition is two consecutive non-terminals and epsilon never appears on the right-hand side of any production.
- The operator precedence parsing techniques can be applied to **Operator grammars.**
- **Operator grammar:** A grammar is said to be operator grammar if there does not exist any production rule on the right-hand side.
  1. as ε(Epsilon)
  2. Two non-terminals appear consecutively, that is, without any terminal between them operator precedence parsing is not a simple technique to apply to most the language constructs, but it evolves into an easy technique to implement where a suitable grammar may be produced.

Predictive Parser

LL(1) Grammar

- Predictive parsers can be constructed for a class of grammars called LL(1).
- L->Left

  L->Leftmost derivation

  1->One input symbol at each step
- No left recursive or ambiguous grammar can be LL(1)

Conditions of LL(1)

Grammar G is called LL(1) if and only if whenever,

If A->α|β are two distinct productions of G, the following conditions hold :-

1. (a) FIRST(α) , FIRST(β) must be disjoint. This is to be able to deterministically guess the production.

   (b) At most one of the strings α or β can derive ε (Since FIRST(α), FIRST(β) are disjoint.

2. If α -> ε then FIRST(β) and FOLLOW(A) must be disjoint

Algorithm for construction of parsing table

INPUT :- Grammar G

OUTPUT:- Parsing table M

For each production A -> α , do the following :

1. For each terminal ‘a’ in FIRST(A), add A-> α to M[A,α].
2. If ε is in FIRST(α) then for each terminal b in FOLLOW(A). A -> α to M[A,b]. If b is $ then also add A -> α to M[A,$].
3. If there is no production in M[A,a] , then set M[A,a] to error.

Example of LL(1) grammar

E -> TE’

E -> +TE’|ε

T -> FT’

T’ -> *FT’|ε

F -> (E)|id

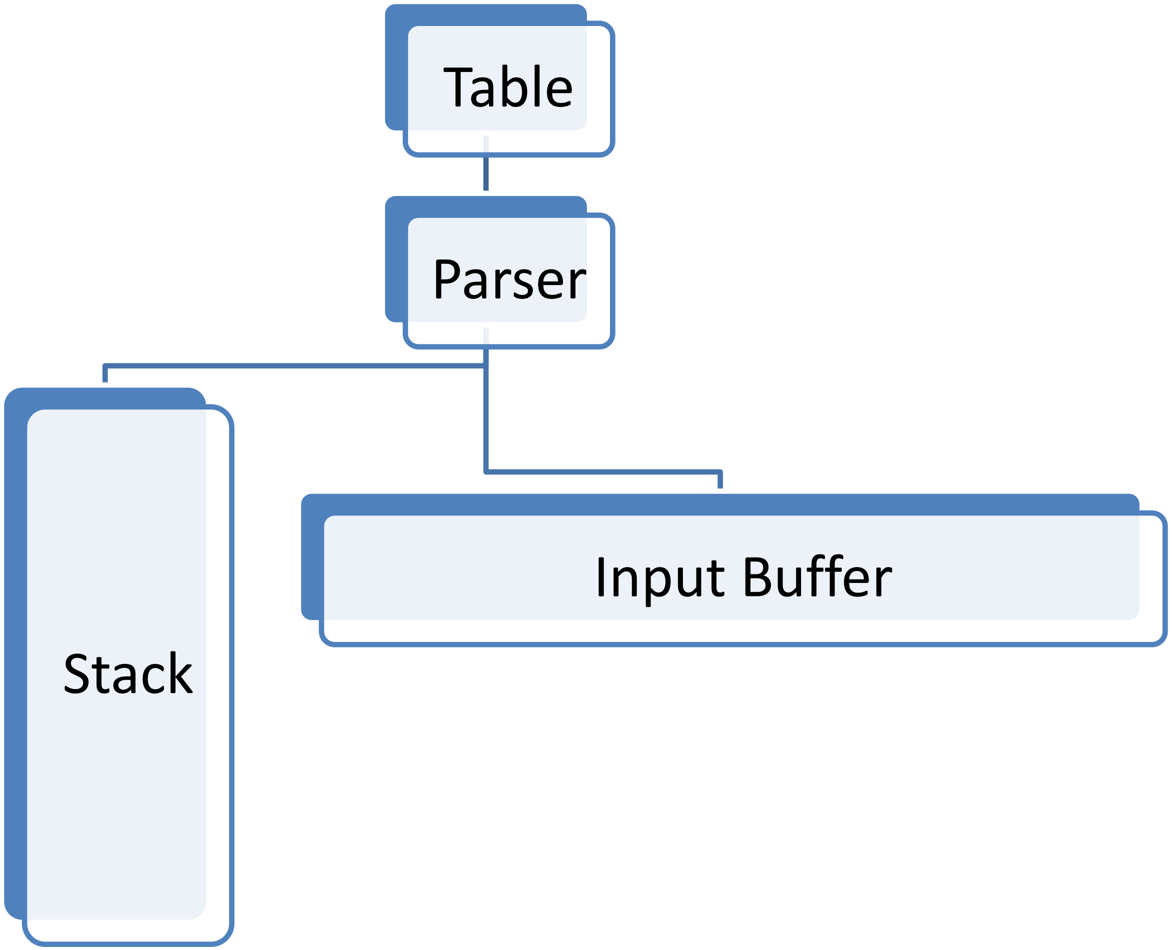
Generated Parser Table For String id + id * id

Non Terminal			INPUT SYMBOLS			
	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)		

Table driven predictive parsing

- A predictive parser can be built by maintaining a stack explicitly.
- The table driven parser has an input buffer, stack containing sequence of grammar symbols, parsing table and an output stream.
- The input buffer contains the string to be parsed followed by $.
- Initially the stack contains start symbol of the grammar on the top followed by $.
- The parsing table deterministically guesses the correct production to be used.

Flow Chart



Procedure of predictive parser

- The current symbol of the input string is maintained by a pointer say 'ip'.
- In every step consider the set {a,α} where 'a' is the symbol pointed by the 'ip' and 'α' is the top of the stack.
- If 'α' is a Non Terminal ,then see the table cell M{α,a} for the production.

1.If M{α,a} is a valid production then pop the stack , push the production into the stack.

2.If M{α,a} is error or blank then report an error

- .If ‘α’ is a terminal then pop it from the stack and also increment the input pointer ‘ip’ to point the next symbol in the input string.
- The output will be the set of productions
- The following example illustrates the top-down predictive parser using parser table .

String: id + id * id

Grammar: Mentioned LL(1) Grammar in Previous slides

MATCHED	STACK	INPUT	ACTION
	E$	id+id * id$	
	TE’$	id+id * id$	E->TE’
	FT’E’$	id+id * id$	T->FT’
	id T’E’$	id+id * id$	F->id
id	T’E’$	+id * id$	Match id
id	E’$	+id * id$	T’->Є
id	+TE’$	+id * id$	E’-> +TE’
id+	TE’$	id * id$	Match +
id+	FT’E’$	id * id$	T-> FT’
id+	idT’E’$	id * id$	F-> id
id+id	T’E’$	* id$	Match id
id+id	* FT’E’$	* id$	T’-> *FT’
id+id *	FT’E’$	id$	Match *
id+id *	idT’E’$	id$	F-> id
id+id * id	T’E’$	$	Match id
id+id * id	E’$	$	T’-> Є
id+id * id	$	$	E’-> Є

Predictive Parsing:-

Consider the grammar.

$E \rightarrow E+T / T$ —— ①

$T \rightarrow T*F / F$ —— ②

$F \rightarrow (E) / id$ —— ③

I/p string id + id * id $

Solution :-

step-1 Left-Recursion Elimination:-

The start symbol and the right symbol should be same?

$A \rightarrow A\alpha / \beta$

Rewrite as

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' / \wedge$

Eliminate LR in ①

$\underset{A}{E} \rightarrow \underset{A}{E} \underset{\alpha}{+T} / \underset{\beta}{T}$ —— ①

Rewrite as

$E \rightarrow TE'$ —— ④

$E' \rightarrow +TE' / \wedge$ —— ⑤

Scanned with CamScanner

Eliminate LR in ②

$$T \rightarrow T*F \mid F \quad — ②$$
$$\underset{A}{} \qquad \underset{A}{}\ \alpha \quad \beta$$

Rewrite as

$T \rightarrow FT'$ — ⑥

$T' \rightarrow *FT' \mid \Lambda$ — ⑦

Eliminate LR in ③

$F \rightarrow (E) \mid id$ — ③

No LR

Hence

$F \rightarrow (E) \mid id$ — ⑧

$E \rightarrow TE'$ — ④

$E' \rightarrow +TE' \mid \epsilon$ — ⑤

$T \rightarrow FT'$ — ⑥

$T' \rightarrow *FT' \mid \Lambda$ — ⑦

$F \rightarrow (E) \mid id$ — ⑧

$A \rightarrow \overset{\alpha}{\cancel{A}}B\beta$ = first

$A \rightarrow \alpha B\beta$ = foll

step-II Find First(LHS)

First occurion terminals

FIRST(E) = { (, id }

First (E') = { +, ∧ } ε

First (T) = { (, id }

First (T') = { *, ∧ }

First (F) = { (, id }

E → TE' — (4)

T → FT' — (6)

F → (E) | id — (8)

f → (E)

f → id

Step - III Finding Follow:-

$\boxed{\text{Follow}(E) = \{ \$,) \}}$

Find the expression where is in RHS

Rule

1) The start symbol should be $

2) A → αBβ (first find(β))

↑ N.T for which follow value is to be find.

3) Follow(B)

A → αB → Follow (A)

4) If first(β) = ∧ or ε then Follow(B) = Follow(A)

(8) F → (E) | id

α B β

→ first (β) = first ( ))

$\boxed{\text{Follow}(E') = \{ \$,) \}}$

(4) E → T E'

A → α B

Follow(E)

(5) E' → +TE'

A → α B

Follow (E)

$\boxed{\text{Follow}(T) = \{+, \$,)\}}$

(4) $\rightarrow E \rightarrow TE'$

$A \rightarrow B\beta$

first(β)

first(E') = $\{+, \Lambda\}$

follow(E)

(5) $\rightarrow E' \rightarrow +TE'$

$A \rightarrow \alpha B\beta$

first(E') = $\{+, \Lambda\}$

follow(E')

$\boxed{\text{Follow}(T') = \{+, \$,)\}}$

(6) $\Rightarrow T \rightarrow FT$

$A \rightarrow \alpha B$

Follow(T)

(7) $\Rightarrow T' \rightarrow *FT'$

$A \rightarrow \alpha B$

Follow(T')

Scanned with CamScanner

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

⑥ ⇒ $T \rightarrow F T'$

$A \rightarrow B \beta$

first $(T') = \{*, \wedge\}$

Follow(T)

⑦ ⇒ $T' \rightarrow * F T'$

$\alpha B \beta$

first $(T') = \{* \wedge\}$

Follow (T')

Follow(E) = {$,)}

Follow (E') = {$,)}

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

Follow (T') = {+, $,)}

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

step4:- Generate parsing table

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

Stack	I/P	Action
$E	id+id*id$	E → TE'
$E'T	id+id*id$	T → FT'
$E'T'F	id+id*id$	F → id
$E'T'id	id+id*id$	pop id and increment ptr
$E'T'	+id*id$	T' → λ
$E'	+id*id$	E' → +TE'
$E'T'+	+id*id$	pop + increment ptr
$E'T'	id*id$	T' → *FT'
$E'T'F*	*id$	pop * ++ptr
$E'T'F	id$	F → id
$E'T'id	id$	pop id, ++ ptr
$E'T'	$	T' → λ
$E'	$	E' → λ
$	$	—