Predictive LL(1) Parser

- This top-down parsing algorithm is of non-recursive type.
- In this type of parsing a table is built.
- For LL(1) - the first L means the input is scanned from left to right. The second L means it uses leftmost derivation for input string.
And the number 1 in the input symbol means it uses only one input symbol (lookahead) to predict the parsing process.

- The simple block diagram for LL(1) parser is given below.

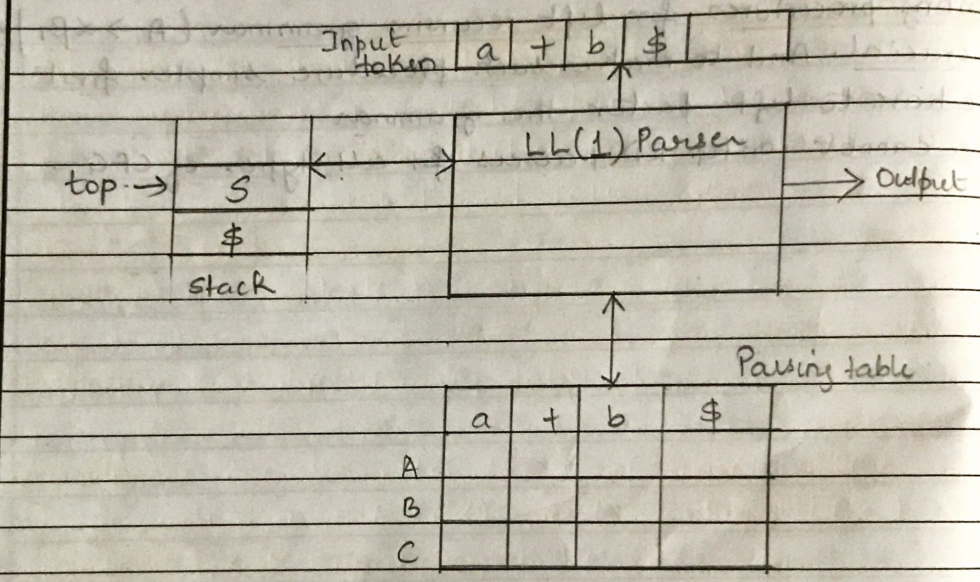

fig: Model for LL(1) Parser.

- The data structures used by LL(1) are
  i) the input buffer.
  ii) the stack
  iii) Parsing table.
- The LL(1) parser uses input buffer to store the input tokens.
- The stack is used to hold the left sentential form.
- The symbols in RHS of rule are pushed into the stack

is reverse order ie from right to left.

- Thus use of stack makes this algorithm non-recursive.
- The table is basically a 2-D array.
- The table has row for NT and column for Terminals.
- The table can be represented as M[A,a] where A is a NT and a is current input symbol.
- The parser works as follows :-
- The parsing program reads top of the stack and a current input symbol. With the help of these two symbols the parsing action is determined. The parsing actions can be

Top	Input token	Parsing action
$	$	Parsing succesful, halt.
a	a	Pop a and advance lookahead to next token
a	B	Error.
A	a	Refer table M[A,a] if entry at M[A,a] = error report Error
A	a	Refer table M[A,A] if entry at M[A,a] is A → PQR then pop A then push R, then push Q, then push P.

- The parser consults the table M[A,a] each time while taking the parsing actions hence this type of parsing method is called table driven parsing algorithm.
- The configuration of LL(1) parser can be defined by top of the stack and a lookahead token.
- One by one configuration is performed and the input is succesfully parsed if the parser reaches the halting configuration. When the stack is empty and next token is $ then it corresponds to succesful parse.

Construction of Predictive LL(1) Parser

- The construction of predictive LL(1) parser is based on two very important functions and those are FIRST and FOLLOW.
- For construction of Predictive LL(1) parser we have to follow the following steps:

1) Computation of FIRST and FOLLOW function.
2) Construct the Predictive Parsing table using FIRST and FOLLOW functions.
3) Parse the input string with the help of Predictive Parsing table.

Algorithm for Predictive Parsing table :-

- The construction of predictive parsing table is an important activity in predictive parsing method. This algorithm requires FIRST and FOLLOW functions.

Input : The context free Grammar G

Output : Predictive Parsing table M

Algorithm : For the rule $A \to \alpha$ of grammar G

1) For each a in FIRST(α) create entry $M[A,a] = A \to \alpha$ where a is terminal symbol.
2) For ϵ in FIRST(α) create entry $M[A,b] = A \to \alpha$ where b is the symbols from FOLLOW(A).
3) If ϵ is in FIRST(α) and $ is in FOLLOW(A) then create entry in the table $M[A,\$] = A \to \alpha$
4) All the remaining entries in the table M are marked as SYNTAX ERROR.

NOTE :- A grammar whose parsing table has no multiply-defined entries is said to be LL(1).

eg

Consider the grammar:

$E \to TE'$

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

$T \to FT'$

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

$F \to (E) \mid id$

FIRST(E) = { (}
FIRST(E') = { }
FIRST(T) = { (, id }
FIRST(T') = { }
FIRST(F) = { (, id }

FOLLOW(E) = { }
FOLLOW(E') = {), $ }
FOLLOW(T) = { }
FOLLOW(T') = { +,), $ }
FOLLOW(F) = { }

Now create the table as follows:

	+	*	(	)	id	$
E	Error	Error	$E \to TE'$	Error	$E \to TE'$	Error
E'	$E' \to +TE'$	Error	Error	$E' \to \epsilon$	Error	$E' \to \epsilon$
T	Error	Error	$T \to FT'$	Error	$T \to FT'$	Error
T'	$T' \to \epsilon$	$T' \to *FT'$	Error	$T' \to \epsilon$	Error	$T' \to \epsilon$
F	Error	Error	$F \to (E)$	Error	$F \to id$	Error

Now we will ~~map the~~ fill up the entries in the table using the above given algorithm. Consider each rule one by one.

$E \to TE'$

$A \to \alpha$

$A = E, \alpha = TE'$

FIRST(TE') if $E' = \epsilon$ then FIRST(T) = { (, id }

entry M[E, (] = $E \to TE'$

entry M[E, id] = $E \to TE'$

$E' \to +TE'$

$A \to \alpha$

$A = E', \alpha = +TE'$

FIRST(+TE') = {+}
entry M[E', +] = E' → +TE'

E' → ϵ
A → α
A = E', α = ϵ then.
FOLLOW(E') = {), $}
M[E',)] = E' → ϵ
M[E', $] = E' → ε

T → FT'
A → α
A = T, α = FT'
FIRST(FT') = FIRST(F) = {(, id}.
M[T, (] = T → FT'
M[T, id] = T → FT'

T' → * FT'
A → α
A = T', α = * FT'
FIRST(* FT') = {*}
M[T, *] = T' → * FT'

T' → ϵ
A → α
A = T', α = ϵ
FOLLOW(T') = {+,), $}
M[T', +] = T' → ϵ
M[T',)] = T' → ϵ
M[T', $] = T' → ϵ

F → (E)
A → α
A = F, α = (E)
FIRST((E)) = {(}
M[F, (] = F → (E)

$F \to id$

$A \to \alpha$

$A = F, \alpha = id$

$FIRST(id) = \{id\}$

$M[F, id] = F \to id$

Now the input string id + id * id $ can be parsed using above table. At the initial configuration the stack will contain start symbol E, in the input buffer input string is placed.

Stack	Input	Action
$ E	id + id * id $	

Now symbol E is at top of the stack and input pointer is at first id, hence M[E, id] is refered. This entry tells us $E \to TE'$, so we will push E' first then T

Stack	Input	Action
$ E	id + id * id $	$M[E, id] = E \to TE'$ pop E, Push E', Push T
$ E'T	id + id * id $	$M[T, id] = T \to FT'$ pop T, Push T', Push F
$ E'T'F	id + id * id $	$M[F, id] = F \to id$ pop F push id
$ E'T'id	id + id * id $	$M[id, +] =$ Pops id off the stack & advances the i/p ptr to next i/p symbol.
$ E'T'	+ id * id $	$M[T', +] = T' \to \epsilon$
$ E'	+ id * id $	$M[E', +] = E' \to +TE'$
$ E'T+	+ id * id $	
$ E'T	id * id $	$M[T, id] = T \to FT'$
$ E'T'F	id * id $	$M[F, id] = F \to id$
$ E'T'id	id * id $	
$ E'T'	* id $	$M[T', *] = T' \to *FT'$
$ E'T'F*	* id $	
$ E'T'F	id $	$M[F, id] = F \to id$
$ E'T'id	id $	
$ E'T'	$	$M[T', \$] = T' \to \epsilon$
$ E'	$	$M[E', \$] = E' \to \epsilon$
$	$	Successful.