Parsing

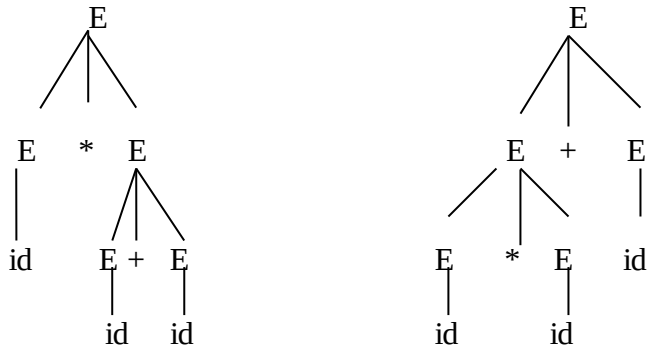
Top-Down parsing (non recursive predictive parsing) ; can be constructed automatically from class of grammars called LL(1) grammar. (the first L stands for scanning the input from left to right, the second L for producing a leftmost derivation, 1 for using one input symbol of look ahead at each step to make parsing action decisions.
LL(1) grammar has the following conditions:
A- Unambiguous B- No left-recursion C- No factoring

A- Unambiguous: By ambiguous grammar we mean it can have more than one parse tree generating a given string of tokens. Since a string with more than one parse tree usually has more than one meaning, for compiling applications we need to design unambiguous grammars.

Ex: G1:

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

Parse id * id + id

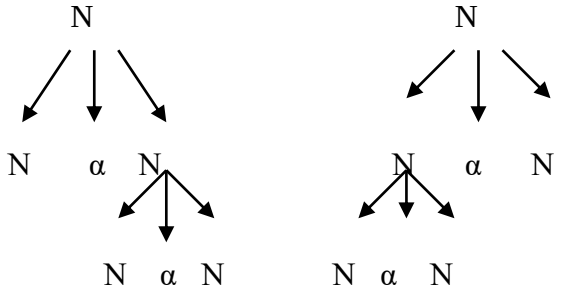


Ambiguous grammar may be detected if it has the following form:

i- $N \rightarrow N\ \alpha\ N$ where α is a string of terminals or non-terminals

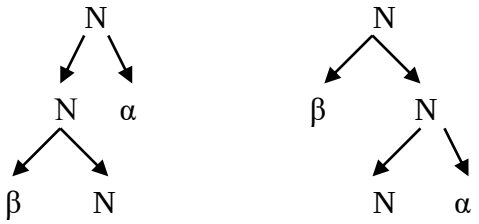
left & right recursive

EX: N α N α N



ii- $N \rightarrow N\ \alpha \mid \beta\ N$

EX: β N α

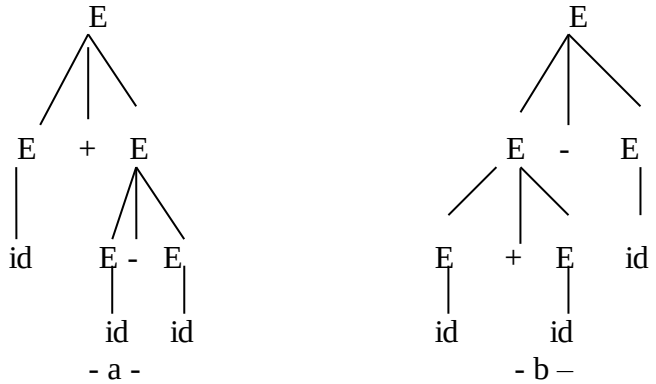


However, a simple set of sufficient conditions can be developed such that when they are applied to a grammar, then the grammar is guaranteed to be unambiguous (disambiguating rules).

i- Associativity of operations

E → E + E | E - E | id ambiguous

id + id - id



- a - - b –

Since (+,-) are left associative, parse tree in (b) is more sufficient than tree in (a) (b-grows down towards the left).

(+,-) left ; (*,/) left ; (=,^) right

E → E + F | E - F | F

F → id unambiguous

Ex:

E → E ^ E | id ambiguous

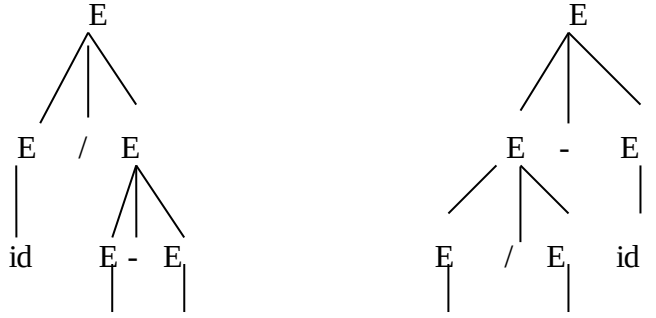
E → F ^ E | F

F → id unambiguous

ii- Operator Precedence

E → E - E | E / E | id ambiguous

id / id - id



id id id id

- a - - b –

$E \rightarrow E - T \mid T$
$T \rightarrow T / F \mid F$
$F \rightarrow id$

Ex:
Write an unambiguous grammar for the language of parenthesized arithmetic expressions containing (+,-,*,/,(), unary -).

Ex:
Write an unambiguous grammar for the language of parenthesized logical expressions containing

~	(not)	Highest
&	(and)	
|	(or)	Lowest

Sol: $E \rightarrow E$ or $T \mid T$
$T \rightarrow T$ and $F \mid F$
$F \rightarrow id \mid (E) \mid$ not E

B. Elimination of Left recursion:
A grammar is said to be left recursive if it has a non terminal A such that there is a derivation $A \overset{+}{\rightarrow} A\alpha$ for some string α. From + sign we can notice that there is two types of left recursion, either immediate or non-immediate. The below algorithm will systematically eliminate left recursion from a grammar.

Hint:-
The grammar must have no cycles ($A \rightarrow A$), or **ε**-productions, it may contains harmless **ε**-production.

Algorithm: Eliminating left recursion
Input: G with no cycles or ε-productions.
Output: G with no left recursions (may contain ε-production)

1. Arrange the nonterminals in some order $A_1, A_2, \ldots, A_n$.
2. For i=1 to n do begin
   For j=1 to i-1 do begin
   (2.1) replace each production $A_i \rightarrow A_j \gamma$, by

   $A_i \rightarrow \delta_1 \gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$
   $A_i \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$
   end
   (2.2) eliminate immediate left recursion among the A_i - Productions
   $A_i \rightarrow A_i \alpha_1 \mid A_i \alpha_2 \mid \ldots\ldots \mid A_i \alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots. \mid \beta_n$
   to be replaced by

$A_i \rightarrow \beta_1 A_i' \mid \beta_2 A_i' \mid \ldots\ldots \mid \beta_n A_i'$
$A_i' \rightarrow \alpha_1 A_i' \mid \alpha_2 A_i' \mid \ldots. \mid \alpha_m A_i' \mid \varepsilon$
end.

Ex1:-

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid id$

Sol:-

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow F T'$
$T' \rightarrow * F T' \mid \varepsilon$
$F \rightarrow (E) \mid id$

Ex2:-

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid \varepsilon$

جب

There is no immediate left recursion among the S-productions,so nothing happens during step (2) for the case i=1.
For i=2, we substitute the S-productions in $A \rightarrow Sd$ to pobtain the following A-productions.

$A \rightarrow Ac \mid Aad \mid bd \mid \varepsilon$

Eliminating left recursion yields the following grammar:

$S \rightarrow Aa \mid b$
$A \rightarrow bdA \mid A'$
$A' \rightarrow cA' \mid adA' \mid \varepsilon$

C- Left Factoring: it is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

Algorithm: left factoring a grammar
Input: Grammar G
Output: left-factored grammar
Method: for each nonterminal A

1. find the longest prefix α common to two or more of its alternatives.
2. if $\alpha \neq \varepsilon$ then replace

$A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots\ldots \mid \alpha \beta_n \mid \gamma$ where γ represents all alternatives that do not begin with α

By $A \rightarrow \alpha A' \mid \gamma$
$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots\ldots \mid \beta_n$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

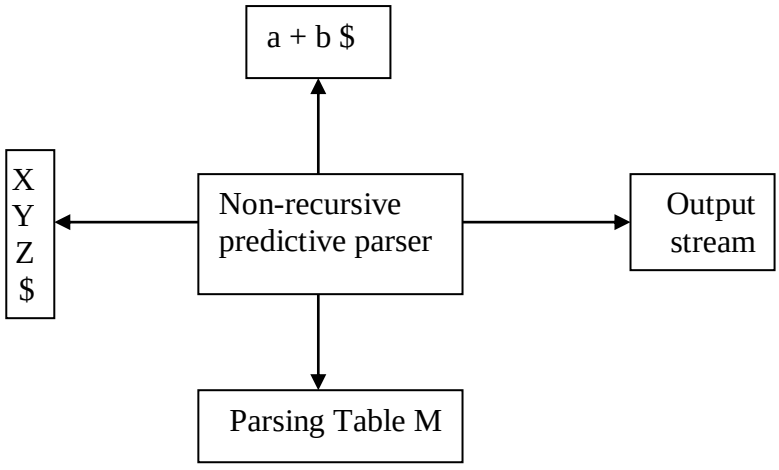
Ex:-

$S \rightarrow iEtS \mid iEtSeS \mid a$
$E \rightarrow b$

Sol:

S → iEtSS' | a
S' → eS | ε
E → b

Non-recursive predictive parser:-
It is possible to write a predictive parser using a stack, rather than recursive calls. The non-recursive parser looks up a production to be applied in a parsing table. The parser has an input buffer, a stack, a parsing table and the output stream.
The input buffer contains the string to be parsed followed by $ (right most end marker). The stack contains a sequence of grammar symbols with $ (on the bottom of stack). Initially the stack contains S $ where S is the start symbol of grammar.
The parsing table is a 2-dimensional array M[A,a] where a is a nonterminal symbol and a is a terminal or '$', contains the production to be applied. The output consists of the production rules that have been applied. The parser behaves as follows:



Let X be the top element of stack. Let a be the current input symbol.
1. if X=a=$ then a successful parse.
2. if X=a≠$ then pop the top element of stack and advance the input pointer to the next input symbol.
3. if X is a non-terminal then consult the entry M[X,a] of the parsing table M. if the entry contains an X production of the form X → UVW, then the parser replace X by WVU where U on top.
However, the M[X,a] entry might not contain an X production, and in which case there is an error and the parser calls the required error routine to recover.

The algorithm (Non-recursive predictive parsing):-
A string W and parsing table M for grammar G are input. if W is in L(G), then the left most derivation of W is output, otherwise an error indication. Set ip (input pointer) to point to first symbol of W$

Repeat
Let X be the top symbol of stack and a be the symbol pointed to by ip.
If X=a then pop X from stack and advance ip
else error ( )
else /* X is non-terminal */

if M[X,a] = $X \rightarrow Y_1 Y_2 \ldots\ldots Y_K$
then begin
pop X from top of stack
push $Y_k, Y_{k-1}, \ldots, Y_1$ on to stack with Y_1 on top output the production $X \rightarrow Y_1 Y_2 \ldots Y_K$
end
else error ( )
until X = $ /* stack is empty */

ex:-Consider the grammar

$E \rightarrow TE'$
$E' \rightarrow +TE' \mid \varepsilon$
$T \rightarrow F T'$
$T' \rightarrow * F T' \mid \varepsilon$
$F \rightarrow (E) \mid id$

N.T.	Input-symbol					
	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)		

Parsing table M for grammar

❖ Suppose input id + id * id , steps of predictive parsing of the input is:-

Stack	Input buffer	output
$E	id + id * id $	
$ E' T	id + id * id $	$E \rightarrow TE'$
$ E' T' F	id + id * id $	$T \rightarrow F T'$
$ E' T' id	id + id * id $	$F \rightarrow id$
$ E' T'	+ id * id $	Pop id
$ E'	+ id * id $	$T' \rightarrow \varepsilon$

First and Follow functions:
Two functions allow us to fill in the entries of a predictive parsing table for LL(1) grammar.