$$
\begin{array}{lcl}
E & \rightarrow & E + T \mid E - T \mid T \\
T & \rightarrow & T * F \mid T / F \mid F \\
F & \rightarrow & (E) \mid \mathbf{id}
\end{array}
$$

Grammar 4.2

2.4.5 Left Recursion

It is possible for a recursive-descent parser to loop forever. A problem arises with "left-recursive" productions like

$$expr \rightarrow expr + term$$

where the leftmost symbol of the body is the same as the nonterminal at the head of the production. Suppose the procedure for *expr* decides to apply this production. The body begins with *expr* so the procedure for *expr* is called recursively. Since the lookahead symbol changes only when a terminal in the body is matched, no change to the input took place between recursive calls of *expr*. As a result, the second call to *expr* does exactly what the first call did, which means a third call to *expr*, and so on, forever.

A left-recursive production can be eliminated by rewriting the offending production. Consider a nonterminal A with two productions

$$A \rightarrow A\alpha \mid \beta$$

where α and β are sequences of terminals and nonterminals that do not start with A. For example, in

$$expr \rightarrow expr + term \mid term$$

nonterminal $A = expr$, string $\alpha = + term$, and string $\beta = term$.

The nonterminal A and its production are said to be *left recursive*, because the production $A \rightarrow A\alpha$ has A itself as the leftmost symbol on the right side.[4] Repeated application of this production builds up a sequence of α's to the right of A, as in Fig. 2.20(a). When A is finally replaced by β, we have a β followed by a sequence of zero or more α's.

The same effect can be achieved, as in Fig. 2.20(b), by rewriting the productions for A in the following manner, using a new nonterminal R:

[4] In a general left-recursive grammar, instead of a production $A \rightarrow A\alpha$, the nonterminal A may derive $A\alpha$ through intermediate productions.

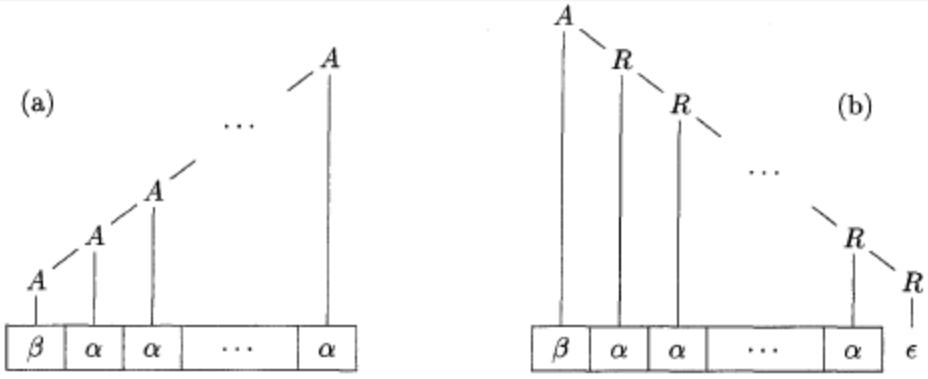


Figure 2.20: Left- and right-recursive ways of generating a string

$$\begin{aligned} A &\rightarrow \beta R \\ R &\rightarrow \alpha R \mid \epsilon \end{aligned}$$

Nonterminal R and its production $R \rightarrow \alpha R$ are *right recursive* because this production for R has R itself as the last symbol on the right side. Right-recursive productions lead to trees that grow down towards the right, as in Fig. 2.20(b). Trees growing down to the right make it harder to translate expressions containing left-associative operators, such as minus. In Section 2.5.2, however, we shall see that the proper translation of expressions into postfix notation can still be attained by a careful design of the translation scheme.

In Section 4.3.3, we shall consider more general forms of left recursion and show how all left recursion can be eliminated from a grammar.

4.3.3 Elimination of Left Recursion

A grammar is *left recursive* if it has a nonterminal A such that there is a derivation $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α. Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion. In Section 2.4.5, we discussed *immediate left recursion*, where there is a production of the form $A \rightarrow A\alpha$. Here, we study the general case. In Section 2.4.5, we showed how the left-recursive pair of productions $A \rightarrow A\alpha \mid \beta$ could be replaced by the non-left-recursive productions:

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \epsilon \end{aligned}$$

without changing the strings derivable from A. This rule by itself suffices for many grammars.

Example 4.17: The non-left-recursive expression grammar (4.2), repeated here,

$$\begin{aligned} E &\rightarrow T\ E' \\ E' &\rightarrow +\ T\ E' \\ T &\rightarrow F\ T' \\ T' &\rightarrow *\ F\ T' \\ F &\rightarrow (\ E\ ) \mid \mathbf{id} \end{aligned}$$

is obtained by eliminating immediate left recursion from the expression grammar (4.1). The left-recursive pair of productions $E \rightarrow E + T \mid T$ are replaced by $E \rightarrow T\ E'$ and $E' \rightarrow +\ T\ E' \mid \epsilon$. The new productions for T and T' are obtained similarly by eliminating immediate left recursion. □

Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions. First, group the productions as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

The nonterminal A generates the same strings as before but is no longer left recursive. This procedure eliminates all left recursion from the A and A' productions (provided no α_i is ϵ), but it does not eliminate left recursion involving derivations of two or more steps. For example, consider the grammar

Context free grammar:

$G = \{V, T, P, S\}$

V = set of non terminals / variables

T = " " terminals / alphabets

P = set of production rules

S = the start symbol.

Example1:

1. $A \rightarrow XYZ$
2. $X \rightarrow a$
3. $Y \rightarrow b$
4. $Z \rightarrow \epsilon$

$V = \{A, X, Y, Z\}$

$T = \{a, b, \epsilon\}$

$P = \{1, 2, 3, 4\}$

$S = A$

Drawing a parse tree on a CFG:

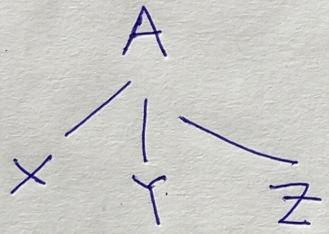
* Replace the head with its production rule
* Continues until match found

If I want to draw the parse tree over the grammar of Example 1 based on input "ab", then,

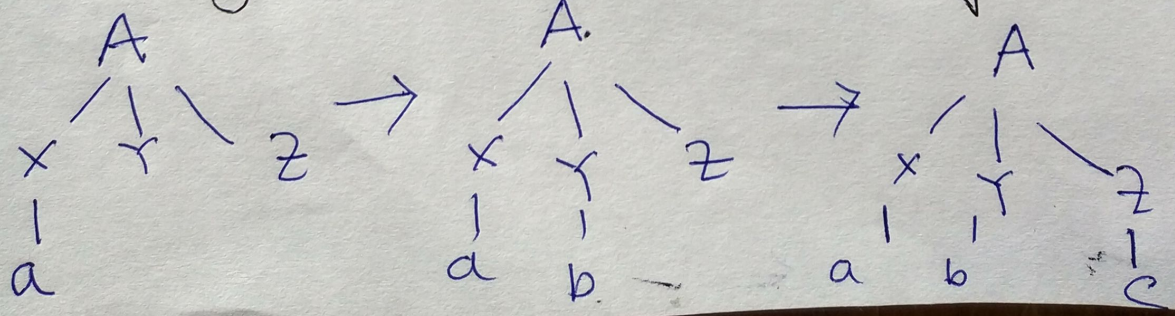
1. Start with the start symbol:

A

2. Replace it with its production rules:



3. Do the same for the new nonterminals then generated in a left to right approach



How a compiler processes parse trees based on tokens :-

Suppose, we have a programming language, where the methods declared may be non-parameterised i.e. don't take arguments at all, but if they do they only take integer argument(s). of range (0-9)

Example: max(2,3), toString(3) are all methods here

The CFG would be

1. call → id (optparams)
2. optparams → params | ε
3. params → params, param | param
4. param → 0 | 1 | 2 | ... | 9

V = {call, optparams, params, param}

T = {0, 1, ..., 9, id}

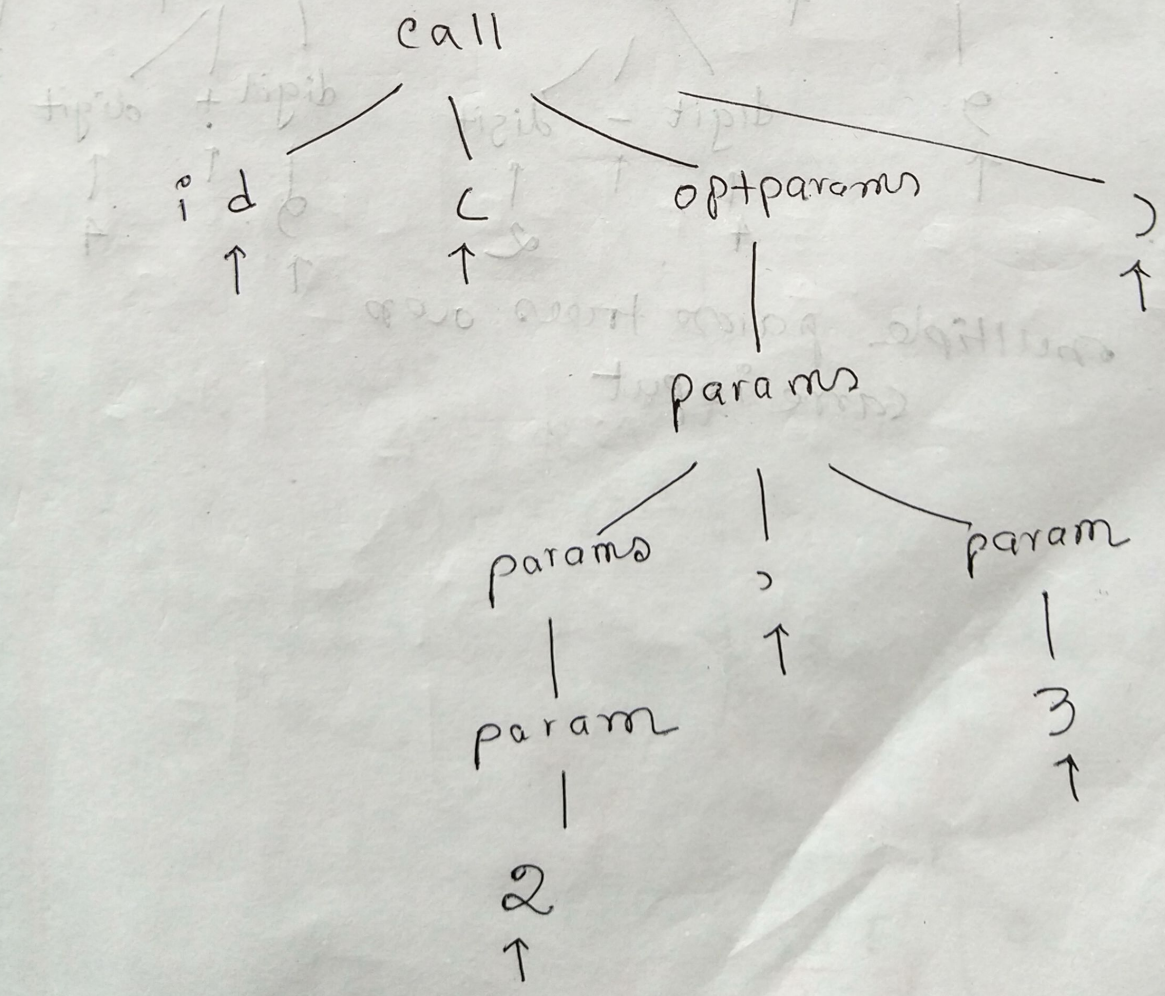
P = {1, 2, 3, 4}, S = {call}.

If I want to process "max(2,3)" as input, the following tokens will be generated by lexical analyzer:

i/p: max (2 , 3)

tokens: <id, 0> <(> <2, 1> <,> <3, 2> <)>

<0 → name, 0 → pointer; 1 → pointer>



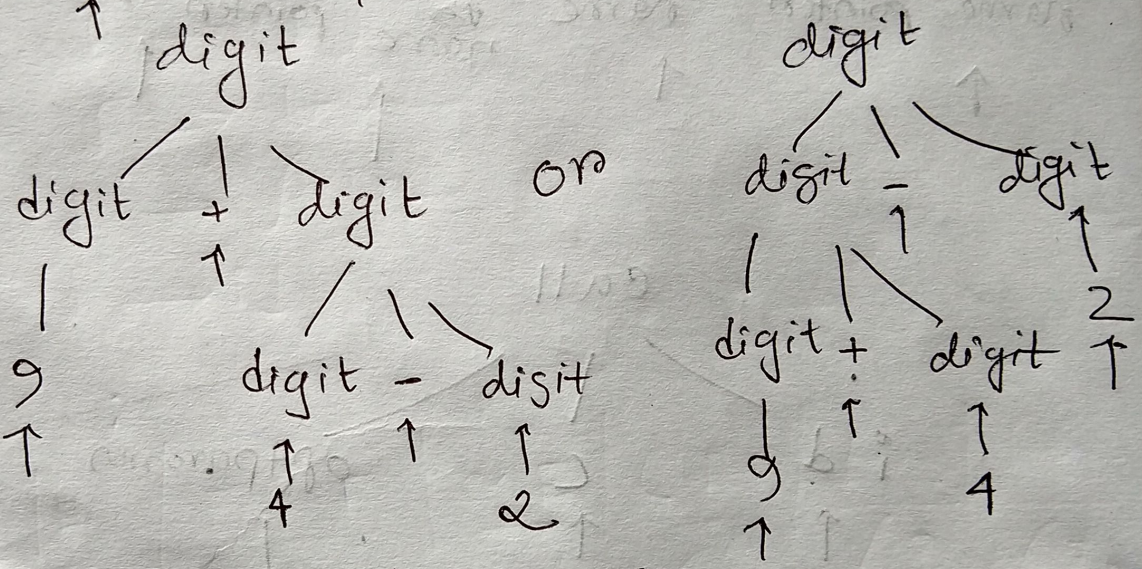
Ambiguity:

digit → digit + digit | digit − digit | d --- 19

Given expression:

9 + 4 − 2

<9, 0> <+> <4, 1> <−> <2, 2>



multiple parse trees over same input

Left Recursion

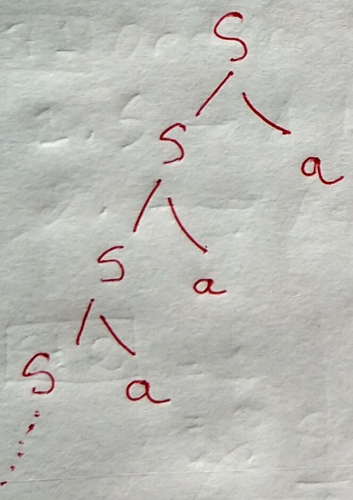
A grammar is called ~~left~~ left recursive if it can be represented in the form:

$$A \overset{+}{\Rightarrow} A\alpha$$

Immediate left recursion

Example:

$S \rightarrow Sa \mid b$



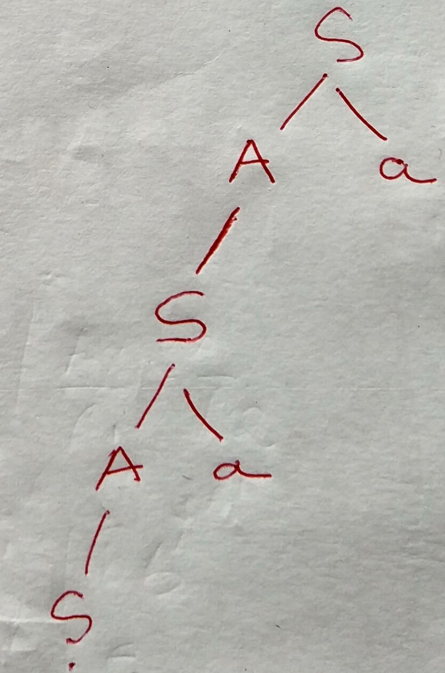
infinite loop (for input ba)

Non immediate left recursion

Example:

$S \rightarrow Aa \mid b$

$A \rightarrow S$



infinite loop (for input ba)

Immediate left Recursion Removal

Example 1: Remove immediate left recursion from the following grammar:

$E \rightarrow E \mid E$

$E \rightarrow E \& E$

$E \rightarrow \sim E$

$E \rightarrow (E)$

$E \rightarrow true$

$E \rightarrow false$

$E \rightarrow id$

Solution:

(if $A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid \beta_1 \mid \beta_2 \mid \cdots$

$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \cdots$

$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \epsilon$)

$$\underset{A}{E} \rightarrow \underset{A}{E} \underset{\alpha_1}{|} \underset{A}{E} \underset{\alpha_2}{\&} \underset{A}{E} \mid \sim \underset{\beta_1}{E} \mid (\underset{\beta_2}{E}) \mid \underset{\beta_3}{true} \mid \underset{\beta_4}{false} \mid \underset{\beta_5}{id}$$

So,

$$\underset{A}{E} \rightarrow \sim \underset{\beta_1}{E}\ \underset{A'}{E'} \mid (\underset{\beta_2}{E})\ \underset{A'}{E'} \mid \underset{\beta_3}{true}\ \underset{A'}{E'} \mid \underset{\beta_4}{false}\ \underset{A'}{E'} \mid \underset{\beta_5}{id}\ \underset{A'}{E'}$$

$$\underset{A'}{E'} \rightarrow \underset{\alpha_1}{|E}\ \underset{A'}{E'} \mid \underset{\alpha_2}{\&E}\ \underset{A'}{E'} \mid \underset{\epsilon}{\epsilon}$$