7. 1. Parse Trees and Derivations

A parse tree is a graphical representation of a derivation that filters out the order in which productions are applied to replace non terminals. Each interior node of a parse tree represents the application of a production. The interior node is labeled with the onto terminal A in the head of the production; the children of the node are labeled, from left to right, by the symbols in the body of the production by which this A was replaced during the derivation.

For example, the parse tree for -(id + id) in Fig. 7.1, results from the derivation (4.8) as well as derivation (4.9). The leaves of a parse tree are labeled by non-terminals or terminals and, read from left to right, constitute a sentential form, called the *yield* or *frontier* of the tree. To see the relationship between derivations and parse trees.

BASIS: The tree for a1 = A is a single node labeled A.

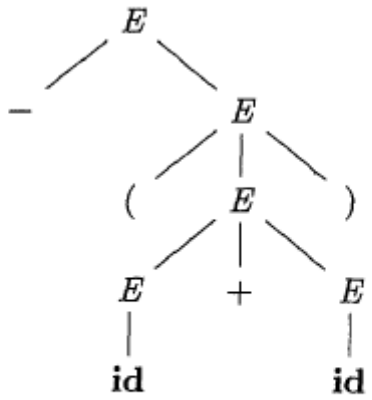


Figure 7.1: Parse tree for $-(\mathbf{id} + \mathbf{id})$

In what follows, we shall frequently parse by producing a leftmost or a rightmost derivation, since there is a one-to-one relationship between parse trees and either leftmost or rightmost derivations. Both leftmost and rightmost derivations pick a particular order for replacing symbols in sentential forms, so they too filter out variations in the order.

It is not hard to show that every parse tree has associated with it a unique leftmost and a unique rightmost derivation.

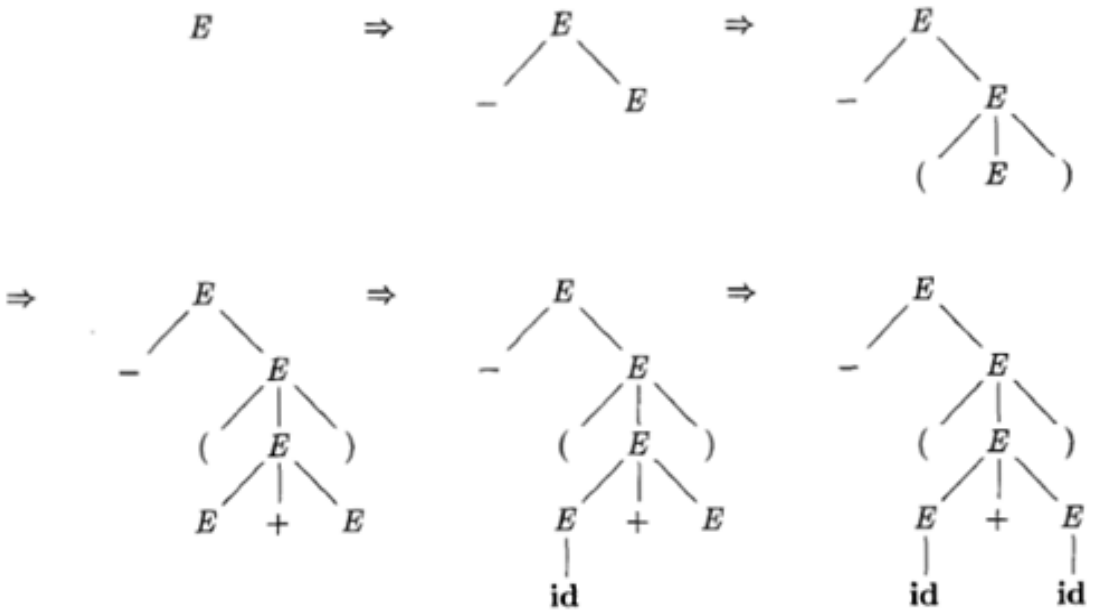


Figure 7.2. Sequence of parse tree for derivation (4.8)

7.2.Ambiguity

a grammar that produces more than one parse tree for some sentence is said to be *ambiguous.* Put another way, an ambiguous grammar is one that produces more than one leftmost derivation or more than one rightmost derivation for the same sentence.

Example 1 : The arithmetic expression grammar (4.3) permits two distinct leftmost derivations for the sentence id + id * id:

E ⟶ E+E

E ⟶ E*E

E ⟶ id

$$
\begin{aligned}
E &\Rightarrow E+E \\
&\Rightarrow \mathbf{id}+E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
\qquad
\begin{aligned}
E &\Rightarrow E*E \\
&\Rightarrow E+E*E \\
&\Rightarrow \mathbf{id}+E*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*E \\
&\Rightarrow \mathbf{id}+\mathbf{id}*\mathbf{id}
\end{aligned}
$$

Parse tree for id+id*id

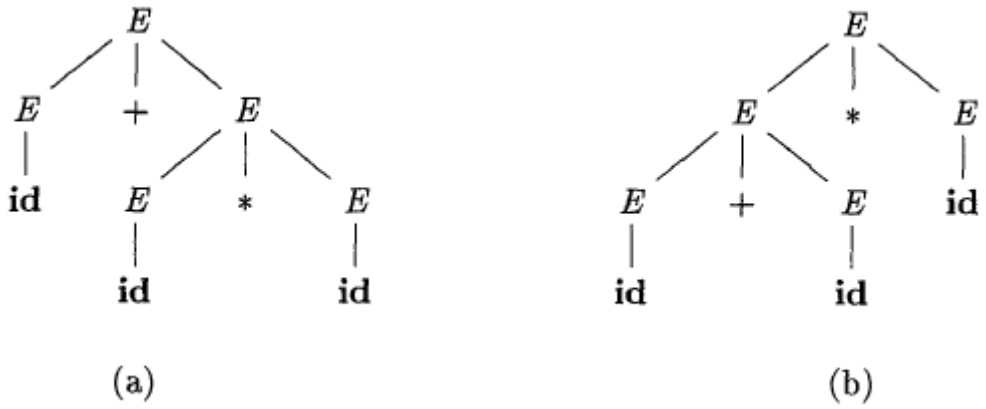
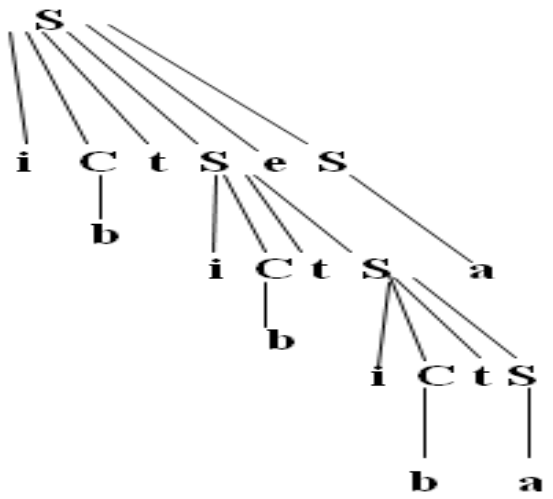


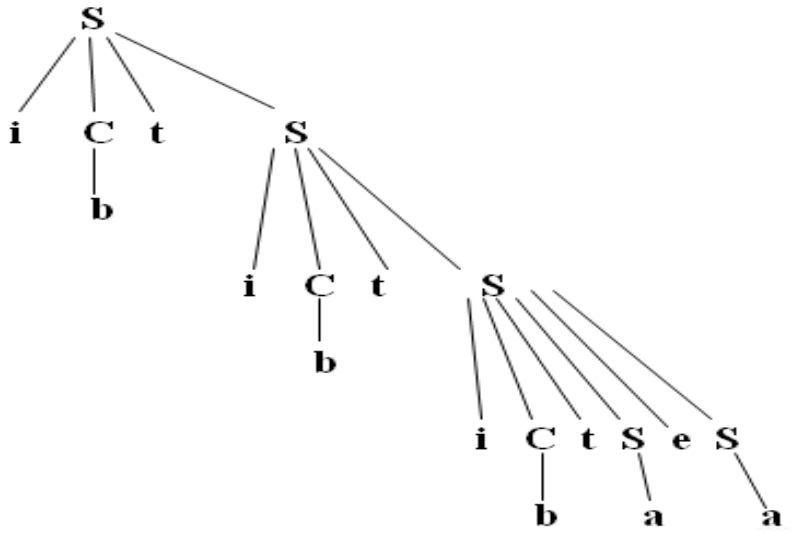
Figure 7.3. Two parse tree for id+id*id

Example 2: Check whether the given grammar is ambiguous or not

S ⟶ iCtS
S ⟶ iCtSeS
S ⟶ a
C ⟶ b



Or



Thus we have got more than two parse tree. Hence the given grammar is ambiguous.

7.3. Writing a Grammar

Grammars are capable of describing most, but not all, of the syntax of programming languages. For instance, the requirement that identifiers be declared before they are used, cannot be described by a context-free grammar. Therefore, the sequences of tokens accepted by a parser form a superset of the programming language; subsequent phases of the compiler must analyze the output of the parser to ensure compliance with rules that are not checked by the parser. This section begins with a discussion of how to divide work between a lexical analyzer and a parser. We then consider several transformations that could be applied to get a grammar more suitable for parsing. One technique can eliminate ambiguity in the grammar, and other techniques - left-recursion elimination and left factoring - are useful for rewriting grammars so they become suitable for top-down parsing.

We conclude this section by considering some programming language constructs that cannot be described by any grammar.

7.4. Top down parser

In this section there are basic ideas behind top-down parsing and show how constructs an efficient non- backtracking form of top-down parser called a predictive parser. Top down parsing can be viewed as attempt to find a left most derivation for an input string. Equivalently, it can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.The following grammar requires **backtracking**:

Example 1 : The sequence of parse trees in Fig. 7.4 for the input **id+id*id** is a top-down parse according to grammar (4.2), repeated here:

$E \longrightarrow TE'$
$E' \longrightarrow +TE1 \backslash \ \varepsilon$
$T \longrightarrow FT'$
$T' \longrightarrow *FT \backslash \mathcal{E}$
$F \longrightarrow (E) \backslash$ **id**

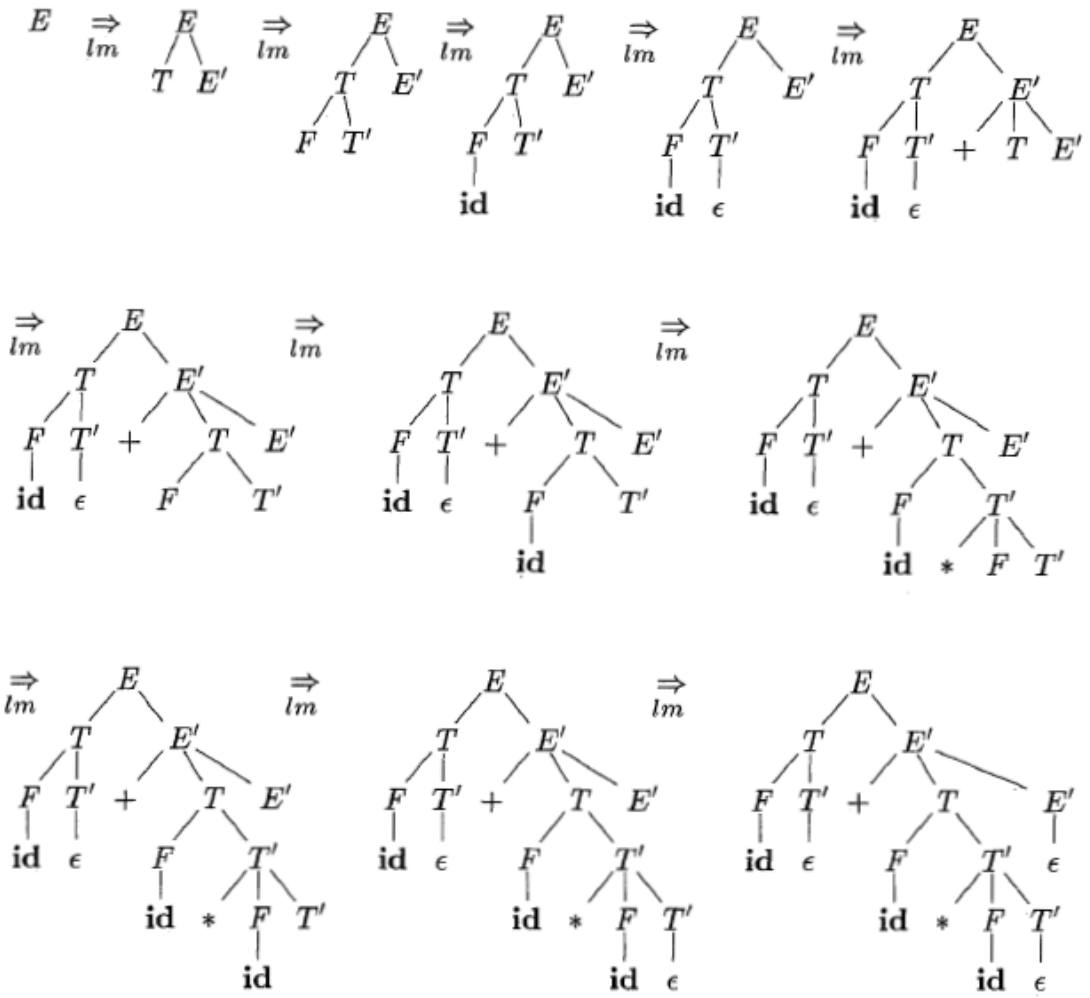


Figure 7.4: Top-down parse for **id + id * id**

7.5. Predictive Parsing Method

In many cases, by carefully writing a grammar eliminating left recursion from it, and left factoring the resulting grammar, we can obtain a grammar that can be parsed by a non backtracking predictive parser.

We can build a predictive parser by maintaining a stack. The key problem during predictive parser is that of determining the production to be applied for a nonterminal. The non recursive parser looks up the production to be applied in a parsing table.

A table-driven predictive parser has an input buffer, a stack, a parsing table, and an output stream.

The input buffer contains the string to be parsed, followed by $, (a symbol used as a right end marker to indicate the end of the input string). The stack contains a sequence of grammar symbols with $ on the bottom,(indicating the bottom of the stack). Initially, the stack contains the start symbol of the grammar on the top of $. The parsing table is a two-dimensional array M[A,a], where A is a nonterminal, and **a** is a terminal or the symbol $.

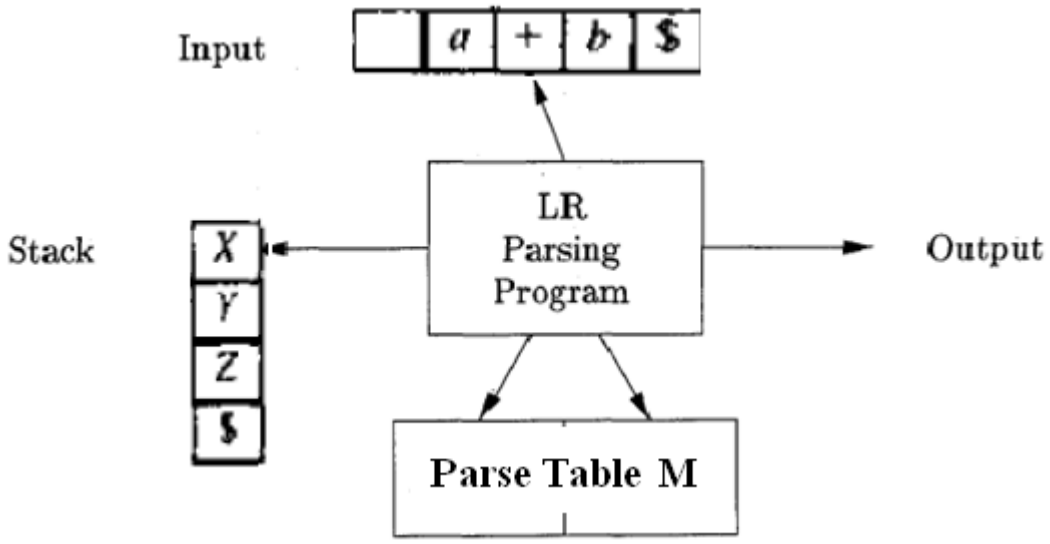


Figure 7.5.: Model of an LR parser

Example: Parse the input **id * id + id** in the grammar:

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

1- The parse table M for the grammar:

NONTER-MINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to id$			$F \to (E)$		

2- The moves made by predictive parser on input id+id*id

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	$E \to TE'$
$E'T'F	id + id * id$	$T \to FT'$
$E'T'id	id + id * id$	$F \to id$
$E'T'	+ id * id$	
$E'	+ id * id$	$T' \to \epsilon$
$E'T+	+ id * id$	$E' \to +TE'$
$E'T	id * id$	
$E'T'F	id * id$	$T \to FT'$
$E'T'id	id * id$	$F \to id$
$E'T'	* id$	
$E'T'F*	* id$	$T' \to *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \to id$
$E'T'	$	
$E'	$	$T' \to \epsilon$
$	$	$E' \to \epsilon$