This can be derived by the following ways.

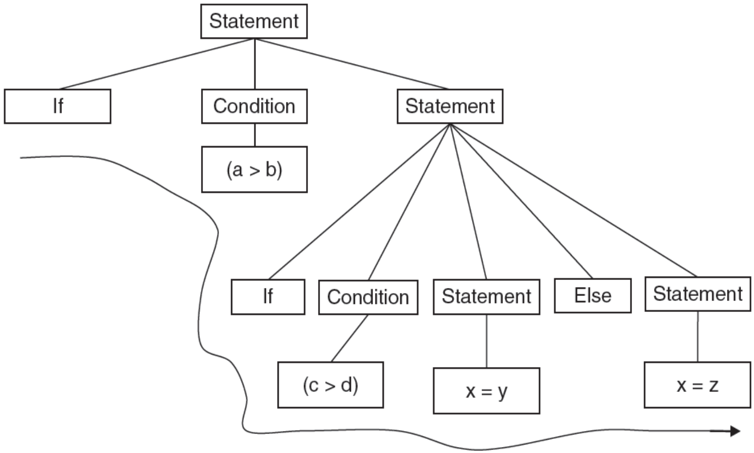
This can be derived by the following ways.

$$\begin{aligned}\text{statement} &\rightarrow \text{if (condition) } \underline{\text{statement}}\\ &\rightarrow \text{if (condition) } \overline{\text{if (condition) statement else statement}}\\ &\rightarrow \text{if}(a > b)\ \text{if}(c > d)\ x = y \text{ else } x = z\end{aligned}$$

or

$$\begin{aligned}\text{statement} &\rightarrow \text{if (condition) } \underline{\text{statement}} \text{ else statement}\\ &\rightarrow \text{if (condition) } \overline{\text{if (condition)}} \text{ statement else statement}\\ &\rightarrow \text{if}(a > b)\ \text{if}(c > d)\ x = y \text{ else } x = z\end{aligned}$$

The parse trees for the previous two derivations are given in Fig. 13.4.


Statement
If
Condition
Statement
(a > b)
If
Condition
Statement
Else
Statement
(c > d)
x = y
x = z

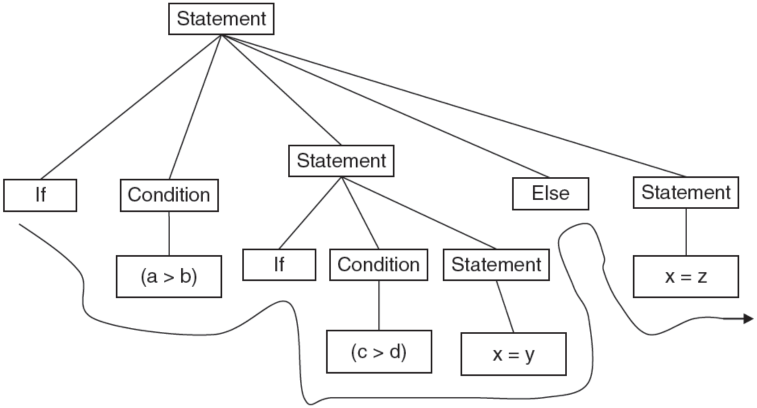


Fig. 13.4 *Nested If-Else Problem*

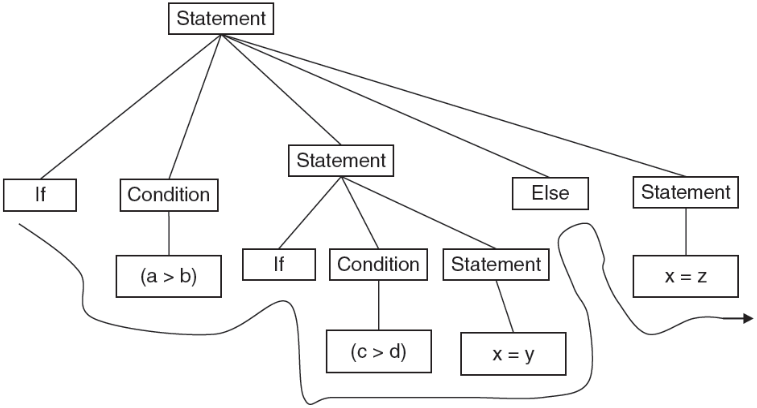


Fig. 13.4 *Nested If-Else Problem*

Now the 'else' part is for which if ?–the first or the second. Attaching 'else' with each of two results different. So, ambiguity must be removed. Here the ambiguity is removed by the following rule– 'Match each else with the closest previous unmatched if'

In context free grammar section another common case of ambiguity has already been described for
the grammar.

$$E \rightarrow E + E/E - E/E * E/id$$

In this case the ambiguity is removed by putting precedence of operators.

13.3.3 Parser

The syntax analyzer checks whether a given source program satisfies the rules implied by a context-free grammar or not by constructing a parse tree for a given statement. For this reason syntax analyzer is sometimes known as parser. Parser is of mainly two types–top down parser and bottom up parser. In this chapter we shall discuss $LL(1)$ parsing as an example of top down parsing and LR parsing, as an example of bottom up parsing.

Top down parsing technique can not handle left recursive grammar. A grammar is called left recursive if it has a non-terminal 'A' such that there is a derivation.

$$A \overset{+}{\Rightarrow} A\alpha \text{ for some string } \alpha$$

The process of the removal of left recursion is described in context free grammar chapter.

Backtracking is also a problem related to parsing.

Let a grammar is in the form.

The process of the removal of left recursion is described in context free grammar chapter.
Backtracking is also a problem related to parsing.
Let a grammar is in the form.

$$
\begin{aligned}
A &\rightarrow aBC/aB \\
B &\rightarrow bB/bC \\
C &\rightarrow c
\end{aligned}
$$

And the string to be generated is *abbc*. Compiler can scan one symbol at a time. As *aBC* and *aB* both are started with the symbol 'a'; compiler can take any of the productions to generate *abbc*. Let it take $A \rightarrow aBC$ and the derivations are as follows.

$$\mathrm{A} \rightarrow \mathrm{a\underline{B}C} \rightarrow \mathrm{ab\underline{B}C} \rightarrow \mathrm{abbCC} \rightarrow \mathrm{abbcC}$$

The string *abbc* is already generated but a non-terminal 'C' still remains. So, the parser has to backtrack to choose the alternative production $A \rightarrow aB$, and the following derivations are

$$\mathrm{aB} \rightarrow \mathrm{ab\underline{B}} \rightarrow \mathrm{abbC} \rightarrow \mathrm{abbc}$$

It succeeds in generating the string. For backtracking the parser wastes a lot of lime. Hence the backtracking must be removed. Left factoting of the grammar is a process to remove backtracking. Actually top-down parsing technique insists that the grammar must be left factored. The process of left factoring a grammar is described in Context free grammar chapter.

It succeeds in generating the string. For backtracking the parser wastes a lot of lime. Hence the backtracking must be removed. Left factoting of the grammar is a process to remove backtracking. Actually top-down parsing technique insists that the grammar must be left factored. The process of left factoring a grammar is described in Context free grammar chapter.

13.3.3.1 Top-down Parsing

In top down parsing a parse is created from root to leaf. Top down parsing corresponds to a preorder traversal of the parse tree. In this parsing a left most derivation is applied at each derivation step. There are mainly two types of top-down parsing–recursive decent parsing and predictive parsing.

Recursive Decent Parsing: Recursive decent parsing is most straight forward form of parsing. In this type of top down parsing the parser attempts to verify the input string read from left to right with the grammar underlying. A basic operation of recursive decent parsing involves reading characters from the input stream one at a time, and matching it with terminals from the production rule of the grammar that describes the syntax of the input. In recursive decent parsing backtracking occurs. This backtracking is caused because in recursive decent parsing left recursion is not removed. Due to this it is not efficient and not widely used.

Predictive Parsing: This is a type of top down parsing which never backtracks. To prevent the occurrence of backtracking from the grammar left recursion is removed and the grammar is left factored. Non recursive predictive parsing is a mixture of recursive decent parsing and predictive parsing, which never backtracks.

13.3.3.2 LL(1) Parsing

LL(1) parsing is an example of predictive parsing. The fi rst ‘L’ in LL(1) denotes that input string is scanned from left to right. Second ‘L’ denotes that by left most derivation the input string is constructed. ‘1’ denotes that only one symbol of the input string is used as a look-head symbol to determine parser action. LL(1) is a table driven parser as the parser consults with a parsing table for its actions. A LL(1) parser consists of a input tape, a stack and a parsing table. The block diagram of a LL(1) parser is given in Fig. 13.5.

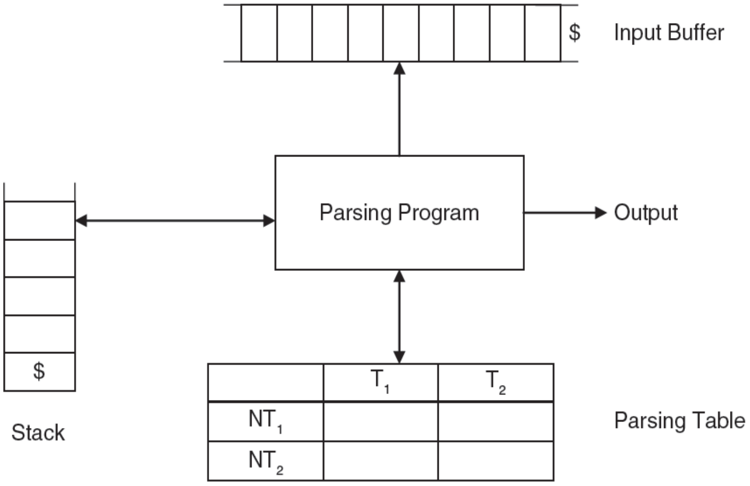


Fig. 13.5 *LL(1) Parser*

$DifferentComponentsofLL(1)Parser$

- **Input buffer:** Input buffer contains the input string to be parsed. It contains a \$ as the end marker of the input string.
- **Stack:** Stack contains the terminal and non-terminal symbols. In brief it can be said that Stack contains grammar symbols. Bottom of the stack contains a special symbol $\$ \in (V_N \cup \Sigma)$. At the beginning of parsing the stack contains $\$S$, where S is the start symbol.
- **Parsing table:** It is most important part of a $LL(1)$ parser. It is a two dimensional array whose left most column contains the non-terminal symbols $\in V_N$ and top most row contains the terminal symbol $\in \Sigma \cup \$$. Some entries of the parsing table contain the production rules of the grammar and remaining contains error.

How LL(1) Parser Works? As described in earlier section, the input buffer contains the input string to be parsed ended by a \$ and stack contains the grammar symbols including \$. The symbols in the input buffer are terminal symbols. As it is $LL(1)$ parser, therefore one symbol of the input buffer is scanned at a time. The parser action is determined by the top of the stack symbol let T_Stack and the current scanned input symbol from the input buffer let I_i/p. Four types actions are possible for a $LL(1)$ parser.

1. T_Stack **and** I_i/p **are same terminal symbol:** Parser pops T_Stack from the top of the stack and points next symbol in the input buffer.
2. T_Stack **is a non-terminal:** Parser consults with the corresponding entries for T_Stack and I_i/p in the parsing table. The top of the stack is replaced by the right hand side of the production in reverse. [Parsing table entries is a production rule of the grammar]
3. T_Stack **is \$ and** I_i/p **is \$:** parsing successful. Declare accept.
4. **None of the above three:** Error.

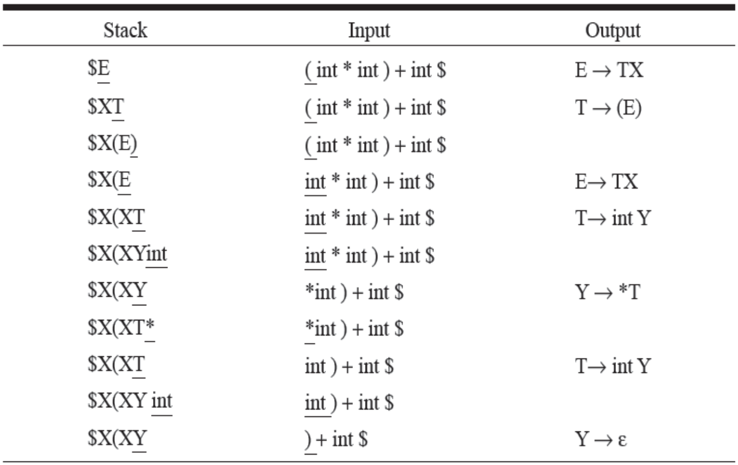
Let take an example to describe it in detail.

Example 13.1 `Consider the following parsing table.`

	int	$*$	$+$	$($	$)$	$\$$
E	$E \to TX$			$E \to TX$		
X			$X \to +E$		$X \to \varepsilon$	$X \to \varepsilon$
T	$T \to intY$			$T \to (E)$		
Y		$Y \to *T$	$Y \to \varepsilon$		$Y \to \varepsilon$	$Y \to \varepsilon$

Parse the string $(int * int) + int$ using the following parsing table.

Solution:



Stack	Input	Output
$E	(int * int) + int $	E → TX
$XT	(int * int) + int $	T → (E)
$X(E)	(int * int) + int $	
$X(E	int * int) + int $	E→ TX
$X(XT	int * int) + int $	T→ int Y
$X(XYint	int * int) + int $	
$X(XY	*int) + int $	Y → *T
$X(XT*	*int) + int $	
$X(XT	int) + int $	T→ int Y
$X(XY int	int) + int $	
$X(XY	) + int $	Y → ε