This example shows the two things we must do in order to prove that a grammar generates a language \( L \). We must show that every sentence generated by the grammar is in \( L \), and we must show that every string in \( L \) can be generated by the grammar. □

We have already seen a grammar for arithmetic expressions. The following grammar fragment (4.11) generates conditional statements.

\[
\text{stat} \rightarrow \text{if cond then stat} \\
| \text{if cond then stat else stat} \\
| \text{other-stat} \tag{4.11}
\]

Thus the string

\[
\text{if } C_1 \text{ then } S_1 \\
\text{else if } C_2 \text{ then } S_2 \text{ else } S_3
\]

would have the parse tree shown in Fig. 4.4.

![Parse tree](image)

**Fig. 4.4.** Parse tree.

Grammar (4.11) is ambiguous, however, since the string

\[
\text{if } C_1 \text{ then if } C_2 \text{ then } S_1 \text{ else } S_2 \tag{4.12}
\]

has the two parse trees shown in Fig. 4.5.

In all programming languages with conditional statements of this form, the first parsing is preferred. The general rule is "Each else is to be matched with the closest previous unmatched then."

We could incorporate this disambiguating rule directly into the grammar if we wish. For example, we could rewrite grammar (4.11) as the following
4.3 CAPABILITIES OF CONTEXT-FREE GRAMMARS

Fig. 4.5. Two parse trees for ambiguous sentence.

unambiguous grammar.

\[
\begin{align*}
\text{stat} & \rightarrow \text{matched-stat} \\
& \quad | \text{unmatched-stat} \\
\text{matched-stat} & \rightarrow \text{if cond then matched-stat else matched-stat} \\
& \quad | \text{other-stat} \\
\text{unmatched-stat} & \rightarrow \text{if cond then stat} \\
& \quad | \text{if cond then matched-stat else unmatched-stat}
\end{align*}
\]

This grammar generates the same set of strings as (4.11), but it allows only one parsing for string (4.12), namely the one that associates each else with the previous unmatched then.
Construction of Parsing Tables

The following algorithm can be used to construct a predictive parsing table for a grammar $G$. The idea behind the algorithm is simple. Suppose $A \rightarrow \alpha$ is a production with $a$ in $\text{FIRST}(\alpha)$. Then, whenever the parser has $A$ on top of the stack with $a$ the current input symbol, the parser will expand $A$ by $\alpha$. The only complication occurs when $\alpha = \epsilon$ or $\alpha \Rightarrow \epsilon$. In this case, we should also expand $A$ by $\alpha$ if the current input symbol is in $\text{FOLLOW}(A)$, or if the $\$\$ on the input has been reached and $\$\$ is in $\text{FOLLOW}(A)$.

**Algorithm 5.4.** Constructing a predictive parsing table.

*Input.* Grammar $G$.

*Output.* Parsing table $M$.

*Method.*

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal $a$ in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
3. If $\epsilon$ is in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal $b$ in $\text{FOLLOW}(A)$. If $\epsilon$ is in $\text{FIRST}(\alpha)$ and $\$\$ is in $\text{FOLLOW}(A)$, add $A \rightarrow \alpha$ to $M[A, \$\$].
4. Make each undefined entry of $M$ *error*.

**Example 5.20.** Let us apply Algorithm 5.4 to grammar (5.9). Since $\text{FIRST}(TE') = \text{FIRST}(T) = \{ (, \text{id}) \}$, production $E \rightarrow TE'$ causes $M[E, \{ (, \text{id}) \}$ and $M[E, \text{id}]$ to acquire the entry $E \rightarrow TE'$.

Production $E' \rightarrow +TE'$ causes $M[E', +]$ to acquire $E' \rightarrow +TE'$. Production $E' \rightarrow \epsilon$ causes $M[E', \{ \}]$ and $M[E', \$\$]$ to acquire $E' \rightarrow \epsilon$ since $\text{FOLLOW}(E') = \{ (, \text{id}), \$\$ \}$.

The parsing table produced by Algorithm 5.4 for $G$ was shown in Fig. 5.24.

**LL(1) Grammars**

Algorithm 5.4 can be applied to any grammar $G$ to produce a parsing table $M$. For some grammars, however, $M$ may have some entries that are multiply-defined. For example, if $G$ is left-recursive or ambiguous, then $M$ will have at least one multiply-defined entry.

**Example 5.21.** Consider the grammar from Example 5.16.

$$S \rightarrow \epsilon \text{SS'} \mid a$$

$$S' \rightarrow eS \mid \epsilon$$

$$C \rightarrow b$$

(5.11)

The parsing table for grammar (5.11) is shown in Fig. 5.26.
<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>e</th>
<th>i</th>
<th>t</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>S → a</td>
<td></td>
<td></td>
<td></td>
<td>S → iCtSS'</td>
<td></td>
</tr>
<tr>
<td>S'</td>
<td></td>
<td>S' → e</td>
<td></td>
<td></td>
<td></td>
<td>S' → e</td>
</tr>
<tr>
<td>C</td>
<td></td>
<td>C → b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 5.26.** Parsing table.

The entry for $M[S', e]$ contains both $S' → eS$ and $S' → e$, since $\text{FOLLOW}(S') = \{e, S\}$. The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an $e$ (else) is seen. We can resolve the ambiguity if we choose $S' → eS$. This choice corresponds to associating else’s with the closest previous then’s. Note that the choice $S' → e$ would prevent $e$ from ever being put on the stack or removed from the input, and is therefore surely wrong. 

There arises the question of what should be done when a parsing table has multiply-defined entries. The easiest recourse is to transform the grammar by eliminating all left-recursion and then left-factoring whenever possible, hopefully to produce a grammar for which the parsing table has no multiply-defined entries.

A grammar whose parsing table has no multiply-defined entries is said to be $LL(1)$. It can be shown that Algorithm 5.4 produces a parsing table for every $LL(1)$ grammar, and that this table parses all and only the sentences of the grammar. It can also be shown that a grammar $G$ is $LL(1)$ if and only if whenever $A → α | β$ are two distinct productions of $G$ the following conditions hold:

1. For no terminal $a$ do $α$ and $β$ derive strings beginning with $a$.
2. At most one of $α$ and $β$ can derive the empty string.
3. If $β \Rightarrow e$, then $α$ does not derive any strings beginning with a terminal in $\text{FOLLOW}(A)$.

Clearly, the grammar of Example 5.19 for arithmetic expressions is $LL(1)$. The grammar (5.11) of Example 5.21 modeling if-then-else statements is not.

Unfortunately, there are some grammars for which no amount of rewriting will yield an $LL(1)$ grammar. Grammar (5.11) is one such example. As we saw, we can still parse (5.11) with a predictive parser by arbitrarily making $M[S', e] = \{S' → eS\}$. In general, however, there are no universal rules by which multiply-defined entries can be made single-valued without affecting the language recognized by the parser.