Recursive-descent parsers are highly stylized. Can use single table-driven program instead, using two data structures:

**Parsing table** is 2-dimensional table \( M[X, a] \)

- One entry for every non-terminal \( X \) and terminal \( a \).
- Entries are productions or error indicators.
- Entry \( M[X, a] \) says “what to do” when looking for non-terminal \( X \) while next input symbol is \( a \).

**Parsing stack** handles recursion explicitly
- Holds “what’s left to match” in the input (in reverse order)

---

**EXAMPLE TABLE AND EXECUTION**

Recall arithmetic expression grammar (after left-recursion removal):

\[
\begin{align*}
E & \rightarrow TE' \\
E' & \rightarrow +TE' | \epsilon \\
T & \rightarrow FT' \\
T' & \rightarrow *FT' | \epsilon \\
F & \rightarrow (E) | id
\end{align*}
\]

The corresponding parsing table is:

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>+</th>
<th>*</th>
<th>(</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E ) ( E \rightarrow TE' )</td>
<td>( E \rightarrow TE' )</td>
<td>( E \rightarrow TE' )</td>
<td>( E \rightarrow \epsilon )</td>
<td>( E \rightarrow \epsilon )</td>
<td></td>
</tr>
<tr>
<td>( E' ) ( E' \rightarrow +TE' )</td>
<td>( E' \rightarrow +TE' )</td>
<td>( E' \rightarrow \epsilon )</td>
<td>( E' \rightarrow \epsilon )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( T ) ( T \rightarrow FT' )</td>
<td>( T \rightarrow FT' )</td>
<td>( T \rightarrow FT' )</td>
<td>( T \rightarrow \epsilon )</td>
<td>( T \rightarrow \epsilon )</td>
<td></td>
</tr>
<tr>
<td>( T' ) ( T' \rightarrow *FT' )</td>
<td>( T' \rightarrow *FT' )</td>
<td>( T' \rightarrow \epsilon )</td>
<td>( T' \rightarrow \epsilon )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( F ) ( F \rightarrow id )</td>
<td>( F \rightarrow id )</td>
<td>( F \rightarrow (E) )</td>
<td>( F \rightarrow (E) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“Semantic action” code is executed once for each step in the **leftmost derivation** of an input sentence.

and a sample execution is...
### Table Construction Algorithm

For each production \( A \rightarrow \alpha \) do
- for each \( a \in \text{FIRST}(\alpha) \) do
  - add \( A \rightarrow \alpha \) to \( M[A, a] \)
- if \( \epsilon \in \text{FIRST}(\alpha) \) then
  - for each \( b \in \text{FOLLOW}(A) \) do
    - add \( A \rightarrow \alpha \) to \( M[A, b] \)
- set any empty elements of \( M \) to error

---

### Parsing Table Construction

\( \text{FIRST}(\alpha) \) is the set of **terminals** (and possibly \( \epsilon \)) that begin strings derived from \( \alpha \), where \( \alpha \) is any string of grammar symbols (terminals or non-terminals). (Book 1st ed. defines \( \text{FIRST()} \) only on individual symbols rather than strings of symbols; our definition is a consistent extension of this.)

\( \text{FOLLOW}(A) \) is the set of **terminals** (possibly including \( \$ \)) that can follow the **non-terminal** \( A \) in some **sentential form** (intermediate phrase in a derivation), i.e., the set of terminals

\[ \{ a \mid S \rightarrow^* \alpha A a \beta \text{ for some } \alpha, \beta \} \]

(This definition is equivalent to the book's. Note that there is an erratum for 1st edition Figure 3.5.)

---

### Computing FIRST

For any string of symbols \( \alpha \), \( \text{FIRST}(\alpha) \) is the **smallest** set of terminals (and \( \epsilon \)) obeying these rules:

- \( \text{FIRST}(a\alpha) = \{ a \} \) for any terminal \( a \) and any \( \alpha \) (empty or non-empty)
- \( \text{FIRST}(\epsilon) = \{ \epsilon \} \)
- \( \text{FIRST}(A) = \text{FIRST}(\alpha_1) \cup \text{FIRST}(\alpha_2) \cup \ldots \cup \text{FIRST}(\alpha_n) \)
  - where \( A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \) are all the productions for \( A \)
- \( \text{FIRST}(A\alpha) = \) if \( \epsilon \notin \text{FIRST}(A) \) then \( \text{FIRST}(A) \)
  - else \( \{ \epsilon \} \cup \text{FIRST}(\alpha) \)
**Example FIRST computation**

\[
\begin{align*}
\text{FIRST}(F) &= \text{FIRST}(E) \cup \text{FIRST}(\text{id}) = \{ \text{id} \} \\
\text{FIRST}(T') &= \text{FIRST}(FT') \cup \text{FIRST}(\epsilon) = \{ \epsilon \} \\
\text{FIRST}(T) &= \text{FIRST}(FT') = \text{FIRST}(F) = \{ \text{id} \} \\
\text{FIRST}(E') &= \text{FIRST}(TE') \cup \text{FIRST}(\epsilon) = \{ \epsilon \} \\
\text{FIRST}(E) &= \text{FIRST}(TE') = \text{FIRST}(T) = \{ \text{id} \}
\end{align*}
\]

**Example FOLLOW computation**

<table>
<thead>
<tr>
<th>Computation</th>
<th>Relevant Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{FOLLOW}(E) ) = { \text{id} } \cup \text{FIRST}(\text{id}) )</td>
<td>( F \rightarrow (E) )</td>
</tr>
<tr>
<td>( \text{FOLLOW}(E') ) = \text{FOLLOW}(E) = { \text{id} } )</td>
<td>( E \rightarrow T E' )</td>
</tr>
<tr>
<td>( \text{FOLLOW}(T) ) = ( \text{FIRST}(E') - { \epsilon } ) \cup \text{FOLLOW}(E) \cup \text{FOLLOW}(E') )</td>
<td>( E \rightarrow T E' )</td>
</tr>
<tr>
<td>( \text{FOLLOW}(T') ) = ( \text{FOLLOW}(T) ) = ( \text{FIRST}(T') - { \epsilon } ) \cup \text{FOLLOW}(T) \cup \text{FOLLOW}(T') )</td>
<td>( T \rightarrow FT' )</td>
</tr>
<tr>
<td>( \text{FOLLOW}(F) ) = ( \text{FIRST}(T') - { \epsilon } ) \cup \text{FOLLOW}(T) \cup \text{FOLLOW}(T') )</td>
<td>( T' \rightarrow FT' )</td>
</tr>
</tbody>
</table>

**Computing FOLLOW**

Must compute simultaneously for all non-terminals \( A \).

FOLLOW sets are **smallest** sets obeying these rules:

- \( \$$ \text{is in FOLLOW}(S) \)
- If there is a production \( A \rightarrow \alpha B \beta \), then everything in \( \text{FIRST}(\beta) - \{ \epsilon \} \) is in \( \text{FOLLOW}(B) \).
- If there is a production \( A \rightarrow \alpha B \beta \) where \( \beta = \epsilon \) or \( \epsilon \in \text{FIRST}(\beta) \), then everything in \( \text{FOLLOW}(A) \) is in \( \text{FOLLOW}(B) \).

**LL(1) grammars**

A grammar can be used to build a predictive table-driven parser \( \Leftrightarrow \) parsing table \( M \) has no duplicate entries.

In terms of FIRST and FOLLOW sets, this means that, for each production

\( A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n \)

- All \( \text{FIRST}(\alpha_i) \) are disjoint, and
- There is at most one \( i \) such that \( \epsilon \in \text{FIRST}(\alpha_i) \), and, if there is such an \( i \), \( \text{FOLLOW}(A) \cap \text{FIRST}(\alpha_j) = \emptyset \) for all \( j \neq i \).

Such grammars are called **LL(1)**.

- The first **L** stands for “Left-to-right scan of input.”
- The second **L** stands for “Leftmost derivation.”
- The **1** stands for “1 token of lookahead.”

No LL(1) grammar can be ambiguous or left-recursive.