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)

### Table-Driven Bottom-Up Parsing Algorithm

(assuming $= EOF; S = start symbol)

```
push($); push(S); repeat
  a ← input
  if top is a terminal or $ then
    if top = a then
      pop(); advance();
    else error();
  else if $[top, a]$ is $X \rightarrow Y_1 \cdots Y_k$ then
    pop(); push($Y_k$); push($Y_{k-1}$); ...; push($Y_1$);
    /* do “semantic action” here */
  else error();
  until top = $ 
```

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

### Example Table and Execution

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

- $E \rightarrow T E'$
- $E' \rightarrow + T E' | \epsilon$
- $T \rightarrow F T'$
- $T' \rightarrow * F T' | \epsilon$
- $F \rightarrow (E) | id$

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

and a sample execution is...
Stack        Input        "Output"
\$E\$        idy*idz$        \$E\$\rightarrow T N  \
\$E'T\$        idy*idz$        \$T\rightarrow F\$  \
\$E'T'id\$        idy*idz$        \$F\rightarrow id\$  \
\$E'T'\$        idy*idz$        \$T'\rightarrow \epsilon\$.  \
\$E'T'F\$        idy*idz$        \$E'\rightarrow +T E'\$  \
\$E'T'id\$        idy*idz$        \$T'\rightarrow +T E'\$  \
\$E'T'\$        idy*idz$        \$T\rightarrow FT\$  \
\$E'T'F\$        idy*idz$        \$F\rightarrow id\$  \
\$E'T'\$        idy*idz$        \$F\rightarrow id\$  \
\$E'T'id\$        idy*idz$        \$T'\rightarrow \epsilon\$.  \
\$E'T'\$        idy*idz$        \$T\rightarrow \epsilon\$.  \
\$E'T'F\$        idy*idz$        \$F\rightarrow id\$  \
\$E'T'\$        idy*idz$        \$T\rightarrow FT\$  \
\$E'T'id\$        idy*idz$        \$T'\rightarrow \epsilon\$.  

**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**

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

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

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

(This definition is equivalent to the book's. Note there is an erratum for 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:

\[
\text{FIRST}(a\alpha) = \{a\} \text{ for any terminal } a \text{ and any } \alpha \text{ (empty or non-empty)}
\]

\[
\text{FIRST}(\epsilon) = \{\epsilon\}
\]

\[
\text{FIRST}(A) = \text{FIRST}(\alpha_1) \cup \text{FIRST}(\alpha_2) \cup \ldots \cup \text{FIRST}(\alpha_n)
\]

where $A \rightarrow \alpha_1 | \alpha_2 | \ldots | \alpha_n$ are all the productions for $A$

\[
\text{FIRST}(A\alpha) = \text{if } \epsilon \notin \text{FIRST}(A) \text{ then } \text{FIRST}(A)
\]

else $(\text{FIRST}(A) - \{\epsilon\}) \cup \text{FIRST}(\alpha)$
Example FIRST computation

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

Example FOLLOW computation

<table>
<thead>
<tr>
<th>Computation</th>
<th>Relevant Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>FOLLOW(E)</td>
<td>{$} \cup FIRST()</td>
</tr>
<tr>
<td></td>
<td>{$}$</td>
</tr>
<tr>
<td>FOLLOW(E')</td>
<td>FOLLOW(E) = {$}$</td>
</tr>
<tr>
<td></td>
<td>E \rightarrow TE'</td>
</tr>
<tr>
<td></td>
<td>(what about E \rightarrow +TE' ?)</td>
</tr>
<tr>
<td>FOLLOW(T)</td>
<td>{FIRST(E') - {\epsilon}}</td>
</tr>
<tr>
<td></td>
<td>\cup FOLLOW(E) E \rightarrow TE'</td>
</tr>
<tr>
<td></td>
<td>\cup FOLLOW(E') E' \rightarrow +TE'</td>
</tr>
<tr>
<td></td>
<td>= {+ }$}</td>
</tr>
<tr>
<td>FOLLOW(T')</td>
<td>FOLLOW(T) = {+ }$}</td>
</tr>
<tr>
<td>FOLLOW(F)</td>
<td>{FIRST(T') - {\epsilon}}</td>
</tr>
<tr>
<td></td>
<td>\cup FOLLOW(T) T \rightarrow FT'</td>
</tr>
<tr>
<td></td>
<td>\cup FOLLOW(T') T' \rightarrow +FT'</td>
</tr>
<tr>
<td></td>
<td>= {+ }$}</td>
</tr>
</tbody>
</table>

Computing FOLLOW

Must compute simultaneously for all non-terminals A.

FOLLOW sets are smallest sets obeying these rules:

- \$ is in FOLLOW(S)
- If there is a production \(A \rightarrow \alpha B\beta\), then everything in FIRST(\beta) - \{\epsilon\} is in FOLLOW(B).
- If there is a production \(A \rightarrow \alpha B\beta\) where \(\beta = \epsilon\) or \(\epsilon \in FIRST(\beta)\), then everything in FOLLOW(A) is in 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 FIRST(\alpha_i) are disjoint, and
- There is at most one \(i\) such that \(\epsilon \in FIRST(\alpha_i)\), and, if there is such an \(i\), FOLLOW(A) \cap 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.