Table-driven Top-down Parsing

Recursive-descent parsers are highly stylized.

Can use single table-driven program instead

“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 reverse order)
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 M[top,a] is X → Y₁ Y₂ ... Yₖ then
    pop();
    push(Yₖ); push(Yₖ₋₁); ...; push(Y₁);
    /* do “semantic action” here */
  else error();
until top = $

“Semantic action” code is executed once for each step in the left-most derivation of an input sentence.
### Example

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

#### Sample Execution:

<table>
<thead>
<tr>
<th>Stack</th>
<th>Input</th>
<th>“Output”</th>
</tr>
</thead>
<tbody>
<tr>
<td>$E$</td>
<td>id+id*id$</td>
<td></td>
</tr>
<tr>
<td>$E'T$</td>
<td>id+id*id$</td>
<td>$E \rightarrow TE'$</td>
</tr>
<tr>
<td>$E'T'F$</td>
<td>id+id*id$</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>id+id*id$</td>
<td>$F \rightarrow \text{id}$</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>+id*id$</td>
<td></td>
</tr>
<tr>
<td>$E'$</td>
<td>+id*id$</td>
<td>$T' \rightarrow \epsilon$</td>
</tr>
<tr>
<td>$E'T+$</td>
<td>+id*id$</td>
<td>$E' \rightarrow +TE'$</td>
</tr>
<tr>
<td>$E'T$</td>
<td>id*id$</td>
<td></td>
</tr>
<tr>
<td>$E'T'F$</td>
<td>id*id$</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>id*id$</td>
<td>$F \rightarrow \text{id}$</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>*id$</td>
<td></td>
</tr>
<tr>
<td>$E'T'F*$</td>
<td>*id$</td>
<td>$T' \rightarrow *FT'$</td>
</tr>
<tr>
<td>$E'T'F$</td>
<td>id$</td>
<td></td>
</tr>
<tr>
<td>$E'T'id$</td>
<td>id$</td>
<td>$F \rightarrow \text{id}$</td>
</tr>
<tr>
<td>$E'T'$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$E'$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
</tbody>
</table>
Parsing Table Construction

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

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

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

(This definition is equivalent to the book’s. Note there is an erratum for Figure 3.5.)

Table Construction Algorithm

\begin{verbatim}
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
\end{verbatim}
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 \( A \)

\[
\text{FIRST}(A\alpha) = \begin{cases} \text{FIRST}(A) & \text{if } \epsilon \notin \text{FIRST}(A) \\ \text{FIRST}(A) - \{\epsilon\} \cup \text{FIRST}(\alpha) & \text{else} \end{cases}
\]

Example

\[
\text{FIRST}(F) = \text{FIRST}((E)) \cup \text{FIRST}(\text{id}) = \{( \text{id}\}
\]

\[
\text{FIRST}(T') = \text{FIRST}(*F T') \cup \text{FIRST}(\epsilon) = \{* \epsilon\}
\]

\[
\text{FIRST}(T) = \text{FIRST}(FT') = \text{FIRST}(F') = \{( \text{id}\}
\]

\[
\text{FIRST}(E') = \text{FIRST}(+T E') \cup \text{FIRST}(\epsilon) = \{+ \epsilon\}
\]

\[
\text{FIRST}(E) = \text{FIRST}(T E') = \text{FIRST}(T) = \{( \text{id}\}
\]
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 $FOLLOW(B)$.

Example

<table>
<thead>
<tr>
<th>Computation</th>
<th>Relevant Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FOLLOW(E)$</td>
<td>$F \rightarrow (E)$</td>
</tr>
<tr>
<td></td>
<td>$E \rightarrow TE'$</td>
</tr>
<tr>
<td></td>
<td>(what about $E' \rightarrow +TE'$ ?)</td>
</tr>
<tr>
<td>$FOLLOW(E')$</td>
<td>$E \rightarrow TE'$</td>
</tr>
<tr>
<td></td>
<td>$E' \rightarrow +TE'$</td>
</tr>
<tr>
<td>$FOLLOW(T)$</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td></td>
<td>$T' \rightarrow ^*FT'$</td>
</tr>
<tr>
<td>$FOLLOW(T')$</td>
<td>$T \rightarrow FT'$</td>
</tr>
<tr>
<td></td>
<td>$T' \rightarrow ^*FT'$</td>
</tr>
</tbody>
</table>

| $FOLLOW(T')$      | $T \rightarrow FT'$ |
|                   | $T' \rightarrow ^*FT'$ |

$FIRST(E) = \{$$\} \cup \{\epsilon\}$
LL(1) Grammars

A grammar can be used to build a predictive table-driven parser ⇔ parsing table $M$ has no duplicate entries.

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

\[ A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \]

- 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.