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 $ = \text{EOF}; S = \text{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 $M[\text{top}, a]$ is $X \rightarrow Y_1 Y_2 \ldots Y_k$ 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 left-most derivation of an input sentence.

Example

<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'$</td>
<td>$E' \rightarrow \epsilon$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E'$</td>
<td>$E' \rightarrow TE'$</td>
<td>$E \rightarrow TE'$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$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$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>$F \rightarrow \text{id}$</td>
<td>$F \rightarrow (E)$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sample Execution:

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

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

*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

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

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

**Table Construction Algorithm**

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

**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**

\[
\begin{align*}
*FOLLOW(E) &= \{ $ \} \cup *FIRST(1) \quad F \rightarrow (E) \\
*FOLLOW(E') &= *FOLLOW(E) = \{ $ \} \quad E \rightarrow TE' \\
&= \{ $ \} \quad (\text{what about } E' \rightarrow + TE' ~?) \\
*FOLLOW(T) &= (FIRST(E') - \{ \epsilon \}) \cup *FOLLOW(E) \\
&= \{+\} \quad E \rightarrow TE' \\
&\quad \cup *FOLLOW(E') \\
&= \{+\} \quad E' \rightarrow + TE' \\
*FOLLOW(T') &= *FOLLOW(T) = \{+\} \quad T \rightarrow FT' \\
*FOLLOW(F) &= (FIRST(T') - \{ \epsilon \}) \cup *FOLLOW(T) \\
&= \{ * + \} \quad T \rightarrow FT' \\
&\quad \cup *FOLLOW(T') \\
&= \{ * + \} \quad T' \rightarrow *FT'
\end{align*}
\]

**LL(1) Grammars**

A grammar can be used to build a predictive table-driven parser \( \Leftrightarrow \text{parsing table } M \) 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 I stands for “I token of lookahead.”

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