Syntax Analysis, IV

Comp 412

Copyright 2017, Keith D. Cooper & Linda Torczon, all rights reserved.

Students enrolled in Comp 412 at Rice University have explicit permission to make copies of these materials for their personal use.

Faculty from other educational institutions may use these materials for nonprofit educational purposes, provided this copyright notice is preserved.

Chapter 3 in EaC2e
Review

Last Class

• Introduced **FIRST**, **FOLLOW**, and **FIRST**\(^+\) sets
• Introduced the **LL(1)** condition

\[ \text{A grammar } G \text{ can be parsed predictively with one symbol of lookahead if} \]
\[ \text{for all pairs of productions } A \rightarrow \beta \text{ and } A \rightarrow \gamma \text{ that have the same lhs } A: \]
\[ \text{FIRST}(A \rightarrow \beta) \cap \text{FIRST}(A \rightarrow \gamma) = \emptyset \]

• Observed that predictively parsable, or **LL(1)** grammars
• Showed how to construct a recursive-descent parser for an **LL(1)** grammar

**We did not cover**

• An algorithm to construct **FIRST** sets
• An algorithm to construct **FOLLOW** sets
FIRST and FOLLOW Sets

**FIRST(α)**

For some $\alpha \in (T \cup NT \cup \text{EOF} \cup \varepsilon)^*$, define $\text{FIRST}(\alpha)$ as the set of tokens that appear as the first symbol in some string that derives from $\alpha$

That is, $x \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* x \gamma$, for some $\gamma$

**FIRST** is defined over strings of grammar symbols: $(T \cup NT \cup \text{EOF} \cup \varepsilon)^*$

**FOLLOW(A)**

For some $A \in NT$, define $\text{FOLLOW}(A)$ as the set of symbols that can occur immediately after $A$ in a valid sentential form

$\text{FOLLOW}(S) = \{\text{EOF}\}$, where $S$ is the start symbol

$\text{FOLLOW}$ is defined over the set of nonterminal symbols, $NT$

To build $\text{FOLLOW}$ sets, we need $\text{FIRST}$ sets ...

EOF $\equiv$ end of file
**FIRST and FOLLOW Sets**

**FIRST(α)**
For some \( \alpha \in (T \cup NT \cup EOF \cup \varepsilon)^* \), define **FIRST(α)** as the set of tokens that appear as the first symbol in some string that derives from \( \alpha \)

That is, \( x \in **FIRST(α) iff \alpha \Rightarrow^* x \gamma, \) for some \( \gamma \)

**FIRST** is defined over strings of grammar symbols: \((T \cup NT \cup EOF \cup \varepsilon)^*\)

**FOLLOW(A)**
For some \( A \in NT \), define **FOLLOW(A)** as the set of symbols that can occur immediately after \( A \) in a valid sentential form

**FOLLOW(S) = {EOF}, where S is the start symbol**

**FOLLOW** is defined over the set of nonterminal symbols, NT

To build **FOLLOW** sets, we need **FIRST** sets ...

**EOF ≡ end of file**
Conceptual Sketch: Computing \textbf{FIRST} Sets

\begin{verbatim}
for each \( x \in (T \cup \text{EOF} \cup \epsilon) \)
\hspace{1cm} \text{FIRST}(x) \leftarrow \{x\} \\
for each \( A \in NT, \text{FIRST}(A) \leftarrow \emptyset \)
while (FIRST sets are still changing) do \\
\hspace{1cm} for each \( p \in P, \) of the form \( A \rightarrow \beta \) do \\
\hspace{2cm} rhs \leftarrow \text{FIRST}(B_1) \setminus \{\epsilon\} \\
\hspace{3cm} Some details go here to handle \( \epsilon \) productions \\
\hspace{1cm} \text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup \text{rhs} \\
end // for loop \\
end // while loop
\end{verbatim}

To begin, we will ignore \( \epsilon \) productions

- Initially, set \text{FIRST} for each nonterminal, terminal \text{EOF}, and \( \epsilon \)
- Then, loop through the productions and set \text{FIRST} for the \text{lhs} nonterminal to \text{FIRST} of the leading symbol on the \text{rhs}
- Need to iterate because \( \text{rhs} \) can start with a nonterminal

COMP 412, Fall 2017
Filling in the Details: Computing \textbf{FIRST} Sets

\begin{verbatim}
for each \( x \in (T \cup \text{EOF} \cup \varepsilon) \)
    \( \text{FIRST}(x) \leftarrow \{ x \} \)
for each \( A \in \text{NT}, \text{FIRST}(A) \leftarrow \emptyset \)
while (FIRST sets are still changing) do
    for each \( p \in P, \text{of the form } A \rightarrow B_1B_2\ldots B_k \) do
        \( \text{rhs} \leftarrow \text{FIRST}(B_1) - \{ \varepsilon \} \)
        for \( i \leftarrow 1 \) to \( k-1 \) by 1 while \( \varepsilon \in \text{FIRST}(B_i) \) do
            \( \text{rhs} \leftarrow \text{rhs} \cup (\text{FIRST}(B_{i+1}) - \{ \varepsilon \}) \)
        end // for loop
    if \( i = k \) and \( \varepsilon \in \text{FIRST}(B_k) \)
        then \( \text{rhs} \leftarrow \text{rhs} \cup \{ \varepsilon \} \)
    \( \text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup \text{rhs} \)
end // for loop
end // while loop
\end{verbatim}

\( \varepsilon \) complicates matters

If \( \text{FIRST}(B_1) \) contains \( \varepsilon \), then we need to add \( \text{FIRST}(B_2) \) to \( \text{rhs} \), and ... 

If the entire \( \text{rhs} \) can go to \( \varepsilon \), then we add \( \varepsilon \) to \( \text{FIRST}(lhs) \)
Computing **FIRST** Sets

\[\text{for each } x \in (T \cup \text{EOF} \cup \varepsilon)\]
\[\text{FIRST}(x) \leftarrow \{x\}\]
\[\text{for each } A \in \text{NT}, \text{FIRST}(A) \leftarrow \emptyset\]

while (FIRST sets are still changing) do
  for each \( p \in P \), of the form \( A \rightarrow B_1 B_2 \ldots B_k \) do
    \( \text{rhs} \leftarrow \text{FIRST}(B_1) - \{\varepsilon\}\)
    for \( i \leftarrow 1 \) to \( k-1 \) by \( 1 \) while \( \varepsilon \in \text{FIRST}(B_i) \) do
      \( \text{rhs} \leftarrow \text{rhs} \cup (\text{FIRST}(B_{i+1}) - \{\varepsilon\})\)
    end // for loop
  if \( i = k \) and \( \varepsilon \in \text{FIRST}(B_k) \)
  then \( \text{rhs} \leftarrow \text{rhs} \cup \{\varepsilon\}\)
  end // for loop
end // while loop

Outer loop is **monotone increasing** for FIRST sets
\[\Rightarrow |T \cup \text{NT} \cup \text{EOF} \cup \varepsilon| \text{ is bounded, so it terminates}\]

Inner loop is bounded by the length of the productions in the grammar

See also, Fig. 3.7, EaC2e, p. 104
Example

Consider the *SheepNoise* grammar & its FIRST sets

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal</td>
<td>→</td>
</tr>
<tr>
<td>1</td>
<td><em>SheepNoise</em></td>
<td>→</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Left-recursive SheepNoise Grammar*

Clearly and intuitively, $\text{FIRST}(x) = \{\text{baa}\}, \forall x \in (T \cup NT)$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>{ baa }</td>
</tr>
<tr>
<td><em>SheepNoise</em></td>
<td>{ baa }</td>
</tr>
<tr>
<td>baa</td>
<td>{ baa }</td>
</tr>
</tbody>
</table>
Computing \textbf{FIRST} Sets

\begin{verbatim}
for each \( x \in (T \cup \text{EOF} \cup \varepsilon) \)
\hspace{1em} FIRST(x) \leftarrow \{ x \}

for each \( A \in NT, \text{FIRST}(A) \leftarrow \emptyset \)

while (FIRST sets are still changing) do
\hspace{1em} for each \( p \in P, \text{of the form } A \rightarrow \beta \) do
\hspace{2em} rhs \leftarrow \text{FIRST}(B_1) \setminus \{ \varepsilon \}
\hspace{2em} if \( \beta \) is \( B_1B_2...B_k \) then begin;
\hspace{3em} for \( i \leftarrow 1 \) to \( k-1 \) by \( 1 \) while \( \varepsilon \in \text{FIRST}(B_i) \) do
\hspace{4em} rhs \leftarrow rhs \cup (\text{FIRST}(B_{i+1}) \setminus \{ \varepsilon \})
\hspace{3em} end \hspace{1em} // for loop
\hspace{2em} end \hspace{1em} // if-then

if \( i = k \) and \( \varepsilon \in \text{FIRST}(B_k) \)
\hspace{1em} then rhs \leftarrow rhs \cup \{ \varepsilon \}
\hspace{1em} \text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup \text{rhs}
\hspace{1em} end \hspace{1em} // for loop
\end{verbatim}

See also, Fig. 3.7, EaC2e, p. 104

Initialization assigns each FIRST set a value

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
Symbol & FIRST Set \\
\hline
Goal & \emptyset \\
SheepNoise & \emptyset \\
baa & \{ baa \} \\
\hline
\end{tabular}
\end{table}
Computing **FIRST** Sets

For each $x \in (T \cup EOF \cup \varepsilon)$

$\text{FIRST}(x) \leftarrow \{x\}$

For each $A \in NT$, $\text{FIRST}(A) \leftarrow \emptyset$

While (FIRST sets are still changing) do

For each $p \in P$, of the form $A \rightarrow \beta$ do

$rhs \leftarrow \text{FIRST}(B_1) - \{\varepsilon\}$

If $\beta$ is $B_1B_2...B_k$ then begin;

For $i \leftarrow 1$ to $k-1$ by 1 while $\varepsilon \in \text{FIRST}(B_i)$ do

$rhs \leftarrow rhs \cup (\text{FIRST}(B_{i+1}) - \{\varepsilon\})$

End // for loop

End // if-then

If $i = k$ and $\varepsilon \in \text{FIRST}(B_k)$

Then $rhs \leftarrow rhs \cup \{\varepsilon\}$

$\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup rhs$

End // for loop

End // while loop

See also, Fig. 3.7, EaC2e, p. 104

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST Set</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>$\emptyset$</td>
</tr>
<tr>
<td><strong>SheepNoise</strong></td>
<td>${\text{baa}}$</td>
</tr>
<tr>
<td><strong>baa</strong></td>
<td>${\text{baa}}$</td>
</tr>
</tbody>
</table>
Computing \textbf{FIRST} Sets

\begin{itemize}
  \item \textit{for each} \(x \in (T \cup \text{EOF} \cup \varepsilon)\) \\
      \(\text{FIRST}(x) \leftarrow \{x\}\)
  \item \textit{for each} \(A \in \text{NT}, \text{FIRST}(A) \leftarrow \emptyset\)
  \item \textit{while} (FIRST sets are still changing) \textit{do}
    \item \textit{for each} \(p \in P, \text{of the form } A \rightarrow \beta\) \textit{do}
      \item \(\text{rhs} \leftarrow \text{FIRST}(B_1) - \{\varepsilon\}\)
      \item \textit{if} \(\beta \text{ is } B_1B_2\ldots B_k\) \textit{then begin;}
        \item \textit{for } \(i \leftarrow 1 \text{ to } k-1 \text{ by } 1 \text{ while } \varepsilon \in \text{FIRST}(B_i)\) \textit{do}
          \item \(\text{rhs} \leftarrow \text{rhs} \cup (\text{FIRST}(B_{i+1}) - \{\varepsilon\})\)
        \item \textit{end} // for loop
      \item \textit{end} // if-then
      \item \textit{if } \(i = k\) \text{ and } \varepsilon \in \text{FIRST}(B_k)\)
        \item \(\text{then } \text{rhs} \leftarrow \text{rhs} \cup \{\varepsilon\}\)
      \item \(\text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup \text{rhs}\)
    \item \textit{end} // for loop
  \item \textit{end} // while loop
\end{itemize}

\textbf{Production 0}

(1) \textit{sets } \text{rhs} \textit{ to } \text{FIRST(}
\textit{Sheepnoise) &}

(2) \textit{copies } \text{rhs} \textit{ into } \text{FIRST(} \text{Goal) }

\ldots \textit{and one more iteration to recognize that the FIRST sets have stopped changing}

\begin{tabular}{|l|l|}
\hline
\textbf{Symbol} & \textbf{FIRST Set} \\
\hline
\textit{Goal} & \{ baa \} \\
\textit{SheepNoise} & \{ baa \} \\
\textit{baa} & \{ baa \} \\
\hline
\end{tabular}

See also, Fig. 3.7, EaC2e, p. 104
An Example

Consider the simple parentheses grammar

\begin{align*}
0 \quad Goal & \rightarrow \ List \\
1 \quad List & \rightarrow \ Pair \ List \\
2 \quad & \mid \ \varepsilon \\
3 \quad Pair & \rightarrow \ LP \ List \ RP
\end{align*}

where LP is ( and RP is )

\begin{center}
\begin{tabular}{|c|} 
\hline 
Symbol \hspace{2cm} Initial \\
\hline 
Goal \hspace{1cm} \emptyset \\
List \hspace{1cm} \emptyset \\
Pair \hspace{1cm} \emptyset \\
LP \hspace{0.5cm} LP \\
RP \hspace{0.5cm} RP \\
EOF \hspace{0.5cm} EOF \\
\hline
\end{tabular}
\end{center}
An Example

Consider the simple parentheses grammar

\[
\begin{align*}
0 & \quad \text{Goal} & \rightarrow & \text{List} \\
1 & \quad \text{List} & \rightarrow & \text{Pair} \ \text{List} \\
2 & \quad & | & \varepsilon \\
3 & \quad \text{Pair} & \rightarrow & \text{LP} \ \text{List} \ \text{RP}
\end{align*}
\]

where LP is ( and RP is )

- Iteration 1 adds LP to FIRST(Pair) and LP, ε to FIRST(List) & FIRST(Goal)
  → If we take them in order 3, 2, 1, 0
- Algorithm reaches fixed point†

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Initial</th>
<th>1\text{st}</th>
<th>2\text{nd}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>\emptyset</td>
<td>LP, ε</td>
<td>LP, ε</td>
</tr>
<tr>
<td>List</td>
<td>\emptyset</td>
<td>LP, ε</td>
<td>LP, ε</td>
</tr>
<tr>
<td>Pair</td>
<td>\emptyset</td>
<td>LP</td>
<td>LP</td>
</tr>
<tr>
<td>LP</td>
<td>LP</td>
<td>LP</td>
<td>LP</td>
</tr>
<tr>
<td>RP</td>
<td>RP</td>
<td>RP</td>
<td>RP</td>
</tr>
<tr>
<td>EOF</td>
<td>EOF</td>
<td>EOF</td>
<td>EOF</td>
</tr>
</tbody>
</table>

†In the adversarial order (0, 1, 2, 3), propagating \{LP, ε\} through Pair, List, and Goal would require one iteration for each set.
FIRST and FOLLOW Sets

**FIRST(α)**

For some $\alpha \in (T \cup NT \cup EOF \cup \varepsilon)^*$, define $\text{FIRST}(\alpha)$ as the set of tokens that appear as the first symbol in some string that derives from $\alpha$.

That is, $x \in \text{FIRST}(\alpha)$ iff $\alpha \Rightarrow^* x \gamma$, for some $\gamma$.

$\text{FIRST}$ is defined over strings of grammar symbols: $(T \cup NT \cup EOF \cup \varepsilon)^*$

**FOLLOW(A)**

For some $A \in NT$, define $\text{FOLLOW}(A)$ as the set of symbols that can occur immediately after $A$ in a valid sentential form.

$\text{FOLLOW}(S) = \{\text{EOF}\}$, where $S$ is the start symbol.

$\text{FOLLOW}$ is defined over the set of nonterminal symbols, $NT$.

To build $\text{FOLLOW}$ sets, we need $\text{FIRST}$ sets ...

**EOF ≜ end of file**
Computing **FOLLOW** Sets

for each $A \in \text{NT}$

$$\text{FOLLOW}(A) \leftarrow \emptyset$$

$\text{FOLLOW}(S) \leftarrow \{ \text{EOF} \}$

while (FOLLOW sets are still changing)

for each $p \in P$, of the form $A \rightarrow B_1B_2 \ldots B_k$

$$\text{TRAILER} \leftarrow \text{FOLLOW}(A)$$

for $i \leftarrow k$ down to 1

if $B_i \in \text{NT}$ then // domain check

$$\text{FOLLOW}(B_i) \leftarrow \text{FOLLOW}(B_i) \cup \text{TRAILER}$$

if $\varepsilon \in \text{FIRST}(B_i)$ // add right context

then TRAILER $\leftarrow$ TRAILER $\cup (\text{FIRST}(B_i) \setminus \{\varepsilon\})$

else TRAILER $\leftarrow$ FIRST($B_i$) // no $\varepsilon$ => truncate the right context

else TRAILER $\leftarrow \{B_i\}$ // $B_i \in T$ => only 1 symbol

---

**Figure 3.8, page 106, EaC2e**
Computing **FOLLOW** Sets

This algorithm has a completely different feel than computing **FIRST** sets

For a production $A \rightarrow B_1 B_2 \ldots B_k$:

- It works its way backward through the production: $B_k, B_{k-1}, \ldots B_1$
- It builds the **FOLLOW** sets for the *rhs* symbols, $B_1, B_2, \ldots B_k$, not $A$
- In the absence of $\varepsilon$, **FOLLOW**($B_i$) is just **FIRST**($B_{i+1}$)
  - *As always, $\varepsilon$ makes the algorithm more complex*

To handle $\varepsilon$, the algorithm keeps track of the *first word* in the trailing right context as it works its way back through the *rhs*: $B_k, B_{k-1}, \ldots B_1$

- It uses **FOLLOW**($A$) to initialize the *Trailer* for $B_k$
  - That use is the only mention of **FOLLOW**($A$) in the algorithm
- *Trailer* approximates the **FIRST**$^+$ set for the trailing left context
### An Example

Consider, again, the simple parentheses grammar

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal → List</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>List → Pair List</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>ε</td>
</tr>
<tr>
<td>3</td>
<td>Pair → LP List RP</td>
<td></td>
</tr>
</tbody>
</table>

**Initial Values:**

- *Goal*, *List*, and *Pair* are set to $\emptyset$
- *Goal* is then set to \{ *EOF* \}

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FOLLOW Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Initial</td>
</tr>
<tr>
<td>Goal</td>
<td>EOF</td>
</tr>
<tr>
<td>List</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>Pair</td>
<td>$\emptyset$</td>
</tr>
</tbody>
</table>
An Example

Consider, again, the simple parentheses grammar

|   | Goal → List
|---|---|
| 1 | List → Pair List
| 2 | | ε
| 3 | Pair → LP List RP

**Iteration 1:**
- Production 0 adds **EOF** to FOLLOW(List)
- Production 1 adds **LP** to FOLLOW(Pair)
  → from **FIRST(List)**
- Production 2 does nothing
- Production 3 adds **RP** to FOLLOW(List)
  → from **FIRST(RP)**

**FOLLOW Sets**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Initial</th>
<th>1&lt;sup&gt;st&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>EOF</td>
<td>EOF</td>
</tr>
<tr>
<td>List</td>
<td>φ</td>
<td>EOF, RP</td>
</tr>
<tr>
<td>Pair</td>
<td>φ</td>
<td>EOF, LP</td>
</tr>
</tbody>
</table>

**FIRST**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>LP, ε</td>
</tr>
<tr>
<td>List</td>
<td>LP, ε</td>
</tr>
<tr>
<td>Pair</td>
<td>LP</td>
</tr>
<tr>
<td>LP</td>
<td>LP</td>
</tr>
<tr>
<td>RP</td>
<td>RP</td>
</tr>
<tr>
<td>EOF</td>
<td>EOF</td>
</tr>
</tbody>
</table>
Consider, again, the simple parentheses grammar

<table>
<thead>
<tr>
<th></th>
<th>Goal</th>
<th>List</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal</td>
<td>List</td>
</tr>
<tr>
<td>1</td>
<td>List</td>
<td>Pair  List</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Pair</td>
<td>LP    List RP</td>
</tr>
</tbody>
</table>

Iteration 2:
- Production 0 adds nothing new
- Production 1 adds RP to FOLLOW(Pair)
  - from FOLLOW(List), $\varepsilon \in$ FIRST(List)
- Production 2 does nothing
- Production 3 adds nothing new

Iteration 3 produces the same result $\Rightarrow$ reached a fixed point
## Classic Expression Grammar

<table>
<thead>
<tr>
<th>#</th>
<th>Symbol</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal</td>
<td>$\rightarrow$ Expr</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Expr</td>
<td>$\rightarrow$ Term Expr'</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Expr'</td>
<td>$\rightarrow$ + Term Expr'</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>$\mid$ - Term Expr'</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>$\mid$ ε</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Term</td>
<td>$\rightarrow$ Factor Term'</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Term'</td>
<td>$\rightarrow$ * Factor Term'</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>$\mid$ / Factor Term'</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>$\mid$ ε</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Factor</td>
<td>$\rightarrow$ ( Expr )</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>$\mid$ number</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>$\mid$ identifier</td>
<td></td>
</tr>
</tbody>
</table>

**FIRST**$(A \rightarrow \beta)$ is identical to **FIRST**$(\beta)$ except for productions 4 and 8

**FIRST**$(\text{Expr'} \rightarrow \epsilon)$ is \{\epsilon,\}, \text{eof}

**FIRST**$(\text{Term'} \rightarrow \epsilon)$ is \{\epsilon,+,\-, \}, \text{eof}
### Classic Expression Grammar

<table>
<thead>
<tr>
<th>Prod’n</th>
<th>FIRST⁺</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>(id,num</td>
</tr>
<tr>
<td>4</td>
<td>ε</td>
</tr>
<tr>
<td>8</td>
<td>ε</td>
</tr>
<tr>
<td>11</td>
<td>identifier</td>
</tr>
</tbody>
</table>
Recursive Descent Parsing

A couple of routines from the expression parser

**Goal( )**

- \( \text{token} \leftarrow \text{next_token}( ) \);
- \( \text{if (Expr( ) = true & token = EOF)} \)
  - then next compilation step;
  - else
    - report syntax error;
    - return false;

**Expr( )**

- \( \text{if (Term( ) = false)} \)
  - then return false;
  - else return \text{Eprime}( ) ;

**Factor( )**

- \( \text{if (token = number)} \) then
  - \( \text{token} \leftarrow \text{next_token}( ) ; \)
  - return true;
- \( \text{else if (token = identifier)} \) then
  - \( \text{token} \leftarrow \text{next_token}( ) ; \)
  - return true;
- \( \text{else if (token = lparen)} \)
  - \( \text{token} \leftarrow \text{next_token}( ) ; \)
  - \( \text{if (Expr( ) = true & token = rparen)} \) then
    - \( \text{token} \leftarrow \text{next_token}( ) ; \)
    - return true;
  - // fall out of if statement
  - report syntax error;
  - return false;

**EPrime, Term, & TPrime follow the same basic lines (Figure 3.10, EaC2e)**
Implementing a Recursive Descent Parser

A nest of if-then else statements may be slow

- A good case statement would be an improvement†
  - See EaC2e, § 7.8.3
  - Encode with computation rather than repeated branches
- Order the cases by expected frequency, to drop average cost

What about encoding the decisions in a table?

- Replace if then else or case statement with an address computation
- Branches are slow and disruptive
- Interpret the table with a skeleton parser, as we did in scanning

† a good case statement can be hard to find
Building Table-Driven Top-down Parsers

**Strategy**
- Encode knowledge in a table
- Use a standard “skeleton” parser to interpret the table

**Example**
- The non-terminal `Factor` has 3 expansions
  - `(Expr)` or `Identifier` or `Number`
- Table might look like:

```
<table>
<thead>
<tr>
<th></th>
<th>Goal</th>
<th>Expr</th>
<th>Term Expr'</th>
<th>+ Term Expr'</th>
<th>- Term Expr'</th>
<th>ε</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal</td>
<td>Expr</td>
<td>Term Expr'</td>
<td>+ Term Expr'</td>
<td>- Term Expr'</td>
<td>ε</td>
</tr>
<tr>
<td>1</td>
<td>Expr</td>
<td>Term Expr'</td>
<td>+ Term Expr'</td>
<td>- Term Expr'</td>
<td>ε</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Term</td>
<td>Factor Term'</td>
<td>* Factor Term'</td>
<td>/ Factor Term'</td>
<td>ε</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Term'</td>
<td>Factor Term'</td>
<td>* Factor Term'</td>
<td>/ Factor Term'</td>
<td>ε</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Factor</td>
<td>(Expr)</td>
<td>number</td>
<td>identifier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

**Terminal Symbols**

<table>
<thead>
<tr>
<th></th>
<th>EOF</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>(</th>
<th>)</th>
<th>id.</th>
<th>num.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factor</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>9</td>
<td>—</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

COMP 412, Fall 2017
Building Top-down Parsers

Building the complete table

- Need a row for every $NT$ & a column for every $T$
### LL(1) Table for the Expression Grammar

<table>
<thead>
<tr>
<th></th>
<th>EOF</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>(</th>
<th>)</th>
<th>id.</th>
<th>num.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Goal</strong></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>0</td>
<td>—</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Expr</strong></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>1</td>
<td>—</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td><strong>Expr'</strong></td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>4</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>Term</strong></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>5</td>
<td>—</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td><strong>Term'</strong></td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>6</td>
<td>7</td>
<td>—</td>
<td>8</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>Factor</strong></td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>9</td>
<td>—</td>
<td>11</td>
</tr>
</tbody>
</table>

*Row we built earlier*
Building Top-down Parsers

Building the complete table

• Need a row for every $NT$ & a column for every $T$
• Need an interpreter for the table ($skeleton$ $parser$)
word ← NextWord()  // Initial conditions, including
push EOF onto Stack   // a stack to track local goals
push the start symbol, S, onto Stack
TOS ← top of Stack

loop forever
  if TOS = EOF and word = EOF then
    break & report success   // exit on success
  else if TOS is a terminal then
    if TOS matches word then
      pop Stack  // recognized TOS
      word ← NextWord()
      else report error looking for TOS  // error exit
    else  // TOS is a non-terminal
      if TABLE[TOS,word] is A → B₁B₂...Bₖ then
        pop Stack  // get rid of A
        push Bₖ, Bₖ₋₁, ..., B₁  // in that order
      else break & report error expanding TOS
  TOS ← top of Stack
Building Top-down Parsers

Building the complete table

- Need a row for every $NT$ & a column for every $T$
- Need a table-driven interpreter for the table
- Need an algorithm to build the table

Filling in $\text{TABLE}[X,y]$, $X \in NT$, $y \in T$

1. entry is the rule $X \rightarrow \beta$, if $y \in \text{FIRST}^+(X \rightarrow \beta)$
2. entry is $\text{error}$ if rule 1 does not define

If any entry has more than one rule, $G$ is not $LL(1)$

This algorithm is the $LL(1)$ table construction algorithm

Incrementally tests the $LL(1)$ criterion on each $NT$.
An efficient way to determine if a grammar is $LL(1)$

In Lab 2, you would have built a recursive descent parser for a modified form of $BNF$ and build $LL(1)$ tables for the grammars that are $LL(1)$. (A good weekend project)
Recap of Top-down Parsing

• Top-down parsers build syntax tree from root to leaves

• Left-recursion causes non-termination in top-down parsers
  – Transformation to eliminate left recursion
  – Transformation to eliminate common prefixes in right recursion

• FIRST, FIRST⁺, & FOLLOW sets + LL(1) condition
  – LL(1) uses left-to-right scan of the input, leftmost derivation of the sentence, and 1 word lookahead
  – LL(1) condition means grammar works for predictive parsing

• Given an LL(1) grammar, we can
  – Build a recursive descent parser
  – Build a table-driven LL(1) parser

• LL(1) parser doesn’t build the parse tree
  – Keeps lower fringe of partially complete tree on the stack