Midterm Exam: Thursday
October 18, 7PM
Herzstein Amphitheater

Updated slides posted for lectures 10 & 11

Syntax Analysis, IV

Comp 412

Copyright 2018, 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.
### Lab 2 Schedule

**N.B.:** Code checks 1 & 2 are milestones, not exhaustive tests. Many students still find bugs after code check 2, to say nothing of tuning for effectiveness & efficiency.

<table>
<thead>
<tr>
<th>Sunday</th>
<th>Monday</th>
<th>Tuesday</th>
<th>Wednesday</th>
<th>Thursday</th>
<th>Friday</th>
<th>Saturday</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Lab 2 Specs available</td>
</tr>
<tr>
<td></td>
<td></td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Deadline: Code Check 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>18 Tutorial 5 PM McMurry</td>
<td>21</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>24</td>
<td>25 Dan Grove Talk (Dart Group)</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Advice:** Pay attention to the timing / scaling tests.

**Focus on connecting to Lab 1 IR, and Rename**

**Allocate correctly**

**Improve performance & allocation**
Review

Last Parsing Lecture (Last Wednesday)

- Introduced FIRST, FOLLOW, and \( \text{FIRST}^+ \) sets
- Introduced the LL(1) condition

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

\[
\text{FIRST}^+(A \rightarrow \beta) \cap \text{FIRST}^+(A \rightarrow \gamma) = \emptyset \]

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

We did not discuss

- An algorithm to construct FIRST sets
- An algorithm to construct FOLLOW sets
FIRST and FOLLOW Sets

**FIRST(α)**

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

That is, $x \in \text{FIRST(α)}$ 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

$\text{FOLLOW}(S) = \{EOF\}$, 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 ...

$EOF \cong$ end of file
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 $\gamma$

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

$\text{FOLLOW}(S) = \{EOF\}$, 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 FIRST Sets

for each
x ∈ (T ∪ EOF ∪ ε)
FIRST(x) ← {x}

for each
A ∈ NT, FIRST(A) ← ∅

while (FIRST sets are still changing)
do

for each
p ∈ P, of the form
A → B₁B₂...Bₖ
do

rhs ← FIRST(B₁) − {ε}
Some details go here to handle ε productions
FIRST(A) ← FIRST(A) ∪ rhs

end // for loop

end // while loop

To begin, we will ignore ε productions

• Initialize FIRST set for each terminal EOF, ε, and nonterminal

• Then, loop through the productions and add the FIRST set for the leading symbol on the rhs to the FIRST set of the lhs (a nonterminal)

• Because rhs can start with a nonterminal, iterate to a fixed point
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 $A \rightarrow B_1B_2\ldots B_k$ do
    \[ \text{rhs} \leftarrow \text{FIRST}(B_1) - \{\varepsilon\} \]
    Some details go here to handle $\varepsilon$ productions
    \[ \text{FIRST}(A) \leftarrow \text{FIRST}(A) \cup \text{rhs} \]
  end  // for loop
end  // while loop

Loop nest is \textit{monotone increasing} for FIRST sets

- Outer loop is bounded by the number of grammar symbols:
  \[ |T \cup NT \cup EOF \cup \varepsilon| \]
  which is finite
- A single iteration of the while loop is bounded by the size of the grammar
- \textit{For loop} is bounded by the $|productions|$
Filling in the Details: Computing \textbf{FIRST} Sets

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

\varepsilon \text{ complicates matters}

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

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

See also, Fig. 3.7, EaC2e, p. 104
Computing **FIRST** Sets

for each \( x \in (T \cup EOF \cup \varepsilon) \)

\[
FIRST(x) \leftarrow \{x\}
\]

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

while (FIRST sets are still changing) do

for each \( p \in P, \) of the form \( A \rightarrow B_1B_2...B_k \) do

\[
rhs \leftarrow FIRST(B_1) - \{\varepsilon\}
\]

for \( i \leftarrow 1 \) to \( k-1 \) by 1 while \( \varepsilon \in FIRST(B_i) \) do

\[
rhs \leftarrow rhs \cup (FIRST(B_{i+1}) - \{\varepsilon\})
\]

end // for loop

if \( i = k \) and \( \varepsilon \in FIRST(B_k) \)

then \( rhs \leftarrow rhs \cup \{\varepsilon\} \)

\[
FIRST(A) \leftarrow FIRST(A) \cup rhs
\]

end // for loop

end // while loop

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

**Loop nest is monotone increasing for FIRST sets**

- Handling \( \varepsilon \)-productions adds a loop that iterates over the productions
- Adds a multiplier of the sum of the length of the productions
- Slower, but still finite
- Still terminates

Remember, we pay this cost at **build time**
Clear 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>SheepNoise</td>
<td>{ baa }</td>
</tr>
<tr>
<td>baa</td>
<td>{ baa }</td>
</tr>
</tbody>
</table>
Computing FIRST Sets

for each \( x \in (T \cup EOF \cup \varepsilon) \)

\[
\text{FIRST}(x) \leftarrow \{x\}
\]

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

while (FIRST sets are still changing) do
  for each \( p \in P \), of the form \( A \rightarrow B_1B_2...B_k \) do
    \( rhs \leftarrow \text{FIRST}(B_1) - \{\varepsilon\} \)
    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
  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>Goal</td>
<td>\emptyset</td>
</tr>
<tr>
<td>SheepNoise</td>
<td>\emptyset</td>
</tr>
<tr>
<td>baa</td>
<td>{ baa }</td>
</tr>
</tbody>
</table>
Computing \textbf{FIRST} Sets

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

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

\textbf{Production 2}

(1) sets $\text{rhs}$ to $\text{FIRST}\{\text{baa}\}$ &
(2) copies $\text{rhs}$ into $\text{FIRST}(\text{SheepNoise})$

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

*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* $p \in P$, of the form $A \rightarrow B_1B_2...B_k$ *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*

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

---

**Production 0**

1. sets rhs to $\text{FIRST}(\text{Sheepnoise})$ &
2. copies rhs into $\text{FIRST}(\text{Goal})$

... and one more iteration to recognize that the FIRST sets have stopped changing

---

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>{ baa }</td>
</tr>
<tr>
<td>SheepNoise</td>
<td>{ baa }</td>
</tr>
<tr>
<td>baa</td>
<td>{ baa }</td>
</tr>
</tbody>
</table>
An Example

Consider the simple parentheses grammar

0. \( Goal \rightarrow List \)
1. \( List \rightarrow Pair \ List \)
2. \( Pair \rightarrow LP \ List \ RP \)

where LP is ( and RP is )

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Initial</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>List</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>Pair</td>
<td>( \emptyset )</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>
An Example

Consider the simple parentheses grammar

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

where LP is \( \{ \) and RP is \( \} \)

- Iteration 1 adds LP to FIRST(Pair) and LP, \( \varepsilon \) to FIRST(List) & FIRST(Goal)
  \( \rightarrow \) *If we take them in order 3, 2, 1, 0*

- Algorithm reaches fixed point\(^\dagger\)

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Initial</th>
<th>1(^{st})</th>
<th>2(^{nd})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Goal</td>
<td>( \emptyset )</td>
<td>LP, ( \varepsilon )</td>
<td>LP, ( \varepsilon )</td>
</tr>
<tr>
<td>List</td>
<td>( \emptyset )</td>
<td>LP, ( \varepsilon )</td>
<td>LP, ( \varepsilon )</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>

\(^\dagger\) In the adversarial order (0, 1, 2, 3), propagating \{LP, \( \varepsilon \}\} through would require an iteration to reach List and an iteration to reach Goal, plus an iteration to realize that they stopped changing.
### FIRST and FOLLOW Sets

**FIRST(α)**

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

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

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

**FOLLOW(A)**

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

$\text{FOLLOW}(S) = \{EOF\}$, 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 ...

$EOF ≡ \text{end of file}$
Computing **FOLLOW** Sets

for each \( A \in NT \)
\[
FOLLOW(A) \leftarrow \emptyset
\]

\[
FOLLOW(S) \leftarrow \{ \text{EOF} \}
\]

while (FOLLOW sets are still changing)

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

\[
\text{TRAILER} \leftarrow FOLLOW(A)
\]

for \( i \leftarrow k \) down to 1

if \( B_i \in NT \) then

\[
\text{FOLLOW}(B_i) \leftarrow \text{FOLLOW}(B_i) \cup \text{TRAILER}
\]

if \( \varepsilon \in \text{FIRST}(B_i) \) then

\[
\text{TRAILER} \leftarrow \text{TRAILER} \cup (\text{FIRST}(B_i) \setminus \{\varepsilon\})
\]

else

\[
\text{TRAILER} \leftarrow \text{FIRST}(B_i)
\]

else

\[
\text{TRAILER} \leftarrow \{B_i\}
\]

Don’t add \( \varepsilon \)

// domain check

// add right context

// no \( \varepsilon \) => truncate the right context

// \( B_i \in T \) => only 1 symbol
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 $\varepsilon$, $\text{FOLLOW}(B_i)$ is just $\text{FIRST}(B_{i+1})$
  - 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 $\text{FOLLOW}(A)$ to initialize the *Trailer* for $B_k$
  - That use is the only mention of $\text{FOLLOW}(A)$ in the algorithm
- *Trailer* approximates the $\text{FIRST}^*$ set for the trailing right context
An Example

Consider, again, the simple parentheses grammar

<table>
<thead>
<tr>
<th></th>
<th>Symbol</th>
<th>FOLLOW Sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal \rightarrow List</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>List \rightarrow Pair List</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>\varepsilon</td>
<td>EOF</td>
</tr>
<tr>
<td>3</td>
<td>Pair \rightarrow LP List RP</td>
<td>EOF, LP</td>
</tr>
</tbody>
</table>

Initial Values:

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

Consider, again, the simple parentheses grammar

<table>
<thead>
<tr>
<th>Iteration 1:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Production 0 adds <strong>EOF</strong> to FOLLOW(<em>List</em>)</td>
</tr>
</tbody>
</table>
| • Production 1 adds **LP** to FOLLOW(*Pair*)  
  → from FIRST(*List*) |
| • Production 2 does nothing |
| • Production 3 adds **RP** to FOLLOW(*List*)  
  → from FIRST(*RP*) |

COMP 412, Fall 2018
An Example

Consider, again, the simple parentheses grammar

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>\textit{Goal}</td>
<td>\rightarrow</td>
<td>\textit{List}</td>
</tr>
<tr>
<td>1</td>
<td>\textit{List}</td>
<td>\rightarrow</td>
<td>\textit{Pair} \textit{List}</td>
</tr>
<tr>
<td>2</td>
<td>\textit{Pair}</td>
<td>\rightarrow</td>
<td>\textit{LP} \textit{List} \textit{RP}</td>
</tr>
</tbody>
</table>

\begin{tabular}{|c|c|c|c|}
\hline
\textbf{Symbol} & \textbf{Initial} & \textbf{\textit{1}st} Set & \textbf{\textit{2}nd} Set \\
\hline
\textit{Goal} & \textit{EOF} & \textit{EOF} & \textit{EOF} \\
\textit{List} & \emptyset & \textit{EOF, RP} & \textit{EOF, RP} \\
\textit{Pair} & \emptyset & \textit{EOF, LP} & \textit{EOF, LP, RP} \\
\hline
\end{tabular}

\textbf{Iteration 2:}

- Production 0 adds nothing new
- Production 1 adds \textit{RP} to \textit{FOLLOW(Pair)}
  \rightarrow \textit{from} \textit{FOLLOW(List), }\varepsilon \in \textit{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>Symbol</th>
<th>FIRST</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>num</td>
<td>Ø</td>
</tr>
<tr>
<td>id</td>
<td>id</td>
<td>Ø</td>
</tr>
<tr>
<td>+</td>
<td>+</td>
<td>Ø</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>Ø</td>
</tr>
<tr>
<td>*</td>
<td>*</td>
<td>Ø</td>
</tr>
<tr>
<td>/</td>
<td>/</td>
<td>Ø</td>
</tr>
<tr>
<td>(</td>
<td>(</td>
<td>Ø</td>
</tr>
<tr>
<td>)</td>
<td>)</td>
<td>Ø</td>
</tr>
<tr>
<td>eof</td>
<td>eof</td>
<td>Ø</td>
</tr>
<tr>
<td>ε</td>
<td>ε</td>
<td>Ø</td>
</tr>
</tbody>
</table>

FIRST\(^+(A\rightarrowβ)\) is identical to FIRST(β) except for productions 4 and 8.

FIRST\(^+(Expr'\rightarrowε)\) is \{ε,\}, eof

FIRST\(^+(Term'\rightarrowε)\) is \{ε,+,-,\}, eof

FIRST\(^+(Factor\rightarrowε)\) is \{ε,+,-,\}, eof

FIRST\(^+(Expr\rightarrowε)\) is \{ε,+,-,\}, 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>1</td>
<td>{,id,num}</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>ε</td>
</tr>
<tr>
<td>5</td>
<td>{,id,num}</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>/</td>
</tr>
<tr>
<td>8</td>
<td>ε,+,-,},,eof</td>
</tr>
<tr>
<td>9</td>
<td>{</td>
</tr>
<tr>
<td>10</td>
<td>number</td>
</tr>
<tr>
<td>11</td>
<td>identifier</td>
</tr>
</tbody>
</table>

**Goal** → Expr

**Expr** → Term Expr’

**Expr’** → + Term Expr’

<table>
<thead>
<tr>
<th>Prod’n</th>
<th>FIRST$^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{,id,num}</td>
</tr>
<tr>
<td>1</td>
<td>{,id,num}</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>ε</td>
</tr>
<tr>
<td>5</td>
<td>{,id,num}</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>/</td>
</tr>
<tr>
<td>8</td>
<td>ε,+,-,},,eof</td>
</tr>
<tr>
<td>9</td>
<td>{</td>
</tr>
<tr>
<td>10</td>
<td>number</td>
</tr>
<tr>
<td>11</td>
<td>identifier</td>
</tr>
</tbody>
</table>

**Term** → Factor Term’

**Term’** → * Factor Term’

<table>
<thead>
<tr>
<th>Prod’n</th>
<th>FIRST$^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{,id,num}</td>
</tr>
<tr>
<td>1</td>
<td>{,id,num}</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>ε</td>
</tr>
<tr>
<td>5</td>
<td>{,id,num}</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>/</td>
</tr>
<tr>
<td>8</td>
<td>ε,+,-,},,eof</td>
</tr>
<tr>
<td>9</td>
<td>{</td>
</tr>
<tr>
<td>10</td>
<td>number</td>
</tr>
<tr>
<td>11</td>
<td>identifier</td>
</tr>
</tbody>
</table>

**Factor** → ( Expr )

**Factor** → ( Expr )

<table>
<thead>
<tr>
<th>Prod’n</th>
<th>FIRST$^+$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{,id,num}</td>
</tr>
<tr>
<td>1</td>
<td>{,id,num}</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>ε</td>
</tr>
<tr>
<td>5</td>
<td>{,id,num}</td>
</tr>
<tr>
<td>6</td>
<td>*</td>
</tr>
<tr>
<td>7</td>
<td>/</td>
</tr>
<tr>
<td>8</td>
<td>ε,+,-,},,eof</td>
</tr>
<tr>
<td>9</td>
<td>{</td>
</tr>
<tr>
<td>10</td>
<td>number</td>
</tr>
<tr>
<td>11</td>
<td>identifier</td>
</tr>
</tbody>
</table>

**Number**

**Identifier**
A couple of routines from the expression parser

\[ \text{Goal( )} \]
\[
\begin{align*}
token & \leftarrow \text{next\_token( )}; \\
\text{if (Expr( ) = true & token = EOF)} & \quad \text{then next compilation step; } \\
\text{else} & \\
\quad \text{report syntax error; } \\
\quad \text{return false;}
\end{align*}
\]

\[ \text{Expr( )} \]
\[
\begin{align*}
\text{if (Term( ) = false)} & \\
\quad \text{then return false; } \\
\text{else return Eprime( );}
\end{align*}
\]

\[ \text{Factor( )} \]
\[
\begin{align*}
\text{if (token = number) then} & \\
\quad \text{token } \leftarrow \text{next\_token( ); } \\
\quad \text{return true;}
\end{align*}
\]
\[
\begin{align*}
\text{else if (token = identifier) then} & \\
\quad \text{token } \leftarrow \text{next\_token( ); } \\
\quad \text{return true;}
\end{align*}
\]
\[
\begin{align*}
\text{else if (token = lparen)} & \\
\quad \text{token } \leftarrow \text{next\_token( ); } \\
\text{if (Expr( ) = true & token = rparen)} & \\
\quad \text{then} \\
\quad \text{token } \leftarrow \text{next\_token( ); } \\
\quad \text{return true; } \\
\quad \text{// fall out of if statement} \\
\text{report syntax error; } \\
\quad \text{return false;}
\end{align*}
\]

Looking for number, identifier, or (, found token instead, or failed to find \textit{Expr} or ) after (.

\[ \text{EPrime, Term, & TPrime follow the same basic lines (Figure 3.10, EaC2e)} \]
Review

Recursive Descent

Page 111 in EaC2e sketches a recursive descent parser for the RR CEG.

- One routine per NT
- Check each RHS by checking each symbol
- Includes ε-productions

Review from last lecture

```
Main()
    /* Goal → Expr */
    word ← NextWord();
    if (Expr())
        then if (word = eof)
            then report success;
        else Fail();

Fail()
    report syntax error;
    attempt error recovery or exit;

Expr()
    /* Expr → Term Expr' */
    if (Term())
        then return EPrime();
    else Fail();

EPrime()
    /* Expr' → + Term Expr' */
    /* Expr' → - Term Expr' */
    if (word = + or word = -)
        then begin;
            word ← NextWord();
            if (Term())
                then return EPrime();
            else Fail();
        end;
    else if (word = ) or word = eof)
        /* Expr' → ε */
        then return true;
    else Fail();

Term()
    /* Term → Factor Term' */
    if (Factor())
        then return TPrime();
    else Fail();

TPrime()
    /* Term' → × Factor Term' */
    /* Term' → ÷ Factor Term' */
    if (word = × or word = ÷)
        then begin;
            word ← NextWord();
            if (Factor())
                then return TPrime();
            else Fail();
        end;
    else if (word = + or word = - or
            word = ) or word = eof)
        /* Term' → ε */
        then return true;
    else Fail();

Factor()
    /* Factor → ( Expr ) */
    if (word = ( ) then begin;
        word ← NextWord();
        if (not Expr())
            then Fail();
        if (word ≠ )
            then Fail();
        word ← NextWord();
        return true;
    end;
    /* Factor → num */
    /* Factor → name */
    else if (word = num or
            word = name)
        then begin;
            word ← NextWord();
            return true;
        end;
    else Fail();
```
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

† a good case statement can be hard to find
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
  - \( (\text{Expr}) \) or Identifier or Number
- Table might look like:

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

Cannot expand Factor into an operator \( \Rightarrow \) error

Expand Factor by rule 10 with input “number”

COMP 412, Fall 2018
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>9</td>
<td>—</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

*Figure 3.11(b), page 112, EaC2e*
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*)
LL(1) 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 TABLE[\(X, y\)], \(X \in NT\), \(y \in T\)

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

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

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

This algorithm is the LL(1) table construction algorithm

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

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

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

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