Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel
Top-Down Parsing
Top-down Parsing Algorithm

Construct the root node of the parse tree, label it with the start symbol, and set the current-node to root node

Repeat until all the input is consumed (i.e., until the frontier of the parse tree matches the input string)

1. If the label of the current node is a non-terminal node A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child

2. If the current node is a terminal symbol:
   - If it matches the input string, consume it (advance the input pointer)
   - If it does not match the input string, backtrack

3. Set the current node to the next node in the frontier of the parse tree
   - If there is no node left in the frontier of the parse tree and input is not consumed, then backtrack

The key is picking the right production in step 1
- That choice should be guided by the input string
Predictive Parsing

• The main idea is to look ahead at the next token and use that token to pick the production that you should apply

\[ X \rightarrow + X \]
\[ | - Y \]

Here we can use the + and – to decide which rule to apply

This technique is more general!

• Definition of FIRST sets

\[ x \in \text{FIRST}(\alpha) \iff \alpha \Rightarrow^* x \gamma, \text{ for some } \gamma \]

(\(\Rightarrow^*\) means a series of (0 or more) productions)

• This means that we have to examine ALL tokens that our productions could potentially start with…
FIRST Sets

• Intuitively, FIRST(S) is the set of all terminals that we could possibly see when starting to parse S

• If we want to build a predictive parser, we need to make sure that the look-ahead token tells us with 100% confidence which production to apply

• In order for this to be true, anytime we have a production that looks like $A \rightarrow \alpha \mid \beta$, we need to make sure that FIRST($\alpha$) is distinct from the FIRST($\beta$)

• “Distinct” means that there is no element in FIRST($\alpha$) that is also in FIRST($\beta$) … or formally, that FIRST($\alpha$) $\cap$ FIRST($\beta$) = {}
Slightly More Tricky Examples

- Here is an example of FIRST sets where the first symbol in the production is a non-terminal

- In this case, we have to examine all possible terminals that could begin a sentence derived from S

- If we have an ε, then we need to look past the first non-terminal

- If all the non-terminals have ε in their first sets, then add ε to the first set

\[
S \rightarrow AB \\
A \rightarrow x \mid y \\
B \rightarrow 0 \mid 1
\]

\[
\text{FIRST}(S) = \{ x, y \}
\]

\[
S \rightarrow AB \\
A \rightarrow x \mid y \mid \epsilon \\
B \rightarrow 0 \mid 1
\]

\[
\text{FIRST}(S) = \{ x, y, 0, 1 \}
\]

\[
S \rightarrow AB \\
A \rightarrow x \mid y \mid \epsilon \\
B \rightarrow 0 \mid 1 \mid \epsilon
\]

\[
\text{FIRST}(S) = \{ x, y, 0, 1, \epsilon \}
\]
How to Generate FIRST Sets

For a string of grammar symbols $\alpha$, define $\text{FIRST}(\alpha)$ as

- Set of tokens that appear as the first symbol in some string that derives from $\alpha$
- If $\alpha \Rightarrow^* \varepsilon$, then $\varepsilon$ is in $\text{FIRST}(\alpha)$

To construct $\text{FIRST}(X)$ for a grammar symbol $X$, apply the following rules until no more symbols can be added to $\text{FIRST}(X)$

- If $X$ is a terminal $\text{FIRST}(X)$ is \{X\}
- If $X \rightarrow \varepsilon$ is a production, then $\varepsilon$ is in $\text{FIRST}(X)$
- If $X$ is a non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then put every symbol in $\text{FIRST}(Y_i)$ other than $\varepsilon$ to $\text{FIRST}(X)$
- If $X$ is a non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then put terminal $a$ in $\text{FIRST}(X)$ if $a$ is in $\text{FIRST}(Y_i)$ and $\varepsilon$ is in $\text{FIRST}(Y_j)$ for all $1 \leq j < I$
- If $X$ is a non-terminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production, then put $\varepsilon$ in $\text{FIRST}(X)$ if $\varepsilon$ is in $\text{FIRST}(Y_i)$ for all $1 \leq i \leq k$
Computing FIRST Sets

To construct the FIRST set for any string of grammar symbols $X_1X_2 \ldots X_n$ (given the FIRST sets for symbols $X_1$, $X_2$, ... $X_n$), apply the following rules.

\[
\text{FIRST}(X_1X_2 \ldots X_n) \text{ contains:}
\]

- any symbol in FIRST($X_1$) other than $\varepsilon$
- any symbol in FIRST($X_i$) other than $\varepsilon$, if $\varepsilon$ is in FIRST($X_j$) for all $1 \leq j < i$
- $\varepsilon$, if $\varepsilon$ is in FIRST($X_i$) for all $1 \leq i \leq n$
### FIRST Sets

<table>
<thead>
<tr>
<th></th>
<th>Symbol</th>
<th>FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Expr</td>
<td>num, id</td>
</tr>
<tr>
<td>2</td>
<td>Expr</td>
<td>num, id</td>
</tr>
<tr>
<td>3</td>
<td>Expr'</td>
<td>ε, +, -</td>
</tr>
<tr>
<td>4</td>
<td>Term</td>
<td>num, id</td>
</tr>
<tr>
<td>5</td>
<td>Term'</td>
<td>ε, *, /</td>
</tr>
<tr>
<td>6</td>
<td>Factor</td>
<td>num, id</td>
</tr>
<tr>
<td>7</td>
<td>Factor</td>
<td>num, id</td>
</tr>
</tbody>
</table>

**Grammar Rules:**

1. $S \rightarrow Expr$
2. $Expr \rightarrow Term \ Expr'$
3. $Expr' \rightarrow + \ Term \ Expr'$
4. $Expr' \rightarrow - \ Term \ Expr'$
5. $Expr' \rightarrow ε$
6. $Term \rightarrow Factor \ Term'$
7. $Term' \rightarrow * \ Factor \ Term'$
8. $Term' \rightarrow / \ Factor \ Term'$
9. $Term' \rightarrow ε$
10. $Factor \rightarrow num$
11. $Factor \rightarrow id$
We still have those pesky epsilons …

Despite our efforts to look past all of the $\varepsilon$ when defining our FIRST sets, sometime we still have $\varepsilon$ in our FIRST sets (as in the above example). So, what can we do?

The trick to doing it is to look past the current non-terminal and examine the set of characters that can follow the current non-terminal.

This is what the FOLLOW set defines.

We use the special character $\$ to denote the end of the file.

$S \rightarrow AB$

$A \rightarrow x | y | \varepsilon$

$B \rightarrow 0 | 1 | \varepsilon$

FIRST(S) = \{ x, y, 0, 1, \varepsilon \}
FOLLOW Sets

For a non-terminal symbol $A$, define $\text{FOLLOW}(A)$:

The set of terminal symbols that can appear immediately to the right of $A$ in some sentential form

To construct $\text{FOLLOW}(A)$ for a non-terminal symbol $A$, apply the following rules until no more symbols can be added to $\text{FOLLOW}(A)$:

- Place $\$ \in \text{FOLLOW}(S)$ ($\$ \text{ is the end-of-file symbol, } S \text{ is the start symbol}$)
- If there is a production $A \rightarrow \alpha \ B \ \beta$, then everything in $\text{FIRST}(\beta)$ - except $\varepsilon$ - is placed in $\text{FOLLOW}(B)$
- If there is a production $A \rightarrow \alpha \ B$, then everything in $\text{FOLLOW}(A)$ is placed in $\text{FOLLOW}(B)$
- If there is a production $A \rightarrow \alpha \ B \ \beta$, and $\varepsilon$ is in $\text{FIRST}(\beta)$, then everything in $\text{FOLLOW}(A)$ is placed in $\text{FOLLOW}(B)$
FOLLOW Sets

1. $ S \rightarrow Expr$
2. $ Expr \rightarrow Term Expr'$
3. $ Expr' \rightarrow + Term Expr'$
4. $ - Term Expr'$
5. $\epsilon$
6. $ Term \rightarrow Factor Term'$
7. $ Term' \rightarrow * Factor Term'$
8. $ \mid l Factor Term'$
9. $\epsilon$
10. $ Factor \rightarrow num$
11. $\mid id$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ S $</td>
<td>{ $ }</td>
</tr>
<tr>
<td>$ Expr $</td>
<td>{ $ }</td>
</tr>
<tr>
<td>$ Expr'$</td>
<td>{ $ }</td>
</tr>
<tr>
<td>$ Term $</td>
<td>{ $, +, - }</td>
</tr>
<tr>
<td>$ Term'$</td>
<td>{ $, +, - }</td>
</tr>
<tr>
<td>$ Factor$</td>
<td>{ $, +, - , *, / }</td>
</tr>
</tbody>
</table>
Another FIRST/FOLLOW Example

Example Input: \( x + y ( z + a ( b ) ) \)

\[
\begin{align*}
\text{Expression} & \rightarrow \text{Function} \\
& | \quad ( \text{Expression} ) \\
& | \quad \text{Primary} + \text{Expression} \\
& | \quad \text{Primary} \\
\text{Primary} & \rightarrow \text{id} \\
& | \quad \text{integer} \\
\text{Function} & \rightarrow \text{id} ( \text{ParamList} ) \\
\text{ParamList} & \rightarrow \text{Expression} \ \text{ParamList} \\
& | \quad \varepsilon
\end{align*}
\]

FIRST (Expression) = \{ (, integer, id) \}
FIRST (Primary) = \{ integer, id \}
FIRST (Function) = \{ id \}
FIRST (ParamList) = \{ id, integer, (, \varepsilon \}

FOLLOW (Expression) = \{ $, (, ) , \text{id}, \text{integer} \}
FOLLOW (Primary) = \{ $, (, ) , +, \text{id}, \text{integer} \}
FOLLOW (Function) = \{ $, (, ) , \text{id}, \text{integer} \}
FOLLOW (ParamList) = \{ ) \}
LL(1) Grammars

Left-to-right scan of the input, Leftmost derivation, 1-token look-ahead

A grammar $G$ is LL(1) if for each set of its productions

$A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$:

- FIRST($\alpha_1$), FIRST($\alpha_2$), ..., FIRST($\alpha_n$), are all pair-wise disjoint
- If $\alpha_i \Rightarrow^* \varepsilon$, then \( \text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset \) for all $1 \leq i \leq n$, $i \neq j$

- In other words, LL(1) grammars
  - have no left recursion (direct or indirect)
  - productions are uniquely predictable given a context (look-ahead)
Recursive Descent Parsing

- Use a set of *mutually recursive* procedures
  - one procedure for each non-terminal symbol
  - start the parsing process by calling the procedure that corresponds to the start symbol
  - each production becomes one clause in procedure

- Use a look-ahead symbol to decide which production to use
  - based on the elements in the FIRST sets

- When no element in FIRST set matches, check the FOLLOW set
  - if look-ahead symbol is in FOLLOW set and there is an epsilon production, return from procedure (i.e., take epsilon production)
  - otherwise, terminate with a parsing error
Recursive Descent Parsing

1. \( S \rightarrow \text{if } E \text{ then } S \text{ else } S \)
2. \( | \begin{array}{l} \text{begin } S \ L \end{array} \)
3. \( | \begin{array}{l} \text{print } E \end{array} \)
4. \( L \rightarrow \text{end} \)
5. \( | \begin{array}{l} ; S \ L \end{array} \)
6. \( E \rightarrow \text{num } = \text{num} \)

void \text{main}() {
    \text{lookahead}=\text{getNextToken}();
    \text{S}();
    \text{match(EOF)};
}

void \text{E}() {
    \text{match(NUM)};
    \text{match(EQ)};
    \text{match(NUM)};
}
Recursive Descent:  if 2=2 then print 5=5 else print 1=1

main: call S();
S1: find the production for (S, IF) : S → if E then S else S
S1: match(IF);
S1: call E();
   E1: find the production for (E, NUM): E → num = num
   E1: match(NUM); match(EQ); match(NUM);
   E1: return from E1 to S1
S1: match(THEN);
S1: call S();
S2: find the production for (S, PRINT): S → print E
S2: match(PRINT);
S2: call E();
   E2: find the production for (E, NUM): E → num = num
   E2: match(NUM); match(EQ); match(NUM);
   E2: return from E2 to S2
S3: return from S2 to S1
S1: match(ELSE);
S1: call S();
S3: find the production for (S, PRINT): S → print E
S3: match(PRINT);
S3: call E();
   E3: find the production for (E, NUM): E → num = num
   E3: match(NUM); match(EQ); match(NUM);
   E3: return from E3 to S3
S4: return from S3 to S1
S1: return from S1 to main
main: match(EOF); return success;
Alternative: Table-Driven Parsers

A table-driven parser looks like

- **Source code**
  - Scanner
  - Table-driven Parser
  - Stack

- **Grammar**
  - Parser Generator
  - Parsing Table

Parsing tables can be built automatically!
Stack-Based, Table-Driven Parsing

The parsing table

- A two dimensional array
  \[ M[A, a] \rightarrow \text{gives a production} \]
  \[ A: \text{non-terminal symbol} \]
  \[ a: \text{terminal symbol} \]
- What does it mean?
  - If top of the stack is \( A \) and the look-ahead symbol is \( a \), then we apply the production \( M[A, a] \)

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>BEGIN</th>
<th>PRINT</th>
<th>END</th>
<th>SEMI</th>
<th>NUM</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S )</td>
<td>( S \rightarrow \text{if } E \text{ then } S \text{ else } S )</td>
<td>( S \rightarrow \text{begin } S \ L )</td>
<td>( S \rightarrow \text{print } E )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>( L )</td>
<td></td>
<td></td>
<td></td>
<td>( L \rightarrow \text{end} )</td>
<td>( L \rightarrow ; S \ L )</td>
<td></td>
</tr>
<tr>
<td>( E )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>( E \rightarrow \text{num } = \text{num} )</td>
</tr>
</tbody>
</table>
Table-Driven Parsing Algorithm

• Push the end-of-file symbol ($) and the start symbol onto the stack
• Consider the symbol $X$ on the top of the stack and look-ahead (terminal) symbol $a$
  – If $X = \$ \text{ and } a = \$ \text{ announce successful parse and halt}$
  – If $X = a$ (and $a \neq \$)$, pop $X$ off the stack and advance the input pointer to the next input symbol (read in new $a$)
  – If $X$ is a non-terminal, look at the production $M[X, a]$
    • If there is no such production ($M[X, a] = \text{error}$), then call an error routine
    • If $M[X, a]$ is a production $X \rightarrow Y_1 Y_2 \ldots Y_k$, then pop $X$ and push $Y_k, Y_{k-1}, \ldots, Y_1$ onto the stack with $Y_1$ on top
  – If none of the cases above apply, then call an error routine
Table-Driven Parsing Algorithm

Push($); // $ is the end-of-file symbol
Push(S); // S is the start symbol of the grammar
lookahead = get_next_token();
repeat
    X = top_of_stack();
    if (X is a terminal or X == $) then
        if (X == lookahead) then
            pop(X);
            lookahead = get_next_token();
        else error();
    else // X is a non-terminal
        if ( M[X, lookahead] == X → Y_1 Y_2 ... Y_k ) then
            pop(X);
            push(Y_k); push(Y_{k-1}); ... push(Y_1);
        else error();

until (X = $)
Table-Driven: if 2=2 then print 5=5 else print 1=1$

<table>
<thead>
<tr>
<th>Stack</th>
<th>lookahead</th>
<th>Parse-table lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S</td>
<td>IF</td>
<td>M[$S,IF]: S→if E then S else S</td>
</tr>
<tr>
<td>$S,ELSE,S,THEN,E,IF</td>
<td>IF</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$S,ELSE,S,THEN,E</td>
<td>NUM</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$S,ELSE,S,THEN,NUM,EQ,NUM</td>
<td>EQ</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$S,ELSE,S,THEN,NUM</td>
<td>NUM</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$S,ELSE,S</td>
<td>THEN</td>
<td>M[$S,PRINT]: S→print E</td>
</tr>
<tr>
<td>$S,ELSE,E,PRINT</td>
<td>PRINT</td>
<td>M[$S,PRINT]: S→print E</td>
</tr>
<tr>
<td>$S,ELSE,E</td>
<td>NUM</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$S,ELSE,NUM,EQ,NUM</td>
<td>EQ</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$S,ELSE,NUM</td>
<td>NUM</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$S,ELSE</td>
<td>ELSE</td>
<td>M[$S,PRINT]: S→print E</td>
</tr>
<tr>
<td>$S</td>
<td>PRINT</td>
<td>M[$S,PRINT]: S→print E</td>
</tr>
<tr>
<td>$E,PRINT</td>
<td>PRINT</td>
<td>M[$S,PRINT]: S→print E</td>
</tr>
<tr>
<td>$E</td>
<td>NUM</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$NUM,EQ,NUM</td>
<td>EQ</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$NUM,EQ</td>
<td>NUM</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$NUM</td>
<td>EQ</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$NUM</td>
<td>NUM</td>
<td>M[$E,NUM]: E→num = num</td>
</tr>
<tr>
<td>$NUM</td>
<td>report success!</td>
<td></td>
</tr>
</tbody>
</table>
LL(1) Parse Table Construction

• For all productions $A \rightarrow \alpha$, perform the following steps:
  – For each terminal symbol $a$ in $\text{FIRST}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$
  – If $\varepsilon$ is in $\text{FIRST}(\alpha)$, then add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal symbol $b$ in $\text{FOLLOW}(A)$.
  – Add $A \rightarrow \alpha$ to $M[A, \$]$ if $\$ is in $\text{FOLLOW}(A)$

• Set all the undefined entries in $M$ to ERROR
LL(1) Parse Table Construction

Grammar:

$$
\begin{align*}
1 & \quad S \rightarrow \text{Expr} \\
2 & \quad \text{Expr} \rightarrow \text{Term Expr}′ \\
3 & \quad \text{Expr}′ \rightarrow + \text{Term Expr}′ \\
4 & \quad \text{Expr}′ \mid - \text{Term Expr}′ \\
5 & \quad \text{Expr}′ \mid \epsilon \\
6 & \quad \text{Term} \rightarrow \text{Factor Term}′ \\
7 & \quad \text{Term}′ \rightarrow * \text{Factor Term}′ \\
8 & \quad \text{Term}′ \mid / \text{Factor Term}′ \\
9 & \quad \text{Term}′ \mid \epsilon \\
10 & \quad \text{Factor} \rightarrow \text{num} \\
11 & \quad \text{Factor} \mid \text{id}
\end{align*}
$$

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FIRST</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>{num, id}</td>
</tr>
<tr>
<td>Expr</td>
<td>{num, id}</td>
</tr>
<tr>
<td>Expr$'$</td>
<td>{\epsilon, +, -}</td>
</tr>
<tr>
<td>Term</td>
<td>{num, id}</td>
</tr>
<tr>
<td>Term$'$</td>
<td>{\epsilon, *, /}</td>
</tr>
<tr>
<td>Factor</td>
<td>{num, id}</td>
</tr>
<tr>
<td>num</td>
<td>{num}</td>
</tr>
<tr>
<td>id</td>
<td>{id}</td>
</tr>
<tr>
<td>+</td>
<td>{+}</td>
</tr>
<tr>
<td>-</td>
<td>{-}</td>
</tr>
<tr>
<td>*</td>
<td>{\ast}</td>
</tr>
<tr>
<td>/</td>
<td>{/}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>FOLLOW</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>{$$}</td>
</tr>
<tr>
<td>Expr</td>
<td>{$$}</td>
</tr>
<tr>
<td>Expr$'$</td>
<td>{$$}</td>
</tr>
<tr>
<td>Term</td>
<td>{$$, +, -}</td>
</tr>
<tr>
<td>Term$'$</td>
<td>{$$, +, -}</td>
</tr>
<tr>
<td>Factor</td>
<td>{$$, +, -, *, /}</td>
</tr>
</tbody>
</table>
LL(1) Parse Table Construction

LL(1) Parse table:

<table>
<thead>
<tr>
<th></th>
<th>id</th>
<th>num</th>
<th>+</th>
<th>-</th>
<th>*</th>
<th>/</th>
<th>$</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>$S \rightarrow E$</td>
<td>$S \rightarrow E$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>$E \rightarrow TE'$</td>
<td>$E \rightarrow TE'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E'$</td>
<td>$E' \rightarrow + TE'$</td>
<td>$E' \rightarrow - TE'$</td>
<td>$E' \rightarrow$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\epsilon$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>$T \rightarrow FT'$</td>
<td>$T \rightarrow FT'$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T'$</td>
<td></td>
<td>$T' \rightarrow \epsilon$</td>
<td>$T' \rightarrow \epsilon$</td>
<td>$T' \rightarrow * FT'$</td>
<td>$T' \rightarrow / FT'$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T' \rightarrow \epsilon$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>$F \rightarrow id$</td>
<td>$F \rightarrow num$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
LL(1) Grammar

Left-to-right scan of the input, Leftmost derivation, 1-token look-ahead

Two alternative definitions of LL(1) grammars:

1. A grammar $G$ is LL(1) if there are no multiple entries in its LL(1) parse table

2. A grammar $G$ is LL(1), if for each set of its productions $A \rightarrow \alpha_1 \mid \alpha_2 \mid ... \mid \alpha_n$

   \[
   \text{FIRST}(\alpha_1), \text{FIRST}(\alpha_2), ..., \text{FIRST}(\alpha_n) \text{ are all pair-wise disjoint}
   \]

   If $\alpha_j \Rightarrow^* \varepsilon$, then $\text{FIRST}(\alpha_j) \cap \text{FOLLOW}(A) = \emptyset$ for all $1 \leq i \leq n, i \neq j$
Side Note: Removing $\epsilon$ from a Grammar

- In theory, one can remove $\epsilon$ from a grammar, so why don’t we do that before we start
- Good idea, but there is a problem with this predictive parsing

| 1) Term $\rightarrow$ Factor Term’ |
| 2) Term’ $\rightarrow$ * Factor Term’ |
| 3) / Factor Term’ |
| 4) $\epsilon$ |

Remove $\epsilon$

| 1) Term $\rightarrow$ Factor Term’ |
| 2) | Factor |
| 3) Term’ $\rightarrow$ * Factor Term’ |
| 4) | * Factor |
| 5) / Factor Term’ |
| 6) / Factor |

Now the problem is that we need left factoring

| 1) Term $\rightarrow$ Factor Term’ |
| 2) | Factor |

Left Factor

| 1) Term $\rightarrow$ Factor X |
| 2) X $\rightarrow$ Term’ | $\epsilon$ |

Remove $\epsilon$

We need everything left factored for predictive parsing, but removing the $\epsilon$ is our factoring. This is why we need FIRST and FOLLOW sets.
The Verdict on Top-Down Parsing

• Top down parsers are great
  – They are (relatively) simple to construct by hand
  – They have many real-world applications
  – They provide the most intuitive way to reason about parsing
  – Predictive parsing is fast

• Top down has some problems
  – It gets really messy for complex grammars (like full Java)
  – It doesn’t handle left-recursion nicely, which is how we would like to specify left-associative operators
  – It is quite restrictive on the the types of grammars we can parse

• What we need is a fast and automated approach that can handle a more general set of grammars
  – This requires a different way of thinking about parsing ...
Where are we in the process?

- compiler
  - scan
  - parse
    - top down
    - bottom up
      - top down parsing with backtracking
      - predictive parsing

- predictable grammar
  - eliminating left recursion
  - left factoring
- algorithms
  - recursive descent
  - table-driven parsing