Syntax Analysis, III

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
The Classic Expression Grammar

<table>
<thead>
<tr>
<th></th>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Goal $\rightarrow$ Expr</td>
<td>This left-recursive expression grammar encodes standard algebraic precedence and left-associativity (which corresponds to left-to-right evaluation).</td>
</tr>
<tr>
<td>1</td>
<td>Expr $\rightarrow$ Expr + Term</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>$ Expr - Term</td>
</tr>
<tr>
<td></td>
<td>$</td>
<td>$ Term</td>
</tr>
<tr>
<td>2</td>
<td>Term $\rightarrow$ Term * Factor</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$</td>
<td>$ Term / Factor</td>
</tr>
<tr>
<td>4</td>
<td>$</td>
<td>$ Factor</td>
</tr>
<tr>
<td>5</td>
<td>Factor $\rightarrow$ ( Expr )</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>$</td>
<td>$ number</td>
</tr>
<tr>
<td>7</td>
<td>$</td>
<td>$ id</td>
</tr>
</tbody>
</table>

**Classic Expression Grammar, Left-recursive Version**

Both have the same acronym, **CEG**
The Algorithm

• A top-down parser starts with the root of the parse tree
• The root node is labeled with the goal symbol of the grammar

The key is selecting the right production in step 1
  – That choice should be guided by the input string
Top-down parsing with the CEG

With deterministic choices, the parse can expand indefinitely

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>Goal</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>0</td>
<td>Expr</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>1</td>
<td>Expr + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>1</td>
<td>Expr + Term + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>1</td>
<td>Expr + Term + Term + Term</td>
<td>$x - 2 \cdot y$</td>
</tr>
<tr>
<td>1</td>
<td>... and so on ....</td>
<td>$x - 2 \cdot y$</td>
</tr>
</tbody>
</table>

This expansion doesn’t terminate

- Wrong choice of expansion leads to non-termination
- **Non-termination** is a bad property for a parser to have
- Parser must make the right choice
Recursion in the CEG

The problems with unbounded expansion arise from left-recursion in the CEG

- LHS symbol cannot derive, in one or more steps, a sentential form that starts with the LHS
- Left recursion is incompatible with top-down leftmost derivations
- A leftmost derivation matches the scanner’s left-to-right scan

We need a technique to transform left recursion into right recursion (and, possible, the reverse)

†Similarly, right recursion is incompatible with rightmost derivations
Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

\[ Fee \rightarrow Fee \ \alpha \]
\[ \quad \mid \beta \]

where neither \( \alpha \) nor \( \beta \) start with \( Fee \)

We can rewrite this fragment as

\[ Fee \rightarrow \beta \ Fie \]
\[ Fie \rightarrow \alpha \ Fie \]
\[ \quad \mid \epsilon \]

where \( Fie \) is a new non-terminal

The new grammar defines the same language as the old grammar, using only right recursion.

New Idea: the \( \epsilon \) production

Problem: added a reference to the empty string
Eliminating Left Recursion

The expression grammar contains two cases of left recursion

\[
\begin{align*}
Expr & \rightarrow Expr + Term \\
& \quad | Expr - Term \\
& \quad | Term \\
Term & \rightarrow Term * Factor \\
& \quad | Term * Factor \\
& \quad | Factor
\end{align*}
\]

Applying the transformation yields

\[
\begin{align*}
Expr & \rightarrow Term Expr' \\
Expr' & \rightarrow + Term Expr' \\
& \quad | - Term Expr' \\
& \quad | \epsilon
\end{align*}
\]

\[
\begin{align*}
Term & \rightarrow Factor Term' \\
Term' & \rightarrow * Factor Term' \\
& \quad | / Factor Term' \\
& \quad | \epsilon
\end{align*}
\]

These fragments use only right recursion
Eliminating Left Recursion

Substituting them back into the grammar yields

Right-recursive expression grammar

- This grammar is correct, if somewhat counter-intuitive.
- A top-down parser will terminate using it.
- A top-down parser may need to backtrack with it.
- It is left associative, as was the original

COMP 412, Fall 2017
## Parsing with RR CEG

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>Goal</td>
</tr>
<tr>
<td>0</td>
<td>Expr</td>
</tr>
<tr>
<td>1</td>
<td>Term Expr’</td>
</tr>
<tr>
<td>5</td>
<td>Factor Term’ Expr’</td>
</tr>
<tr>
<td>11</td>
<td>&lt;id,x&gt; Term’ Expr’</td>
</tr>
<tr>
<td>8</td>
<td>&lt;id,x&gt; Expr’</td>
</tr>
<tr>
<td>3</td>
<td>&lt;id,x&gt; - Term Expr’</td>
</tr>
<tr>
<td>5</td>
<td>&lt;id,x&gt; - Factor Term’ Expr’</td>
</tr>
<tr>
<td>10</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; Term’ Expr’</td>
</tr>
<tr>
<td>6</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * Factor Term’ Expr’</td>
</tr>
<tr>
<td>11</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt; Term’ Expr’</td>
</tr>
<tr>
<td>8</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt; Expr’</td>
</tr>
<tr>
<td>4</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt;</td>
</tr>
</tbody>
</table>

\[
x - 2 \times y \text{ (again)}
\]

\[
x = \frac{\text{(again)}}{2}
\]
The two-production transformation eliminates immediate left recursion

What about more general, indirect left recursion?

The general algorithm to eliminate left recursion:

arrange the NTs into some order $A_1, A_2, \ldots, A_n$
for $i \leftarrow 2$ to $n$
  for $s \leftarrow 1$ to $i - 1$ {
    replace each production $A_i \rightarrow A_s \gamma$ with $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma$,
    where $A_s \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current productions for $A_s$
  }

eliminate any immediate left recursion on $A_i$ using the direct transformation

The algorithm assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$), and no epsilon productions

EaC2e shows the $i$ loop running from 1 to $n$, so 1st iteration of inner loop never executes.
Eliminating Left Recursion

How does this algorithm work?
1. Impose arbitrary order on the non-terminals
2. Outer loop cycles through NT in order
3. Inner loop ensures that a production expanding $A_i$ cannot directly derive a non-terminal $A_s$ at the start of its RHS, for $s < i$
4. Last step in outer loop converts any direct recursion on $A_i$ to right recursion using the transformation showed earlier
5. New non-terminals are added at the end of the order & have no left recursion; they will have right recursion on themselves

At the start of the $i^{th}$ outer loop iteration

For all $k < i$, no production that expands $A_k$ begins with a non-terminal $A_s$, for $s < k$
Eliminating Indirect Left Recursion

Example Grammar

• This grammar generates $a (ba)^*$
• Subscripts indicate the imposed order
• Indirect left recursion is $A \rightarrow B \rightarrow A$

arrange the NTs into some order $A_1, A_2, ..., A_n$
for $i \leftarrow 2$ to $n$
  for $s \leftarrow 1$ to $i - 1$
    replace each production $A_i \rightarrow A_s \gamma$ with $A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid ... \mid \delta_k \gamma$,
    where $A_s \rightarrow \delta_1 \mid \delta_2 \mid ... \mid \delta_k$ are all the current productions for $A_s$
  }

eliminate any immediate left recursion on $A_i$ using the direct transformation
Eliminating Indirect Recursion

General algorithm is a diagonalization, similar to LU decomposition

<table>
<thead>
<tr>
<th></th>
<th>NT₁</th>
<th>NT₂</th>
<th>NT₃</th>
</tr>
</thead>
<tbody>
<tr>
<td>NT₁</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>NT₂</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>NT₃</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Original Grammar

0 Start₁ → A₂
1 A₂ → B₃ a
2  
3 B₃ → A₂ b

- If [i, j] element is k, then NTᵢ derives NTⱼ in the kᵗʰ position of its RHS
- Diagonal elements represent self-recursion
- Upper triangle represents forward references in the order
- Lower triangle represents backward references in the order
  - Inner loop systematically zeroes the lower triangle
  - Last step of outer loop eliminates ones on the diagonal
Eliminating Indirect Left Recursion

Applying the Algorithm

<table>
<thead>
<tr>
<th></th>
<th>$s = 1$</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 1$</td>
<td>forward</td>
<td>forward</td>
<td>forward</td>
</tr>
<tr>
<td>2</td>
<td>no action</td>
<td>forward</td>
<td>forward</td>
</tr>
<tr>
<td>3</td>
<td>no action</td>
<td>see below</td>
<td>forward</td>
</tr>
</tbody>
</table>

For $B_3$ ($i = 3$) and $A_2$ ($s = 2$):

First, it rewrites rule 3 with

$$B_3 \rightarrow B_3 \ a \ b$$
$$\ | \ a \ b$$

Next, it uses the rule for immediate left recursion to rewrite this pair of rules as:

$$B_3 \rightarrow a \ b \ C$$
$$C \rightarrow a \ b \ C$$
$$\ | \ \epsilon$$

Algorithm iterates through the possible backward references.

- Handles indirect left recursion with forward substitution of RHS for LHS
- Handles direct left recursion with the transformation
Before and After

Original Grammar

- \( A_2 \rightarrow B_3 \ a \)
- \( B_3 \rightarrow A_2 \ b \)

Transformed Grammar

- \( A_2 \rightarrow B_3 \ a \)
- \( B_3 \rightarrow a \ b \ C \)
- \( C \rightarrow a \ b \ C \)
- \( C \rightarrow \epsilon \)

• The transformed grammar generates \( a \ (ba)^* \)
Eliminating Indirect Recursion

General algorithm is a diagonalization, similar to LU decomposition

<table>
<thead>
<tr>
<th></th>
<th>$NT_1$</th>
<th>$NT_2$</th>
<th>$NT_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$NT_1$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$NT_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$NT_3$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Original Grammar**

<table>
<thead>
<tr>
<th></th>
<th>$NT_1$</th>
<th>$NT_2$</th>
<th>$NT_3$</th>
<th>$NT_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$NT_1$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$NT_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$NT_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>$NT_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

**Transformed Grammar**

0 $Start_1 \rightarrow A_2$
1 $A_2 \rightarrow B_3\ a$
2 $\vdash \ a$
3 $B_3 \rightarrow A_2\ b$
4 $\vdash \ a\ b\ C$
5 $C \rightarrow \epsilon$
Eliminating Indirect Left Recursion

Before and After

Original Grammar

- The transformed grammar generates \( a (ba)^* \)
- The transformed grammar is still not predictively parsable.

COMP 412, Fall 2017
Predictive Parsing

If a top-down parser picks the “wrong” production, it may need to backtrack. The alternative is to look ahead in the input & use context to pick the correct production.

How much lookahead is needed for a context-free grammar?
• In general, an arbitrarily large amount
• Use the Cocke-Younger, Kasami algorithm or Earley’s algorithm

Fortunately,
• Large subclasses of CFGs can be parsed with limited lookahead
• Most programming language constructs fall in those subclasses

Among the interesting subclasses are $LL(1)$ and $LR(1)$ grammars

*We will focus, for now, on $LL(1)$ grammars & predictive parsing*
Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between $\alpha$ & $\beta$

FIRST sets

For some RHS $\alpha \in G$, 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 $\gamma$

We will defer the problem of how to compute FIRST sets for the moment and assume that they are given.
Predictive Parsing

Basic idea

Given $A \rightarrow \alpha \mid \beta$, the parser should be able to choose between $\alpha$ & $\beta$

FIRST sets

For some RHS $\alpha \in G$, 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 $\gamma$

The Predictive, or LL(1), Property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ both appear in the grammar, we would like

$$\text{FIRST}(\alpha) \cap \text{FIRST}(\beta) = \emptyset$$

This would allow the parser to make a correct choice with a lookahead of exactly one symbol!

This rule is intuitive. Unfortunately, it is not correct, because it does not handle $\varepsilon$ rules. See the next slide
Predictive Parsing

What about $\varepsilon$-productions?

$\Rightarrow$ They complicate the definition of the predictive, or LL(1) property

If $A \rightarrow \alpha$ and $A \rightarrow \beta$ and $\varepsilon \in \text{FIRST}(\alpha)$, then we need to ensure that $\text{FIRST}(\beta)$ is disjoint from $\text{FOLLOW}(A)$, too, where

**FOLLOW(A) is the set of terminal symbols that can appear immediately after A in some sentential form**

Define $\text{FIRST}^+(A \rightarrow \alpha)$ as

- $\text{FIRST}(\alpha) \cup \text{FOLLOW}(A)$, if $\varepsilon \in \text{FIRST}(\alpha)$
- $\text{FIRST}(\alpha)$, otherwise

Then, a grammar is LL(1) iff $A \rightarrow \alpha$ and $A \rightarrow \beta$ implies that

$$\text{FIRST}^+(A \rightarrow \alpha) \cap \text{FIRST}^+(A \rightarrow \beta) = \emptyset$$

Note that $\text{FIRST}^+$ is defined over productions, not NTs
### The LL(1) Property

Our transformed grammar for $a \ (ba)^* \ fails \ the \ test$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$Start_1 \rightarrow A_2$</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$A_2 \rightarrow B_3 \ a$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$\mid a$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$B_3 \rightarrow a \ b \ C$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$C \rightarrow a \ b \ C$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$\mid \epsilon$</td>
<td></td>
</tr>
</tbody>
</table>

#### Transformed Grammar

Derivations for $A$ begin with $a$:

- $A \rightarrow B \ a \ | \ a$
- $B \rightarrow a$

$$\text{FIRST}^+(A \rightarrow B \ a) = \{ a \}$$  
$$\text{FIRST}^+(A \rightarrow a) = \{ a \}$$

These sets are identical, rather than disjoint. So, left recursion elimination did not produce a grammar with the LL(1) condition — a grammar that would be predictively parsable.
The **LL(1) Property**

Can we write an **LL(1) grammar for** $a \ (ba)^*$?

**Transformed Grammar**

<table>
<thead>
<tr>
<th>$0$</th>
<th>$Start_1 \rightarrow A_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td>$A_2 \rightarrow B_3 \ a$</td>
</tr>
<tr>
<td>$2$</td>
<td>$\mid \ a$</td>
</tr>
<tr>
<td>$3$</td>
<td>$B_3 \rightarrow a \ b \ C$</td>
</tr>
<tr>
<td>$4$</td>
<td>$C \rightarrow a \ b \ C$</td>
</tr>
<tr>
<td>$5$</td>
<td>$\mid \ \varepsilon$</td>
</tr>
</tbody>
</table>

**LL(1) Grammar for** $a \ (ba)^*$

<table>
<thead>
<tr>
<th>$0$</th>
<th>$Start \rightarrow a \ B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1$</td>
<td>$B \rightarrow b \ a \ B$</td>
</tr>
<tr>
<td>$2$</td>
<td>$\mid \ \varepsilon$</td>
</tr>
</tbody>
</table>
What If My Grammar Is Still Not LL(1) ?

Can we transform a non-LL(1) grammar into an LL(1) grammar?

• In general, the answer is no
• In some cases, however, the answer is yes

Assume a grammar $G$ with productions $A \rightarrow \alpha \beta_1$ and $A \rightarrow \alpha \beta_2$
• If $\alpha$ derives anything other than $\varepsilon$, then

$$\text{FIRST}^+ (A \rightarrow \alpha \beta_1) \cap \text{FIRST}^+ (A \rightarrow \alpha \beta_2) \neq \emptyset$$

• And the grammar is not LL(1)

If we pull the common prefix, $\alpha$, into a separate production, we may make the grammar LL(1).

$$A \rightarrow \alpha A', \ A' \rightarrow \beta_1, \text{ and } A' \rightarrow \beta_2$$

Now, if $\text{FIRST}^+ (A' \rightarrow \beta_1) \cap \text{FIRST}^+ (A' \rightarrow \beta_2) = \emptyset$, $G$ may be LL(1)
What If My Grammar Is Not LL(1) ?

**Left Factoring**

For each NT A

- find the longest prefix α common to 2 or more alternatives for A
- if α ≠ ε then
  - replace all of the productions
    
    \[ A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \alpha \beta_3 | \ldots | \alpha \beta_n | \gamma \]

  with

    \[ A \rightarrow \alpha A' | \gamma \]
    
    \[ A' \rightarrow \beta_1 | \beta_2 | \beta_3 | \ldots | \beta_n \]

Repeat until no NT has alternative rhs’ with a common prefix

This transformation makes some grammars into LL(1) grammars

There are languages for which no LL(1) grammar exists
Consider a short grammar for subscripted identifiers

To choose between expanding by 0, 1, or 2, a top-down parser must look beyond the name to see the next word.

\[ \text{FIRST}^+(0) = \text{FIRST}^+(1) = \text{FIRST}^+(2) = \{ \text{name} \} \]

Left factoring productions 0, 1, & 2 fixes this problem.

Left factoring cannot fix all non-LL(1) grammars.
Left Factoring Example

After left factoring:

0. \( \text{Factor} \rightarrow \text{name } \text{Args} \)
1. \( \text{Args} \rightarrow [ \text{ArgList} ] \)
2. \( \text{ArgList} \rightarrow ( \text{ArgList} ) \)
3. \( \text{ArgList} \rightarrow \varepsilon \)
4. \( \text{MoreArgs} \rightarrow \text{Expr } \text{MoreArgs} \)
5. \( \text{MoreArgs} \rightarrow \varepsilon \)

Clearly, \( \text{FIRST}^+(1) \) & \( \text{FIRST}^+(2) \) are disjoint.
If neither \( \downarrow \) or \( [ \) can follow \( \text{Factor} \), then \( \text{FIRST}^+(3) \) will be, too.

This tiny grammar now has the \( \text{LL}(1) \) property

Left factoring can transform some non-\( \text{LL}(1) \) grammars into \( \text{LL}(1) \) grammars.
There are languages for which no \( \text{LL}(1) \) grammar exists.
See exercise 3.12 in \text{EaC2e}.
Predictive Parsing

Given a grammar that has the LL(1) property

• We can write a simple routine to recognize an instance of each LHS
• Code is patterned, simple, & fast

Consider $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with

$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset$ if $i \neq j$

/* find an $A$ */
if (current_word $\in$ FIRST$^+(A \rightarrow \beta_1)$)
    find a $\beta_1$ and return true
else if (current_word $\in$ FIRST$^+(A \rightarrow \beta_2)$)
    find a $\beta_2$ and return true
else if (current_word $\in$ FIRST$^+(A \rightarrow \beta_3)$)
    find a $\beta_3$ and return true
else
    report an error and return false

Grammars that have the LL(1) property are called **predictive grammars** because the parser can “predict” the correct expansion at each point in the parse.

Parsers that capitalize on the LL(1) property are called **predictive parsers**.

One kind of predictive parser is the **recursive descent** parser.

Of course, there is more detail to “find a $\beta_i$” — typically a recursive call to another small routine (see pp. 108--111 in EaC2e)
Recursive Descent Parsing

Recall the expression grammar, after transformation

This grammar leads to a parser that has six mutually recursive routines:
1. Goal
2. Expr
3. EPrime
4. Term
5. TPrime
6. Factor

Each routine recognizes an rhs for that NT.

The term descent refers to the direction in which the parse tree is built.

This grammar has the LL(1) property
A couple of routines from the expression parser

**Goal( )**

```
token ← next_token( );
if (Expr( ) = true & token = EOF)
    then next compilation step;
else
    report syntax error;
    return false;
```

**Expr( )**

```
if (Term( ) = false)
    then return false;
else return Eprime( );
```

**Factor( )**

```
if (token = number) then
    token ← next_token( );
    return true;
else if (token = identifier) then
    token ← next_token( );
    return true;
else if (token = lparen)
    token ← next_token( );
    if (Expr( ) = true & token = rparen) then
        token ← next_token( );
        return true;
// fall out of if statement
    report syntax error;
    return false;
```

looking for number, identifier, or (, found token instead, or failed to find Expr or ) after ( 

EPrime, Term, & TPrime follow the same basic lines (Figure 3.10, EaC2e)
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 \( \epsilon \)-productions

Your lab 2 parsers are not much more complex than the example.

COMP 412, Fall 2017
Top-Down Recursive Descent Parser

At this point, you almost have enough information to build a top-down recursive-descent parser

- Need a right-recursive grammar that meets the LL(1) condition
  - Can use left-factoring to eliminate common prefixes
  - Can transform direct left recursion into right recursion
  - Need a general algorithm to handle indirect left recursion

- Need algorithms to construct FIRST and FOLLOW

Next Class

- Algorithms for FIRST and FOLLOW
- Algorithm for generalized elimination of left recursion
- An LL(1) skeleton parser, table, and table-construction algorithm