Midterm Exam: Thursday  
October 18, 7PM  
Herzstein Amphitheater

Syntax Analysis, III

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.

Chapter 3 in EaC2e
This left-recursive expression grammar encodes standard algebraic precedence and left-associativity (which corresponds to left-to-right evaluation).

We refer to this grammar as either the classic expression grammar or the canonical expression grammar.

Both have the same acronym, CEG.
Top-down Parsing

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

Construct the root node of the parse tree
Repeat until lower fringe of the parse tree matches the input string

1. At a node labeled with NT A, select a production with A on its LHS and, for each symbol on its RHS, construct the appropriate child
2. When a terminal symbol is added to the fringe and it doesn’t match the fringe, backtrack
3. Find the next node to be expanded (label ∈ NT)

The key is selecting the right production in step 1
  – That choice should be guided by the input string
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>( \uparrow x - 2 \times y )</td>
</tr>
<tr>
<td>0</td>
<td>Expr</td>
<td>( \uparrow x - 2 \times y )</td>
</tr>
<tr>
<td>1</td>
<td>Expr + Term</td>
<td>( \uparrow x - 2 \times y )</td>
</tr>
<tr>
<td>1</td>
<td>Expr + Term + Term</td>
<td>( \uparrow x - 2 \times y )</td>
</tr>
<tr>
<td>1</td>
<td>Expr + Term + Term + Term</td>
<td>( \uparrow x - 2 \times y )</td>
</tr>
<tr>
<td>1</td>
<td>... and so on ....</td>
<td>( \uparrow x - 2 \times y )</td>
</tr>
</tbody>
</table>

Consumes no input!

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

---

Classic Expression Grammar, Left-recursive Version

<table>
<thead>
<tr>
<th></th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>Goal</strong> → <strong>Expr</strong></td>
</tr>
<tr>
<td>1</td>
<td><strong>Expr</strong> → <strong>Expr</strong> + <strong>Term</strong></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><strong>Term</strong> → <strong>Term</strong> * <strong>Factor</strong></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td><strong>Factor</strong> → ( <strong>Expr</strong> )</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>
Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

\[ Fee \rightarrow Fee \alpha \]
\[ \mid \beta \]
where neither \( \alpha \) nor \( \beta \) start with \( Fee \)

We can rewrite this fragment as

\[ Fee \rightarrow \beta Fie \]
\[ Fie \rightarrow \alpha Fie \]
\[ \mid \varepsilon \]
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 \( \varepsilon \) production

Language is \( \beta \) followed by 0 or more \( \alpha \)'s

Added a reference to the empty string
Eliminating Left Recursion

The expression grammar contains two cases of left recursion

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

Applying the transformation yields

\[
\begin{align*}
\text{Expr} & \quad \rightarrow \quad \text{Term} \text{Expr'} & \quad \text{Term} & \quad \rightarrow \quad \text{Factor} \text{Term'} \\
\text{Expr'} & \quad \rightarrow \quad + \text{Term} \text{Expr'} & \quad \text{Term'} & \quad \rightarrow \quad \ast \text{Factor} \text{Term'} \\
& \quad \mid \quad - \text{Term} \text{Expr'} & & \quad \mid \quad / \text{Factor} \text{Term'} \\
& \quad \mid \quad \varepsilon & & \quad \mid \quad \varepsilon
\end{align*}
\]

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
Eliminating Left Recursion

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 A₀, A₁, ..., Aₙ
for i ← 1 to n
    for s ← 0 to i − 1 {
        replace each production Aᵢ → Aₛγ with Aᵢ → δ₁γ | δ₂γ | ... | δₖγ,
        where Aₛ → δ₁ | δ₂ | ... | δₖ are all the current productions for Aₛ
    }
    eliminate any immediate left recursion on Aᵢ using the direct transformation
```

The algorithm assumes that the initial grammar has no cycles
(Aᵢ ⇒⁺ Aᵢ), and no epsilon productions

EaC2e shows the inner 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 \( A_0, A_1, ..., A_n \)
for \( i \leftarrow 1 \) to \( n \)
  for \( s \leftarrow 0 \) 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>LHS</th>
<th>RHS</th>
</tr>
</thead>
<tbody>
<tr>
<td>$NT_0$</td>
<td>$NT_1$</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

$Start_0 \rightarrow A_1$

$A_1 \rightarrow B_2 \ a$

$\rightarrow a$

$B_2 \rightarrow A_1 \ b$

- If $[i, j]$ element is $k$, then $NT_i$ derives $NT_j$ in the $k^{th}$ 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>$s = 0$</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$i = 1$</td>
<td>no action</td>
<td>no iteration</td>
<td>no iteration</td>
</tr>
<tr>
<td>2</td>
<td>no action</td>
<td>see below</td>
<td>no iteration</td>
</tr>
</tbody>
</table>

For $B_2$ ($i = 2$) and $A_1$ ($s = 1$):

First, it rewrites rule 3 with

$$B_2 \rightarrow B_2 \ a \ b$$
$$| \ a \ b$$

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

$$B_2 \rightarrow a \ b \ C_3$$
$$C_3 \rightarrow a \ b \ C_3$$
$$| \ \varepsilon$$

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>Start</strong>ₐ → <strong>A</strong>₁</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td><strong>A</strong>₁ → <strong>B</strong>₂ <strong>a</strong></td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td><strong>B</strong>₂ → <strong>A</strong>₁ <strong>b</strong></td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

Transformed Grammar

The transformed grammar *(still)* generates **a (ba)*
Eliminating Indirect Recursion

General algorithm is a diagonalization, similar to LU decomposition

Original Grammar

<table>
<thead>
<tr>
<th></th>
<th>$NT_0$</th>
<th>$NT_1$</th>
<th>$NT_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$NT_0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$NT_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$NT_2$</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Transformed Grammar

<table>
<thead>
<tr>
<th></th>
<th>$NT_0$</th>
<th>$NT_1$</th>
<th>$NT_2$</th>
<th>$NT_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$NT_0$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$NT_1$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$NT_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>$NT_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Original matrix is now upper triangular

0 $Start_0 \rightarrow A_1$
1 $A_1 \rightarrow B_2 \ a$
2 $\ | \ a$
3 $B_2 \rightarrow A_1 \ b$
4 $Start_0 \rightarrow A_1$
5 $A_1 \rightarrow B_2 \ a$
6 $\ | \ a$
7 $B_2 \rightarrow a \ b \ C_3$
8 $C_3 \rightarrow a \ b \ C_3$
9 $\ | \ \varepsilon$
### Before and After

<table>
<thead>
<tr>
<th>Original Grammar</th>
<th>Transformed Grammar</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 ( \text{Start}_0 \to A_2 )</td>
<td>0 ( \text{Start}_0 \to A_1 )</td>
</tr>
<tr>
<td>1 ( A_1 \to B_2 \ a )</td>
<td>1 ( A_1 \to B_2 \ a )</td>
</tr>
<tr>
<td>2 ( \vDash a )</td>
<td>2 ( \vDash a )</td>
</tr>
<tr>
<td>3 ( B_2 \to A_1 \ b )</td>
<td>3 ( B_2 \to a \ b \ C_3 )</td>
</tr>
<tr>
<td>4 ( C_3 \to a \ b \ C_3 )</td>
<td>5 ( \vDash \varepsilon )</td>
</tr>
</tbody>
</table>

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

What does “predictively parsable” even mean?
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 $A \rightarrow \alpha | \beta$, 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 ε-productions?
⇒ 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
Our transformed grammar for $a (ba)^*$ fails the test

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$Start_0$ $\rightarrow A_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>$A_1$ $\rightarrow B_2 a$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$B_2$ $\rightarrow a b C_3$</td>
<td>$a$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$C_3$ $\rightarrow a b C_3$</td>
<td></td>
<td>$a$</td>
<td>$\varepsilon$</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Transformed Grammar

Derivations for $A$ begin with $a$:

$A \rightarrow B a \mid a$

$B \rightarrow a$

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

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

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**

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$Start_1$</td>
<td>$A_2$</td>
</tr>
<tr>
<td>1</td>
<td>$A_2$</td>
<td>$B_3 \ a$</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>$a$</td>
</tr>
<tr>
<td>3</td>
<td>$B_3$</td>
<td>$a \ b \ C_3$</td>
</tr>
<tr>
<td>4</td>
<td>$C_3$</td>
<td>$a \ b \ C_3$</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>$\varepsilon$</td>
</tr>
</tbody>
</table>

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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$Start$</td>
<td>$a \ B$</td>
</tr>
<tr>
<td>1</td>
<td>$B$</td>
<td>$b \ a \ B$</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>$\varepsilon$</td>
</tr>
</tbody>
</table>

COMP 412, Fall 2018
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 $\alpha$ common to 2 or more alternatives for $A$

if $\alpha \neq \varepsilon$ 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
Left Factoring Example

Consider a short grammar for subscripted identifiers

\[
\begin{align*}
0 & \quad \text{Factor} \rightarrow \text{name} \\
1 & \quad | \quad \text{name} \ [ \text{ArgList} ] \\
2 & \quad | \quad \text{name} \ ( \text{ArgList} ) \\
3 & \quad \text{ArgList} \rightarrow \text{Expr} \ \text{MoreArgs} \\
4 & \quad \text{MoreArgs} \rightarrow \text{Expr} \ \text{MoreArgs} \\
5 & \quad | \quad \varepsilon
\end{align*}
\]

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  Factor  →  name Args
1  Args   →  [ ArgList ]
2     ∣  ( ArgList )
3     ∣  ε
4  ArgList  |  Expr MoreArgs
5  MoreArgs →  , Expr MoreArgs
6                  ∣  ε

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

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

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

/* 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

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>Goal</strong> → <strong>Expr</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td><strong>Expr</strong> → <strong>Term</strong> <strong>Expr’</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><strong>Expr’</strong> → + <strong>Term</strong> <strong>Expr’</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>— <strong>Term</strong> <strong>Expr’</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>— &amp;epsilon;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><strong>Term</strong> → <strong>Factor</strong> <strong>Term’</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td><strong>Term’</strong> → * <strong>Factor</strong> <strong>Term’</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>— \devide <strong>Factor</strong> <strong>Term’</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>— &amp;epsilon;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td><strong>Factor</strong> → { <strong>Expr</strong> }</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>— <strong>number</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>— <strong>id</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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
Recursive Descent Parsing  (Procedural)

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;
    else
        // 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 ε-productions

```
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();

Factor()
    /* Factor → ( Expr ) */
    /* Factor → num */
    /* Factor → name */
    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();
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```
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 Parsing Lecture

• Algorithms for $FIRST$ and $FOLLOW$
• An $LL(1)$ skeleton parser, table, and table-construction algorithm

Next Lecture

• Local Register Allocation and Lab 2
• Read materials on local allocation from projects page of web site