Parsing — Part II
(Top-down parsing, left-recursion removal)

CS434
Lecture 7
Spring 2005
Department of Computer Science
University of Alabama
Joel Jones
Parsing Techniques

Top-down parsers (LL(1), recursive descent)
- Start at the root of the parse tree and grow toward leaves
- Pick a production & try to match the input
- Bad “pick” ⇒ may need to backtrack
- Some grammars are backtrack-free (predictive parsing)

Bottom-up parsers (LR(1), operator precedence)
- Start at the leaves and grow toward root
- As input is consumed, encode possibilities in an internal state
- Start in a state valid for legal first tokens
- Bottom-up parsers handle a large class of grammars
A top-down parser starts with the root of the parse tree
The root node is labeled with the goal symbol of the grammar

Top-down parsing algorithm:

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

1. At a node labeled $A$, 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 $(\text{label} \in \text{NT})$

- The key is picking the right production in step 1
  - That choice should be guided by the input string
Remember the expression grammar?

Version with precedence derived last lecture

And the input $x - 2 * y$. 
Example

Let's try $x - 2 * y$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Expr</td>
<td>$x - 2 * y$</td>
</tr>
<tr>
<td>2</td>
<td>Expr + Term</td>
<td>$x - 2 * y$</td>
</tr>
<tr>
<td>4</td>
<td>Term + Term</td>
<td>$x - 2 * y$</td>
</tr>
<tr>
<td>7</td>
<td>Factor + Term</td>
<td>$x - 2 * y$</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id,x&gt; + Term</td>
<td>$x - 2 * y$</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id,x&gt; + Term</td>
<td>$x - 2 * y$</td>
</tr>
</tbody>
</table>

Leftmost derivation, choose productions in an order that exposes problems
Example

Let's try $x - 2 \cdot y$:

This worked well, except that "-" doesn't match "+

The parser must backtrack to here.
Example

Continuing with $x - 2 * y$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>Goal</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>1</td>
<td>Expr</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>3</td>
<td>Expr – Term</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>4</td>
<td>Term – Term</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>7</td>
<td>Factor – Term</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id,x&gt; – Term</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id,x&gt; – Term</td>
<td>$x \uparrow -2 * y$</td>
</tr>
<tr>
<td>—</td>
<td>&lt;id,x&gt; – Term</td>
<td>$x - \uparrow 2 * y$</td>
</tr>
</tbody>
</table>

Diagram:

- Goal
  - Expr
    - Term
      - Fact.
        - <id,x>
Example

Continuing with $x - 2 \times y$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Expr</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<tr>
<td>3</td>
<td>Expr - Term</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<tr>
<td>4</td>
<td>Term - Term</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<tr>
<td>7</td>
<td>Factor - Term</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<tr>
<td>9</td>
<td>$&lt;id,x&gt;$ - Term</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<tr>
<td>9</td>
<td>$&lt;id,x&gt;$ - Term</td>
<td>$x \uparrow 2 \times y$</td>
</tr>
<tr>
<td>0</td>
<td>$&lt;id,x&gt;$ - Term</td>
<td>$x \uparrow x - 2 \times y$</td>
</tr>
</tbody>
</table>

Goal

Expr

Term

Fact.

<id,x>

This time, “–” and “–” matched

We can advance past “–” to look at “2”

⇒ Now, we need to expand Term - the last NT on the fringe
Example

Trying to match the "2" in $x - 2 \cdot y$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>&lt;id,x&gt; - Term</td>
<td>$x - \uparrow 2 \cdot y$</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id,x&gt; - Factor</td>
<td>$x - \uparrow 2 \cdot y$</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id,x&gt; - &lt;num,2&gt;</td>
<td>$x - \uparrow 2 \cdot y$</td>
</tr>
<tr>
<td>-</td>
<td>&lt;id,x&gt; - &lt;num,2&gt;</td>
<td>$x - \downarrow 2 \cdot y$</td>
</tr>
</tbody>
</table>

Diagram:

```
Goal
  ▼
Expr
  ▼
Expr
  ▼
Term
  ▼
Term
  ▼
Fact.
  ▼
<num,2>
  ▼
{id,x>}
```
Example

Trying to match the “2” in \( x - 2 * y \):

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>−</td>
<td>(&lt;id,x&gt; - \text{Term})</td>
<td>(x - \uparrow 2 * y)</td>
</tr>
<tr>
<td>7</td>
<td>(&lt;id,x&gt; - \text{Factor})</td>
<td>(x - \uparrow 2 * y)</td>
</tr>
<tr>
<td>9</td>
<td>(&lt;id,x&gt; - \langle\text{num,2}\rangle)</td>
<td>(x - \uparrow 2 * y)</td>
</tr>
<tr>
<td>−</td>
<td>(&lt;id,x&gt; - \langle\text{num,2}\rangle)</td>
<td>(x - 2 * y)</td>
</tr>
</tbody>
</table>

Where are we?
• “2” matches “2”
• We have more input, but no NTs left to expand
• The expansion terminated too soon
⇒ Need to backtrack
Example

Trying again with “2” in $x - 2 \times y$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>&lt;id,x&gt; - Term</td>
<td>$x - \uparrow 2 \times y$</td>
</tr>
<tr>
<td>5</td>
<td>&lt;id,x&gt; - Term * Factor</td>
<td>$x - \uparrow 2 \times y$</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id,x&gt; - Factor * Factor</td>
<td>$x - \uparrow 2 \times y$</td>
</tr>
<tr>
<td>8</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * Factor</td>
<td>$x - \uparrow 2 \times y$</td>
</tr>
<tr>
<td>—</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * Factor</td>
<td>$x - \downarrow 2 \times y$</td>
</tr>
<tr>
<td>—</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt;</td>
<td>$x - \downarrow 2 \times y$</td>
</tr>
<tr>
<td>—</td>
<td>&lt;id,x&gt; - &lt;num,2&gt; * &lt;id,y&gt;</td>
<td>$x - 2 \times y$</td>
</tr>
</tbody>
</table>

This time, we matched & consumed all the input

⇒ Success!
Another possible parse

Other choices for expansion are possible

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td><em>Goal</em></td>
<td>↑x - 2 * y</td>
</tr>
<tr>
<td>1</td>
<td><em>Expr</em></td>
<td>↑x - 2 * y</td>
</tr>
<tr>
<td>2</td>
<td><em>Expr + Term</em></td>
<td>↑x - 2 * y</td>
</tr>
<tr>
<td>2</td>
<td><em>Expr + Term + Term</em></td>
<td>↑x - 2 * y</td>
</tr>
<tr>
<td>2</td>
<td><em>Expr + Term + Term + Term</em></td>
<td>↑x - 2 * y</td>
</tr>
<tr>
<td>2</td>
<td><em>Expr + Term + Term + ... + Term</em></td>
<td>↑x - 2 * y</td>
</tr>
</tbody>
</table>

This doesn’t terminate (obviously)

- 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
Left Recursion

Top-down parsers cannot handle left-recursive grammars

Formally,

A grammar is left recursive if $\exists A \in NT$ such that

$\exists$ a derivation $A \Rightarrow^+ A\alpha$, for some string $\alpha \in (NT \cup T)^*$

Our expression grammar is left recursive

- This can lead to non-termination in a top-down parser
- For a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion

Non-termination is a bad property in any part of a compiler
Eliminating Left Recursion

To remove left recursion, we can transform the grammar

Consider a grammar fragment of the form

\[
\text{Fee} \rightarrow \text{Fee} \; \alpha \\
\mid \beta
\]

where neither \( \alpha \) nor \( \beta \) start with \( \text{Fee} \)

We can rewrite this as

\[
\text{Fee} \rightarrow \beta \; \text{Fie}
\]

\[
\text{Fie} \rightarrow \alpha \; \text{Fie}
\]

\[
\mid \varepsilon
\]

where \( \text{Fie} \) is a new non-terminal

This accepts the same language, but uses only right recursion
Eliminating Left Recursion

The expression grammar contains two cases of left recursion

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

Applying the transformation yields

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

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

These fragments use only right recursion
They retain the original left associativity
Eliminating Left Recursion

Substituting them back into the grammar yields

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

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

The transformation eliminates immediate left recursion

What about more general, indirect left recursion?

The general algorithm:

- Arrange the NTs into some order $A_1, A_2, \ldots, A_n$

  for $i \leftarrow 1$ to $n$
  for $s \leftarrow 1$ to $i - 1$
  replace each production $A_i \rightarrow A_s \gamma$ with $A_i \rightarrow \delta_1 \gamma | \delta_2 \gamma | \ldots | \delta_k \gamma$,
  where $A_s \rightarrow \delta_1 | \delta_2 | \ldots | \delta_k$ are all the current productions for $A_s$
  eliminate any immediate left recursion on $A_i$

This assumes that the initial grammar has no cycles ($A_i \Rightarrow^+ A_i$),
and no epsilon productions
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 $A_i$ has no non-terminal $A_s$ in 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

At the start of the $i^{th}$ outer loop iteration
   For all $k < i$, no production that expands $A_k$ contains a non-terminal $A_s$ in its rhs, for $s < k$
Example

- **Order of symbols:** $G, E, T$

\[
\begin{align*}
G & \rightarrow E \\
E & \rightarrow E + T \\
E & \rightarrow T \\
T & \rightarrow E \sim T \\
T & \rightarrow \text{id}
\end{align*}
\]
Example

- Order of symbols: $G$, $E$, $T$

1. $A_i = G$

$G \rightarrow E$

$E \rightarrow E + T$

$E \rightarrow T$

$T \rightarrow E \sim T$

$T \rightarrow \text{id}$
Example

- **Order of symbols**: $G, E, T$

1. $A_i = G$
   
   - $G \rightarrow E$
   - $E \rightarrow E + T$
   - $E \rightarrow T$
   - $T \rightarrow E \sim T$
   - $T \rightarrow \text{id}$

2. $A_i = E$
   
   - $G \rightarrow E$
   - $E \rightarrow T E'$
   - $E' \rightarrow + T E'$
   - $E' \rightarrow \varepsilon$
   - $T \rightarrow E \sim T$
   - $T \rightarrow \text{id}$
## Example

- **Order of symbols:** $G, E, T$

1. $A_i = G$
   - $G \rightarrow E$
   - $E \rightarrow E + T$
   - $E \rightarrow T$
   - $T \rightarrow E \sim T$
   - $T \rightarrow \text{id}$

2. $A_i = E$
   - $G \rightarrow E$
   - $E \rightarrow T E'$
   - $E' \rightarrow + T E'$
   - $E' \rightarrow \varepsilon$
   - $T \rightarrow E \sim T$
   - $T \rightarrow \text{id}$

3. $A_i = T, A_S = E$
   - $G \rightarrow E$
   - $E \rightarrow T E'$
   - $E' \rightarrow + T E'$
   - $E' \rightarrow \varepsilon$
   - $T \rightarrow T E' \sim T$
   - $T \rightarrow \text{id}$

**Go to Algorithm**
Example

- **Order of symbols:** $G, E, T$

<table>
<thead>
<tr>
<th>$A_i = G$</th>
<th>$A_i = E$</th>
<th>$A_i = T, A_s = E$</th>
<th>$A_i = T$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G \rightarrow E$</td>
<td>$G \rightarrow E$</td>
<td>$G \rightarrow E$</td>
<td>$G \rightarrow E$</td>
</tr>
<tr>
<td>$E \rightarrow E + T$</td>
<td>$E \rightarrow T E'$</td>
<td>$E \rightarrow T E'$</td>
<td>$E \rightarrow T E'$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E' \rightarrow + T E'$</td>
<td>$E' \rightarrow + T E'$</td>
<td>$E' \rightarrow + T E'$</td>
</tr>
<tr>
<td>$T \rightarrow E \sim T$</td>
<td>$E' \rightarrow \varepsilon$</td>
<td>$E' \rightarrow \varepsilon$</td>
<td>$E' \rightarrow \varepsilon$</td>
</tr>
<tr>
<td>$T \rightarrow id$</td>
<td>$T \rightarrow E \sim T$</td>
<td>$T \rightarrow T E' \sim T$</td>
<td>$T \rightarrow id T'$</td>
</tr>
<tr>
<td>$T \rightarrow id$</td>
<td>$T \rightarrow id$</td>
<td>$T \rightarrow id$</td>
<td>$T' \rightarrow E' \sim T T'$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>$T' \rightarrow \varepsilon$</td>
</tr>
</tbody>
</table>
Roadmap (Where are we?)

We set out to study parsing

• Specifying syntax
  → Context-free grammars ✓
  → Ambiguity ✓

• Top-down parsers
  → Algorithm & its problem with left recursion ✓
  → Left-recursion removal ✓

• Predictive top-down parsing
  → The LL(1) condition today
  → Simple recursive descent parsers today
  → Table-driven LL(1) parsers today
Picking the “Right” Production

If it picks the wrong production, a top-down parser may backtrack.

Alternative is to look ahead in input & use context to pick correctly.

How much lookahead is needed?

- 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.
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 until we look at the LR(1) table construction algorithm.
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 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 is almost correct
See the next slide
Predictive Parsing

What about \(\varepsilon\)-productions?

⇒ They complicate the definition of LL(1)

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 \(\text{FOLLOW}(\alpha)\), too.

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

- \(\text{FIRST}(\alpha) \cup \text{FOLLOW}(\alpha)\), 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

\[
\text{FIRST}^+(\alpha) \cap \text{FIRST}^+(\beta) = \emptyset
\]

\(\text{FOLLOW}(\alpha)\) is the set of all words in the grammar that can legally appear immediately after an \(\alpha\).
Predictive Parsing

Given a grammar that has the LL(1) property

- Can write a simple routine to recognize each lhs
- Code is both simple & fast

Consider \( A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \), with

\[ \text{FIRST}^+ (\beta_1) \cap \text{FIRST}^+ (\beta_2) \cap \text{FIRST}^+ (\beta_3) = \emptyset \]

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

Of course, there is more detail to “find a \( \beta_i \)” (§ 3.3.4 in EAC)

Grammars with 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.
Recursive Descent Parsing

Recall the expression grammar, after transformation

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
</tr>
<tr>
<td>2</td>
<td>Expr</td>
</tr>
<tr>
<td>3</td>
<td>Expr'</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Term</td>
</tr>
<tr>
<td>7</td>
<td>Term'</td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Factor</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

This produces a parser with six mutually recursive routines:
- Goal
- Expr
- EPrime
- Term
- TPrime
- Factor

Each recognizes one NT or T

The term descent refers to the direction in which the parse tree is built.
Recursive Descent Parsing (Procedural)

A couple of routines from the expression parser

**Goal()**

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

**Expr()**

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

**Factor()**

\[ \text{if } (\text{token} = \text{Number}) \text{ then} \]
\[ \text{token} \leftarrow \text{next\_token()}; \]
\[ \text{return true}; \]
\[ \text{else if } (\text{token} = \text{Identifier}) \text{ then} \]
\[ \text{token} \leftarrow \text{next\_token()}; \]
\[ \text{return true}; \]
\[ \text{else} \]
\[ \text{report syntax error}; \]
\[ \text{return false}; \]

**EPrime, Term, & TPrime** follow the same basic lines (Figure 3.7, EAC)

Looking for EOF, found token

Looking for Number or Identifier, found token instead
Recursive Descent Parsing

To build a parse tree:
  • Augment parsing routines to build nodes
  • Pass nodes between routines using a stack
  • Node for each symbol on rhs
  • Action is to pop rhs nodes, make them children of lhs node, and push this subtree

To build an abstract syntax tree
  • Build fewer nodes
  • Put them together in a different order

This is a preview of Chapter 4

Expr()
result ← true;
if (Term() = false)
  then return false;
else if (EPrime() = false)
  then result ← false;
else
  build an Expr node
  pop EPrime node
  pop Term node
  make EPrime & Term children of Expr
  push Expr node
return result;

Success ⇒ build a piece of the parse tree
Left Factoring

What if my grammar does not have the LL(1) property?
⇒ Sometimes, we can transform the grammar

The Algorithm

∀ A ∈ NT,
find the longest prefix \( \alpha \) that occurs in two or more right-hand sides of \( A \)
if \( \alpha \neq \varepsilon \) then replace all of the \( A \) productions,
\[ A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \gamma, \]
with
\[ A \rightarrow \alpha Z \mid \gamma \]
\[ Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \]
where \( Z \) is a new element of \( NT \)
Repeat until no common prefixes remain
Left Factoring

A graphical explanation for the same idea

\[
A \rightarrow \alpha \beta_1 \\
| \alpha \beta_2 \\
| \alpha \beta_3
\]

becomes ...

\[
A \rightarrow \alpha Z \\
Z \rightarrow \beta_1 \\
| \beta_2 \\
| \beta_n
\]
Consider the following fragment of the expression grammar

\[
\begin{align*}
\text{Factor} & \rightarrow \text{Identifier} \\
& \mid \text{Identifier} \ [ \text{ExprList} ] \\
& \mid \text{Identifier} \ ( \text{ExprList} )
\end{align*}
\]

After left factoring, it becomes

\[
\begin{align*}
\text{Factor} & \rightarrow \text{Identifier Arguments} \\
\text{Arguments} & \rightarrow \ [ \text{ExprList} ] \\
& \mid ( \text{ExprList} ) \\
& \mid \varepsilon
\end{align*}
\]

This form has the same syntax, with the LL(1) property
Left Factoring

Graphically

No basis for choice

becomes ...

Word determines correct choice