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

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

Bottom-up parsers \((LR(1), \text{operator precedence})\)
- 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
Top-down Parsing

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 input, backtrack
3. Find the next node to be expanded (label ∈ 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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Goal</strong> → <strong>Expr</strong></td>
</tr>
<tr>
<td>2</td>
<td><strong>Expr</strong> → <strong>Expr</strong> + <strong>Term</strong></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td><strong>Term</strong> → <strong>Term</strong> * <strong>Factor</strong></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td><strong>Factor</strong> → <strong>number</strong></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

And the input $x - 2 * y$
Example

Let’s try $x - 2 \times y$:

<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>2</td>
<td>Expr + Term</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<tr>
<td>3</td>
<td>Term + Term</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<tr>
<td>4</td>
<td>Factor + Term</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<tr>
<td>5</td>
<td>&lt;id,x&gt; + Term</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<tr>
<td>6</td>
<td>&lt;id,x&gt; + Term</td>
<td>$x \uparrow - 2 \times y$</td>
</tr>
</tbody>
</table>

Leftmost derivation, choose productions in an order that exposes problems
Let's try $x - 2 \times y$:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>$-$</td>
<td>Goal</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<tr>
<td>1</td>
<td>Expr</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<tr>
<td>2</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>
</tbody>
</table>

This worked well, except that “−” doesn’t match “+”
The parser must backtrack to here.
Example

Continuing with $x - 2 \cdot y$:

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

Graph:

```
     Goal
      ↙   ↘
    Expr    Term
      ↙   ↘
    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>—</td>
<td>Goal</td>
<td>$\uparrow x - 2 \times y$</td>
</tr>
<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>—</td>
<td>&lt;id,x&gt; - Term</td>
<td>$x - \uparrow 2 \times y$</td>
</tr>
</tbody>
</table>

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 $x - 2 * y$:

<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>$x - 2 \uparrow y$</td>
</tr>
<tr>
<td>7</td>
<td>$&lt;id,x&gt; - \text{Factor}$</td>
<td>$x - 2 \uparrow y$</td>
</tr>
<tr>
<td>9</td>
<td>$&lt;id,x&gt; - &lt;\text{num},2&gt;$</td>
<td>$x - 2 \uparrow 2 * y$</td>
</tr>
<tr>
<td></td>
<td>$&lt;id,x&gt; - &lt;\text{num},2&gt;$</td>
<td>$x - 2 \uparrow y$</td>
</tr>
</tbody>
</table>
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>7</td>
<td>&lt;id,x&gt; - Factor</td>
<td>( x - \uparrow 2 * y )</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id,x&gt; - &lt;num,2&gt;</td>
<td>( x - \uparrow 2 * y )</td>
</tr>
<tr>
<td></td>
<td>&lt;id,x&gt; - &lt;num,2&gt;</td>
<td>( x - 2 \uparrow * y )</td>
</tr>
</tbody>
</table>

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

Trying again with “2” in \( x - 2 * y \):

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>(&lt;\text{id},x&gt; - \text{Term* Factor})</td>
<td>(x - \uparrow 2 * y)</td>
</tr>
<tr>
<td>7</td>
<td>(&lt;\text{id},x&gt; - \text{Factor* Factor})</td>
<td>(x - \uparrow 2 * y)</td>
</tr>
<tr>
<td>8</td>
<td>(&lt;\text{id},x&gt; - \text{&lt;num,2&gt;* Factor})</td>
<td>(x - \uparrow 2 * y)</td>
</tr>
<tr>
<td>9</td>
<td>(&lt;\text{id},x&gt; - \text{&lt;num,2&gt;* &lt;id,y&gt;})</td>
<td>(x - 2 * \uparrow 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>Goal</td>
<td>( \uparrow x - 2 \cdot y )</td>
</tr>
<tr>
<td>1</td>
<td>Expr</td>
<td>( \uparrow x - 2 \cdot y )</td>
</tr>
<tr>
<td>2</td>
<td>Expr + Term</td>
<td>( \uparrow x - 2 \cdot y )</td>
</tr>
<tr>
<td>2</td>
<td>Expr + Term + Term</td>
<td>( \uparrow x - 2 \cdot y )</td>
</tr>
<tr>
<td>2</td>
<td>Expr + Term + Term</td>
<td>( \uparrow x - 2 \cdot y )</td>
</tr>
<tr>
<td>2</td>
<td>Expr + Term + Term + ... + Term</td>
<td>( \uparrow x - 2 \cdot y )</td>
</tr>
</tbody>
</table>

This 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

(Obviously) consuming no input!
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

\[ \text{Fee} \rightarrow \text{Fee} \ \alpha \]
\[ \quad | \quad \beta \]

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

We can rewrite this as

\[ \text{Fee} \rightarrow \beta \ \text{Fie} \]
\[ \text{Fie} \rightarrow \alpha \ \text{Fie} \]
\[ \quad | \quad \epsilon \]

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

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

Applying the transformation yields

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

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

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

Substituting them back into the grammar yields

```
1   Goal  →  Expr
2   Expr  →  Term Expr’
3   Expr’ →  + Term Expr’
4            |  - Term Expr’
5            |  ε
6   Term  →  Factor Term’
7   Term’ →  * Factor
8            |  / Factor
9            |  ε
10  Factor →  number
11            |  id
12            |  (Expr)
```

- 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:

\[
\text{arrange the NTs into some order } A_1, A_2, \ldots, A_n
\]

for \( i \leftarrow 1 \) to \( n \)

for \( s \leftarrow 1 \) to \( i - 1 \)

replace each production \( A_i \rightarrow A_s \gamma \) with \( A_i \rightarrow \delta_1 \gamma \mid \delta_2 \gamma \mid \ldots \mid \delta_k \gamma \),

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

This assumes that the initial grammar has no cycles \( (A_i \Rightarrow^* A_i) \),
and no epsilon productions
Lookahead in predictive parsing

- The Lookahead token (next token in the input) is used to determine which rule should be used next.
- Consider the following example:

\[
\begin{align*}
term & \rightarrow num \ term' \\
term' & \rightarrow '+' \ num \ term' \ \\
    & \mid '-' \ num \ term' \\
    & \mid \varepsilon \\
num & \rightarrow '0' \mid '1' \mid \ldots \mid '9'
\end{align*}
\]
Lookahead in predictive parsing

• The Lookahead token (next token in the input) is used to determine which rule should be used next
• Consider the following example:

\[
\begin{align*}
\text{term} & \rightarrow \text{num} \text{ term'} \\
\text{term'} & \rightarrow '+' \text{ num} \text{ term'} \\
& \quad | '-' \text{ num} \text{ term'} \\
& \quad | \varepsilon \\
\text{num} & \rightarrow '0'| '1'| \ldots | '9' \\
\end{align*}
\]

\[
7 + 3 - 2
\]

\[
\text{term} \\
\downarrow \\
\text{num} \\
\downarrow \\
7 \\
\downarrow \\
+ \\
\downarrow \\
\text{num} \quad \text{term'}
\]
Lookahead in predictive parsing

- The Lookahead token (next token in the input) is used to determine which rule should be used next.
- Consider the following example:

\[\text{term} \rightarrow \text{num term'}\]
\[\text{term'} \rightarrow '+\text{ num term'}\]
\[\quad \mid '-' \text{ num term'}\]
\[\quad \mid \epsilon\]
\[\text{num} \rightarrow '0'|'1'|'2'|'3'|\ldots|'9'\]

\[
\begin{align*}
7+3\rightarrow 10-2
\end{align*}
\]

Diagram:
- `term` node with children `num` and `term'`.
- `num` node with children `7`, `+`, `num`, and `term'`.
- `3` node as a child of `term'`.
Lookahead in predictive parsing

• The Lookahead token (next token in the input) is used to determine which rule should be used next

• Consider the following example:

```
term → num term'

term' → ‘+’ num term‘
| ‘-’ num term'
| ε

num → ‘0’| ‘1’| … | ‘9’
```

```
7+3-2
```

```
    term
    /   
term' num term'
/      /   
7     term'
     /   
  term' num term'
  /     /   
3     num term'
```

Lookahead in predictive parsing

- The Lookahead token (next token in the input) is used to determine which rule should be used next.
- Consider the following example:

\[
\begin{align*}
term & \rightarrow num \ term' \\
term' & \rightarrow '+' num \ term' \\
& \quad | '-' num \ term' \\
& \quad | \varepsilon \\
um & \rightarrow '0' | '1' | '2' | ... | '9'
\end{align*}
\]
Lookahead in predictive parsing

- The Lookahead token (next token in the input) is used to determine which rule should be used next.
- Consider the following example:

\[
\begin{align*}
term & \rightarrow num \ term' \\
term' & \rightarrow ' + ' num \ term' \\
& \quad | ' - ' num \ term' \\
& \quad | \epsilon \\
num & \rightarrow '0' | '1' | '2' | \ldots | '9'
\end{align*}
\]
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 $\gamma$

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?

$\Rightarrow$ They complicate the definition of LL(1)

If $A \to \alpha$ and $A \to \beta$ 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 $A \to \alpha$ and $A \to \beta$ 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 $\alpha$. 

Determining First Sets

- If \( \alpha \) is a terminal symbol, \( \alpha \in T \)
  \[ \text{FIRST}(\alpha) = \{\alpha}\]  
- If \( A \) is a non-terminal symbol, \( A \in NT \)
  For each production \( A \rightarrow \beta \), where \( \beta \) is \( \beta_1 \beta_2 \ldots \beta_k \)
  \[ \text{FIRST}(A) = \emptyset \]  
  \[ \text{FIRST}(A) = \text{FIRST}(A) \cup (\text{FIRST}\{\beta_1}\) - \( \varepsilon \)\]  
  \[ i = 1\]  
  While \( \varepsilon \in \text{FIRST}(\beta_i) \) and \( i \leq k-1 \)
  \[ \text{FIRST}(A) = \text{FIRST}(A) \cup (\text{FIRST}\{\beta_{i+1}\) - \( \varepsilon \)\]  
  If \( i == k \) and \( \varepsilon \in \text{FIRST}(\beta_k) \)
  \[ \text{FIRST} (A) = \text{FIRST}(A) \cup \varepsilon \]
First Sets (Example)

What is FIRST(C)?

\[ \text{FIRST}(C) = \text{FIRST}(c) \cup \text{FIRST}(\varepsilon) = \{c, \varepsilon\} \]

What is FIRST(B)?

\[ \text{FIRST}(B) = \text{FIRST}(b) \cup \text{FIRST}(d) = \{b, d\} \]

What is FIRST(A)?

\[ \text{FIRST}(A) = \text{FIRST}(aBCd) \cup \text{FIRST}(BQ) \cup \text{FIRST}(\varepsilon) \]

\[ \text{FIRST}(A) = \{a, b, d, \varepsilon\} \]

What is FIRST(Q)?

\[ \text{FIRST}(Q) = \{q\} \]

What is FIRST(S)?

\[ \text{FIRST}(S) = \text{FIRST}(AC) \]

\[ \text{FIRST}(S) = \text{FIRST}(A) - \varepsilon \cup \text{FIRST}(C) \]

\[ \text{FIRST}(S) = \{a, b, c, d, \varepsilon\} \]
Determining Follow Sets

For each $A \in NT$

$\text{FOLLOW}(A) = \emptyset$

$\text{FOLLOW}(S) = \{\$\}$

While (Follow sets are still changing)

For each production $A \rightarrow \alpha B \beta$, $\varepsilon \notin \text{FIRST}(\beta)$

$\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup (\text{FIRST}(\beta))$

For each production $A \rightarrow \alpha B \beta$, $\varepsilon \in \text{FIRST}(\beta)$

$\text{FOLLOW}(B) = \text{FOLLOW}(B) \cup (\text{FIRST}(\beta) - \varepsilon) \cup \text{Follow}(A)$
## Follow Sets (Example)

<table>
<thead>
<tr>
<th>Grammar Symbol</th>
<th>Production</th>
<th>What is FOLLOW(\textit{Symbol})?</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>( \rightarrow ) AC</td>
<td>\textit{FOLLOW(S)}?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\textit{FOLLOW(S)} = {$}</td>
</tr>
<tr>
<td>C</td>
<td>( \rightarrow ) c</td>
<td>\textit{FOLLOW(C)}?</td>
</tr>
<tr>
<td></td>
<td>( \mid ) ( \epsilon )</td>
<td>\textit{FOLLOW(C)} = {d} \cup \textit{FOLLOW(S)} = {d, $}</td>
</tr>
<tr>
<td>A</td>
<td>( \rightarrow ) aBCd</td>
<td>\textit{FOLLOW(B)}?</td>
</tr>
<tr>
<td></td>
<td>( \mid ) BQ</td>
<td>\textit{FOLLOW(B)} = \text{FIRST}(Cd) \cup \text{FIRST}(Q) \cup \textit{FOLLOW(B)}</td>
</tr>
<tr>
<td></td>
<td>( \mid ) ( \epsilon )</td>
<td>\textit{FOLLOW(B)} = {c, d, q}</td>
</tr>
<tr>
<td>B</td>
<td>( \rightarrow ) bB</td>
<td>\textit{FOLLOW(A)}?</td>
</tr>
<tr>
<td></td>
<td>( \mid ) d</td>
<td>\textit{FOLLOW(A)} = \text{FIRST}(C) \cup \textit{FOLLOW(S)} = {c, $}</td>
</tr>
<tr>
<td>Q</td>
<td>( \rightarrow ) q</td>
<td>\textit{FOLLOW(Q)}?</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\textit{FOLLOW(Q)} = \textit{FOLLOW(A)} = {c, $}</td>
</tr>
</tbody>
</table>
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

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

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.

Of course, there is more detail to “find a $\beta_i$” ($\S$ 3.3.4 in EAC)
Recursive DescentParsing

Recall the expression grammar, after transformation

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()**

```
    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
        report syntax error;
        return false;
```

**Eprime()**

```
    if (token = Number) then
        token ← next_token( );
        return true;
    else if (token = Identifier) then
        token ← next_token( );
        return true;
    else
        report syntax error;
        return false;
```
Left Factoring

What if my grammar does not have the LL(1) property?

⇒ Sometimes, we can transform the grammar

The Algorithm

\[ \forall A \in NT, \]
\[ \text{find the longest prefix } \alpha \text{ that occurs in two or more right-hand sides of } A \]
\[ \text{if } \alpha \neq \epsilon \text{ then replace all of the } A \text{ productions,} \]
\[ A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \gamma, \]
\[ \text{with} \]
\[ A \rightarrow \alpha \ Z \mid \gamma \]
\[ Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n \]

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 ...

\[ A \rightarrow \alpha Z \]
\[ Z \rightarrow \beta_1 \]
\[ | \beta_2 \]
\[ | \beta_n \]
Left Factoring

Consider the following fragment of the expression grammar

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

After left factoring, it becomes

\[
\begin{align*}
\text{Factor} & \rightarrow \text{Identifier} \, \text{Arguments} \\
\text{Arguments} & \rightarrow \, [\, \text{ExprList} \, ] \\
& \quad | \, \, (\, \text{ExprList} \, ) \\
& \quad | \, \epsilon
\end{align*}
\]

This form has the same syntax, with the \textit{LL(1)} property
Left Factoring

Graphically

Factor  →  Identifier

Factor  →  Identifier  [ ExprList ]

Factor  →  Identifier  ( ExprList )

becomes ...

Factor  →  Identifier  [ ExprList ]

Factor  →  Identifier  ( ExprList )

No basis for choice

Word determines correct choice
Question
By eliminating left recursion and left factoring, can we transform an arbitrary CFG to a form where it meets the LL(1) condition? (and can be parsed predictively with a single token lookahead?)

Answer
Given a CFG that doesn’t meet the LL(1) condition, it is undecidable whether or not an equivalent LL(1) grammar exists.

Example
\{a^n 0 b^n | n \geq 1\} \cup \{a^n 1 b^{2n} | n \geq 1\} has no LL(1) grammar
Language that Cannot Be LL(1)

Example

\{a^n 0 b^n \mid n \geq 1\} \cup \{a^n 1 b^{2n} \mid n \geq 1\} has no LL(1) grammar

\[
G \to aAb \\
\mid aBbb
\]

\[
A \to aAb \\
\mid 0
\]

\[
B \to aBbb \\
\mid 1
\]

Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group.
Recursive Descent (Summary)

1. Build FIRST (and FOLLOW) sets
2. Massage grammar to have $LL(1)$ condition
   a. Remove left recursion
   b. Left factor it
3. Define a procedure for each non-terminal
   a. Implement a case for each right-hand side
   b. Call procedures as needed for non-terminals
4. Add extra code, as needed
   a. Perform context-sensitive checking
   b. Build an IR to record the code

Can we automate this process?