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

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 leaves 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 (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?

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal → Expr</td>
<td>x – 2 * y</td>
</tr>
<tr>
<td>2</td>
<td>Expr → Expr + Term</td>
<td>x – 2 * y</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>x – 2 * y</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>x – 2 * y</td>
</tr>
<tr>
<td>5</td>
<td>Term → Term * Factor</td>
<td>x – 2 * y</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>x – 2 * y</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>x – 2 * y</td>
</tr>
<tr>
<td>8</td>
<td>Factor → number</td>
<td>x – 2 * y</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>x – 2 * y</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>x – 2 * y</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>x – 2 * y</td>
</tr>
</tbody>
</table>

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 + 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>
**Example**

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

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

<table>
<thead>
<tr>
<th>Rule</th>
<th>Sentential Form</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>—</td>
<td>(&lt;id,x&gt; - \text{Term})</td>
<td>(x - \uparrow 2 \times y)</td>
</tr>
<tr>
<td>7</td>
<td>(&lt;id,x&gt; - \text{Factor})</td>
<td>(x - \uparrow 2 \times y)</td>
</tr>
<tr>
<td>9</td>
<td>(&lt;id,x&gt; - \text{&lt;num,2&gt;})</td>
<td>(x - \uparrow 2 \times y)</td>
</tr>
<tr>
<td>—</td>
<td>(&lt;id,x&gt; - \text{&lt;num,2&gt;})</td>
<td>(x - \uparrow 2 \times 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
  \[
\Rightarrow \text{Need to backtrack}
\]

### 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; - \text{Term})</td>
<td>(x - \uparrow 2 \times y)</td>
</tr>
<tr>
<td>5</td>
<td>(&lt;id,x&gt; - \text{Term} \times \text{Factor})</td>
<td>(x - \uparrow 2 \times y)</td>
</tr>
<tr>
<td>7</td>
<td>(&lt;id,x&gt; - \text{Factor} \times \text{Factor})</td>
<td>(x - \uparrow 2 \times y)</td>
</tr>
<tr>
<td>8</td>
<td>(&lt;id,x&gt; - \text{&lt;num,2&gt;} \times \text{Factor})</td>
<td>(x - \uparrow 2 \times y)</td>
</tr>
<tr>
<td>—</td>
<td>(&lt;id,x&gt; - \text{&lt;num,2&gt;} \times \text{Factor})</td>
<td>(x - \uparrow 2 \times y)</td>
</tr>
<tr>
<td>—</td>
<td>(&lt;id,x&gt; - \text{&lt;num,2&gt;} \times \text{&lt;id,y&gt;})</td>
<td>(x - \uparrow 2 \times y)</td>
</tr>
<tr>
<td>9</td>
<td>(&lt;id,x&gt; - \text{&lt;num,2&gt;} \times \text{&lt;id,y&gt;})</td>
<td>(x - \uparrow 2 \times y)</td>
</tr>
<tr>
<td>—</td>
<td>(&lt;id,x&gt; - \text{&lt;num,2&gt;} \times \text{&lt;id,y&gt;})</td>
<td>(x - \uparrow 2 \times y)</td>
</tr>
</tbody>
</table>

This time, we matched & consumed all the input
  \[
\Rightarrow \text{Success!}
\]
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>$x - 2 \times y$</td>
</tr>
<tr>
<td>1</td>
<td>Expr</td>
<td>$x - 2 \times y$</td>
</tr>
<tr>
<td>2</td>
<td>Expr + Term</td>
<td>$x - 2 \times y$</td>
</tr>
<tr>
<td>2</td>
<td>Expr + Term + Term</td>
<td>$x - 2 \times y$</td>
</tr>
<tr>
<td>2</td>
<td>Expr + Term + Term + ... + Term</td>
<td>$x - 2 \times y$</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 \( \exists A \in N \) such that
\( \exists \) a derivation \( A \Rightarrow^{*} A\alpha \), for some string \( \alpha \in (N \cup T)^{+} \)

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

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

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

Note that: \( Fee \Rightarrow \beta \alpha^* \)

We can rewrite this to generate \( \beta \) first, as

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

where \( Fie \) 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*}
Expr & \rightarrow Expr + Term \\
& \quad | Expr - Term \\
& \quad | Term \\
\end{align*}
\[
\begin{align*}
Term & \rightarrow Term * Factor \\
& \quad | Term / Factor \\
& \quad | Factor \\
\end{align*}
\]

Applying the transformation yields

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

These fragments use only right recursion

They retains the original left associativity
Eliminating Left Recursion

Substituting back into the grammar yields

1. \( \text{Goal} \rightarrow \text{Expr} \)
2. \( \text{Expr} \rightarrow \text{Term} \text{Expr}' \)
3. \( \text{Expr}' \rightarrow + \text{Term} \text{Expr}' \)
4. \( \text{Expr}' \rightarrow - \text{Term} \text{Expr}' \)
5. \( \text{Expr}' \rightarrow \epsilon \)
6. \( \text{Term} \rightarrow \text{Factor} \text{Term}' \)
7. \( \text{Term}' \rightarrow * \text{Factor} \text{Term}' \)
8. \( \text{Term}' \rightarrow \text{Term} \text{Term}' \)
9. \( \text{Term}' \rightarrow \epsilon \)
10. \( \text{Factor} \rightarrow \text{number} \)
11. \( \text{Factor} \rightarrow \text{id} \)
12. \( \text{Factor} \rightarrow ( \text{Expr} ) \)

- 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:

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 \( A_s \rightarrow \delta_1 \delta_2 \gamma \ldots \gamma \delta_k \) 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 \( (A_i \Rightarrow^* A_j) \),
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: $G, E, T$

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></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>$E \rightarrow TE'$</td>
<td>$E \rightarrow TE'$</td>
<td>$E \rightarrow TE'$</td>
</tr>
<tr>
<td>$E \rightarrow T$</td>
<td>$E' \rightarrow + TE'$</td>
<td>$E' \rightarrow + TE'$</td>
<td>$E' \rightarrow + TE'$</td>
</tr>
<tr>
<td>$T \rightarrow E \sim T$</td>
<td>$T \rightarrow T \sim T$</td>
<td>$T \rightarrow T \sim T$</td>
<td>$T \rightarrow T \sim T$</td>
</tr>
<tr>
<td>$T \rightarrow \text{id}$</td>
<td>$T \rightarrow \text{id}$</td>
<td>$T \rightarrow \text{id}$</td>
<td>$T \rightarrow \text{id}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{id}$</td>
<td>$T \rightarrow \text{id}$</td>
<td>$T \rightarrow \text{id}$</td>
<td>$T \rightarrow \text{id}$</td>
</tr>
<tr>
<td>$T \rightarrow \text{id}$</td>
<td>$T \rightarrow \text{id}$</td>
<td>$T \rightarrow \text{id}$</td>
<td>$T \rightarrow \text{id}$</td>
</tr>
<tr>
<td>$T \rightarrow T' \sim T$</td>
<td>$T \rightarrow T' \sim T$</td>
<td>$T \rightarrow T' \sim T$</td>
<td>$T \rightarrow T' \sim T$</td>
</tr>
<tr>
<td>$T' \rightarrow E \sim TT'$</td>
<td>$T' \rightarrow E \sim TT'$</td>
<td>$T' \rightarrow E \sim TT'$</td>
<td>$T' \rightarrow E \sim TT'$</td>
</tr>
<tr>
<td>$T' \rightarrow \epsilon$</td>
<td>$T' \rightarrow \epsilon$</td>
<td>$T' \rightarrow \epsilon$</td>
<td>$T' \rightarrow \epsilon$</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** – When can we make the right decision without backtracking?
  - The LL(1) condition
  - Simple recursive descent parsers

---

**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
- E.g., the Cocke-Younger-Kasami algorithm or Earley’s algorithm  
  - $O(n^3)$ on size of input.

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* \( A \rightarrow \alpha \mid \beta \), 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 \( \gamma \)

**When is First(\alpha) useful?**

- When there is no choice in what production to choose.

**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!

*(Pursuing this idea leads to LL(1) parser generators...)*

---

**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 pairwise emptiness, i.e., \( i \neq j \)

\[
\text{FIRST}(\beta_i) \cap \text{FIRST}(\beta_j) = \emptyset
\]

```
/* find an A */
if (current_word \in \text{FIRST}(\beta_i))
   find a \beta_i and return true
else if (current_word \in \text{FIRST}(\beta_j))
   find a \beta_j and return true
else if (current_word \in \text{FIRST}(\beta_3))
   find a \beta_3 and return true
else
   report an error and return false
```

Grammar 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 \( \beta \)"
### LL(1) process

#### Given $X \rightarrow \alpha$, compute $\text{FIRST}(\alpha)$

<table>
<thead>
<tr>
<th>NT</th>
<th>FIRST(\alpha)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>$ETF$</td>
</tr>
<tr>
<td>$E$</td>
<td>$ETF$</td>
</tr>
<tr>
<td>$T$</td>
<td>$TF$</td>
</tr>
<tr>
<td>$F$</td>
<td>$\text{Num id}$</td>
</tr>
</tbody>
</table>

#### Table

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>-</th>
<th>(</th>
<th>)</th>
<th>*</th>
<th>/</th>
<th>id</th>
<th>num</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>$G\rightarrow E$</td>
<td>$G\rightarrow E$</td>
<td>$G\rightarrow E$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>$E\rightarrow ET$</td>
<td>$E\rightarrow ET$</td>
<td>$E\rightarrow ET$</td>
<td>$E\rightarrow ET$</td>
<td>$E\rightarrow ET$</td>
<td>$E\rightarrow ET$</td>
<td>$E\rightarrow ET$</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$T\rightarrow TF$</td>
<td>$T\rightarrow TF$</td>
<td>$T\rightarrow TF$</td>
<td>$T\rightarrow TF$</td>
<td>$T\rightarrow TF$</td>
<td>$T\rightarrow TF$</td>
<td>$T\rightarrow TF$</td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>$F\rightarrow (E)$</td>
<td>$F\rightarrow \text{id}$</td>
<td>$F\rightarrow \text{Num}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* We know since grammar is left recursive that it can't be LL(1).
* Multiple entries in table prove it -- no explicit rule to use.

### But recall grammar after left recursion eliminated

#### Given $X \rightarrow \alpha$, compute $\text{FIRST}(\alpha)$

<table>
<thead>
<tr>
<th>NT</th>
<th>FIRST(\alpha)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>$ETF$</td>
</tr>
<tr>
<td>$E$</td>
<td>$TF$</td>
</tr>
<tr>
<td>$F$</td>
<td>$\text{Num id}$</td>
</tr>
</tbody>
</table>

#### Table

<table>
<thead>
<tr>
<th></th>
<th>+</th>
<th>-</th>
<th>(</th>
<th>)</th>
<th>*</th>
<th>/</th>
<th>id</th>
<th>num</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G$</td>
<td>$G\rightarrow ET$</td>
<td>$G\rightarrow ET$</td>
<td>$G\rightarrow ET$</td>
<td>$G\rightarrow ET$</td>
<td>$G\rightarrow ET$</td>
<td>$G\rightarrow ET$</td>
<td>$G\rightarrow ET$</td>
<td></td>
</tr>
<tr>
<td>$E$</td>
<td>$E\rightarrow TF$</td>
<td>$E\rightarrow TF$</td>
<td>$E\rightarrow TF$</td>
<td>$E\rightarrow TF$</td>
<td>$E\rightarrow TF$</td>
<td>$E\rightarrow TF$</td>
<td>$E\rightarrow TF$</td>
<td></td>
</tr>
<tr>
<td>$T$</td>
<td>$T\rightarrow F$</td>
<td>$T\rightarrow F$</td>
<td>$T\rightarrow F$</td>
<td>$T\rightarrow F$</td>
<td>$T\rightarrow F$</td>
<td>$T\rightarrow F$</td>
<td>$T\rightarrow F$</td>
<td></td>
</tr>
<tr>
<td>$F$</td>
<td>$F\rightarrow (E)$</td>
<td>$F\rightarrow \text{id}$</td>
<td>$F\rightarrow \text{Num}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### What do we do with \( \varepsilon \) rules?

**FIRST(\( \varepsilon \))**:

- if $X \Rightarrow \varepsilon$ then $\text{FIRST}(\varepsilon) = \text{FIRST}($FOLLOW(X)$)$
But recall grammar after left recursion eliminated

<table>
<thead>
<tr>
<th></th>
<th>Goal</th>
<th>Expr</th>
<th>Expr’</th>
<th>+ Term Expr’</th>
<th>– Term Expr’</th>
<th>Term</th>
<th>Factor Term’</th>
<th>* Factor Term’</th>
<th>Factor</th>
<th>number</th>
<th>id</th>
<th>(Expr’)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>→ Expr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Expr</td>
<td>→ Term Expr’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Expr’</td>
<td>→ + Term Expr’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>[\text{c}]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>[\varepsilon]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Term</td>
<td>→ Factor Term’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Term’</td>
<td>→ * Factor Term’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>[\varepsilon]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Factor</td>
<td>→ number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>[\text{E}]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>[\text{F}]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Recall the expression grammar, after transformation

<table>
<thead>
<tr>
<th></th>
<th>Goal</th>
<th>Expr</th>
<th>Expr’</th>
<th>+ Term Expr’</th>
<th>– Term Expr’</th>
<th>Term</th>
<th>Factor Term’</th>
<th>* Factor Term’</th>
<th>Factor</th>
<th>number</th>
<th>id</th>
<th>(Expr’)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Goal</td>
<td>→ Expr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Expr</td>
<td>→ Term Expr’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Expr’</td>
<td>→ + Term Expr’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>[\text{c}]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>[\varepsilon]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Term</td>
<td>→ Factor Term’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Term’</td>
<td>→ * Factor Term’</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>[\varepsilon]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Factor</td>
<td>→ number</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td>id</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>[\text{E}]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>[\text{F}]</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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

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

A couple of routines from the expression parser

\[ \text{Goal( )} \]
\begin{align*}
\text{token} & \leftarrow \text{next_token( );} \\
\text{if (Expr( ) = true)} & \\
\text{then next compilation step; } & \\
\text{else } & \\
\text{return false; } & \\
\end{align*}

\[ \text{Expr( )} \]
\begin{align*}
\text{result} & \leftarrow \text{true; } \\
\text{if (Term( ) = false)} & \\
\text{then result} & \leftarrow \text{false; } & \\
\text{else if (EPrime( ) = false)} & \\
\text{then result} & \leftarrow \text{true; if term found } & \\
\text{return result; } & \\
\end{align*}

\[ \text{Factor( )} \]
\begin{align*}
\text{result} & \leftarrow \text{true; } \\
\text{if (token = Number)} & \\
\text{then token} & \leftarrow \text{next_token( ); } & \\
\text{else if (token = identifier)} & \\
\text{then token} & \leftarrow \text{next_token( ); } & \\
\text{else } & \\
\text{report syntax error; } & \\
\text{result} & \leftarrow \text{false; } & \\
\text{return result; } & \\
\end{align*}

EPrime, Term, & TPrime follow along the same basic lines (Figure 3.4, EAC)

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
**Recursive Descent in Object-Oriented Languages**

- Shortcomings of Recursive Descent
  - Procedural
  - Parse tree construction is a side activity
- Solution
  - Associate a class with each non-terminal symbol
    → Allocated object contains pointer to the parse tree

```java
abstract class NonTerminal {

  protected Scanner s;
  protected TreeNode tree;

  public NonTerminal(Scanner scnr) { s = scnr; tree = null; }

  public abstract boolean isPresent();

  public TreeNode abSynTree() { return tree; }
}
```

**Implementation of Expr**

```java
class Expr extends NonTerminal {

  public Expr(Scanner scnr) { super(scnr); }

  public boolean isPresent() {
    // construct AST too
    Term operand1 = new Term(s);
    if (!operand1.isPresent()) return false;
    tree = operand1.abSynTree();

    EPrime operand2 = new EPrime(s, tree);
    if (operand2.isPresent())
      tree = operand2.absSynTree();

    // here tree is either the tree for the Term
    // or the tree for Term followed by EPrime
    return true;
  }
}
```
**Implementation of EPrime**

```java
class EPrime extends NonTerminal {
    protected TreeNode exprSofar;

    public EPrime(Scanner scnr, TreeNode p)
    { super(scnr); exprSofar = p; }

    public boolean isPresent() { // construct AST too
        TokenType op = s.nextToken();
        if (op == PLUS | op == MINUS) {
            s.advance();
            Term operand2 = new Term(s);
            if (!operand2.isPresent()) throw new SyntaxError(s);
            tree = new TreeNode(op, exprSofar, operand2.absSynTree());
            Eprime operand3 = new Eprime(s, tree);
            if (operand3.isPresent()) tree = operand3.absSynTree();
        } else return false;
    }
}
```

**Tree Building in EPrime**

![Tree Building Diagram]
**Implementation of Factor**

class Factor extends NonTerminal {
    public Factor(Scanner scnr) {super(scnr);}

    public boolean isPresent() {  // with semantic processing
        TokenType op = s.nextToken();
        if (op == IDENTIFIER | op == NUMBER) {
            tree = new TreeNode(op, s.tokenValue());
            s.advance();
            return true;
        }  
        else if (op == LPAREN) {
            s.advance();
            Expr operand = new Expr(s);
            if (!operand.isPresent()) throw new SyntaxError(s);
            if (s.nextToken() != RPAREN) throw new SyntaxError(s);
            s.advance();
            tree = operand.absSynTree();
            return true;
        }  
        else return false;
    }
}

**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 α that occurs in two or more right-hand sides of A 

if α ≠ ε then replace all of the A productions,

A → αβ₁ | αβ₂ | ... | αβₙ | γ

with

A → αZ | γ
Z → β₁ | β₂ | ... | βₙ

where Z is a new element of NT

Repeat until no common prefixes remain
**Left Factoring** *(An example)*

Consider the following fragment of the expression grammar

```
Factor \rightarrow Identifier \\
    | Identifier [ ExprList ] \\
    | Identifier ( ExprList )
```

After left factoring, it becomes

```
Factor \rightarrow Identifier Arguments \\
Arguments \rightarrow [ ExprList ] \\
    | ( ExprList ) \\
    | ε
```

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

\[
\text{FIRST}(\text{rhs}_1) = \{ \text{Identifier} \} \\
\text{FIRST}(\text{rhs}_2) = \{ \} \\
\text{FIRST}(\text{rhs}_3) = \{ \} \\
\text{FIRST}(\text{rhs}_4) = \text{FOLLOW}(\text{Factor})
\]

⇒ It has the LL(1) property

---

**Left Factoring**

A graphical explanation for the same idea

```
A \rightarrow αβ_1 \\
    | αβ_2 \\
    | αβ_3
```

becomes …

```
A \rightarrow αZ \\
Z \rightarrow β_1 \\
    | β_2 \\
    | β_n
```

---

*CMSC430 Spring 2007*
**Left Factoring**

*(Generality)*

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

\[ G \rightarrow aAb \]

\[ \quad | aBB \]

\[ A \rightarrow aAb \]

\[ \quad | 0 \]

\[ B \rightarrow aBB \]

\[ \quad | 1 \]

**Problem:** need an unbounded number of a characters before you can determine whether you are in the A group or the B group.