Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel
Top-Down Parsing
Top-down parsers  \((\text{LL}(1), \text{recursive descent parsers})\)

- Start at the root of the parse tree from the start symbol and grow toward leaves (similar to a derivation)
- Pick a production and try to match the input
- Bad “pick” \(\Rightarrow\) may need to backtrack
- Some grammars are backtrack-free \((\text{predictive parsing})\)

Bottom-up parsers \((\text{LR}(1), \text{shift-reduce parsers})\)

- Start at the leaves and grow toward root
- We can think of the process as reducing the input string to the start symbol
- At each reduction step, a particular substring matching the right-side of a production is replaced by the symbol on the left-side of the production
- Bottom-up parsers handle a large class of grammars
Construct the root node of the parse tree, label it with the start symbol, and set the current-node to root node

Repeat until all the input is consumed (i.e., until the frontier of the parse tree matches the input string)

1. If the label of the current node is a non-terminal node A, select a production with A on its lhs and, for each symbol on its rhs, construct the appropriate child

2. If the current node is a terminal symbol:
   - If it matches the input string, consume it (advance the input pointer)
   - If it does not match the input string, backtrack

3. Set the current node to the next node in the frontier of the parse tree
   - If there is no node left in the frontier of the parse tree and input is not consumed, then backtrack

The key is picking the right production in step 1
   - That choice should be guided by the input string
Example

Using version with correct precedence and associativity

\[
\begin{align*}
1 & \quad S \rightarrow Expr \\
2 & \quad Expr \rightarrow Expr + Term \\
3 & \quad \quad | \quad Expr - Term \\
4 & \quad \quad | \quad Term \\
5 & \quad Term \rightarrow Term \ast Factor \\
6 & \quad \quad | \quad Term / Factor \\
7 & \quad \quad | \quad Factor \\
8 & \quad Factor \rightarrow num \\
9 & \quad \quad | \quad id
\end{align*}
\]

And the input: \( x - 2 \ast y \)
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>-</td>
<td>$S$</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>1</td>
<td>$Expr$</td>
<td>$\uparrow x - 2 * y$</td>
</tr>
<tr>
<td>2</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></td>
<td>$&lt;id,x&gt; + Term$</td>
<td>$x \uparrow - 2 * y$</td>
</tr>
</tbody>
</table>
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>( S )</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></td>
<td>( &lt;id,x&gt; + Term )</td>
<td>( x \uparrow - 2 \times y )</td>
</tr>
</tbody>
</table>

Note 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>$S$</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>-</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>
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>( S )</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>-</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 extend \( Term \), the last \( NT \) in the fringe of the parse tree
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; – Term</td>
<td>x ( - \uparrow^{2} \times y )</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id,x&gt; – Factor</td>
<td>x ( - \uparrow^{2} \times y )</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id,x&gt; – &lt;num,2&gt;</td>
<td>x ( - \uparrow^{2} \times y )</td>
</tr>
<tr>
<td>–</td>
<td>&lt;id,x&gt; – &lt;num,2&gt;</td>
<td>x ( - 2 \uparrow^{1} \times y )</td>
</tr>
</tbody>
</table>

Where are we?
- num 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 \rightarrow 2 \times y )</td>
</tr>
<tr>
<td>5</td>
<td>&lt;id,x&gt; – Term * Factor</td>
<td>( x \rightarrow 2 \times y )</td>
</tr>
<tr>
<td>7</td>
<td>&lt;id,x&gt; – Factor * Factor</td>
<td>( x \rightarrow 2 \times y )</td>
</tr>
<tr>
<td>8</td>
<td>&lt;id,x&gt; – &lt;num,2&gt; * Factor</td>
<td>( x \rightarrow 2 \times y )</td>
</tr>
<tr>
<td>-</td>
<td>&lt;id,x&gt; – &lt;num,2&gt; * Factor</td>
<td>( x \rightarrow 2 \times y )</td>
</tr>
<tr>
<td>-</td>
<td>&lt;id,x&gt; – &lt;num,2&gt; * Factor</td>
<td>( x \rightarrow 2 \times y )</td>
</tr>
<tr>
<td>9</td>
<td>&lt;id,x&gt; – &lt;num,2&gt; * &lt;id,y&gt;</td>
<td>( x \rightarrow 2 \times y )</td>
</tr>
<tr>
<td>-</td>
<td>&lt;id,x&gt; – &lt;num,2&gt; * &lt;id,y&gt;</td>
<td>( x \rightarrow 2 \times y )</td>
</tr>
</tbody>
</table>

This time, we matched and 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>( S )</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>2</td>
<td>( Expr + Term + Term )</td>
<td>( \uparrow x - 2 \times y )</td>
</tr>
<tr>
<td>2</td>
<td>( Expr + Term + Term + Term )</td>
<td>( \uparrow x - 2 \times y )</td>
</tr>
<tr>
<td>2</td>
<td>( Expr + Term + Term + \ldots + Term )</td>
<td>( \uparrow x - 2 \times y )</td>
</tr>
</tbody>
</table>

This does not terminate

- Wrong choice of expansion leads to non-termination, the parser will not backtrack since it does not get to a point where it can backtrack
- 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 there exists a non-terminal $A$ such that there 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 (without changing the language that is defined by the grammar)
To remove left recursion, we can transform the grammar.

Consider a grammar fragment of the form

\[ A \rightarrow A \alpha | \beta \]

where \( \alpha \) or \( \beta \) are strings of terminal and non-terminal symbols and neither \( \alpha \) nor \( \beta \) start with \( A \).

We can rewrite this as

\[ A \rightarrow \beta \ R \]
\[ R \rightarrow \alpha \ R | \varepsilon \]

where \( R \) is a new non-terminal.

*This accepts the same language, but uses only right recursion.*
The expression grammar contains two cases of left recursion

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

Applying the transformation yields

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

These fragments use only right recursion
Substituting back into the grammar yields

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S  →  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 Term’</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>/ Factor Term’</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>ε</td>
</tr>
<tr>
<td>10</td>
<td>Factor → num</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td>id</td>
</tr>
</tbody>
</table>

- This grammar is correct, if somewhat non-intuitive.
- A top-down parser will terminate using it.
### Left-Recursive and Right-Recursive Grammar

**Left-Recursive Grammar**

1. \( S \rightarrow \text{Expr} \)
2. \( \text{Expr} \rightarrow \text{Expr} + \text{Term} \)
3. \( \text{Expr} \rightarrow \text{Expr} - \text{Term} \)
4. \( \text{Term} \rightarrow \text{Term} \)
5. \( \text{Term} \rightarrow \text{Term} \ast \text{Factor} \)
6. \( \text{Term} \rightarrow \text{Term} / \text{Factor} \)
7. \( \text{Factor} \rightarrow \text{num} \)
8. \( \text{Factor} \rightarrow \text{id} \)

**Right-Recursive Grammar**

1. \( S \rightarrow \text{Expr} \)
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. \( \text{Term'} \rightarrow \ast \text{Factor} \text{Term'} \)
8. \( \text{Term'} \rightarrow / \text{Factor} \text{Term'} \)
9. \( \text{Term'} \rightarrow \epsilon \)
10. \( \text{Factor} \rightarrow \text{num} \)
11. \( \text{Factor} \rightarrow \text{id} \)
Eliminating Left Recursion

The previous transformation eliminates immediate left recursion
What about more general, indirect left recursion?

The general algorithm (Algorithm 4.1 in the Textbook):

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

for $i \leftarrow 1$ to $n$

    for $j \leftarrow 1$ to $i-1$

        replace each production $A_i \rightarrow A_j \gamma$ with

        $A_i \rightarrow \delta_1 \gamma / \delta_2 \gamma / \ldots / \delta_k \gamma$, where $A_j \rightarrow \delta_1 / \delta_2 / \ldots / \delta_k$

        are all the current productions for $A_j$

        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 ($A_i \rightarrow \epsilon$)
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 $A_j$ in its rhs, for $j < i$
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 and 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 $s < k$
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 look-ahead is needed?
• In general, an arbitrarily large amount
• Use the Cocke-Younger, Kasami, or Earley’s algorithm
  – Complexity is $O(|x|^3)$ where $x$ is the input string

Fortunately,
• Large subclasses of context free grammars can be parsed efficiently with limited look-ahead
  – Linear complexity, $O(|x|)$ where $x$ is the input string
• Most programming language constructs fall in those subclasses

Among the interesting subclasses are $LL(1)$ and $LR(1)$ grammars
Why is this better? It is no longer left recursive so we eventually need to eat a token before we can continue expanding our parse tree…. We can use this token to help us figure out which rule to apply.

*Enter Predictive Parsing*
Predictive Parsing

**Basic idea**

Given \( A \to \alpha \mid \beta \), the parser should be able to choose between \( \alpha \) & \( \beta \) based on peeking at the next token in the stream.

**FIRST sets**

For a string of grammar symbols \( \alpha \), define 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 \( \gamma\)

(\( \Rightarrow^* \) means a bunch of (0 or more) productions applied in series)

**The LL(1) Property**

If \( A \to \alpha \) and \( A \to \beta \) 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 look-ahead of exactly one symbol!

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

Recursive-descent parsing

- A top-down parsing method

- The term *descent* refers to the direction in which the parse tree is traversed (or built).

- Use a set of *mutually recursive* procedures (one procedure for each non-terminal symbol)
  - Start the parsing process by calling the procedure that corresponds to the start symbol
  - Each production becomes one clause in procedure

- We consider a special type of recursive-descent parsing called predictive parsing
  - *Use a look-ahead symbol to decide which production to use*
Recursive Descent Parsing

### Production Rules

1. \( S \rightarrow \text{if } E \text{ then } S \text{ else } S \)
2. \( S \rightarrow \text{begin } S L \)
3. \( S \rightarrow \text{print } E \)
4. \( L \rightarrow \text{end} \)
5. \( L \rightarrow S L \)
6. \( E \rightarrow \text{num} = \text{num} \)

### Code Snippet

```c
void S() {
    switch(lookahead) {
        case IF: match(IF); E(); match(THEN); S();
        match(ELSE); S(); break;
        case BEGIN: match(BEGIN); S(); L(); break;
        case PRINT: match(PRINT); E(); break;
        default: error();
    }
}

void E() { match(NUM); match(EQ); match(NUM); }

void L() {
    switch(lookahead) {
        case END: match(END); break;
        case SEMI: match(SEMI); S(); L(); break;
        default: error();
    }
}

void main() {
    lookahead=getNextToken();
    S();
    match(EOF);
}

void match(int token) {
    if (lookahead==token)
        lookahead=getNextToken();
    else
        error();
}
```
Execution For Input: if 2=2 then print 5=5 else print 1=1

main: call S();
S₁: find the production for (S, IF): S → if E then S else S
S₁: match(IF);
S₁: call E();
   E₁: find the production for (E, NUM): E → num = num
   E₁: match(NUM); match(EQ); match(NUM);
   E₁: return from E₁ to S₁
S₁: match(THEN);
S₁: call S();
S₂: find the production for (S, PRINT): S → print E
S₂: match(PRINT);
S₂: call E();
   E₂: find the production for (E, NUM): E → num = num
   E₂: match(NUM); match(EQ); match(NUM);
   E₂: return from E₂ to S₂
S₃: return from S₂ to S₁
S₁: match(ELSE);
S₁: call S();
S₃: find the production for (S, PRINT): S → print E
S₃: match(PRINT);
S₃: call E();
   E₃: find the production for (E, NUM): E → num = num
   E₃: match(NUM); match(EQ); match(NUM);
   E₃: return from E₃ to S₃
S₄: return from S₃ to S₁
S₁: return from S₁ to main
main: match(EOF); return success;
Left Factoring

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

• We already learned one transformation: Removing left-recursion
• There is another transformation called left-factoring

Left-Factoring Algorithm:

\[ \forall A \in NT, \]
\[ \text{find the longest prefix } \alpha \text{ that occurs in two or more right-hand sides of } A \]
\[ \text{if } \alpha \neq \varepsilon \text{ then replace all of the } A \text{ productions,} \]
\[ A \rightarrow \alpha \beta_1 | \alpha \beta_2 | \ldots | \alpha \beta_n | \gamma_1 | \gamma_2 | \ldots | \gamma_k, \]
\[ \text{with} \]
\[ A \rightarrow \alpha Z | \gamma_1 | \gamma_2 | \ldots | \gamma_k \]
\[ Z \rightarrow \beta_1 | \beta_2 | \ldots | \beta_n \]
\[ \text{where } Z \text{ is a new element of } NT \]

Repeat until no common prefixes remain
Left Factoring

A graphical explanation for the left-factoring

\[ A \rightarrow \alpha \beta_1 \]
\[ \quad | \quad \alpha \beta_2 \]
\[ \quad | \quad \alpha \beta_n \]

becomes ...

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

Consider the following fragment of the expression grammar

1  $Factor \rightarrow \text{Id}$
2      | $\text{Id} \ [ \text{ExprList} ]$
3      | $\text{Id} ( \text{ExprList} )$

After left factoring, it becomes

1  $Factor \rightarrow \text{Id Arguments}$
2  $Arguments \rightarrow \ [ \text{ExprList} ]$
3      | $( \text{ExprList} )$
4      | $\varepsilon$

$\text{FIRST}(rhs_1) = \{ \text{Id} \}$
$\text{FIRST}(rhs_2) = \{ \text{Id} \}$
$\text{FIRST}(rhs_3) = \{ \text{Id} \}$

(Intuitively, we can think of the FOLLOW of $Arguments$ as the first of $rhs_4$)

$\text{FOLLOW}(Arguments) = \text{FOLLOW}(Factor) = \{ $}

They are all distinct

$\Rightarrow$ Grammar has the $LL(1)$ property