Top-Down Parsing
Eliminating Left Recursion

Michael Wollowski

Top-Down Parsers

- A top-down parser, as the name suggests, begins with the start symbol and systematically applies the rules of the CFG, until there are no more non-terminal symbols left.
- By contrast, a bottom-up parser, begins with the tokens returned by the lexer and using the rules of the CFG replaces them with non-terminals until the start symbol is reached.
- Graphically speaking, a top-down parser builds the parse tree from the root, while a bottom-up parser begins with the leaves, combining partial parse trees.
Classes of Context-free Grammars

- LR(1) grammars include a large subset of the unambiguous CFGs.
- LR(1) grammars can be parsed, bottom-up, in a linear scan from left to right, looking at most one word ahead of the current input symbol. The “R” stands for a right-most derivation.
- The widespread availability of tools that derive parsers from LR(1) grammars has made LR(1) parsers “everyone’s” favorite grammar.
- LL(1) grammars are an important subset of the LR(1) grammars.
- LL(1) grammars can be parsed, top-down, in a linear scan from left to right, looking at most one word ahead of the current input symbol. The “L” stands for a left-most derivation.
- Many programming languages can be written in an LL(1) grammar.
Top-Down Parsing: Eliminating Left-Recursion

- We will focus on Top-down parsing for now.
- Consider the following excerpt from our grammar:

\[
\text{Expr} \rightarrow \text{Expr} + \text{Term} \\
\text{\quad |} \text{Expr} - \text{Term} \\
\text{\quad |} \text{Term}
\]

- Suppose we wish to derive: \(a + b \times c\)
- We know that \(a\) needs to be eventually replaced by \(\text{Term}\)
- The grammar is recursive.
- Computers being as systematic as they are, our compiler may decide to always use the first rule, leading to:

\[
\text{Expr} \rightarrow \text{Expr} + \text{Expr} + \text{Expr} + \ldots
\]

Left Recursion

- Formally, a grammar is left recursive if there exist an \(A \in NT\) such that there is a derivation \(A \Rightarrow^* A\alpha\), for some string \(\alpha \in (NT \cup T)^*\)
- Left-recursion typically, leads to non-termination in a top-down parser
- In a top-down parser, any recursion must be right recursion
- We would like to convert the left recursion to right recursion
Top-Down Parsing: Eliminating Left-Recursion

- We are going to re-write the productions that are left-recursive to be right-recursive.
- This requires the introduction of additional non-terminals.
- The general pattern of this transformation is as follows:
  
  $$A \rightarrow A\alpha \mid \beta$$

  is transformed to:
  
  $$A \rightarrow \beta R$$
  
  $$R \rightarrow \alpha R \mid \epsilon$$

---

Top-Down Parsing: Eliminating Left-Recursion

- Below is the result of this process as applied to the productions for $Expr$ and $Term$.

```
<table>
<thead>
<tr>
<th>Original</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Expr \rightarrow Expr + Term$</td>
<td>$Expr \rightarrow Term Expr'$</td>
</tr>
<tr>
<td>\hspace{1em} $</td>
<td>$ \hspace{1em} $Expr - Term$</td>
</tr>
<tr>
<td>\hspace{1em} $</td>
<td>$ \hspace{1em} $Term$</td>
</tr>
<tr>
<td>$Term \rightarrow Term \times Factor$</td>
<td>$Term \rightarrow Factor Term'$</td>
</tr>
<tr>
<td>\hspace{1em} $</td>
<td>$ \hspace{1em} $Term + Factor$</td>
</tr>
<tr>
<td>\hspace{1em} $</td>
<td>$ \hspace{1em} $Factor$</td>
</tr>
<tr>
<td>\hspace{1em}</td>
<td>\hspace{1em} $</td>
</tr>
</tbody>
</table>
```
Top-Down Parsing: Eliminating Left-Recursion

- Visually, this works out to be like so (courtesy of the dragon book):

![Diagram](image)

*Fig. 2.18. Left- and right-recursive ways of generating a string.*

Indirect Left Recursion

- In addition to left-recursion that occurs for a given production, there is what is called indirect left-recursion.
- *Indirect left-recursion* occurs when a sequence of productions creates indirect left-recursion.
- Example:
  
  S → Aa | b
  A → Ac | Sd

  Derivation: S → Aa → Sda
Handling Transitive 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 $A_i$ using the direct transformation

- This assumes that the initial grammar has no cycles ($A_i \Rightarrow^* A_i$), and no epsilon productions
- The inner loop must start with 1 to ensure that $A_1 \rightarrow A_1 \beta$ is transformed

---

Handling Transitive Left Recursion

- Example revised:

  $S \rightarrow Aa | b$
  
  $A \rightarrow Ac |
  
  Sd$

  - Order of non-terminals: $S, A$
  - Pairings according to algorithm: $<S, S>$ and $<A, S>$
  - There is no production $S \rightarrow S$
  - We do have a production of the form $A \rightarrow S\gamma$
  - In $A \rightarrow Sd$ we replace $S$ with $Aa$ and $b$, giving us:

    $A \rightarrow Aad | bd$
Handling Transitive Left Recursion

• We now replace all immediate left-recursion in the two $A$ productions:
  $A \rightarrow A c \mid A d \mid b d$

  like so:
  $A \rightarrow b d A'$
  $A' \rightarrow c A' \mid a d A' \mid \varepsilon$

• We’ll throw in the unmodified productions of $S$ for free:
  $S \rightarrow A a \mid b$

Handling Transitive 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 $A_s$ starting 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 $s < k$
Predictive Parsing

- A predictive parser is one that takes advantage of a predictive grammar.
- An LL(1) grammar is considered a predictive grammar.
- So far, we focused on the second L.
- In particular, we removed left-recursion.
- However, there is still a good amount of guess-work involved when parsing an expression with no look-ahead.

Predictive Parsing

- Consider the following right-recursive grammar.
- Even though all left-recursion is eliminated, we still have choices and as such backtrack points when attempting to parse the following expression: a+b*c.

```
0  Goal  ->  Expr
1  Expr  ->  Term Expr'
2  Expr' ->  + Term Expr'
3        |  - Term Expr'
4        |  ε
5  Term  ->  Factor Term'
6  Term' ->  x Factor Term'
7        |  + Factor Term'
8        |  ε
9  Factor ->  ( Expr )
10       |  num
11       |  name
```
Predictive Parsing

• Given the input a+b*c, the lexer will eventually produce the following sequence of tokens:
  <ID, a> <Operator, +> <ID, b> <Operator, *> <ID, c>
• A parser with 0 token look-ahead will proceed as follows:
  Expr
    Term Expr'
  Factor Term' Expr'
• Now it has three choices, with 0 token look ahead, the parser would try all three, say in sequence and for each attempt, it would ask the lexer what token it has and if it is not the right token, it would backtrack and try again, until, in our case, it reaches the last production and has success.
• This is silly, instead, grab the next symbol and make it available to the parser.
• This is an LL(1) grammar, also called a predictive grammar.

Left-Factoring to Eliminate Backtracking

• We now have an almost back-track free grammar.
• Consider:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Factor</td>
<td>→</td>
<td>name</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>name [ ArgList ]</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>name ( ArgList )</td>
</tr>
<tr>
<td>15</td>
<td>ArgList</td>
<td>→</td>
<td>Expr MoreArgs</td>
</tr>
<tr>
<td>16</td>
<td>MoreArgs</td>
<td>→</td>
<td>, Expr MoreArgs</td>
</tr>
<tr>
<td>17</td>
<td></td>
<td></td>
<td>ε</td>
</tr>
</tbody>
</table>

• Rules 11, 12 and 13 all begin with name.
• Name is a common pre-fix to all three rules and can be eliminated by introducing a new production:

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Factor</td>
<td>→</td>
<td>name Arguments</td>
</tr>
<tr>
<td>12</td>
<td>Arguments</td>
<td>→</td>
<td>[ ArgList ]</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td>( ArgList )</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td>ε</td>
</tr>
</tbody>
</table>
Left-Factoring to Eliminate Backtracking

- In general, we can left-factor any set of rules that has alternate right-hand sides with a common prefix.
- Convert a set of productions:
  
  \[ A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \ldots \mid \alpha \beta_n \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j \]
  
  where \( \alpha \) is a common prefix and the \( \gamma_i \)'s represent rhs that do not begin with \( \alpha \).
- To:
  
  \[ A \rightarrow \alpha B \mid \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_j \]

B → \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n

Top-Down Recursive-Descent Parsers

```java
// Main:
/* Start = Expr */
word = NextWord();
if (Expr())
    then if (word = eof)
        then return Success();
    else Fail();
Fail() report syntax error;
attempt error recovery or exit;
Expr() /* Expr = Term Expr */
    if (Term())
        then return Expr();
    else Fail();
Expr() /* Expr = Term */
    if (Term())
        then return Expr();
    else Fail();
Prime() /* Prime = Factor Prime */
    if (Prime())
        then return Prime();
    else Fail();
Factor() /* Factor = (Expr) */
    if (Expr())
        then return Factor();
    else Fail();
Term() /* Term = Factor Term */
    if (Factor())
        then return Term();
    else Fail();
```