Left Recursion

Lecture 7

Robb T. Koether

Hampden-Sydney College

Fri, Feb 6, 2009
Outline

1. Trouble with Recursive Descent
2. Left Recursion
3. Eliminating Left Recursion
   - Example
4. Advantages of Left Recursion
5. Assignment
Suppose the grammar were

\[
S \rightarrow AB \mid CD \\
A \rightarrow BC \mid CA \mid a \\
B \rightarrow CA \mid DB \mid b \\
C \rightarrow BA \mid AD \mid a \\
D \rightarrow AC \mid BD \mid b
\]

How could a top-down parser decide which production for \(S\) to use to generate \(babbb\)?
Another Problem with Recursive Descent Parsers

- Suppose the grammar were

\[ S \rightarrow SS \mid a \]

- How could the parser decide how many times to use the production \( S \rightarrow SS \) before using the production \( S \rightarrow a \)?
Futile Attempt

```c
void S()   // Match S → SS | a
{
    if (token == a)
        match(a);
    else
    {
        S();
        S();
    }
    return;
}
```
Definition (Left recursive production)

A production is left recursive if it is of the form

\[ A \rightarrow A \alpha. \]

Definition (Left recursive grammar)

A grammar is left recursive if there is a derivation

\[ A \Rightarrow^+ A \alpha \]

for some nonterminal \( A \) and string \( \alpha \).

- The method of recursive descent does not work if the grammar is left recursive.
**Left Recursion**

```c
void A() // Match A → Aα
{
    A();
    // Process a
    return;
}
```

- Attempting to match the left-recursive production
  \[ A \rightarrow A\alpha. \]
Is this grammar left recursive?
Recall that in the earlier example, we added the production

\[ S' \rightarrow SS' | \varepsilon, \]

not the production

\[ S' \rightarrow S'S | \varepsilon. \]

Why?

Are they equivalent as far as the language of the grammar is concerned?
Left recursion in a production may be removed by transforming the grammar in the following way.

Replace

\[ A \rightarrow A\alpha \mid \beta \]

with

\[ A \rightarrow \beta A' \]
\[ A' \rightarrow \alpha A' \mid \varepsilon. \]
Under the original productions, a derivation of $\beta\alpha\alpha\alpha$ is

\[ A \rightarrow A\alpha \]
\[ \rightarrow A\alpha\alpha \]
\[ \rightarrow A\alpha\alpha\alpha \]
\[ \rightarrow \beta\alpha\alpha\alpha. \]
Under the new productions, a derivation of $\beta\alpha\alpha\alpha$ is

\[
\begin{align*}
A & \rightarrow \beta A' \\
& \rightarrow \beta\alpha A' \\
& \rightarrow \beta\alpha\alpha A' \\
& \rightarrow \beta\alpha\alpha\alpha A' \\
& \rightarrow \beta\alpha\alpha\alpha.
\end{align*}
\]
Example (Eliminating Left Recursion)

Consider the left recursive grammar

\[

e \rightarrow e + t | t \\
t \rightarrow t * f | f \\
f \rightarrow (e) | \text{id}
\]
Example (Eliminating Left Recursion)

- Apply the transformation to $E$:

$$E \rightarrow TE'$$
$$E' \rightarrow + TE' \mid \varepsilon.$$

- Then apply the transformation to $T$:

$$T \rightarrow FT'$$
$$T' \rightarrow * FT' \mid \varepsilon.$$
Example (Eliminating Left Recursion)

Now the grammar is

\[
egin{align*}
E & \rightarrow TE' \\
E' & \rightarrow + TE' | \varepsilon \\
T & \rightarrow FT' \\
T' & \rightarrow \ast FT' | \varepsilon \\
F & \rightarrow (E) | \text{id}
\end{align*}
\]
void Eprime() // Match $E' \rightarrow + TE'$
{
    if (token == PLUS)
    {
        match(PLUS);
        T();
        Eprime();
    }
    return;
}

This is the function for $E'$. 
Advantages of Left Recursion

- A left recursive grammar is often more intuitive than the transformed grammar.
- A left recursive grammar will match expressions earlier, leading to shallow recursion.
- Consider parsing $a + b + c + d + e$.
- Bottom-up parsing takes advantage of the benefits of left recursion.
Example

Consider the simple grammar

\[ E \rightarrow E + E \mid \text{num} \]

Convert it to

\[ E \rightarrow \text{num } E' \]
\[ E' \rightarrow + EE' \mid \varepsilon \]
Example

Run ExpressionParser.
Assignment

Homework

- The grammar

\[ R \rightarrow R \cup R \mid RR \mid R^* \mid (R) \mid a \mid b \]

generates all regular expressions over the alphabet \{a, b\}.

- Rewrite the grammar to reflect the precedence rules.
- Eliminate left recursion.