# Left Recursion

## Lecture 7

Robb T. Koether

Hampden-Sydney College

Fri, Feb 6, 2009

# Outline

# A Problem with Recursive Descent Parsers



- Suppose the grammar were

$$\begin{aligned} S &\rightarrow AB \mid CD \\ A &\rightarrow BC \mid CA \mid \mathbf{a} \\ B &\rightarrow CA \mid DB \mid \mathbf{b} \\ C &\rightarrow BA \mid AD \mid \mathbf{a} \\ D &\rightarrow AC \mid BD \mid \mathbf{b} \end{aligned}$$

- How could a top-down parser decide which production for $S$ to use to generate $\mathbf{babbb}$?

# Another Problem with Recursive Descent Parsers



- Suppose the grammar were

$$S \quad \rightarrow \quad SS \mid a$$

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

# Futile Attempt



## Futile Attempt

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

# Left Recursion



## Definition (Left recursive production)

A production is left recursive if it is of the form

$$A \to A\alpha.$$

## 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



## Left Recursion

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

- Attempting to match the left-recursive production $A \to A\alpha$.

# Left Recursion



$$
\begin{aligned}
S &\rightarrow AB \mid CD \\
A &\rightarrow BC \mid CA \mid \mathbf{a} \\
B &\rightarrow CA \mid DB \mid \mathbf{b} \\
C &\rightarrow BA \mid AD \mid \mathbf{a} \\
D &\rightarrow AC \mid BD \mid \mathbf{b}
\end{aligned}
$$

- Is this grammar left recursive?

# Left Recursion



- Recall that in the earlier example, we added the production

$$S' \quad \rightarrow \quad SS' \mid \varepsilon,$$

not the production

$$S' \quad \rightarrow \quad S'S \mid \varepsilon.$$

- Why?
- Are they equivalent as far as the language of the grammar is concerned?

# Eliminating Left Recursion



- Left recursion in a production may be removed by transforming the grammar in the following way.
- Replace

$$A \rightarrow A\alpha \mid \beta$$

with

$$\begin{aligned} A &\rightarrow \beta A' \\ A' &\rightarrow \alpha A' \mid \varepsilon. \end{aligned}$$

# Eliminating Left Recursion



- Under the original productions, a derivation of $\beta\alpha\alpha\alpha$ is

$$\begin{aligned} A &\rightarrow A\alpha \\ &\rightarrow A\alpha\alpha \\ &\rightarrow A\alpha\alpha\alpha \\ &\rightarrow \beta\alpha\alpha\alpha. \end{aligned}$$

# Eliminating Left Recursion



- Under the new productions, a derivation of $\beta\alpha\alpha\alpha$ is

$$\begin{aligned} A &\rightarrow \beta A' \\ &\rightarrow \beta\alpha A' \\ &\rightarrow \beta\alpha\alpha A' \\ &\rightarrow \beta\alpha\alpha\alpha A' \\ &\rightarrow \beta\alpha\alpha\alpha. \end{aligned}$$

# Example



## Example (Eliminating Left Recursion)

- Consider the left recursive grammar

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

# Example



## Example (Eliminating Left Recursion)

- Apply the transformation to $E$:

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \varepsilon. \end{aligned}$$

- Then apply the transformation to $T$:

$$\begin{aligned} T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \varepsilon. \end{aligned}$$

# Example



## Example (Eliminating Left Recursion)

- Now the grammar is

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +TE' \mid \varepsilon \\ T &\rightarrow FT' \\ T' &\rightarrow *FT' \mid \varepsilon \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

# Eliminating Left Recursion



## Eliminating Left Recursion

```
void Eprime()          // Match E' → + 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

$$E \rightarrow E + E \mid \mathbf{num}$$

- Convert it to

$$\begin{aligned} E &\rightarrow \mathbf{num}\ E' \\ E' &\rightarrow +\ E E' \mid \varepsilon \end{aligned}$$

# Example



## `ExpressionParser`

- Run `ExpressionParser`.

# Assignment



## Homework

- The grammar

$$R \quad \rightarrow \quad R \cup R \mid RR \mid R^* \mid (R) \mid \mathbf{a} \mid \mathbf{b}$$

  generates all regular expressions over the alphabet $\{\mathbf{a}, \mathbf{b}\}$.
- Rewrite the grammar to reflect the precedence rules.
- Eliminate left recursion.