

COEN 175

Lecture 5: Parsing of Declarations and Statements

Using Right Recursion

- Many left-recursive grammar rules involve lists and can simply be made right-recursive:
 - A left-recursive rule for a list is $L \rightarrow L , \text{id} \mid \text{id}$.
 - A right-recursive rule for a list is $L \rightarrow \text{id} , L \mid \text{id}$.
- Often rules have actions that are performed when the rule is matched:
$$E \rightarrow T \mid E + T \{ \text{cout} \ll \text{"add"} \ll \text{endl}; \}$$
- We must be sure to carry along the action when we eliminate left recursion:

$$E \rightarrow TE'$$

$$E' \rightarrow +T \{ \text{cout} \ll \text{"add"} \ll \text{endl}; \} E' \mid \epsilon$$

Another Problem

- Consider the following grammar:

$$S \rightarrow a A \mid a B$$

$$A \rightarrow a$$

$$B \rightarrow b$$

- How do we decide which rule to take when both begin with the same prefix?

- We must rewrite the grammar to eliminate this problem by factoring out the common prefix:

$$S \rightarrow a S'$$

$$S' \rightarrow A \mid B$$

- This technique is called **left factoring**.

Left Factoring

- A grammar must be left-factored to be LL(k).
- Consider the following grammar, where each β_i begins with a different symbol:
$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n.$$

- The left-factored grammar is therefore:

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \beta_n. \end{aligned}$$

- Unlike left recursion elimination, left factoring usually involves declarations and statements, rather than expressions.

Two Examples

- Left factor the following grammar:

$$\text{declarator} \rightarrow \text{pointers } \mathbf{id} \mid \text{pointers } \mathbf{id} [\mathbf{integer}]$$

- After left factoring, the grammar becomes:

$$\text{declarator} \rightarrow \text{pointers } \mathbf{id} \text{ declarator}'$$
$$\text{declarator}' \rightarrow [\mathbf{integer}] \mid \epsilon$$

- Left factor the following grammar:

$$\text{stmt} \rightarrow \mathbf{if} (\text{expr}) \text{stmt} \mid \mathbf{if} (\text{expr}) \text{stmt} \, \mathbf{else} \, \text{stmt} \mid \text{expr} ;$$

- After left factoring, the grammar becomes:

$$\text{stmt} \rightarrow \mathbf{if} (\text{expr}) \text{stmt} \text{stmt}' \mid \text{expr} ;$$
$$\text{stmt}' \rightarrow \mathbf{else} \, \text{stmt} \mid \epsilon$$

Dangling-Else Ambiguity

- Actually, the previous grammar is ambiguous. Consider the following phrase:
if (E_1) **if** (E_2) S_1 **else** S_2
- When is S_2 executed?
 - One interpretation: when E_1 is true and E_2 is false.
 - Another interpretation: when E_1 is false.
- We can resolve this ambiguity by rewriting the grammar. However, the result is quite ugly.
 - Instead, we simply associate the **else** with the nearest unmatched **if**. So, the first interpretation is used.

Looking Across Rules

- Consider the two expressions `(int) x` and `(x + y)`.
- What rule should be used at the left parenthesis?
 - The simplest solution here is to add look-ahead.

```
void prefixExpr() {  
    ...  
    else if (lookahead == '(') {  
        match('(');  
        if (isSpecifier(lookahead)) {  
            ...  
        } else  
            postfixExpr(true);  
    } else  
        postfixExpr(false);  
}
```

```
void postfixExpr(bool lp) {  
    primaryExpr(lp);  
    ...  
}  
  
void primaryExpr(bool lp) {  
    if (lp) {  
        expr();  
        match(')');  
    } else if (...)  
        ...  
}
```

A Problem in Simple C

- Consider the following input:

```
int *foo(int a);  
int *foo(int a) { }
```

- At what point do we know that the function is being declared instead of being defined?
 - We don't know until the end of the parameters.
- Adding look-ahead will not help here.
 - Why? We can have an arbitrary number of pointer declarators.
 - The only solution is to left factor the grammar.

Rewriting the Grammar

- Consider the relevant grammar rules:

function-definition

→ specifier pointers **id** (parameters) { ... }

global-declaration

→ specifier global-declarator-list ;

global-declarator-list

→ global-declarator
| global-declarator , global-declarator-list

global-declarator

→ pointers **id**
| pointers **id** (parameters)
| pointers **id** [**integer**]

Rewriting the Grammar

- We can pull out the first declared identifier revealing the necessary left factoring:

function-or-global

→ specifier pointers **id** (parameters) { ... }
| specifier pointers **id** (parameters) *remaining-decls*
| specifier pointers **id** [**integer**] *remaining-decls*
| specifier pointers **id** *remaining-decls*

remaining-decls

→ ;
| , *global-declarator* *remaining-decls*

Left-Factored Grammar

- After initial left factoring:

function-or-global

→ *specifier pointers id function-or-global'*

function-or-global'

→ *(parameters) { ... }*

| *(parameters) remaining-decls*

| *[integer] remaining-decls*

| *remaining-decls*

remaining-decls

→ ;

| , *global-declarator remaining-decls*