CSC 3201 Compiler Construction

Department of Computer Science

SZABIST (Islamabad Campus)

Week 4 (Lecture 2)

Left Recursion

- Nonterminal symbol can derive to a sentential form with itself as the leftmost symbol.
- Direct Left recursion:
  - Direct left recursion occurs when the definition can be satisfied with only one substitution.
    - $A \rightarrow A\alpha$, where α is a sequence of nonterminals and terminals.
    - expr $\rightarrow$ expr + term

Left Recursion

- Indirect Left recursion:
  - The definition of left recursion is satisfied via several substitutions.

$A_0 \to \beta_0 A_1 \alpha_0$

$A_1 \to \beta_1 A_2 \alpha_1$

…

$A_n \to \beta_n A_0 \alpha_n$, where $\beta_0, \beta_1, \ldots, \beta_n$ are sequences that can each yield the empty string, while $\alpha_0, \alpha_1, \ldots, \alpha_n$ may be any sequences of terminal and nonterminal symbols at all.

Left Recursion

- Problem Example: No Input Consumed

P	*Sentential Form*	*input*
-	*Goal*	$\uparrow \underline{x} - \underline{2} * \underline{y}$
1	*expr*	$\uparrow \underline{x} - \underline{2} * \underline{y}$
2	*expr* + *term*	$\uparrow \underline{x} - \underline{2} * \underline{y}$
2	*expr* + *term* + *term*	$\uparrow \underline{x} - \underline{2} * \underline{y}$
2	*expr* + *term* + *term* + *term*	$\uparrow \underline{x} - \underline{2} * \underline{y}$
2	*expr* + *term* + *term* + *term* +....	$\uparrow \underline{x} - \underline{2} * \underline{y}$

Removing Left Recursion

1. $A \rightarrow A\alpha$
2. $\quad\quad | \ \beta$

Modified:

1. $A \rightarrow \beta A'$
2. $A' \rightarrow \alpha A'$
3. $\quad\quad | \ \varepsilon$

Removing Left Recursion

1. $A \rightarrow A\alpha$
2. $\quad\quad | \ \beta$

Modified:

1. $A \rightarrow \beta A'$
2. $A' \rightarrow \alpha A'$
3. $\quad\quad | \ \varepsilon$

Notes:

- (If $A' \rightarrow \varepsilon$) then A becomes β

Removing Left Recursion

```
1. expr → expr + term
2.        | expr - term
3.        | term
```

Modified:

```
1. expr  → term expr'
2. expr' → + term expr'
3. expr' → - term expr'
4.         | ε
```

Removing Left Recursion

```
1. term → term * factor
2.        | term / factor
3.        | factor
```

Modified:

```
1. term  → factor term′
2. term′ → * factor term′
3. term′ → / factor term′
4.         | ε
```

Removing Left Recursion

```
1. goal     → expr
2. expr     → expr + term
3.          | expr - term
4.          | term
5. term     → term * factor
6.          | term / factor
7.          | factor
8. factor   → number
9.          | id
10.         | (expr)
```

Removing Left Recursion

```
1.  goal    → expr
2.  expr    → term expr′
3.  expr′   → + term expr′
4.  expr′   → - term expr′
5.          | ε
6.  term    → factor term′
7.  term′   → * factor term′
8.  term′   → / factor term′
9.          | ε
10. Factor  → number
11.         | id
12.         | (expr)
```

Left Factoring

- Grammar With Common Prefixes.
  - RHS of more than one production starts with the same symbol.
- Top down parsers can not decide which production must be chosen to parse the string in hand.
- To remove this confusion, we use left factoring.

Left Factoring

- Example:

  $$E \to T+E \mid T$$

  $$T \to int \mid int^* \ T \mid (E)$$

- Impossible to predict because for T, two productions start with int.
- For E, it is not clear how to predict; the two productions start with the non-terminal T.

Left Factoring

If $\alpha \neq \varepsilon$ replace all productions

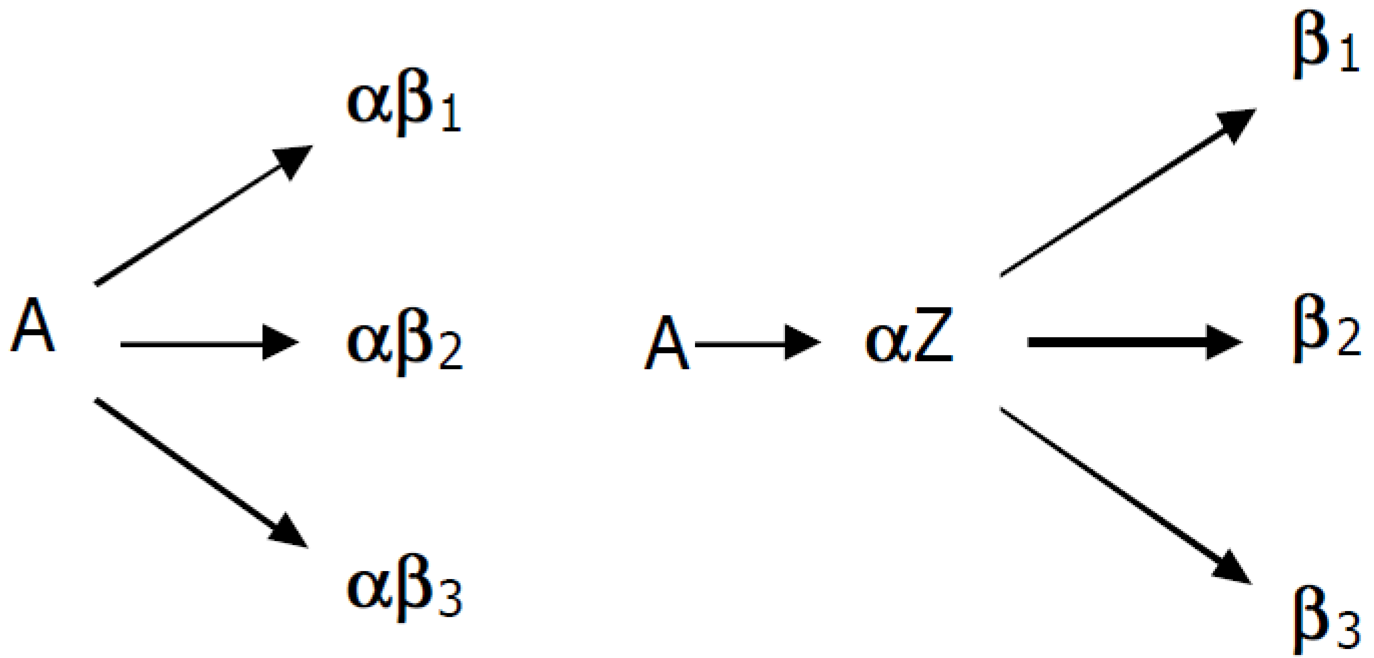
$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \dots \mid \alpha\beta_n \mid \gamma$$

with

$$A \rightarrow \alpha Z \mid \gamma$$

$Z \rightarrow \beta_1 \mid \beta_2 \mid \dots \mid \alpha\beta_n$, where Z is a new nonterminal.

Left Factoring



Left Factoring

```
Factor → id
       | id [exprList]
       | id (exprList)
```

After Left Factoring

```
Factor → id Args
Args   → [exprList]
       |  (exprList)
       |  ε
```

Bottom-Up Parsering

- More general than top-down parsing.
- Handle a large class of grammars.
- Preferred method in practice.
- Also called LR parsing; L means that tokens are read left to right and R means that the parser constructs a rightmost derivation.
- Do not need left-factored grammars.
- Can handle left-recursive grammars.

Bottom-Up Parsering

$S \rightarrow aABe$

$A \rightarrow Abc \mid b$

$B \rightarrow d$

The sentence abbcde can be reduced to S

abbcde

aAbcde

aAde

aABe

S

Right-most derivation in reverse:

$$
\begin{aligned}
S &\Rightarrow aABe \\
&\Rightarrow aAde \\
&\Rightarrow aAbcde \\
&\Rightarrow abbcde
\end{aligned}
$$

Bottom-Up Parsering

```
1. E → E + (E)
2.     | int
```

Parse of the string int + (int) + (int):

```
int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
E
```