Left

Elimination of Left Recursion

- A grammar is left recursive if it has a nonterminal A such that there is a derivation $A \overset{+}{\Rightarrow} A\alpha$ for some string α.
- Top-down parsing methods cannot handle left-recursive grammars, so a transformation that eliminates left recursion is needed.
- In Section 2.4, we discussed simple left recursion, where there was one production of the form, $A \rightarrow A\alpha$. Here we study the general case.
- In Section 2.4, we showed how the left-recursive pair of production $A \rightarrow A\alpha \mid \beta$ could be replaced by the non-left-recursive productions

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A' \mid \varepsilon$

  without changing the set of strings derivable from A. This rule by itself suffices in many grammars.

- **Example 4.8.** Consider the following grammar for arithmetic expressions.

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid id$$

- Eliminating the immediate left recursion (production of the form $A \rightarrow A\alpha$) to the production for E and then for T, we obtain

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \varepsilon$$
$$F \rightarrow (E) \mid id$$

- No matter how many A-production there are, we can eliminate immediate left recursion from them by the following technique. First, we group the A-productions as

$$A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

where no β_i begins with an A.

Then, we replace the A-productions by

$$A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$$
$$A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \varepsilon$$

- The nonterminal A generates the same strings as before but is no longer left recursive. This procedure eliminates all immediate left recursion from the A and A' productions (provided no α_i is ε), but it does not eliminate left recursion involving derivations of two or more steps. For example, consider the grammar

$$S \rightarrow Aa \mid b$$
$$A \rightarrow Ac \mid Sd \mid \varepsilon$$

The nonterminal S is left-recursive because $S \Rightarrow Aa \Rightarrow Sda$, but it is not immediately left recursive.

- Algorithm 4.1, below, will systematically eliminate left recursion from a grammar.

- It is guaranteed to work if the grammar has no cycles (derivations of the form $A \Rightarrow A$) or ε-productions (productions of the form $A \rightarrow \varepsilon$).

- Cycles can be systematically eliminated from a grammar as can ε-productions.

- **Algorithm 4.1.** Eliminating left recursion.
- *Input.* Grammar G with no cycles or ε-productions.
- *Output.* An equivalent grammar with no left recursion.
- *Method.* Apply the algorithm in Fig. 4.7 to G. Note that the resulting non-left-recursive grammar may have ε-productions.

1. Arrange the nonterminal in some order $A_1, A_2, \ldots, A_n$.
2. **for** i:= 1 **to** n **do begin**
    for j:= 1 **to** i-1 **do begin**
        replace each production of the form $A_i \rightarrow A_j\gamma$
        by the productions $A_i \rightarrow \delta_1\gamma \mid \delta_2\gamma \mid \ldots \mid \delta_k\gamma$,
        where $A_j \rightarrow \delta_1 \mid \delta_2 \mid \ldots \mid \delta_k$ are all the current
        A_j–productions;
    end
    eliminate the immediate left recursion among the
    A_i–productions
end

Fig.4.7. Algorithm to eliminate left recursion from a grammar

- The reason the procedure in Fig.4.7 works is that after the i-1st iteration of the outer for loop in step (2), any production of the form $A_k \rightarrow A_m\alpha$, where $k < i$, must have $l > k$.

- As a result, on the next iteration, the inner loop (on j) progressively raises the lower limit on m in any production $A_i \rightarrow A_m\alpha$, until we must have $m \geq i$.

- Then, eliminating immediate left recursion for the A_i-productions forces m to be greater than i.

- **Example 4.9.** Let us apply this procedure to grammar (4.12). Technically, Algorithm 4.1 is not guaranteed to work, because of the ε-production, but in this case the production $A \to \varepsilon$ turns out to be harmless.

- We order the nonterminals S, A. There is no immediate left recursion among the S-productions, so nothing happens during step (2) for the case i= 1. For i= 2, we substitute the S-productions in $A \to Sd$ to obtain the following A-productions.

  $A \to Ac \mid Aad \mid bd \mid \varepsilon$.

  Eliminating the immediate left recursion among the A-productions yields the following grammar.

  $S \to Aa \mid b$

  $A \to bdA' \mid A'$

  $A' \to cA' \mid adA' \mid \varepsilon$

Left Factoring

- Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive parsing.

- The basic idea is that when it is not clear which of two alternative productions to use to expand a nonterminal A, we may be able to rewrite the A-productions to defer the decision until we have seen enough of the input to make the right choice.

- For example, if we have the two productions

  stmt $\rightarrow$ **if** expr **then** stmt **else** stme | **if** expr **then** stmt

  on seeing the input token **if**, we cannot immediately tell which production to choose to expand stmt. In general, if $A \rightarrow \alpha\beta_1 | \alpha\beta_2$ are two A-productions and the input begins with A to $\alpha\beta_1$ or $\alpha\beta_2$.
- However, we may defer the decision by expanding A to $\alpha A'$.
- Then, after seeing the input derived from α, we expand A' to β_1 or β_2. That is, left-factored, the original production become

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$

- **Algorithm 4.2.** Left factoring a grammar
- Input. Grammar G.
- Output. An equivalent left-factored grammar.
- Method. For each nonterminal A find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \varepsilon$, i.e., there is a nontrivial common prefix, replace all the A productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α by

$$A \rightarrow \alpha A' \mid \gamma$$
$$A' \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$$

- Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

- **Example 4.10.** The following grammar abstracts the dangling-else problem:

  $$S \rightarrow iEtS \mid iEtSeS \mid a$$
  $$E \rightarrow b$$

- Here i, t, and e stand for **if**, **then** and **else**, E and S for "expression" and "statement." Left-factored, this grammar becomes:

  $$S \rightarrow iEtSS' \mid a$$
  $$S' \rightarrow eS \mid \varepsilon$$
  $$E \rightarrow b$$

- Thus, we may expand S to iEtSS’ on input i, and wait iEtS has been seen to decide whether to expand S’ to eS or to ε.
- Of course, grammars (4.13) and (4.14) are both ambiguous, and on input e, it will not be clear which alternative for S’ should be chosen.
- Example 4.19 discusses a way out of this dilemma.

- exp $\rightarrow$ exp + term | exp – term | term

    A $\rightarrow$ Bb | …
    B $\rightarrow$ Aa | …

- CASE 1: Simple immediate left recursion: A $\rightarrow$ Aα | β

    A $\rightarrow$ βA'
    A'$\rightarrow$ αA' | ε

- **Example 4.1** exp $\rightarrow$ exp addop term | term

    exp $\rightarrow$ term exp'
    exp'$\rightarrow$addop term exp' | ε

- CASE 2: General immediate left recursion:

  $A \to A\alpha_1 \mid A\alpha_2 \mid \ldots \mid A\,\alpha_n \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_m$

- **Example 4.2** exp $\to$ exp + term | exp – term | term

  exp $\to$ term exp'

  exp' $\to$ + term exp' | - term exp' | ε

- CASE 3: General left recursion:

  for i:= 1 to n do

    for j:= 1 to i-1 do

      replace each grammar rule choice of the form $A_i \rightarrow A_j\beta$ by the rule $A_i \rightarrow \alpha_1\beta \mid \alpha_2\beta \mid \ldots \mid \alpha_k\beta$ where $A_j \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_k$ is the current rule for A_j

    remove, if necessary, immediate left recursion involving A_i

- **Example 4.3**
- $A \rightarrow Ba \mid Aa \mid c$
  $B \rightarrow Bb \mid Ab \mid d$

GRAMMARS

- A grammar for a programming language is a formal description of the syntax, or form, of programs and individual statements written in the language. The grammar does not describe the semantics, or meaning.

- **FIGURE 5.2** Simplified Pascal grammar:

1.<prog> ::= PROGRAM <prog-name>;
VAR <dec-list> BEGIN <stmt-list> END.

2.<prog-name> ::= id

3.<dec-list> ::= <dec> | <dec-list> ; <dec>

4.<dec> ::= <id-list> : <type>

5.<type> ::= INTEGER

6. <id-list> ::= id | <id-list>, id

7. <stmt-list> ::= <stmt> | <stmt-list> ; <stmt>
8. <stmt> ::= <assign> | <read> | <write> | <for>
9. <assign> ::= id := <exp>
10. <exp> ::= <term> | <exp> + <term> | <exp> - <term>
11. <term> ::= <factor> | <term> * <factor> | <term> DIV <factor>
12. <factor> ::= id | int | (<exp>)
13. <read> ::= READ (<id-list>)
14. <write> ::= WRITE (<id-list>)
15. <for> ::= FOR <index-exp> DO <body>
16. <index-exp> ::= id := <exp> TO <exp>
17. <body> ::= <stmt> | BEGIN <stmt-list> END

- FIGURE 5.9 Simplified Pascal grammar modified for recursive-descent parse.

1. <prog>::= PROGRAM <prog-name>;
   VAR <dec-list> BEGIN <stmt-list> END.

2.<prog-name> ::= id

3a.<dec-list> ::= <dec> { ; <dec> }

4. <dec> ::= <id-list> : <type>

5. <type> ::= INTEGER

6a.< id-list> ::= id { , id }

7a.<stmt-list> ::= <stmt> { ; <stmt> }

8. <stmt> ::= <assign> | <read> | <write> | <for>

9. <assign> ::= id := <exp>

10a. <exp> ::= <term> { + <term> | - <term> }

11a.	<term>	::= <factor> { * <factor> | DIV <factor> }
12.	<factor>	::= id | int | (<exp>)
13.	<read>	::= READ (<id-list>)
14.	<write>	::= WRITE (<id-list>)
15.	<for>	::= FOR <index-exp> DO <body>
16.	<index-exp>	::= id := <exp> TO <exp>
17.	<body>	::= <stmt> | BEGIN <stmt-list> END