

CSE 309 (Compilers)

Basic Concepts of Parsing in Compiler

Dr. Muhammad Masroor Ali

Professor

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
Dhaka-1205, Bangladesh

January 2021

Version: 1.1, Last modified: March 16, 2021

Top-Down Parsing

- We introduce top-down parsing by considering a grammar that is well-suited for this class of methods.

Top-Down Parsing — *continued*

```
stmt   →  expr ;  
        |  if ( expr ) stmt  
        |  for ( optexpr ; optexpr ; optexpr ) stmt  
        |  other  
  
optexpr →   $\epsilon$   
        |  expr
```

A grammar for some statements in C and Java

- The grammar generates a subset of the statements of C or Java.
- We use the boldface terminals **if** and **for** for the keywords “if” and “for”, respectively, to emphasize that these character sequences are treated as units, *i.e.*, as single terminal symbols.

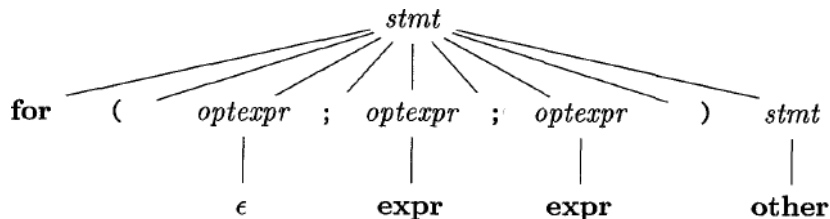
Top-Down Parsing — *continued*

```
stmt  →  expr ;  
        |  if ( expr ) stmt  
        |  for ( optexpr ; optexpr ; optexpr ) stmt  
        |  other  
  
optexpr →   $\epsilon$   
          |  expr
```

A grammar for some statements in C and Java

- Further, the terminal **expr** represents expressions.
- A more complete grammar would use a nonterminal *expr* and have productions for nonterminal *expr*.
- Similarly, **other** is a terminal representing other statement constructs.

Top-Down Parsing — *continued*



A parse tree according to the grammar

- The top-down construction of a parse tree is done by starting with the root, labeled with the starting nonterminal *stmt*, and repeatedly performing the following two steps.

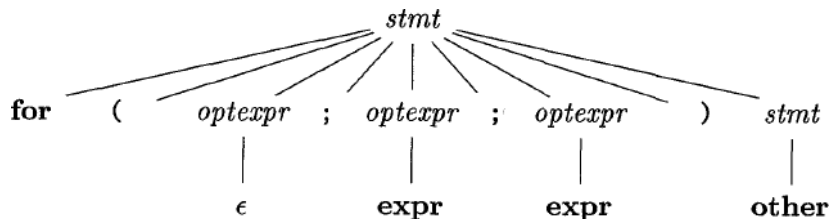
Top-Down Parsing — *continued*

$$\begin{array}{lcl} stmt & \rightarrow & \mathbf{expr} ; \\ & | & \mathbf{if} (\mathbf{expr}) stmt \\ & | & \mathbf{for} (optexpr ; optexpr ; optexpr) stmt \\ & | & \mathbf{other} \end{array}$$
$$\begin{array}{lcl} optexpr & \rightarrow & \epsilon \\ & | & \mathbf{expr} \end{array}$$

A grammar for some statements in C and Java

- The top-down construction of a parse tree is done by starting with the root, labeled with the starting nonterminal *stmt*, and repeatedly performing the following two steps.

Top-Down Parsing — *continued*



A parse tree according to the grammar

1. At node N , labeled with nonterminal A , select one of the productions for A and construct children at N for the symbols in the production body.

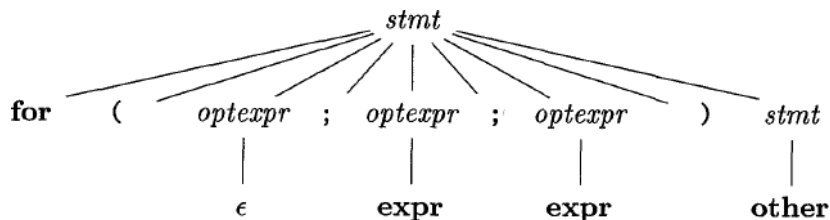
Top-Down Parsing — *continued*

$$\begin{array}{lcl} stmt & \rightarrow & \mathbf{expr} ; \\ & | & \mathbf{if} (\mathbf{expr}) stmt \\ & | & \mathbf{for} (optexpr ; optexpr ; optexpr) stmt \\ & | & \mathbf{other} \end{array}$$
$$\begin{array}{lcl} optexpr & \rightarrow & \epsilon \\ & | & \mathbf{expr} \end{array}$$

A grammar for some statements in C and Java

1. At node N , labeled with nonterminal A , select one of the productions for A and construct children at N for the symbols in the production body.

Top-Down Parsing — *continued*



A parse tree according to the grammar

- Find the next node at which a subtree is to be constructed, typically the leftmost unexpanded nonterminal of the tree.

Top-Down Parsing — *continued*

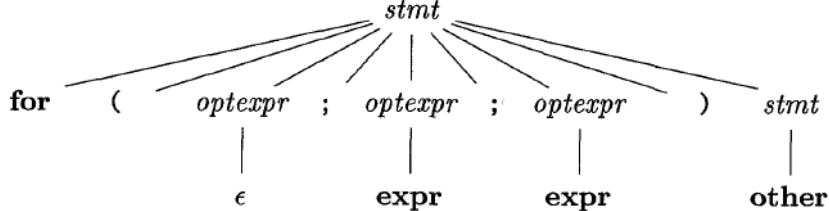
$$\begin{array}{lcl} stmt & \rightarrow & \mathbf{expr} ; \\ & | & \mathbf{if} (\mathbf{expr}) stmt \\ & | & \mathbf{for} (optexpr ; optexpr ; optexpr) stmt \\ & | & \mathbf{other} \end{array}$$
$$\begin{array}{lcl} optexpr & \rightarrow & \epsilon \\ & | & \mathbf{expr} \end{array}$$

A grammar for some statements in C and Java

- Find the next node at which a subtree is to be constructed, typically the leftmost unexpanded nonterminal of the tree.

Top-Down Parsing — *continued*

- For some grammars, the above steps can be implemented during a single left-to-right scan of the input string.
- The current terminal being scanned in the input is frequently referred to as the *lookahead* symbol.
- Initially, the lookahead symbol is the first, *i.e.*, leftmost terminal of the input string.

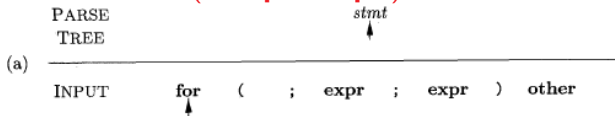


A parse tree according to the grammar .

- Figure illustrates the construction of the parse tree in for the input string

for (; expr ; expr) other

for (; expr ; expr) other



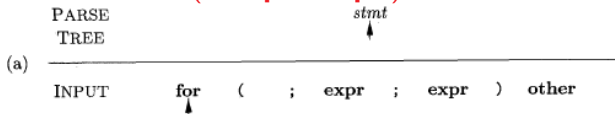
$stmt \rightarrow$ **expr ;**
 | **if (expr) stmt**
 | **for (optexpr ; optexpr ; optexpr) stmt**
 | **other**

$optexpr \rightarrow$ ϵ
 | **expr**

A grammar for some statements in C and Java

- Initially, the terminal **for** is the lookahead symbol, and the known part of the parse tree consists of the root, labeled with the starting nonterminal *stmt*.
- The objective is to construct the remainder of the parse tree in such a way that the string generated by the parse tree matches the input string.

for (; expr ; expr) other



$stmt \rightarrow$

- `expr ;`
- `if (expr) stmt`
- `for (optexpr ; optexpr ; optexpr) stmt`
- `other`

$optexpr \rightarrow$

- ϵ
- `expr`

A grammar for some statements in C and Java

- For a match to occur, the nonterminal *stmt* in must derive a string that starts with the lookahead symbol **for**.
- In the grammar, there is just one production for *stmt* that can derive such a string, so we select it, and construct the children of the root labeled with the symbols in the production body.
- This expansion of the parse tree is shown in (b).

for (; expr ; expr) other

PARSE
TREE



(b)

INPUT **for** (; **expr** ; **expr**) **other**

$stmt \rightarrow$ **expr ;**
 | **if (expr) stmt**
 | **for (optexpr ; optexpr ; optexpr) stmt**
 | **other**

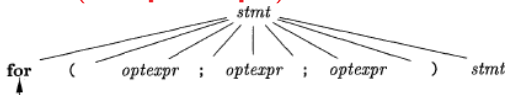
$optexpr \rightarrow$ ϵ
 | **expr**

A grammar for some statements in C and Java

- For a match to occur, the nonterminal *stmt* in must derive a string that starts with the lookahead symbol **for**.
- In the grammar, there is just one production for *stmt* that can derive such a string, so we select it, and construct the children of the root labeled with the symbols in the production body.
- This expansion of the parse tree is shown in (b).

for (; expr ; expr) other

PARSE
TREE



(b)

INPUT **for** (; **expr** ; **expr**) other

stmt → **expr ;**
 | **if (expr) stmt**
 | **for (optexpr ; optexpr ; optexpr) stmt**
 | **other**

optexpr → ϵ
 | **expr**

A grammar for some statements in C and Java

- Each of the three snapshots has arrows marking the lookahead symbol in the input and the node in the parse tree that is being considered.
- Once children are constructed at a node, we next consider the leftmost child.
- In (b), children have just been constructed at the root, and the leftmost child labeled with **for** is being considered.

for (; expr ; expr) other

```
stmt  →  expr ;  
      |  if ( expr ) stmt  
      |  for ( optexpr ; optexpr ; optexpr ) stmt  
      |  other  
  
optexpr → ε  
        |  expr
```

A grammar for some statements in C and Java

- When the node being considered in the parse tree is for a terminal, and the terminal matches the lookahead symbol, then we advance in both the parse tree and the input.
- The next terminal in the input becomes the new lookahead symbol, and the next child in the parse tree is considered.

for (; expr ; expr) other

PARSE
TREE



(c)

INPUT

for (; expr ; expr) other



- In (c), the arrow in the parse tree has advanced to the next child of the root, and the arrow in the input has advanced to the next terminal, which is (.

$stmt \rightarrow$ **expr ;**
 $\quad \quad \quad |$ **if (expr) stmt**
 $\quad \quad \quad |$ **for (optexpr ; optexpr ; optexpr) stmt**
 $\quad \quad \quad |$ **other**

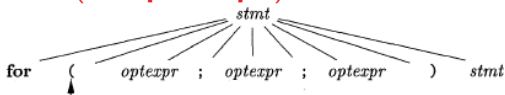
$optexpr \rightarrow$ ϵ
 $\quad \quad \quad |$ **expr**

A grammar for some statements in C and Java

- A further advance will take the arrow in the parse tree to the child labeled with nonterminal *optexpr* and take the arrow in the input to the terminal ;.

for (; expr ; expr) other

PARSE
TREE



(c)

INPUT

for (; expr ; expr) other

$stmt \rightarrow$ **expr ;**
 | **if (expr) stmt**
 | **for (optexpr ; optexpr ; optexpr) stmt**
 | **other**

$optexpr \rightarrow$ ϵ
 | **expr**

A grammar for some statements in C and Java

- At the nonterminal node labeled *optexpr*, we repeat the process of selecting a production for a nonterminal.
- Productions with ϵ as the body (“ ϵ -productions”) require special treatment.
- For the moment, we use them as a default when no other production can be used.

for (; expr ; expr) other

PARSE
TREE



(c)

INPUT

for (; expr ; expr) other

- With nonterminal *optexpr* and lookahead ;, the ϵ -production is used, since ; does not match the only other production for *optexpr*, which has terminal **expr** as its body.

stmt \rightarrow **expr ;**
 | **if (expr) stmt**
 | **for (optexpr ; optexpr ; optexpr) stmt**
 | **other**

optexpr \rightarrow ϵ
 | **expr**

A grammar for some statements in C and Java

Top-Down Parsing — *continued*

- In general, the selection of a production for a nonterminal may involve trial- and-error.
- That is, we may have to try a production and backtrack to try another production if the first is found to be unsuitable.
- A production is unsuitable if, after using the production, we cannot complete the tree to match the input string.
- Backtracking is not needed, however, in an important special case called predictive parsing.

Predictive Parsing

- Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input.
- One procedure is associated with each nonterminal of a grammar.
- Here, we consider a simple form of recursive-descent parsing, called predictive parsing, in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal.
- The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree, if desired.

Predictive Parsing

- Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input.
- One procedure is associated with each nonterminal of a grammar.
- Here, we consider a simple form of recursive-descent parsing, called predictive parsing, in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal.
- The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree, if desired.

Predictive Parsing

- Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input.
- One procedure is associated with each nonterminal of a grammar.
- Here, we consider a simple form of recursive-descent parsing, called predictive parsing, in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal.
- The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree, if desired.

```

void stmt() {
    switch ( lookahead ) {
        case expr:
            match(expr); match(';'); break;
        case if:
            match(if); match('('); match(expr); match(')'); stmt();
            break;
        case for:
            match(for); match('(');
            optexpr(); match(';'); optexpr(); match(';'); optexpr();
            match(')'); stmt(); break;
        case other:
            match(other); break;
        default:
            report("syntax error");
    }
}

void optexpr() {
    if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
    if ( lookahead == t ) lookahead = nextTerminal;
    else report("syntax error");
}

```

Pseudocode for a predictive parser

- The predictive parser consists of procedures for the nonterminals *stmt* and *optexpr* of the grammar and an additional procedure *match*, used to simplify the code for *stmt* and *optexpr*.

$$\begin{array}{lcl}
 \textit{stmt} & \rightarrow & \textbf{expr} ; \\
 & | & \textbf{if} (\textbf{expr}) \textit{stmt} \\
 & | & \textbf{for} (\textit{optexpr} ; \textit{optexpr} ; \textit{optexpr}) \textit{stmt} \\
 & | & \textbf{other}
 \end{array}$$

$$\begin{array}{lcl}
 \textit{optexpr} & \rightarrow & \epsilon \\
 & | & \textbf{expr}
 \end{array}$$

A grammar for some statements in C and Java

- The predictive parser consists of procedures for the nonterminals *stmt* and *optexpr* of the grammar and an additional procedure *match*, used to simplify the code for *stmt* and *optexpr*.

Predictive Parsing — *continued*

```
void match(terminal t) {  
    if ( lookahead == t ) lookahead = nextTerminal;  
    else report("syntax error");  
}
```

- Procedure *match*(*t*) compares its argument *t* with the lookahead symbol and advances to the next input terminal if they match.
- Thus match changes the value of variable *lookahead*, a global variable that holds the currently scanned input terminal.

```

void stmt() {
    switch ( lookahead ) {
    case expr:
        match(expr); match(''); break;
    case if:
        match(if); match('('); match(expr); match(')'); stmt();
        break;
    case for:
        match(for); match('(');
        optexpr(); match(''); optexpr(); match(''); optexpr();
        match(')'); stmt(); break;
    case other:
        match(other); break;
    default:
        report("syntax error");
    }
}

```

- Parsing begins with a call of the procedure for the starting nonterminal *stmt*.

```

void stmt() {
    switch ( lookahead ) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')'); stmt();
        break;
    case for:
        match(for); match('(');
        optexpr(); match(';'); optexpr(); match(';'); optexpr();
        match(')'); stmt(); break;
    case other:
        match(other); break;
    default:
        report("syntax error");
    }
}

```

- With the input **for (; expr ; expr) other**, lookahead is initially the first terminal **for**.

```

void stmt() {
    switch ( lookahead ) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')'); stmt();
        break;
    case for:
        match(for); match('(');
        optexpr(); match(';'); optexpr(); match(';'); optexpr();
        match(')'); stmt(); break;
    case other:
        match(other); break;
    default:
        report("syntax error");
    }
}

```

- Procedure *stmt* executes code corresponding to the production

$stmt \rightarrow \mathbf{for} (optexpr ; optexpr ; optexpr) stmt$

```

void stmt() {
    switch ( lookahead ) {
    case expr:
        match(expr); match(' '); break;
    case if:
        match(if); match(' ( '); match(expr); match(' ) '); stmt();
        break;
    case for:
        match(for); match(' ( ');
        optexpr(); match(' ; '); optexpr(); match(' ; '); optexpr();
        match(' ) '); stmt(); break;
    case other:
        match(other); break;
    default:
        report("syntax error");
    }
}

```

- In the code for the production body — that is, the **for** case of procedure *stmt*— each terminal is matched with the lookahead symbol.


```

void stmt() {
    switch ( lookahead ) {
    case expr:
        match(expr); match(';'); break;
    case if:
        match(if); match('('); match(expr); match(')'); stmt();
        break;
    case for:
        match(for); match('(');
        optexpr(); match(';'); optexpr(); match(';'); optexpr();
        match(')'); stmt(); break;
    case other:
        match(other); break;
    default:
        report("syntax error");
    }
}

```

- Each nonterminal leads to a call of its procedure, in the following sequence of calls:

```

match(for); match('(');
optexpr(); match(';'); optexpr(); match(';'); optexpr();
match(')'); stmt();

```

Predictive Parsing — *continued*

- Predictive parsing relies on information about the first symbols that can be generated by a production body.
- More precisely, let α be a string of grammar symbols (terminals and/or nonterminals).
- We define $\text{FIRST}(\alpha)$ to be the set of terminals that appear as the first symbols of one or more strings of terminals generated from α .
- If α is ϵ or can generate ϵ , then ϵ is also in $\text{FIRST}(\alpha)$.

- Here, we shall just use ad hoc reasoning to deduce the symbols in $\text{FIRST}(\alpha)$.
- Typically, α will either begin with a terminal, which is therefore the only symbol in $\text{FIRST}(\alpha)$.
- Or α will begin with a nonterminal whose production bodies begin with terminals, in which case these terminals are the only members of $\text{FIRST}(\alpha)$.

Predictive Parsing — *continued*

$$\begin{array}{lcl} stmt & \rightarrow & \mathbf{expr} ; \\ & | & \mathbf{if} (\mathbf{expr}) stmt \\ & | & \mathbf{for} (optexpr ; optexpr ; optexpr) stmt \\ & | & \mathbf{other} \end{array}$$
$$\begin{array}{lcl} optexpr & \rightarrow & \epsilon \\ & | & \mathbf{expr} \end{array}$$

A grammar for some statements in C and Java

- With respect to the grammar, the following are correct calculations of FIRST.
 - $\text{FIRST}(stmt) = \{\mathbf{expr}, \mathbf{if}, \mathbf{for}, \mathbf{other}\}$
 - $\text{FIRST}(\mathbf{expr} ;) = \{\mathbf{expr}\}$

- The FIRST sets must be considered if there are two productions $A \rightarrow \alpha$, and $A \rightarrow \beta$.
- Ignoring ϵ -productions for the moment, predictive parsing requires $\text{FIRST}(\alpha)$ and $\text{FIRST}(\beta)$ to be disjoint.
- The lookahead symbol can then be used to decide which production to use.
- If the lookahead symbol is in $\text{FIRST}(\alpha)$, then α is used.
- Otherwise, if the lookahead symbol is in $\text{FIRST}(\beta)$, then β is used.

When to Use ϵ -Productions

```
void optexpr() {  
    if ( lookahead == expr ) match(expr);  
}
```

- Our predictive parser uses an ϵ -production as a default when no other production can be used.
- With the input **for** (; **expr** ; **expr**) **other**, after the terminals **for** and (are matched, the lookahead symbol is ;.

When to Use ϵ -Productions

```
void optexpr() {  
    if ( lookahead == expr ) match(expr);  
}
```

- At this point procedure *optexpr* is called, and the code
 if (*lookahead* == **expr**) *match*(**expr**);
 in its body is executed.

When to Use ϵ -Productions

```
void optexpr() {  
    if ( lookahead == expr ) match(expr);  
}
```

- Nonterminal *optexpr* has two productions, with bodies **expr** and ϵ .
- The lookahead symbol “;” does not match the terminal **expr**, so the production with body **expr** cannot apply.

When to Use ϵ -Productions

```
void optexpr() {  
    if ( lookahead == expr ) match(expr);  
}
```

- In fact, the procedure returns without changing the lookahead symbol or doing anything else.
- Doing nothing corresponds to applying an ϵ -production.

When to Use ϵ -Productions — *continued*

- More generally, consider a variant of the productions where *optexpr* generates an expression nonterminal instead of the terminal **expr**:

$$\begin{array}{ccc} \textit{optexpr} & \rightarrow & \textit{expr} \\ & | & \epsilon \end{array}$$

- Thus, *optexpr* either generates an expression using nonterminal *expr* or it generates ϵ .
- While parsing *optexpr*, if the lookahead symbol is not in $\text{FIRST}(\textit{expr})$ then the ϵ -production is used.

Designing a Predictive Parser

- A predictive parser is a program consisting of a procedure for every nonterminal.
- The procedure for nonterminal A does two things.
 1. It decides which A -production to use by examining the lookahead symbol.

The production with body α (where α is not ϵ , the empty string) is used if the lookahead symbol is in $\text{FIRST}(\alpha)$.

If there is a conflict between two nonempty bodies for any lookahead symbol, then we cannot use this parsing method on this grammar.

In addition, the ϵ -production for A , if it exists, is used if the lookahead symbol is not in the FIRST set for any other production body for A .

Designing a Predictive Parser

- A predictive parser is a program consisting of a procedure for every nonterminal.
 - The procedure for nonterminal A does two things.
2. The procedure then mimics the body of the chosen production.

That is, the symbols of the body are “executed” in turn, from the left.

A nonterminal is “executed” by a call to the procedure for that nonterminal, and a terminal matching the lookahead symbol is “executed” by reading the next input symbol.

If at some point the terminal in the body does not match the lookahead symbol, a syntax error is reported.

$$\begin{array}{lcl}
 stmt & \rightarrow & \mathbf{expr} \ ; \\
 & | & \mathbf{if} \ (\ \mathbf{expr} \) \ stmt \\
 & | & \mathbf{for} \ (\ optexpr \ ; \ optexpr \ ; \ optexpr \) \ stmt \\
 & | & \mathbf{other}
 \end{array}$$

$$\begin{array}{lcl}
 optexpr & \rightarrow & \epsilon \\
 & | & \mathbf{expr}
 \end{array}$$

A grammar for some statements in C and Java

```

void stmt() {
    switch ( lookahead ) {
        case expr:
            match(expr); match(';'); break;
        case if:
            match(if); match('('); match(expr); match(')'); stmt();
            break;
        case for:
            match(for); match('(');
            optexpr(); match(';'); optexpr(); match(';'); optexpr();
            match(')'); stmt(); break;
        case other:
            match(other); break;
        default:
            report("syntax error");
    }
}

void optexpr() {
    if ( lookahead == expr ) match(expr);
}

void match(terminal t) {
    if ( lookahead == t ) lookahead = nextTerminal;
    else report("syntax error");
}

```

Pseudocode for a predictive parser

Designing a Predictive Parser — *continued*

- Just as a translation scheme is formed by extending a grammar, a syntax-directed translator can be formed by extending a predictive parser.
- The following limited construction suffices for the present:
 1. Construct a predictive parser, ignoring the actions in productions.
 2. Copy the actions from the translation scheme into the parser.

If an action appears after grammar symbol X in production p , then it is copied after the implementation of X in the code for p .

Otherwise, if it appears at the beginning of the production, then it is copied just before the code for the production body.

Designing a Predictive Parser — *continued*

- Just as a translation scheme is formed by extending a grammar, a syntax-directed translator can be formed by extending a predictive parser.
- The following limited construction suffices for the present:
 1. Construct a predictive parser, ignoring the actions in productions.
 2. Copy the actions from the translation scheme into the parser.

If an action appears after grammar symbol X in production p , then it is copied after the implementation of X in the code for p .

Otherwise, if it appears at the beginning of the production, then it is copied just before the code for the production body.

Designing a Predictive Parser — *continued*

- Just as a translation scheme is formed by extending a grammar, a syntax-directed translator can be formed by extending a predictive parser.
- The following limited construction suffices for the present:
 1. Construct a predictive parser, ignoring the actions in productions.
 2. Copy the actions from the translation scheme into the parser.

If an action appears after grammar symbol X in production p , then it is copied after the implementation of X in the code for p .

Otherwise, if it appears at the beginning of the production, then it is copied just before the code for the production body.

Left Recursion

- It is possible for a recursive-descent parser to loop forever.
- A problem arises with “left-recursive” productions like

$$expr \rightarrow expr + term$$

where the leftmost symbol of the body is the same as the nonterminal at the head of the production.

- Suppose the procedure for *expr* decides to apply this production.
- The body begins with *expr* so the procedure for *expr* is called recursively.
- Since the lookahead symbol changes only when a terminal in the body is matched, no change to the input took place between recursive calls of *expr*.
- As a result, the second call to *expr* does exactly what the first call did, which means a third call to *expr*, and so on, forever.

Left Recursion

- It is possible for a recursive-descent parser to loop forever.
- A problem arises with “left-recursive” productions like

$$expr \rightarrow expr + term$$

where the leftmost symbol of the body is the same as the nonterminal at the head of the production.

- Suppose the procedure for *expr* decides to apply this production.
- The body begins with *expr* so the procedure for *expr* is called recursively.
- Since the lookahead symbol changes only when a terminal in the body is matched, no change to the input took place between recursive calls of *expr*.
- As a result, the second call to *expr* does exactly what the first call did, which means a third call to *expr*, and so on, forever.

Left Recursion — *continued*

- A left-recursive production can be eliminated by rewriting the offending production.
- Consider a nonterminal A with two productions

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

where α and β are sequences of terminals and nonterminals that do not start with A .

- For example, in

$$expr \rightarrow expr + term \mid term$$

nonterminal $A = expr$, string $\alpha = +term$, and string $\beta = term$.

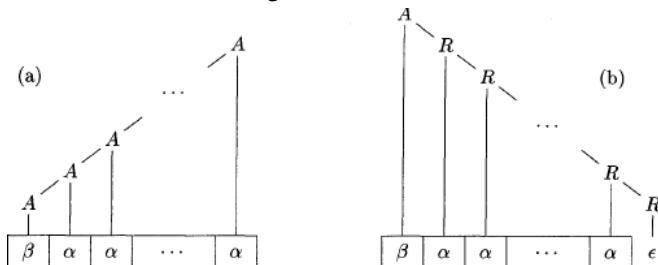
$$A \rightarrow A\alpha \mid \beta$$

- The nonterminal A and its production are said to be left recursive, because the production $A \rightarrow A\alpha$ has A itself as the leftmost symbol on the right side.

Left Recursion — *continued*

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

- Repeated application of this production builds up a sequence of α 's to the right of A .



Left- and right-recursive ways of generating a string

- When A is finally replaced by β , we have β followed by a sequence of zero or more α 's.

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

- The same effect can be achieved, by rewriting the productions for A in the following manner, using a new nonterminal R :

$$A \rightarrow \beta R$$

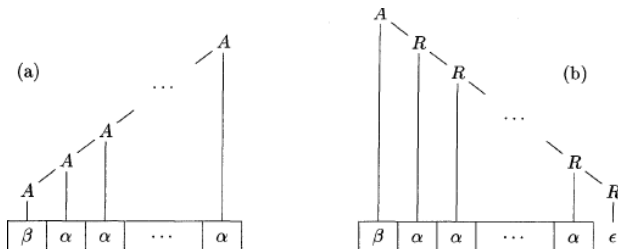
$$R \rightarrow \alpha R \mid \epsilon$$

Left Recursion — *continued*

$$A \rightarrow \beta R$$

$$R \rightarrow \alpha R \mid \epsilon$$

- Right-recursive productions lead to trees that grow down towards the right.



Left- and right-recursive ways of generating a string

End of Slides

