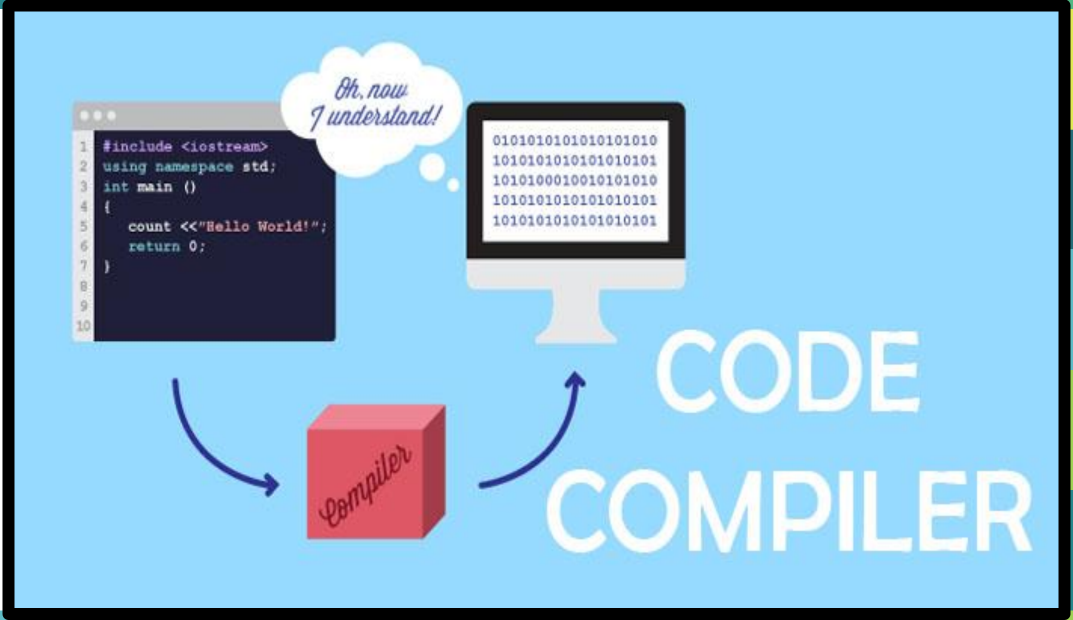
CSE- 303

Compiler

Chapter - 2



Context-free Grammers

- CFG has four components:

1) A set of terminals
2) A set of nonterminals
3) A set of productions
4) Start symbol

$$
\begin{array}{llll}
L_1 = & S & \rightarrow & ABC \\
& A & \rightarrow & \mathbf{a}A\mathbf{b} \mid \varepsilon \\
& B & \rightarrow & \mathbf{b}B \mid \mathbf{b} \\
& C & \rightarrow & \mathbf{c}C \mid \varepsilon \\
\\
L_2 = & S & \rightarrow & ASC \mid B\mathbf{c} \\
& A & \rightarrow & \mathbf{a} \\
& B & \rightarrow & \mathbf{b}B \mid \varepsilon \\
& C & \rightarrow & \mathbf{c}C \mid \mathbf{c}
\end{array}
$$

Basic Concepts of Parsing in Compiler

- Parsing: process of determining how a string of terminals can be generated by a grammar.

- Mostly of two types:

1) Top-down Parsing: construction starts at the root and proceeds towards the leaves
2) Bottom-up Parsing: construction starts at the leaf and proceeds towards the root

Top-down Parsing

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

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

A grammar for some statements in C and Java

- The grammar generates a subset of the statements of C or Java.

Top-down Parsing

- The **terminal expr** represents expressions. A more complete grammar would use a **nonterminal expr** and have productions for **nonterminal expr**.

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

- Similarly, **other** is a terminal representing other statement constructs.

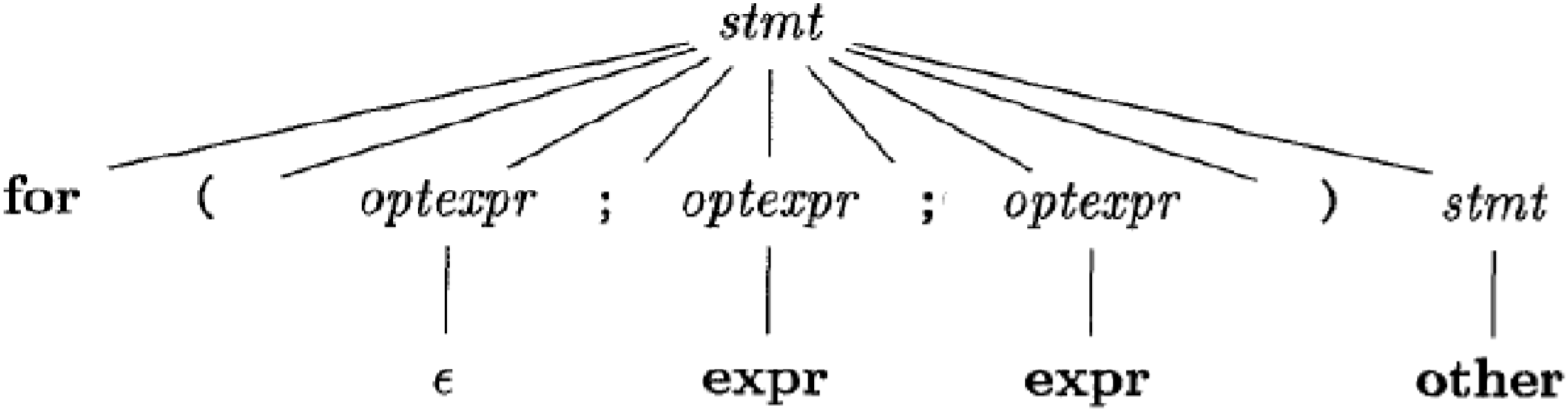
Top-down Parsing

- 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.

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

Top-down Parsing

- Input String: for (; expr ; expr) other



A parse tree according to the grammar

Top-down Parsing

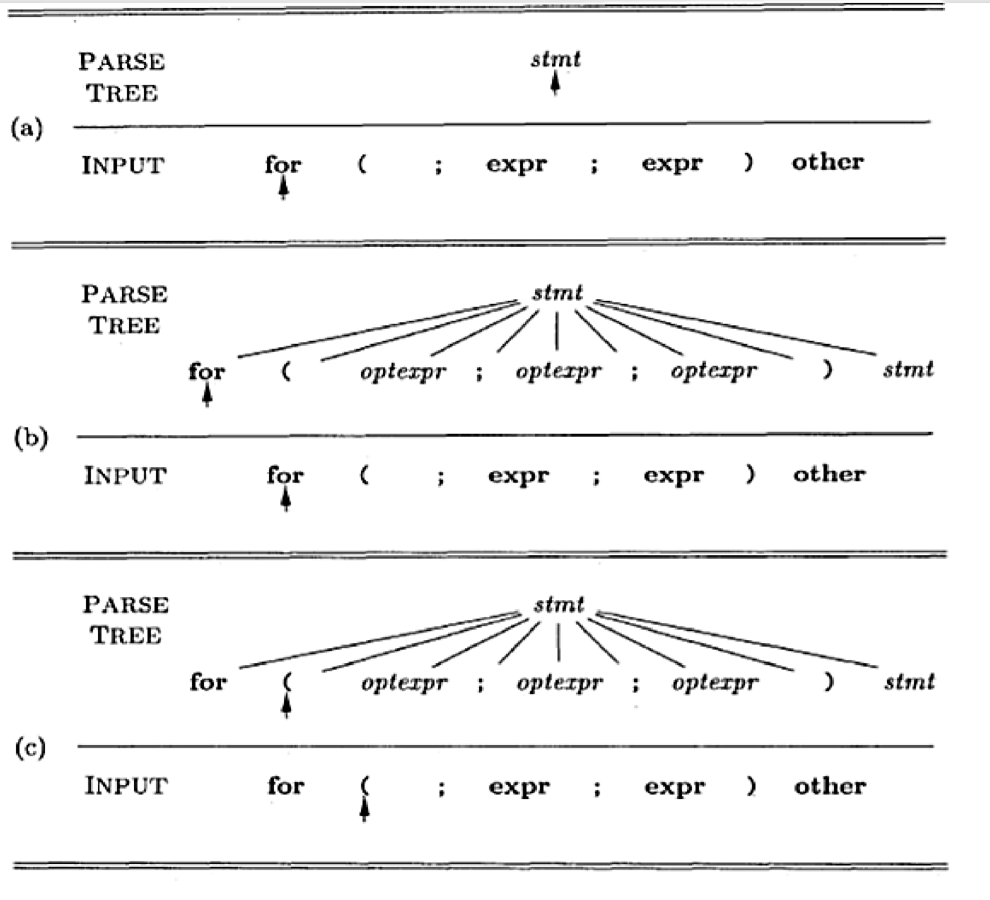
- For some grammars, the above steps can be implemented during a single left-to-right scan of the input string.

- 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.

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

for (; expr ; expr) other



Top-down parsing while scanning the input from left to right

Top-down Parsing

- 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.

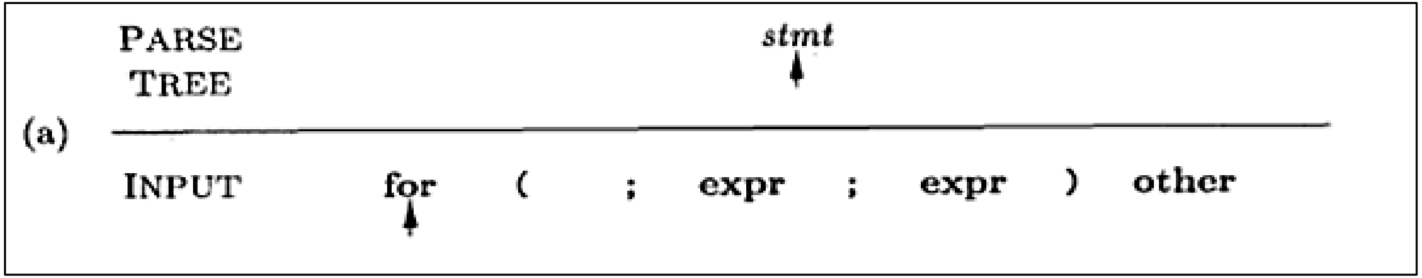
$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{expr} \ ; \\
& | & \textbf{if} \ (\ \textbf{expr} \ ) \ \mathit{stmt} \\
& | & \textbf{for} \ (\ \mathit{optexpr} \ ; \ \mathit{optexpr} \ ; \ \mathit{optexpr} \ ) \ \mathit{stmt} \\
& | & \textbf{other}
\end{array}
$$

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

Top-down Parsing

- For a match to occur, the nonterminal *stmt* 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.

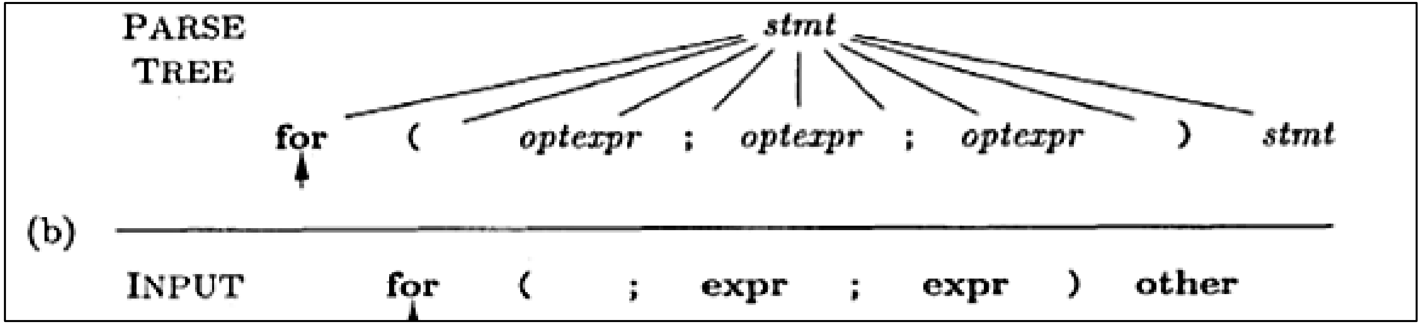
Input String: for (; expr ; expr) other



Top-down Parsing

- This expansion of the parse tree is shown in **(b)**.
- 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.

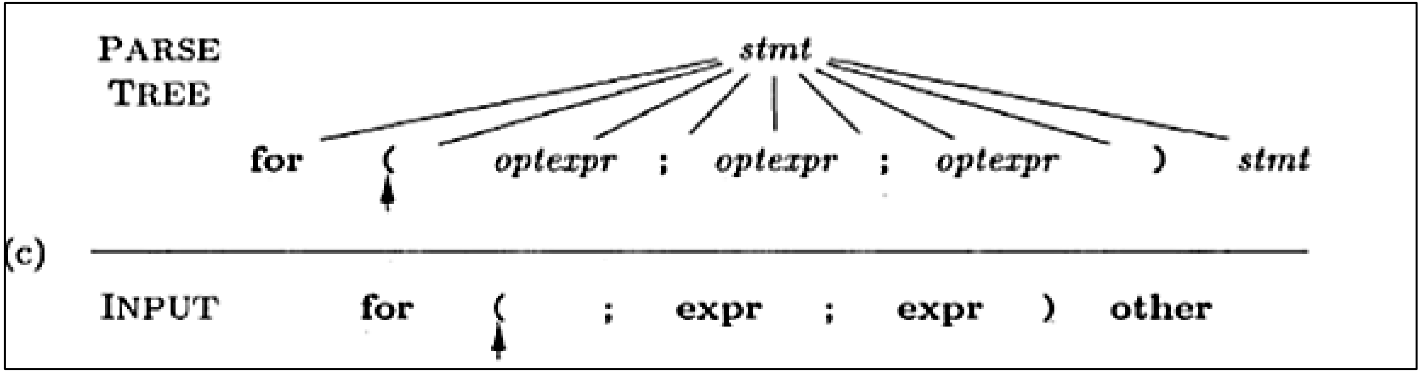
Input String: for (; expr ; expr) other



Top-down Parsing

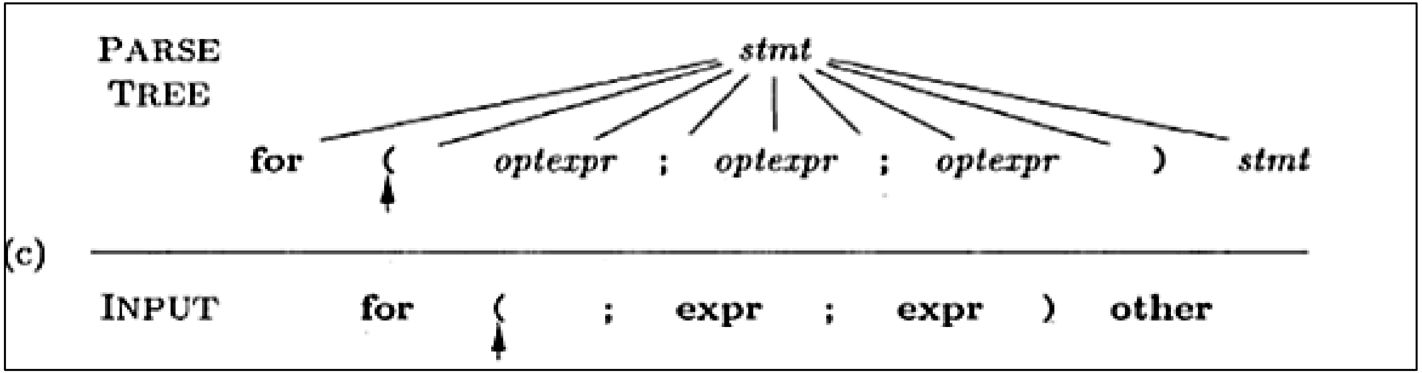
- When the node being considered in the parse tree is for a terminal, and the terminal matches the lookahead symbol, then we advance in both the parse tree and the input.
- The next terminal in the input becomes the new lookahead symbol, and the next child in the parse tree is considered.

Input String: for (; expr ; expr) other



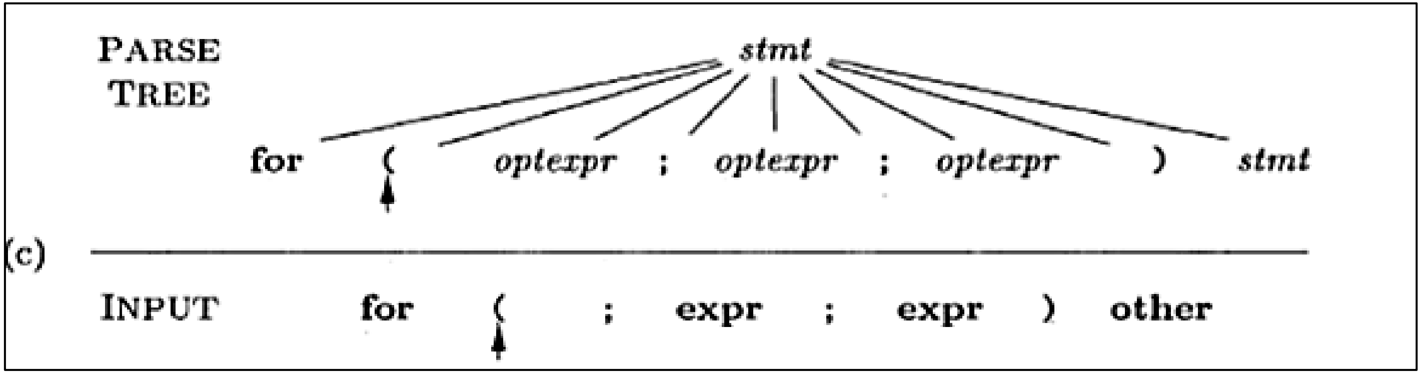
Top-down Parsing

- 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 **(**
- 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 **;**



Top-down Parsing

- At the nonterminal node labeled *optexpr*, we repeat the process of selecting a production for a nonterminal.
- Productions with ∈ as the body require special treatment.
- For the moment, we use them as a default when no other production can be used.
- With nonterminal *optexpr* and lookahead **;**, the ∈ -production is used, since **;** does not match the only other production for optexpr



Top-down Parsing

- 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.

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.

```
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

Predictive Parsing

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

Pseudocode for a predictive parser

Predictive Parsing

- 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");
        }
}
```

Predictive Parsing

- With the input for (; expr ; expr) other, lookahead is initially the first terminal for.
- Procedure *stmt* executes code corresponding to the production

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

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

Predictive Parsing

- 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 FIRST(α) 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 FIRST(α).

Predictive Parsing

- Typically, α will either begin with a terminal, which is therefore the only symbol in FIRST(α).
- Or α will begin with a nonterminal whose production bodies begin with terminals, in which case these terminals are the only members of FIRST(α).
- With respect to the grammar, the following are correct calculations of FIRST.

  - $\text{FIRST}(\mathit{stmt}) = \{\textbf{expr}, \textbf{if}, \textbf{for}, \textbf{other}\}$
  - $\text{FIRST}(\textbf{expr}\ ;) = \{\textbf{expr}\}$

Predictive Parsing

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

- $\text{FIRST}(stmt) = \{\textbf{expr}, \textbf{if}, \textbf{for}, \textbf{other}\}$
- $\text{FIRST}(\textbf{expr} \ ;) = \{\textbf{expr}\}$

Predictive Parsing

- 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 FIRST(α) and FIRST(β) to be disjoint.
- The lookahead symbol can then be used to decide which production to use.
- If the lookahead symbol is in FIRST(α), then α is used.
- Otherwise, if the lookahead symbol is in FIRST(β), then β is used.

When to Use ∈-Productions

- Our predictive parser uses an ∈-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 "**;**"
- At this point procedure *optexpr* is called, and the code in its body is executed.

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

When to Use ∈-Productions

- Nonterminal *optexpr* has two productions, with bodies **expr** and ∈.
- The lookahead symbol "**;**" does not match the terminal **expr**, so the production with body **expr** cannot apply.
- In fact, the procedure returns without changing the lookahead symbol or doing anything else.
- Doing nothing corresponds to applying an ∈ -production.

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

When to Use ∈-Productions

- More generally, consider a variant of the productions where *optexpr* generates an expression nonterminal instead of the terminal **expr**
- Thus, optexpr either generates an expression using nonterminal *expr* or it generates ∈.
- While parsing *optexpr*, if the lookahead symbol is not in FIRST(*expr*) then the ∈ -production is used.

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

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 is used if the lookahead symbol is in FIRST(α).
- 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 ∈ -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

2. The procedure then mimics the body of the chosen production.

- The symbols of the body are “executed” in turn, from the left.
- A nonterrninal 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.

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

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

- 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

$$expr \rightarrow expr + term$$

- Here 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 Elimination

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

$$A \to A\alpha \mid \beta$$

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

- For example, in

$$expr \to expr + term \mid term$$

nonterminal A = *expr*, string α = +*term*, and string β = *term*.

Left Recursion Elimination

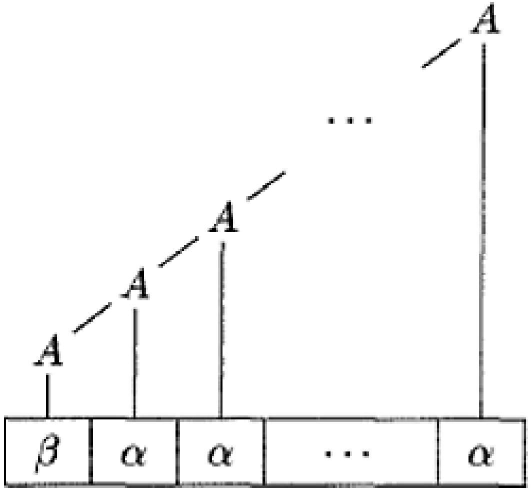
$$A \to A\alpha \mid \beta$$

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

Left Recursion Elimination

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

- Repeated application of this production builds up a sequence of α's to the right of A.
- When A is finally replaced by β, we have β followed by a sequence of zero or more α's.

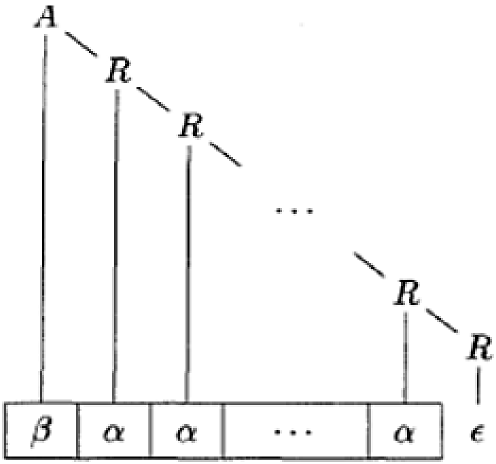


Left Recursion Elimination

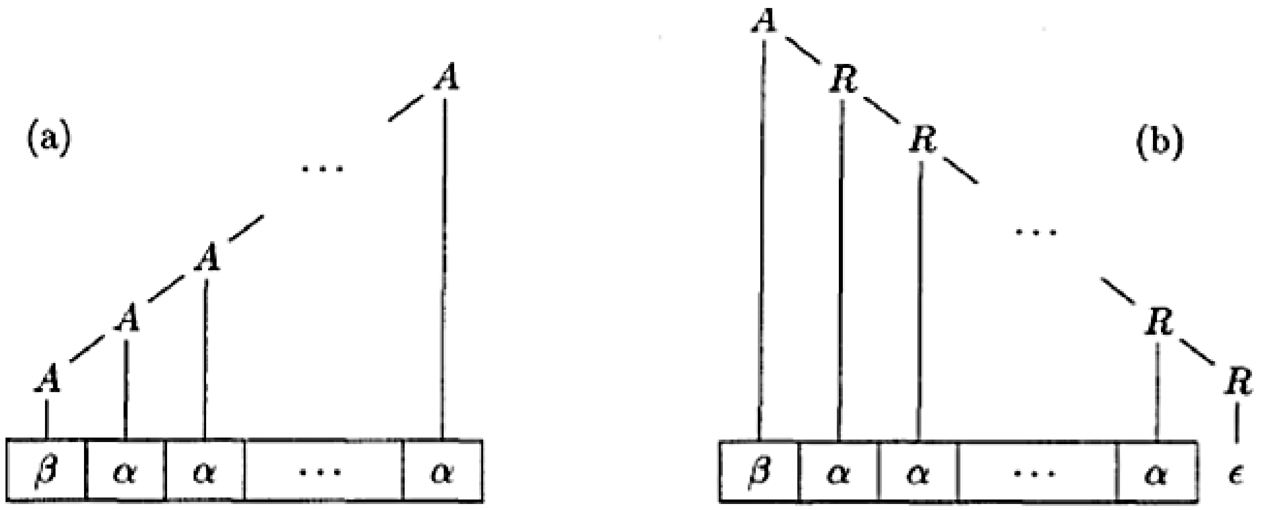
$$A \to A\alpha \mid \beta$$

- The same effect can be achieved, by rewriting the productions for *A* in the following manner, using a new nonterminal *R*:
- Right-recursive productions lead to trees that grow down towards the right.

$$\begin{aligned} A &\to \beta R \\ R &\to \alpha R \mid \epsilon \end{aligned}$$



Left Recursion Elimination



Left- and right-recursive ways of generating a string

Thank You