CS 4300: Compiler Theory

Chapter 2
A Simple Syntax-Directed Translator

Dr. Xuejun Liang

Outline

- This chapter is an introduction to the compiling techniques in Chapters 3 to 6 of the Dragon book
- It illustrates the techniques by developing a working Java program that translates representative programming language statements into three-address code
- The major topics are
  2. Syntax Definition
  3. Syntax-Directed Translation
  4. Parsing
  5. A Translator for Simple Expressions
  6. Lexical Analysis
  7. Symbol Tables
  8. Intermediate Code Generation

3. Syntax-Directed Translation

Syntax-Directed Definition

- Uses a CF grammar to specify the syntactic structure of the language
- AND associates a set of *attributes* with the terminals and nonterminals of the grammar
  - An attribute is any quantity associated with a programming construct
- AND associates with each production a set of *semantic rules* to compute values of attributes
- A parse tree is traversed and semantic rules applied: after the tree traversal(s) are completed, the attribute values on the nonterminals contain the translated form of the input

Synthesized and Inherited Attributes

- An attribute is said to be …
  - *synthesized* if its value at a parse-tree node is determined from the attribute values at the children of the node
  - *inherited* if its value at a parse-tree node is determined by the parent (by enforcing the parent's semantic rules)

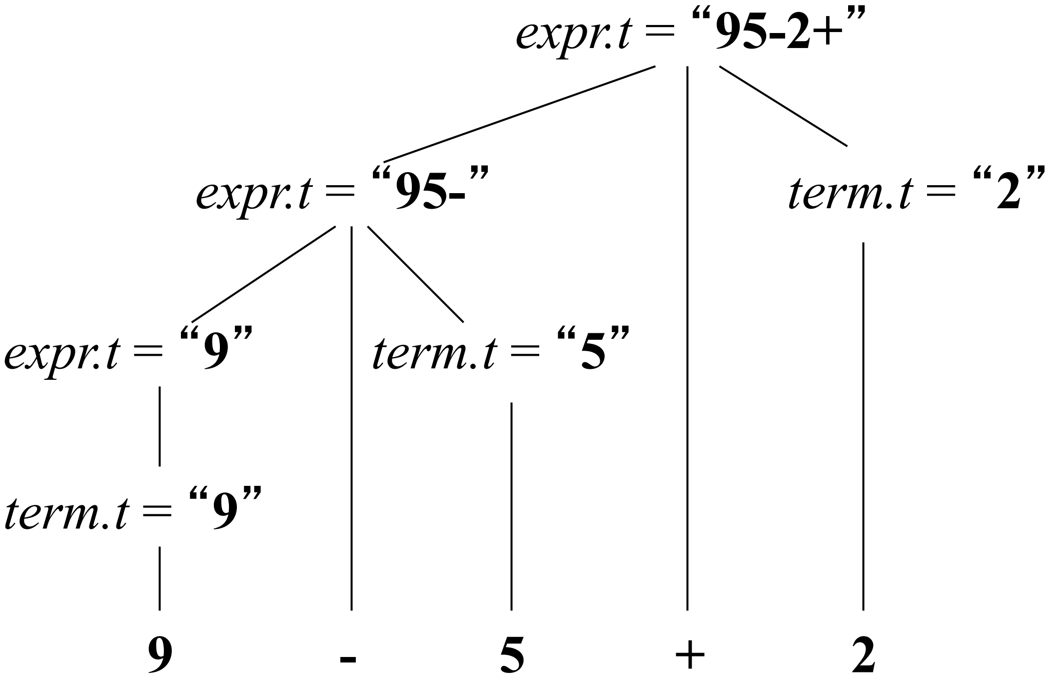
Example Attribute Grammar

String concat operator

Production	Semantic Rule
$expr \rightarrow expr_1 + term$	$expr.t := expr_1.t \,\|\, term.t \,\|\, \text{“+”}$
$expr \rightarrow expr_1 - term$	$expr.t := expr_1.t \,\|\, term.t \,\|\, \text{“-”}$
$expr \rightarrow term$	$expr.t := term.t$
$term \rightarrow \mathbf{0}$	$term.t := \text{“0”}$
$term \rightarrow \mathbf{1}$	$term.t := \text{“1”}$
…	…
$term \rightarrow \mathbf{9}$	$term.t := \text{“9”}$

Syntax-directed definition for infix to postfix translation

Example Annotated Parse Tree



$expr.t := expr_1.t \,\|\, term.t \,\|$ “+”
$expr.t := expr_1.t \,\|\, term.t \,\|$ “-”
$expr.t := term.t$
$term.t :=$ “0”
$term.t :=$ “1”
…
$term.t :=$ “9”

Attribute values at nodes in a parse tree

Depth-First Traversals

```
procedure visit(n : node);
begin
    for each child c of n, from left to right do
        visit(c);
    evaluate semantic rules at node n
end
```

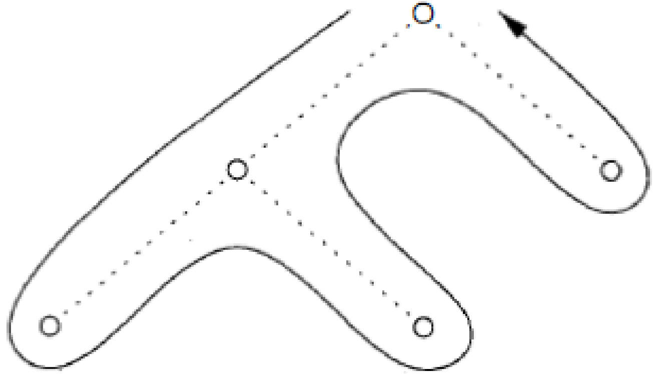
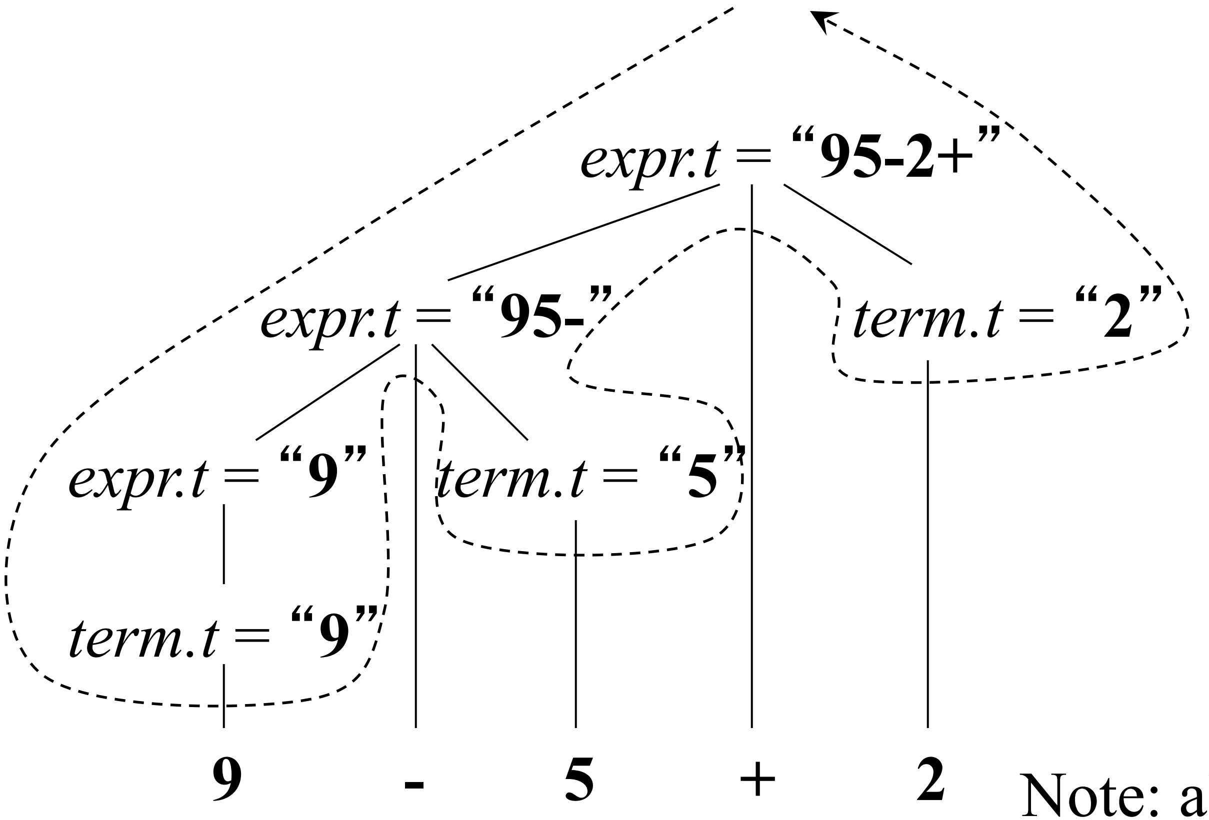


Figure 2.12: Example of a depth-first traversal of a tree

Depth-First Traversals (Example)



expr.t = **“95-2+”**

expr.t = **“95-”**

term.t = **“2”**

expr.t = **“9”**

term.t = **“5”**

term.t = **“9”**

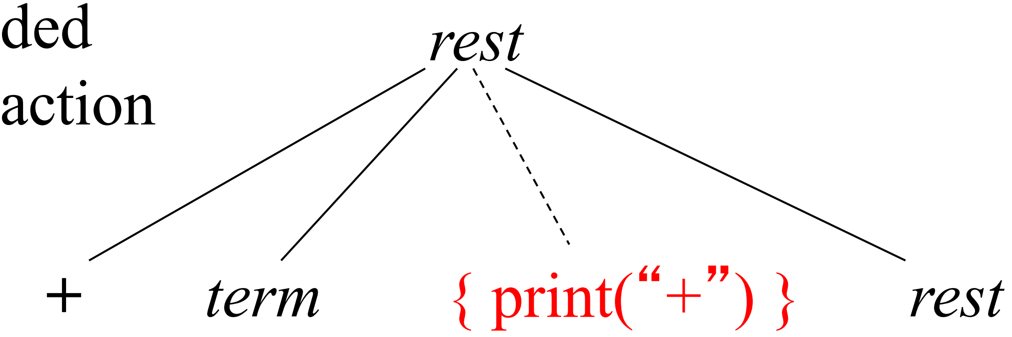
9 **-** **5** + **2**

Note: all attributes are of the synthesized type

Translation Schemes

- A translation scheme is a CF grammar embedded with semantic actions by attaching program fragments to productions in the grammar

$$rest \to + \; term \; \underbrace{\{\ \text{print}(\text{“+”})\ \}}_{\text{Embedded semantic action}} \; rest$$



An extra leaf is constructed for a semantic action

Example Translation Scheme

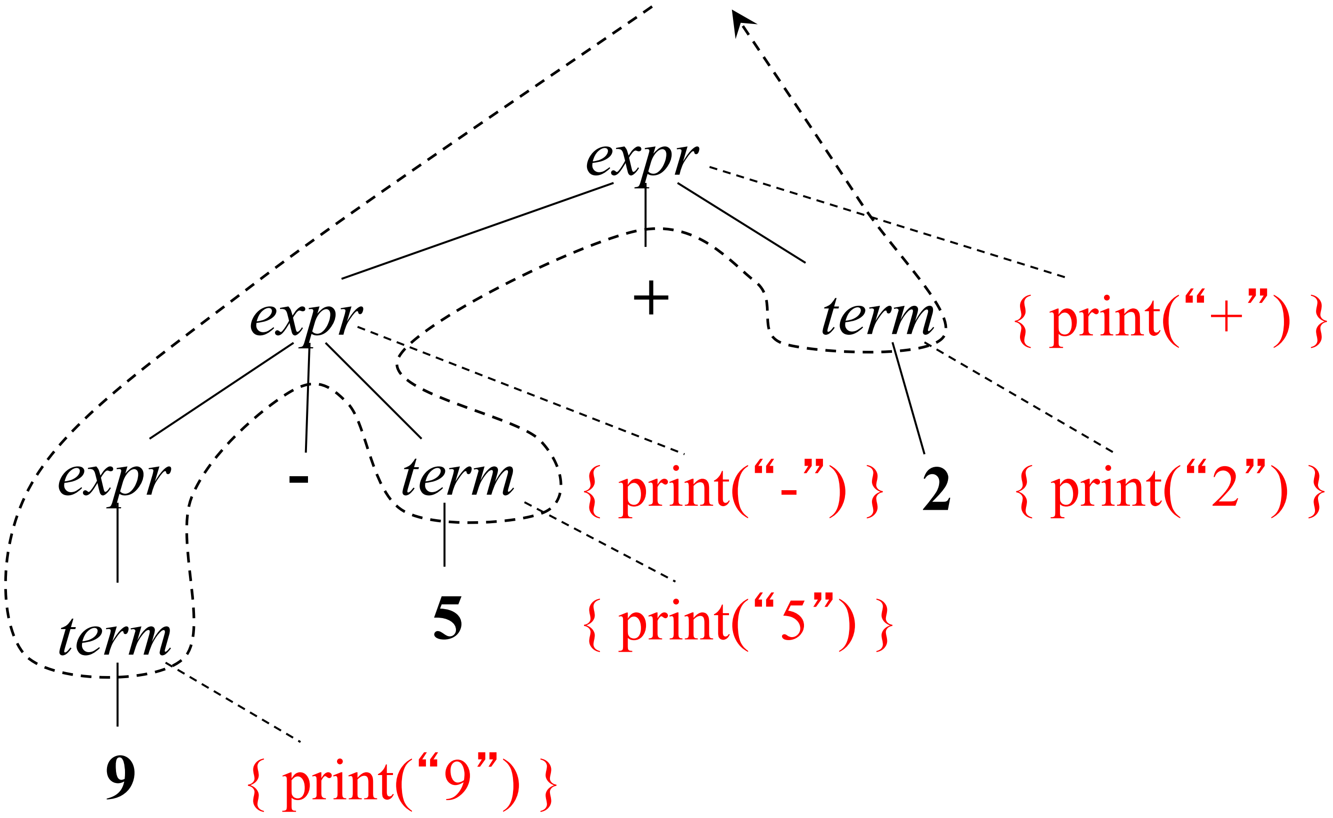
Grammar

$\mathit{expr} \rightarrow \mathit{expr} + \mathit{term}$ { print(“+”) }
$\mathit{expr} \rightarrow \mathit{expr} - \mathit{term}$ { print(“-”) }
$\mathit{expr} \rightarrow \mathit{term}$
$\mathit{term} \rightarrow \mathbf{0}$ { print(“0”) }
$\mathit{term} \rightarrow \mathbf{1}$ { print(“1”) }
… …
$\mathit{term} \rightarrow \mathbf{9}$ { print(“9”) }

Actions for translating infix into postfix notation

Example Translation Scheme

(Annotated) Parse Tree



Translates **9-5+2** into postfix **95-2+**

4. Parsing

- Parsing = *process of determining if a string of tokens can be generated by a grammar*
- For any CF grammar there is a parser that takes at most $O(n^3)$ time to parse a string of n tokens
- Linear algorithms suffice for parsing programming language source code
- *Top-down parsing* “constructs” a parse tree from root to leaves
- *Bottom-up parsing* “constructs” a parse tree from leaves to root

Top-Down Parsing

- The top-down construction of a parse tree is done by starting with the root, labeled with the starting nonterminal , 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 Example

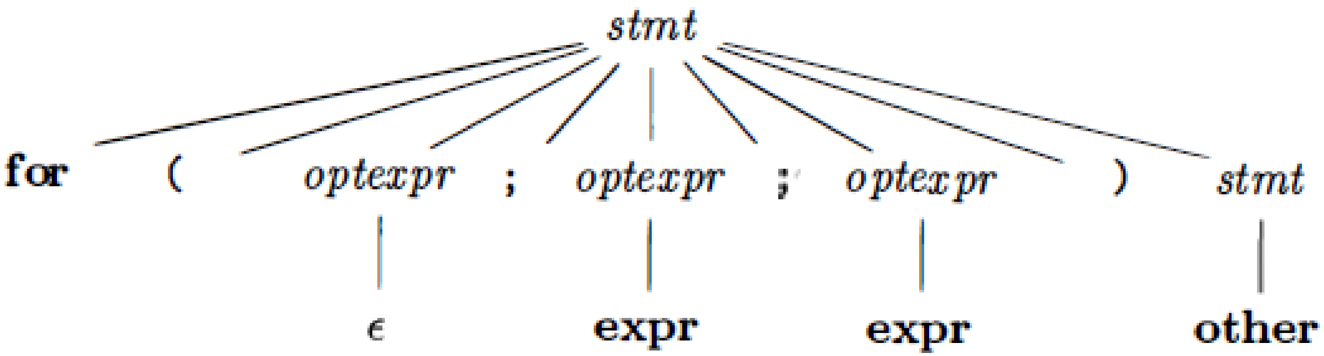
Grammar

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

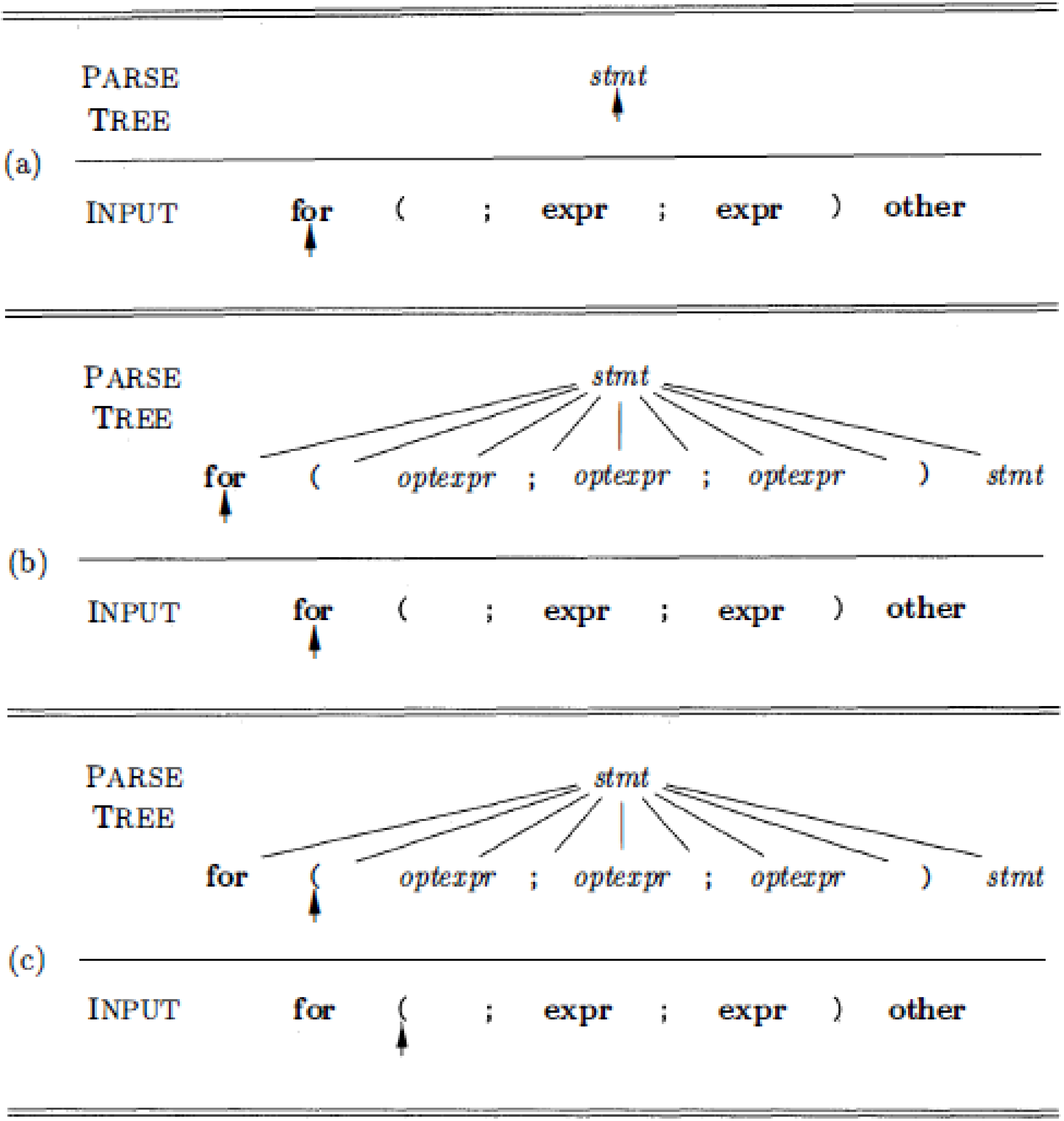
$$
\begin{array}{lcl}
\mathit{optexpr} & \rightarrow & \epsilon\\
 & | & \textbf{expr}
\end{array}
$$

Input string

for (; expr ; expr) other



A parse tree according to the grammar


PARSE TREE
stmt
(a)
INPUT
for (; expr ; expr) other
PARSE TREE
stmt
for (optexpr ; optexpr ; optexpr) stmt
(b)
INPUT
for (; expr ; expr) other
PARSE TREE
stmt
for (optexpr ; optexpr ; optexpr) stmt
(c)
INPUT
for (; expr ; expr) other

Predictive Parsing

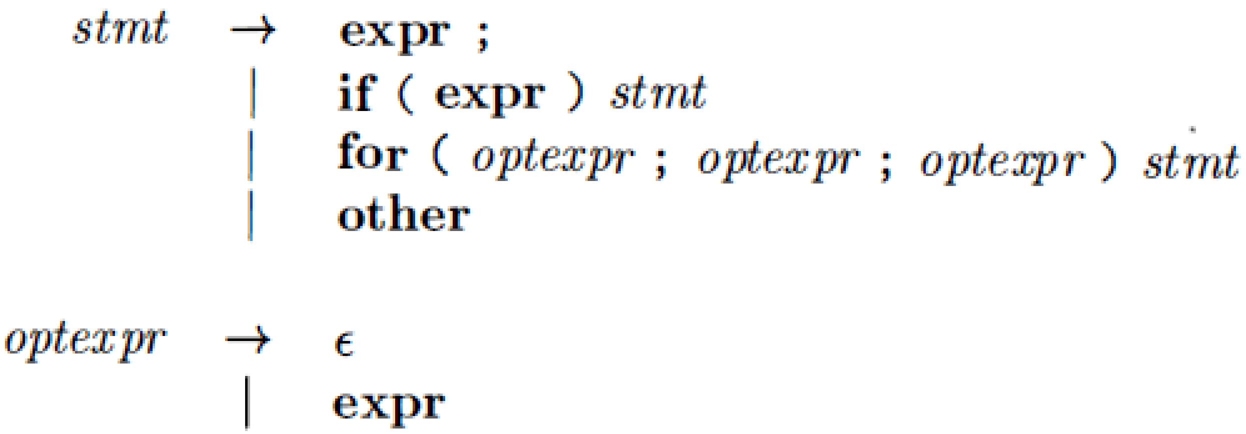
- *Recursive descent parsing* is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input.
  - Each nonterminal has one (recursive) procedure that is responsible for parsing the nonterminal's syntactic category of input tokens
  - When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input look-ahead information
- *Predictive parsing* is a special form of recursive descent parsing where we use one lookahead token to unambiguously determine the parse operations

```
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");
}
```

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

optexpr  →  ϵ
         |  expr
```

FIRST Set

FIRST(α) is the set of terminals that appear as the first symbols of one or more strings generated from α



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

FIRST(*stmt*) = { **expr**, **if**, **for**, **other** }

FIRST(**expr**) = {**expr**}

FIRST(**for** (*optexpr* ; *optexpr ; optexpr) stmt*) = {**for**}

How to use FIRST Set

We use FIRST to write a predictive parser as follows

$$
\begin{aligned}
\mathit{expr} &\rightarrow \mathit{term}\ \mathit{rest} \\
\mathit{rest} &\rightarrow +\ \mathit{term}\ \mathit{rest} \\
&\mid -\ \mathit{term}\ \mathit{rest} \\
&\mid \varepsilon
\end{aligned}
$$

```
procedure rest();
begin
    if lookahead in FIRST(+ term rest) then
        match( '+' ); term(); rest()
    else if lookahead in FIRST(- term rest) then
        match( '-' ); term(); rest()
    else return
end;
```

When a nonterminal *A* has two (or more) productions as in

$$
\begin{aligned}
A &\rightarrow \alpha \\
&\mid \beta
\end{aligned}
$$

Then FIRST (α) and FIRST(β) must be disjoint for predictive parsing to work

Left Factoring

When more than one production for nonterminal *A* starts with the same symbols, the FIRST sets are not disjoint

stmt → **if** *expr* **then** *stmt* **endif**
| **if** expr **then** *stmt* **else** *stmt* **endif**

We can use *left factoring* to fix the problem

stmt → **if** expr **then** *stmt opt_else*
opt_else → **else** *stmt* **endif**
| **endif**

Left Recursion

When a production for nonterminal *A* starts with a self reference then a predictive parser loops forever

$$
\begin{aligned}
A &\rightarrow A\ \alpha \\
&\mid \beta \\
&\mid \gamma
\end{aligned}
$$

We can eliminate *left recursive productions* by systematically rewriting the grammar using *right recursive productions*

$$
\begin{aligned}
A &\rightarrow \beta\ R \\
&\mid \gamma\ R \\
R &\rightarrow \alpha\ R \\
&\mid \varepsilon
\end{aligned}
$$