CSCI312 Principles of Programming Languages

Chapter 2

Syntax

Xu Liu

Review

Principles of PL

syntax, naming, types, semantics

Paradigms of PL design

imperative, OO, functional, logic

What makes a successful PL

simplicity and readability
clarity about binding
reliability
support
abstraction
orthogonality
efficient implementation

Contents

Thinking about Syntax

The *syntax* of a programming language is a precise description of all its grammatically correct programs.

Precise syntax was first used with Algol 60, and has been used ever since.

Three levels:

- *Lexical syntax*
- *Concrete syntax*
- *Abstract syntax*

Levels of Syntax

Lexical syntax = all the basic symbols of the language (names, values, operators, etc.)

Concrete syntax = rules for writing expressions, statements and programs.

Abstract syntax = internal representation of the program, favoring content over form. E.g.,

- *C: if (expr) ... discard ( )*
- *Ada: if (expr) then discard then*

2.1 Grammars

A *metalanguage* is a language used to define other languages.

A *grammar* is a metalanguage used to define the syntax of a language.

Our interest: using grammars to define the syntax of a programming language.

2.1.1 Backus-Naur Form (BNF)

- Stylized version of a context-free grammar (cf. Chomsky hierarchy)
- Sometimes called Backus Normal Form
- First used to define syntax of Algol 60
- Now used to define syntax of most major languages

BNF Grammar

Set of *productions*: P

terminal symbols: T

nonterminal symbols: N

start symbol: $S \in N$

A *production* has the form

$$A \rightarrow \omega$$

where $A \in N$ and $\omega \in (N \cup T)^*$

Example: Binary Digits

Consider the grammar:

$$binaryDigit \rightarrow 0$$
$$binaryDigit \rightarrow 1$$

or equivalently:

$$binaryDigit \rightarrow 0 \mid 1$$

Here, | is a metacharacter that separates alternatives.

2.1.2 Derivations

Consider the grammar:

$$Integer \rightarrow Digit \mid Integer\ Digit$$

$$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

We can *derive* any unsigned integer, like 352, from this grammar.

Derivation of 352 as an *Integer*

A 6-step process, starting with:

Integer

Derivation of 352 (step 1)

Use a grammar rule to enable each step:

$$Integer \Rightarrow Integer\ Digit$$

Derivation of 352 (steps 1-2)

Replace a nonterminal by a right-hand side of one of its rules:

$$
\begin{aligned}
Integer &\Rightarrow Integer\ Digit \\
&\Rightarrow Integer\ 2
\end{aligned}
$$

Derivation of 352 (steps 1-3)

Each step follows from the one before it.

$$\begin{aligned} Integer &\Rightarrow Integer\ Digit \\ &\Rightarrow Integer\ 2 \\ &\Rightarrow Integer\ Digit\ 2 \end{aligned}$$

Derivation of 352 (steps 1-4)

$$
\begin{aligned}
\mathit{Integer} &\Rightarrow \mathit{Integer}\ \mathit{Digit} \\
&\Rightarrow \mathit{Integer}\ 2 \\
&\Rightarrow \mathit{Integer}\ \mathit{Digit}\ 2 \\
&\Rightarrow \mathit{Integer}\ 5\ 2
\end{aligned}
$$

Derivation of 352 (steps 1-5)

$$
\begin{aligned}
Integer &\Rightarrow Integer\ Digit \\
&\Rightarrow Integer\ 2 \\
&\Rightarrow Integer\ Digit\ 2 \\
&\Rightarrow Integer\ 5\ 2 \\
&\Rightarrow Digit\ 5\ 2
\end{aligned}
$$

Derivation of 352 (steps 1-6)

You know you're finished when there are only terminal symbols remaining.

$$
\begin{aligned}
\mathit{Integer} &\Rightarrow \mathit{Integer}\ \mathit{Digit} \\
&\Rightarrow \mathit{Integer}\ 2 \\
&\Rightarrow \mathit{Integer}\ \mathit{Digit}\ 2 \\
&\Rightarrow \mathit{Integer}\ 5\ 2 \\
&\Rightarrow \mathit{Digit}\ 5\ 2 \\
&\Rightarrow 3\ 5\ 2
\end{aligned}
$$

A Different Derivation of 352

$$
\begin{aligned}
\textit{Integer} &\Rightarrow \textit{Integer Digit} \\
&\Rightarrow \textit{Integer Digit Digit} \\
&\Rightarrow \textit{Digit Digit Digit} \\
&\Rightarrow 3\ \textit{Digit Digit} \\
&\Rightarrow 3\ 5\ \textit{Digit} \\
&\Rightarrow 3\ 5\ 2
\end{aligned}
$$

This is called a *leftmost derivation*, since at each step the leftmost nonterminal is replaced.

(The first one was a *rightmost derivation*.)

Notation for Derivations

$Integer \Rightarrow^* 352$

Means that 352 can be derived in a finite number of steps using the grammar for *Integer*.

$352 \in L(G)$

Means that 352 is a member of the language defined by grammar *G*.

$L(G) = \{ \omega \in T^* \mid Integer \Rightarrow^* \omega \}$

Means that the language defined by grammar *G* is the set of all symbol strings ω that can be derived as an *Integer*.

Problem in this Grammar

Consider the grammar:

$$Integer \rightarrow Digit \mid Integer\ Digit$$
$$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

$$Integer \rightarrow Digit \mid SDigit\ AInteger$$
$$AInteger \rightarrow Digit \mid AInteger\ Digit$$
$$Digit \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$SDigit \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

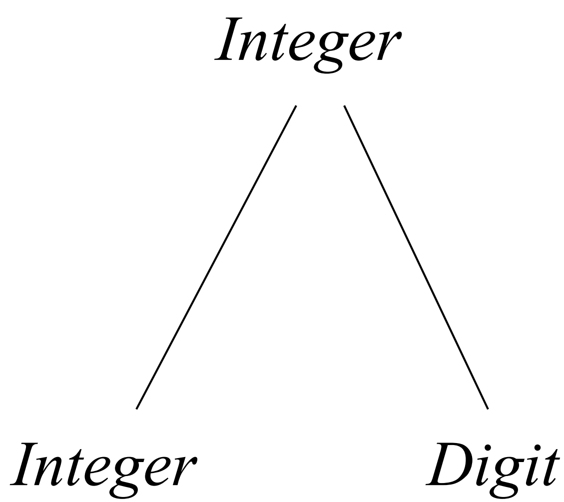
We can derive 031, 0003, 0000

2.1.3 Parse Trees

A *parse tree* is a graphical representation of a derivation.

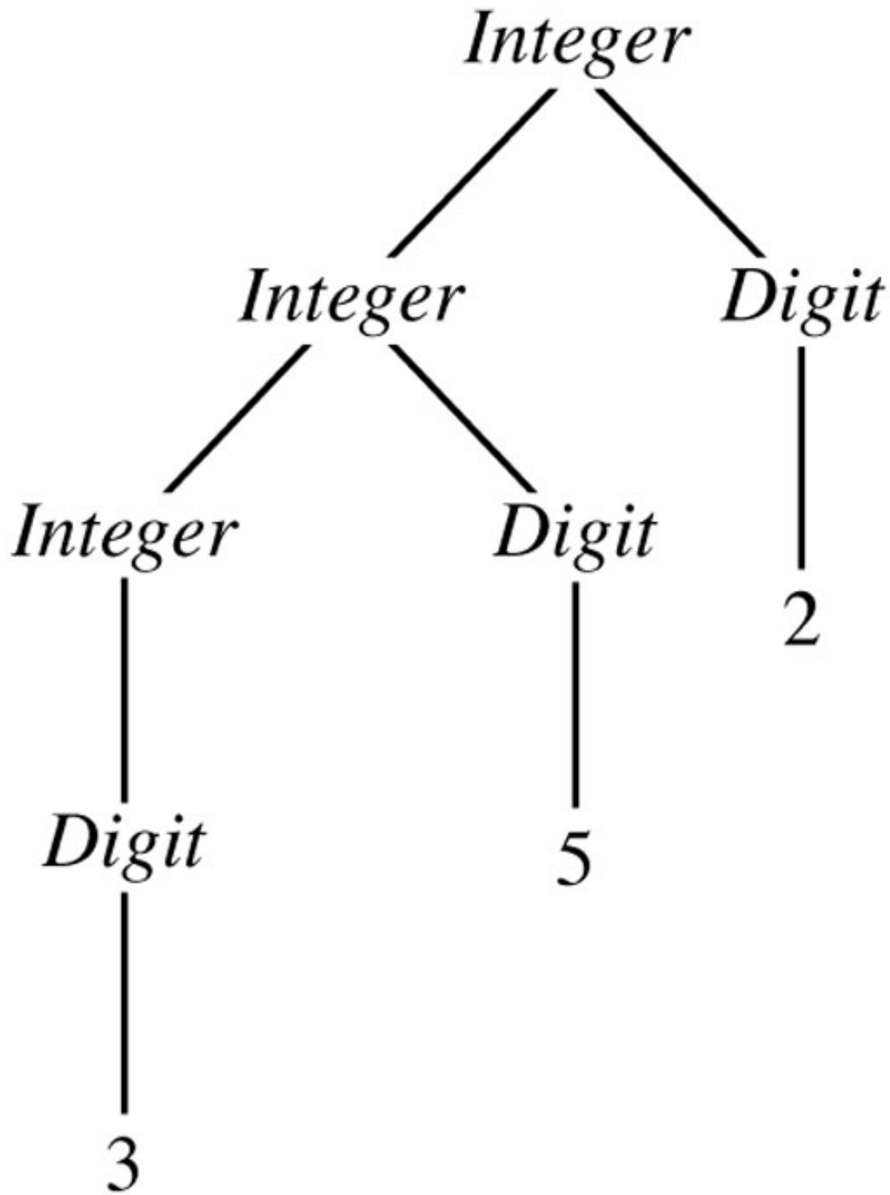
- *Each internal node of the tree corresponds to a step in the derivation.*
- *The children of a node represents a right-hand side of a production.*
- *Each leaf node represents a symbol of the derived string, reading from left to right.*

E.g., The step *Integer* $\Rightarrow$ *Integer Digit* appears in the parse tree as:



Parse Tree for 352 as an *Integer*

Figure 2.1



Arithmetic Expression Grammar

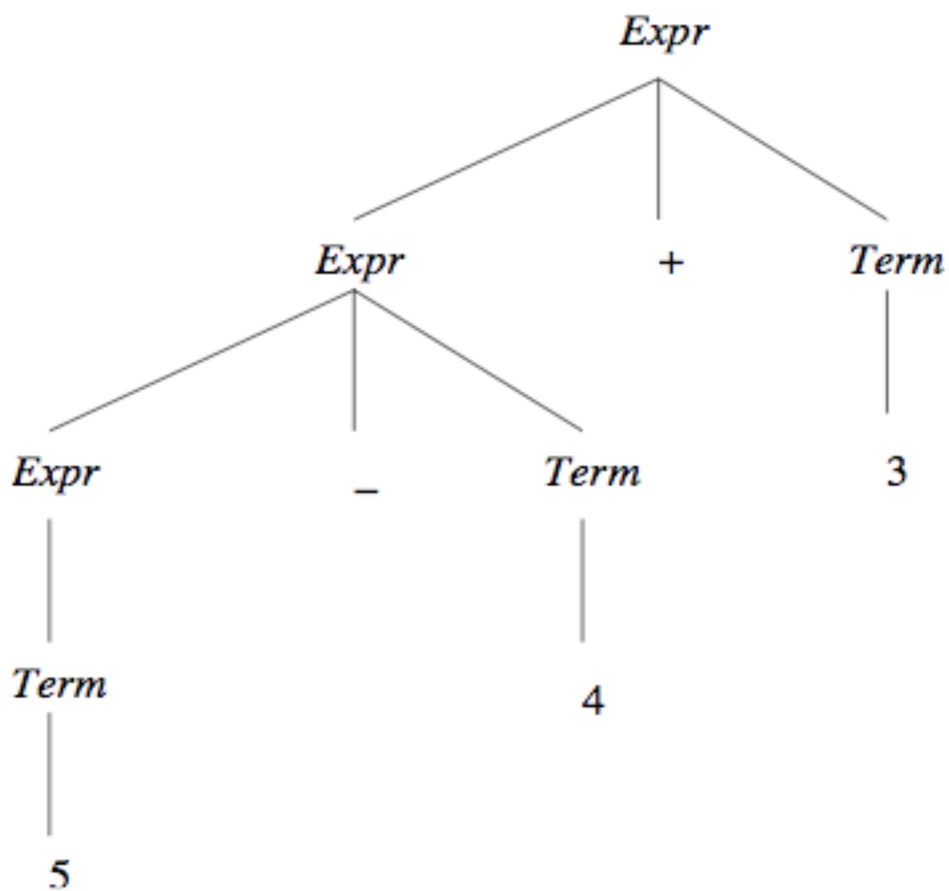
The following grammar defines the language of arithmetic expressions with 1-digit integers, addition, and subtraction.

$$Expr \rightarrow Expr + Term \mid Expr - Term \mid Term$$

$$Term \rightarrow 0 \mid \ldots \mid 9 \mid (\; Expr \;)$$

Parse of the String 5-4+3

Figure 2.2



2.1.4 Associativity and Precedence

A grammar can be used to define associativity and precedence among the operators in an expression.

E.g., + and - are left-associative operators in mathematics;

** and / have higher precedence than + and - .*

Consider the more interesting grammar G_1:

Expr -> Expr + Term | Expr – Term | Term

*Term -> Term * Factor | Term / Factor |*

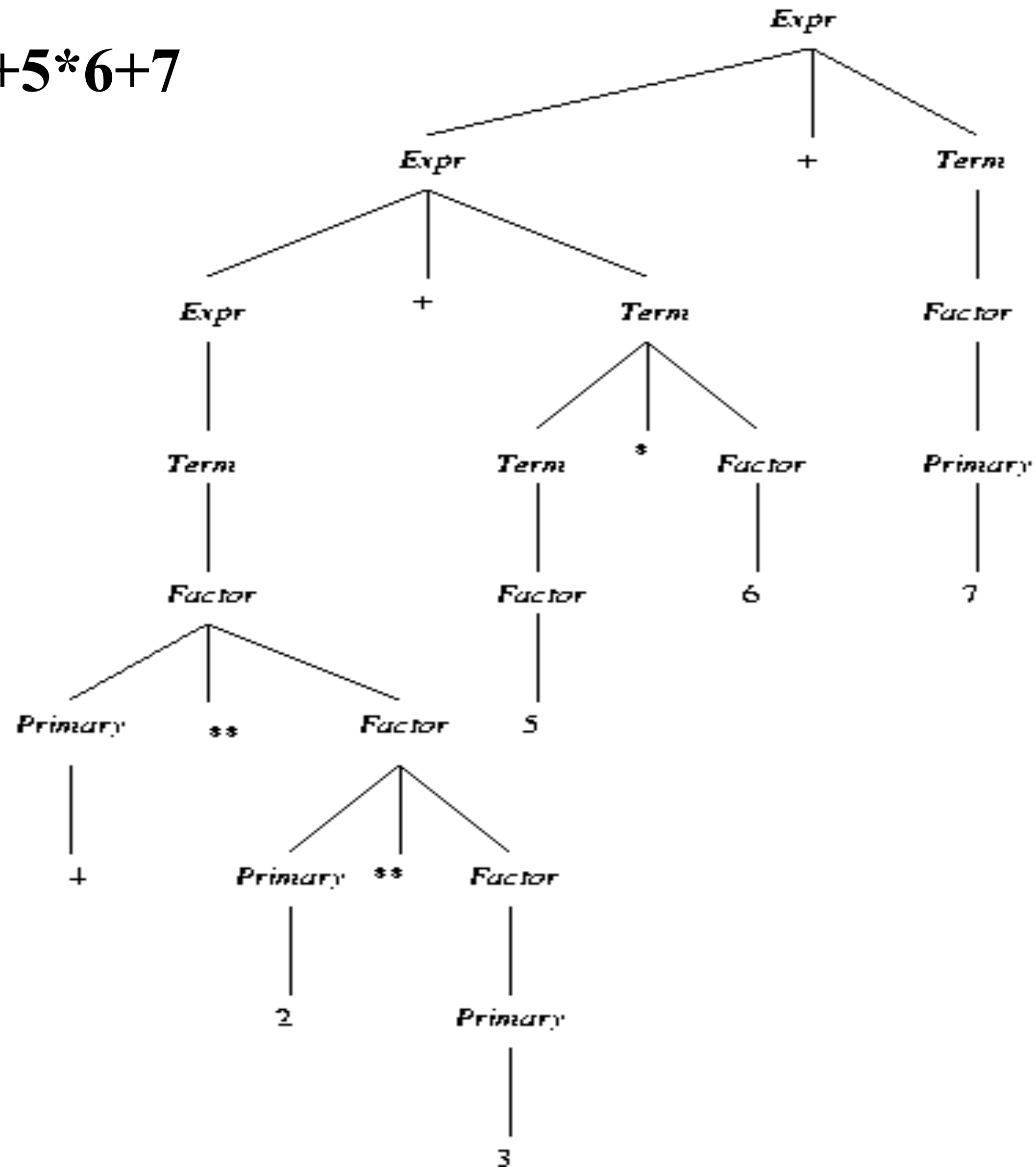
Term % Factor | Factor

*Factor -> Primary ** Factor | Primary*

Primary -> 0 | ... | 9 | (Expr)

Parse of 42**3+5*6+7 for Grammar G_1**

Figure 2.3



Associativity and Precedence for Grammar G_1
Table 2.1

Precedence	Associativity	Operators
3	right	**
2	left	* / %
1	left	+ -

Note: These relationships are shown by the structure of the parse tree: highest precedence at the bottom, and left-associativity on the left at each level.

2.1.5 Ambiguous Grammars

A grammar is *ambiguous* if one of its strings has two or more diffferent parse trees.

E.g., Grammar G_1 *above is unambiguous.*

C, C++, and Java have a large number of

- *operators and*
- *precedence levels*

Instead of using a large grammar, we can:

- *Write a smaller ambiguous grammar, and*
- *Give separate precedence and associativity (e.g., Table 2.1)*

An Ambiguous Expression Grammar G_2

Expr -> *Expr* *Op* *Expr* | (*Expr*) | *Integer*

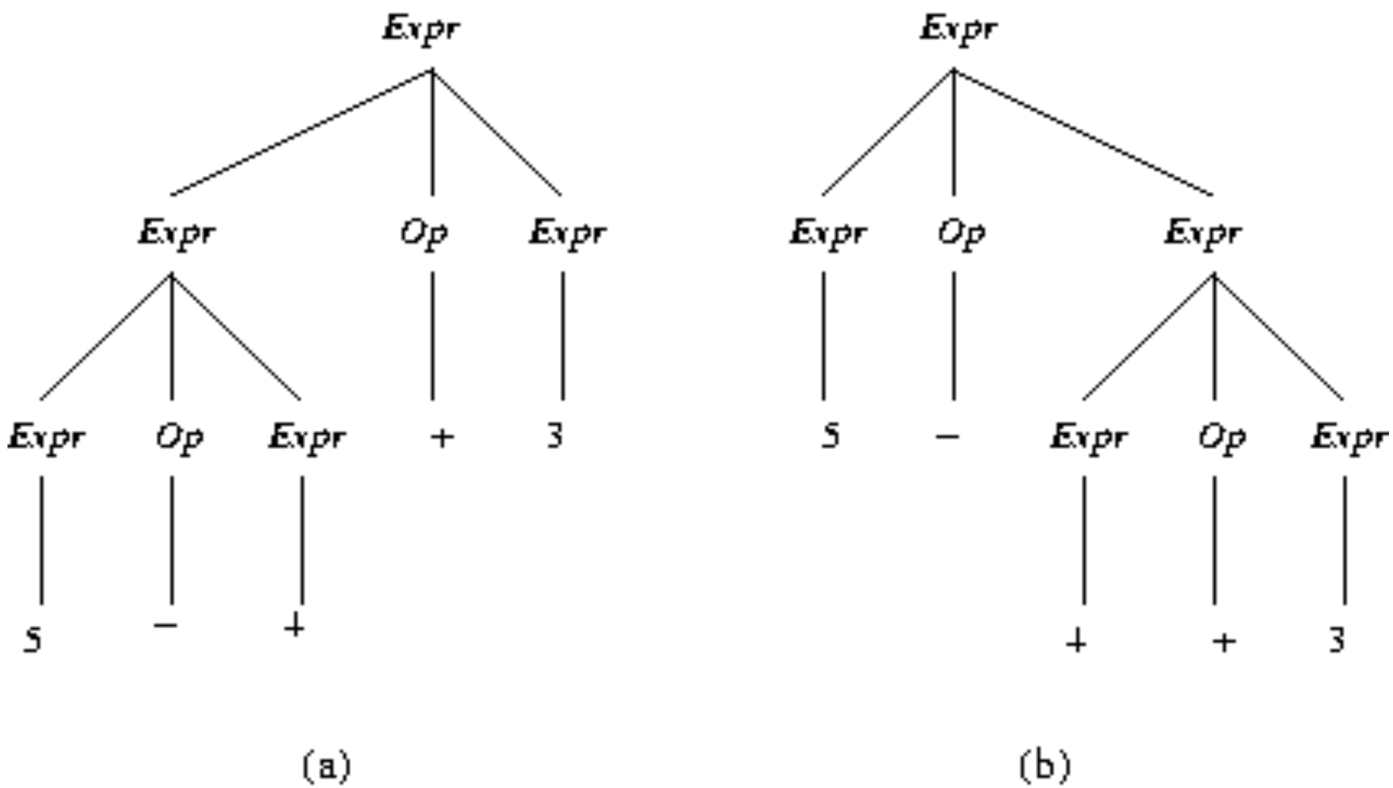
Op -> + | - | * | / | % | **

Notes:

- G_2 is equivalent to G_1. I.e., its language is the same.
- G_2 has fewer productions and nonterminals than G_1.
- However, G_2 is ambiguous.

Ambiguous Parse of 5-4+3 Using Grammar G_2

Figure 2.4



The Dangling Else

IfStatement -> if (*Expression*) *Statement* |

if (*Expression*) *Statement* else *Statement*

Statement -> *Assignment* | *IfStatement* | *Block*

Block -> { *Statements* }

Statements -> *Statements* *Statement* | *Statement*

Example

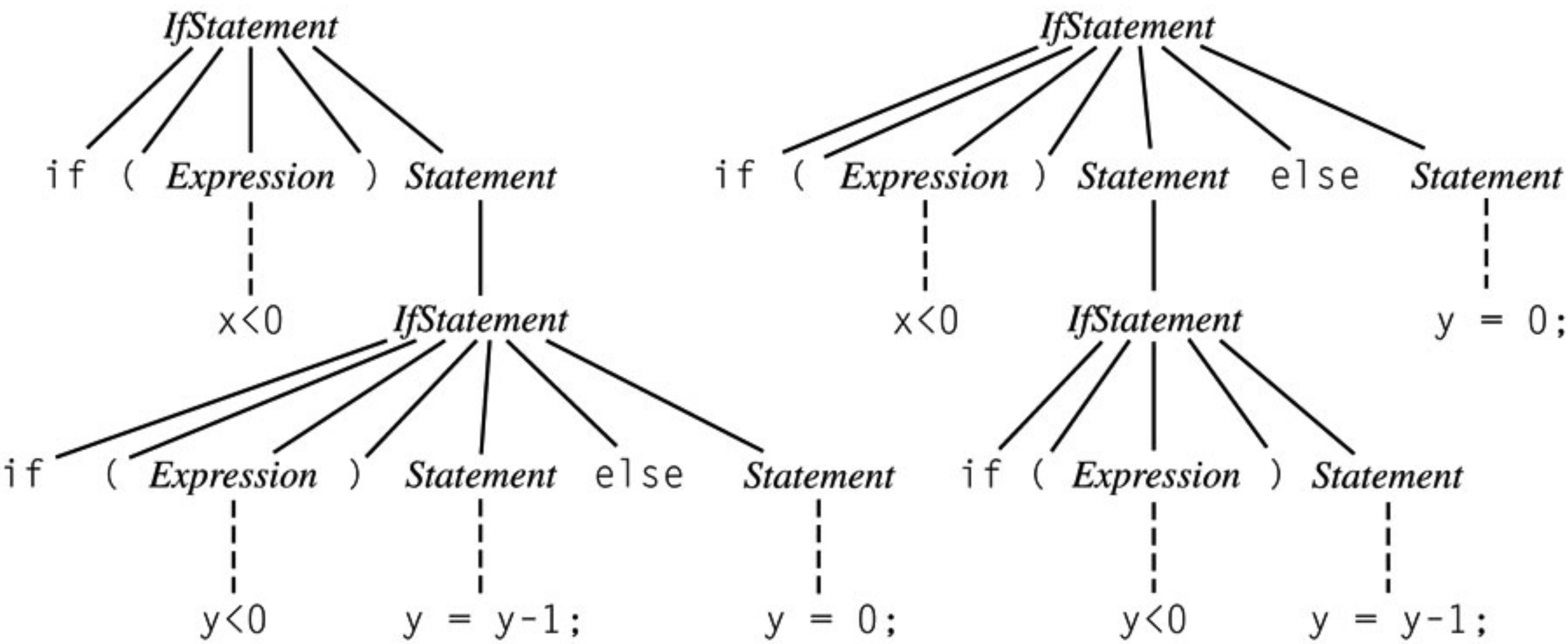
With which ‘if’ does the following ‘else’ associate

```
if (x < 0)
        if (y < 0)  y = y  - 1;
        else y = 0;
```

Answer: *either one!*

The *Dangling Else* Ambiguity

Figure 2.5



Solving the dangling else ambiguity

1. Algol 60, C, C++: associate each else with closest if; use {} or begin…end to override.
2. Algol 68, Modula, Ada: use explicit delimiter to end every conditional (e.g., if…fi)
3. Java: rewrite the grammar to limit what can appear in a conditional:

IfThenStatement -> if (*Expression*) *Statement*

IfThenElseStatement -> if (*Expression*) *StatementNoShortIf*
else *Statement*

The category *StatementNoShortIf* includes all except *IfThenStatement*.

2.2 Extended BNF (EBNF)

BNF:

- *recursion for iteration*
- *nonterminals for grouping*

EBNF: additional metacharacters

- **{ }** for a series of zero or more
- **( )** for a list, must pick one
- **[ ]** for an optional list; pick none or one

EBNF Examples

Expression is a list of one or more *Terms* separated by operators + and -

Expression -> *Term* { (+ | -) *Term* }

IfStatement -> if (*Expression*) *Statement* [else *Statement*]

C-style EBNF lists alternatives vertically and uses $_{opt}$ *to signify optional parts. E.g.,*

IfStatement:

if (*Expression*) *Statement* $ElsePart_{opt}$

ElsePart:

else *Statement*

EBNF to BNF

We can always rewrite an EBNF grammar as a BNF grammar. E.g.,

$$A \text{ -> } x \; \{ \; y \; \} \; z$$

can be rewritten:

$$A \text{ -> } x \; A' \; z$$

$$A' \text{ -> } \mid \; y \; A'$$

(Rewriting EBNF rules with ( ), [ ] is left as an exercise.)

While EBNF is no more powerful than BNF, its rules are often simpler and clearer.