Programming Paradigms: An Introduction

Summer Semester 2023

Dr. Abhishek Tiwari, Prof. Dr. Christian Hammer

UNIVERSITY OF PASSAU
Faculty of Computer Science and Mathematics

What would be the o/p?

```c
#include <stdio.h>

int main(){
    int buffer[4] = {0,1,2,3};
    int *ptr = buffer;
    printf("%d\n", buffer[1]);
    printf("%d\n", *(ptr+2));
}
```

```c
int buffer[4] = {0,1,2,3};
int *ptr;

ptr = buffer;

buffer = ptr;
```

What does it mean?

```c
#include <stdio.h>

int main(){
  char *name = "hello world";
  printf("%c\n", *&*&*name);
}
```

What would be the o/p?

- 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

- Lexical Syntax — all 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
    - C: *if(expr) … discard()*
    - Ada: *if(expr) then discard then*

- Similar to the grammar used in natural language
- Defines the structure of a language
  - Michael goes to the park, *vs*
  - Goes the to Michel park, *vs*
  - The park goes to Michael
  - Similarly, in C — int x = 5; vs 5 int = x;
- Enables the users of the language to communicate clearly

- A metalanguage — A language used to define other languages
- Grammar — a metalanguage used to define the syntax of a language
- Can be formally defined as — $< N, T, S, P >$
  - N denotes nonterminal symbols
  - T denotes terminal symbols
  - S denotes the start symbol $S \in N$
  - P denotes a set of production rules for the Grammar

- These are not part of the language — used to represent the structure of a language
- For a natural language — noun, pronoun, adverb, …
- For the programming languages — statement, loop, condition, numbers, …

- Tokens of the programming languages
    - keywords, identifiers, operators, e.g., while, for, …

- Rules to create a grammar
- Defines nonterminal as a series of terminals or nonterminal
  - e.g., $word \rightarrow letter \,|\, word$

- Multiple approaches to express Grammar
- Backus-Naur form (BNF) — Context-free grammar
- Extended BNF
- Augmented BNF
- Regular grammar
- …

Backus-Naur Form (BNF)

- Stylized version of a context-free grammar
- Sometimes called Backus Normal form
- First used to define syntax of Algol 60
- Defines syntax of major programming languages

- Terminals are written as is, i.e., int, if, for, …
- Nonterminals are denoted in angled bracket, i.e., $< Numbers >$
- Production rules are written as:
    - <nonterminals> ::= <terminals><nonterminals> | <terminals><nonterminals> …
    - | represents or
    - e.g., <Name> ::=<suffix> <Last Name>”,”<First Name> |<suffix> <First Name>”,”<Last Name>

Example

- Binary digits:
  - <binaryDigit> ::= 0 | 1
- Defining integers
  - <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
  - <integer> ::= <digit> | <integer><digit>
- Postal Address[1]:

```
<postal-address> ::= <name-part> <street-address> <zip-part>

      <name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL> | <personal-part> <name-part>

  <personal-part> ::= <initial> "." | <first-name>

 <street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

       <zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""
    <opt-apt-num> ::= <apt-num> | ""
```

1. https://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form

Example: A mini domain specific language

$$
\begin{array}{rcl}
s \in Stmt & ::= & D \mid v = e \mid \textbf{allocate}(p, c) \mid \textbf{free}\ (p) \mid \\
& & \textbf{arraystore}(arr, i, v) \mid \textbf{arrayload}(arr, i, v) \mid \textbf{input}(\mathbf{x}) \mid \\
& & if\ (e)\ \{\overrightarrow{s_1}\}\ else\ \{\overrightarrow{s_2}\} \mid while(e)\ \{\overrightarrow{s}\} \mid v_n = func(\overrightarrow{v}) \mid \\
& & s_1; s_2 \mid skip \\
e \in Exp & ::= & v \mid c \mid v_1 \odot v_2 \mid p \pm c \mid *p \\
D \in Decl & ::= & T\,var \mid T\ arr[n] \mid T\ *p
\end{array}
$$

$v \in variable,\ c \in Constants,\ arr \in Arrays,\ T \in Type$

- Consider the grammar for Integer
    - <digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
    - <integer> ::= <digit> | <integer><digit>
- Let’s derive an integer 352 from this grammar

Derivation of 352 as an Integer

- Process starts with:
  - <Integer>
- Use a grammar rule to enable each step:
  - <integer> ::= <digit> | <integer><digit>
- Replace a nonterminal by a right-hand side of its (one of) rule
  - <integer> ::= <integer><digit>
  - <integer> ::= <integer>2
- Repeat the above steps
  - <integer> ::= <integer><digit>
  - <integer> ::= <integer>2
  - <integer> ::= <integer><digit>2
  - <integer> ::= <integer>52
  - <integer> ::= <digit>52
  - <integer> ::= 352
- Process is finished when there are only terminal symbols remain

A Different Derivation

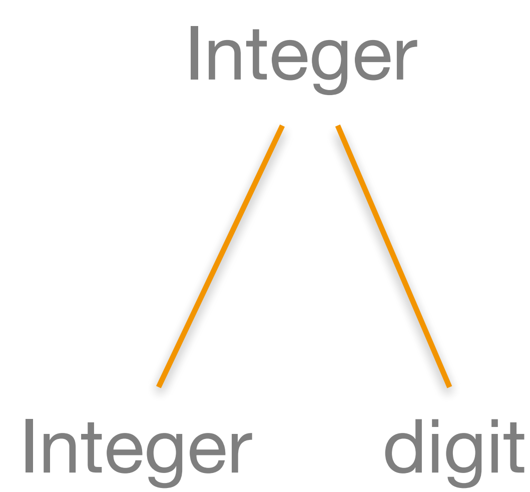
- Derive 352
  - <integer> ::= <integer><digit>
  - ::= <integer><digit><digit>
  - ::= <digit><digit><digit>
  - ::= 3<digit><digit>
  - ::= 35<digit>
  - ::= 352
- The process is called a leftmost derivation

Notations for Derivation

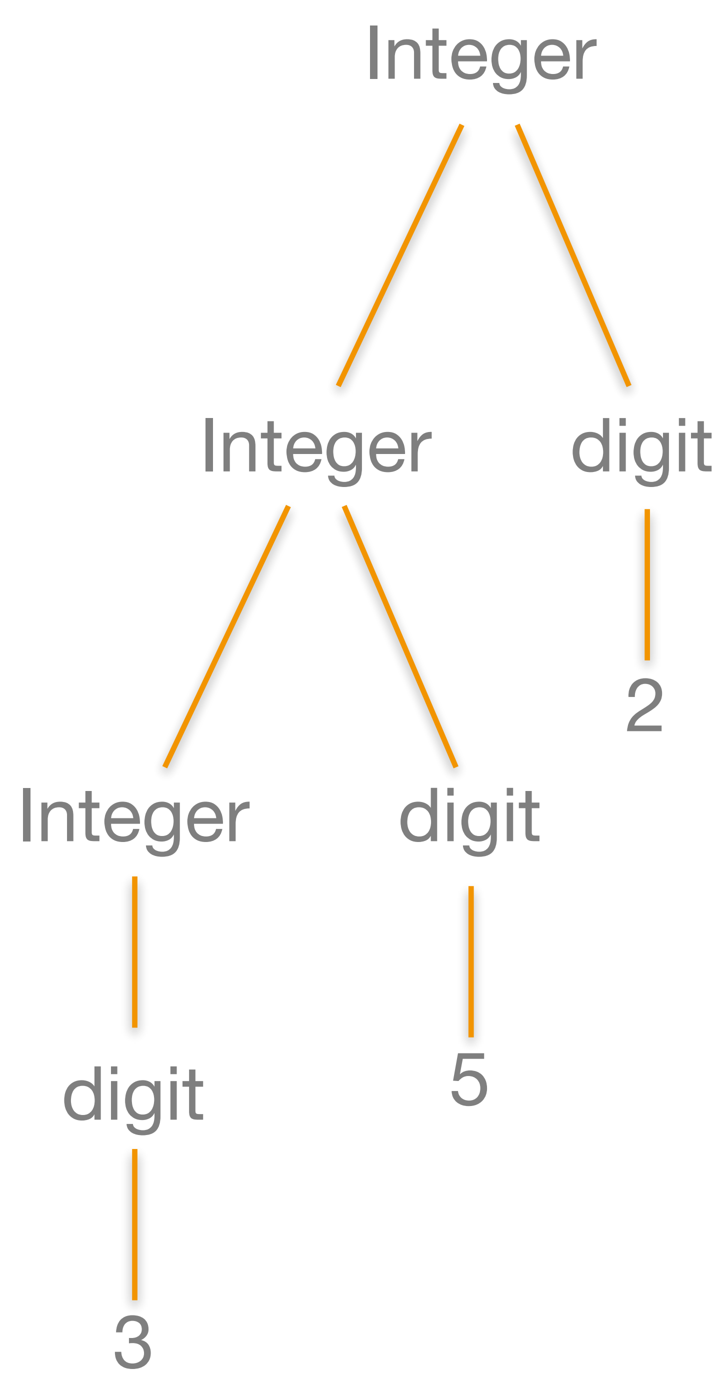
- Integer $\Rightarrow^*$352 means 352 can be derived in a finite number of steps using the grammar for Integer
- 352 $\in$ L(G) means 352 is a member of the language defined by Grammar G
- $L(G) = \{ w \in T^* \mid \text{Integer} \Rightarrow^* w\}$ — language defined by grammar G is the set of all symbol string w that can be derived as an Integer

- Aims to show relations between the elements and the grammar
- A parse tree is a graphical representation of a derivation
    - Each internal node of the tree corresponds to a step in the derivation
    - Each child 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

- The step <Integer> ::= <Integer><digit> appears in a parse tree as:

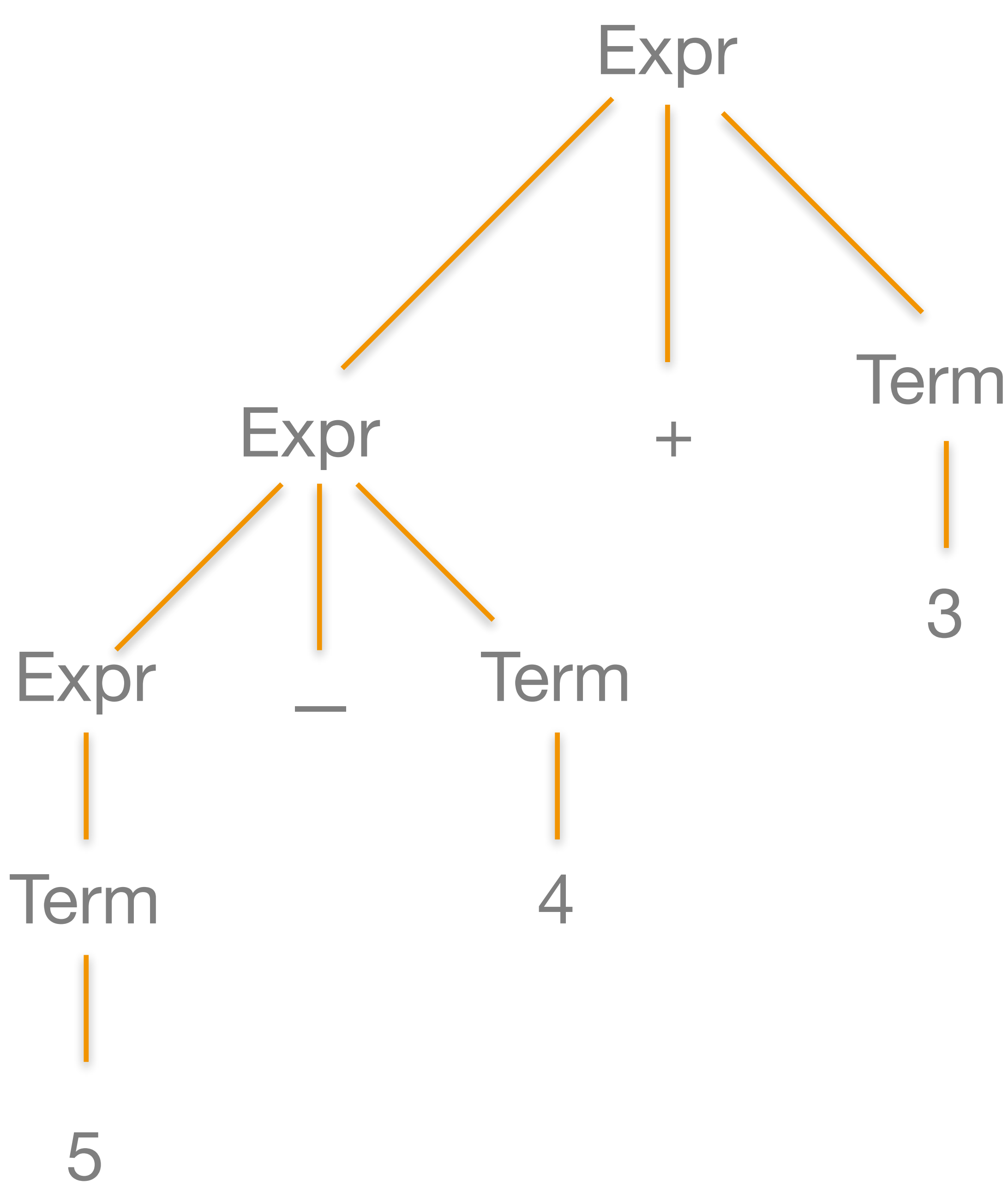


Parse Tree for 352 as an Integer



- Grammar for the language if arithmetic expression with 1 digit integers, addition, and subtraction
  - <Expr> ::= <Expr> + <Term> | <Expr> — <Term> | Term
  - Term ::= 0 | 1 | … | 9
- Draw parse tree of 5-4+3

Parse Tree of 5-4+3



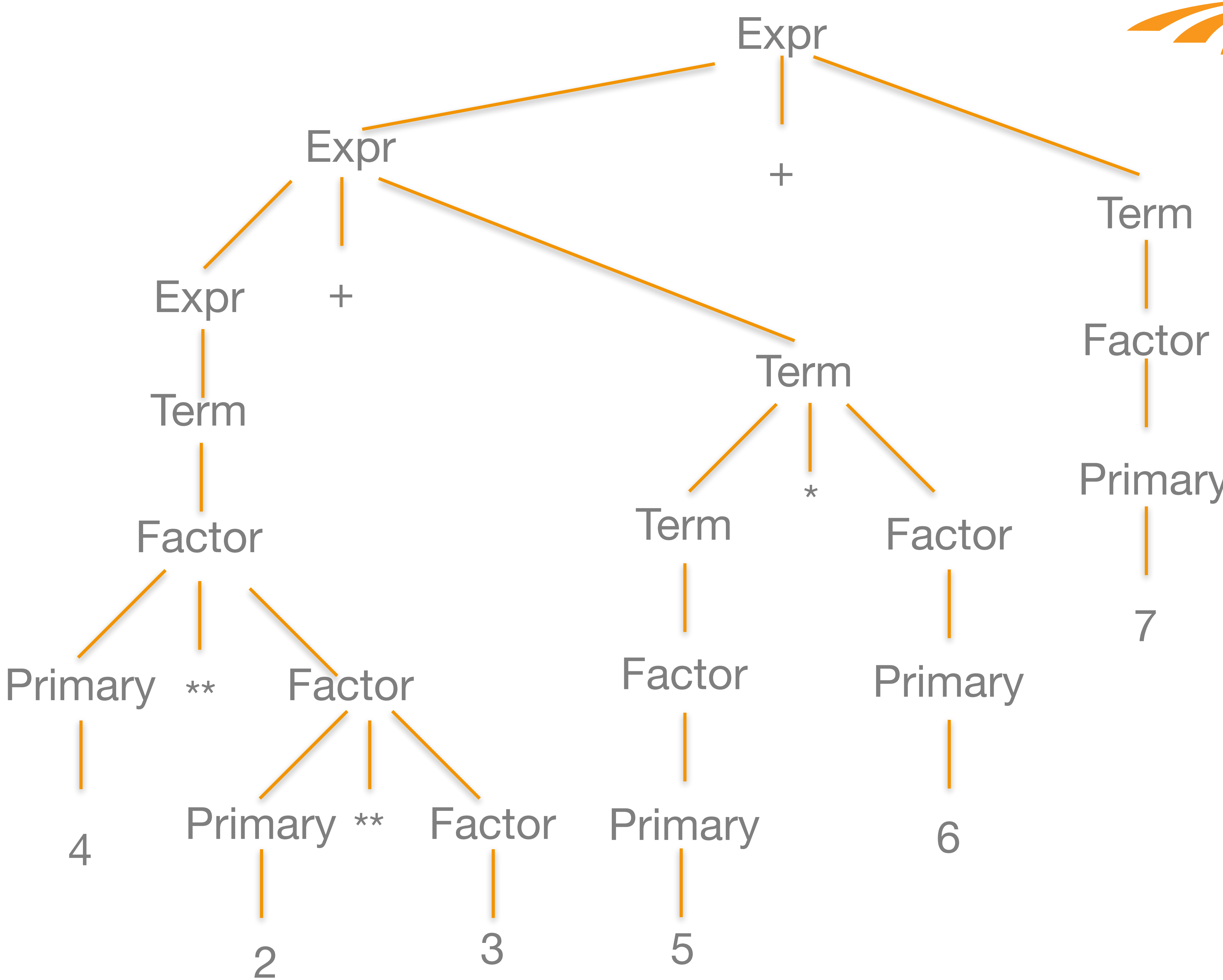
- A grammar can define associativity and precedence among the operators in an expression
  - e.g., + and — are left-associative operators in math and
  - * and / have higher precedence than + and —
- Consider a Grammar G_1:
  - <Expr> ::= <Expr> + <Term> | <Expr> — <Term> | Term
  - <Term> ::= <Term> * <Factor> | <Term> / <Factor> | <Term> % <Factor> | <Factor>
  - <Factor> ::= <Primary> ** <Factor> | <Primary>
  - <Primary> ::= 0 | 1 | … | 9

Associativity and Precedence for G_1

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

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

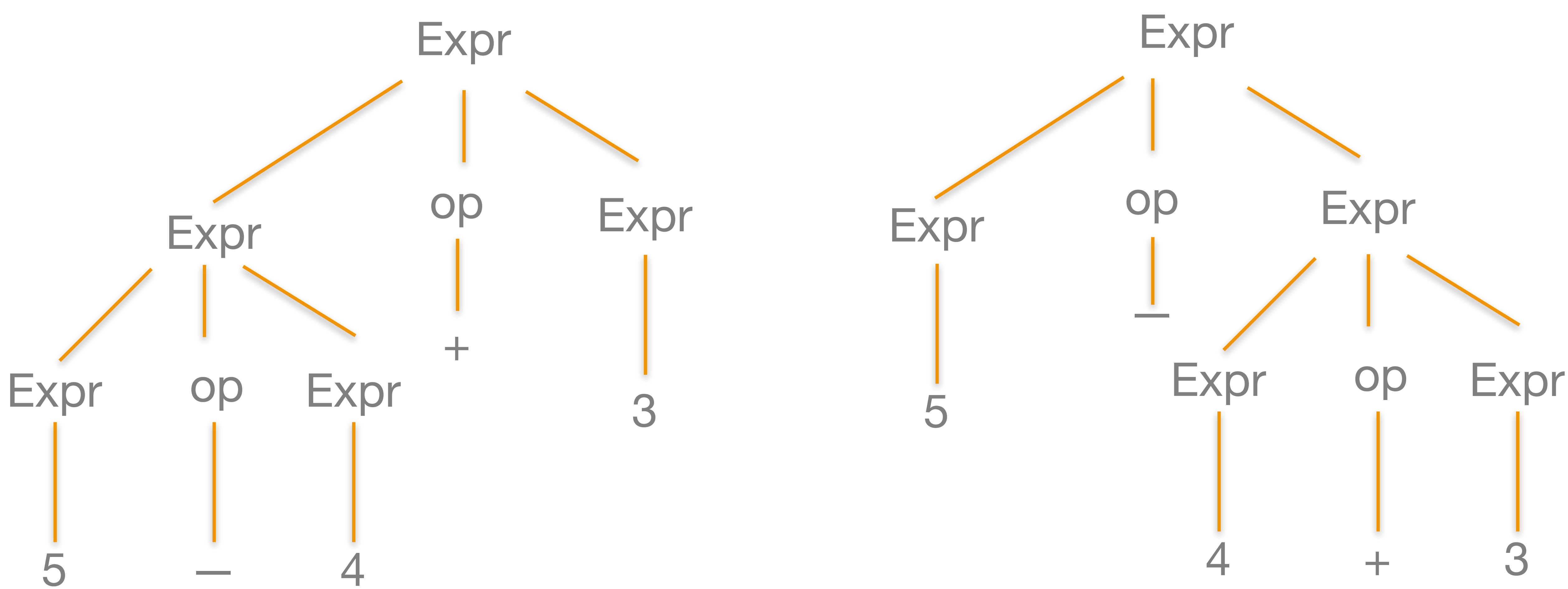
*Parse Tree of 4**2**3+5*6+7*



- An grammar is *ambiguous* if one of its strings has two or more different parse trees
- An ambiguous grammar G_2:
  - <Expr> ::= <Expr> op <Expr> | (Expr) | Integer
  - <op> ::= + | — | / | % | **
- G_2 is equivalent to G_1, i.e., both languages are same
- G_2 has fewer productions and nonterminals than G_1
- G_2 is ambiguous

Parse Tree of 5-4+3 using G₂

- Ambiguous grammar G_2:
  - <Expr> ::= <Expr> op <Expr> | (Expr) | Integer
  - <op> ::= + | — | / | % | **



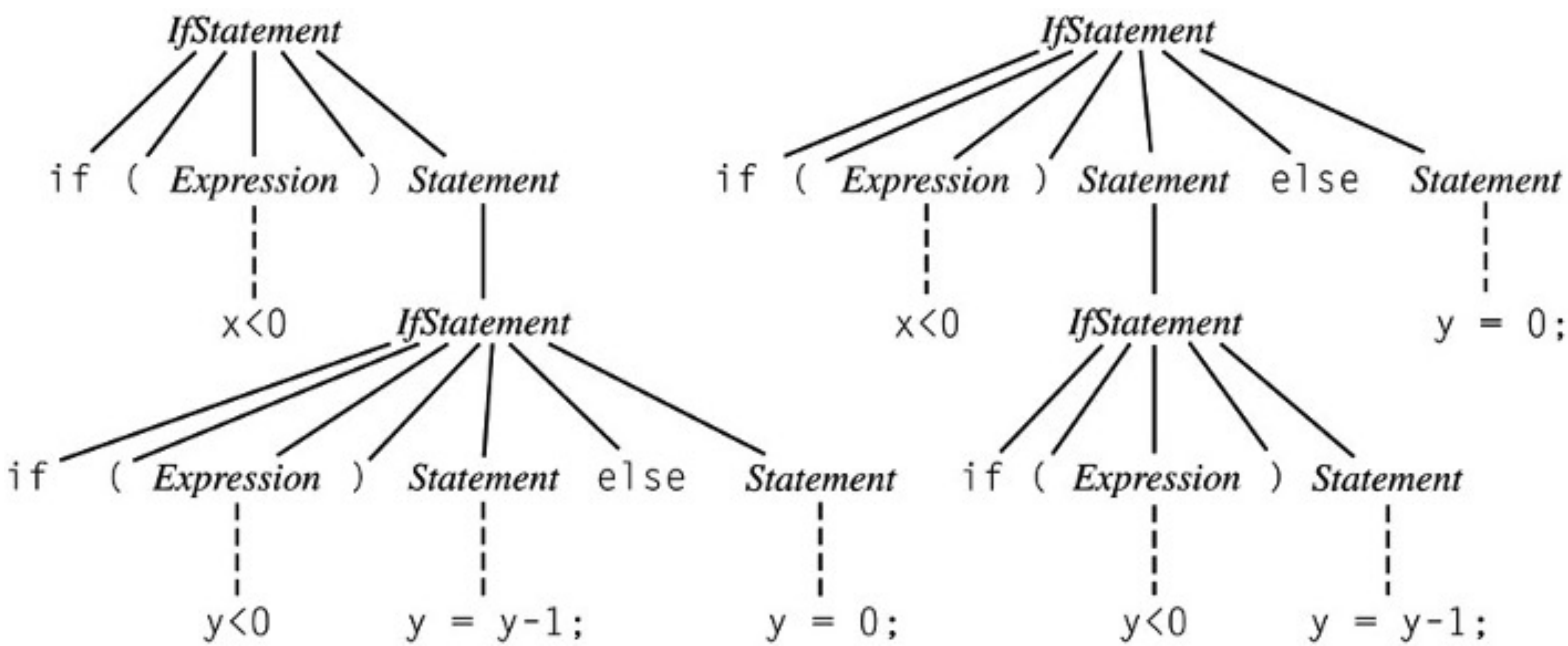
- <IfStatement> ::= <if (Expression)><Statement> | <if (Expression)><Statement> <else> <Statement>
- <Statement> ::= <Assignment> | <IfStatement> | <Block>
- <Block> ::= <{> <Statement> <}>
- <Statement> ::= <Statement> | <Statement> <Statement>

Example

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

With which if the else associate?

The Dangling Else Ambiguity



The Dangling Else Ambiguity

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

Can associate with any if

- Algol 60, C, C++:
  - associate each else with the closest if
  - use {} or begin … end to override
- Algol 68, Modula, Ada:
  - use explicit delimiter to end every conditional
  - e.g., if … fi
- 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 statements except <IfThenStatement>

- Programming Languages: Principles and Paradigms by Allen B. Tucker and Robert E. Noonan