

# Where Syntax Meets Semantics

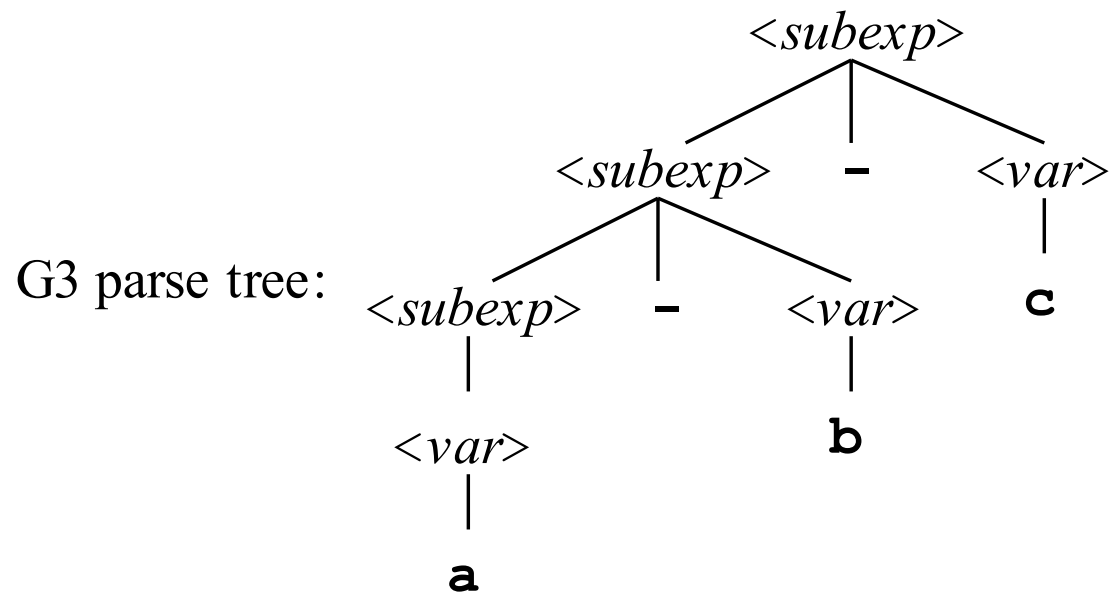
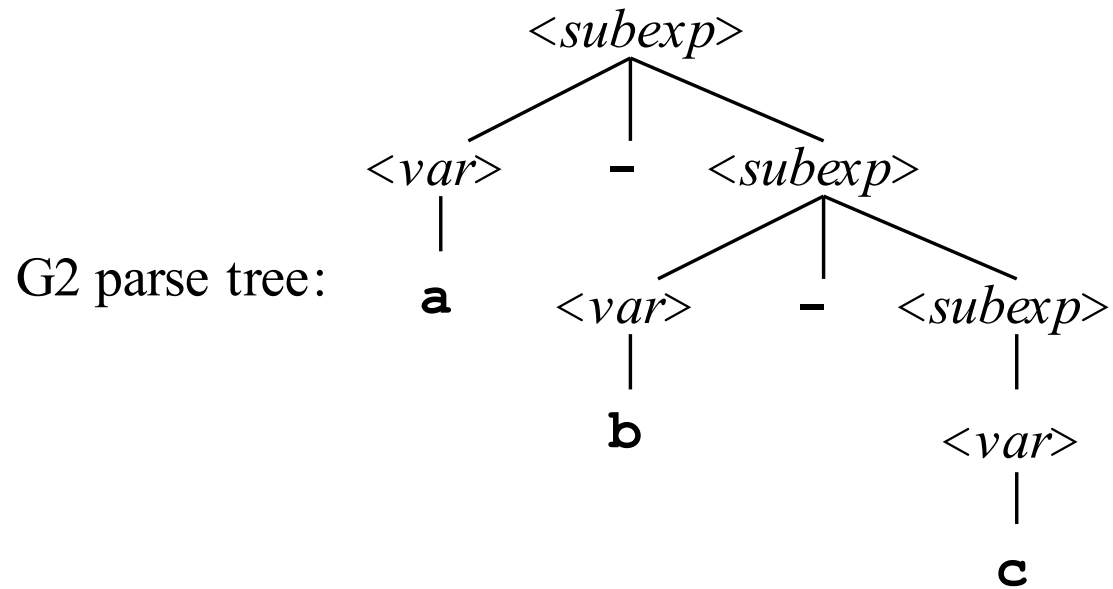
# Three “Equivalent” Grammars

G1 :  $\langle subexp \rangle ::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \mid \langle subexp \rangle - \langle subexp \rangle$

G2 :  $\langle subexp \rangle ::= \langle var \rangle - \langle subexp \rangle \mid \langle var \rangle$   
 $\langle var \rangle ::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

G3 :  $\langle subexp \rangle ::= \langle subexp \rangle - \langle var \rangle \mid \langle var \rangle$   
 $\langle var \rangle ::= \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

These grammars all define the same language: the language of strings that contain one or more **as**, **bs** or **cs** separated by minus signs. But...



# Why Parse Trees Matter

- We want the structure of the parse tree to correspond to the semantics of the string it generates
- This makes grammar design much harder: we're interested in the structure of each parse tree, not just in the generated string
- Parse trees are where syntax meets semantics

# Outline

- Operators
- Precedence
- Associativity
- Other ambiguities: dangling else
- Cluttered grammars
- Parse trees and EBNF
- Abstract syntax trees

# Operators

- Special syntax for frequently-used simple operations like addition, subtraction, multiplication and division
- The word *operator* refers both to the token used to specify the operation (like  $+$  and  $*$ ) and to the operation itself
- Usually predefined, but not always
- Usually a single token, but not always

# Operator Terminology

- *Operands* are the inputs to an operator, like **1** and **2** in the expression **1+2**
- *Unary* operators take one operand: **-1**
- *Binary* operators take two: **1+2**
- *Ternary* operators take three: **a?b:c**

# More Operator Terminology

- In most programming languages, binary operators use an *infix* notation: **a + b**
- Sometimes you see *prefix* notation: **+ a b**
- Sometimes *postfix* notation: **a b +**
- Unary operators, similarly:
  - (Can't be infix, of course)
  - Can be prefix, as in **-1**
  - Can be postfix, as in **a++**



# Outline

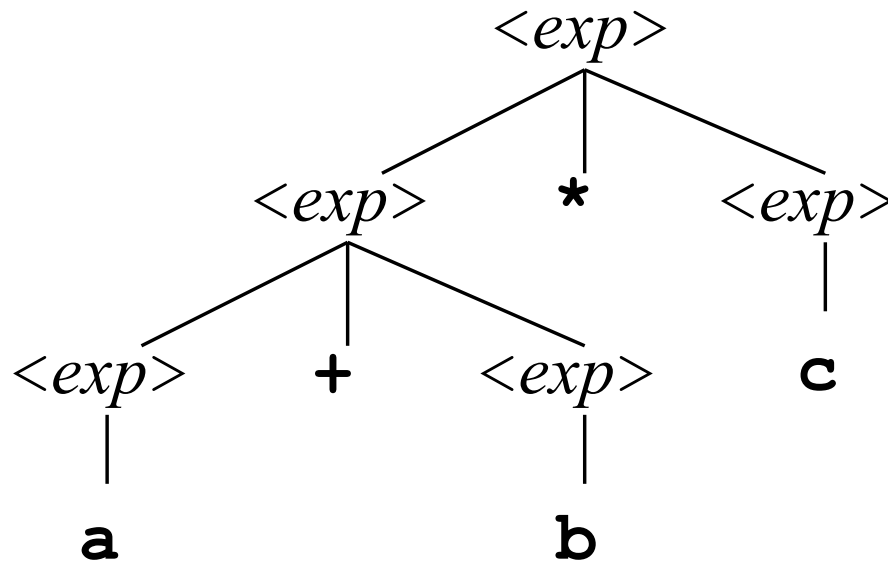
- Operators
- **Precedence**
- Associativity
- Other ambiguities: dangling else
- Cluttered grammars
- Parse trees and EBNF
- Abstract syntax trees

# Working Grammar

$$\begin{array}{lcl} \text{G4: } & \langle exp \rangle & ::= \langle exp \rangle + \langle exp \rangle \\ & & | \langle exp \rangle * \langle exp \rangle \\ & & | (\langle exp \rangle) \\ & & | \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \end{array}$$

This generates a language of arithmetic expressions using parentheses, the operators **+** and **\***, and the variables **a**, **b** and **c**

# Issue #1: Precedence



Our grammar generates this tree for **a+b\*c**. In this tree, the addition is performed before the multiplication, which is not the usual convention for operator *precedence*.

# Operator Precedence

- Applies when the order of evaluation is not completely decided by parentheses
- Each operator has a *precedence level*, and those with higher precedence are performed before those with lower precedence, as if parenthesized
- Most languages put **\*** at a higher precedence level than **+**, so that

$$\mathbf{a+b*c = a+(b*c)}$$

# Precedence Examples

- C (15 levels of precedence—too many?)

`a = b < c ? * p + b * c : 1 << d ()`

- Pascal (5 levels—not enough?)

`a <= 0 or 100 <= a`

- Smalltalk (1 level for all binary operators)

`a + b * c`

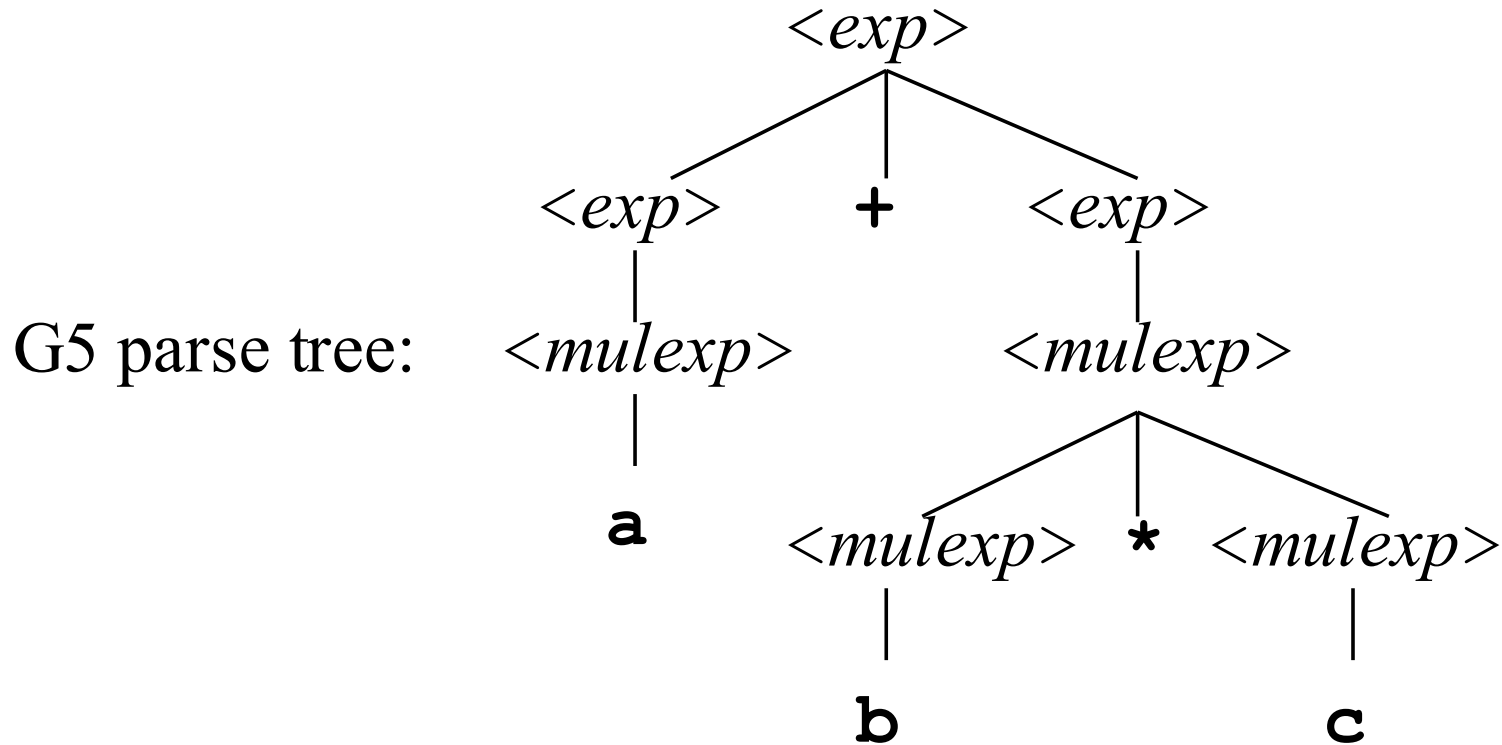
# Precedence In The Grammar

$$\begin{array}{lcl} \text{G4: } <exp> & ::= & <exp> + <exp> \\ & & | <exp> * <exp> \\ & & | (<exp>) \\ & & | \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \end{array}$$

To fix the precedence problem, we modify the grammar so that it is forced to put  $*$  below  $+$  in the parse tree.

$$\begin{array}{lcl} \text{G5:} & \langle \textit{exp} \rangle & ::= \langle \textit{exp} \rangle + \langle \textit{exp} \rangle \mid \langle \textit{mulexp} \rangle \\ & \langle \textit{mulexp} \rangle & ::= \langle \textit{mulexp} \rangle * \langle \textit{mulexp} \rangle \\ & & \mid (\langle \textit{exp} \rangle) \\ & & \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{c} \end{array}$$

# Correct Precedence



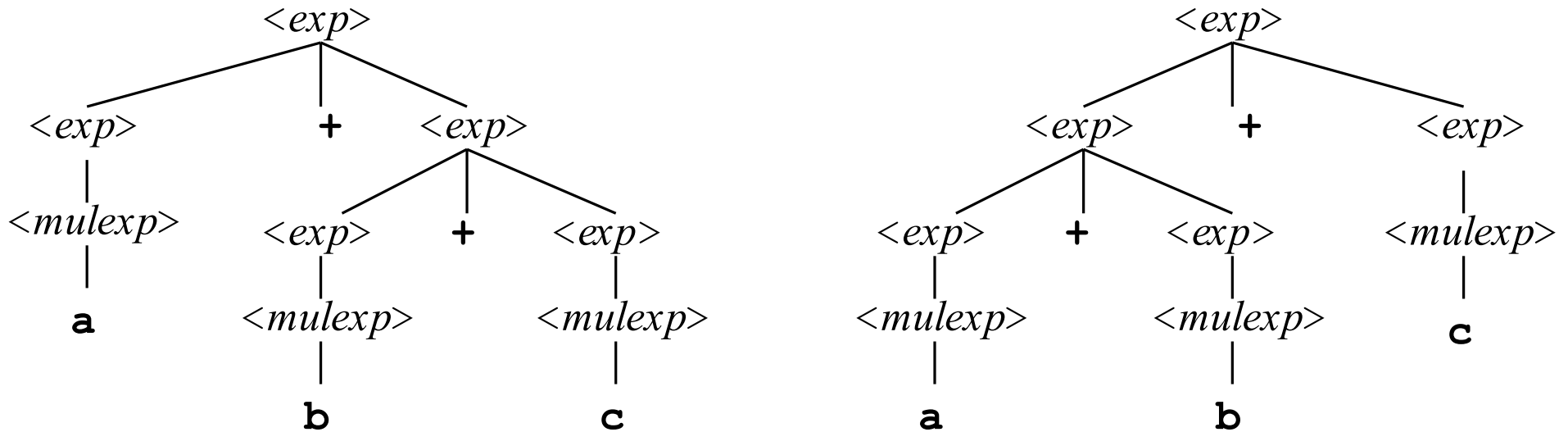
Our new grammar generates this tree for **a+b\*c**. It generates the same language as before, but no longer generates parse trees with incorrect precedence.

# Outline

- Operators
- Precedence
- **Associativity**
- Other ambiguities: dangling else
- Cluttered grammars
- Parse trees and EBNF
- Abstract syntax trees



# Issue #2: Associativity



Our grammar G5 generates both these trees for **a+b+c**.  
The first one is not the usual convention for operator *associativity*.

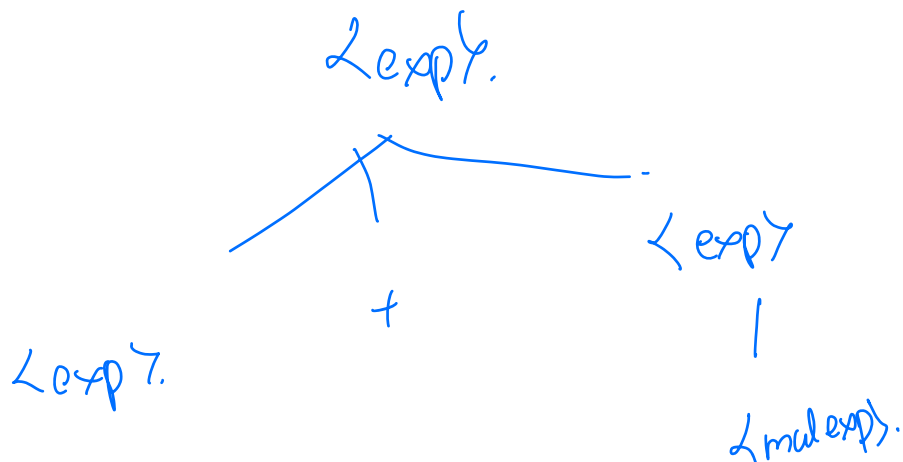
# Operator Associativity

- Applies when the order of evaluation is not decided by parentheses or by precedence
- *Left-associative* operators group left to right:  $a+b+c+d = ((a+b)+c)+d$
- *Right-associative* operators group right to left:  $a+b+c+d = a+(b+(c+d))$
- Most operators in most languages are left-associative, but there are exceptions

$\langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{mulexp} \rangle$  (a + b) + c

$\langle \text{mulexp} \rangle ::= \langle \text{mulexp} \rangle * \langle \text{rotxp} \rangle \mid \langle \text{mulexp} \rangle \mid \langle \text{mulexp} \rangle$

$\langle \text{rotxp} \rangle ::= \langle \text{rotxp} \rangle / a \mid b \mid c$



# Associativity Examples

## □ C

**a<<b<<c** — most operators are left-associative  
**a=b=0** — right-associative (assignment)

## □ ML

**3-2-1** — most operators are left-associative  
**1::2::nil** — right-associative (list builder)

## □ Fortran

**a/b\*c** — most operators are left-associative  
**a\*\*b\*\*c** — right-associative (exponentiation)

~~Assoc~~ Associativity  $\rightarrow$  left for ~~Precedence~~  $\rightarrow$  none for the

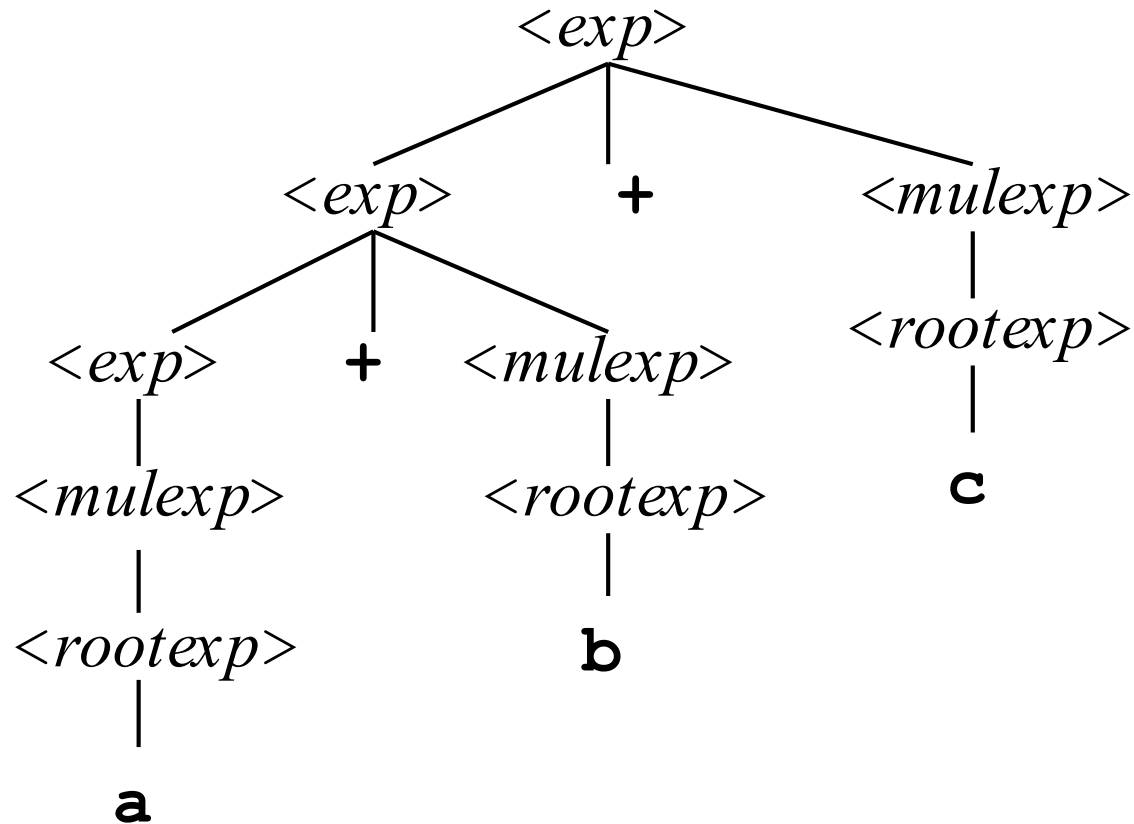
# Associativity In The Grammar *away.*

G5:  $\langle exp \rangle ::= \langle exp \rangle + \langle exp \rangle \mid \langle mulexp \rangle$   
 $\langle mulexp \rangle ::= \langle mulexp \rangle * \langle mulexp \rangle$   
 $\mid (\langle exp \rangle)$   
 $\mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

To fix the associativity problem, we modify the grammar to make trees of **+**s grow down to the left (and likewise for **\***s)

*to get left associativity*  
↓  
G6:  $\langle exp \rangle ::= \langle exp \rangle + \langle mulexp \rangle \mid \langle mulexp \rangle$   
 $\langle mulexp \rangle ::= \langle mulexp \rangle * \langle rootexp \rangle \mid \langle rootexp \rangle$   
 $\langle rootexp \rangle ::= (\langle exp \rangle)$   
 $\mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{c}$

# Correct Associativity



Our new grammar generates this tree for **a+b+c**. It generates the same language as before, but no longer generates trees with incorrect associativity.

# Outline

- Operators
- Precedence
- Associativity
- **Other ambiguities: dangling else**
- Cluttered grammars
- Parse trees and EBNF
- Abstract syntax trees

# Issue #3: Ambiguity

- G4 was *ambiguous*: it generated more than one parse tree for the same string
- Fixing the associativity and precedence problems eliminated all the ambiguity
- This is usually a good thing: the parse tree corresponds to the meaning of the program, and we don't want ambiguity about that
- Not all ambiguity stems from confusion about precedence and associativity...



# Dangling Else In Grammars

$\langle \text{stmt} \rangle ::= \langle \text{if-stmt} \rangle \mid \mathbf{s1} \mid \mathbf{s2}$

$\langle \text{if-stmt} \rangle ::= \mathbf{if} \langle \text{expr} \rangle \mathbf{then} \langle \text{stmt} \rangle \mathbf{else} \langle \text{stmt} \rangle$   
 $\mid \mathbf{if} \langle \text{expr} \rangle \mathbf{then} \langle \text{stmt} \rangle$

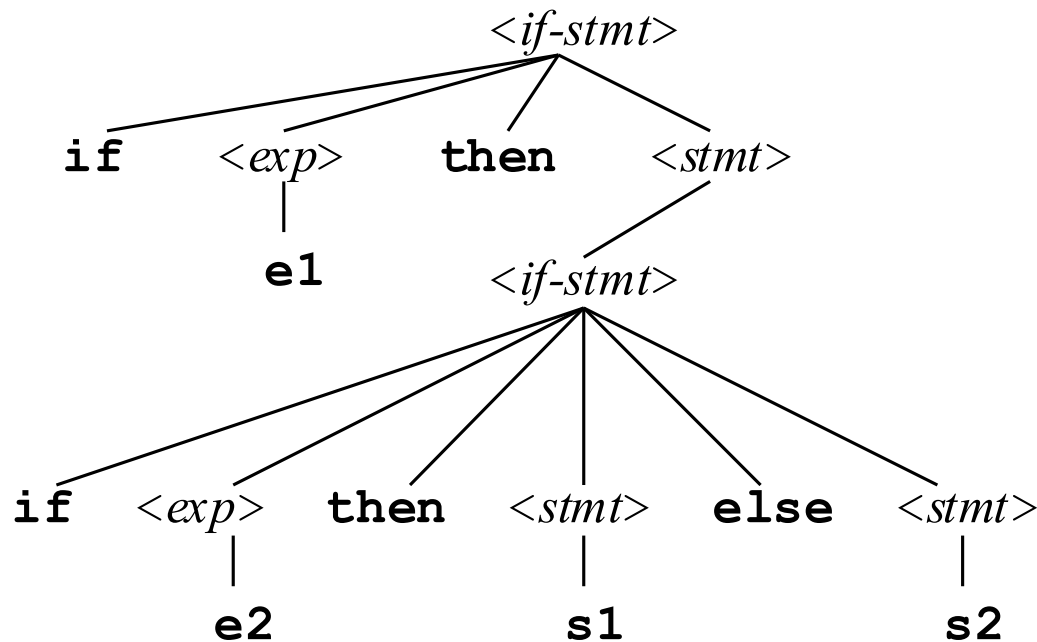
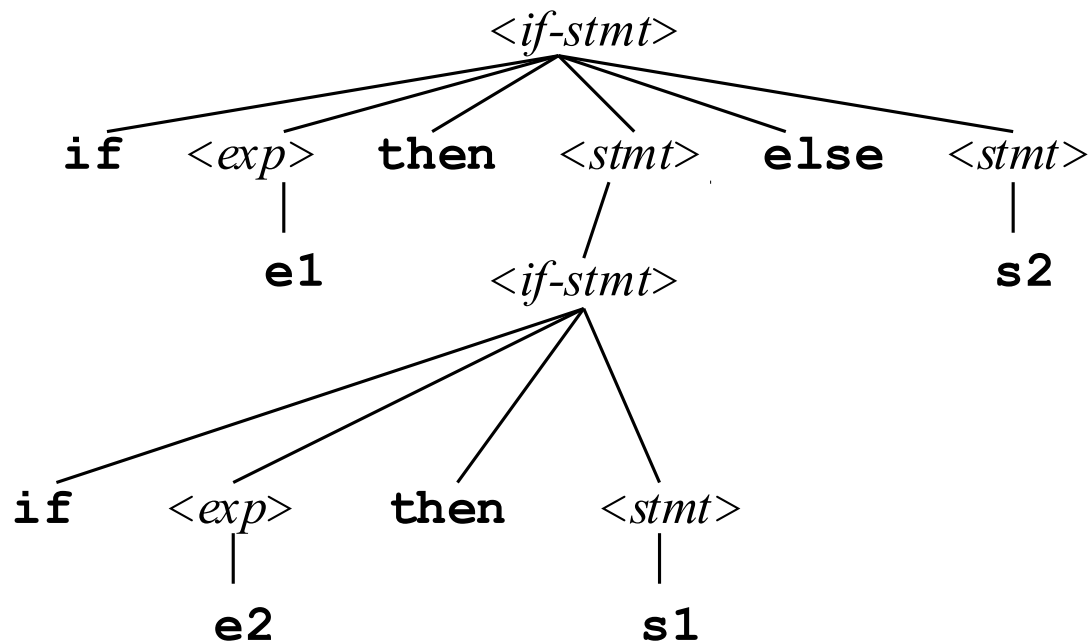
$\langle \text{expr} \rangle ::= \mathbf{e1} \mid \mathbf{e2}$

This grammar has a classic “dangling-else ambiguity.” The statement we want derive is

**if e1 then if e2 then s1 else s2**

*always go with nested if*

and the next slide shows two different parse trees for it...



Most languages that have this problem choose this parse tree: **else** goes with nearest unmatched **then**

# Eliminating The Ambiguity

$\langle stmt \rangle ::= \langle if-stmt \rangle \mid s1 \mid s2$   
 $\langle if-stmt \rangle ::= \text{if } \langle expr \rangle \text{ then } \langle stmt \rangle \text{ else } \langle stmt \rangle$   
 $\qquad \qquad \qquad \mid \text{if } \langle expr \rangle \text{ then } \langle stmt \rangle$   
 $\langle expr \rangle ::= e1 \mid e2$

We want to insist that if this expands into an **if**, that **if** must already have its own **else**. First, we make a new non-terminal  $\langle full-stmt \rangle$  that generates everything  $\langle stmt \rangle$  generates, except that it can not generate **if** statements with no **else**:

$$\begin{aligned} \langle full-stmt \rangle & ::= \langle full-if \rangle \mid \mathbf{s1} \mid \mathbf{s2} \\ \langle full-if \rangle & ::= \mathbf{if} \langle expr \rangle \mathbf{then} \langle full-stmt \rangle \mathbf{else} \langle full-stmt \rangle \end{aligned}$$

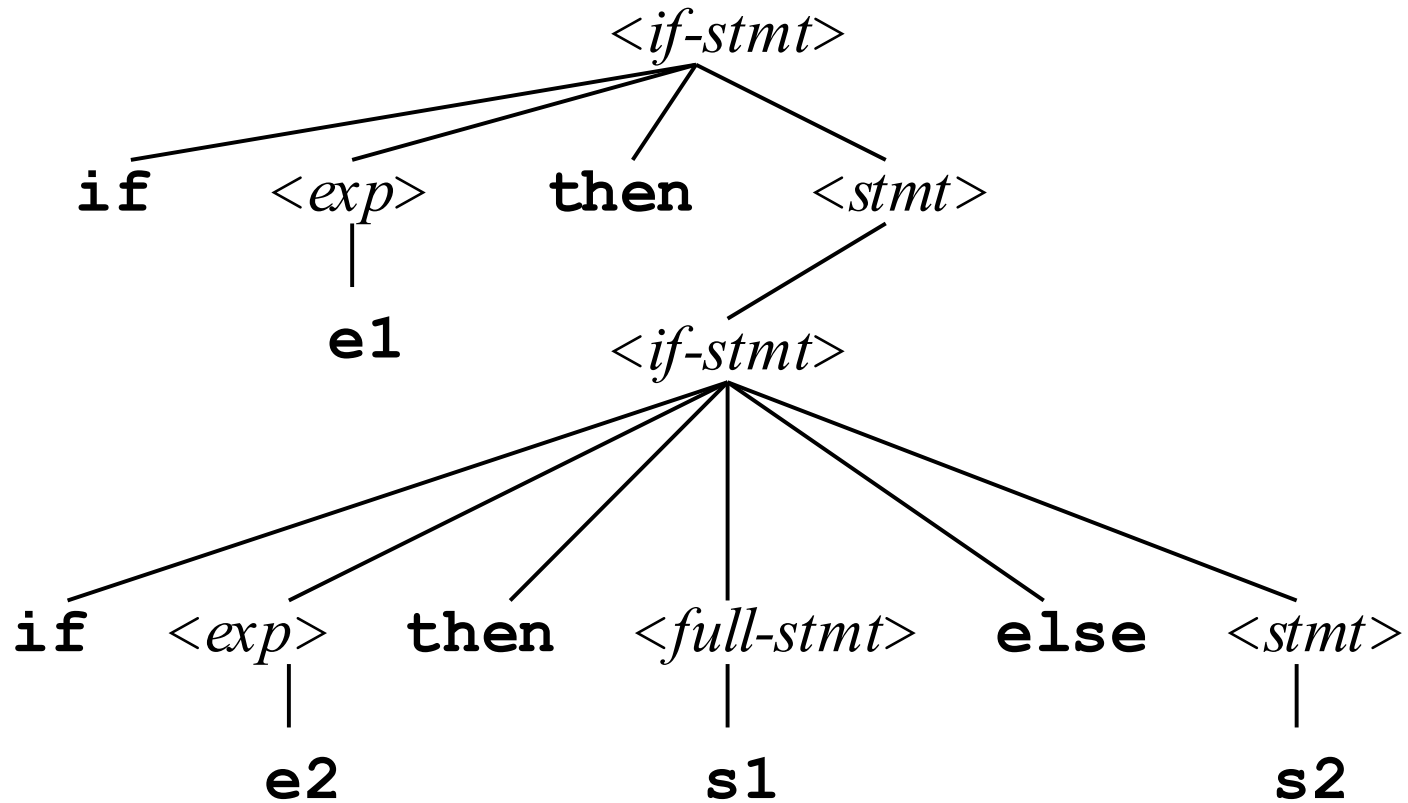
# Eliminating The Ambiguity

$\langle stmt \rangle ::= \langle if-stmt \rangle \mid s1 \mid s2$   
 $\langle if-stmt \rangle ::= \text{if } \langle expr \rangle \text{ then } \langle full-stmt \rangle \text{ else } \langle stmt \rangle$   
 $\qquad \qquad \qquad \mid \text{if } \langle expr \rangle \text{ then } \langle stmt \rangle$   
 $\langle expr \rangle ::= e1 \mid e2$

Then we use the new non-terminal here.

The effect is that the new grammar can match an **else** part with an **if** part only if all the nearer **if** parts are already matched.

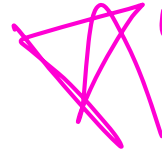
# Correct Parse Tree



# Dangling Else

- ❑ We fixed the grammar, but...
- ❑ The grammar trouble reflects a problem with the language, which we did not change
- ❑ A chain of if-then-else constructs can be very hard for people to read
- ❑ Especially true if some but not all of the else parts are present

# Clearer Styles



```
int a=0;  
if (0==0)  
    if (0==1) a=1;  
    else a=2;
```

Better: correct indentation

```
int a=0;  
if (0==0) {  
    if (0==1) a=1;  
    else a=2;  
}
```

Even better: use of a block  
reinforces the structure

# Languages That Don't Dangle

- Some languages define if-then-else in a way that forces the programmer to be more clear
  - Algol does not allow the **then** part to be another **if** statement – though it can be a block containing an **if** statement
  - Ada requires each **if** statement to be terminated with an **end if**
  - Python requires nested **if** statement to be indented



# Outline

- Operators
- Precedence
- Associativity
- Other ambiguities: dangling else
- **Cluttered grammars**
- Parse trees and EBNF
- Abstract syntax trees

# Clutter

- ❑ The new if-then-else grammar is harder for people to read than the old one
- ❑ It has a lot of clutter: more productions and more non-terminals
- ❑ Same with G4, G5 and G6: we eliminated the ambiguity but made the grammar harder for people to read
- ❑ This is not always the right trade-off

# Reminder: Multiple Audiences

- In Chapter 2 we saw that grammars have multiple audiences:
  - Novices want to find out what legal programs look like
  - Experts—advanced users and language system implementers—want an exact, detailed definition
  - Tools—parser and scanner generators—want an exact, detailed definition in a particular, machine-readable form
- Tools often need ambiguity eliminated, while people often prefer a more readable grammar

# Options

- ❑ Rewrite grammar to eliminate ambiguity
- ❑ Leave ambiguity but explain in accompanying text how things like associativity, precedence, and the dangling else should be parsed
- ❑ Do both in separate grammars

# Outline

- Operators
- Precedence
- Associativity
- Other ambiguities: dangling else
- Cluttered grammars
- **Parse trees and EBNF**
- Abstract syntax trees

# EBNF and Parse Trees

- You know that  $\{x\}$  means "zero or more repetitions of  $x$ " in EBNF
- So  $\langle exp \rangle ::= \langle mulexp \rangle \{ + \langle mulexp \rangle \}$  should mean a  $\langle mulexp \rangle$  followed by zero or more repetitions of " $+ \langle mulexp \rangle$ "
- But what then is the associativity of that  $+$  operator? What kind of parse tree would be generated for **a+a+a**?

# EBNF and Associativity

## □ One approach:

- Use  $\{ \}$  anywhere it helps
- Add a paragraph of text dealing with ambiguities, associativity of operators, etc.

## □ Another approach:

- Define a convention: for example, that the form  $\langle exp \rangle ::= \langle mulexp \rangle \{ + \langle mulexp \rangle \}$  will be used only for left-associative operators
- Use explicitly recursive rules for anything unconventional:

$$\langle expa \rangle ::= \langle expb \rangle [ = \langle expa \rangle ]$$

# About Syntax Diagrams

- Similar problem: what parse tree is generated?
- As in EBNF applications, add a paragraph of text dealing with ambiguities, associativity, precedence, and so on



# Outline

- Operators
- Precedence
- Associativity
- Other ambiguities: dangling else
- Cluttered grammars
- Parse trees and EBNF
- **Abstract syntax trees**



# Full-Size Grammars

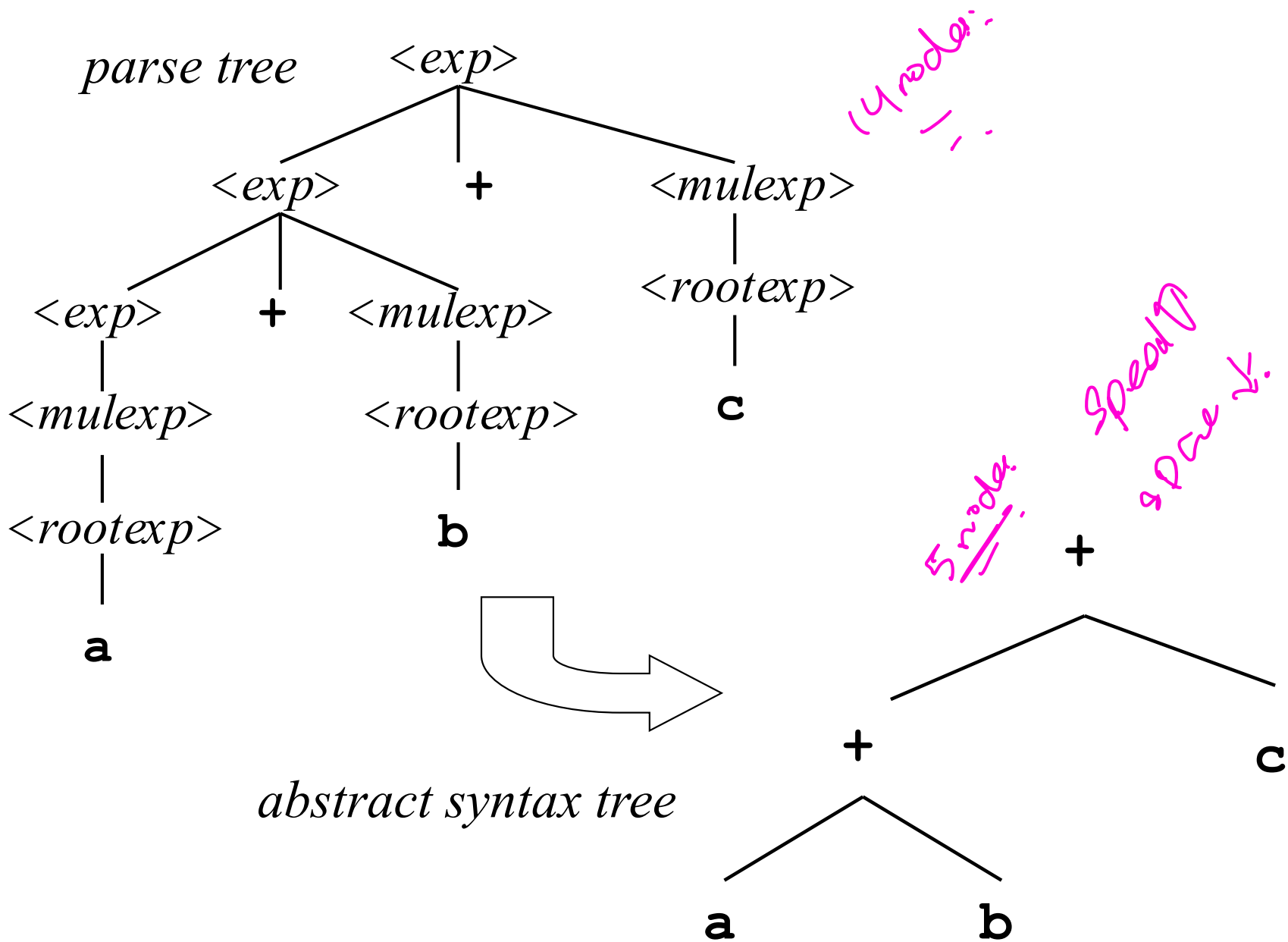
- ❑ In any realistically large language, there are many non-terminals
- ❑ Especially true when in the cluttered but unambiguous form needed by parsing tools
- ❑ Extra non-terminals guide construction of unique parse tree
- ❑ Once parse tree is found, such non-terminals are no longer of interest

# Abstract Syntax Tree

to optimize → space

→ speed.

- Language systems usually store an abbreviated version of the parse tree called the *abstract syntax tree*
- Details are implementation-dependent
- Usually, there is a node for every operation, with a subtree for every operand



# Parsing, Revisited

- When a language system parses a program, it goes through all the steps necessary to find the parse tree
- But it usually does not construct an explicit representation of the parse tree in memory
- Most systems construct an AST instead
- We will see ASTs again in Chapter 23

# Conclusion

- Grammars define syntax, *and more*
- They define not just a set of legal programs, but a parse tree for each program
- The structure of a parse tree corresponds to the order in which different parts of the program are to be executed
- Thus, grammars contribute (a little) to the definition of semantics