Where Syntax Meets Semantics

Three "Equivalent" Grammars

G1: *<subexp>* `::=` **`a`** | **`b`** | **`c`** | *<subexp>* **`-`** *<subexp>*

G2: *<subexp>* ::= *<var>* **-** *<subexp>* | *<var>*
<var> ::= **a** | **b** | **c**

G3: *<subexp>* ::= *<subexp>* - *<var>* | *<var>*
<var> ::= **a** | **b** | **c**

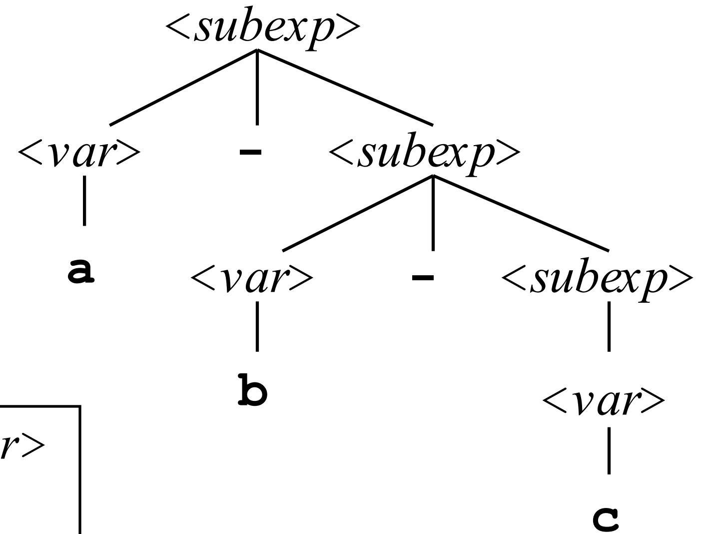
These grammars all define the same language: the language of strings that contain one or more **`a`**'s, **`b`**'s or **`c`**'s separated by minus signs.

But the parse trees are not equivalent.

Parse of:
a-b-c

a-**<u>b-c</u>**
1-**<u>2-3</u>** = 2

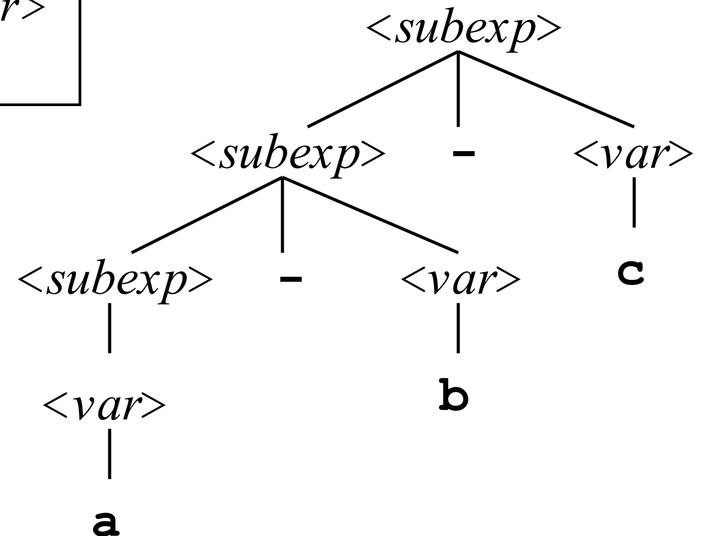
G2 parse tree:



G2: <subexp> ::= <var> - <subexp> | <var>
<var> ::= **a** | **b** | **c**

G3: <subexp> ::= <subexp> - <var> | <var>
<var> ::= **a** | **b** | **c**

G3 parse tree:



<u>a-b</u>-c
<u>1-2</u>-3 = -4

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
- Reverse Polish Notation and Evaluation

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`**
  - **`What is (* (+ 3 4) 2)?`**
  - **`What is * + 3 4 2 ?`**
- Sometimes *postfix* notation: **`a b +`**
  - **`What is (2 (3 4 +) *)?`**
  - **`What is 2 3 4 + * ?`**
- 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
- Reverse Polish Notation and Evaluation

Working Grammar – G4

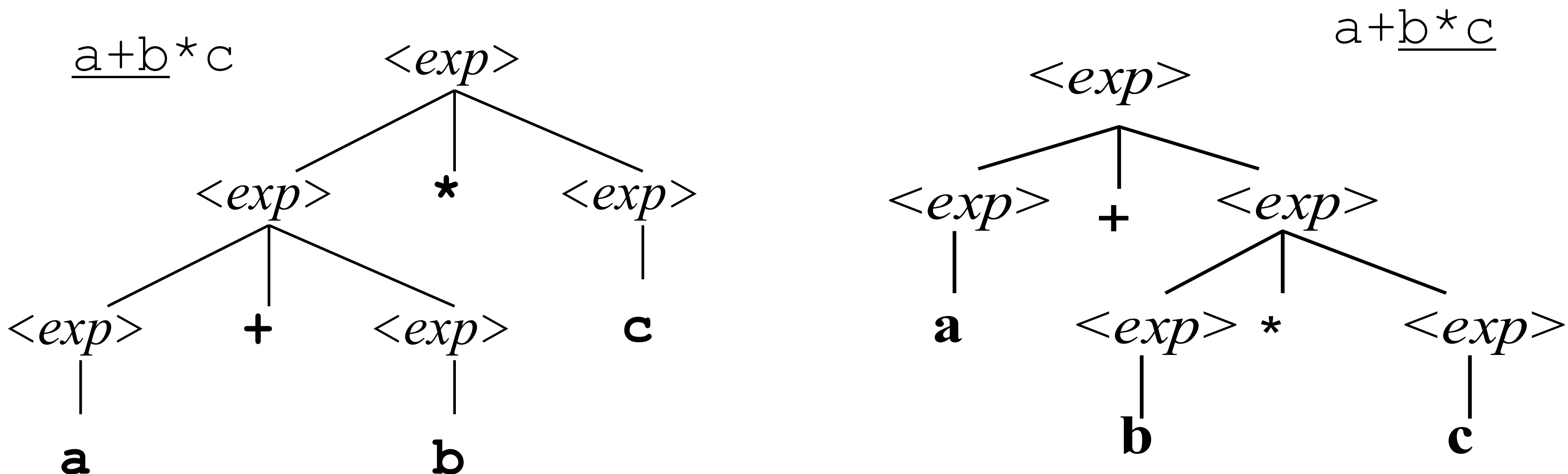
```
G4:  <exp> ::= <exp> + <exp> |
               <exp> * <exp> |
               (<exp>) |
               a | b | c
```

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

Issue #1: Precedence

```
G4:  <exp> ::= <exp> + <exp> |
               <exp> * <exp> |
               (<exp>) |
               a | b | c
```

Our G4 grammar can generate two trees for **`a+b*c`**. In the left 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:

`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
```

Error!

- Smalltalk (1 level for all binary operators)

```
a + b * c
```

Precedence In The Grammar

```
G4:   <exp> ::= <exp> + <exp>
              | <exp> * <exp>
              | (<exp>)
              | a | b | c
```

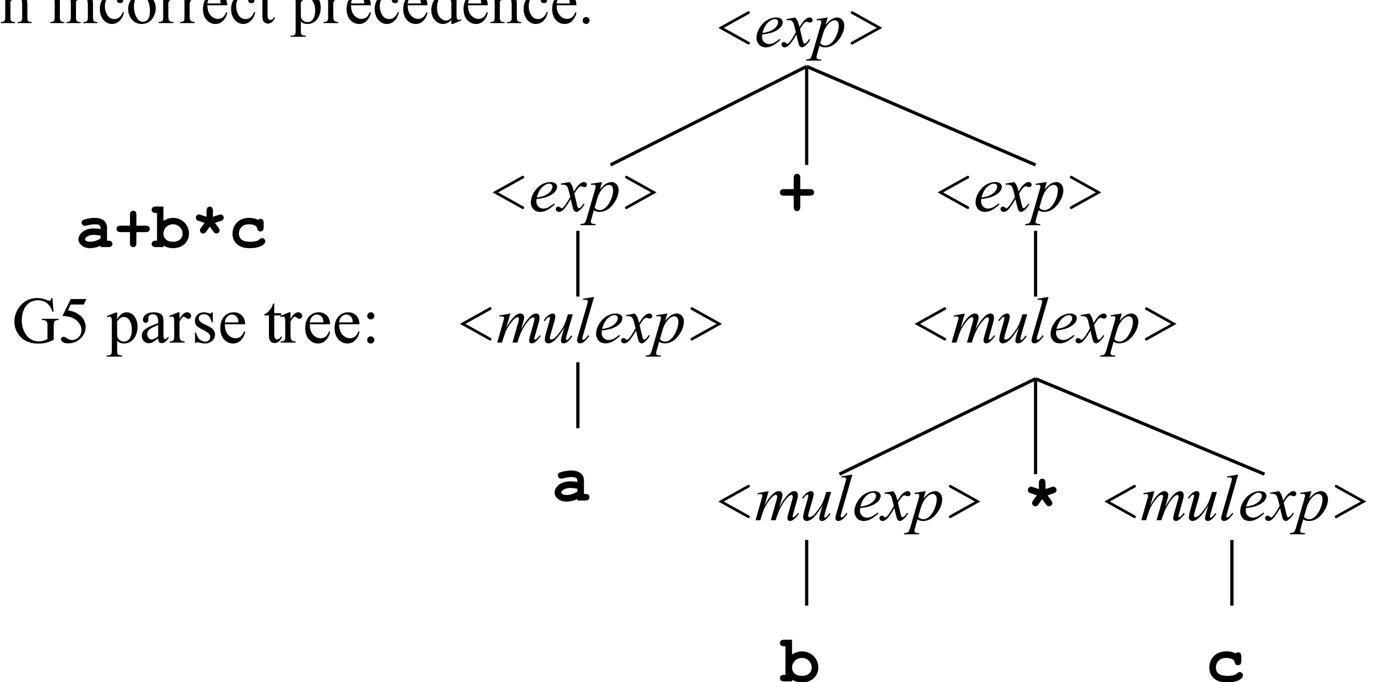
```
G5:   <exp> ::= <exp> + <exp> | <mulexp>
      <mulexp> ::= <mulexp> * <mulexp>
                 | (<exp>)
                 | a | b | c
```

- Fix precedence: Modify grammar to put ***** below **+** in the parse tree in G5, ***** is higher precedence than **+**.
- <exp> defined in terms of <mulexp> therefore higher in parse tree and lower precedence.

Correct Precedence

```
G5:  <exp> ::= <exp> + <exp> | <mulexp>
     <mulexp> ::= <mulexp> * <mulexp>
                | (<exp>)
                | a | b | c
```

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.

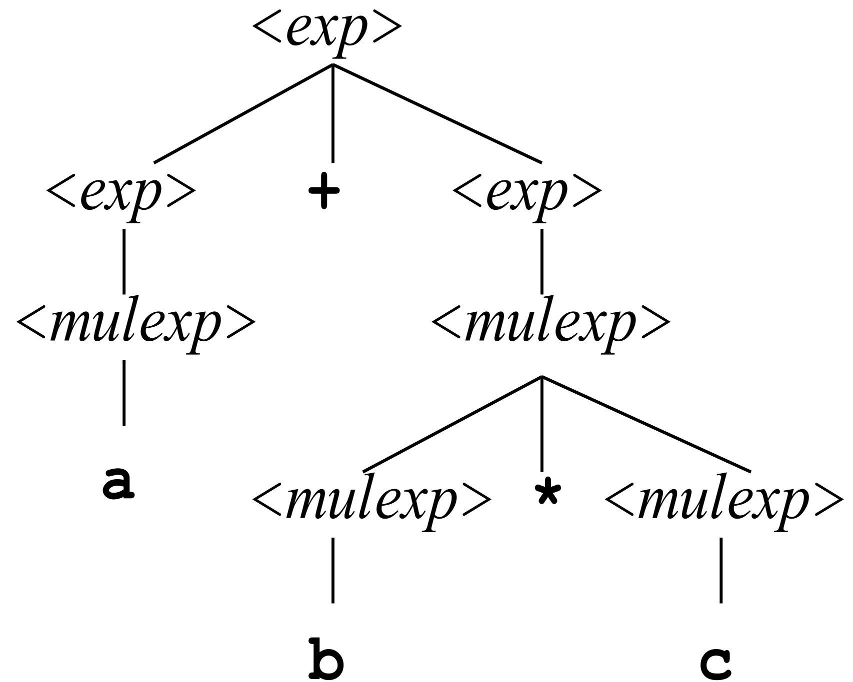


Parse

G5: <*exp*> `::=` <*exp*> **+** <*exp*> | <*mulexp*>
<*mulexp*> `::=` <*mulexp*> ***** <*mulexp*>
| **(**<*exp*>**)**
| **a** | **b** | **c**

a + b * c

<exp> = <exp> **+** <exp>
= <exp> **+** <mulexp>
= <mulexp> **+** <mulexp>
= **a +** <mulexp>
= **a +** <mulexp> ***** <mulexp>
= **a + b * c**



Exercise 1

<exp> `::=` *<exp>* **`-`** *<exp>* | *<mulexp>*
<mulexp> `::=` *<mulexp>* **`*`** *<mulexp>* | *1* | *2* | *3*

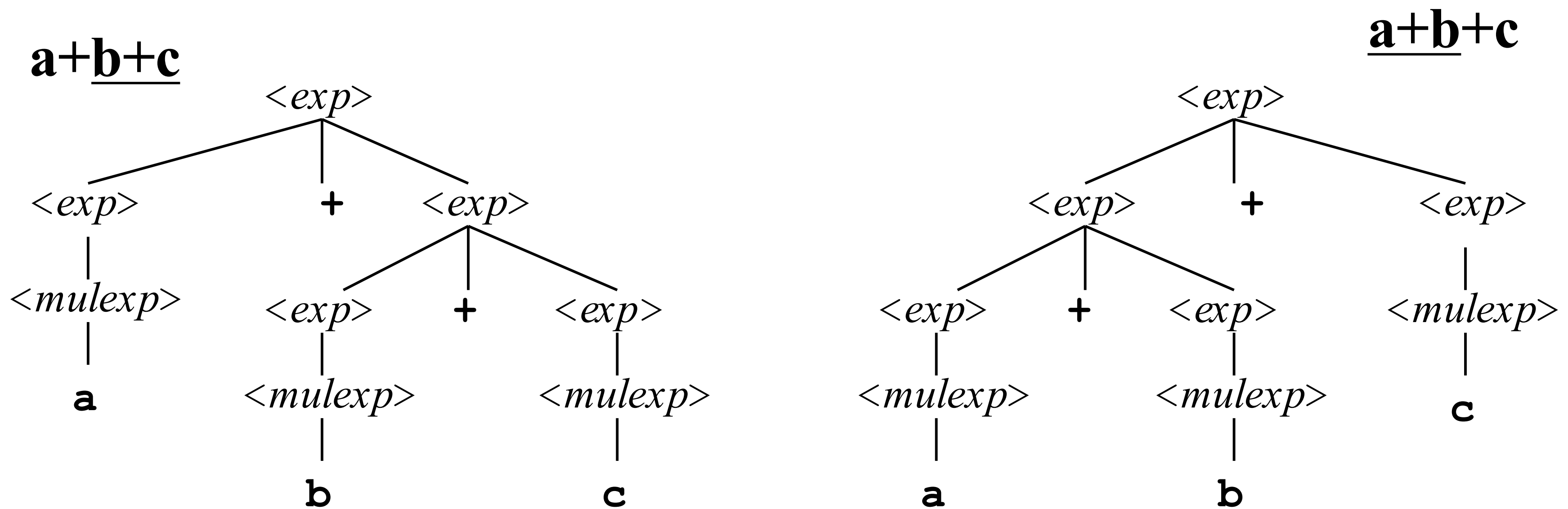
For the language defined by the grammar above, give the parse tree and select the value of each:

1. `1-2*3`? -5 3 –3 7
2. `1-2-3`? -4 2 –2 0
3. `1*2*3`? -4 6 –6 0
4. `2*3-1*2`? -4 2 –2 4

Outline

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

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.*

```
G5:  <exp>    ::= <exp> + <exp> | <mulexp>
     <mulexp> ::= <mulexp> * <mulexp>
                | (<exp>)
                | a | b | c
```

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

Associativity Examples

- C

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

- F#

  `3-2-1`— most operators are left-associative
  `1::2::[]` — right-associative (list builder)

- Fortran

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

Associativity In The Grammar

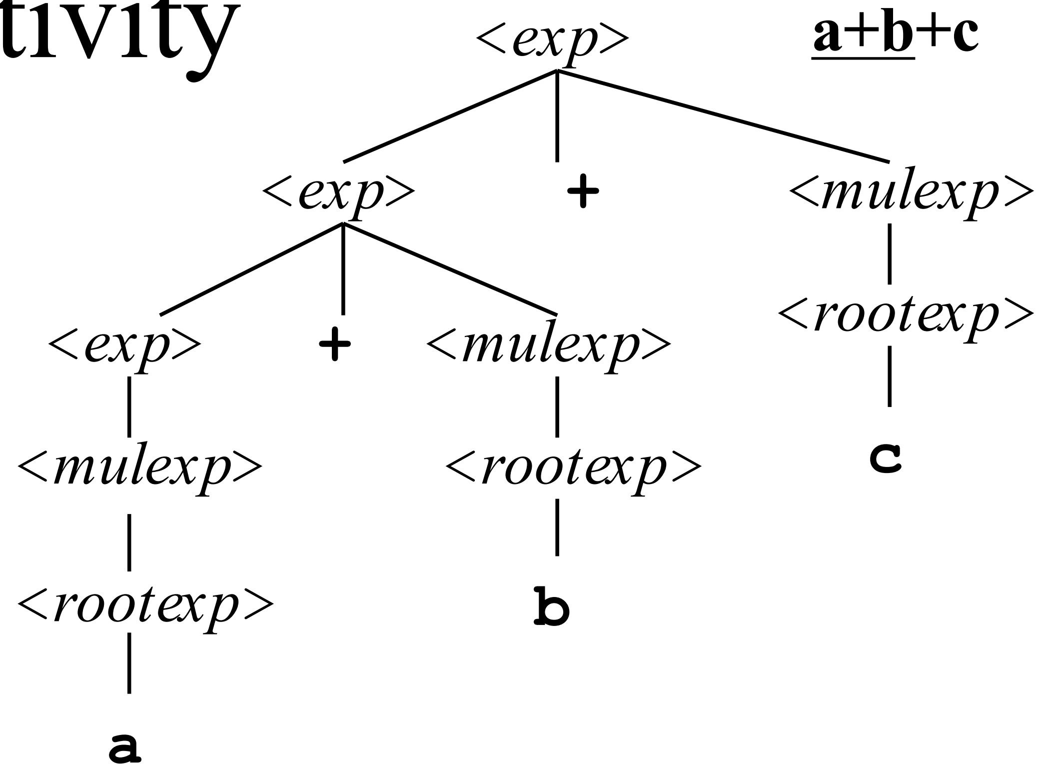
```
G5:   <exp> ::= <exp> + <exp> | <mulexp>
      <mulexp> ::= <mulexp> * <mulexp>
                 | (<exp>)
                 | a | b | c
```

```
G6:   <exp> ::= <exp> + <mulexp> | <mulexp>
      <mulexp> ::= <mulexp> * <rootexp> | <rootexp>
      <rootexp> ::= (<exp>) | a | b | c
```

- Fix associativity: Modify the grammar to make trees of **+'** s grow down to the left (and likewise for ***'** s) in G6.
- G5 ambiguous: <exp> + <exp> is right and left recursive.
- G6: **Left recursive** rule alone defines **left associativity**, ***<exp>*** + *<mulexp>* and ***<mulexp>*** * *<rootexp>*

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.



G6: *<exp>* **::=** *<exp>* **+** *<mulexp>* | *<mulexp>*
<mulexp> **::=** *<mulexp>* ***** *<rootexp>* | *<rootexp>*
<rootexp> **::=** **(***<exp>***)** | **a** | **b** | **c**

Exercise 2

```
<exp> ::= <exp> - <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>
<rootexp> ::= 1 | 2 | 3
```

For the language defined by the grammar, give the parse tree and the value of the following:

a) `1-2*3`? -5 3 –3 7

b) `1-2-3`? -4 2 –2 0

c) `1-3-2*3`? -8 6 –6 -12

Exercise 2 Continued

Starting with this grammar:

```
G6: <exp> ::= <exp> + <mulexp> | <mulexp>
    <mulexp> ::= <mulexp> * <rootexp> | <rootexp>
    <rootexp> ::= (<exp>) | a | b | c | d | e
```

d) Replace + and * with - and / with the customary precedence and associativity. Parse a/b-c/d

e) Add a left-associative operator % to G6 between the + and * in precedence. Parse a % b * c + d

f) Add a right-associative **=** operator to G6, at lower precedence than + or *. Hint: Add a one line definition.

g) Parse under definition f.

   a) a = b + c

   b) a = b = c * d + e

d) Replace + and * with - and / with the customary precedence and associativity. Parse a/b-c/d

G6: *<exp>* ::= *<exp>* **-** *<mulexp>* | *<mulexp>*
<mulexp> ::= *<mulexp>***/***<rootexp>* | *<rootexp>*
<rootexp> ::= **(***<exp>***)** | **a** | **b** | **c** | **d** | **e**

e) Add a left-associative operator % to G6 between the + and * in precedence. a % b * c + d

G6: *<exp>* ::= *<exp>* **+** *<PCexp>* | *<PCexp>*
<PCexp> ::= *<PCexp>***%***<mulexp>* | *<mulexp>*
<mulexp> ::= *<mulexp>********<rootexp>* | *<rootexp>*
<rootexp> ::= **(***<exp>***)** | **a** | **b** | **c** | **d** | **e**

f) Add a right-associative **=** operator to G6, at lower precedence than + or *. Hint: Add a one line definition.

G6: *<assign>* ::= *<exp>* **=** *<assign>* | *<exp>*
<exp> ::= *<exp>* **+** *<mulexp>* | *<mulexp>*
<mulexp> ::= *<mulexp>* ***** *<rootexp>* | *<rootexp>*
<rootexp> ::= **(***<exp>***)** | **a** | **b** | **c** | **d** | **e**

Outline

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

Issue #3: Ambiguity

- G4 was *ambiguous*: it generated more than one parse tree for the same string
- Fixing the associativity (in G5) and precedence (in G6) 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...

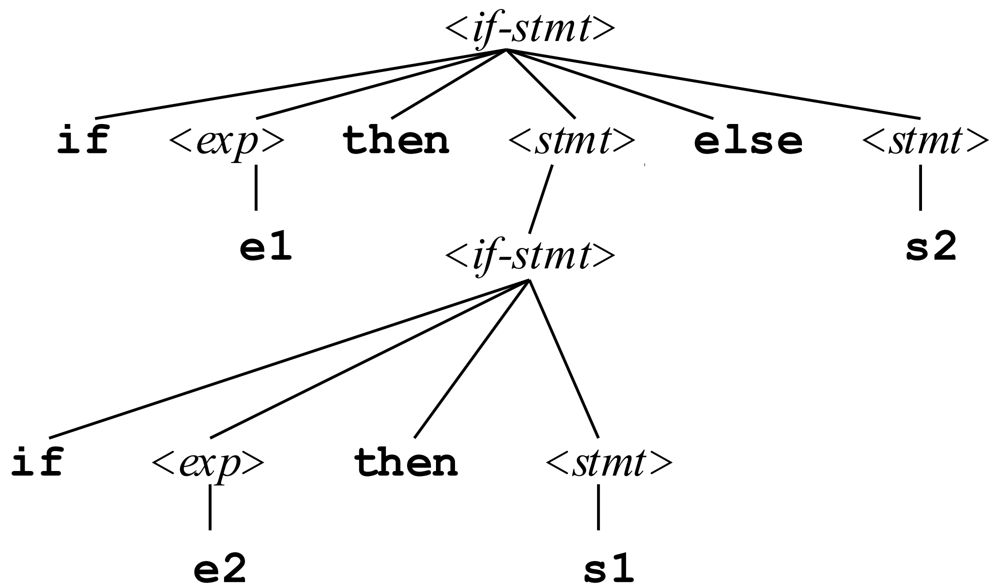
Dangling Else In Grammars

```
<stmt> ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <stmt> else <stmt>
            | if <expr> then <stmt>
<expr> ::= e1 | e2
```

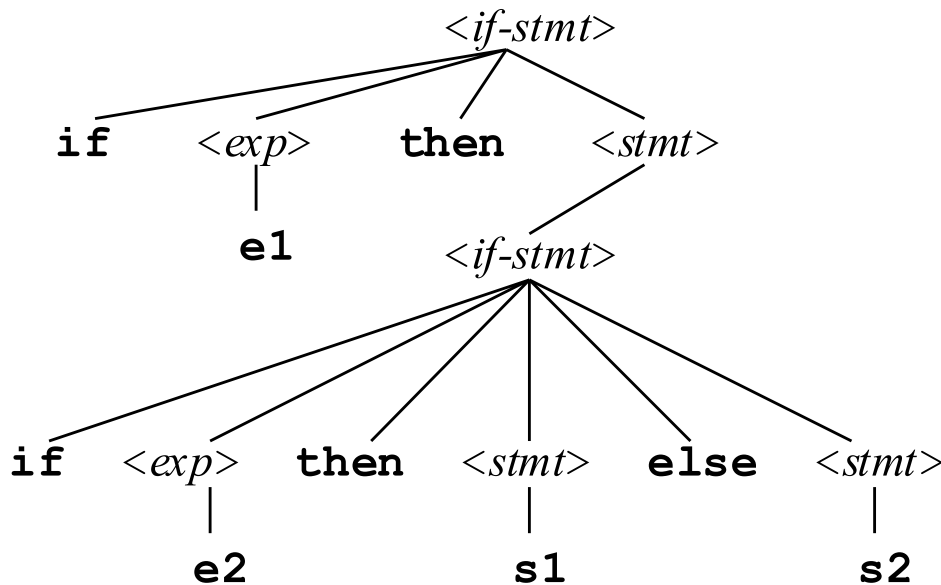
This grammar has a classic "dangling-else ambiguity." The statement we want to derive is

`if e1 then if e2 then s1 else s2`

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



Ambiguous if-then-else parse trees



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

Eliminating The Ambiguity

```
<stmt> ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <stmt> else <stmt>
            | if <expr> then <stmt>
<expr> ::= e1 | 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 <*full-stmt*> that generates everything <*stmt*> generates.

- Except that it can not generate **if** statements with no **else**:

`if e1 then if e2 then s1 else s2`

G7:

<full-stmt> ::= *<full-if>* | **s1** | **s2**
<full-if> ::= **if** *<expr>* **then** *<full-stmt>* **else** *<full-stmt>*

<stmt> ::= *<if-stmt>* | **s1** | **s2**
<if-stmt> ::= **if** *<expr>* **then** *<full-stmt>* **else** *<stmt>* |
if *<expr>* **then** *<stmt>*
<expr> ::= **e1** | **e2**

- Use the new *<full-stmt>* non-terminal here.

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

G7:

<full-stmt> **::=** *<full-if>* | **s1** | **s2**
<full-if> **::=** **if** *<expr>* **then** *<full-stmt>* **else** *<full-stmt>*

<stmt> **::=** *<if-stmt>* | **s1** | **s2**
<if-stmt> **::=** **if** *<expr>* **then** *<full-stmt>* **else** *<stmt>* |
if *<expr>* **then** *<stmt>*
<expr> **::=** **e1** | **e2**

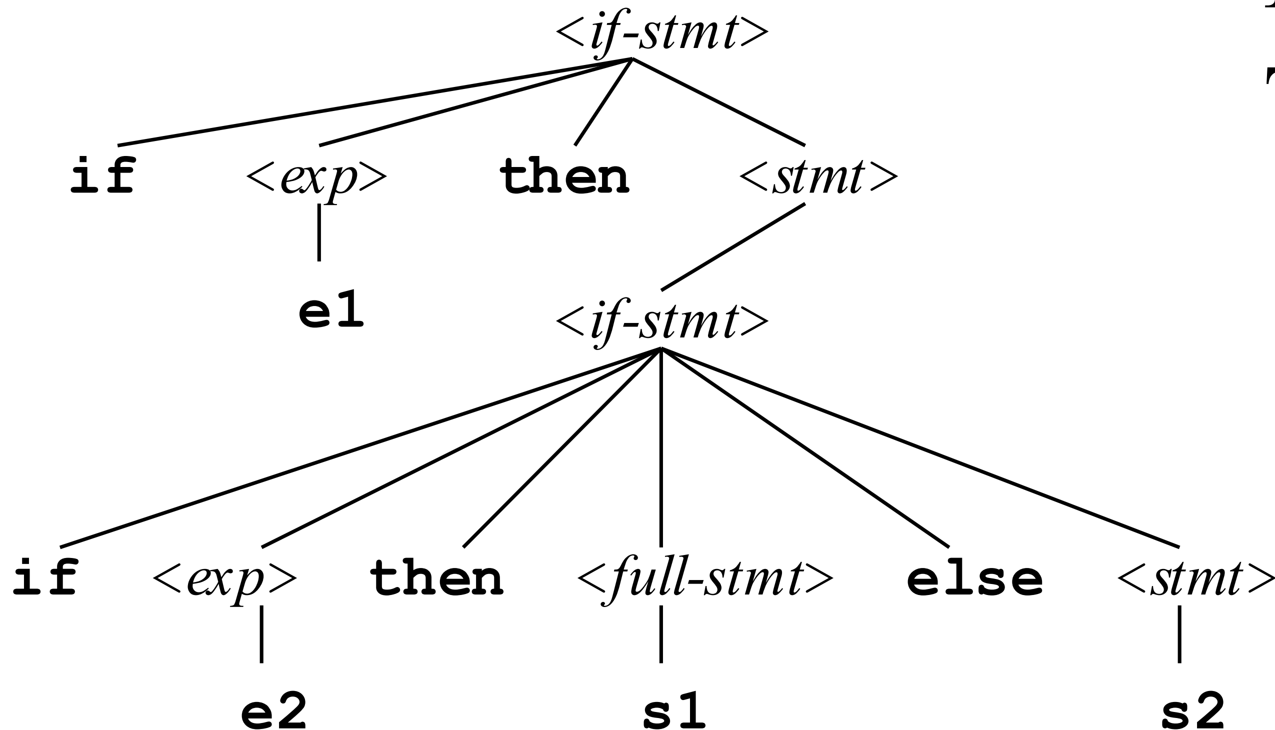
if e1 then if e2 then s1 else s2

<stmt> = <if-stmt>
= **if** *<expr>* **then** *<stmt>*
= **if** *<expr>* **then** *<if-stmt>*
= **if** *<expr>* **then** **if** *<expr>* **then** *<full-stmt>* **else** *<stmt>*
= **if** e1 **then** **if** *<expr>* **then** *<full-stmt>* **else** *<stmt>*
= **if** e1 **then** **if** e2 **then** *<full-stmt>* **else** *<stmt>*
= **if** e1 **then** **if** e2 **then** s1 **else** *<stmt>*
= **if** e1 **then** **if** e2 **then** s1 **else** s2

```
<full-stmt> ::= <full-if> | s1 | s2
<full-if>  ::=  if <expr> then <full-stmt> else <full-stmt>
<stmt>  ::=    <if-stmt> | s1 | s2
<if-stmt>  ::=  if <expr> then <full-stmt> else <stmt> |
                if <expr> then <stmt>
<expr> ::= e1 | e2
```

Correct Parse Tree

```
if e1 then if e2 then s1 else s2
```



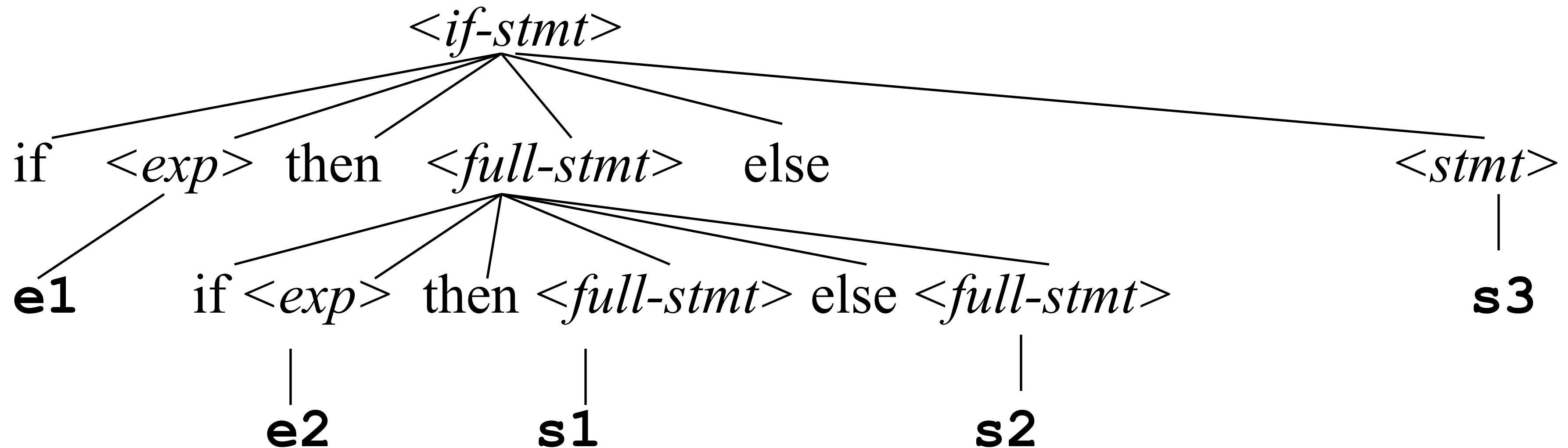
<*full-stmt*> ::= **if** <*expr*> **then** <*full-stmt*> **else** <*full-stmt*> |
s1 | **s2** | **s3**

<*stmt*> ::= <*if-stmt*> | **s1** | **s2** | **s3**

<*if-stmt*> ::= **if** <*expr*> **then** <*full-stmt*> **else** <*stmt*> |
if <*expr*> **then** <*stmt*>

<*expr*> ::= **e1** | **e2**

```
if e1 then if e2 then s1 else s2 else s3
```



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

Exercise 3

Give the parse tree for each <*stmt*>.

```
<full>  ::= if <expr> then <full> else <full> |
            s1 | s2 | s3
<stmt>  ::= if <expr> then <full> else <stmt> |
            if <expr> then <stmt> |
            s1 | s2 | s3
<expr>  ::= e1 | e2
```

```
1. if e1 then s1
2. if e1 then if e2 then s1
3. if e1 then if e2 then s1 else s2
4. if e1 then if e2 then s1 else s2 else s3
5. if e1 then s1 else if e2 then s2 else s3
```

Clearer Styles

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

Good: correct indentation

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

Better: use of a block reinforces the structure

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

Best: pairing **`if`** and **`endif`** defines structure, allowing compiler to check for correctness.

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 (and Visual Basic) requires each **`if`** statement to be terminated with an **`endif`**

Outline

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

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
- Reverse Polish Notation and Evaluation

EBNF and Parse Trees

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

Two Camps

- Some people use EBNF loosely:
  - Use {} anywhere it helps
  - Add a paragraph of text dealing with ambiguities, associativity of operators, etc.
- Other people use EBNF strictly:
  - Use <*exp*> ::= <*mulexp*> {`+` <*mulexp*>} only for left-associative operators
  - Use recursive rules for right-associative operators: <*expa*> ::= <*expb*> [`=` <*expa*>]

About Syntax Diagrams

- Similar problem: what parse tree is generated?
- As in loose 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
- Reverse Polish Notation and Evaluation

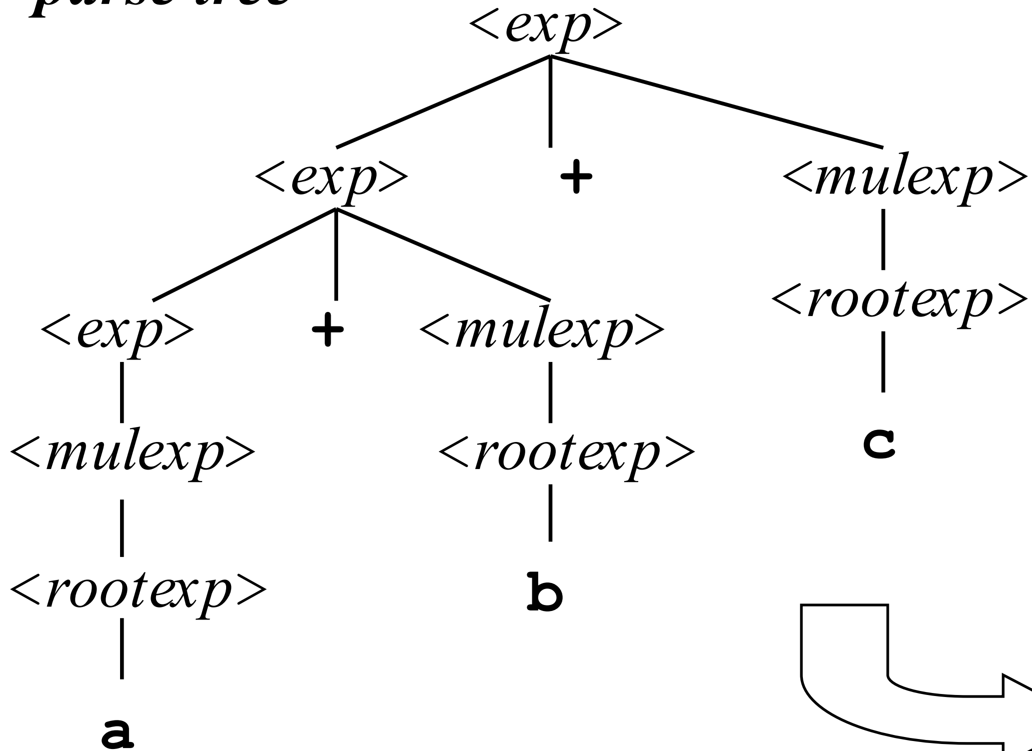
Full-Size Grammars

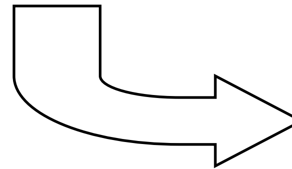
- In any realistically large language, there are many non-terminals
- Especially true when in the cluttered but unambiguous form needed by parsing tools
- Extra non-terminals guide construction of unique parse tree
- Once parse tree is found, such non-terminals are no longer of interest

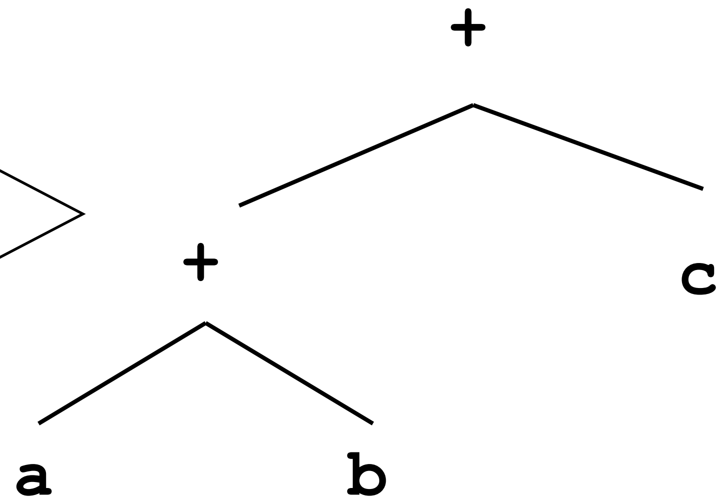
Abstract Syntax Tree

- 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

Example







Abstract Syntax Tree

Outline

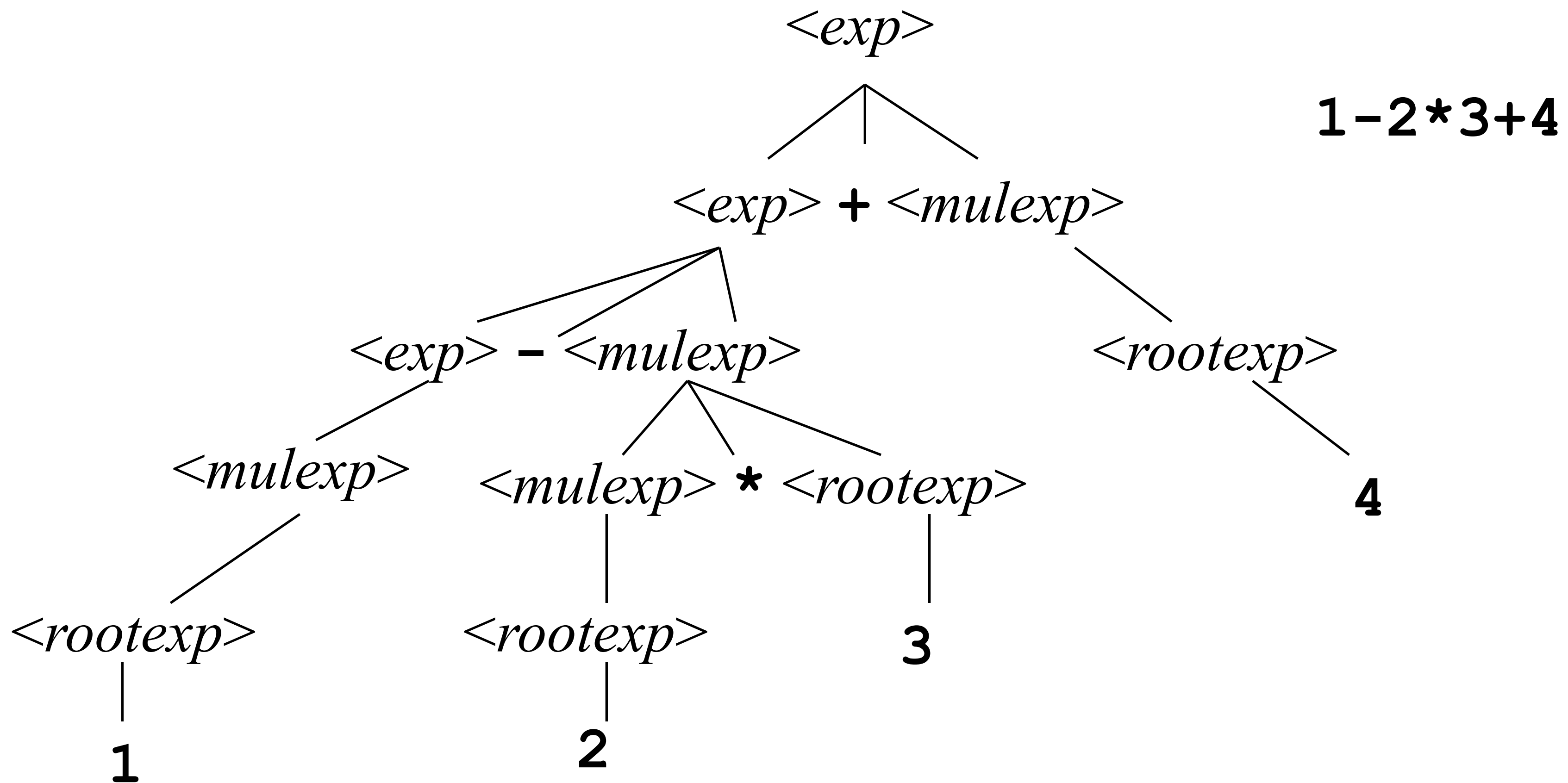
- Operators
- Precedence
- Associativity
- Other ambiguities: dangling else
- Cluttered grammars
- Parse trees and EBNF
- Abstract syntax trees
- Reverse Polish Notation and Evaluation

AST to Reverse Polish Notation and Evaluation

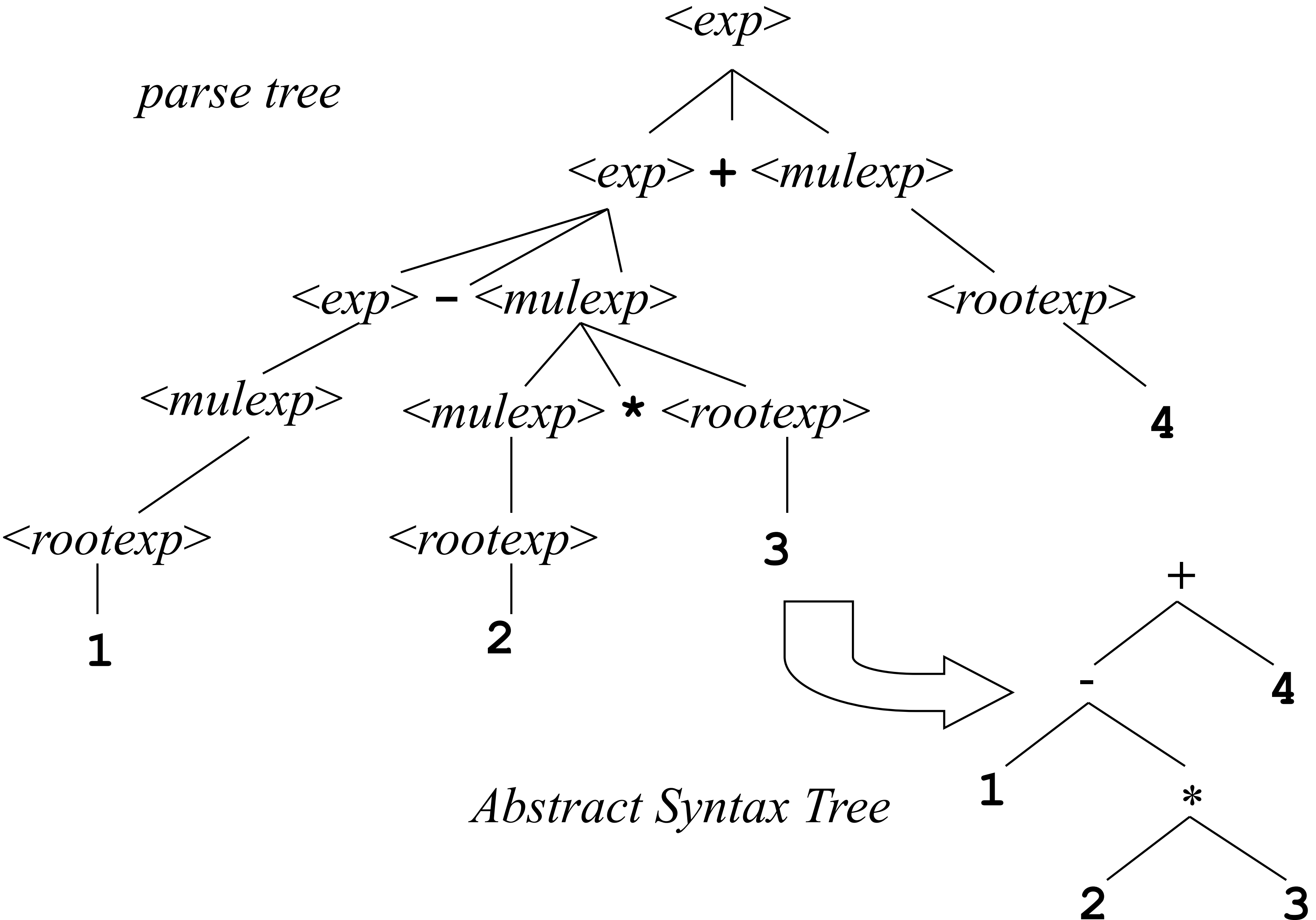
- Abstract Syntax Tree holds the infix expression operations and operands.
- Traversing the AST in *post-order* produces the RPN.
- RPN operator precedence is unambiguous.
- Operands are arranged in proper order for post-order evaluation.
- Easily evaluated on hardware using a stack.

Parse Tree: 1-2*3+4

<exp> ::= <exp> - <mulexp> | <exp> + <mulexp> | <mulexp>
<mulexp> ::= <mulexp> * <rootexp> | <rootexp>
<rootexp> ::= **1 | 2 | 3 | 4**



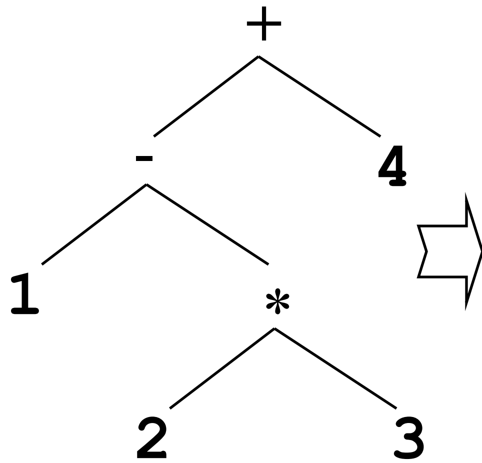
Abstract Syntax Tree: 1-2*3+4



Post-order traversal and RPN

1-2*3+4

abstract syntax tree



Postorder traversal

```
postOrder(tree t)
   if (t != null)
        postOrder( t.left )
        postOrder( t.right )
        print t.data
```

RPN

1 2 3 * - 4 +

RPN Evaluation using a Stack

1. Scan RPN input from left to right.

   1 | 2 | 3 | 4 input
   push input onto stack.

   + | - | * input
   1. pop operand2
      pop operand1
   2. push operand1 * operand2
      or
      push operand1 – operand2
      or
      push operand1 + operand2

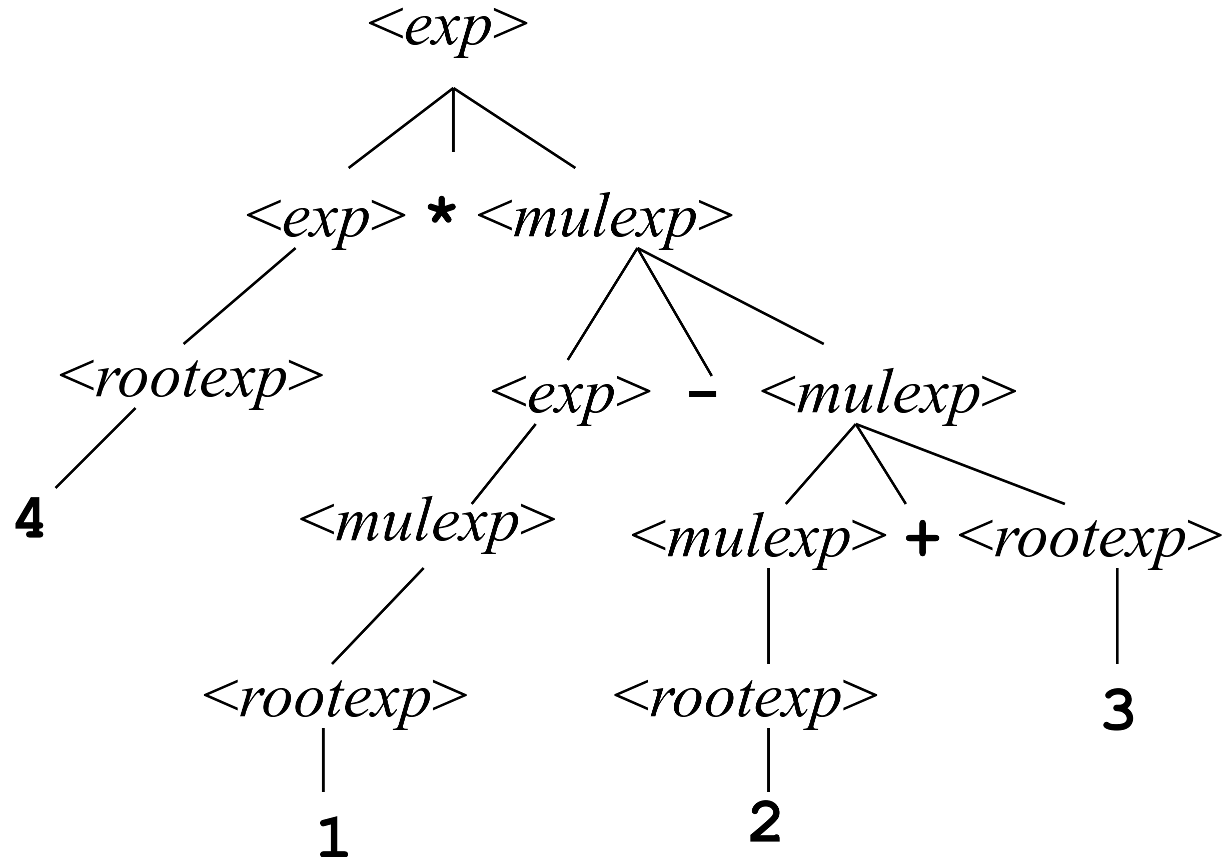
2. After RPN is scanned, the expression value is stack top.

Input	Stack	Push
1 2 3 * - 4 +	1	1
2 3 * - 4 +	1 2	2
3 * - 4 +	1 2 3	3
* - 4 +	1 6	2 * 3
- 4 +	-5	1 - 6
4 +	-5 4	4
+	-1	-5 + 4

Java Expression Parser and Evaluator

Try the Java applet to enter expressions in the form specified in the expression grammar, parse, generate RPN, then evaluate the RPN. Source code for the applet is available.

Exercise 4



1. Give the AST of the parse tree.
2. The corresponding RPN.
3. The value from the RPN evaluation.

Exercise 5

G6: *<exp>* ::= *<exp>* **+** *<mulexp>* | *<mulexp>*
<mulexp> ::= *<mulexp>* ***** *<rootexp>* | *<rootexp>*
<rootexp> ::= **(***<exp>***)** | **1** | **2** | **3** | **4** | **5**

- Given the above BNF rules, parse and generate AST for expressions:
  - 2 + 3 * 4
  - 2 + 3 * (4 + 1)
- Traverse the AST nodes in post order to produce the Reverse Polish Notation.
- Evaluate and compare with the results of the parser applet.

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