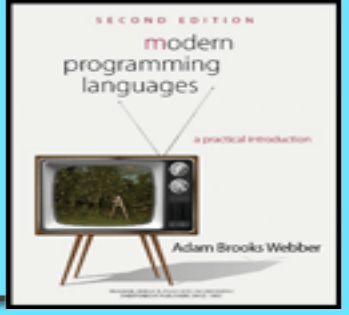
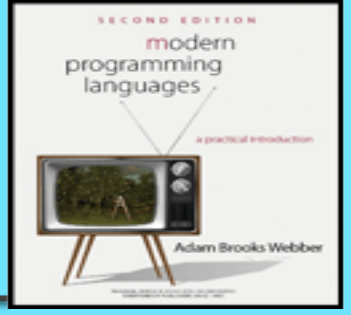


Context Free Grammars (From CS210)

Three “Equivalent” Grammars



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

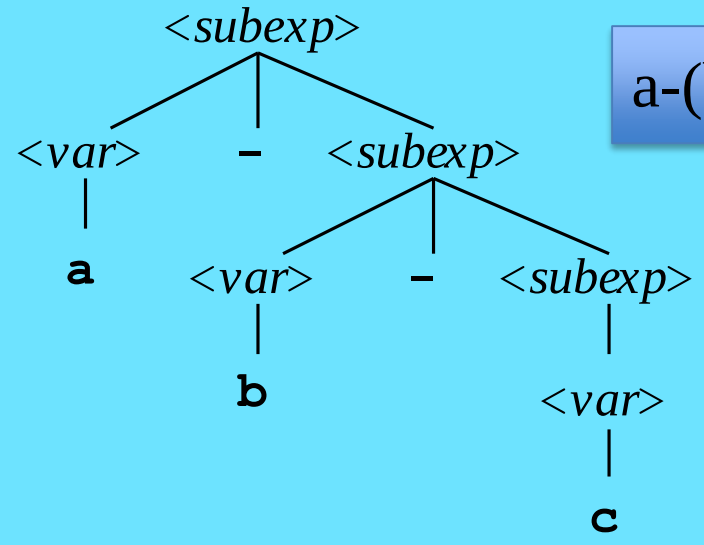
G2: <subexp> ::= <var> - <subexp> | <var>
<var> ::= a | b | c

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

Dr. BC Note:
After we cover left vs. right associative, come back to this slide as see if you can determine which G2 and G3 are.

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...

G2 parse tree:

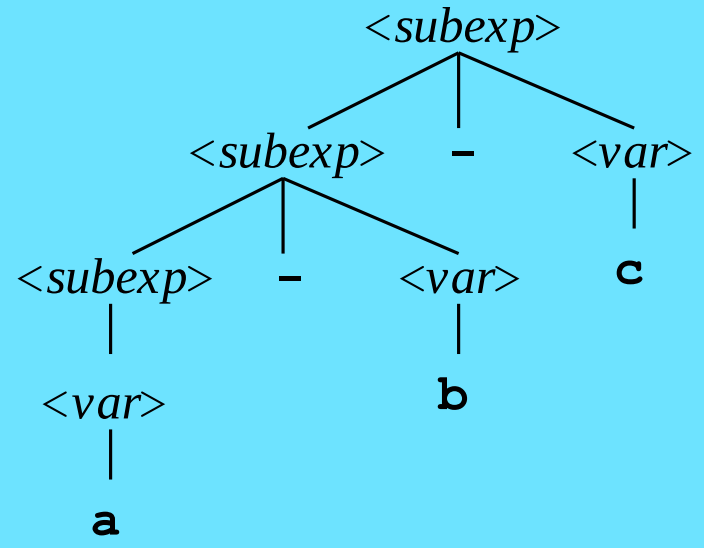


a-(b-c)

1-(2-3) = 2

Dr. BC Note:
To prove that a language is ambiguous, choose ANY sting in the language for which you can produce 2 parse tress and draw the trees.

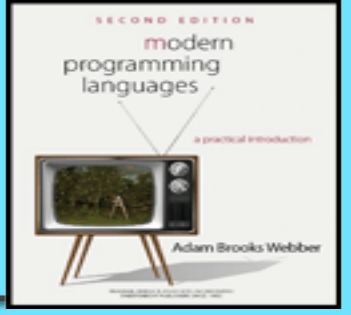
G3 parse tree:



(a-b)-c

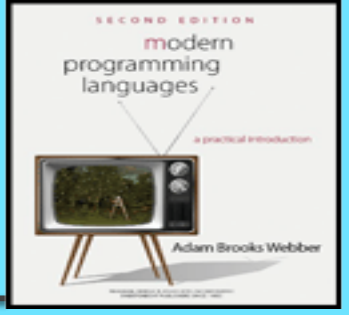
(1-2)-3 = -4

Why Parse Trees Matter



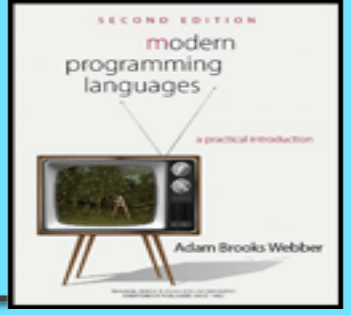
- We want the structure of the parse tree to correspond to the semantics of the string it generates
- This makes grammar design much harder: we're interested in the structure of each parse tree, not just in the generated string

Outline



- ✧ Operators
- ✧ Precedence
- ✧ Associativity
- ✧ Other ambiguities: dangling else
- ✧ Abstract syntax trees

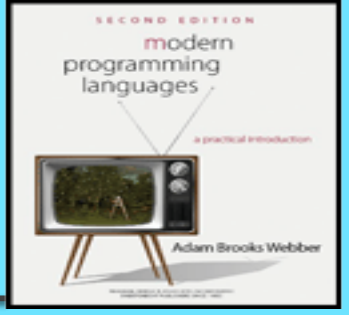
Operators



- 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

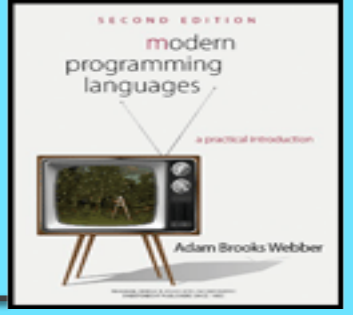
Dr. BC Note: Remember we treated single character operators a bit differently in Assignment 1.

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`

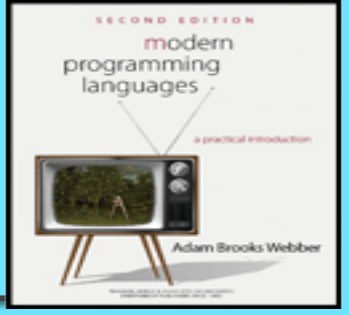
Outline



- ✧ Operators
- ✧ Precedence – PEMDAS (Parentheses, Exponents, */,+-)
- ✧ Associativity
- ✧ Other ambiguities: dangling else
- ✧ Abstract syntax trees

Dr. BC Note:
Bison has tools that will do Associativity and Precedence for you, but you are not allowed to use them.

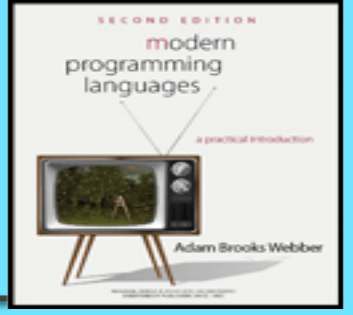
Working Grammar

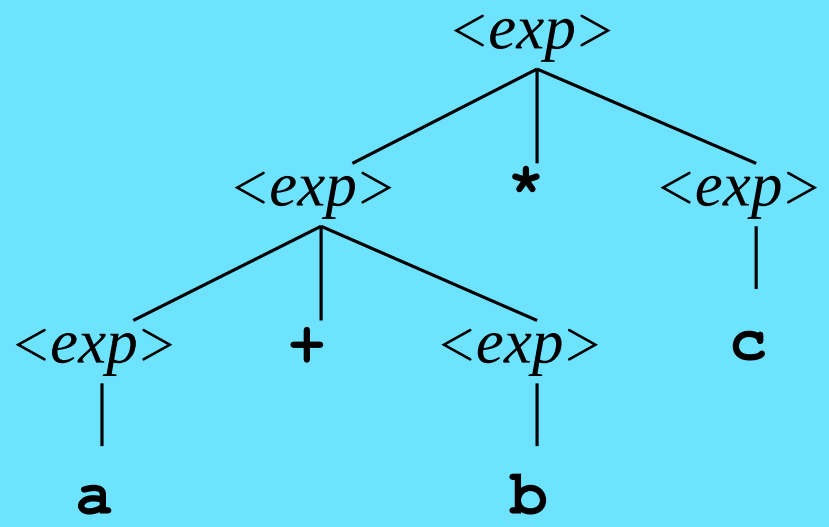


G4: <*exp*> **::=** <*exp*> **+** <*exp*>
| <*exp*> ***** <*exp*>
| **(**<*exp*>**)**
| **a** | **b** | **c**

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

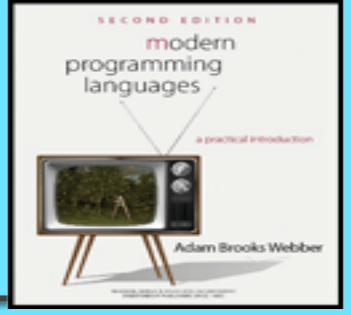
Issue #1: Precedence





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



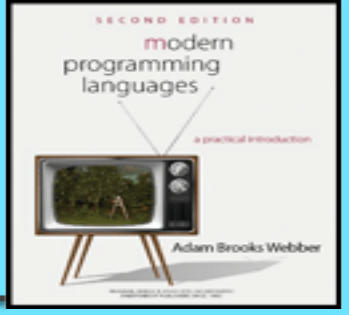
- Applies when the order of evaluation is not completely decided by parentheses

> Remember PEMDAS?
> Parentheses, Exponents, Multiplication and Division (from left to right), Addition and Subtraction (from left to right).

- 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?)

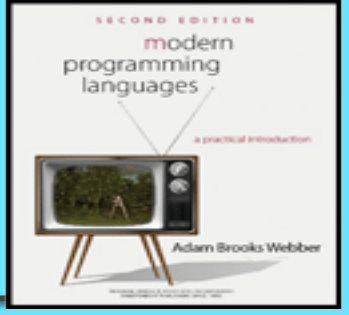
```
a <= 0 or 100 <= a
```

Error! 0 or 100 is evaluated first leaving :
a <= 100 <= a (assuming bitwise or)

✧ Smalltalk (1 level for all binary operators)

```
a + b * c
```

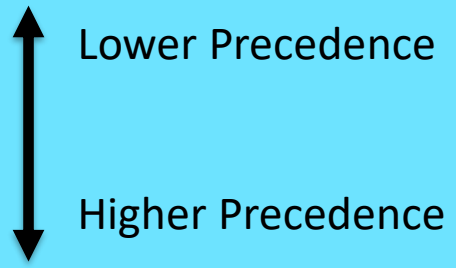
Precedence In The Grammar



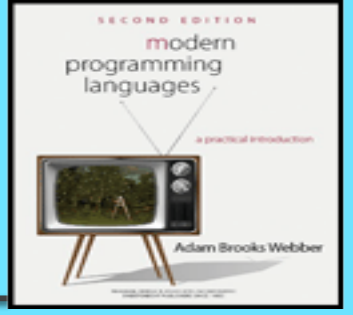
G4: <*exp*> ::= <*exp*> **+** <*exp*>
| <*exp*> ***** <*exp*>
| **(**<*exp*>**)**
| **a** | **b** | **c**

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

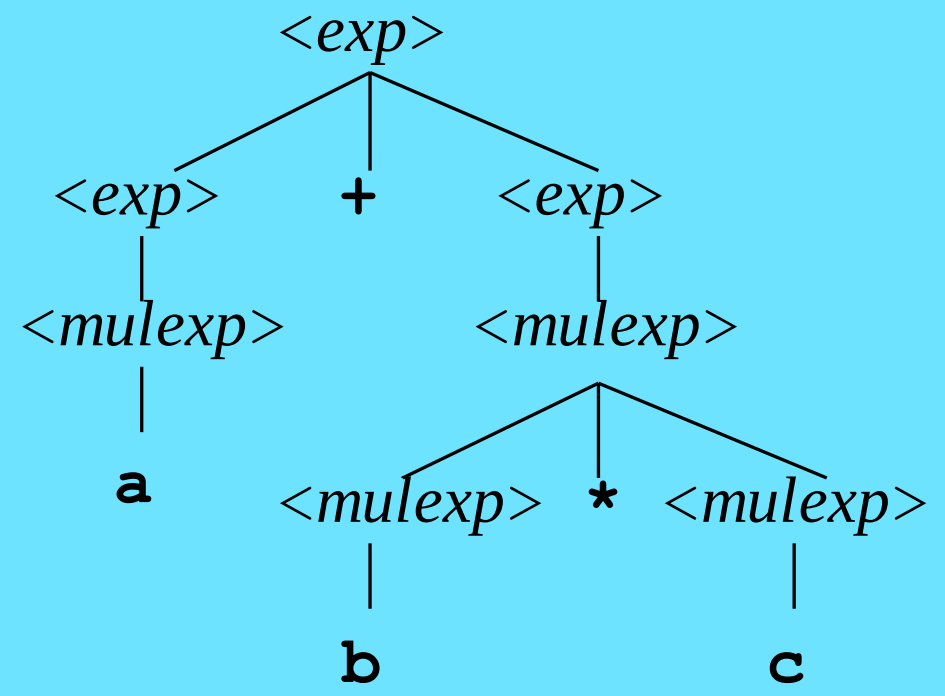
G5: <*exp*> ::= <*exp*> **+** <*exp*> | <*mulexp*>
<*mulexp*> ::= <*mulexp*> ***** <*mulexp*>
| **(**<*exp*>**)**
| **a** | **b** | **c**



Correct Precedence

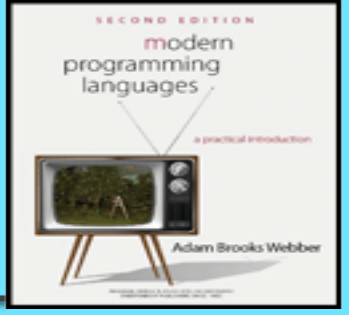


G5 parse tree:



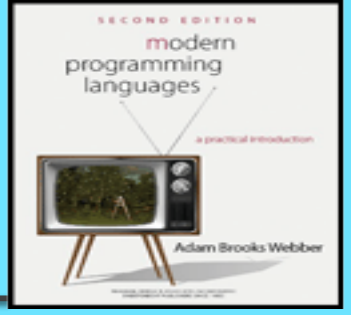
Our new grammar generates this tree for **`a+b*c`**. It generates the same language as before, but no longer generates parse trees with incorrect precedence.

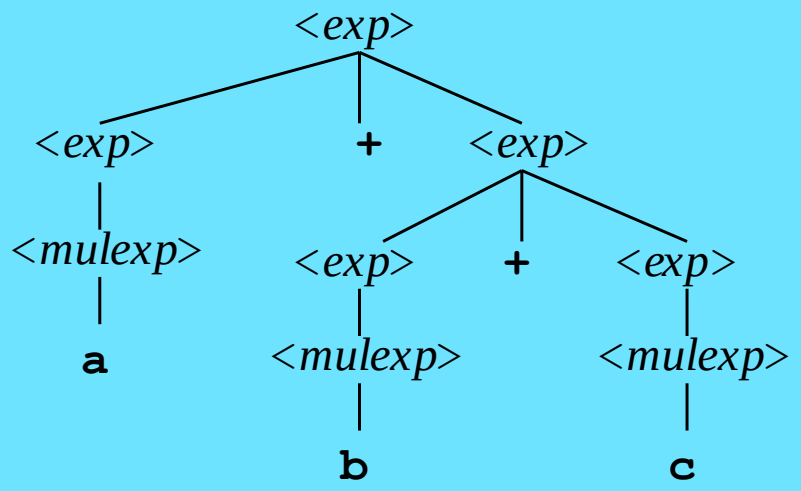
Outline

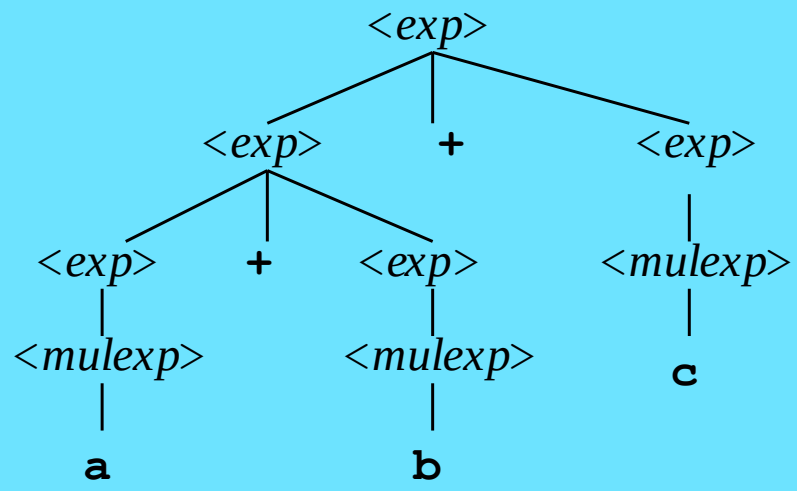


- ✧ Operators
- ✧ Precedence
- ✧ Associativity
- ✧ Other ambiguities: dangling else
- ✧ Abstract syntax trees

Issue #2: Associativity

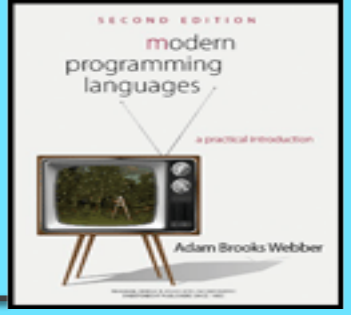






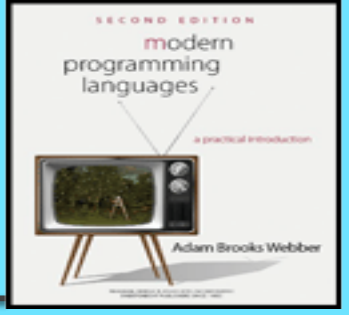
Our grammar G5 generates both these trees for **a+b+c**.
The first one is not the usual convention for operator *associativity*.

Operator Associativity



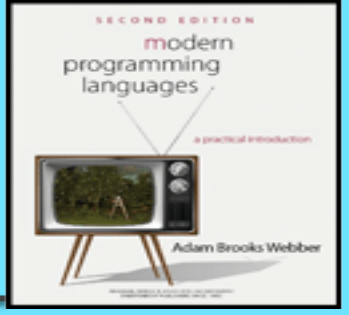
- Applies when the order of evaluation is not decided by parentheses or by precedence
- *Left-associative* operators group left to right: **`a+b+c+d = ((a+b)+c)+d`**
- *Right-associative* operators group right to left: **`a+b+c+d = a+(b+(c+d))`**
- Most operators in most languages are left-associative,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

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)

Associativity In The Grammar



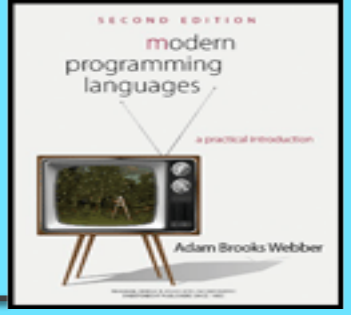
G5: <*exp*> **::=** <*exp*> **+** <*exp*> | <*mulexp*>
<*mulexp*> **::=** <*mulexp*> ***** <*mulexp*>
| **(**<*exp*>**)**
| **a** | **b** | **c**

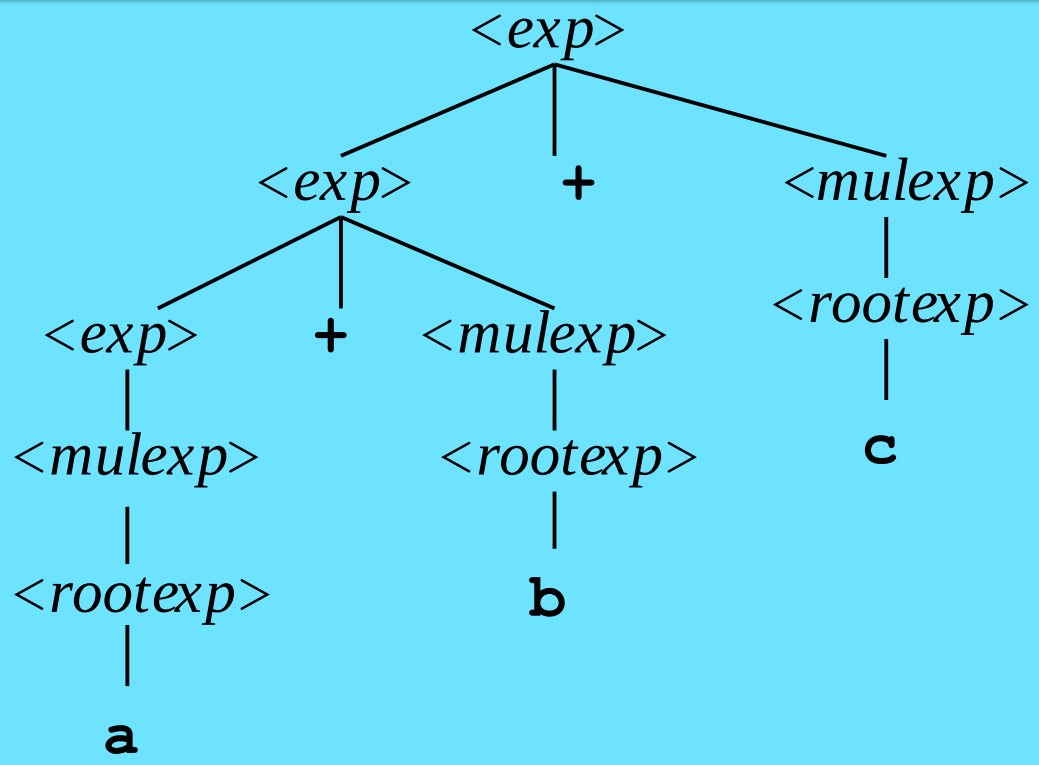
To fix the associativity problem, we modify the grammar to make trees of **+**s grow down to the left (and likewise for *****s)

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

Notice: Only recursive on the left so this is left associative

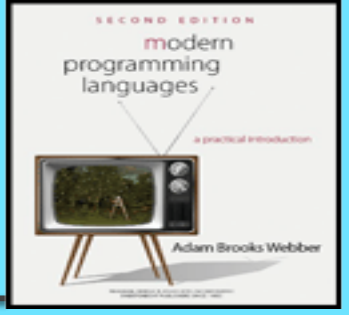
Correct Associativity





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.

Practice

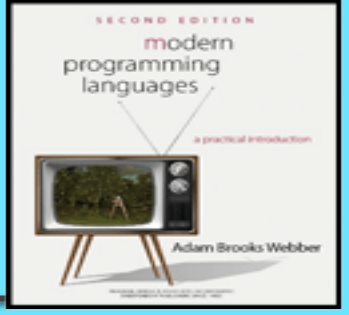


Starting with this grammar:

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

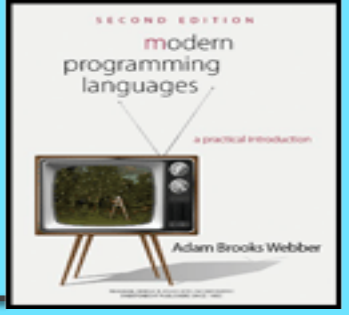
1.) Add a left-associative **&** operator, at lower precedence than any of the others
2.) Then add a right-associative ****** operator, at higher precedence than any of the others except ()

Quick Quiz: How is this language ambiguous?



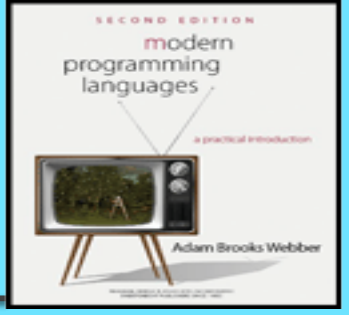
G4: <*exp*> **::=** <*exp*> **+** <*exp*>
| <*exp*> ***** <*exp*>
| <*exp*>
| **a** | **b** | **c**

Outline



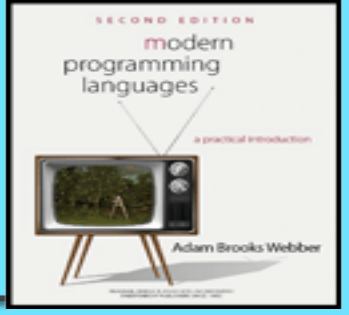
- ✧ Operators
- ✧ Precedence
- ✧ Associativity
- ✧ Other ambiguities: dangling else
- ✧ Abstract syntax trees

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



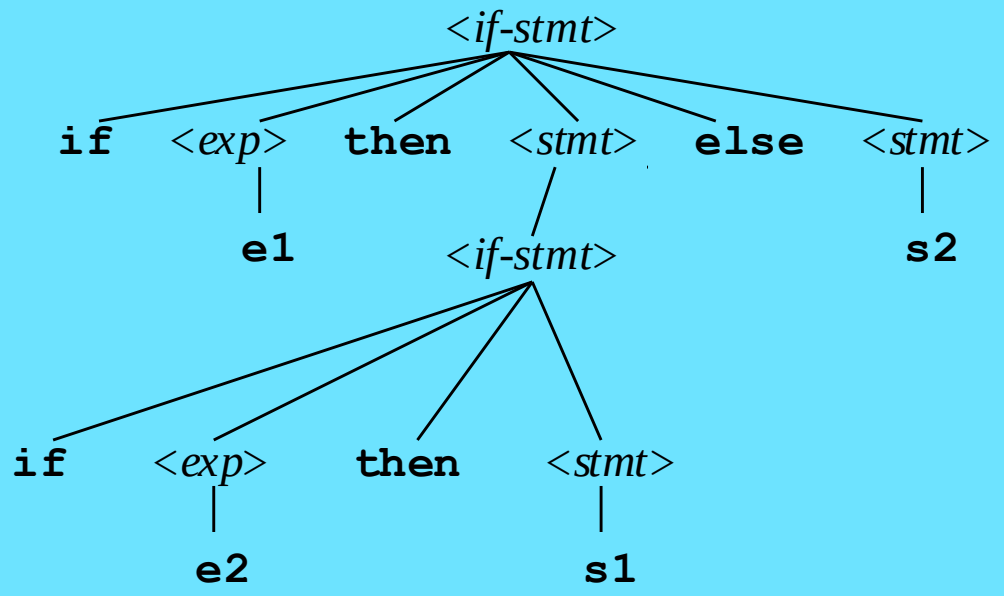

```
<stmt>   ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <stmt> else <stmt>
            | if <expr> then <stmt>
<expr>   ::= e1 | e2
```

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

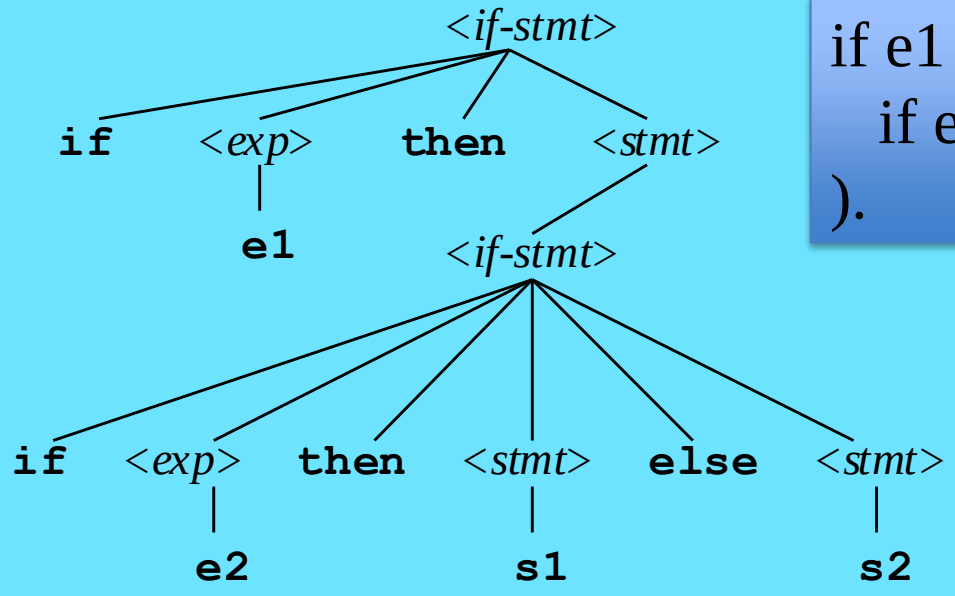
```
if e1 then if e2 then s1 else s2
```

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

Which if/then is this the else for?



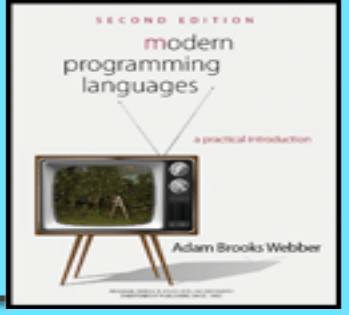
if e1 then
(if e2 then s1)
else s2



if e1 then (
if e2 then s1 else s2
).

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

Eliminating The Ambiguity



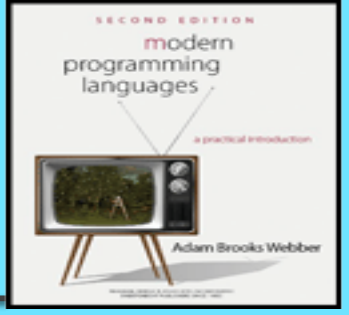

```
<stmt>  ::= <if-stmt> | s1 | s2
<if-stmt> ::= if <expr> then <stmt> else <stmt>
            | if <expr> then <stmt>
<expr>  ::= e1 | e2
```

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`**:

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

Modern Programming Languages, 2nd ed.

Eliminating The Ambiguity

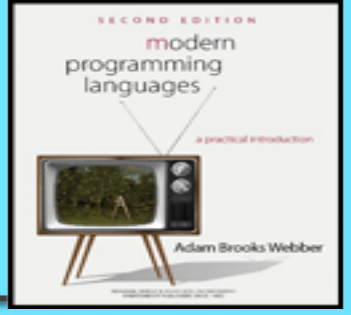


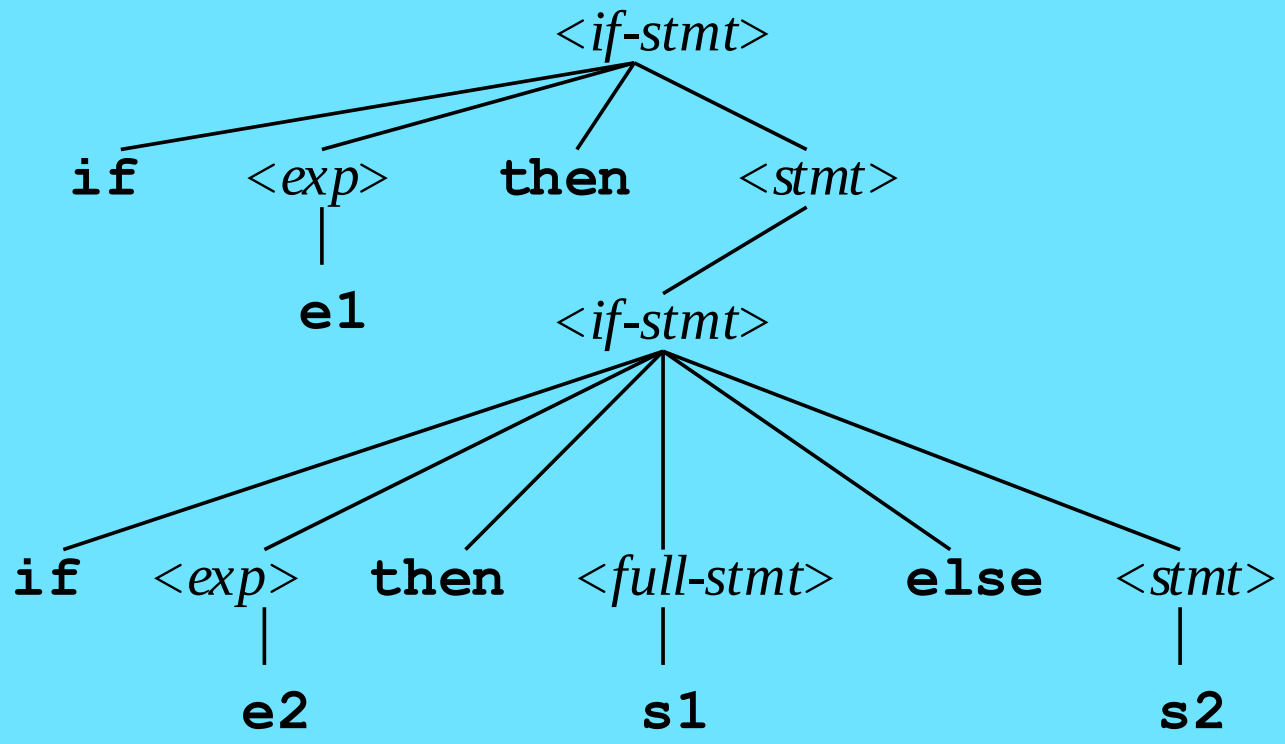

```
<stmt>      ::= <if-stmt> | s1 | s2
<if-stmt>   ::= if <expr> then <full-stmt> else <stmt>
              | if <expr> then <stmt>
<expr>      ::= e1 | e2
<full-stmt> ::= <full-if> | s1 | s2
<full-if>   ::= if <expr> then <full-stmt> else <full-stmt>
```

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.

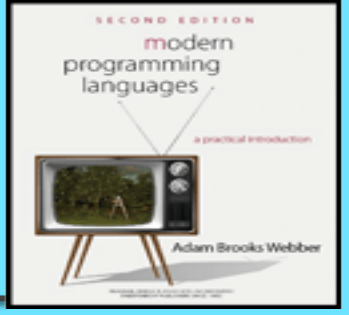
Correct Parse Tree





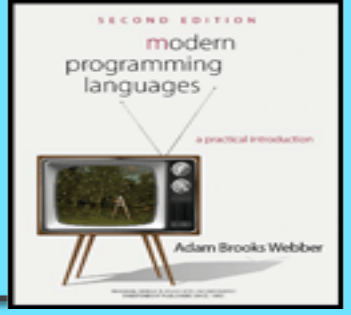
Modern Programming Languages, 2nd ed.

Outline



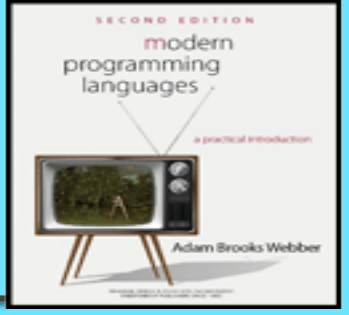
- ✧ Operators
- ✧ Precedence
- ✧ Associativity
- ✧ Other ambiguities: dangling else
- ✧ Abstract syntax trees

Full-Size Grammars



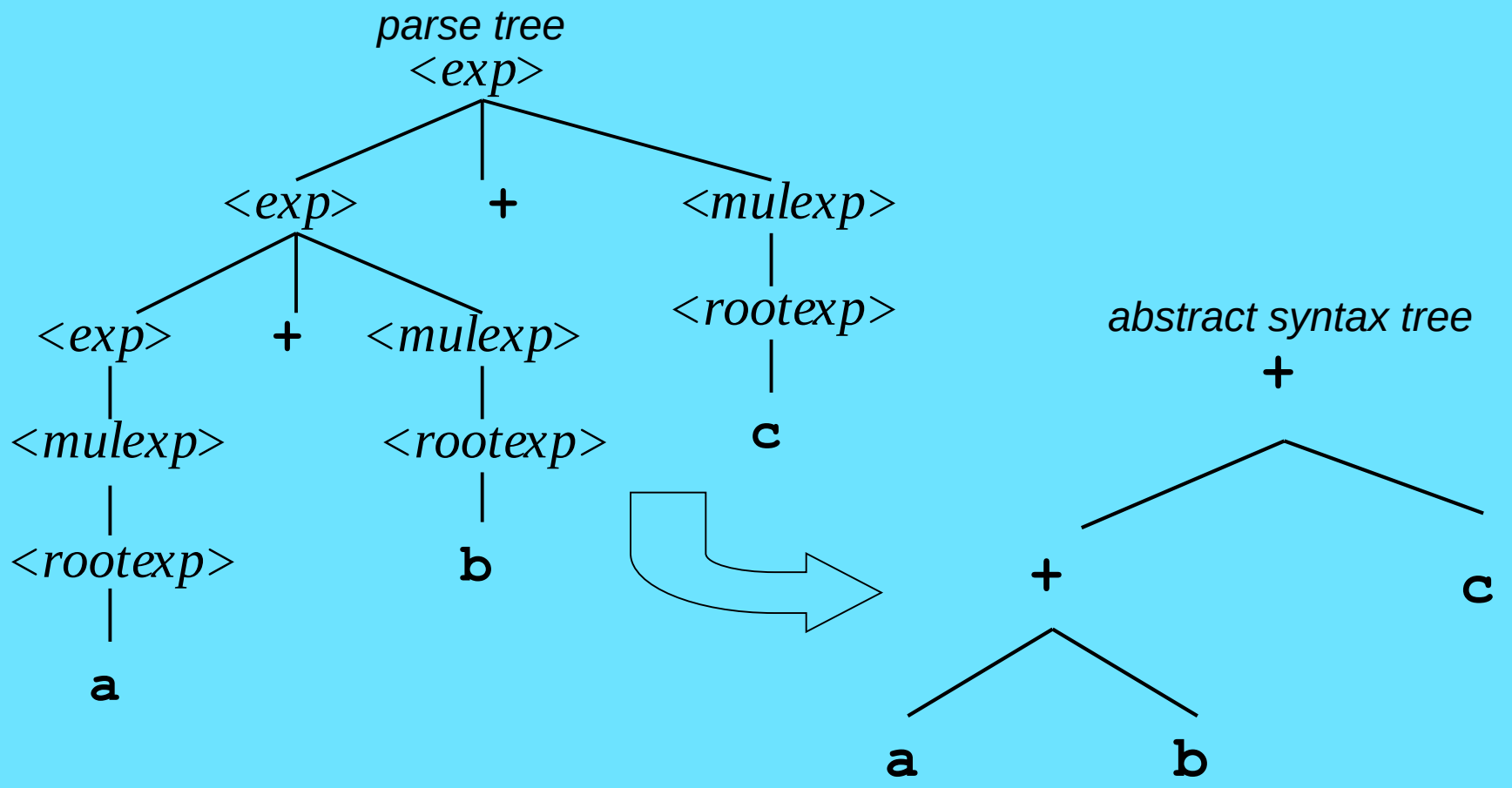
- In any realistically large language, there are many non-terminals
- Especially true when in the cluttered but unambiguous form needed by parsing tools
- Extra non-terminals guide construction of a unique parse tree
- Once a parse tree is found, such non-terminals are no longer of interest

Abstract Syntax Tree

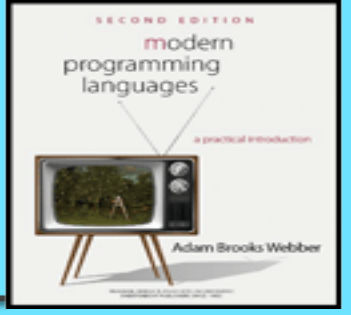


- Language systems usually store an abbreviated version of the parse tree called the *abstract syntax tree*
- Details are implementation-dependent
- Usually, there is a node for every operation, with a subtree for every operand


SECOND EDITION
modern
programming
languages
a practical introduction
Adam Brooks Webber


parse tree
<exp>
<exp>
+
<mulexp>
<exp>
+
<mulexp>
<rootexp>
<mulexp>
<rootexp>
c
<rootexp>
b
a
abstract syntax tree
+
+
c
a
b

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