Where Syntax Meets Semantics
Three “Equivalent” Grammars

G1: \( <\text{subexp}> ::= a \mid b \mid c \mid <\text{subexp}> - <\text{subexp}> \)

G2: \( <\text{subexp}> ::= <\text{var}> - <\text{subexp}> \mid <\text{var}> \)
\( <\text{var}> ::= a \mid b \mid c \)

G3: \( <\text{subexp}> ::= <\text{subexp}> - <\text{var}> \mid <\text{var}> \)
\( <\text{var}> ::= a \mid b \mid c \)

These grammars all define the same language: the language of strings that contain one or more \textit{a}s, \textit{b}s or \textit{c}s separated by minus signs. But...
G2 parse tree:

```
<subexp>
  <var> - <subexp>
    a
    <var> - <subexp>
      b
      <var>
        c
```

G3 parse tree:

```
<subexp>
  - <var>
    <subexp>
      - <var>
        c
        <var>
          b
          <var>
            a
```
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

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 \texttt{a}, \texttt{b} and \texttt{c}
Issue #1: Precedence

Our grammar generates this tree for $a + b \times c$. 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
  \[ a + b \times c = a + (b \times c) \]
Precedence Examples

- C (15 levels of precedence—too many?)
  \[ a = b < c ? * p + b * c : 1 << d () \]

- Pascal (5 levels—not enough?)
  \[ a <= 0 \text{ or } 100 <= a \quad \text{Error!} \]

- Smalltalk (1 level for all binary operators)
  \[ a + b * c \]
Precedence In The Grammar

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

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

G5: \( <exp> ::= <exp> + <exp> | <mulexp> \)
    \( <mulexp> ::= <mulexp> * <mulexp> \)
    \( \quad | \quad (<exp>) \)
    \( \quad | \quad a \quad | \quad b \quad | \quad c \)
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
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**
  
  \[a \ll b \ll c\] — most operators are left-associative
  \[a = b = 0\] — right-associative (assignment)

- **ML**
  
  \[3 - 2 - 1\] — most operators are left-associative
  \[1 : : 2 : : \text{nil}\] — right-associative (list builder)

- **Fortran**
  
  \[a / b * c\] — most operators are left-associative
  \[a ** b ** c\] — right-associative (exponentiation)
Associativity In The Grammar

G5:  \( \langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{exp} \rangle \mid \langle \text{mulexp} \rangle \)

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

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

G6:  \( \langle \text{exp} \rangle ::= \langle \text{exp} \rangle + \langle \text{mulexp} \rangle \mid \langle \text{mulexp} \rangle \)

\( \langle \text{mulexp} \rangle ::= \langle \text{mulexp} \rangle \ast \langle \text{rootexp} \rangle \mid \langle \text{rootexp} \rangle \)

\( \langle \text{rootexp} \rangle ::= (\langle \text{exp} \rangle) \mid a \mid b \mid c \)
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.
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
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

\[
\begin{align*}
<\text{stmt}> & ::= <\text{if-stmt}> \mid s1 \mid s2 \\
<\text{if-stmt}> & ::= \text{if} <\text{expr}> \text{ then } <\text{stmt}> \text{ else } <\text{stmt}> \\
& \quad \mid \text{if} <\text{expr}> \text{ then } <\text{stmt}> \\
<\text{expr}> & ::= e1 \mid e2
\end{align*}
\]

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

\[
\text{if } e1 \text{ then if } e2 \text{ then } s1 \text{ else } s2
\]

and the next slide shows two different parse trees for it...
Most languages that have this problem choose this parse tree: \textit{else} goes with nearest unmatched \textit{then}
Eliminating The Ambiguity

\[<stmt> ::= <if-stmt> \mid s1 \mid s2\]

\[<if-stmt> ::= if <expr> then <stmt> else <stmt> \]
\[\mid if <expr> then <stmt>\]

\[<expr> ::= e1 \mid e2\]

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

\[<full-stmt> ::= <full-if> \mid s1 \mid s2\]

\[<full-if> ::= if <expr> then <full-stmt> else <full-stmt>\]
Eliminating The Ambiguity

\[ \text{<stmt>} ::= \text{<if-stmt>} \mid \text{s1} \mid \text{s2} \]
\[ \text{<if-stmt>} ::= \text{if <expr> then <full-stmt>} \text{ else <stmt>} \]
\[ \text{<expr>} ::= \text{e1} \mid \text{e2} \]

Then we use the new non-terminal here.

The effect is that the new grammar can match an \textbf{else} part with an \textbf{if} part only if all the nearer \textbf{if} parts are already matched.
Correct Parse Tree

```
if <exp> then <stmt>
  
  e1

if <exp> then <stmt>
  
  e2

if <exp> then <full-stmt> else <stmt>
  
  s1
  s2
```
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.
Practice

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

What is the value of `a` after this fragment executes?
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 \(<exp> ::= <mulexp> \{ + <mulexp> \}\) 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 \(a+a+a\)?
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
    \(<exp> ::= <mulexp> \{+ <mulexp>\}\) will be used only for left-associative operators
  - Use explicitly recursive rules for anything unconventional:
    \(<expa> ::= <expb> [ = <expa> ]\)
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

- 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.
parse tree

abstract syntax tree
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