3. Control Flow
Expressions, Structure, Sequencing, Selection, Iteration, Recursion
Expressions

- Basics
- Precedence
- Language comparisons
- Associativity
- Order of evaluation
- Assignment
Operators applied to operands

Syntax examples

- $a \times (b + c)$  C, C++, Java, Python
- $(* a (+ b c))$  Lisp, Scheme
- $a \times b + c$  APL
- $a b c + *$  Forth

Only infix notation requires parentheses (Lisp uses them anyway, so that it can apply operators to more than two operands)

For languages using infix notation, some parentheses can be eliminated using rules of *precedence* and *associativity*
Rules for “tightness” of grouping in the absence of parentheses

Example

In C, \( a + b * c \) means \( a + (b * c) \), because * groups “more tightly” than (has precedence over) +

The more operators, the more need for rules

Language tradeoff

- Rich set of operators and rules
- Simplicity (easy to remember)
## Language Comparisons

<table>
<thead>
<tr>
<th>Fortran</th>
<th>Pascal</th>
<th>C</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>not</td>
<td><code>++</code>, <code>--</code> (post-inc., dec.)</td>
<td><code>abs</code> (absolute value), <code>not</code>, <code>**</code></td>
</tr>
<tr>
<td><code>*</code>, <code>/</code></td>
<td><code>*</code>, <code>/</code>, <code>div</code>, <code>mod</code>, <code>and</code></td>
<td><code>*</code> (binary), <code> </code>/<code>, </code>mod<code>, </code>rem`</td>
<td><code>*</code>, <code>/</code>, <code>mod</code>, <code>rem</code></td>
</tr>
<tr>
<td><code>+</code>, <code>-</code> (unary and binary)</td>
<td><code>+</code>, <code>-</code> (unary and binary), <code>or</code></td>
<td><code>+</code>, <code>-</code> (binary)</td>
<td><code>+</code>, <code>-</code> (binary)</td>
</tr>
<tr>
<td><code>.eq., .ne., .lt., .le., .gt., .ge.</code> (comparisons)</td>
<td><code>&lt;</code>, <code>&lt;=</code>, <code>&gt;=</code>, <code>=</code>, <code>&lt;=</code>, <code>=</code>, <code>IN</code></td>
<td><code>&lt;</code>, <code>&lt;=</code>, <code>&gt;=</code> (inequality tests)</td>
<td><code>=</code>, <code>/=</code>, <code>&lt;</code>, <code>&lt;=</code>, <code>&gt;</code>, <code>&gt;=</code></td>
</tr>
<tr>
<td><code>.not.</code></td>
<td></td>
<td><code>==</code>, <code>!=</code> (equality tests)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>&amp;</code> (bit-wise and)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>^</code> (bit-wise exclusive or)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>`</td>
<td>` (bit-wise inclusive or)</td>
</tr>
<tr>
<td><code>.and.</code></td>
<td></td>
<td><code>&amp;</code> (logical and)</td>
<td>and, or, xor (logical operators)</td>
</tr>
<tr>
<td><code>.or.</code></td>
<td></td>
<td>`</td>
<td>` (logical or)</td>
</tr>
<tr>
<td><code>.eqv., .neqv.</code> (logical comparisons)</td>
<td></td>
<td><code>?:</code> (if...then...else)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>=</code>, <code>+=</code>, <code>-=</code> <code>**=</code>, <code>/=</code>, <code>%=</code>, <code>&gt;&gt;=</code>, <code>&lt;&lt;=</code>, <code>&amp;</code>, `</td>
<td>=<code>, </code>^=`</td>
</tr>
</tbody>
</table>

Copyright © 2009 Elsevier
Rules for grouping of equal-precedence operators in the absence of parentheses

Examples (in C)

- Expression $a + b + c$ means $(a + b) + c$, because + groups left to right
- Expression $a - b + c$ means $(a - b) + c$ because + and - have equal precedence and group left to right
- Expression $a = b = c$ means $a = (b = c)$, because = groups right to left
Order of Evaluation

- Not the same as precedence or associativity
- Example:
  
  \[ a * b + f(b) + f(b) * b \]

- Assuming C-style precedence
  - \( f(b) \) will be added to result of \( a * b \)
  - \( f(b) * b \) will be added to result of above

- But many possibilities exist for order of evaluation
Possibilities for order of evaluation
- Strictly left to right (but respecting precedence)
- Strictly right to left (but respecting precedence)
- Evaluate $f(b)$ twice, immediately before needed
- Evaluate $f(b)$ once at outset

Results can be different in presence of function side effects

A function side effect is a change made in the caller or global environment by executing the function (and only that – not apparent to caller)
In both cases, the answer is $30$: left-to-right evaluation is specified by language rules.
Unlike the two previous examples, the result when the compiler decides this time is not 30 (or 28), but 25 (using g++ compiler)

```cpp
/*
   An example of effect of order of evaluation when a function has a side effect
*/

#include <iostream>
using namespace std;
int b = 1; // global variable
int f(int x)
{
    int y = x + 5;
    ++b; // side effect
    return y;
}
int main()
{
    // Letting compiler decide
    int a = 3;
    int z = a * b + f(b) + f(b) * b;
    cout << z << endl;

    // Left to right
    b = 1;
    int z1 = a * b;
    int z2 = f(b);
    int z3 = f(b) * b;
    z = z1 + z2 + z3;
    cout << z << endl;

    // Right to left
    b = 1;
    z1 = f(b) * b;
    z2 = f(b);
    z3 = a * b;
    z = z1 + z2 + z3;
    cout << z << endl;
    return 0;
}
```
Assignment—Syntax

- **Examples**
  - `a = b`  
    Fortran, C, C++, Java, Python
  - `a := b`  
    Algol, Pascal, Clu, Ada
  - `a ← b`  
    APL, Smalltalk
  - `(SET! a b)`  
    Scheme (Lisp: SETQ)
  - `MOVE b TO a.`  
    Cobol

- Can be a *statement* or an *expression*
  - Expression has a value
  - *Example:* `cout << (a = b);` assigns value of `b` to `a and prints that value`
Assignment—Semantics

- Value model
  - Identifier names *container for value*
  - Assignment makes *copy of value*
  - Copied value not affected by future changes

- Reference model
  - Identifier names *reference to container for value*
  - Assignment makes *copy of reference*
  - Future changes affect both
- Identifier names a container for a value

- Assignment *copies the value*  
  \[a = b;\]

Before

\[
\begin{array}{c}
a \quad 1 \\
\end{array}
\quad \begin{array}{c}
b \quad 3 \\
\end{array}
\]

After

\[
\begin{array}{c}
a \quad 3 \\
\end{array}
\quad \begin{array}{c}
b \quad 3 \\
\end{array}
\]

- Subsequent assignment to \(a\) or \(b\) does not affect the other
Reference Model

- Identifier names a reference to a container

- Assignment copies the reference
  
  \[a = b;\]

- Subsequent assignment to \(a\) or \(b\) affects the other (i.e., the value referred to)
Changes in flow of control are necessary
Changes in flow of control can be hard to follow
Avoid unrestricted control flow
Structured programming: use only constructs that
- Allow determination of flow from program text
- Do not require simulation of execution (flow charts)
Idea is to group a set of statements to be executed one after the other

(Seen earlier in connection with blocks)

Sequencing can be accomplished syntactically by

- Using special notation for grouping
  - Either `begin` and `end` or `{ and }` or the like

- Using fully bracketed constructs
  - Such as `if - then - else - end if`

- Making indentation count (as in Python)
Selection

- **if**
  - Syntax – dangling else *vs.* fully bracketed notation
  - Short-circuit evaluation of Boolean test expression

- **case** (or **switch**)
  - True selection or glorified **goto**
Test at top *(while)*
Test at bottom *(do)*
Counter-controlled *(for)*
  - Scope of control variable
  - Semantics of loop bounds
“Infinite” loops
Iterators
Language issue: simplicity *vs.* expressiveness
  - *Example:* Mid-loop exits: “loop and a half”
Recursion

- Can implement iteration
  - Without a special syntax
  - Without "side effects" (such as assignment)
- Allows elegant solutions to certain kinds of problems
- Examples of recursion usually show it to bad effect – these are bad examples
- Recursion is *not* automatically inefficient
- More later, in connection with functional languages
Examples

// Some examples of recursion in place
// of iteration

public class Recursions
{
    public static void main(String[] args)
    {
        int x = 2, y = 5;
        System.out.println(power(x, y));
        System.out.println(addSequence(x, y));
    }
    static int power(int a, int b)
    {
        if (b == 0)
            return 1;
        else
            return a * power(a, b-1);
    }
    static int addSequence(int a, int b)
    {
        if (a > b)
            return 0;
        else if (a == b)
            return a;
        else
            return a + addSequence(a + 1, b);
    }
}

# Some examples of recursion in place
# of iteration

def power(a, b):
    if b == 0:
        return 1
    else:
        return a * power(a, b-1)
def addSequence(a, b):
    if a > b:
        return 0
    elif a == b:
        return a
    else:
        return a + addSequence(a + 1, b)

x = 2
y = 5
print power(x, y)
print addSequence(x, y)