Elements of Programming

Every non-trivial programming language provides:

- Primitive expressions representing the simplest elements (ex: Int, String)
- Ways to combine expressions (ex: addition, concatenation)
- Ways to abstract expressions (introduce a name for the expression by which it can then be referred to)

Evaluation

- A non-primitive expression is evaluated as follows:
    1. Take the left most operator
    2. Evaluate its operands (left before right)
    3. Apply the operator to the operands
- A name is evaluated by replacing it with the right-hand side of its definition
- The evaluation process stops once it results in a value

Parameter and Return Types

- Primitive types are as in Java, but written capitalized:
    - ***Int*** – 32-bit integers
    - ***Long*** – 64-bit integers
    - ***Float*** – 32-bit floating point numbers
    - ***Double*** – 64-bit floating point numbers
    - ***Char*** – 16-bit Unicode characters
    - ***Short*** – 16-bit integers
    - ***Byte*** – 8-bit integers
    - ***Boolean*** – boolean values: true and false

Evaluation of Function Applications

- Applications of parameterized functions are evaluated in a similar way as operators:
    1. Evaluate all function arguments, from left to right
    2. Replace the function application by the function's right-hand side
    3. Replace the formal parameters of the function by the actual arguments

The Substitution Model

- This scheme of expression evaluation is called the *substitution model*
- The idea underlying this model is that all evaluation does is *reduce an expression to a value*
- It can be applied to all expressions, as long as they have no side effects (they are purely functional)
- The substitution model is formalized in the λ-*calculus,* which gives a foundation for functional programming

⇨ *Does every expression reduce to a value (in a finite number of steps)?*
⇨ *No, here is a counter example:* ***def loop: Int = loop***

Call-by-name and call-by-value

- Call-by-value: evaluates the function arguments first before calling the function
- Call-by-name: evaluates the function first, and then evaluates the arguments if need be
- Both strategies reduce to the same final values as long as:
    - The reduced expression consists of pure functions
    - Both evaluations terminate
- Call-by-value has the advantage that it evaluates every function argument only once
- Call-by-name has the advantage that a function argument is not evaluated if the corresponding parameter is unused in the evaluation of the function body

Question: *Say you are given the following function definition*:

def test (x: Int, y: Int) = x * x

For each of the following function applications, indicate which evaluation strategy is the fastest:

- ***test(2, 3) –*** *same number of steps*
- ***test(3+4, 8) –*** *call-by-value*
- ***test(7, 2*4) –*** *call-by-name*
- ***test(3+4, 2*4) –*** *same number of steps*