Evaluation and Operators

Classes and Substitutions

Question: *How is an instantiation of the class $C(e_1, \ldots e_m)$ evaluated?*
Answer: *The expression arguments $e_1, \ldots e_m$ are evaluated like the arguments of a normal function. The resulting expression, say, $C(v_1, \ldots v_m)$ is already a value.*

- Suppose that we have a class definition:
  class $C(x_1, \ldots x_m)$ { ... def $f(y_1, \ldots y_n)$ = b ... }
  where:
  - The formal parameters of the class are ***$x_1, \ldots x_m$***
  - The class defines a method ***f*** with the formal parameters ***$y_1, \ldots y_n$***

Question: *How is the following expression evaluated?*
$C(v_1, \ldots v_m).f(w_1, \ldots w_n)$
Answer: *The expression is rewritten to:* ***$[w_1/y_1, \ldots w_n/y_n][v_1/x_1, \ldots v_m/x_m][C(v_1, \ldots v_m)/this]b$***
There are three substitutions at work here:

- *The substitution of the formal parameters **$y_1, \ldots y_n$** of the function **f** by the arguments **$w_1, \ldots w_n$***
- *The substitution of the formal parameters **$x_1, \ldots x_m$** of the class **C** by the class arguments **$v_1, \ldots v_m$***
- *The substitution of the self-reference **this** by the value of the object **$C(v_1, \ldots v_m)$***

Extension Methods

- Having to define all methods that belong to a class inside the class itself can lead to very large classes and is not very modular
- Methods that do not need to access the internals of a class can alternatively be defined as extension methods
- The advantage of using extensions instead of class methods is that you can define an extension anywhere; you can define it together with the class, but you can also define it in different modules, and can be defined by different people without having to step over each other's feet

Example:

```
extension (r: Rational)
  def min(s: Rational): Boolean = if s.less(r) then s else r
  def abs: Rational = Rational(r.numerator.abs, r.denominator)
```

- Extensions of a class are visible if they are listed in the companion object of a class or if they are defined or imported in the current scope
- Members of a visible extension of a class can be called as if they were members of that class
- The idea of an extension is that it adds a new functionality to a class without changing existing functionalities; that has two consequences:
  - Extensions can only add new members, not override existing ones
  - Extensions cannot refer to other class members via this

Extension Methods and Substitutions

- Extension method substitution works like normal substitution, but:
    - Instead of this, it's the extension parameter that gets substituted
    - Class parameters are not visible, so do not need to be substituted at all

Operators

- In principle, the rational numbers defined by ***Rational*** are as natural as integers; but for the user of these abstractions, there is a visible difference:
    - We write ***x + y*** if x and y are integers
    - We write ***x.add(y)*** if x and y are rational numbers
- In Scala, we can eliminate the difference, in two steps:

1. Relaxed Identifiers

- Operators such as + or < count as identifiers in Scala
- Thus, an identifier can be:
    - *Alphanumeric*: starting with a letter, followed by a sequence of letters or numbers
    - *Symbolic*: starting with an operator symbol, followed by other operator symbols
    - The '_' character counts as a letter
    - Alphanumeric identifiers can also end in '_', followed by some operator symbols
- Since operators are identifiers, it is possible to use them as methods names

2. Infix Notation

- An operator method with a single parameter can be used as an infix operator
- An alphanumeric method with a single parameter can also be used as an infix operator if it is declared with an ***infix*** modifier

Precedence Rules

- The precedence of an operator is determined by its first character
- The following list presents the characters in increasing order of priority precedence:
    - all letters
    - |
    - ^
    - &
    - <>
    - =!
    - :
    - +-
    - */%
    - all other special characters
- In Java you only have a fixed set of operators, in the exact same order as here, but Scala reused that notion and generalizes it by allowing you to define your own operators that are not only in the Java set, but can also use other special characters or be followed by other operator symbols

Exercise: *Provide a fully parenthesized version of* ***a + b ^? c ?^ d less a ==> b | c*** *. Every binary operation needs to be put into parentheses, but the structure of the expression should not change.*

((a + b) ^? (c ?^ d)) less ((a ==> b) | c)