Week 2 : Evaluating a Function Application (Review)

A simple rule : One evaluates a function application $f(e_1, ..., e_n)$

- by evaluating the expressions $e_1, ..., e_n$ resulting in the values $v_1, ..., v_n$, then
- by replacing the application with the body of the function f, in which
- the actual parameters $v_1, ..., v_n$ replace the formal parameters of f.

This can be formalized as a *rewriting of the program itself*:

$$\textbf{def}\ f\ (x_1, ..., x_n) = B\ ;\ ...\ f\ (v_1, ..., v_n)$$
$$\rightarrow$$
$$\textbf{def}\ f\ (x_1, ..., x_n) = B\ ;\ ...\ [v_1/x_1, ..., v_n/x_n]\ B$$

Here, $[v_1/x_1, ..., v_n/x_n]\ B$ denotes the expression B in which all occurences of x_i have been replaced by v_i.

$[v_1/x_1, ..., v_n/x_n]$ is called a *substitution*.

Example of rewriting:

Consider *gcd*:

$$\textbf{def}\ gcd(a\colon Int,\ b\colon Int)\colon Int = \textbf{if}\ (b == 0)\ a\ \textbf{else}\ gcd(b,\ a\ \%\ b)$$

$gcd(14, 21)$ Evaluated as follows :

	$gcd(14, 21)$
$\rightarrow$	$\textbf{if}\ (21 == 0)\ 14\ \textbf{else}\ gcd(21, 14\ \%\ 21)$
$\rightarrow$	$\textbf{if}\ (\textbf{false})\ 14\ \textbf{else}\ gcd(21, 14\ \%\ 21)$
$\rightarrow$	$gcd(21, 14\ \%\ 21)$
$\rightarrow$	$gcd(21, 14)$
$\rightarrow$	$\textbf{if}\ (14 == 0)\ 21\ \textbf{else}\ gcd(14, 21\ \%\ 14)$
$\rightarrow\rightarrow$	$gcd(14, 21\ \%\ 14)$
$\rightarrow$	$gcd(14, 7)$
$\rightarrow$	$\textbf{if}\ (7 == 0)\ 14\ \textbf{else}\ gcd(7, 14\ \%\ 7)$
$\rightarrow\rightarrow$	$gcd(7, 14\ \%\ 7)$
$\rightarrow$	$gcd(7, 0)$
$\rightarrow$	$\textbf{if}\ (0 == 0)\ 7\ \textbf{else}\ gcd(0, 7\ \%\ 0)$
$\rightarrow\rightarrow$	7

Another example of rewriting:

Consider *factorial*:

$$\textbf{def}\ factorial(n\colon Int)\colon Int = \textbf{if}\ (n == 0)\ 1\ \textbf{else}\ n * factorial(n - 1)$$

$factorial(5)$ can then be rewritten as follows:

	$factorial(5)$
$\rightarrow$	$\textbf{if}\ (5 == 0)\ 1\ \textbf{else}\ 5 * factorial(5 - 1)$
$\rightarrow$	$5 * factorial(5 - 1)$
$\rightarrow$	$5 * factorial(4)$
$\rightarrow ... \rightarrow$	$5 * (4 * factorial(3))$
$\rightarrow ... \rightarrow$	$5 * (4 * (3 * factorial(2)))$
$\rightarrow ... \rightarrow$	$5 * (4 * (3 * (2 * factorial(1))))$
$\rightarrow ... \rightarrow$	$5 * (4 * (3 * (2 * (1 * factorial(0)))))$
$\rightarrow ... \rightarrow$	$5 * (4 * (3 * (2 * (1 * 1))))$
$\rightarrow ... \rightarrow$	120

What are the differences between the two rewritten sequences?

Tail Recursion

Implementation Detail : If a function calls itself as its last action, the function's stack frame can be reused. This is called *tail recursion*.

⇒ Tail recursive functions are iterative processes.

In general, if the last action of a function consists of calling a function (which may be the same), one stack frame is sufficient for both functions. Such calls are called, *tail-calls*.

Exercise: Design a tail recursive version of *factorial*.

Value Definitions

- A definition

```
def f = expr
```

  introduces *f* as a name for the *expression* *expr*.
- *expr* will be evaluated each time that *f* is used.
- In other words, ***def** f* introduces a function without parameters.
- By comparison, a value definition

```
val x = expr
```

  introduces *x* as a name for the *value* of an expression *expr*.
- *expr* will be evaluated once, at the point of definition of the value.

Example:

```
scala> val x = 2
x: Int = 2
scala> val y = square(x)
y: Int = 4
scala> y
res0: Int = 4
```

Example:

```
scala> def loop: Int = loop
loop: Int
scala> val x: Int = loop                    (infinite loop)
^C
```


Higher-Order Functions

Functional languages treat functions as *first-class values.*

This means that, like any other value, a function can be passed as a parameter and returned as a result.

This provides a flexible way to compose programs.

Functions that take other functions as parameters or that return functions as results are called *higher order functions.*

Example:

Take the sum of the integers between *a* and *b*:

```
def sumInts(a: Int, b: Int): Double =
  if (a > b) 0 else a + sumInts(a + 1, b)
```

Take the sum of the cubes of all the integers between *a* and *b* :

```
def cube(x: Int): Double = x * x * x
def sumCubes(a: Int, b: Int): Double =
  if (a > b) 0 else cube(a) + sumCubes(a + 1, b)
```

Take the sum of the reciprocals of the integers between *a* and *b*:

```
def sumReciprocals(a: Int, b: Int): Double =
  if (a > b) 0 else 1.0 / a + sumReciprocals(a + 1, b)
```

These are special cases of $\sum_{n=a}^{b} f(n)$ for different values of f.

Can we factor out the common pattern?

Summing with Higher-Order Functions

We define:

```
def sum(f: Int => Double, a: Int, b: Int): Double = {
  if (a > b) 0
  else f(a) + sum(f, a + 1, b)
}
```

We can then write:

```
def sumInts(a: Int, b: Int): Double = sum(id, a, b)
def sumCubes(a: Int, b: Int): Double = sum(cube, a, b)
def sumReciprocals(a: Int, b: Int): Double = sum(reciprocal, a, b)
```

where

```
def id(x: Int): Double = x
def cube(x: Int): Double = x * x * x
def reciprocal(x: Int): Double = 1.0/x
```

The type *Int* ⇒ *Double* is the type of a function that takes one argument of type *Int* and returns a result of type *Double*.

Anonymous Functions

- Passing functions as parameters leads to the creation of many small functions.
- Sometimes it is cumbersome to have to define (and name) these functions using ***def***.
- A shorter notation makes use of *anonymous functions*.
- Example: A function that raises its argument to a cube is written,

  $(x\colon Int) \Rightarrow x * x * x$

  Here, $x\colon Int$ is the parameter of the function, and $x * x * x$ is it's body.
- The type of the parameter can be omitted if it can be inferred (by the compiler) from the context.

Anonymous Functions are Syntactic Sugar

- In general, $(x_1 : T_1, ..., x_n : T_n) \Rightarrow E$ is a function that relates the result of the expression E to the parameters $x_1, ..., x_n$ (such that E can refer to $x_1, ..., x_n$).
- An anonymous function $(x_1 : T_1, ..., x_n : T_n) \Rightarrow E$ can always be expressed by using ***def*** as follows:

  { ***def*** $f\,(x_1 : T_1, ..., x_n : T_n) = E\ ;\ f$ }

  where *f* is a fresh name (not yet used in the program).
- We say that anonymous functions are *syntactic sugar*.

Summation with Anonymous Functions

We can now write it in a shorter way:

```
def sumInts(a: Int, b: Int): Double = sum(x => x, a, b)
def sumCubes(a: Int, b: Int): Double = sum(x => x * x * x, a, b)
def sumReciprocals(a: Int, b: Int): Double = sum(x => 1.0/x, a, b)
```

Can we still do better by getting rid of *a* and *b* since we only pass them to the *sum* function without actually using them?

Currying

We rewrite *sum* as follows.

```
def sum(f: Int => Double): (Int, Int) => Double = {
  def sumF(a: Int, b: Int): Double =
    if (a > b) 0
    else f(a) + sumF(a + 1, b)
  sumF
}
```

- *sum* is now a function that returns another function. More precisely, the specialized sum function *sumF* applies the function and sums the results. We can then define:

```
def sumInts = sum(x => x)
def sumCubes = sum(x => x * x * x)
def sumReciprocals = sum(x => 1.0/x)
```

- These functions can be applied like the other functions:

```
scala> sumCubes(1, 10) + sumReciprocals(10, 20)
```


Curried Application

How do we apply a function that returns a function?

Example:

```
scala> sum (cube) (1, 10)
3025.0
```

- *sum* (*cube*) applies *sum* to *cube* and returns the *sum of cubes* function (*sum*(*cube*) is therefore equivalent to *sumCubes*).
- This function is next applied to the arguments (*1*, *10*).
- Consequently, function application associates to the left:

```
sum(cube)(1, 10)   ==   (sum (cube)) (1, 10)
```


Definition of Currying

The definition of functions that return functions is so useful in functional programming (FP) that there is a special syntax for it in Scala.

For example, the following definition of *sum* is equivalent to what we saw before, but shorter:

```
def sum(f: Int => Double)(a: Int, b: Int): Double =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)
```

In general, a definition of a curried function

def $f\ (args_1)\ ...\ (args_n) = E$

where $n > 1$, is equivalent to

def $f\ (args_1)\ ...\ (args_{n-1}) = (\ $**def** $g\ (args_n) = E\ ;\ g\ )$

where g is a fresh identifier.

Or for short:

def $f\ (args_1)\ ...\ (args_{n-1}) = (\ args_n \Rightarrow E\ )$

By repeating the process n times

def $f\ (args_1)\ ...\ (args_{n-1})\ (args_n) = E$

becomes equivalent to

def $f = (args_1 \Rightarrow (\ args_2 \Rightarrow ...\ (\ args_n \Rightarrow E\ )\ ...\ ))$

This style of definition and function application is called currying, named for its instigator, Haskell Brooks Curry (1900-1982), a twentieth century logician.

In fact, the idea goes back to Moses Schönfinkel, but the word “currying" has won (perhaps because “schönfinkeling" is more difficult to pronounce).

Function Types

Question : Given,

$$\textbf{def}\ sum(f\colon Int \Rightarrow Double)(a\colon Int,\ b\colon Int)\colon Double = ...$$

What is the type of *sum* ?

Note that functional types associate to the right. That is to say that

$$Int \Rightarrow Int \Rightarrow Int$$

is equivalent to

$$Int \Rightarrow (Int \Rightarrow Int)$$

Exercises:

1. The *sum* function uses linear recursion. Can you write a tail-recursive version by replacing the ???

```
def sum(f: Int => Double)(a: Int, b: Int): Double = {
  def iter(a: Int, result: Double): Double = {
    if (??) ??
    else iter(??, ??)
  }
  iter(??, ??)
}
```

2. Write a *product* function that calculates the product of the values of a function for the points on a given interval.

3. Write *factorial* in terms of *product*.

4. Can you write a more general function, which generalizes both *sum* and *product* ?

Find the fixed points of a function

- A number x is called a *fixed point* of a function f if

  $f(x) = x$
- For some functions, f we can locate the fixed points by starting with an initial estimate and then by applying f in a repetitive way.

  $x,\ f(x),\ f(f(x)),\ f(f(f(x))),\ ...$

  until the value does not vary anymore (or the change is sufficiently small).

This leads to the following function for finding a fixed point:

```
val tolerance = 0.0001
def isCloseEnough(x: Double, y: Double) = abs((x - y) / x) < tolerance
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

Return to Square Roots

Here is a *specification* of the function, *sqrt*.

$$sqrt(x) = \text{the number } y \text{ such that } y * y = x$$
$$= \text{the number } y \text{ such that } y = x \,/\, y$$

Consequently, $sqrt(x)$ is a fixed point function $(y \Rightarrow x \,/\, y)$.

This suggests to calculate $sqrt(x)$ by iteration towards a fixed point:

```
def sqrt(x: Double) =
  fixedPoint(y => x / y)(1.0)
```

Unfortunately it does not converge. If we add a print instruction to the function *fixedPoint* so we can follow the current value of *guess*, we get:


```
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    println(next)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

sqrt(2) then produces:

```
2.0
1.0
2.0
1.0
2.0
...
```


One way to control such oscillations is to prevent the estimation from varying too much. This is done by *averaging* successive values of the original sequence:

```
scala> def sqrt(x: Double) = fixedPoint(y => (y + x / y) / 2)(1.0)
scala> sqrt(2.0)
  1.5
  1.4166666666666665
  1.4142156862745097
  1.4142135623746899
  1.4142135623746899
```

In fact, if we fold the fixed point function *fixedPoint* we find the same square root function that we found last week.

Functions as Return Values

- The previous examples have shown that the expressive power of a language is greatly increased if we can pass function arguments.
- The following example shows that functions that return functions can also be very useful.
- Consider again iteration towards a fixed point.
- We begin by observing that $\sqrt(x)$ is a fixed point of the function. $y \Rightarrow x \,/\, y$.
- Then, the iteration converges by averaging successive values.
- This technique of *stabilizing by averaging* is general enough to merit being in an abstract function.

```
def averageDamp(f: Double => Double)(x: Double) = (x + f(x)) / 2
```

- using *averageDamp*, we can reformulate the square root function as follows.

  def *sqrt*(*x*: *Double*) = *fixedPoint*(*averageDamp*(*y* ⇒ *x*/*y*))(*1.0*)
- This expresses the elements of the algorithm as clearly as possible.

Exercise: Write a square root function by using *fixedPoint* and *averageDamp*.

Résumé

- We saw last week that the functions are essential abstractions because they allow us to introduce general methods to perform computations as explicit and named elements in our programming language.
- This week, we've seen that these abstractions can be combined with higher-order functions to create new abstractions.
- As a programmer, one must look for opportunities to abstract and reuse.
- The highest level of abstraction is not always the best, but it is important to know the techniques of abstraction, so as to use them when appropriate.

Language Elements Seen So Far

- We have seen the language elements to express types, expressions and definitions.
- Below, we give their context-free syntax in Extended Backus-Naur form (EBNF), where '|' denotes an alternative, [...] an option (0 or 1), an {...} a repitition (0 or more).

Types :

```
Type          = SimpleType | FunctionType
FunctionType  = SimpleType '⇒' Type | '(' [Types] ')' '⇒' Type
SimpleType    = Byte | Short | Char | Int | Long | Double | Float
              | Boolean | String
Types         = Type {',' Type}
```

A type can be:

- A numeric type: *Int*, *Double* (and *Byte*, *Short*, *Char*, *Long*, *Float*),
- The *Boolean* type with the values ***true*** and ***false***,
- The *String* type,
- A functional type: *Int* ⇒ *Int*, (*Int*, *Int*) ⇒ *Int*.

Expressions:

Expr	= *InfixExpr* | *FunctionExpr* | ***if*** '(' *Expr* ')' *Expr* ***else*** *Expr*
InfixExpr	= *PrefixExpr* | *InfixExpr Operator InfixExpr*
Operator	= *ident*
PrefixExpr	= ['+' | '−' | '!' | '~'] *SimpleExpr*
SimpleExpr	= *ident* | *literal* | *SimpleExpr* '.' *ident* | *Block*
FunctionExpr	= *Bindings* '⇒' *Expr*
Bindings	= *ident* [':' *SimpleType*] | '(' [*Binding* {',' *Binding*}] ')'
Binding	= *ident* [':' *Type*]
Block	= '{' {*Def* ';'} *Expr* '}'

An expression can be:

- An identifier such as x, *isGoodEnough*,
- A literal, like *0*, *1.0*, *"abc"*,
- A function application, like $sqrt(x)$,
- An operator application, like $-x$, $y + x$,
- A selection, like *Console.println*,
- A conditional expression, like ***if*** $(x < 0)$ $-x$ ***else*** x,
- A block, like { ***val*** $x = abs(y)$; $x * 2$ }
- An anonymous function, like $(x \Rightarrow x + 1)$.

Definitions:

Def	= *FunDef* | *ValDef*
FunDef	= ***def*** *ident* ['(' [*Parameters*] ')'] [':' *Type*] '=' *Expr*
ValDef	= ***val*** *ident* [':' *Type*] '=' *Expr*
Parameter	= *ident* ':' ['⇒'] *Type*
Parameters	= *Parameter* {',' *Parameter*}

A definition can be:

- A function definition like ***def*** $square(x\colon Int) = x * x$
- A value definition like ***val*** $y = square(2)$