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:

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

Here, $[v_1/x_1, ..., v_n/x_n] B$ denotes the expression $B$ in which all occurrences of $x_i$ have been replaced by $v_i$.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

```python
def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
gcd(14, 21) Evaluated as follows:
- $\text{gcd}(14, 21)$
  - $\text{if}(21 == 0)$
    - $14 \text{ else gcd}(21, 14 \% 21)$
  - $\text{gcd}(21, 14 \% 21)$
    - $\text{gcd}(21, 14)$
      - $\text{if}(14 == 0)$
        - $21 \text{ else gcd}(14, 14 \% 14)$
      - $\text{gcd}(14, 14 \% 14)$
        - $\text{gcd}(14, 7)$
          - $\text{if}(7 == 0)$
            - $14 \text{ else gcd}(7, 14 \% 7)$
          - $\text{gcd}(7, 14 \% 7)$
            - $\text{gcd}(7, 0)$
              - $\text{if}(0 == 0)$
                - $7 \text{ else gcd}(0, 7 \% 0)$
              - $\text{gcd}(0, 0)$
                - $7$
```

Another example of rewriting:

Consider factorial:

```python
def factorial(n: Int): Int = if (n == 0) 1 else n * factorial(n - 1)
```

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

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

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.

$\Rightarrow$ 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`.

### Source Code

```python
def gcd(a: int, b: int): int =
    if b == 0:
        a
    else:
        gcd(b, a % b)

gcd(14, 21)  # Evaluates as follows:
    if 21 == 0:
        14
    else:
        gcd(21, 14 % 21)
    gcd(21, 14 % 21)
    gcd(21, 14)
    if 14 == 0:
        21
    else:
        gcd(14, 14 % 14)
    gcd(14, 14 % 14)
    gcd(14, 7)
    if 7 == 0:
        14
    else:
        gcd(7, 14 % 7)
    gcd(7, 14 % 7)
    gcd(7, 0)
    if 0 == 0:
        7
    else:
        gcd(0, 7 % 0)
    gcd(0, 0)
    7
```

```python
def factorial(n: int): int =
    if n == 0:
        1
    else:
        n * factorial(n - 1)

factorial(5)  # Can then be rewritten as follows:
    if 5 == 0:
        1
    else:
        5 * factorial(4)
    5 * factorial(4)
    ... 5 * (4 * factorial(3))
    ... 5 * (4 * (3 * factorial(2)))
    ... 5 * (4 * (3 * (2 * factorial(1))))
    ... 5 * (4 * (3 * (2 * (1 * factorial(0))))))
    ... 5 * (4 * (3 * (2 * (1 * 1))))
    ... 120
```

What are the differences between the two rewritten sequences?
Value Definitions

- A definition
  
  ```scala
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, \texttt{def} \( f \) introduces a function without parameters.
- By comparison, a value definition
  
  ```scala
  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
scala> val x = 2
x: Int = 2
scala> val y = square(x)
y: Int = 4
scala> y
res0: Int = 4
```

Higher-Order Functions

Functional languages treat functions as \textit{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 \textit{higher order functions}.

Example:

Take the sum of the integers between \( a \) and \( b \):

```scala
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 \( b \):

```scala
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 \( b \):

```scala
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:
\[
\text{def } \text{sum}(f: \text{Int} \Rightarrow \text{Double}, a: \text{Int}, b: \text{Int}) : \text{Double} = \{
\text{if } (a > b) 0 \text{ else } f(a) + \text{sum}(f, a + 1, b)\}
\]

We can then write:
\[
\text{def } \text{sumIn ts}(a: \text{Int}, b: \text{Int}) : \text{Double} = \text{sum}(\lambda x: \text{Int} \Rightarrow x, a, b)
\]
\[
\text{def } \text{sumCubes}(a: \text{Int}, b: \text{Int}) : \text{Double} = \text{sum}(\lambda x: \text{Int} \Rightarrow x \times x \times x, a, b)
\]
\[
\text{def } \text{sumReciprocals}(a: \text{Int}, b: \text{Int}) : \text{Double} = \text{sum}(\lambda x: \text{Int} \Rightarrow 1.0/x, a, b)
\]

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,
  \[(x: \text{Int}) \Rightarrow x \times x \times x\]
  Here, \(x: \text{Int}\) is the parameter of the function, and \(x \times x \times x\) is its 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:
  \[
  \{ \text{def } f(x_1: T_1, ..., x_n: T_n) = E ; f \}
  \]
  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:
\[
\text{def } \text{sumInts}(a: \text{Int}, b: \text{Int}) : \text{Double} = \text{sum}(x \Rightarrow x, a, b)
\]
\[
\text{def } \text{sumCubes}(a: \text{Int}, b: \text{Int}) : \text{Double} = \text{sum}(x \Rightarrow x \times x \times x, a, b)
\]
\[
\text{def } \text{sumReciprocals}(a: \text{Int}, b: \text{Int}) : \text{Double} = \text{sum}(x \Rightarrow 1.0/x, a, b)
\]
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.

```scala
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)
}
```

- `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:

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

13

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:

```scala
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

```scala
def f(args1) ... (argsn−1) = ( argsn ⇒ E )
```

where \( n > 1 \), is equivalent to

```scala
def f(args1) ... (argsn−1) = ( def g (argsn) = E ; g )
```

where `g` is a fresh identifier.

Curried Application

How do we apply a function that returns a function?

Example:

```scala
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:

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

For short:

```scala
def f(args1) ... (argsn−1) = ( argsn ⇒ E )
```

By repeating the process \( n \) times

```scala
def f(args1) ... (argsn−1)(argsn) = E
```

becomes equivalent to

```scala
def f = (args1 ⇒ (args2 ⇒ ... (argsn ⇒ 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,
\[
\text{def } \text{sum}( f : \text{Int} \Rightarrow \text{Double})( a : \text{Int}, b : \text{Int}) : \text{Double} = \ldots
\]
What is the type of \text{sum}?
Note that functional types associate to the right. That is to say that
\[\text{Int} \Rightarrow \text{Int} \Rightarrow \text{Int}\]
is equivalent to
\[\text{Int} \Rightarrow (\text{Int} \Rightarrow \text{Int})\]

Exercises:

1. The \text{sum} function uses linear recursion. Can you write a tail-recursive version by replacing the ???
\[
\text{def } \text{sum}( f : \text{Int} \Rightarrow \text{Double})( a : \text{Int}, b : \text{Int}) : \text{Double} = \{
\text{def } \text{iterate}( a : \text{Int}, \text{result} : \text{Double}) : \text{Double} = \{
\text{if } (??) ??
\text{else } \text{iterate}(??, ??)
\}
\text{iterate}(??)
\}
\]
2. Write a \text{product} function that calculates the product of the values of a function for the points on a given interval.
3. Write \text{factorial} in terms of \text{product}.
4. Can you write a more general function, which generalizes both \text{sum} and \text{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. \(x, f(x), f(f(x)), f(f(f(x))), \ldots\) until the value does not vary anymore (or the change is sufficiently small).

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

\[
\text{val tolerance} = 0.0001
\text{def } \text{isCloseEnough}( x : \text{Double}, y : \text{Double}) = \text{abs}(x - y) / x < \text{tolerance}
\text{def } \text{fixedPoint}( f : \text{Double} \Rightarrow \text{Double})( \text{firstGuess} : \text{Double}) = \{
\text{def } \text{iterate}( \text{guess} : \text{Double}) : \text{Double} = \{
\text{val } \text{next} = f(\text{guess})
\text{if } (\text{isCloseEnough}(\text{guess}, \text{next})) \text{ next}
\text{else } \text{iterate}(\text{next})
\}
\text{iterate}(\text{firstGuess})
\}
\]
Return to Square Roots

Here is a specification of the function, \( \sqrt{x} \).

\[ \sqrt{x} = \text{the number } y \text{ such that } y \cdot y = x \]

Consequently, \( \sqrt{x} \) is a fixed point function \((y \mapsto x / y)\).

This suggests to calculate \( \sqrt{x} \) by iteration towards a fixed point:

\[
\text{def } \sqrt{x} : \text{Double } = \text{fixedPoint}(y \mapsto x / y)(1.0)
\]

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

```
def fixedPoint(f: Double => Double)(firstGuess: Double) = {
  def iterate(guess: 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:

```
sqrt(x: Double) = fixedPoint(y => (y + x / y) / 2)(1.0)
```

In fact, if we fold the fixed point function \text{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 \( y \mapsto x / y \).
- Then, the iteration converges by averaging successive values.
- This technique of stabilizing by averaging is general enough to merit being in an abstract function.

\[
\text{def } \text{averageDamp}(f: \text{Double} \Rightarrow \text{Double})(x: \text{Double}) = (x + f(x)) / 2
\]
Using `averageDamp`, we can reformulate the square root function as follows.

```haskell
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:**

```plaintext
Type = SimpleType | FunctionType
FunctionType = SimpleType ⇒ 'Type | ' Type ⇒ 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`. 

---

```plaintext
`T yp e` = `S im pleT yp e` | `F unctionT yp e`
`F unctionT yp e` = `S im pleT yp e` `⇒` `T yp e` | `T yp e` `⇒` `T yp e`
`S im pleT yp e` = `B yte` | `S hort` | `C har` | `I nt` | `L ong` | `D ouble` | `F loat`
| `B oolean` | `S tring`
`T yp es` = `T yp e {` `,' ` T yp e}`
A t y pe can b e:
- A n um eric t y pe: `I nt`, `D ouble` (and `B yte`, `S hort`, `C har`, `L ong`, `F loat`),
- The `B oolean` t y pe w ith t he v alues `t rue` and `f alse`,
- The `S tring` t y pe,
- A f unctional t y pe: `I nt ⇒ I nt`, `(I nt, I nt) ⇒ I nt`.
```
Expressions:

\[
\text{Expr} = \text{InfixExpr} \mid \text{FunctionExpr} \mid \text{if} (\text{Expr}) \text{Expr} \text{else} \text{Expr}
\]

\[
\text{InfixExpr} = \text{PrefixExpr} \mid \text{InfixExpr} \text{Operator} \text{InfixExpr}
\]

\[
\text{Operator} = \text{idt}
\]

\[
\text{PrefixExpr} = [\text{"+"} | \text{"-"} | \text{"!"} | \text{""}] \text{SimpleExpr}
\]

\[
\text{SimpleExpr} = \text{idt} \mid \text{literal} \mid \text{SimpleExpr} \text{.} \text{idt} \mid \text{Block}
\]

\[
\text{FunctionExpr} = \text{Bindings} \Rightarrow \text{Expr}
\]

\[
\text{Bindings} = \text{idt} [\text{"::" SimpleType}] \mid (\text{\{} \text{Binding} {\text{,\text{,}} \text{Binding}} \text{\}})
\]

\[
\text{Binding} = \text{idt} [\text{"::" Type}]
\]

\[
\text{Block} = \{\text{\{} \text{Def} ; \text{\}; Expr} \text{\}}
\]

An expression can be:

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

Definitions:

\[
\text{Def} = \text{FunDef} \mid \text{ValDef}
\]

\[
\text{FunDef} = \text{def idt} [\text{\{} \text{Parameters} \text{\}}] [\text{"::" Type}] \text{\{} Expr
\]

\[
\text{ValDef} = \text{val idt} [\text{"::" Type}] \text{\{} Expr
\]

\[
\text{Parameter} = \text{idt} [\text{"::" Type}]
\]

\[
\text{Parameters} = \text{Parameter} {\text{,\text{,}} \text{Parameter}}
\]

A definition can be:

- A function definition like \text{def square(x: Int) = x \ast x}
- A value definition like \text{val y = square(2)}