Evaluation of Function Application (Repetition)

One simple rule: A function application $f(e_1, \ldots, e_n)$ is evaluated by

- Evaluating exprsssions $e_1, \ldots, e_n$ to values $v_1, \ldots, v_n$, and
- replacing the application with the function's body where,
- actual parameters $v_1, \ldots, v_n$ replace formal parameters of f.

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

$$
\begin{array}{ll}
 & \textbf{def}\ f\ (x_1, \ldots, x_n) = B\ ;\ \ldots\ f\ (v_1, \ldots, v_n) \\
\rightarrow & \\
 & \textbf{def}\ f\ (x_1, \ldots, x_n) = B\ ;\ \ldots\ [v_1/x_1, \ldots, v_n/x_n]\ B
\end{array}
$$

Here, $[v_1/x_1, \ldots, v_n/x_n]\ B$ stands for B with all occurrences of x_i replaced by v_i.

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

Rewriting Example:

Consider *gcd*:

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

Then $gcd(14, 21)$ evaluates as follows:

$$
\begin{array}{ll}
 & gcd(14, 21) \\
\rightarrow & \textbf{if}\ (21 == 0)\ 14\ \textbf{else}\ gcd(21,\ 14\ \%\ 21) \\
\rightarrow & \textbf{if}\ (False)\ 14\ \textbf{else}\ gcd(21,\ 14\ \%\ 21) \\
\rightarrow & gcd(21,\ 14\ \%\ 21) \\
\rightarrow & gcd(21,\ 14) \\
\rightarrow & \textbf{if}\ (14 == 0)\ 21\ \textbf{else}\ gcd(14,\ 21\ \%\ 14) \\
\rightarrow\ \rightarrow & gcd(14,\ 21\ \%\ 14) \\
\rightarrow & gcd(14,\ 7) \\
\rightarrow & \textbf{if}\ (7 == 0)\ 14\ \textbf{else}\ gcd(7,\ 14\ \%\ 7) \\
\rightarrow\ \rightarrow & gcd(7,\ 14\ \%\ 7) \\
\rightarrow & gcd(7,\ 0) \\
\rightarrow & \textbf{if}\ (0 == 0)\ 7\ \textbf{else}\ gcd(0,\ 7\ \%\ 0) \\
\rightarrow\ \rightarrow & 7
\end{array}
$$

Another rewriting example:

Consider *factorial*:

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

Then $factorial(5)$ rewrites as follows:

$$
\begin{array}{ll}
 & factorial(5) \\
\rightarrow & \textbf{if}\ (5 == 0)\ 1\ \textbf{else}\ 5 * factorial(5 - 1) \\
\rightarrow & 5 * factorial(5 - 1) \\
\rightarrow & 5 * factorial(4) \\
\rightarrow \ldots \rightarrow & 5 * (4 * factorial(3)) \\
\rightarrow \ldots \rightarrow & 5 * (4 * (3 * factorial(2))) \\
\rightarrow \ldots \rightarrow & 5 * (4 * (3 * (2 * factorial(1)))) \\
\rightarrow \ldots \rightarrow & 5 * (4 * (3 * (2 * (1 * factorial(0))))) \\
\rightarrow \ldots \rightarrow & 5 * (4 * (3 * (2 * (1 * 1)))) \\
\rightarrow \ldots \rightarrow & 120
\end{array}
$$

What differences are there between the two rewrite sequences?

Tail Recursion

Implementation note: If a function calls itself as its last action, the function's stack frame can be re-used. This is called "tail recursion".

⇒ Tail-recursive functions are iterative processes.

More generally, if the last action of a function is a call to another (possible the same) function, only a single stack frame is needed for both functions. Such calls are called "tail calls".

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

Example:

Sum integers between a and b:

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

Sum square roots of all 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) + sumSqrts(a + 1, b)
```

Sum reciprocals 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 all special cases of $\sum_a^b f(n)$ for different values of f.

Can we factor out the common pattern?

First-Class Functions

Functional languages treat functions as “first-class values”.

That is, like any other value, a function may be passed as a parameter or returned as a result.

This provides a flexible mechanism for program composition.

Functions which take other functions as parameters or return them as results are called “higher-order” functions.

Example:

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

Example:

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


Value Definitions

- A definition

  def *f = expr*

  introduces *f* as a name for the *expression* *expr*.
- *expr* will be evaluated every time the name *f* is used.
- In other words, ***def*** *f* introduces a parameterless function.
- By contrast a value definition

  val *x = expr*

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

Currying

Let's rewrite *sum* as follows.

```
def sum(f: (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 which returns another function, namely the specialized summing function *sumF* which applies the *f* function and sums up the results. Then we can 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 other functions:

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


Summation with Anonymous Functions

Now we can write shorter:

```
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 do even better?

Hint: *a*, *b* appears everywhere and does not seem to take part in interesting combinations. Can we get rid of it?

Anonymous functions

- Parameterisation by functions tends to create many small functions.
- Sometimes it is cumbersome to have to define the functions using ***def***.
- A shorter notation makes use of *anonymous functions*, defined as follows:

  $(x_1 : T_1, ..., x_n : T_n \Rightarrow E)$ defines a function which maps its parameters $x_1, ..., x_n$ to the result of the expression E (where E may refer to $x_1, ..., x_n$).
- An anonymous function $(x_1 : T_1, ..., x_n : T_n \Rightarrow E)$ can always be expressed using a ***def*** as follows:

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

  where *f* is fresh name which is used nowhere else in the program.
- We also say, anonymous functions are "syntactic sugar".
- (One convenience of anonymous functions is that one can leave out the parameter types T_i if they are clear from the context).

Summation with a higher-order function

Define:

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

Then we can 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 functions that take arguments of type *Int* and return results of type *Double*.

Curried Application

How are function-returning functions applied?

Example:

> *sum (cube) (1, 10)*
> *3025.0*

- *sum (cube)* applies *sum* to *cube* and returns the "cube-summing function" (Hence, *sum (cube)* is equivalent to *sumCubes*).
- This function is then applied to the arguments (*1, 10*).
- Hence, function application associates to the left:

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

Curried Definition

The style of function-returning functions is so useful in FP, that we have special syntax for it.

For instance, the next definition of *sum* is equivalent to the previous one, but shorter:

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

Generally, a curried function definition

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

where $n > 1$ expands to

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

where g is a fresh identifier. Or, shorter:

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

Performing this step n times yields that

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

is equivalent to

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

- This style of function definition and application is called *currying* after its promoter, Haskell B. Curry, a logician of the 20th century.
- Actually, the idea goes back further Schönfinkel, but the name "curried" caught on (maybe because "schönfinkeled" is harder to pronounce.)

Function Types

Question: Given

def *sum* (*f*: (*Int*)*Double*) (*a*: *Int*, *b*: *Int*): *Double* = ...

What is the type of *sum*?

Note that function types associate to the *right*. I.e.

(*Int*) (*Int*)*Int*

is equivalent to

(*Int*) ((*Int*)*Int*)

Square Roots Again

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

$$
\begin{aligned}
sqrt(x) &= \text{the } y \text{ such that } y * y = x \\
&= \text{the } y \text{ such that } y = x \,/\, y
\end{aligned}
$$

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

This suggests that $sqrt(x)$ can be computed by fixed point iteration:

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

Unfortunately, this does not converge. Let's instrument the fixedpoint function with a print statement which keeps track of the current *guess* value:

This leads to the following "fixed-point finding function":

```
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)
}
```


Finding Fixed Points of Functions

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

  $f(x) = x$

- For some functions f we can locate the fixed point by beginning with an initial guess and then applying f repeatedly.

  $x,\ f(x),\ f(f(x)),\ f(f(f(x))),\ \ldots$

  until the value does not change anymore (or the change is within a small tolerance).

Exercises:

1. The *sum* function uses a linear recursion. Can you write a tail-recursive one by filling in the ??'s?

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

2. Write a function *product* that computes the product of the values of functions at points over a given range.

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

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

- 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 algorith as clearly as possible.

Exercise: Write a function for cube roots using *fixedPoint* and *averageDamp*.

Functions as Returned Values

- The previous examples showed that the expressive power of a language is considerably enhanced if functions can be passed as arguments.
- The next example shows that functions which return functions can also be very useful.
- Consider again fixed point iterations.
- We started with the observation that $\sqrt(x)$ is a fixed point of the function $y \Rightarrow x \,/\, y$.
- Then we made the iteration converge by averaging successive values.
- This technique of *average dampening* is so general enough that it can be wrapped in another fucntion.

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


One way to control such oscillations is to prevent the guess from chaning to much. This can be achieved by *averaging* successive values of the original sequence:

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

In fact, expanding the *fixedPoint* function yields exactly our previous definition of fixed point from week1.


```
def fixedPoint(f: (Double)Double)(firstGuess: Double) = {
  def iterate(guess: Double): Double = {
    val next = f(guess)
    java.lang.System.out.println(next)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

Then, *sqrt*(2) yields:

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

Summary

- We have seen last week that functions are essential abstractions, because they permit us to introduce general methods of computing as explicit, named elements in our programming language.
- This week we have seen that these abstractions can be combined by higher-order fucntions to create further abstractions.
- As programmers, we should look out for opportunities to abstract and to reuse.
- The highest possiblelevel of abstraction is not always the best, but it is important to know abstraction techniques, so that one can use abstractions where appropriate.

Language Elements Seen So Far

- We have seen language elements to express types, expressions and definitions.
- Their context free syntax is given below in extended Backus-Naur form, where '|' denotes alternatives, [...] denotes option (0 or 1), and {...} denotes repetition (0 or more).

Types:

```
Type      = Int | Double | Boolean | String | '(' Types ')' Type
Types     = Type {',' Type}
```

Types can be:

- number types *Int* or *Double* (also *Byte*, *Short*, *Char*, *Long*, *Float*),
- the type *Boolean* with values *True* and *False*,
- the type *String*,
- function types.

Expressions:

```
Expr      = ident
          | Literal
          | Expr '(' Exprs ')'
          | operator Expr
          | Expr operator Expr
          | Expr '.' id
          | if '(' Expr ')' Expr else Expr
          | Block
          | Function
Exprs     = [Expr {',' Expr}]
Block     = '(' {Def} ';' Expr ')'
          | '{' {Def} Expr '}'       // indentation significant!
Function  |  '(' Bindings '⇒' Expr ')'
Bindings  =  Binding {',' Binding}
Binding   =  ident [':' Type]
```


Expressions can be:

- identifiers such as x, *isGoodEnough*,
- literals, such as 0, 1.0, "abc",
- function applications, such as *sqrt* (x),
- operator applications, such as −x, @y + x@,
- selections, such as *java.lang.System.out.println*,
- conditionals, such as **if** (x < 0) −x **else** x,
- blocks, such as { **val** x = *abs* (y) ; x ∗ 2 }
- anonymous functions, such as (x ⇒ x + 1).

Definitions:

$$\begin{array}{ll}
\mathit{Def} & = \mathit{FunDef} \;\mid\; \mathit{ValDef} \\
\mathit{FunDef} & = \textbf{\textit{def}}\ \mathit{ident}\ [\text{`('}\ [\mathit{Parameters}]\ \text{`)'}]\ [\text{`:'}\ \mathit{Type}]\ \text{`='}\ \mathit{Expr} \\
\mathit{ValDef} & = \textbf{\textit{val}}\ \mathit{ident}\ [\text{`:'}\ \mathit{Type}]\ \text{`='}\ \mathit{Expr} \\
\mathit{Parameter} & = [\textbf{\textit{def}}]\ \mathit{ident}\ \text{`:'}\ \mathit{Type} \\
\mathit{Parameters} & = \mathit{Parameter}\ \{\text{`,'}\ \mathit{Parameter}\}
\end{array}$$

Defintions can be:

- Function definitions such as $\textbf{\textit{def}}\ \mathit{square}(x\colon \mathit{Int}) = x * x$
- Value definitions such as $\textbf{\textit{val}}\ y = \mathit{square}(2)$