香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Principles of Programming Languages (Lecture 2)

COMP 3031, Fall 2025

Lionel Parreaux

Tail Recursion

Review: Evaluating a Function Application

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

- by evaluating the expressions $e_1, \ldots, e_n$ 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`.

Application Rewriting Rule

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

$$\begin{array}{ll} & \mathtt{def}\ \mathtt{f}(\mathtt{x}_1, ..., \mathtt{x}_n) = \mathtt{B};\ \ ...\ \mathtt{f}(\mathtt{v}_1, ..., \mathtt{v}_n) \\ \rightarrow & \\ & \mathtt{def}\ \mathtt{f}(\mathtt{x}_1, ..., \mathtt{x}_n) = \mathtt{B};\ \ ...\ [\mathtt{v}_1/\mathtt{x}_1, ..., \mathtt{v}_n/\mathtt{x}_n]\,\mathtt{B} \end{array}$$

Here, $[\mathtt{v}_1/\mathtt{x}_1, ..., \mathtt{v}_n/\mathtt{x}_n]\,\mathtt{B}$ means:

The expression `B` in which all occurrences of $\mathtt{x_i}$ have been replaced by $\mathtt{v_i}$.

$[\mathtt{v}_1/\mathtt{x}_1, ..., \mathtt{v}_n/\mathtt{x}_n]$ is called a *substitution*.

Rewriting example:

Consider `gcd`, the function that computes the greatest common divisor of two numbers.

Here's an implementation of `gcd` using Euclid's algorithm.

```
def gcd(a: Int, b: Int): Int =
  if b == 0 then a else gcd(b, a % b)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
```

$\rightarrow$ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if 14 == 0 then 21 else gcd(14, 21 % 14)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if 14 == 0 then 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if 14 == 0 then 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
↠ gcd(7, 0)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if 14 == 0 then 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
↠ gcd(7, 0)
→ if 0 == 0 then 7 else gcd(0, 7 % 0)
```

Rewriting example:

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if 14 == 0 then 21 else gcd(14, 21 % 14)
→» gcd(14, 7)
→» gcd(7, 0)
→ if 0 == 0 then 7 else gcd(0, 7 % 0)
→ 7
```

Another rewriting example:

Consider `factorial`:

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

```
factorial(4)
```

Another rewriting example:

Consider `factorial`:

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

```
factorial(4)
```

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0)))`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)`

$\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * 1)))` $\twoheadrightarrow$ `24`

What are the differences between the two sequences?

Tail Recursion

Implementation Consideration:

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 would be sufficient for both functions. Such calls are called *tail-calls*.

Tail Recursion in Scala

In Scala, only directly recursive calls to the current function are optimized.

One can require that a function is tail-recursive using a `@tailrec` annotation:

```
import scala.annotation.tailrec

@tailrec
def gcd(a: Int, b: Int): Int = ...
```

If the annotation is given, and the implementation of `gcd` were not tail recursive, an error would be issued.

Exercise: Tail recursion

Design a tail recursive version of `factorial`.

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

Higher-Order Functions

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): Int =
  if a > b then 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): Int = x * x * x

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

Example (ctd)

Take the sum of the factorials of all the integers between `a` and `b` :

```
def sumFactorials(a: Int, b: Int): Int =
  if a > b then 0 else factorial(a) + sumFactorials(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

Let's define:

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

We can then write:

```
def sumInts(a: Int, b: Int)       = sum(id, a, b)
def sumCubes(a: Int, b: Int)      = sum(cube, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

where

```
def id(x: Int): Int   = x
def cube(x: Int): Int = x * x * x
def fact(x: Int): Int = if x == 0 then 1 else x * fact(x - 1)
```

Function Types

The type `A => B` is the type of a *function* that takes an argument of type `A` and returns a result of type `B`.

So, `Int => Int` is the type of functions that map integers to integers.

Anonymous Functions

Passing functions as parameters leads to the creation of many small functions.

- Sometimes it is tedious to have to define (and name) these functions using `def`.

Compare to strings: We do not need to define a string using `def`. Instead of

```
def str = "abc"; println(str)
```

We can directly write

```
println("abc")
```

because strings exist as *literals*. Analogously we would like function literals, which let us write a function without giving it a name.

These are called *anonymous functions*.

Anonymous Function Syntax

Example: A function that raises its argument to a cube:

```
(x: Int) => x * x * x
```

Here, `(x: 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.

If there are several parameters, they are separated by commas:

```
(x: Int, y: Int) => x + y
```

Note: anonymous functions are also called *lambda expressions*.

Anonymous Function Syntax

Example: A function that raises its argument to a cube:

```
(x: Int) => x * x * x
```

Here, `(x: 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.

If there are several parameters, they are separated by commas:

```
(x: Int, y: Int) => x + y
```

Note: anonymous functions are also called *lambda expressions*.

Anonymous Functions are Syntactic Sugar

An anonymous function $(x_1 : T_1, ..., x_n : T_n) \Rightarrow E$ can always be expressed using `def` as follows:

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

where f is an arbitrary, fresh name (that's not yet used in the program).

- One can therefore say that anonymous functions are *syntactic sugar*.

Summation with Anonymous Functions

Using anonymous functions, we can write sums in a shorter way:

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

Exercise

The `sum` function uses linear recursion. Write a tail-recursive version by replacing the `???`s.

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  def loop(a: Int, acc: Int): Int =
    if ??? then ???
    else loop(???, ???)
  loop(???, ???)
```

Exercise

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

Currying

Motivation

Look again at the summation functions:

```
def sumInts(a: Int, b: Int)       = sum(x => x, a, b)
def sumCubes(a: Int, b: Int)      = sum(x => x * x * x, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

Q:

Note that `a` and `b` get passed unchanged from `sumInts` and `sumCubes` into `sum`.

Can we be even shorter by getting rid of these parameters?

Functions Returning Functions

Let's rewrite `sum` as follows.

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

`sum` is now a function that returns another function.

The returned function `sumF` applies the given function parameter `f` and sums the results.

Stepwise Applications

We can then define:

```
def sumInts       = sum(x => x)
def sumCubes      = sum(x => x * x * x)
def sumFactorials = sum(fact)
```

These functions can in turn be applied like any other function:

```
sumCubes(1, 10) + sumFactorials(10, 20)
```

Consecutive Stepwise Applications

In the previous example, can we avoid the `sumInts`, `sumCubes`, … middlemen?

Of course:

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

Function application associates to the left:

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

- `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)`.

Consecutive Stepwise Applications

In the previous example, can we avoid the `sumInts`, `sumCubes`, … middlemen?

Of course:

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

Function application associates to the left:

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

- `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)`.

Consecutive Stepwise Applications

In the previous example, can we avoid the `sumInts`, `sumCubes`, ... middlemen?

Of course:

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

Function application associates to the left:

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

- `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)`.

Multiple Parameter Lists

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

For example, the following definition of `sum` is equivalent to the one with the nested `sumF` function, but shorter:

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

Expansion of Multiple Parameter Lists

In general, a definition of a function with multiple parameter lists

$$\mathtt{def}\ f(ps_1)...(ps_n) = E$$

where $n > 1$, is equivalent to

$$\mathtt{def}\ f(ps_1)...(ps_{n-1}) = \{\mathtt{def}\ g(ps_n) = E; g\}$$

where g is a fresh identifier.

Or, for short:

$$\mathtt{def}\ f(ps_1)...(ps_{n-1}) = (ps_n) \Rightarrow E$$

Expansion of Multiple Parameter Lists

In general, a definition of a function with multiple parameter lists

$$\text{def } f(ps_1)...(ps_n) = E$$

where $n > 1$, is equivalent to

$$\text{def } f(ps_1)...(ps_{n-1}) = \{\text{def } g(ps_n) = E; g\}$$

where g is a fresh identifier.

Or, for short:

$$\text{def } f(ps_1)...(ps_{n-1}) = (ps_n) \Rightarrow E$$

Expansion of Multiple Parameter Lists (2)

By repeating the process *n* times

$$\mathsf{def}\ \mathsf{f}(\mathsf{ps}_1)...(\mathsf{ps}_{n-1})(\mathsf{ps}_n) = \mathsf{E}$$

is shown to be equivalent to

$$\mathsf{def}\ \mathsf{f} = (\mathsf{ps}_1) \Rightarrow (\mathsf{ps}_2) \Rightarrow ... \Rightarrow (\mathsf{ps}_n) \Rightarrow \mathsf{E}$$

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 even further to Schönfinkel and Frege, but the term “currying” has stuck.

Expansion of Multiple Parameter Lists (2)

By repeating the process *n* times

$$\mathtt{def\ f(ps_1)...(ps_{n-1})(ps_n) = E}$$

is shown to be equivalent to

$$\mathtt{def\ f = (ps_1) \Rightarrow (ps_2) \Rightarrow ... \Rightarrow (ps_n) \Rightarrow E}$$

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 even further to Schönfinkel and Frege, but the term "currying" has stuck.

More Function Types

Question: Given,

```
def sum(f: Int => Int)(a: Int, b: Int): Int = ...
```

What is the type of ‘`sum`’ ?

More Function Types

Question: Given,

```
def sum(f: Int => Int)(a: Int, b: Int): Int = ...
```

What is the type of `sum` ?

Answer:

```
(Int => Int) => (Int, Int) => Int
```

Note that *function types associate to the right*. That is to say that

```
Int => Int => Int
```

is equivalent to

```
Int => (Int => Int)
```

More Function Types

Question: Given,

```
def sum(f: Int => Int)(a: Int, b: Int): Int = ...
```

What is the type of `sum` ?

Answer:

```
(Int => Int) => (Int, Int) => Int
```

Note that *function types associate to the right*. That is to say that

```
Int => Int => Int
```

is equivalent to

```
Int => (Int => Int)
```

Exercise

1. Write a `product` function that calculates the product of the values of a function for the points on a given interval.
2. Write `factorial` in terms of `product`.
3. Can you write a more general function, which generalizes both `sum` and `product`?

Exercise

1. Write a `product` function that calculates the product of the values of a function for the points on a given interval.

Exercise

1. Write a `product` function that calculates the product of the values of a function for the points on a given interval.

```
def product(f: Int => Int)(a: Int, b: Int): Int =
  if a > b then 1 else f(a) * product(f)(a + 1, b)
```

Exercise

1. Write a `product` function that calculates the product of the values of a function for the points on a given interval.

```
def product(f: Int => Int)(a: Int, b: Int): Int =
  if a > b then 1 else f(a) * product(f)(a + 1, b)
```

2. Write `factorial` in terms of `product`.

Exercise

1. Write a `product` function that calculates the product of the values of a function for the points on a given interval.

```
def product(f: Int => Int)(a: Int, b: Int): Int =
  if a > b then 1 else f(a) * product(f)(a + 1, b)
```

2. Write `factorial` in terms of `product`.

```
def factorial(n: Int) = product(identity)(1, n)
```

Exercise

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

Exercise

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

```
def mapReduce(f: Int => Int, combine: (Int, Int) => Int, zero: Int)
             (a: Int, b: Int): Int =
  if a > b then zero
  else combine(f(a), mapReduce(f, combine, zero)(a + 1, b))
```

```
def sum(f: Int => Int) = mapReduce(f, (x, y) => x + y, 0)

def product(f: Int => Int) = mapReduce(f, (x, y) => x * y, 1)
```

Exercise

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

```
def mapReduce(f: Int => Int, combine: (Int, Int) => Int, zero: Int)
             (a: Int, b: Int): Int =
  if a > b then zero
  else combine(f(a), mapReduce(f, combine, zero)(a + 1, b))

def sum(f: Int => Int) = mapReduce(f, (x, y) => x + y, 0)

def product(f: Int => Int) = mapReduce(f, (x, y) => x * y, 1)
```

Exercise

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

```
def mapReduce(f: Int => Int, combine: (Int, Int) => Int, zero: Int)
             (a: Int, b: Int): Int =
  if a > b then zero
  else combine(f(a), mapReduce(f, combine, zero)(a + 1, b))

def sum(f: Int => Int) = mapReduce(f, (x, y) => x + y, 0)

def product(f: Int => Int) = mapReduce(f, (x, y) => x * y, 1)
```

Generalizing Further?

In following weeks, we'll see how to generalize functions like `mapReduce` to:

4. Arbitrary types (not just `Int`).

5. Arbitrary sequences (not just contiguous integer sequences).

Finding a fixed point 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).

Programmatic Solution

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): Double =
  def iterate(guess: Double): Double =
    val next = f(guess)
    if isCloseEnough(guess, next) then next
    else iterate(next)
  iterate(firstGuess)
```

Return to Square Roots

Here is a *specification* of the sqrt function:

`sqrt(x)` = the number `y` such that `y * y = x`.

Or, by dividing both sides of the equation with `y`:

`sqrt(x)` = the number `y` such that `y = x / y`.

Consequently, `sqrt(x)` is a fixed point of the function `(y => x / y)`.

First Attempt

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

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

Unfortunately, this does not converge.

Let's add a `println` instruction to the function `fixedPoint` so we can follow the current value of `guess`:

First Attempt (2)

```
def fixedPoint(f: Double => Double)(firstGuess: Double) =

  def iterate(guess: Double): Double =
    val next = f(guess)
    println(next)
    if isCloseEnough(guess, next) then next
    else iterate(next)

  iterate(firstGuess)
```

`sqrt(2)` then produces:

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

Average Damping

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:

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

This produces

```
1.5
1.4166666666666665
1.4142156862745097
1.4142135623746899
1.4142135623746899
```

In fact, if we expand the fixed point function `fixedPoint` we find a similar square root function to what we developed 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 => x / y`.

Then, the iteration converges by averaging successive values.

This technique of *stabilizing by averaging* is general enough to merit being abstracted into its own function.

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

Exercise: Final Formulation of Square Root

Write a square root function using `fixedPoint` and `averageDamp`.

Exercise: Final Formulation of Square Root

Write a square root function using `fixedPoint` and `averageDamp`.

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

This expresses the elements of the algorithm as clearly as possible.

Summary

We saw last week that 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.

Summary

We saw last week that 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.

Summary

We saw last week that 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 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),

`{...}` a repetition (0 or more).

Types

```
Type         = SimpleType | FunctionType
FunctionType = SimpleType '=>' Type
             | '(' [Types] ')' '=>' Type
SimpleType   = Ident
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 *function type*, like `Int => Int`, `(Int, Int) => Int`.

Later we will see more forms of types.

Expressions

```
Expr         = InfixExpr | FunctionExpr
             | if Expr then Expr else Expr
InfixExpr    = PrefixExpr | InfixExpr Operator InfixExpr
Operator     = ident
PrefixExpr   = [‘+’ | ‘-’ | ‘!’ | ‘~’ ] SimpleExpr
SimpleExpr   = ident | literal | SimpleExpr ‘.’ ident
             | Block
FunctionExpr = Bindings ‘=>‘ Expr
Bindings     = ident
             | ‘(’ [Binding {‘,’ Binding}] ‘)’
Binding      = ident [‘:’ Type]
Block        = ‘{’ {Def ‘;’} Expr ‘}’
             | <indent> {Def ‘;’} Expr <outdent>
```

Expressions (2)

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 `math.abs`,
- A *conditional expression*, like `if x < 0 then -x else x`,
- A *block*, like `{ val x = abs(y) ; x * 2 }`
- An *anonymous function*, like `x => 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: Int) = x * x`
- A *value definition*, like `val y = square(2)`

A *parameter* can be:

- A *call-by-value parameter*, like `(x: Int)`,
- A *call-by-name parameter*, like `(y: => Double)`.

Functions and Data

Functions and Data

In this section, we'll learn how functions create and encapsulate data structures.

Example: Rational Numbers

We want to design a package for doing rational arithmetic.

A rational number $\frac{x}{y}$ is represented by two integers:

- its *numerator* x, and
- its *denominator* y.

Rational Addition

Suppose we want to implement the addition of two rational numbers.

```
def addRationalNumerator(n1: Int, d1: Int, n2: Int, d2: Int): Int
def addRationalDenominator(n1: Int, d1: Int, n2: Int, d2: Int): Int
```

but it would be difficult to manage all these numerators and denominators.

A better choice is to combine the numerator and denominator of a rational number in a data structure.

Classes

In Scala, we do this by defining a *class*:

```
class Rational(x: Int, y: Int):
  def numer = x
  def denom = y
```

This definition introduces two entities:

- A new *type*, named `Rational`.
- A *constructor* `Rational` to create elements of this type.

Scala keeps the names of types and values in *different namespaces*. So there's no conflict between the two entities named `Rational`.

Objects

We call the elements of a class type *objects*.

We create an object by calling the constructor of the class:

Example

```
Rational(1, 2)
```

Members of an Object

Objects of the class `Rational` have two *members*, `numer` and `denom`.

We select the members of an object with the infix operator '`.`'.

Example

```
val x = Rational(1, 2)  // x: Rational = Rational@2abe0e27
x.numer                 // 1
x.denom                 // 2
```

Rational Arithmetic

We can now define the arithmetic functions that implement the standard rules.

$$
\begin{aligned}
\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 d_2 + n_2 d_1}{d_1 d_2} \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 d_2 - n_2 d_1}{d_1 d_2} \\
\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} &= \frac{n_1 n_2}{d_1 d_2} \\
\frac{n_1}{d_1} \Big/ \frac{n_2}{d_2} &= \frac{n_1 d_2}{d_1 n_2} \\
\frac{n_1}{d_1} = \frac{n_2}{d_2} &\quad \text{iff} \quad n_1 d_2 = d_1 n_2
\end{aligned}
$$

Implementing Rational Arithmetic

```
def addRational(r: Rational, s: Rational): Rational =
  Rational(
    r.numer * s.denom + s.numer * r.denom,
    r.denom * s.denom)

def makeString(r: Rational): String =
  s"${r.numer}/${r.denom}"
```

```
makeString(addRational(Rational(1, 2), Rational(2, 3)))
```
> *7/6*

Note: `s"..."` in `makeString` is an *interpolated string*, with values `r.numer` and `r.denom` in the places enclosed by `${...}`.

Methods

One can go further and also package functions operating on a data abstraction in the data abstraction itself.

Such functions are called *methods*.

Example

Rational numbers now would have, in addition to the functions `numer` and `denom`, the functions `add`, `sub`, `mul`, `div`, `equal`, `toString`.

Methods for Rationals

Here's a possible implementation:

```
class Rational(x: Int, y: Int):
  def numer = x
  def denom = y
  def add(r: Rational) =
    Rational(numer * r.denom + r.numer * denom,
             denom * r.denom)
  def mul(r: Rational) = ...
  ...
  override def toString = s"$numer/$denom"
```

Remark: the modifier `override` declares that `toString` redefines a method that already exists (in the class `java.lang.Object`).

Calling Methods

Here is how one might use the new `Rational` abstraction:

```
val x = Rational(1, 3)
val y = Rational(5, 7)
val z = Rational(3, 2)
x.add(y).mul(z)
```

Exercise

Try this in a Visual Studio Code worksheet!

1. In your worksheet, add a method `neg` to class Rational that is used like this:

   ```
   x.neg           // evaluates to -x
   ```

2. Add a method `sub` to subtract two rational numbers.

3. With the values of `x`, `y`, `z` as given in the previous slide, what is the result of

   ```
   x - y - z
   ```

   ?

Data Abstraction

The previous example has shown that rational numbers aren't always represented in their simplest form. (Why?)

One would expect the rational numbers to be *simplified*:

- reduce them to their smallest numerator and denominator by dividing both with a divisor.

We could implement this in each rational operation, but it would be easy to forget this division in an operation.

A better alternative consists of simplifying the representation in the class when the objects are constructed:

Rationals with Data Abstraction

```
class Rational(x: Int, y: Int):
  private def gcd(a: Int, b: Int): Int =
    if b == 0 then a else gcd(b, a % b)
  private val g = gcd(x, y)
  def numer = x / g
  def denom = y / g
  ...
```

`gcd` and `g` are *private* members; we can only access them from inside the `Rational` class.

In this example, we calculate `gcd` immediately, so that its value can be re-used in the calculations of `numer` and `denom`.

Rationals with Data Abstraction (2)

It is also possible to call `gcd` in the code of `numer` and `denom`:

```
class Rational(x: Int, y: Int):
  private def gcd(a: Int, b: Int): Int =
    if b == 0 then a else gcd(b, a % b)
  def numer = x / gcd(x, y)
  def denom = y / gcd(x, y)
```

This can be advantageous if it is expected that the functions `numer` and `denom` are called infrequently.

Rationals with Data Abstraction (3)

It is equally possible to turn `numer` and `denom` into `val`s, so that they are computed only once:

```
class Rational(x: Int, y: Int):
  private def gcd(a: Int, b: Int): Int =
    if b == 0 then a else gcd(b, a % b)
  val numer = x / gcd(x, y)
  val denom = y / gcd(x, y)
```

This can be advantageous if the functions `numer` and `denom` are called often.

The Client's View

Clients observe exactly the same behavior in each case.

This ability to choose different implementations of the data without affecting clients is called *data abstraction*.

It is a cornerstone of software engineering.

Self Reference

On the inside of a class, the name `this` represents the object on which the current method is executed.

Example

Add the functions `lessThan` and `max` to the class `Rational`.

```
class Rational(x: Int, y: Int):

  def lessThan(that: Rational): Boolean =
    numer * that.denom < that.numer * denom

  def max(that: Rational): Rational =
    if this.lessThan(that) then that else this
```

Self Reference (2)

Note that a simple name `m`, which refers to another member of the class, is an abbreviation of `this.m`. Thus, an equivalent way to formulate `lessThan` is as follows.

```
def lessThan(that: Rational): Boolean =
  this.numer * that.denom < that.numer * this.denom
```

Preconditions

Let's say our `Rational` class requires that the denominator is positive.

We can enforce this by calling the `require` function.

```
class Rational(x: Int, y: Int):
  require(y > 0, "denominator must be positive")
  ...
```

`require` is a predefined function.

It takes a condition and an optional message string.

If the condition passed to `require` is `false`, an `IllegalArgumentException` is thrown with the given message string.

Assertions

Besides `require`, there is also `assert`.

Assert also takes a condition and an optional message string as parameters. E.g.

```
val x = sqrt(y)
assert(x >= 0)
```

Like `require`, a failing `assert` will also throw an exception, but it's a different one: `AssertionError` for `assert`, `IllegalArgumentException` for `require`.

This reflects a difference in intent

- `require` is used to enforce a precondition on the caller of a function.
- `assert` is used as to check the code of the function itself.

Constructors

In Scala, a class implicitly introduces a constructor. This one is called the *primary constructor* of the class.

The primary constructor

- takes the parameters of the class
- and executes all statements in the class body (such as the `require` a couple of slides back).

Auxiliary Constructors

Scala also allows the declaration of *auxiliary constructors*.

These are methods named `this`

Example Adding an auxiliary constructor to the class `Rational`.

```
class Rational(x: Int, y: Int):
  def this(x: Int) = this(x, 1)
  ...
```

Rational(2) > *2/1*

End Markers

With longer lists of definitions and deep nesting, it's sometimes hard to see where a class or other construct ends.

End markers are a tool to make this explicit.

```
class Rational(x: Int, y: Int):
  def this(x: Int) = this(x, 1)


  ...
end Rational
```

- And end marker is followed by the name that's defined in the definition that ends at this point.
- It must align with the opening keyword (`class` in this case).

End Markers

End markers are also allowed for other constructs.

```
def sqrt(x: Double): Double =
  ...
end sqrt

if x >= 0 then
  ...
else
  ...
end if
```

If the end marker terminates a control expression such as `if`, the beginning keyword is repeated.

Exercise

Modify the `Rational` class so that rational numbers are kept unsimplified internally, but the simplification is applied when numbers are converted to strings.

Do clients observe the same behavior when interacting with the rational class?

- 0 yes
- 0 no
- 0 yes for small sizes of denominators and nominators and small numbers of operations.

Evaluation and Operators

Classes and Substitutions

We previously defined the meaning of a function application using a computation model based on substitution. Now we extend this model to classes and objects.

Question: How is an instantiation of the class $\texttt{C}(e_1, ..., e_m)$ evaluted?

Answer: The expression arguments $e_1, ..., e_m$ are evaluated like the arguments of a normal function. That's it.

The resulting expression, say, $\texttt{C}(v_1, ..., v_m)$, is already a value.

Classes and Substitutions

Now suppose that we have a class definition,

$$\texttt{class C}(x_1, ..., x_m)\{ \ ... \ \texttt{def f}(y_1, ..., y_n) = b \ ... \ \}$$

where

- The formal parameters of the class are $x_1, ..., x_m$.
- The class defines a method $\texttt{f}$ with formal parameters $y_1, ..., y_n$.

(The list of function parameters can be absent. For simplicity, we have omitted the parameter types.)

Question: How is the following expression evaluated?

$$\texttt{C}(v_1, ..., v_m).\texttt{f}(w_1, ..., w_n)$$

Classes and Substitutions (2)

Answer: The expression $\texttt{C}(\texttt{v}_1, \ldots, \texttt{v}_m).\texttt{f}(\texttt{w}_1, \ldots, \texttt{w}_n)$ is rewritten to:

$$[\texttt{v}_1/\texttt{x}_1, \ldots, \texttt{v}_m/\texttt{x}_m][\texttt{w}_1/\texttt{y}_1, \ldots, \texttt{w}_n/\texttt{y}_n][\texttt{C}(\texttt{v}_1, \ldots, \texttt{v}_m)/\texttt{this}]\,\texttt{b}$$

There are three substitutions at work here:

- the substitution of the formal parameters $\texttt{y}_1, \ldots, \texttt{y}_n$ of the function $\texttt{f}$ by the arguments $\texttt{w}_1, \ldots, \texttt{w}_n$,
- the substitution of the formal parameters $\texttt{x}_1, \ldots, \texttt{x}_m$ of the class $\texttt{C}$ by the class arguments $\texttt{v}_1, \ldots, \texttt{v}_m$,
- the substitution of the self reference *this* by the value of the object $\texttt{C}(\texttt{v}_1, \ldots, \texttt{v}_n)$.

Object Rewriting Examples

```
Rational(1, 2).numer
```

Object Rewriting Examples

`Rational(1, 2).numer`

$\rightarrow$ $[1/\mathtt{x}, 2/\mathtt{y}]\ []\ [\mathtt{Rational}(1, 2)/\mathtt{this}]$ `x`

Object Rewriting Examples

`Rational(1, 2).numer`

$\rightarrow [1/x, 2/y]\ []\ [\texttt{Rational}(1, 2)/\texttt{this}]\ \texttt{x}$

$= 1$

Object Rewriting Examples

`Rational(1, 2).numer`

$\rightarrow$ $[1/\mathtt{x}, 2/\mathtt{y}]\ []\ [\mathtt{Rational}(1, 2)/\mathtt{this}]$ `x`

$=$ `1`

`Rational(1, 2).lessThan(Rational(2, 3))`

Object Rewriting Examples

`Rational(1, 2).numer`

$\rightarrow [1/x, 2/y]\ []\ [\texttt{Rational}(1, 2)/\texttt{this}]$ `x`

$=$ `1`

`Rational(1, 2).lessThan(Rational(2, 3))`

$\rightarrow [1/x, 2/y]\ [\texttt{Rational}(2, 3)/\texttt{that}]\ [\texttt{Rational}(1, 2)/\texttt{this}]$

`this.numer * that.denom < that.numer * this.denom`

Object Rewriting Examples

```
Rational(1, 2).numer
```

$\rightarrow$ $[1/x, 2/y]$ $[]$ $[\texttt{Rational}(1, 2)/\texttt{this}]$ `x`

$=$ `1`

```
Rational(1, 2).lessThan(Rational(2, 3))
```

$\rightarrow$ $[1/x, 2/y]$ $[\texttt{Rational}(2, 3)/\texttt{that}]$ $[\texttt{Rational}(1, 2)/\texttt{this}]$

```
    this.numer * that.denom < that.numer * this.denom
```

$=$

```
Rational(1, 2).numer * Rational(2, 3).denom <
    Rational(2, 3).numer * Rational(1, 2).denom
```

Object Rewriting Examples

```
Rational(1, 2).numer
```

$\rightarrow$ [1/x, 2/y] [] [Rational(1, 2)/this] `x`

$=$ `1`

```
Rational(1, 2).lessThan(Rational(2, 3))
```

$\rightarrow$ [1/x, 2/y] [Rational(2, 3)/that] [Rational(1, 2)/this]

```
    this.numer * that.denom < that.numer * this.denom
```

$=$
```
Rational(1, 2).numer * Rational(2, 3).denom <
    Rational(2, 3).numer * Rational(1, 2).denom
```

$\twoheadrightarrow$ `1 * 3 < 2 * 2`

$\twoheadrightarrow$ `true`

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.

For instance, we can add `min` and `abs` methods to class `Rational` like this:

```
extension (r: Rational)
  def min(s: Rational): Rational = if s.lessThan(r) then s else r
  def abs: Rational = Rational(r.numer.abs, r.denom)
```

Using Extension Methods

Extensions of a class are visible if they are listed in the companion object of a class (as in the code above) or if they defined or imported in the current scope.

Members of a visible extensions of class `C` can be called as if they were members of `C`. E.g.

```
Rational(1/2).min(Rational(2/3))
```

Caveats:

- 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.

```
Rational(1, 2).min(Rational(2, 3))
```

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.

```
Rational(1, 2).min(Rational(2, 3))
```

$\rightarrow$ $[\texttt{Rational}(1,2)/\texttt{r}]\ [\texttt{Rational}(2,3)/\texttt{s}]$ `if x.lessThan(r) then s else r`

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.

```
Rational(1, 2).min(Rational(2, 3))
```

$\rightarrow$ $[\texttt{Rational}(1,2)/\texttt{r}]\ [\texttt{Rational}(2,3)/\texttt{s}]$ `if x.lessThan(r) then s else r`

$=$

```
if Rational(2, 3).lessThan(Rational(1, 2)
then Rational(2, 3)
else Rational(1, 2)
```

Operators

In principle, the rational numbers defined by `Rational` are as natural as integers.

But for the user of these abstractions, there is a noticeable difference:

- We write `x + y`, if `x` and `y` are integers, but
- We write `r.add(s)` if `r` and `s` are rational numbers.

In Scala, we can eliminate this difference. We proceed in two steps.

Step 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 underscore character '_' counts as a letter.
- Alphanumeric identifiers can also end in an underscore, followed by some operator symbols.

Examples of identifiers:

```
x1      *      +?%&      vector_++      counter_=
```

Step 1: Relaxed Identifiers

Since operators are identifiers, it is possible to use them as method names. E.g.

```
extension (x: Rational)
  def + (y: Rational): Rational = x.add(y)
  def * (y: Rational): Rational = x.mul(y)
  ...
```

This allows rational numbers to be used like `Int` or `Double`:

```
  val x = Rational(1, 2)
  val y = Rational(1, 3)
  x * x + y * y
```

Step 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. E.g.

```
extension (x: Rational)
  infix def min(that: Rational): Rational = ...
```

It is therefore possible to write

```
r + s                                       r.+(s)
r < s              /* in place of */        r.<(s)
r min s                                     r.min(s)
```

Precedence Rules

The *precedence* of an operator is determined by its first character.

The following table lists the characters in increasing order of priority precedence:

```
(all letters)
|
^
&
< >
= !
:
+ -
* / %
(all other special characters)
```

Exercise

Provide a fully parenthesized version of

```
a + b ^? c ?^ d lessThan a ==> b | c
```

Every binary operation needs to be put into parentheses, but the structure of the expression should not change.