Functional languages treat functions as first-class values. This means that, as any other value, a function can be passed as a parameter and returned as a result.

Functions that take other functions as parameters or that return other functions as results are called **higher order functions**. First-order functions act on Ints, Longs, etc.

Let’s take a function that sums ints in the range:

def sumInts(a: Int, b: Int): Int =

if (a > b) 0 else a + sumInts(a + 1, b)

Let’s 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) 0 else cube(a) + sumCubes(a + 1, b)

Let’s take the sum of factorials of all the integers between a and b:

def fact(x : Int): Int =

if (x == 1) x

x \* fact(x - 1)

def sumFactorials(a: Int, b: Int): Int =

if (a > b) 0 else fact(a) + sumFactorials(a + 1, b)

We can pass f as a parameter to some function:

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

if (a > b) 0 else f(a) + sum(f, a + 1, b)

We can then write:

def sumCubes(a: Int, b: Int) = sum(cube, a, b)

Passing functions as parameters can lead to the creation of many small functions. Sometimes it is tedious to define these functions using def. **Anonymous functions** let us write functions without giving them a name.

Example

A function that raises an argument to cube:

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

The type of the parameter can be omitted if it can be inferred by a compiler from the context. Several parameters are separated by commas:

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

Anonymous functions are syntactic sugar.

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

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

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

Tail-recursive version of sum function is:

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

def loop(a: Int, acc: Int): Int =

if (b > a) 0

else loop(a + 1, f(a) + acc)

loop(a, 0)

In the sumInts and subCubes **we can get rid of the parameters a and b because they are passed unchanged.**

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

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

Let’s rewrite sum as follows:

def sum(f: Int => Int): (Int, Int) => Int =

def sumF(a: Int, b: Int): Int =

if(a > b) 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.

We can use consecutive stepwise application to call function sum without using sumCubes/sumInts:

sum(cube)(a, b)

sum(cube) applies sum to cube and returns the sum of the cubes function. sum(cube) is equivalent to sumCubes. This function is next applied to arguments (1, 10).

Generally, function application associates to the left:

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

We can rewrite this function:

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

if (a > b) 0 else f(a) + sum(f, a + 1, b)

As this function:

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

if (a > b) 0 else f(a) + sum(f)(a + 1, b)

This syntax is useful because one can write an expression like **sum(cube) which will be valid without parameters a and b.**

We can repeat the process n times:

def f(args\_1)...(args\_n-1)(args\_n) = E

is shown to be equivalent to:

def f = (args\_1 => (args\_2 => … (args\_n => E) … ))

The style of definition and function application is called currying.

Product can be evaluated like that:

def product(f: Int => Int)(a: Int, b: Int): Int =

if (a > b) 1 else f(a) \* product(f)(a + 1, b)

A function that **generalizes both product and sum functions:**

def mapReduce(f: Int => Int, combine: (Int, Int) => Int, zero\_val: Int)(a: Int, b: Int): Int =

if (a > b) zero\_val else combine(f(a), mapReduce(f, combine, zero\_val)(a + 1, b))

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

f(x) = x

For some functions f we can locate fixed points by starting with an initial estimate and then applying f in a repetitive way:

x, f(x), f(f(x)), f(f(f(x))) …

This is continued until the value does not vary anymore.

The function can be implemented like that:

val tolerance = 0.0001

def isCloseEnough(x: Double, y: Double) =

abs((x - y) / x) / x < tolerance

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

def iterate(guess: Double): Double =

val next = f(guess)

if (isCloseEnough(guess, next)) next

else iterate(next)

iterate(firstGuess)

Now this function can be called with:

fixedPoint(x => 1 + x / 2)(1)

We remember that square root of the number can be expressed through division:

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

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

However, if we use this function in the fixedPoint, it will not converge. In order to achieve convergence we can average successive values of the original sequence:

def sqrt(x: Double) = fixedPoint(y => (y + x / y) / 2)(1)

This technique is called **stabilizing by averaging**, it is general enough to be implemented as a separate function:

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

An sqrt function that uses averageDamp is:

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

Scala context-free syntax in Extended Backus-Naur form EBNF where:

* | denotes an alternative
* [...] denotes an option (0 or 1)
* {...} denotes a repetition (1 or more)

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

Let’s say we want to design a package for doing rational arithmetic. A rational number x/y is represented by two integers: its numerator x and its denominator y.

We can combine rational numerator and denominator in a class:

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

def numer = x

def denom = y

}

This definition introduces two new entities:

* a new type, named Rational
* a constructor Rational to create elements of this type

We call the elements of a class type objects. We create an object by prefixing an application of the constructor of the class with the operator new.

Example

val x = new Rational(1, 2)

println(x.numer)

println(x.denom)

We can also implement rational arithmetic:

addRational(r: Rational, s: Rational): Rational =

new Rational(r.numer \* s.denom + s.numer \* r.denom, r.denom \* s.denom)

def makeString(r: Rational) =

r.numer + “/” + r.denom

makeString(addRational(new Rational(1, 2), new Rational(2, 3)))

We can also implement these functions as methods inside class:

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

def numer = x

def denom = y

def add(that: Rational){

new Rational(

numer \* that.denom + that.numer \* denom,

denom \* that.denom

)

}

override def toString = numer + “/” + denom

def neg: Rational = new Ration(-numer, denom)

def sub(that: Rational) = add(that.neg)

}

The previous example has shown that rational numbers aren’t always expressed in their simplest form. One would expect them to be simplified: we can reduce them to their smallest numerator and denominator by dividing both with a divisor.

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

private def gcd(a: Int, b: Int): Int =

if (b == 0) a else gcd(b, a % b)

private val g = gcd(x, y)

def numer = x / g

def denom = y / g

def add(that: Rational){

new Rational(

numer \* that.denom + that.numer \* denom,

denom \* that.denom

)

}

override def toString = numer + “/” + denom

def neg: Rational = new Ration(-numer, denom)

def sub(that: Rational) = add(that.neg)

}

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

We can also convert numer and denom into vals in the case they are called often.

Ability to choose different implementations of the data without affecting clients is called data abstraction.

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

private def gcd(a: Int, b: Int): Int =

if (b == 0) a else gcd(b, a % b)

private val g = gcd(x, y)

def numer = x / g

def denom = y / g

def add(that: Rational){

new Rational(

numer \* that.denom + that.numer \* denom,

denom \* that.denom

)

}

override def toString = numer + “/” + denom

def neg: Rational = new Ration(-numer, denom)

def sub(that: Rational) = add(that.neg)

def less(that: Rational) = numer \* that.denom < that.numer \* denom

def max(that: Rational) = if (this.less(that)) that else this

}

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

Note that **a simple name x which refers to another member of the class is an abbreviation of this.x.** Thus, an equivalent way to formulate less is as follows:

def less(that: Rational) =

this.numer \* that.denom < that.numer \* this.denom

We can also add a requirement to our class so that the denominator should be non-zero:

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

require(y != 0, “denominator must be nonzero”)

private def gcd(a: Int, b: Int): Int =

if (b == 0) a else gcd(b, a % b)

private val g = gcd(x, y)

def numer = x / g

def denom = y / g

def add(that: Rational){

new Rational(

numer \* that.denom + that.numer \* denom,

denom \* that.denom

)

}

override def toString = numer + “/” + denom

def neg: Rational = new Ration(-numer, denom)

def sub(that: Rational) = add(that.neg)

def less(that: Rational) = numer \* that.denom < that.numer \* denom

def max(that: Rational) = if (this.less(that)) that else this

}

require 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.

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, failing assert will also throw an exception, but it’s a different one: AssertionError for assert and 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**

In Scala, a class implicitly introduces a constructor, it is called the primary constructor of the class.

The primary constructor:

* takes the parameters of the class
* executes all statements of the class body

We can also create a second constructor of the class that will take only one number as a parameter:

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

require(y != 0, “denominator must be nonzero”)

def this(x: Int) = this(x, 1)

private def gcd(a: Int, b: Int): Int =

if (b == 0) a else gcd(b, a % b)

private val g = gcd(x, y)

def numer = x / g

def denom = y / g

def add(that: Rational){

new Rational(

numer \* that.denom + that.numer \* denom,

denom \* that.denom

)

}

override def toString = numer + “/” + denom

def neg: Rational = new Rational(-numer, denom)

def sub(that: Rational) = add(that.neg)

def less(that: Rational) = numer \* that.denom < that.numer \* denom

def max(that: Rational) = if (this.less(that)) that else this

}

This second constructor calls the implicit primary constructor of the class Rational.

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

require(y != 0, “denominator must be nonzero”)

def this(x: Int) = this(x, 1)

private def gcd(a: Int, b: Int): Int =

if (b == 0) a else gcd(b, a % b)

def numer = x

def denom = y

def add(that: Rational) = {

new Rational(

numer \* that.denom + that.numer \* denom,

denom \* that.denom

)

}

override def toString = {

val g = gcd(numer, denom)

numer / g + “/” + denom / g

}

def neg: Rational = new Ration(-numer, denom)

def sub(that: Rational) = add(that.neg)

def less(that: Rational) = numer \* that.denom < that.numer \* denom

def max(that: Rational) = if (this.less(that)) that else this

}

In this case we’ll keep the number in the unsimplified form but will print it in the simplified form.

Any method with a parameter **can be used like an infix operator.** It is possible to write:

r add s instead of r.add(s)

Operators can be used as identifiers.

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 symbol
* alphanumeric identifiers can also end in an underscore, followed by some operator symbols.

Examples of identifiers:

x1 \* +?%& vector\_++ counter\_=

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

require(y != 0, “denominator must be nonzero”)

def this(x: Int) = this(x, 1)

private def gcd(a: Int, b: Int): Int =

if (b == 0) a else gcd(b, a % b)

def numer = x

def denom = y

def + (that: Rational) = {

new Rational(

numer \* that.denom + that.numer \* denom,

denom \* that.denom

)

}

override def toString = {

val g = gcd(numer, denom)

numer / g + “/” + denom / g

}

def unary\_ : Rational = new Rational(-numer, denom)

def - (that: Rational) = this + -that

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

def max(that: Rational) = if (this < (that)) that else this

}

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