Week 2

Higher-Order Functions

Functions are first-class citizens in functional languages

It means, a function can be passed as a parameter and returned as a result

Functions that can do that are called higher order functions

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

```
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)
```

and

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

A => B - is the type of a function that takes an argument of type A and returns a result of type B

Int => Int - from int to int

Anonymous functions

functions without names

$$\underbrace{(x: Int)}_{\text{parameter}} \Rightarrow \underbrace{x * x * x}_{\text{body}}$$

type can be omitted if can be derived

$$\underbrace{(x: Int, y: Int)}_{\text{several parameters}} \Rightarrow x + y$$

def sumInts(a... b...) = sum($\underbrace{x \Rightarrow x}_{\text{shorter!}}$, a, b)

def sumCubes(...) = sum ($\overbrace{x \Rightarrow x * x * x}$, a, b)

Currying

Can a ~~and~~ b get passed unchanged ~~to~~
Can we make them even shorter?

def sumInts(a... b...) = sum(x => x, a, b)

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

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

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

    sumF

}
```

sum now a function that returns another function

the returning function knows f and does the summing

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

sumCubes(1, 10) !

↓

the same as

$\underbrace{\text{sum(cube)}}_{\text{return a function}}\ \underbrace{(1, 10)}_{\text{apply the arguments to}}$ == (sum(cube)) (1, 10) and

$$\text{def sum}\underbrace{(f: Int => Int)}_{\text{I parameter list}}\underbrace{(a: Int, b: Int)}_{\text{II parameter list}}: Int =$$

$$\text{if } (a > b)\ 0 \text{ else } f(a) + sum(f)(a+1, b)$$

There's a special syntax in Scala for functions that return functions

general syntax:

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

which is equivalent to

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

  def $g(args_n) = E$

  g

$\}$

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

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

=>

def $f = (args_1 => (args_2 => ... (args_n => E) ...))$

This style is called <u>currying</u>

What's the type of

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

?

Answer:

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

Finding a fixed point: Example

x is a fixed point of f if

$f(x) = x$

For some functions we can find the fixed point by starting with an initial estimate and then by applying f again and again

$x \quad f(x) \quad f(f(x)) \ldots \quad f(f(f(x))) \ldots$

until it doesn't vary anymore (or the change is very small)

```
val tolerance = 0.0001

def isCloseEnought(x: Double, y: Double) =
    abs((x - y) / x) / 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 (~~next~~ firstGuess)
}
```

sqrt (x) = the number y such that $y * y = x$

or, (if divide by y)

$$y = \frac{x}{y} \Rightarrow$$

sqrt(x) is a fixed point of the function

$$y = \frac{x}{y}$$

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

but that won't converge

it oscillates 1.0 - 2.0 - 1.0 - 2.0 ...

We can prevent it from varying too much by averaging successive values

def sqrt(x: Double) =

    fixedPoint(y => (x + ~~y~~ / ~~y~~) / 2)(1.0)

This technique of stabilizing by averaging is general enough to be a function

def averageDamp(f: Double => Double)(x: Double) =

    (x + f(x)) / 2.

def sqrt(x: Double) =

    fixedPoint(averageDamp(x => x/y))(1.0)

Summary

- Functions are essential abstractions - they allow us to introduce general methods to perform computations as
  - explicit
  - named

  components } our programming language
- These abstractions can be combined with higher-order functions to create new abstractions

Scala Syntax Summary

Extended Backus-Naur form (EBNF)

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

Type = SimpleType | FunctionType

FunctionType = SimpleType '=>' Type |
'(' [Types] ')' '=>' Type

SimpleType = Ident
Types = Type {',' Type}

a type can be

- a numeric type:
  Int, Double, Byte, Short, Char, Long, Float
- Boolean
- String
- a function 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 '}'
```

Expression can be

- An identifier such as x, isGoodEnough
- A literal 0, 1.0, "abc"
- A function operator, like sqrt(x)
- An operation application like -x, y+x
- A selection like math.abs
- A conditional expression if (x<0) -x else x
- A block { val x = math.abs(y) ; x * 2 }
- An anonymous function, 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 def square (x: Int) = x * x
- a value definition , val y = square (2)

A parameter can be

- a call-by-value parameter , (x: Int)
- a call-by-name parameter (x: => Double)

Functions and Data

how functions create and encapsulate data structures

· Rational numbers

We want to design a package for doing rational arithmetic

A rational number is represented by 2 integers

$\frac{x}{y}$ — numerator / — denominator

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

    def numer = x
    def denom = y      } members

}
```

This definition introduces

- a new type – Rational
- a constructor to create elements of this type

Elements of a class type – objects

$\underbrace{\text{new Rational}}_{\text{constructor}}(1, 2)$

~~x.number~~
x.numer
x.denom

infix operator '.' for selecting members

Rational Arithmetics:

- $\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$

- $\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{n_1 d_2 - n_2 d_1}{d_1 d_2}$

- $\frac{n_1}{d_1} \cdot \frac{n_2}{d_2} = \frac{n_1 n_2}{d_1 d_2}$

- $\frac{n_1}{d_1} / \frac{n_2}{d_2} = \frac{n_1 d_2}{d_1 n_2}$

- $\frac{n_1}{d_1} = \frac{n_2}{d_2}$ iff $n_1 d_2 = d_1 n_2$

```
def 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
```

But we can add these functions to the data abstraction itself

Such functions are called <u>methods</u>

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

    def numer = x
    def denom = y

    def add (r: Rational) =
        new Rational (numer * r.denom +
                      r.numer * denom,
                      denom * r.denom)

    def mul (r: Rational) = ...


    override def toString = numer + "/" + denom

}
```

declares that this method
redefines a method that
already exists

```
val x = new Rational (1, 3)
val y = new Rational (5, 7)

    x.add(y).mul(z)
```

<u>But</u> Rational numbers should be
simplified

```
class Rational (...) {

    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
...
```

gcd and g are <u>private</u> members – we can access them only from inside the Rational class

The ability to choose different implementation of the data without affecting clients is called <u>data abstraction</u>

```
class Rational (...) {

    ...

    def max (that: Rational) =

        if (this.less(that)) that
        else this

}
```

reference to the current object

<u>Preconditions</u>, <u>Assertions</u>

```
class Rational (...) {

    require (y > 0, "denominator must be
                     positive")

    throws IllegalArgumentException
}
```

to enforce a precondition

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

used to check the code of the function

AssertionError

Constructors

a class implicitly introduces a constructor -
this is the primary constructor.

The primary ~~cons~~ constructor:

- takes the parameters of the class
- executes all statements in the class body

We also can declare auxiliary constructors - they are methods called "this"

```
class Rational(...) {

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

    ...

}
```

new Rational(2) -> 2/1

Classes and Substitution

$$\text{new } C(\underbrace{e_1, \dots, e_m}_{})$$

$$\Downarrow$$

e_i evaluated

$$\text{new } C(v_1, \dots, v_m).$$

$$\text{new } C(v_1, \dots, v_m).\, f(w_1, \dots, w_n)$$

$$\Downarrow$$

$$[\{w_i : y_i\}]\ [\{v_i : x_i\}]\ [\text{new } C \dots / \text{this}]\ b$$

name-value   name-value

$$\text{new Rational}(1, 2).\text{numer}$$

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

$= 1$

...

Operators

it's possible to write

r add s   instead of   r.add(s)

r less s   r.less(s)

And operators can be used as identifiers

An identifier can be:

- alphanumeric - starts with a letter, followed by letters and numbers
- symbolic - started with an operator symbol followed by other operator symbols
- '_' counts as a letter
- alphanumeric identifiers can also end in an underscore, followed by some operator symbol

x1   *   +?%&   vector_++   counter_=

```
class Rational (...) {
    ...
    def + (r: Rational) = ...
    def - (r: Rational) = ...
    def * ...
    ...
}
```

```
val x = new Rational ..
val y = new Rational ..
```

`x * x + y * y`

$\Downarrow$

`(x * x) + (y * y)`

~~Prede~~ Precedence of an operator is determined by its first character

The priorities:

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

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

$\Downarrow$

`((a + b) ^? (c ^? d)) less ((a ==> b) | c)`

Assignment - Sets.

- type alias

  type Set Int => Boolean

  takes int

  $(x : Int) \Rightarrow x < 0$

  a set with negative integers

  just a function that returns true if element x is in the set.

  def contains(s: Set, elem: Int): Boolean =
      s(elem)

- forall (s : Set, $\underbrace{p : Int \Rightarrow Boolean}_{\text{predicate}}$): Boolean -

  universal quantifier - return true if all elements of ~~the~~ set s the predicate p is true

- exists (s: Set, p: Int => Boolean): Boolean

  existential quantifier - returns true if at least one element matches the predicate.

  Implementation in terms of forall:

  !forall(s, p) and !forall(s, $\bar{p}$)

  $\bar{p}$ = inverse predicate

• map (s: Set, f: Int => Int): Set

transforms a set S by applying function f
to each element

$S \xrightarrow{f} Y$

element $y \in Y$ iff there exists $x \in S$

original set

such that $f(x) = y$

(so if we are able to ~~funct~~ apply a predicate
"exists $f(x) == y$" to S and get true -
y is in Y).