Scala Notes

Peter Thompson

July 1, 2020

Contents

Part I

Functional Principles

Chapter 1

Functional Paradigms

1.1 Functional Programming

Mutation is where an attribute of a variable can change while the identity of the variable is maintained. For example, could define a polynomial class, then set a certain coefficient to a particular value.

Functional programming is a programming strategy that avoids mutation/reliance on state information. Immutable values are used. These can be transformed, but the idea is to minimise side effects. We don't want to pass an argument to a function that will then modify that argument. Input goes in, return value comes out with inputs unchanged.

The restricted definition of a functional programming framework is one in which there are no mutable variables, assignments, or imperative conttrol structures In a wider sense, functional programming can be carried out in any language that allows the construction of elegant programs that focus on functions

In scala, functions are first class objects. They can be treated and passed around just like any other variable (as in python)

1.2 call-by-name, call-by-value

Functions arguments in scala can be handled two ways. The argument can be called by name, or called by value.

```
//call by value
def CBVfunc(a:Int):Int = {...}
```

Function CBVfunc takes an int, returns an Int. The integer argument a is evaluated when the function is called.

```
//call by name
def CBNfunc(a: =>Int): Int = {...}
```

Function CBNfunc takes an integer argment (a), which is evaluated when it needs to be (or not at all!).

Not all functions terminate, infinte loops are a thing

Both call by value and call by name will reduce to the same outcome provided

- the reduced expression consists of pure functions (no state information/-side effects?)
- both evaluations terminate (no infinte loops)

if call by value terminates, then call by name will also converse is not guarenteed: call by name termination does not imply call by value termination in call by name, unused arguments are not evaluated

- could have a function that takes two arguments. The second argument is not used (always)
- could pass a non-terminating input to the CBN function, which is not used. no big deal
- call by value will try to evaluate it and get stuck

Below is an example of a function that will terminate when arguments are called by name but not when called by value.

```
// an infinite loop, this run indefinitely when evaluated
def loop = loop

// call by value function, arguments are evaluated once when the
    function is called
def mooseCBV(a: Int, b: Int):Int = a

// this evaluates loop, starting the infinte loop...
mooseCBV(1,loop)

// same as above, but using call-by-name (=>)
def mooseCBN(a: =>Int, b: => Int) = a

// returns 1, the loop is not evaluated
mooseCBN(1,loop)
```

Scala uses call by value by default, unless the function arguments are defined with `=>`

1.3 Conditionals

Boolean operations don't always need to evaluate the right hand operand (short circuit evaluation)

```
true && e -> e
false && e -> false
true || e -> true
false || e -> e
```

Things can be defined by name or by value. so `def x =loop` is a function, it is not evaluated untill it needs to be (call by name).

`val x =loop` evaluates to loop (call by value) immediately. this will kill your scala session/repl.

1.4 Recursion

Recursive functions must always have their return type explicitly defined (to make the compiler's life easier).

1.4.1 Tail Recursion

If a function calls itself as its last action, the stack frame can be reused Essentially it acts the same as a loop

If a functions last action is to call a function, (maybe the same, maybe different function), then the stack frame can be used - this is a tail-call (tail recursion is recursive tail-calling).

Tail recursive factorial example(works)

```
def factorial(N:Int) = {
    @scala.annotation.tailrec
    def currentProd(n:Int, prod:Int) :Int = {
        if (n==0) prod
        else currentProd(n-1,n*prod)
    }
    currentProd(N,1)
}
```

A lot of loops can be replaced by tail recursion. Usually this involves defining an inner function for the actual recursion, which accepts an accumulator argument in addition to other parameters. Recursive invocations of the inner function pass the current value of the accumulator, or return something when termination condition is met. For tail recursion (or recursion in general), it seems helpful to define the termination conditions at the very begining, then figure out the remaining logic.

1.5 Blocks and Scope

A block is defined by curly braces `{}`

Definitiones inside a block are invisible outside the block. Definitions from outside the block are visible inside, provided they have not been shadowed. A lot of object oriented functionality can be warngled from scopes and closures. Methods defined within a class constructor have acesss to the parameters of that constructor, even if those parameters are not assigned to fields of the class. For example:

```
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean = if (x < elem) left.contains(x)
    else if (x > elem) right.contains(x)
    else true
    ...
```

the method `contains` refers to `elem`.

1.6 higher order functions

functions are first class values they can be passed and returned a function that does this is called a higher order function

this can be used to factor out common procedures. For example,

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

defines a function sumFunc, that takes two integers and a function that takes an Int and returns an Int (`Int => Int`) For example, we could sum all squares or cubes between 2 and 5 by calling

```
sumFunc(2,5,square)
sumFunc(2.5.cube)
```

The notation `A => B` is a function type. it is a type that defines a mapping from type A to type B (by a function)

1.7 anonymous functions

strings exist as literals. We can just write ”abc”, and the compiler knows it to be a string. We don't need to do `def str ="abc"; println(str)` instead `println("abc")` works just fine Same can be done with functions, we don't always need to define a function, we can define anonymous functions as needed. (same as lambda functions in python)

these are defined like this

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

the type of x can be omitted if it can be inferred.

anonymous functions are syntactic sugar: `(x:Int)= x*x` and `def f(x:Int)= x*x;` `f` evaluate to the same.

tail recursive sum

```
def sum(f: Int => Int, a:Int, b:Int) = {
    @scala.annotation.tailrec
    def doSum(total:Int, aval:Int):Int = {
        if (aval > b) total else doSum(total + f(aval),aval+1)
    }
    doSum(0,a)
}
```

Chapter 2

currying

from the scala docs

> "Methods may define multiple parameter lists. When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments."

For example, consider a function `sum`, which takes a function `f` and returns a function taking two integers as parameters (the bounds).

```
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
}
```

The function sum now takes a function, and returns a function (return type is `(Int,Int)=> Int`). the returned function will take two Ints and return one. In this case, when an initial function (f) is passed into sum, it will return a function that sums the initial function (f) within the supplied bounds.

so, we could do

```
def sumCubes = sum((x:Int) => x*x*x)
def sumSquares = sum((x:Int) => x*x)
// and then invoke them
sumSquares(2,3) // 13
sumCubes(4,7) // 748
```

Alternatively, we could invoke the returned function directly

```
val moose = sum((x:Int)=>x*x) (2,3) //13
```

Again, there is some syntactic sugar for currying. For example

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

is equivalent to the definition of sum above, but without the definition of the inner function. When invoked as `sum(func)` the return type will be a function that takes two integers as parameters: the bounds a and b of the original sum function (the second parameter list). As mentioned above, when a function taking multiple parameter lists is invoked with one or more parameter lists absent, then the return type is a function that takes the missing parameter lists.

2.1 multiple parameter lists

Functions can be specified with multiple sets of parameter lists

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

For $n > 1$, this is equivalent to

```
def f(args_1)...(args_n-1) = {def g(argsn) = E ; g}
// or
def f(args_1)...(args_n-1) = (args_n => E)
```

Carrying this through gives

```
def f = (args1 => (args2 => ...(argsn => E)))
```

named after Haskell Brooks Curry (same guy Haskell language is named after).
As an example, we can write a function that calculates the product of values of a function for points on an interval. We can then define a factorial function in terms of these products.
Our original product and factorial functions look like this

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

def factorial(n:int) = product((x:Int) =>x)(1,n)
```

Now we will use currying to generalise. In this case we will calculate the sum and product of squares (rather than factorials)

```

def CumulativeFunctionOperation(operation: (Int,Int) => Int,
    initVal:Int)(f:Int=>Int)(a:Int,b:Int):Int = {
```

```
    if (a>b) initVal else
        operation(f(a),CumulativeFunctionOperation(operation,initVal)(f)(a+1,b))
}

def sum2:(Int=>Int)=>(Int,Int)=>Int =
    CumulativeFunctionOperation((x:Int,y:Int)=> x+y,0)

sum2(x=>x*x)(2,3) // 13

def prod2:(Int=>Int)=>(Int,Int)=>Int =
    CumulativeFunctionOperation((x:Int,y:Int)=> x*y,1)

prod2(x=>x*x)(2,3) // 36
// factorial
def fact(n:Int) = prod2(x=>x)(1,n)
```

CumulativeFunctionOperation is a form of map-reduce. The operation being apply is a reducer (it takes two inputs and returns a single value). The supplied function is the mapper, for sum2 and prod2 this is the anonymous function `x=>x*x` which computes squares, for fact this is the identity function `x=>x`. The bounds of CFO define the sequence that we are map/reducing. This is pretty neat. These functions are in the worksheet "curying.sc" under week2/misc_worksheets project

2.2 Aside - functional implementation of sets

The week 2 assignment was interesting. A set can be implemented as a function that returns true if it contains the supplied argument. The union of two sets is then the "or" of their charateristic functions, and the intersection the "and". The implementation of a "map" function took me a while to figure out. Instead of thinking of map as `[f(x) for x in set]`, we define the mapped characteristic as a check to see whether the initial set contains (any) element that would map to x. It's a bit backwards.

```
def map(s: Set, f: Int => Int): Set = (x:Int) => exists(s,(y:Int)=> f(y)
    == x)
// exists returns true if any element in the set s satisfies the
    supplied condition, false otherwise.
```

The code for this assignment is under `week2/funsets/src/main/scala/funsets/FunSets.scala`

Chapter 3

Classes

Classes are a type of object that contain methods (member functions) and fields (member data). A class representing a rational number is defined below. Some more examples related to classes are in the rational worksheet (`week2/misc\_worksheets/rationals.sc`).

```
class Rational(x:Int, y:Int) {
  def method(args:Int) = block
}
```

The class definition is the constructor - a new instance of Rational can be created using `val rat =new Rational(2,3)`. Class methods are public by default, private methods should be prefaced with the `private` declaration. Overriding methods should be prefaces with the `override` declaration.

In a class method, the instance of the class can be referenced with `this` (In lectures, the argument to class methods is often "that", so multiplication could be written `new Rational(this.numer*that.numer,this.denom*that.denom)`)

Class fields can be explicitly assigned in the constructor

```
class foo(A: Int) {
  val memberInt = A
}
```

or alternatively the fields can be defined implicitly in the constructor - `class foo(val memberInt: Int){}` . Both of these definitions are equivalent.

Instances of a class are objects. This can be ambiguous, as objects are a thing in Scala (see below)

3.1 Aside - require and assert

Require is a function that can be used to make sure the class being initialised meets some conditions, for example see the rational worksheet (week2/misc_worksheets/rationals.sc).

We ırequire that a class be initialised with a nonzero denominator. require takes a condition and a string as arguments. If the condition is not met, then an `illegalArgumentException` is thrown and the string is printed.

The `assert` function behaves similarly. Assert also takes a condition and a string, but throws an AssertionError when the condition is not met. The intent is that require enforces a precondition on the caller of a function, while assert is used to check the code of the function itself. Require is to make sure the function is called as intended, assert is to check that things are not borked in the internals.

3.2 Alternate constructors

Alternate constructors can be created by defining "this" as a function that takes alternate arguments (by signature), and then invoking the primary constructor appropriately (again, using this) For example, an alternate constructor for Rational could be defined (within Rational) as `def this(x:Int)=this(x,1)`. Then `val moose =new Rational(3)` will be mapped to `Rational(3,1)`.

3.3 Objects

Objects are defined similarly to classes, but using the keyword `object` instead of `class`. Objects can also inherit from (extend) other classes and traits. The difference is that objects are singleton, that is, only a single instance of an object may be created. Objects may be referenced in the scope they are defined in, or imported from another package.

3.3.1 companion objects

If an object has the same name as a class then it is a companion object. In scala, class or static methods do not exist. Things like class methods can be defined in a companion object, class instances can always access the methods of their companion objects

This example is taken from the scala docs `https://docs.scala-lang.org/tour/singleton-objects.html`

```
import scala.math._
case class Circle(radius: Double) {
  import Circle._
  def area: Double = calculateArea(radius)
}
object Circle {
  private def calculateArea(radius: Double): Double = Pi * pow(radius,
      2.0)
}
val circle1 = new Circle(5.0)
circle1.area
```

```
}
```

Objects can be useful for defining factory methods.

3.4 Defining operators/ infix notation

Any method with a parameter can be used like an infix operator. `r add s` is equivalent to `r.add(s)`, `r less s` to `r.less(s)`, etc.

Operators may also be used as identifiers. Valid identifiers in scala can have the following forms

- alphanumeric - starts with letter, followed by letters/numbers
- symbolic - start with operator symbol `+:?~#`, followed by other operator symbols
- underscore - counts as letter
- alphanumeric can end in _ followed by operator symbols

so `x_, x_+*, x1, *, vector_++, counter_=` are all valid identifiers. Things like `+, -, <, >` can just be defined as class methods.

unary operators should be defined using the prefix `unary_`. For example, there is a distinction between the binary version of "-" (subtraction) and the unary "-" (negation). `-Foo` is equivalent to `Foo.unary_-()`

3.5 operator precedence

Details on operator precedence can be found here: `https://docs.scala-lang.org/tour/operators.html`

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

can be rewritten (parenthesised) as

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

3.6 Classes, hierarchies, and dynamic binding

Classes (and objects) can inherit from existing classes or traits.

```
class baseAdder() {
  def addTwo(x:Int) = x +2
}
class badAdder() extends baseAdder {
  def addThree(x:Int) = x + 4
  override def addTwo(x: Int) = x +1
}
val moose = new baseAdder()
moose.addTwo(2) // Int = 5
val caribou = new badAdder()
caribou.addTwo(1) // Int = 2
caribou.addThree(4) //Int = 8
```

3.6.1 abstract classes

Abstract classes, like in c++, are classes with definitions that lack implementations. They are prefaced with the keyword `abstract`. If a class is defined as abstract, then it can not be instantiated using `new` (compiler will give an error).

```
abstract class IntSet {
    def incl(x:Int): IntSet
    def contains(x:Int) Boolean
    def union(other: IntSet): IntSet
}
```

3.6.2 traits

Traits are like interfaces in Java. They can declare methods and fields but provide no implementation. Traits take no parameters, and can not be instatiated.

```
trait Pet {
  val name: String
}
```

3.6.3 subtyping

We use the extend keyword to derive a class.

```
class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean = if (x < elem) left.contains(x)
    else if (x > elem) right.contains(x)
    else true
  def incl(x:Int): IntSet = if (x < elem) new
      NonEmpty(elem,left.incl(x), right)
    else if (x > elem) new NonEmpty(elem,left, right.incl(x))
    else this
  override def toString = s"{ ${left} ${elem} ${right} }"
  override def union(other: IntSet) =
      left.union(right).union(other).incl(elem)
}
class Empty extends IntSet {
  def contains( elem: Int): Boolean = false
  def incl(x:Int): IntSet = new NonEmpty(x,new Empty, new Empty)
  override def toString = "."
}
```

this implies that both empty and nonempty meet all the criteria of intset (implement union, contains and incl), but may have additional functionality. Instances of thee classes can be used whenever IntSets are required. IntSet is the **superclass** of both Empty and NonEmpty (which are **subtypes** of IntSet). If no extend clause is given when defining a class, then the standard java class "object" is assumed.

In java and scala a class can only have one superclass (not true in python or c++). A class may only extend from a single class (or trait). If there are several natural (potential) superclasses, use traits instead.

```
trait Planar {
  def height: Int
  def width: Int
  def surface = width*height
}
class Square extends Shape with Planar with Movable with ...
```

3.6.4 dynamic binding

Scala can use dynamic binding. The type of an object is determined at runtime. For example, a function f may take an instance of class A as an argument, and invoke `A.method()`. There may be a subclass B derived from A, which can override `A.method()`. At runtime, the type of the object passed to f is checked, and either A.method or B.method is invoked. Static binding is where the type of the object is known, for example

```
val theInstance = new A()
theInstance.method()
```

will invoke `A.method()`.

Chapter 4

Scala class Hierarchy and Packages

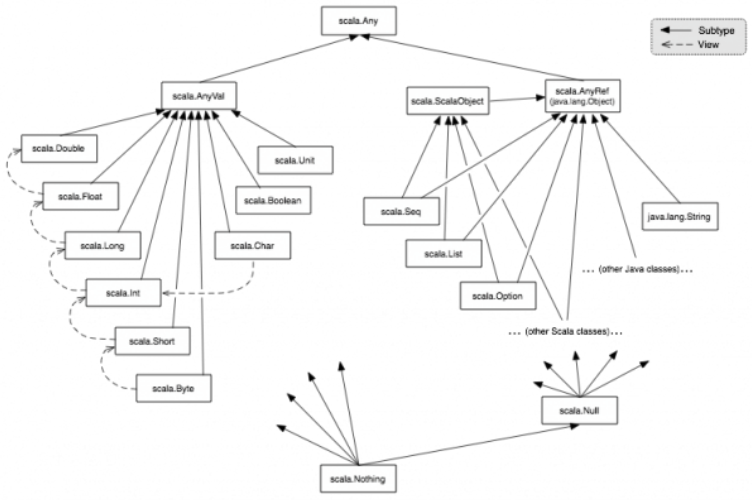

Figure 4.1: scala type hierarchy

- **Any** is the base type of all types.
- **AnyVal** inherits from Any. Numeric types (boolean, Int) inherit from AnyVal
- **Any** defines methods '==','!=', toString

Nothing is at the bottom of scala's type hierarchy. it is a subtype of every other type. Nothing can signal abnormal termination. Empty lists/containers can have elememt type Nothing

Null is a subtype of all reference types, and the type of the null value (null is the value, Null is the type)

Chapter 5

placeholder

5.1 for comprehension

backward arrows `<-` call flatmap final `yield` calls map

5.2 list functionality

sublists and element access. The list **xs** has the following methods available

- **xs.head** - first element in list
- **xs.tail** - remainder of list, with first element removed
- **xs.last** - last element of list
- **xs.init** - list containing all elements except first
- **xs.length** - number of elements in list
- **xs.take(n)** - sublist containing the first n elements of xs
- **xs.drop(n)** - sublist containing all remaining elements of xs, after the first n have been removed
- **xs(n)** - the nth element of list xs. The item access function is implemented through the apply function(xs.apply(n)), which can be invoked through just xs(n).
- **xs.splitAt(n)** - splits the list at position n, returns two lists in a tuple

other list operations

- **xs ++ ys** - concatenates xs and ys. This can also be done with the cons-like operator xs:::ys.

- **xs.reverse** - reverse the list
- **xs.updated.(n,x)** - replace the nth element with x
- **xs.indexOf(x)** - get the index of the first occurence of x (or -1 if x does not occur in xs)
- **xs.contains** - true if xs contains x, false otherwise

5.3 Pairs and Tuples

Some examples in mergesort.sc `week6/misc_worksheets/mergesort.sc`.

Functions can return multiple objects wrapped in a tuple, for example splitAt: `def splitAt(n:Int):(List[A],List[A])`. This returns a pair of lists, the first list contains the first n elements of the original, the second contains the remaining elements. Pairs in scala can be written `(x,y)`.

Pairs can be generalised to Tuples, which are collections of more than two elements.

A tuple is an instance of a parameterised scala type scala.Tuple$N[T_1, ..., T_N]$ The expression $(e_1, ...e_N)$ is equivalent to scala.Tuple$N(e_1, ...e_N)$. Tuples can also be used in pattern matching.

TupleN classes follow the definition below

```
case class Tuple2[T1,T2](_1: +T1,_2: +T2) {
  override def toString = "(" + _1 +", " + _2 + ")"
}
```

fields can be accessed by the names `_1, _2` etc. As tuples are case classes, they can be used for pattern matching (see above worksheet).

5.4 Implicit Parameters

Would like to extend our mergesort to work on arbitrary parameter types. If we just specify a type parameter $[T]$ in the constructor (and don't do anything else), we'll get an error as the $<$ operator is not defined for all types. A solution is to take a function as an argument, and use that to make the $x < y$ comparison.

There is, however, a class in the scala standard library that handles ordering - `scala.math.ordering[T]` provides ways for ordering objects of type T

Instead of parameterising our function with a custom function for parameterisation, we could use Ordering for comparison instead:

```
def msort_ord[T](xs:List[T])(ord: Ordering[T]): List[T] = {
...
case (x::xs,y::ys) => if (ord.lt(x,y)) x::merge(xs,y::ys) ...
...
merge(msort_ord(left)(ord),msort_ord(right)(ord)) }
}
```

```
msort_ord(myList2)(Ordering.String))
```

even better is to declare *ord* as an *implicit* parameter `def msort_ord[T](xs:List[T])(implicit ord: Ordering[T]): List[T] =...` Then ord no longer needs to be specified with the msort function is invoked. The compilier will figure out what should be used. The compiler will search for an implicit definition that

- is marked `implicit`
- has a type compatible with `T`
- is visible at the point of the function call, or is defined in a companion object associated with T.

If there is a single (most specific) definition, it will be taken as the argument. Otherwise an error will be thrown

5.5 More Lists

5.5.1 Higher order List functions

- **xs.map** - takes a function, returns a list formed by applying the function to each element of the list
- **xs.filter** - takes a function, returns a subset of the list containing all elements for which the function evaluates to true
- **xs.filterNot** - takes a function, returns a subset of the list containing all elements for which the function evaluates to false
- **xs.partition** - takes a function evaluating to bool, returns two lists, one containing elements that evaluate to true, the other elements that evaluate to false function evaluates to false
- **xs.takeWhile** - takes a function evaluating to bool, returns all elements up to (excluding) the first that evaluates to false
- **xs.dropWhile** - takes a function evaluating to bool, returns all elements after (including) the first that evaluates to false
- **xs.span** - takes a function evaluating to bool, returns two lists. The first list is equivalent to takeWhile, the second to dropWhile

5.6 Higher order List functions

- **xs.map** - takes a function, returns a list formed by applying the function to each element of the list

- **xs.filter** - takes a function, returns a subset of the list containing all elements for which the function evaluates to true
- **xs.filterNot** - takes a function, returns a subset of the list containing all elements for which the function evaluates to false
- **xs.partition** - takes a function evaluating to bool, returns two lists, one containing elements that evaluate to true, the other elements that evaluate to false function evaluates to false
- **xs.takeWhile** - takes a function evaluating to bool, returns all elements up to (excluding) the first that evaluates to false
- **xs.dropWhile** - takes a function evaluating to bool, returns all elements after (including) the first that evaluates to false
- **xs.span** - takes a function evaluating to bool, returns two lists. The first list is equivalent to takeWhile, the second to dropWhile

5.6.1 reduction of lists

reduction - reduces a list to a single value. For example, the sum/product functions we implemented earlier.

The reducLeft method can be used to reduce a list.

```
List(x1,x2,...,xn).reduceLeft(op) = ( ... ( x1 op x2) op x3) ...) op xn
```

So sum and product could be written as

```
def sum(xs:List[Int]) = (0::xs).reduceLeft((x,y) => x+y)
def product(xs:List[Int]) = (1::xs).reduceLeft((x,y) => x*y)
```

reduceLeft can not be applied to an empty list. FoldLeft can. Foldleft takes a n accumulator variable in addition to the reduction function. If the list is empty, then FoldLeft returns the value of the accumulator.

5.6.2 (

aside - anonymous parameters) Instead of passing reduceLeft the function `(x,y) => x + y`, we can simply write `(_ + _)` each underscore (_) represents a new parameter, going from left to right. The parameters are defined at the next outer pair of parentheses) or the whole expression).

Analogously to reduceLeft and foldLeft, there exist functions foldRight and reduceRight, which start at the tail of the tree and work towards the beginning.

In cases where the operation is associative and commutative, then it does not matter whether foldLeft or foldRight is used. If this is not true then the direction in which we fold will affect the result. Consider list concatenation `(xs foldleft ys)(_ :: _)`. Beacuse the cons operator expects an element of type T as its first parameter and of List[T] for it's second, if foldRight is used then an error will be thrown.

5.7 logic

We can prove that a program is correct if we can prove that definitions in the program obey certain laws. One of the core claims about functional programming is that it lends itself more easily to reasoning about programs.

Consider concatenation. We expect concatenation to satisfy the following properties

```
(xs ++ ys) ++ zs == xs ++ (ys ++zs) //associativity
xs ++ Nil == xs
Nil ++ xs == xs
```

We can use *structural induction do prove these. This is similar to proof by induction in mathematics (Assuming a rule is true for some n, prove that this implies truth for n+1. If the rule can be shown to be true for a particular value of n, then it is proven to be true for all n above this value).*

Referential transparency - a proof can freely apply reduction (term rewriting) steps as equalities to some part of a term. This works because functional programs don't have side effects, so a term is equivalent to the term to which it reduces.

5.7.1 S

tructural induction To prove a property P(xs) for all lists xs -

- *show that P(Nil) holds*
- *for a list xs and some element x, show the* ***induction step*** *holds*
  - *if P(xs) holds, then P(x::xs) also holds*

For the inductive step, we assume that the hypothesis is true for a given list xs, and then show that it is also true for (x::xs).

We will show that concatenation is associative, that is, `(xs ++ ys)++ zs ==xs ++ (ys ++zs)` *Our definition for concatenation was*

```
def concat[T](xs:List[T], ys:List[T]) = xs match
{
  case Nil => ys
  case x::xs1 => x :: concat(xs1,ys)
  }
```

From the above implementation, we can conclude

- *Nil ++ ys = ys*

- *(x ::xs1) ++ ys = x:: (xs1 ++ ys)*

which satisfies the nil case

For the induction step, we wish to show that `((x::xs)++ ys)++ zs =x::(xs ++ (ys ++ zs))`

```
((x::xs) ++ ys) ++ zs = (x::(xs ++ ys)) ++ zs
// from 2nd line above
= x::((xs ++ ys) ++ zs) // same again
= x::(xs ++ (ys ++ zs)) // by induction hypothesis
```

last line is the rhs above. As we have now proven the inductive step, and demonstrated that thr rule holds for a Nil list, we have thus proven the associativity of concatenation.

5.7.2 further equational proofs

consider the reverse function of a list

```
Nil.reverse = Nil
(x::xs).reverse = xs.reverse ++ List(x)
```

we would like to prove that `xs.reverse.reverse =xs`*. The Nil case is easy.*

```
(x::xs).reverse.reverse = x::xs
(xs.reverse ++ List(x)).reverse = (x::xs).reverse.reverse
```

For the inductive step, first we generalise. We aim to show `(ys ++ list(x)).reverse =x::(ys.reverse)` *Nil case*

```
(Nil ++ list(x)).reverse = List(x).reverse
= (x::Nil).reverse
= Nil.reverse ++ List(x)
= Nil ++ (x::Nil)
= x::Nil
= x::(Nil.reverse)
```

inductive step - assume `(ys ++ list(x)).reverse =x::(ys.reverse)`*, show* `((y::ys)++ list(x)).reverse =x::((y::ys).reverse)`

```
((y::ys) ++ List(x)).reverse
(y::(ys ++ List(x))).reverse
(ys ++ List(x)).reverse ++ List(y) //2nd clause reverse
(x::ys.reverse) ++ List(y) // induction
x:: (ys.reverse ++ List(y))
x:: ( y::ys).reverse
```

excercise - show that `(xs ++ ys)map f =(xs map f)++ (ys map f)` *requires clauses of ++ as well as those of map:*

```
Nil map f = Nil
(x::xs) map f = f(x) :: (xs map f)
```

5.8 Other Collections

5.8.1 Vectors

Lists are structured recursively as `x::xs`*, which means that it is faster to access elements near the beginning. Vectors use a tree like list structure - If there are more than 32 elements, then the tree is two levels deep, and can hold up to* $32^2 = 1024$ *elements. Access patterns across Vectors are much more balanced.*

Vectors and lists are very similar, the constructors are the same, and all the same map, fold, foreach, are supported. Cons is the exception, instead of `::`*, Vectors have the operators* `+:` *for adding an element to the left (prepending) and* `:+` *for adding an element to the right (appending).*

Vectors and Lists are subclasses of Sequence. Scala also contains Sets and Maps. Sequences, Sets, and Maps are subclasses of Iterable.

Arrays and Strings support the same operations as sequences, and can be implicitly converted to sequence (but not subclasses).

ranges can be used to construct sequences of integers `val r: Range =1 to 10 by 3`

5.9 for expressions

in imperative programming, nested loops are used frequently. In functional programming, we recursively build a data structure. Take a range, map, flatmap etc., and then filter.

For expresssions produce no side effects, in contrast to for loops in imperative programming.

For epressions take the form `for (s)yield e` *, where s is* sequence of generatprs and filters and e is an expression that returns a value for each iteration.

a generator is of the form `p <- e`, where p is a pattern and e an expression that evaluates to a collection. a filter is of the form `if f`, where f is a boolean expression. if there are several generators in a expression, the last vary faster than the first.

5.9.1 combinatorial search

Sets are like sets in python. Sets do not contain duplicate elements, the elements are unordered, and `set.contains(x)` is a fundamental operation (isin).

Part II

Program Design

Chapter 6

more with for

6.1 queries with for

For expresions in scala can be likened to queries in an RDBMS.

```
for (b <-books, a<- b.authors if a.startsWith "Bird," ) yield b.title

for {
   b1 <- books
   b2 <- books
   if b1.title < b2.title
   a1 <- b1.authors
   a2 <- b2.authors
   if a1 == a2
   } yield a1
```

If the author has written 3 books, they will be printed 3 times.

6.2 translating for-expressions

For expressions are pretty handy. For expressions can generally be translated into expressions based on flatmap, map, and filter. Conversely, all of these functions can be defined in terms of for -

```
def mapFun[T,U](xs: List[T], f: T => U) : List[U] = for (x <- xs) yield
    f(x)

def flatMapFun[T,U]( xs:List[T], f: T => List[U]): List[U] = for (x <-
    xs, y <- f(x)) yield y

def filterFun[T](xs: List[T], f: T => Boolean): List[T] = for (x <-xs if
    f(x)) yield x
```

Scala translates for expressions into expressions based on map, flatmap and filter.

A really simple for-expression

```
for (x <-e1) yield e2
```

can be translated to

```
e1.map( x => e2)
```

Expressions of the form `for (x <-e1 if f; s)yield e2`, where f is a filter and s is a (potentially empty) arbitrary sequence of generators and filters can be translated to

```
for (x<- e1.withFilter(x => f) ; s ) yield e2
```

withfilter is a lazy (i.e. smarter) implementation of filter. It does not create a new (intermediate) collection. The above expression is still contains a for expression, but we have removed one element (the if).

Cases containing more than one leading generator can be translated using flatmap

```
for (x <-e1 ; y<-e2 ; s) yield e3
```

can be translated to

```
e1.flatMap(x => for (y <-e2 ; s) yield e3
```

In all of these cases, we are removing one element from the for expression. Thus an arbitrary expression can be reduced to a sequence of maps and flatmaps.

```
for {
   i <- 1 to N
   j <- 1 to i
   if isPrime(i + j)
} yield (i, j)
```

can be rewritten as

```
//(1 until N).flatMap(i => for ( y <- 1 until i if isPrime(i,j)) yield
     (i,j) )

(1 until N).flatMap( i =>
   (1 until i).withFilter(j => isPrime(i + j)
   .map(j => (i,j)))
```

The for query above on books can be translated to `books.flatmap(b =>b.authors.withFilter(a =>a.startswith("Bird").map(y =>y.title)))`

Note that for expressions are not limited to lists//sequences/iterables. The translation only depends on the prescence of the methods map, flatmap, and withFilter. User defined types can be used in for expressions, provided these three methods are implemented.

For example, the collection books might instead be an interface to a database. Provided the methods are are implemented, for expressions can be used to query. the Scala database connection frameworks ScalaQuery and Slick make use of this.

6.3 Monads

monads must have an associated unit function, and have a flatmap method. I don't really get monads right now.

6.4 Structural Induction

Structural induction can be applied to trees. To prove a property $P(t)$ for all trees of a certain type t

- show that $P(l)$ holds for all leaves l of a tree
- For each type of internal node t with subtrees $s_1, s_2 \ldots s_n$, show that $P(s_1) \wedge P(s_2) \wedge \ldots \wedge P(s_n)$ implies $P(t)$

if the property holds on all of the tree's subtrees, then it holds on the tree
consider the implementation of IntSets

```
abstract class IntSet {
  def incl(x: Int) : IntSet
  def contains(x: Int): Boolean
}

object Empty extends IntSet {
  def contains(x: Int): Boolean = false
  def incl(x: Int): IntSet = NonEmpty(x, Empty, Empty)
}

case class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends
    IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: Int): IntSet =
    if  (x < elem) NonEmpty(elem, left incl x, right)
    else if (x > elem) NonEmpty(elem, left, right incl x)
    else this
}
```

how do we prove the correctness of this implementation holds? consider the following three laws (for integers x,y and Intset s):

- `Empty contains x =false`
- `(s incl x)contains x =true`
- `(s incl x)contains y =s contains y`

The first law is straightforward, and can be seen to be true (Empty.contains is false).

For the second law (proposition), we can do strutural induction. Consider the base case when *s* is an empty set. The we would like to show that `(Empty incl x)contains x =true`. `Empty incl x` evaluates to `NonEmpty(x,...)`, and contains x will evaluate to true in this case.

For the base case when *s* is NonEmpty, assume it takes the form NonEmpty(z,l,r), where l and r are subtrees. There are two cases to consider, z == x and z != x. If z == x, then NonEmpty(x,...) incl x will return this, which contains x, so we're good. if z != x, then NonEmpty(z) incl x will contain x, so we're good.

6.5 streams

Streams are like lists, but are evaluated lazily. The tail of the stream is not evaluated until it is needed. Streams are constructed from the object Stream.empty and the constructor stream.cons.

For lists, we might have something like

```
def listrange(lo:Int, hi: Int): List[Int] =
   if lo >= hi Nil
   else lo::listrange(lo+1,hi)
```

For streams, we would do

```
def StreamRange(lo:Int, hi: Int): Stream[Int] =
   if lo >= hi Stream.empty
   else Stream.cons(lo,StreamRange(lo+1,hi))
```

The standard shorthand for the cons operator, `::`, will always produce a list. There is an equivalent for streams, the hash operator: `#::`, which can be used in expressions and patterns.

6.6 State

Up untill now we've been doing this purely functionally, as much as possible. Some things will have a state.

Everything witha mutable state will be constructed from variables. variables are declared with `var` instead of `val`. Variables can have their value changed later through assignment.

When assignment is possible, then determining whether or not things are equivalent becomes more difficult. Previously, if things evaluate to the same expression, then they are equal.

```
val x = E; val y = E;
val x = E; val y = x;
```

The two lines above produce the same result. In both cases, x and y evaluate to E

```
val x = new BankAccount; val y = new BankAccount;
val x = new BankAccount; val y = x;
```

In this case, the two lines give different results. In the first, two new bankaccounts are created. In the second, y is copied from x.

How do we define "the same"? Operational equivalence - Execute the definitions of x and y, followed by an arbitrary set of operations involving x and y (S), observing all results. Then, execute the definitions followed by a different sequence of operations, S', in which every occurence of y in S has been replaced by x. If the results are different, then x and y are certainly different. Else, if every possible pair of sequences (S, S') are indistinguishable, then x and y are the same.

Assignment breaks the substitution model that we have been using up until now. In general, if we are not using purely functional code, then the substitution model will not hold.

6.7 Loops

Here's a possible definition of while that can be used to constrruct loops

```
def WHILE(condition: => Boolean)(command :=> unit): unit =
  if condition
  { command
     WHILE(condition)(command)
     }
  else ()
```

6.8 event simulation

digital circuits - states are boolean. Will consider inverters (NOT), AND, and OR gates.

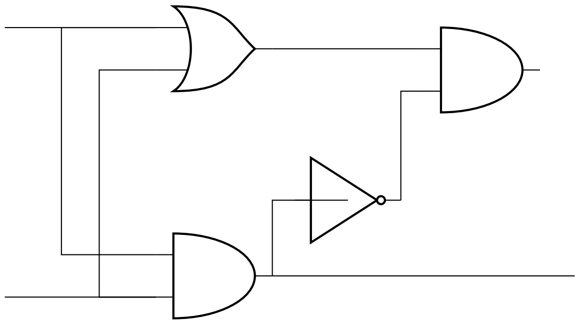

Figure 6.1: half adder

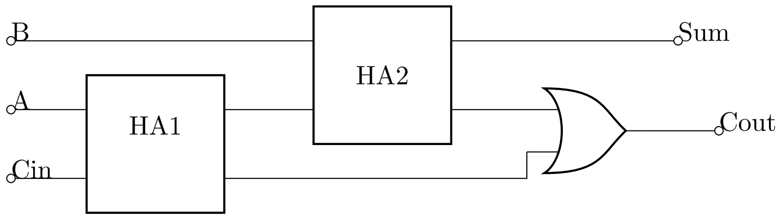

Figure 6.2: full 1-bit adder

Half adder - takes two inputs (A and B), and has two outputs (SUM and CARRY). CARRY is equal to A AND B, while SUM is A OR B AND NOT A AND B

two half adders can be combined (with an or gate) to form a full one-bit adder

6.8.1 event handling

Oberver pattern, also called publish/subscribe or model/view/controller. Views *subscribe* to the model. When the something in the model changes, it *publishes* an update, sending the update to all subscribed views.

```
trait Publisher {
  private var subscribers: Set[Subscriber] = Set()

  def subscribe(subscriber:Subscriber): Unit =
    subscribers += subscriber

  def unsubscribe(subscriber:Subscriber): Unit =
    subscribers -= subscriber

  def publish(): Unit =
    subscribers.foreach(_.handler(this))
}
```

Our BanAccount class could extend from Publisher, and then it would be able to inform other parts of the application when changes to the balance occur

```
class BankAccount extends Publisher {
   private var balance = 0
   def currentBalance:Int = balance

   def deposit(amount:Int):Unit =
      if (amount > 0)
      {
         balance = balance + amount
         publish()
      }

   def withdraw(amount:Int):Unit =
      if (amount > 0 && balance >= amount) {
      balance = balance - amount
      publish()
      } else throw new Error("insufficient funds")
}
\end{lstlisting }

The balance can be queried through currentBalance, and any changes
    (deposits/withdrawals) are published to observers
Speaking of observers

\begin{lstlisting}

class Consolidator(observed: List[BankAccount]) extends Subscriber {
   observed.foreach(_.subscribe(this))

   private var total: Int = _
   compute()

   private def compute() =
      total = observed.map(_.currentBalance).sum

   def handler(pub: Publisher) = compute()

   def totalBalance = total

}
```

The observer pattern (good points)

- is simple to set up
- decouples views from state
- allows multiple views of the same (or similar) states

but (bad points)

- forces imperative style (handlers are unit functions)
- lots of moving parts that need to be coupled
- concurrency makes things complicated

6.8.2 functional reactive programming

reactive programming is reacting to a sequence of events in time.
Functionally, we aggregate an event sequence into a signal.

- The signal is a value that changes over time.
- it is represented as a function that maps from time domain to value.
- instead of propagating updates to a mutable state, we define new signals in terms of existing ones.

There are two fundamental operations over signals - obtaining the value of the signal at the current time, e.g. `mousePosition()` and defining a signal in terms of other signals -

```
def inReactangle(LL: Position, UR: Position): Signal[Boolean] =
Signal {
  val pos = mousePosition()
  LL <= pos && pos <= UR
}
```

Values of Signal are immutable There is a subclass of Signal, `Var`, that has an update operation

```
val sig = Var(3) // signal with constant value of 3
sig.update(5) // sig now has value of 5
```

Updates can be written as assignments (syntactic sugar). Variable signals look a bit like mutable variable, but we can map over them. If we define a signal b to be equal to 2 time a, then updates to a automatically propagate to b. With mutable variables this is not the case, b would need to be redefined after a is changed.
bank account with signals looks much simpler. no need to explicitly publish stuff.

```
class BankAccount {
  val balance = Var(0)

  def deposit(amount:Int):Unit =
    if (amount > 0)
```

```
      {
         balance() = balance() + amount
         // syntatic sugar for balance.update(balance() + amount)
      }

   def withdraw(amount:Int):Unit =
      if (amount > 0 && balance() >= amount) {
      balance() = balance() - amount
      } else throw new Error("insufficient funds")
}
\end{lstlisting }


\section{more functional reactive programming}

\url{https://github.com/rohgar/scala-design-2/wiki/A-Simple-FRP-Implementation}

signals take an expression, and implement an apply method
\begin{lstlisting}
class Signal[T]( expr: => T) {
    def apply(): T = ???
}

object Signal {
    def apply[T] (expr: => T) = new Signal(expr)
}
```

`Signal(expr)`will invoke the object's apply method, returning a new instance of the Signal class, with the supplied expression.

Vars extend signals, such that they can be updated. Vars implement and update method.

```
class Var[T] (expr: => T) extends Signal[T](expr) {
    def update(expr: => T): Unit = ???
}

object Var {
    def apply[T](expr: => T) = new Var(expr)
}

moose = Var(expr)
moose.update(expr2)
```

6.8.3 Syntatic sugar

`moose()= expr2` is equivalent to `moose.update(expr2)`, just as `moose(expr)` is syntatic sugar for `moose.apply(expr)`.

signal maintains:

- itś current value
- the expression that defines the signal value
- a set of observers - other signals that depend on value.

if signal changes, observers are re-evaluated.
How do we record dependencies?

6.9 latency/asynchoronicity

Computations can take time. Scala has a Try monad for exception handling. Try is an abstract class with Success and Failure case subclasses. We can iterate over stuff, and return an instance of success when it works or Failure when it does not. Then we can make use of the successes and handle the failures.

Future is another monad, Instead of handling success/failure it handles latency - the computation may be complete or ongoing (or waiting/whatever)

Really simple network stack:

```
trait Socket {
  def readFromMemory(): Array[Byte]
  def sendToEuropet(packet: Array[Byte]): Array[Byte]
  }

val socket = Socket()
val packet = socket.readFromMemory()
val confirmation = socket.sendToEurope(packet)
```

Reading from memory takes some time, sending to Europe takes a lot of time.
`Future[T]` is a monda that handles both Exceptions and Latency.

```

import scala.concurrent._
import scala.concurrent.Execution.Implicits.global

trait Future[T] {
  def onComplete(callback: Try[T] => Unit)
  //(implicit executor: ExecutionContext): Unit
}
```

When the execution is complete, the callback function is invoked. Note that as the try monad is used, the computiation we are waiting for may have suceeded or failed, but this can be handled.

Javascript uses a shit-tonne of callbacks.

Our reading/sending example above can be modified to

```

val socket = Socket()
val packet: Future[Array[Byte]] = socket.readFromMemory()

val confirmation: Future[Array[Byte]] = packet.onComplete{
   case Sucess(p) => socket.sendToEurope(p)
   case Failure(t) => ...
   }
```

This can result in a lot of nesting. But monads are sweet, because they can be flatmapped.

The implementation of Future might look like

```
trait Future[T] {
   def onComplete( callback: Try[T] => unit) = ...
   def flatMap[S] (f: T => Future[S]) : Future[S] = ???
}
```

How would we implement flatMap using onComplete?

```
trait Future[T] { self =>
   def flatMap( f: T => Future[S]): Future[S] = {
      new Future[S]{
         def onComplete(callback: Try[S] => Unit): Unit = {
            self onComplete {
               case Success(x) => f(x).onComplete(callback)
               case Failure(e) => callback(Failure(e))
            }
         }
      }
   }
}
```

Flatmap creates a new `Future[S]`, and defines it's onComplete method. The new future's oncomplete relies on the onComplete method of the initial Future (self). If the self computation completes sucessfully, then the mapping function f is applied and the onComplete/callback for the Future[S] is invoked. if self's computation fails, then the failure is passed to the callback function directly.

Part III

Parallel Programming

Chapter 7

parallel programming

parallelism and concurrency

parallelism is doing several things at the same time, by efficently using hardware parallelism is mainly concerned with organising the computational prbolem into multiple subproblems that can be executed simultaneously and independently. parallel programming focusses on structuring the algorithms and/or data, such that as many numbers can be crunched as quickly as possible.

Bit-level parallelism is structuring the hardware/software so that a larger number of bits are involved in a single operation. Think of addition - one bit addition would add the ones bits, then carry over (if needed) and add the twos bits, then carry over and do the fours bits, and so on. Alternatively, a four bit algorithm/hardware could add all four of these bits simultaneously in a single operation.

Instruction level parallelism is executing different instructions from the same instruction stream in parallel

concurrency is where multiple instructions may or may not execute at the same time. Main focus of conncurrency is figuring out how to structure a program such that some parts can continue to execute while other parts wait for things (fetch data from memory/disk/database/web). Concurrent programming is targeted towards writing asynchrounous applications.

There is some overlap between the two, but neither is a superset of the other.

$c = a2 + a2$ $b = a1 + a2$ $d = b + c$

b and c can be computed in parallel. This is instruction level parallelism

task level parallism - carrying out operations from two (or more) seperate instruction streams at the same time. THis is what we deal with mostly.

7.1 paralleism in the jvm

will assume we are running on a multicore/multiprocessor system

- OS - software that manages hardware and software resources, and schedules program execution

- process - an instance of a program
  - The same program can be started more than once, and thus may be running in multiple processes
- threads

There are usually many processes running on a small number of cpus. The OS schedules execution - each process will run for a small chunk of time (*time slice*), then the os switches execution to another. This is multitasking.

two different processes cannot directly access each other's memory - they are isolated.

A single program (process) may make use of multiple threads (concurrency units). All threads within a program have access to the program's memory space.

Each thread has a program counter and a program stack. A given thread may not access the stack (or stack variables) of another thread. For threads to communicate, they must read/write to shared (heap) memory.

7.2 threading in the jvm

- define a subclass of `Thread`
- instantiate an object of this subclass
- invoke the `start` method of the subclass

The definition of the subclass defines the code that will be used.

An operation is *atomic* if it occurs instantaenously from the perspective of other threads. The `synchronized` block can be invoked to ensure atomicity. No more than one thread may execute the same synchronized block at the same time.

synchronized blocks can be nested. For example, when transferring from a source to a destination, you might want to invoke synchoronised on both objects, such that the transfer is atomic.

7.3 deadlock

Deadlock occurs when two (or more) threads wait to acquire resources without releasing resources that they hold. None of the threads can proceed, as all are waiting for another thread to finish. This can happen when `synchronized` blocks are nested.

Deadlocks can be avoided by always acquiring resources in a specific order. In the transfer example, each account is associated with a unique id. For a given transfer, the lowest uid is acquired first, followed by the other. This avoids circular dependencies, as things hold the lowest and wait for the highest.

Assuming self-transfers are guarded against, then one of the threads involving the highest account will be able to complete, freeing the lower, allowing other threads to proceed.

7.4 memory model

the memory model is a set of rules used in the JVM that describes how threads will interact when accessing shared memory.

- two threads writing to seperate locations do not require synchronisation
- A thread X that joins on thread Y is guaranteed to observe all writes performed by Y after join returns.

Threads and Synchronization are fairly low level functionalities. In practice we will probably work with higher level abstractions.

7.5 running computations in parallel

Consider the case where we want to compute the pnorm for an array

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p} \tag{7.1}$$

Simplest case is to just sequentially loop over all the elements. We can use two threads. Define $m = n/2$, then one thread computes the sum for $1 \leq i < m$, the other for $m \leq i \leq n$. Then we add the results and raise to the power $1/p$.

Following will compute the sum of the raised elements of an array (worksheet pnorm) **Note** - the ending index on slice is exclusive.

```
def sumSegment(a: Array[Int], p:Double, s: Int, t:Int): Int =
  {
    def raise(aa:Int): Int = floor(pow(aa.abs,p)).toInt

    a.slice(s,t).map(raise).sum
  }

val myArray = Array(1,2,3,4,5,6)

sumSegment(myArray,2,0,3)
```

Four four threads, we could similarly divide the initial array into four pieces. What if we have an unbounded number of threads? If there is no predefined limit on the number of threads we may use, then we can break this into as many pieces as is suitable. There is some overhead associated with setting up threads, so we define some threshold. For a (sub) array with size less than the threshold, we just operate on it sequentially, else we break our array into two pieces and parallelise across these.

```
def pNormRec(a: Array[Int],p:Double): Int =
    power(segmentRec(a,p,0,a.length),1/p)

def segmentRec(a: Array[Int],p: Double,s: Int,t: Int) = {
    if (t-s < threshold)
        sumSegment(a,p,s,t)
    else
    {
    val m = s + (t - s)/2 // (t +s) /2 ?
    val sum1, sum2 = parallel(
        segmentRec(a,p,s,m),
        segmentRec(a,p,m,t))
    sum1 +sum2
    }

}
```

Again, note that as the ending index on slice is exclusive, we can specify m as both the end index for the first segment and the start index for the second segment.

7.6 first class tasks

Previously considered a construct, parallel, that would evaluate expressions in parallel

```
val (v1,v2) = parallel(e1,e2)
```

Alternatively, we could use a task construct.

```
val t1 = task(e1) // spawns a new thread computing e1
val t2 = task(e2) // spawns a new thread computing e2
val v1 = t1.join // joins t1, obtaining result v1 (waits till t1 thread
    completes)
val v2 = t2.join // joins t2 with result v2
```

When creating the tasks, the current computation (presumably the main thread) continues. The join calls are *blocking* - the main thread above will wait until task t1 concludes. Subsequent calls to t1.join quickly return the same result.

parallel can be expressed in terms of task

```
def parallel[A,B](cA: => A, cB: => B): (A,B) = {
```

```
//val ta = task cA
val tb = task cB
//(ta.join, tb.join)
val aa = cA
(aa,tb.join)

}
```

Note that the "task" construct that is mentioned here is implemented in the assignment. The task and parallel implemtations wrap functionality from `java.util.concurrent` (ForkJoinTask).

7.7 complexity of parallel programs

W(e) - the work of a sequential program (single thread) treat parallel as pair - parallel(e1,e2) -¿ (e1,e2) D(e) - the depth of a parallel program (or span) - the number of steps it would be broken into given unbounded parallelism.

W(parallel(e1,e2)) = W(e1) + W(e2) + c1 D(parallel(e1,e2)) = max(D(e1),D(e2)) + c2

If work is divided into equal parts, for depth we only count one part. for parts where we do not explicitly call parallel, we add the constituents

Suppose our platform has P processors. The fastest we could do all the work is W(e)/P Suppose we had unbounded parallelism (infinite threads). The fastest we could do everything is D(e)

D(e) + W(e)/P is often used as an estimate for running time. Given W and D, we can estimate how performance time will behave for different P. If P is constant but inputs grow, then the running time scales linearly with the size of the inputs. Even with infinite resources, we are still bounded by D(e).

7.7.1 Amahl's law

Suppose a program has two parts. part p_1 takes 40% of the time, while part p_2 takes the reamining 60% of the time, but can be sped up. If we make part 2 faster by P times, the speedup for the entire program is

$$S = \frac{1}{f + \frac{1-f}{p}} \tag{7.2}$$

which tends to $1/f$ as $P \to \infty$.

7.8 parallel sorting

7.8.1 Merge sort

This is interesting, as it's not a pure merge sort. Paralleisation is a function of the depth at which we're operating - if we are up high, then we split the array and create new threads. Once a sufficient depth has been reached, we stop spawning new threads. However, instead of doing a mergesort sequentially, we change the algorithm to a quickSort, which operates sequentially.

Sorting is done in place, but auxillary storage is used for merging. At each depth level, we alternate between merging from xs into ys with merging from ys into xs.

```
def parMergeSort(xs: Array[Int, maxDepth:Int): Unit = {
    // aux storage
    val ys = new Array[Int](xs.length)

    def sort(from: Int, until: Int, depth:Int) = {
        if (depth == maxDepth) {
            quickSort(xs, from, until - from)
        } else {
            val mid = (from + until)/2
            parallel(sort(from,mid,depth+1),sort(mid,until,depth+1))

            val flip = depth %2 == 0
            val src = if (flip) ys else xs
            val dst = if (flip) xs else ys
            merge(src,dst,from,mid,until)
        }
    }
}
```

7.8.2 parrallel operations on collections

operations on collections (such as map, fold, scan) are pretty foundational to functional programming.

scan List(1,3,8).scan(100)((s,x) =¿ s + x) == List(100, 101, 104, 112)

like zip + fold? can think of scan as applying fold to all list prefixes, or storing the intermediate results of fold

List is not an ideal data structure for parallel computations, as we need to search through the list to find the middle. concatenation also takes linear time.

7.8.3 Fold Operations and Associativity

Map applies an operation to each element in the sequence. Fold uses a reduction operator (and an initial value) to combine elements, returning a single result. FoldLeft and FoldRight start at the beginning and end of the sequence, respectively.

The result of a fold operation can depend on the order in which the operation is applied to the collection.

```
List(1,3,8).foldLeft(100)((s,x) => s - x) == (((100 -1) -3) -8) == 88
List(1,3,8).foldRight(100)((s,x) => s - x) == (1- (3 - (8 - 100))) == -94
```

To enable parallel operations on collections, we consider associative operations - the order of the operations does not matter

$$\begin{array}{rcl} f(a, f(b,c)) & = & f(f(a,b),c) \\ 1+(2+3) & = & (1+2)+3 \\ 1-(2-3) & \neq & (1-2)-3 \end{array}$$

Addition (and multiplication) are associative, whereas subtraction and division are not. Multiplication is commutative also, in that $f(a,b) = f(b,a)$, whereas division and subtraction are not. String concatenation is also associative.

The operation $f(a,b)$ can be written in infix form $a \otimes b$. A sequence of these operations can be written in the form of a tree, where the leaves are values and nodes are $\otimes$. If the operation is associative, then two expressions with the same list of operands, but parentheses placed in different locations, will evaluate to the same result.

A reduce operation on any tree with this list of operands will yield the same result.

commutative but not associative

Think of the operation that yields the winning move in a game of rock, paper, scissors, $r \otimes p = p$. As $p \otimes r = p$, this operation commutes. but $(r \otimes s) \otimes p = r \otimes p = p$, whereas $r \otimes (s \otimes p) = r \otimes s = r$, so the operation is not associative.

associative but not commutative

- concatenation of strings, lists
- matrix multiplication
- composition of relations
- composition of functions

Associativity is not always preserved by mapping. When combining and optimising map/reduce operations, we need to be careful that the operations given to reduce remain associative (otherwise the result depends on the order in which computations run, which may be more or less random).

Floating point addition and multiplication may not be associative, due to precision/rounding errors.

7.8.4 parallel scan and prefix sums

Scan is an operation that reduces a collection, but returns the intermediate results of the reduction. Instead of just keeping track of the total (the final result), the cumulative results of the reduction are stored in an array.

```
def scanLeft[A](inp: Array[A], a0: A, f: (A,A) => A, out: Array[A]):
    Unit = {

    out(0) = 0
    var a = a0
    var i = 0
    while (i < inp.length) {
        a = f(a, inp(i))
        i = i+1
        out(i) = a
    }
}
```

Parallelising scanLeft seems like a problem, as ordering is important. We assume that each element of the output requires the preceding element to be computed. This isn't strictly true, We can have pieces run independantly if we give up on using the intermediate results. We call the reduction function f many more times, but the gains from parallelism make up for this.

```
def reduceSeg1[A](inp: Array[A], left: Int, right: Int, a0:A, f: (A,A)
    => A ) :A = {

}
def mapSeg[A,B] (inp: Array[A], left: Int, right: Int, fi: (int,A) => B,
    out: Array[B] ) :Unit = {}


def scanLeft[A](inp: Array[A], a0: A, f: (A,A) => A, out: Array[A]):
    Unit = {
val fi = { (i: Int, v: A) => reduceSeg1(inp1,0,i,a0,f)} // like
    currying? fi is a function
mapSeg(inp,0,inp.length,fi,out)
val last = inp1.length -1
out(last+1) = f(out(last),inp(last))
}
```

This implementation does not keep track of "the preceding element". `fi` is a function that takes an index i and carries out the mapping (of `f` on `inp`) for the first i elements of `inp`. `mapSeq` carries out the mapping of `fi` on `inp`. Lots of computation could be saved by using an intermediate/output array for storage, but there is io associated with this. if the mapping function f is fast, or if things can be implemented tail recursively, then the additional computation (running in parallel) may be faster than the sequential version. In fact, as the parallelised map has $\log(n)$ complexity, the overall complexity of the scan is log|(n).

Can use trees to keep track of what has already been computed, so we are not doing the same thing over and over (and over).

define a couple of tree structures:

```
// data structure for our initial input collection
sealed abstract class Tree[A]
case class Leaf[A] (val a:A) extends Tree[A]
case class Node[A] (l: Tree[A], r: Tree[A]) extends Tree[A]

// result tree, here nodes have values also, not just leafs
sealed abstract class TreeRes[A](val res:A)
case class LeafRes[A] (override val res:A) extends TreeRes[A]
case class NodeRes[A] (l: TreeRes[A], override val res:A, r: TreeRes[A],
    ) extends TreeRes[A]

// reduction that preserves computation tree
// sequential
def reduceRes[A](t :Tree[A], f: (A,A) => A): TreeRes[A] = t match {
    case Leaf(v): LeafRes(v)
    case Node(l,r): {
        val (tL,tR) = (reduceRes(l,f),reduceRes(r,f))
        NodeRes(tL, f(tL.res,tR.res),tR)
        }
    }
```

We can easily parallelise this. The function upsweep carries this out (bottom-up computation of the result tree). For scans, would like to move through the result tree in a particular order. In Downsweep, we move through the result tree, and output a tree containing the scan results.

```
def upsweep(t: Tree[A],f: (A,A) => A) = t match {
    case Leaf(v): LeafRes(v)
    case Node(l,r): {
        val (tL,tR) = parallel(upsweep(l,f),upsweep(r,f))
        NodeRes(tL, f(tL.res,tR.res),tR)
        }
    }
//a0 is the reduce of all elements to the left
def downsweep(t: Tree[A], a0: A, f: (A,A) => A):Tree[A] = t match {
    case LeafRes(a): Leaf(f(a0),a)
    case NodeRes(l,a,r): {
        val (tL,tR) =
            parallel(downsweep[A](l,a0,f),downsweep[A](r,f(a0,l.res),f))
        Node(tL,tR)
        }
    }
```

To do scanleft, we build the result tree (with upsweep), then downsweep.

```
def scanLeft[A](t: Tree[A], f: (A,A) =>A): Tree[A] = {
    val tRes = upsweep(t,f)
    val scanl = downsweep(tRes,a0,f)
    prepend(a0,scanl)
}

def prepend[A](x: A, t: Tree[A]) = t match {
    case Node(l,r): Node(prepend[A](x,l))
    case Leaf(v): Node(Leaf(x), Leaf(v))

}
```

Extend this to work with collections that are initially arrays. Will still use a tree for the intermediate results (because trees are awesome). We don't keep track of array elements, instead we track ranges and pass around a reference to the array.

```
sealed abstract class TreeRes[A](val res:A)
case class LeafRes[A] (from: Int, to: Int, override val res:A) extends
     TreeRes[A]
case class NodeRes[A] (l: TreeRes[A], override val res:A, r: TreeRes[A],
     ) extends TreeRes[A]

def upsweep(inp: Array[A], from: Int, to: Int, f: (A,A) => A):
     TreeRes[A] = {
    if (to - from < threshold)
        LeafRes(from,to,reduceSeg1(inp, from+1, to, inp(from),f))
    else {
        val mid = from + (from - to)/2
        val (tL,tR) =
             parallel(upsweep(inp,from,mid,f),upsweep(inp,mid,to,f))
        NodeRes(tL,f(tL.res,tR.res),tR)}
    }

}
```

Chapter 8

Data Parallelism

Task parallelism is a form of parallisation that distributes execution processes across computing nodes. data parallelism is a form of parallelism that distributes data across copmuting nodes.

```
def initializeArray(xs: Array[Int])(v: Int): Unit = {
    for (i <- (0 until xs.length).par) {
    xs(i) = v
    }
}
```

The par makes use of scala parallel collections.

The amount of processing required by each element of data is the workload. In the case of initializeArray, the workload is uniform, which means that the data-parallel processing works well (all elements take the same time, good speedup, linear performance). For the case of the Mandelbrot set processing, some pixels take more iterations than others (pixels in the set compared to those that are determined to be out of the set almost immediately). This means that the workload is unbalanced/irregular.

8.1 parallel collections

the `.par` call converts a sequential collection into a parallel collection.

```
(1 untill 1000).par
.filter(n => n%3 ==0)
.count(n => n.toString == n.toString.reverse)
```

Not everything can be parallelised. FoldLeft requires that the reduction operation be carried out in a specific order (as the collection's element type and the reduction type may be dfferent). For the fold operation, the result type is the same as the element type, so there are mulitple ways in which the element-wise reductions can be ordered. This can be parallelised, as different processes can reduce different chunks of the data, and then the ”subtotals” may be merged.

8.1.1 fold

For fold to work consistently on a parallel collection, there are a couple of conditions that must be satisfied:

- the folding operation must be associative - `f(a, f(b,c))==f(f(a,b),c)`
- the neutral element must be ignored - `f(a,z) == f(z,a) == a`

We say that the neutral element and the function f form a *monoid*.

The signature of of the function given to fold must be of the form `f: (A,A)=> A` , with the neutral element of type `A`. This is fairly rigid. foldLeft was more flexible, but is not assciative `f: (B,A)=> B`, where A is the type of the collection element and B is the type of the reduction.

Aggregate is like fold, but is more flexible. It takes a neutral element, and two combination operations, one for combinig two subtotals, and one for adding a collection element to a subtotal. collection.par.aggregate

8.2 collections - hierarchy review

- `Traversable[T]` - collection of elements of type T, with operations implemented using foreach
- `Iterable[T]` - collection of elements of type T, with operations implemented using iterator
- `Seq[T]` an ordered sequence of elements of type T
- `Set[T]` a set of elements of type T, no duplicates.
- `Map[K,V]` a mapping from keys of type K to values of type V, no duplicate keys.

`ParIterable, ParSet, ParMap, ParSeq` are the parallel analogs of the above. Generic collections `GenIterable, GenSet, GenMap, GenSeq` are types for code that is agnostic about parallelism.

Generic collections allow us to write code that is unaware of parallelism. **These were apparently deprecated in later versions of scala.**

Need to use parallel collections with care:

- Never read from a parallel collection that is concurrently modified.
- Never write to a parallel collection that is concurrently traversed

TrieMap is an exception, can create a snapshot of the current map state, and then update the map based off the snapshot.

8.3 builders and combiners

transformer operations take collections and return collections (map, flatmap, filter, groupBy) fold, sub, aggregate are not transformer operations

Builder is a trait that is used in sequential collection methods Elements can be added to the builder, and when the result method is called it returns the desired collection.

Parallel collections cannot use builders, but instead can make use of combiners. Multiple combiners can be combined.

When the collection (repr) is a set or map, combine represents the union. when the collection is a sequence, combine represents concatenation.

In order for combiners to be efficient, the combine operation must have complexity $O(\log n + \log m)$, where n and m are the size of the component combiners.

8.3.1 two phase construction

As mentioned above, the combiner must have a a sufficiently efficient implementation for parallelism to be worthwhile. This can be difficult if we assume that the combiner internally stores data using the same structure as it operates on (e.g. a combiner that returns arrays using arrays internally). Two-phase construction is a method in which the internal data structure differs from the resulting data structure. Obviously, the internal data structure must have an efficient combine method ($O(\log n + \log m)$ or better). The conversion from the internal structure to the result structure must be able to take place in $O(n/p)$ time, where n is the number of elements and p is the number of processors (level of parallelism). In other words, the internal structure must be able to be copied in parallel.

In the first phase, different processors build intermediate structures in parallel, invoking the `+=` method. The these component structures are combined in (in parallel) using a reduction tree.

When result is invoked, the final structure is built in parallel.

8.3.2 conc-trees

Lists are sequential containers, `x::xs` . They are unsuited for parallel computation. Trees are better. When we reach a node, we can launch two computations in parallel, one recursing down the left subtree, the other down the right.

Trees are only good for parallism if they are balanced. If we apply a filter operation to a tree structure, we may be left with a tree that is imbalanced. This can defeat the purpose of parallism, as most (all) of the work may be done by one thread.

```
sealed trait Conc[+T] {
   def level: Int
   def size: Int
   def left: Conc[T]
```

```
    def right Conc[T]
}
```

concrete implemetations of conc contain the following

```
class object Empty extends Conc[Nothing] {
    def level = 0
    def size =0
}
class Single[T] (val x: T) extends Conc[T]{
    def size = 1
    def level = 0
}

// node/fork
case class <>[T] (left: Conc[T], right: Conc[T]) extends Conc[T] {
    val level = 1 + math.max(left.level,right.level)
    val size = left.size + right.size
}
```

Tree invariants:

- A <> node can never contain Empty as a subtree
- the level difference between the left and right subtree of a <> node must be 1 or less

These constrain the depth of a tree to be at most log N. Consequently, operations on the tree will follow this bound.

8.3.3 amortized conc-trees

conc trees are awesome. We can use them to implement a combiner. This requires a += method

```
var xs: Conc[T] = Empty
def +=(elem: T){
    xs  = xs <> Single(elem)
}
```

This would work, and would allow insertion in log(N) complexity. But we can do better. If we extend the conc-tree to allow a new append node type.

```
case class Append[T](left: Conc[t], right: Conc[T]) extends Conc[T] {
val level = 1 + math.max(left.level,right.level)
val size = left.size + right.size
}
```

We relax the earlier assumptions, and allow arbitrary height differences between left and right trees. This no longer guarantees that concatenation can be carried out with logarithmic complexity. We could manage this if we could eliminate append nodes in log(n) time, but this is not possible. Instead, we ensure that for a tree of size n, there are never more than log(n) append nodes. This results in O(1) appends and O(log n) concatenation. https://myfavoritemurder.com/