Updated: 2021-02-25

4.1
The Substitution Principle

Northeastern University

Floating Point Problem

```
def evaluate_3_tenths = 1.0/10 + 2.0/10

def multiply_by_10_over_3(x: Double) = x / 3 * 10

"doublePrecision" should "work properly" in {
  val x = FunctionalProgramming.evaluate_3_tenths
  val y = FunctionalProgramming.multiply_by_10_over_3(x)
  y should be (1)
}
```

This test fails. Why? Because, unlike pure algebraic numbers, floating point numbers are ***not associative***.

That's to say that (a+b)+c != a+(b+c), at least in general. Much of the time, it will hold true but not always.

Floating Point Solution 1

ScalaTest:

```
"Floating Point Problem" should "be OK" in {
   val x = Rational(1,10)+Rational.normalize(2,10)
   val y = x * 10 / 3
   y shouldBe 'unity
 }
```

Rational is a class defined in the class git repo. There is also a chapter on a *Rational* class in the text book.

ScalaTest works fine for particular cases. We could do more testing to cover the domain, especially using *scalacheck*. But none of this is as good as…

Floating Point Solution 2

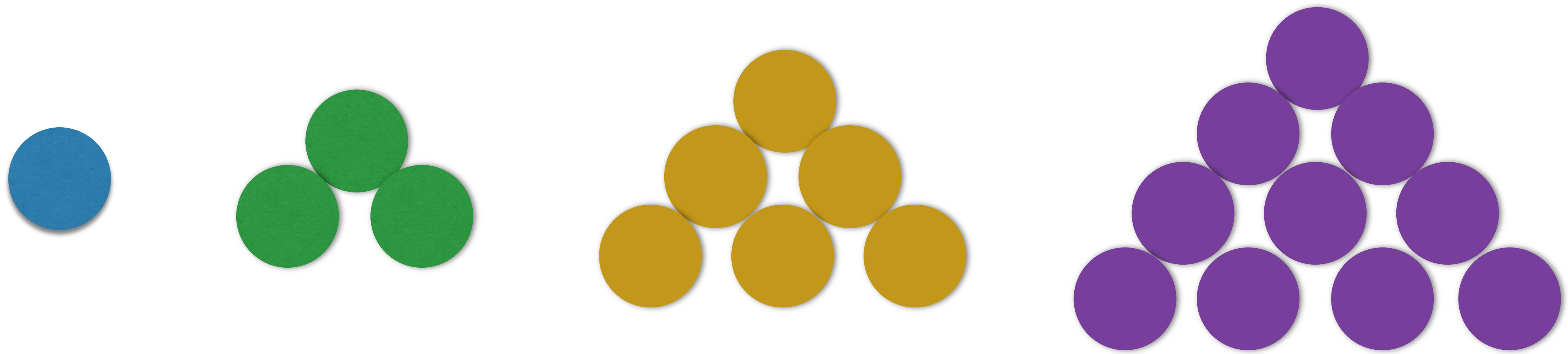
We can do better with substitution*:

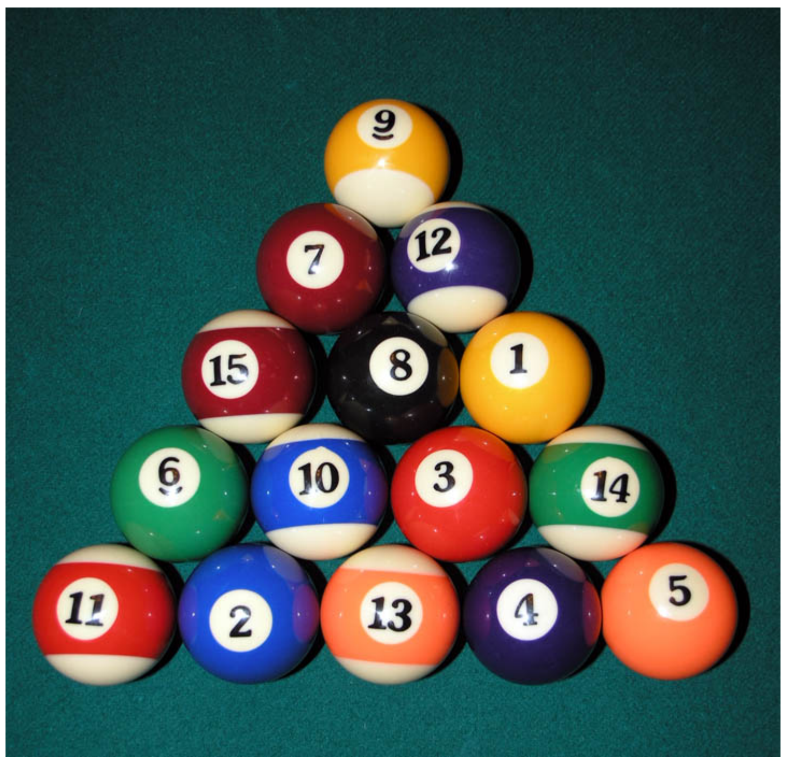
```
 (1) Rational.normalize(2,10) → Rational(2/gcd(2,10),10/gcd(2,10))
 → Rational(2/gcd(2,0),10/gcd(2,0)) → Rational(2/2,10/5) →
Rational(1,5)
 (2) Rational(1,10)+Rational(1,5) →
Rational.normalize(1*5+10*1,10*5) → Rational.normalize(15,50) →
Rational(15/gcd(15,50),50/gcd(15,50)) → Rational(3,10)
 (3) Rational(3,10) * 10 → Rational(3,1)
 (4) Rational(3,1) / 3 → Rational(1,1)
 (5) Rational(1,1).isUnity → 1==1 && isWhole → 1==1 && 1==1. → true
```

* The "Substitution Model" is an important aspect of functional programming which we will refer to again and again.

The Substitution Model

- Formalized in the **λ** calculus, introduced by Church (1932)
- Can be used to **prove** the equivalence of expressions *provided* all expressions reduce to “pure” functions (or constants), i.e. **there are no side-effects!**
- Termination: not all expressions can be substituted in a finite number of steps: e.g. `def x = x`
- In which order should we evaluate expressions?
  - `val x = 3*4*5`: two options: (3*4)*5 and 3*(4*5)
  - `val x = square(3+4)`: two options: (3+4)*(3+4) and 7*7
    - These are known as call-by-name and call-by-value
    - Which do you think is fastest? Best?





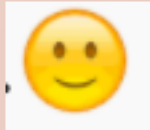
https://commons.wikimedia.org/wiki/File:Eight_Ball_Rack_2005_SeanMcClean.jpg

9	5	4	13	2	11
7	12	14	3	10	6
15	8	1	1	8	15
6	10	3	14	12	7
11	2	13	4	5	9

N(N+1)

Sum of 1 thru N

Side-bar on
"Proof by Induction"

- Consider the "discrete sum" of consecutive whole numbers:
  - $P(n) = 0 + 1 + 2 + \ldots + n$
  - I will assert (see previous slide) that $P(n) = n(n+1)/2$ for any positive integer *n.*
    - This seems to work for n = 4: $0+1+2+3+4 = 10 = 4(5)/2$
  - But can we ***prove*** it for all *n*?
- Proof by induction involves proving, independently, two cases: the base case and the inductive step:
  - Base case: *n = 1*
    - $0 + 1 = 1$ and $1(1+1)/2 = 1$
  - Inductive step:
    - *if* $P(n) = n(n+1)/2$ *then* $P(n+1)$ *should equal* $(n+1)(n+2)/2$
    - i.e. $(n+1)(n+2)/2 - n(n+1)/2$ should equal $n+1$
    - i.e. $(n+2-n)(n+1)/2$ should equal $n+1$
    - i.e. $2(n+1)/2$ should equal $n+1$
- Now, we have proven that the identity holds for *n=1* and that, if it holds for *n*, it also holds for *n+1*
- Therefore: $P(n) = n(n+1)/2$ 

Substitution proof

- Let us prove a definition of ***List.length***:

```
package edu.neu.coe.csye7200.list

trait List[+A] {
  def length: Int
 }
case object Nil extends List[Nothing] {
  def length: Int = 0
}
case class Cons[+A](head: A, tail: List[A]) extends List[A] {
 def length: Int = 1 + tail.length
}
object List {
 def apply[A](as: A*): List[A] =
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

- By *induction*: two parts:
  - Prove the case for Nil (0)
  - Prove that: *if* it holds for list of length *N*, *then* it holds for list of length *N+1*

Do this proof by induction. You may confer.

Substitution proof (2)

- (1):

  `Nil.length` → `0`

- (2):
  - Assume that *listN.length* yields the length of *ListN*, i.e. N

  `Cons("a",listN).length` → `1 + listN.length` → `1 + N`

- Statement is proved because of (1) and (2)

Call-by-name/value

- Remember from a few slides ago?
  - In which order should we evaluate expressions?
    - `val x = 3*4*5`: two options: (3*4)*5 and 3*(4*5)
    - `val x = square(3+4):` two options: (3+4)*(3+4) and 7*7
      - These are known as call-by-name and call-by-value
      - Which do you think is fastest? Best?
- Did you decide what was best?

Call-by-name/value

- If both CBN and CBV terminate, then the result is equivalent…
  - So, does it matter which we use?
  - It can matter a lot!!
  - if CBV terminates, then so must CBN (but not other way around)
  - def zip[A,B](x: Seq[A], y: Seq[B]): Seq[(A,B)]

“Seq” is a Trait extended by List, LazyList, etc.

```
Welcome to Scala version 2.11.6 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_05).
Type in expressions to have them evaluated.
Type :help for more information.

scala> val x = List("a","b")
x: List[String] = List(a, b)

scala> val y = LazyList from 1
Y: scala.collection.immutable.LazyList[Int] = LazyList(1, ?)

scala> x.zip(y)
res6: List[(String, Int)] = List((a,1), (b,2))
```

Strict/non-strict functions

- First off—a question:
  - You've been tasked with implementing a method which takes two boolean arguments and yields the "and" of the two values.
  - However, with a mind to efficiency, your "customer" says that he doesn't want to evaluate the second argument if the first one is false.
  - Your first idea is this:

```
def and(a: Boolean, b: Boolean): Boolean =
  if (a) b else false
```

    - How can you change the definition to achieve your goal?

Non-strict (lazy) functions

```
scala> def and(a: Boolean, b: Boolean): Boolean = if (a) b else false
and: (a: Boolean, b: Boolean)Boolean

scala> def myTrue = {println("hello"); true}
myTrue: Boolean

scala> and(false,myTrue)
hello
res9: Boolean = false
```

However, we did have to explicitly invoke *b* to yield a *Boolean* from a *=>Boolean*.

Note that when we define *and* this way, we have to “partially apply” the value of *myTrue*.

```
scala> def and(a: Boolean, b: () => Boolean): Boolean = if (a) b() else false
and: (a: Boolean, b: => Boolean)Boolean

scala> and(false, myTrue _)
res10: Boolean = false
```

Syntactic sugar for the form involving *Function0[Boolean]*

- Note that we replaced the “normal” value-type Boolean parameter “b” with a function that takes no arguments but returns a Boolean.
- We could also write the “and” method and its invocation thus:

```
def and(a: Boolean, b: => Boolean): Boolean = if (a) b else false
and(false, myTrue)
```

Type inference with isomorphism

- Let's say we have a *List* of *Int*s and we convert it to a list of *String*s:
  - `val x = List(1,2,3)`
  - `val y = x map {n => n.toString}`
  - The compiler knows that *y* is a *List[String]* because of ***Type Inference**** so we don't have to explicitly write *val y: List[String]*=…
- Let's say we need a greatest common denominator method:
  - `@tailrec private def gcd(a: Long, b: Long) = if (b==0) a else gcd(b, a % b)`
  - The compiler cannot figure out what the return type of *gcd* is supposed to be
  - `@tailrec private def gcd(a: Long, b: Long): Long = if (b==0) a else gcd(b, a % b)`
- More on this later…

However, explicitly specifying types can be a very helpful aid to getting programs to compile

We must explicitly specify the return type of a recursive method

* and because of type isomorphism, *map* retains the "shape" of *x* in *y*

Type inference/isomorphism continued

- We just saw this (where annotating the type of ***y*** is optional):
  - `val y: List[String] = List(1,2,3) map (_.toString)`
- But we could also have written any of the following:
  - `val y: Seq[String] = Seq(1,2,3) map (_.toString)`
  - `val y: LazyList[String] = LazyList.from(1) map (_.toString)`
  - `val y: Option[String] = Option(1) map (_.toString)`
- But there is really only one definition of ***map***: it's in *IterableOnce*…
- How does it work? It's a little bit of magic and some ordinary code. You can look at the source code yourself. Rather advanced for now, though.