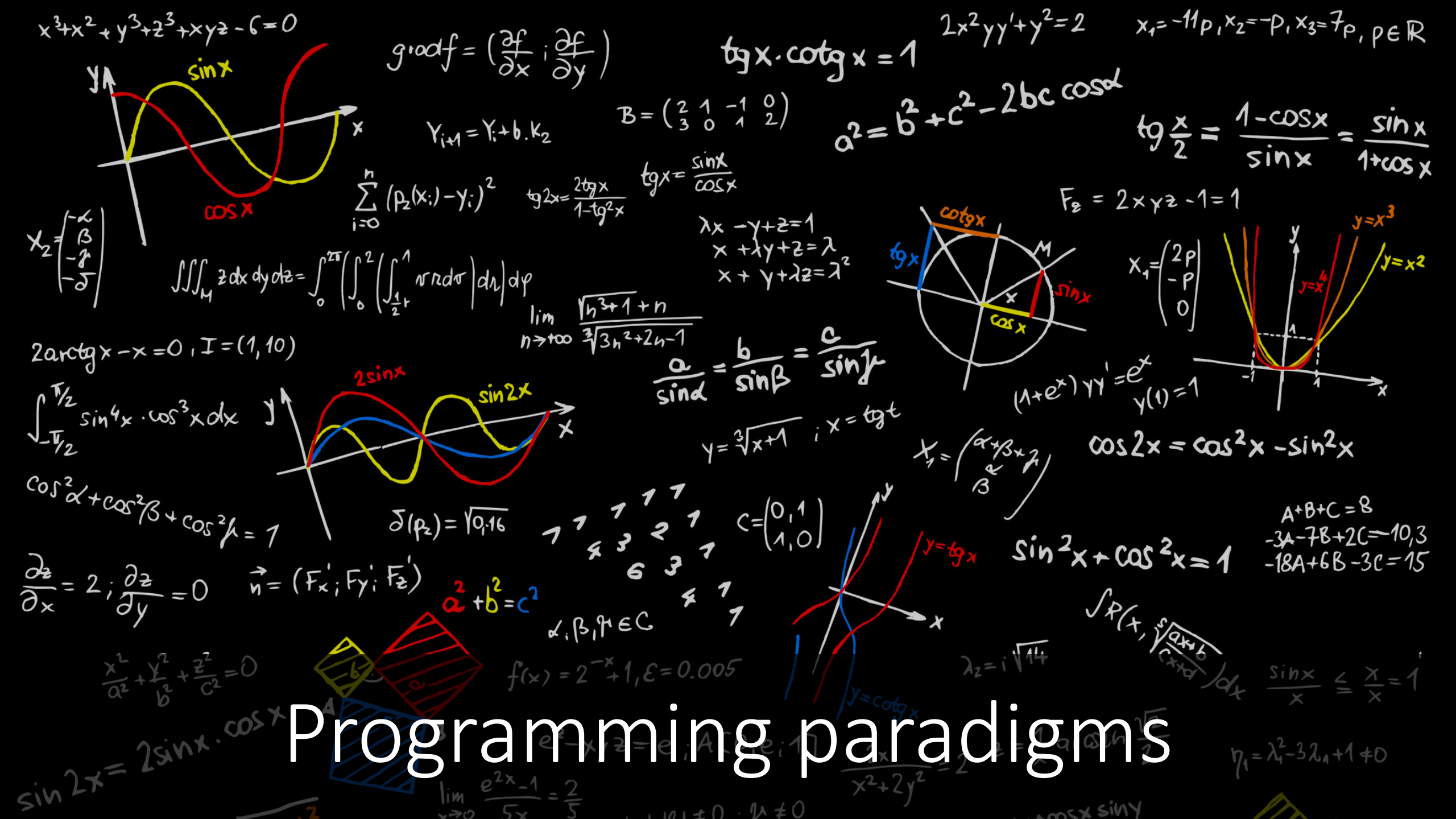


Programming paradigms

```
$psd = md5($_POST('pwd'])
$admin = 0

If($pwd == "0e123456789123456789123456789012")
        $admin =1
}
```

STRING VS STRING

“1.00000000000000001” == “0.1e1”

“+1” == “0.1e1”

“1e0” == “0.1e1”

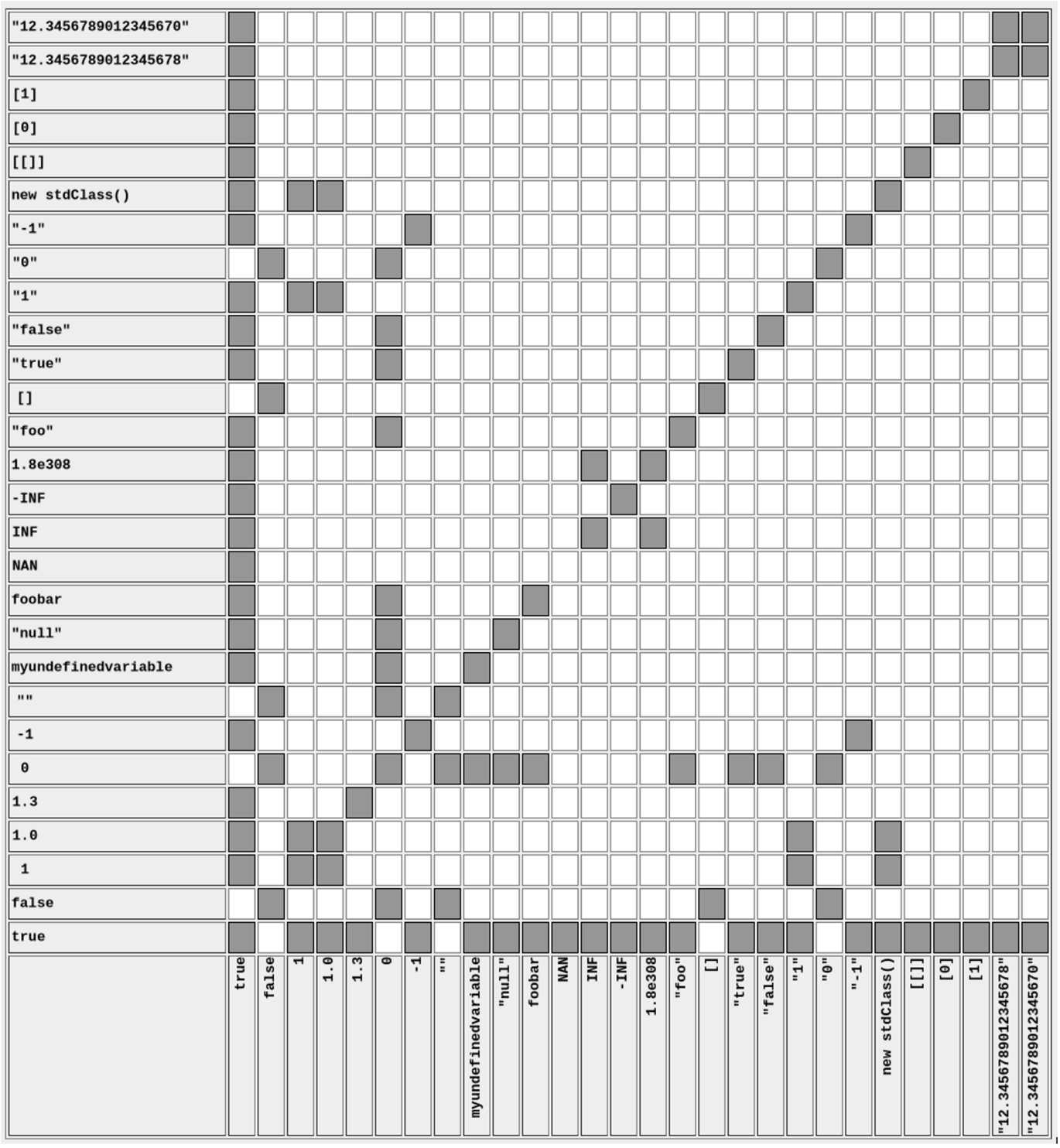
“-0e10” == “0”

“1000” == “0x3e8”

“1234” == “ \t\r\n 1234”

“PHP”== 0


I DON'T HAVE ANY IDEA
WHAT'S GOING ON HERE


"12.3456789012345670"
"12.3456789012345678"
[1]
[0]
[[]]
new stdClass()
"-1"
"0"
"1"
"false"
"true"
[]
"foo"
1.8e308
-INF
INF
NAN
foobar
"null"
myundefinedvariable
""
-1
0
1.3
1.0
1
false
true
true
false
1
1.0
1.3
0
-1
""
myundefinedvariable
"null"
foobar
NAN
INF
-INF
1.8e308
"foo"
[]
"true"
"false"
"1"
"0"
"-1"
new stdClass()
[[]]
[0]
[1]
"12.3456789012345678"
"12.3456789012345670"

240610708
QNKCDZO
aabg7XSs

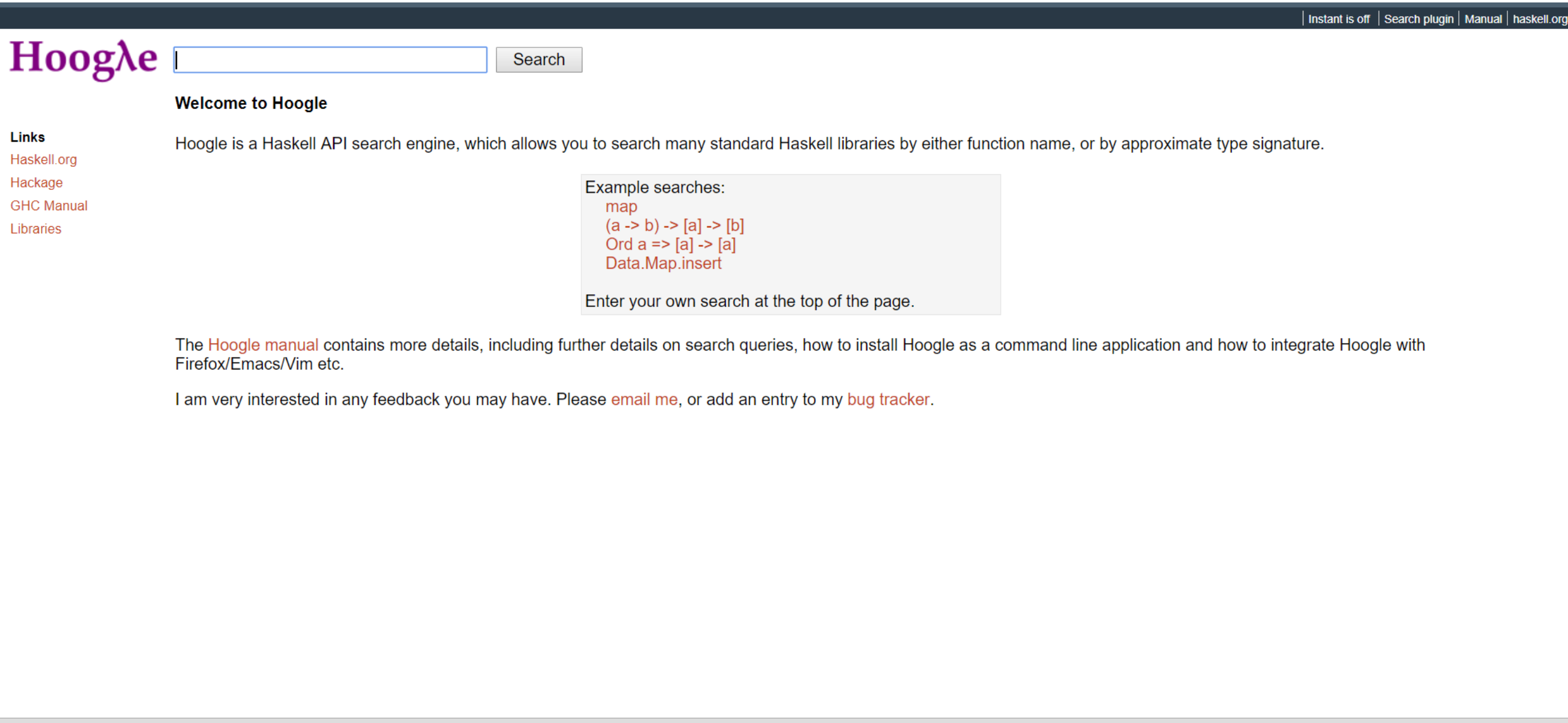
```
$psd = md5($_POST(‘pwd’])
$admin = 0

If($pwd == “0e123456789123456789123456789012”)
        $admin =1
}
```

What we do not have in functional programming

- Nulls
- Mutable varables
- Void
- Objects
- Exceptions
- Loops
- Side effects

Hoogle


Instant is off | Search plugin | Manual | haskell.org
Hoogle
Search
Welcome to Hoogle
Links
Haskell.org
Hackage
GHC Manual
Libraries
Hoogle is a Haskell API search engine, which allows you to search many standard Haskell libraries by either function name, or by approximate type signature.
Example searches:
map
(a -> b) -> [a] -> [b]
Ord a => [a] -> [a]
Data.Map.insert
Enter your own search at the top of the page.
The Hoogle manual contains more details, including further details on search queries, how to install Hoogle as a command line application and how to integrate Hoogle with Firefox/Emacs/Vim etc.
I am very interested in any feedback you may have. Please email me, or add an entry to my bug tracker.


Null

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference,

simply because it was so easy to implement.

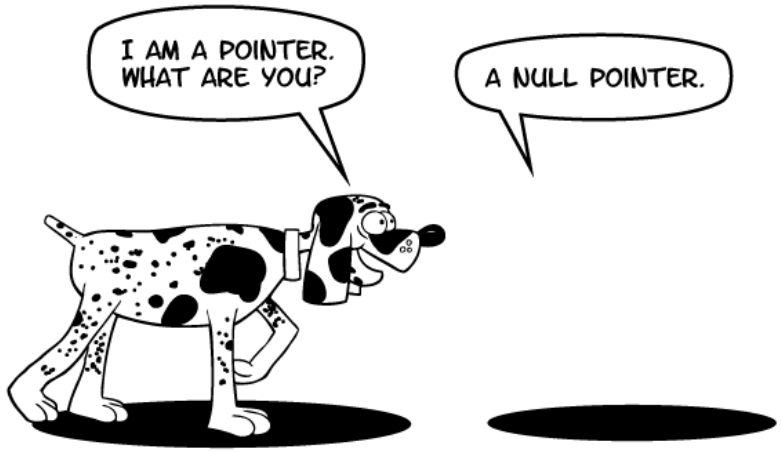
This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Null Safety

(Nullable types and Non-Null Types)

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok
```



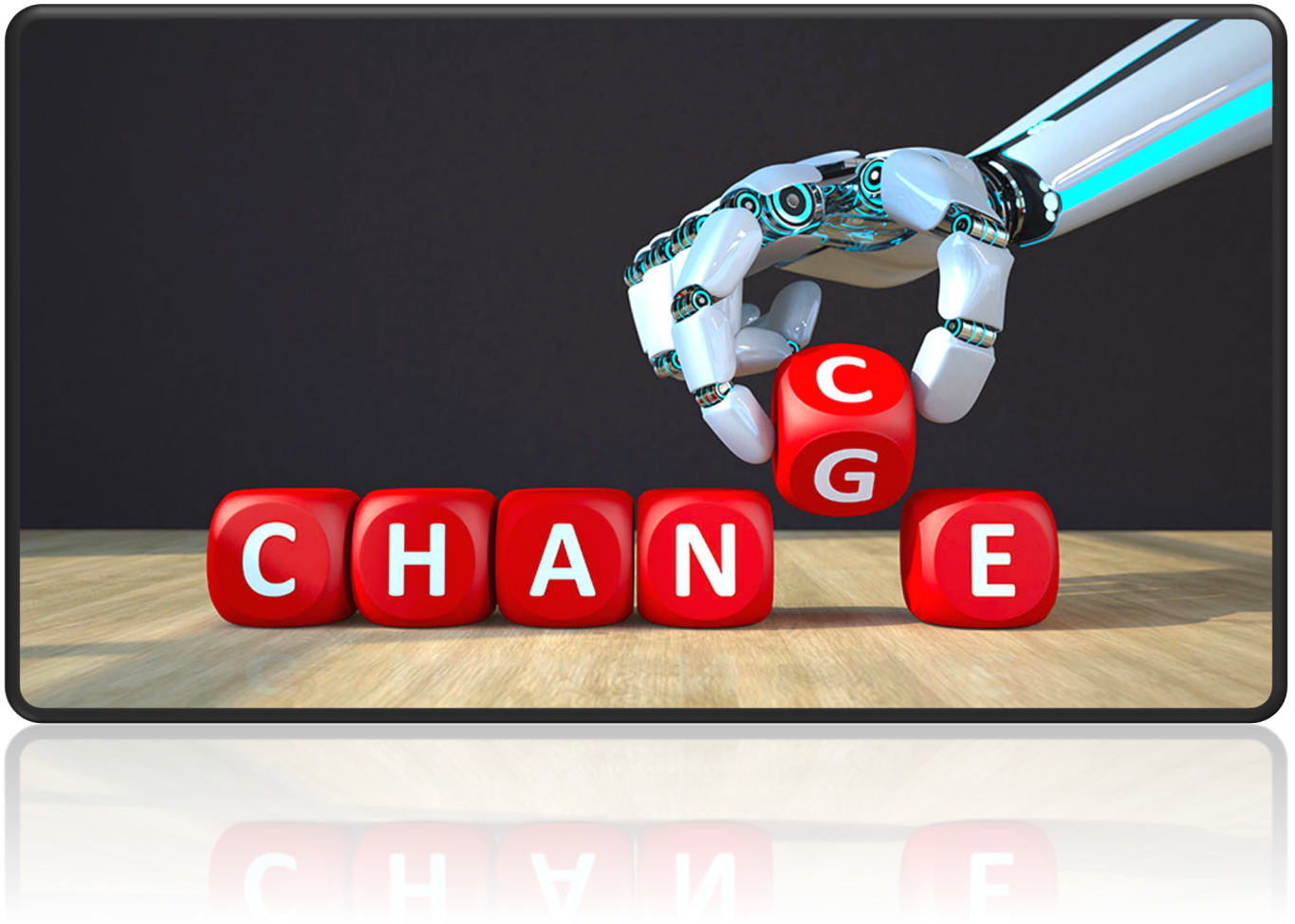
Varables

val / var

Types… why we need them?

- Communication
- High level reasoning tool
- Safety feature
- Laying out bytes in memory

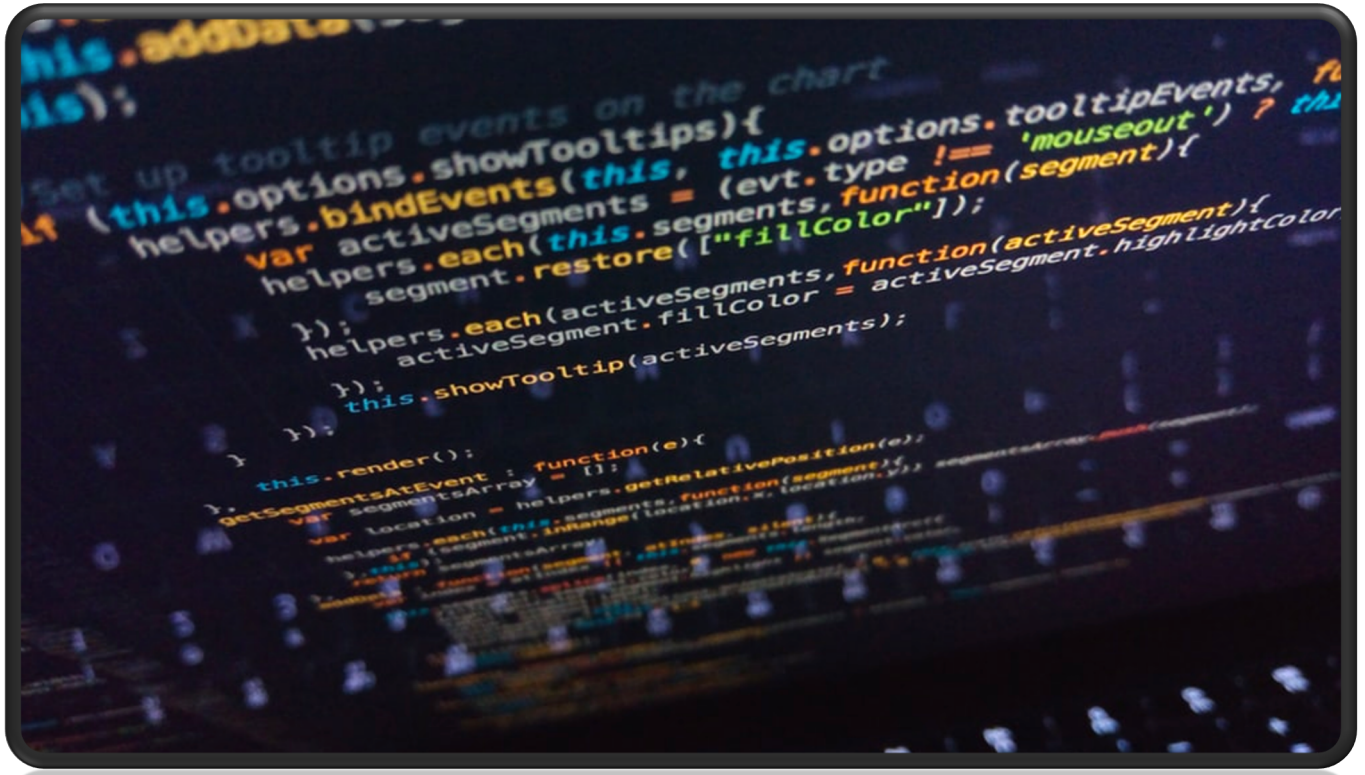
Immutability



Exception

Recursion vs Loop

- Natural way to solve problems in math


(this.options.showTooltips){
helpers.bindEvents(this, this.options.tooltipEvents,
var activeSegments = (evt.type !== 'mouseout') ?
helpers.each(this.segments, function(segment){
segment.restore(["fillColor"]);
helpers.each(activeSegments, function(activeSegment){
activeSegment.fillColor = activeSegment.highlightColor
this.showTooltip(activeSegments);
this.render();

Time to code

Higher Order Function

In mathematics and computer science, a higher-order function is a function that does at least one of the following:

- takes one or more functions as arguments,
- returns a function as its result.

$$h(x) = f(g(x))$$

Why?

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

```
def cube(x: Int): Int = x * x * x

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

HOF allow us to do it like that:

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

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

```
def urlBuilder(ssl: Boolean, domainName: String): (String, String) =>
String = {
  val schema = if (ssl) "https://" else "http://"
  (endpoint: String, query: String) =>
s"$schema$domainName/$endpoint?$query"
}

val domainName = "www.example.com"
def getURL = urlBuilder(ssl=true, domainName)
val endpoint = "users"
val query = "id=1"
val url = getURL(endpoint, query)
```

Notice the return type of urlBuilder (String, String) => String. This means that the returned anonymous function takes two Strings and returns a String. In this case, the returned anonymous function is (endpoint: String, query: String) => s"https://www.example.com/$endpoint?$query".

Function types

- The type A => B is the type of a *function* that takes an argument of type A and returns a result of type B.
- So, Int => Int is the type of functions that map integers to integers.
- Similarly, (A1, A2) => B is the type of functions that take two arguments (of types A1 and A2, respectively) and return a result of type B.
- More generally, (A1, ..., An) => B is the type of functions that take n arguments (of types A1 to An) and return a result of type B.

Operator

Operator precedence (in descending order)

Operator precedence in Scala is based on the first character of the method name used in operator notation (except for assignment operators):

(all other special characters)

* / %

+ -

:

=!

< >

&

^

(all letters)

(all assignment operators)

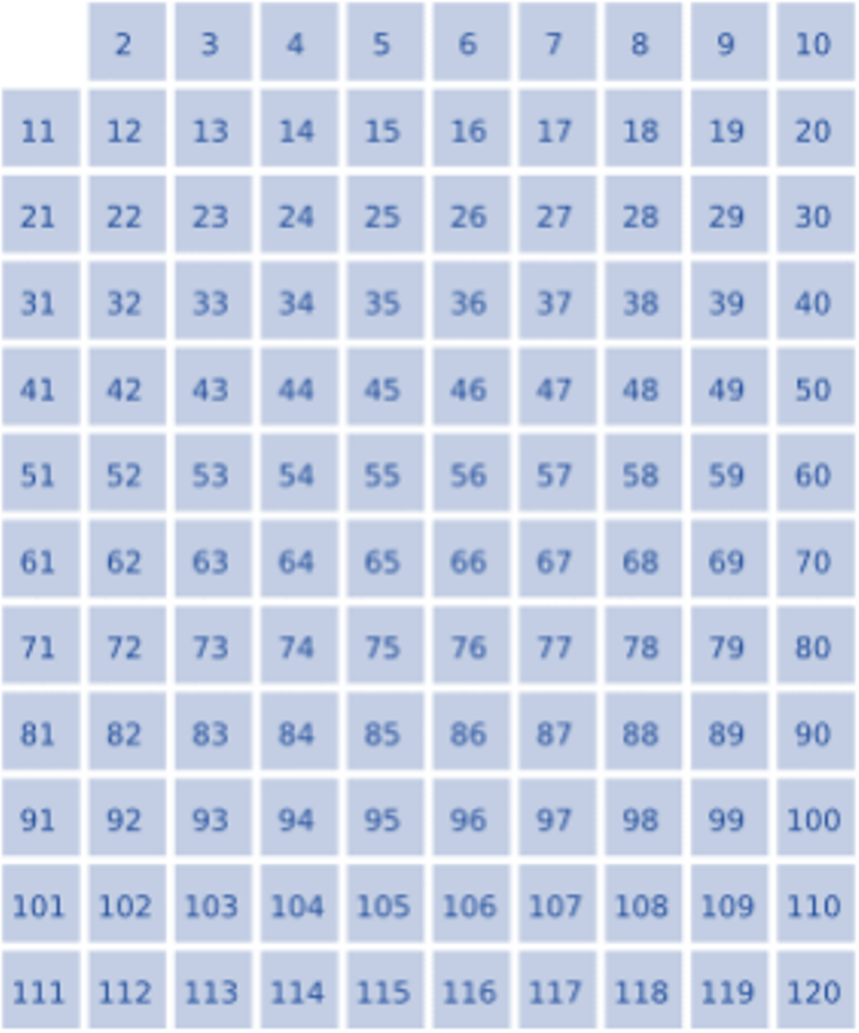
Operator connectivity

Operator connectivity in Scala bases on the last character of the method name. If the method name ends with a colon ':', then we have right-hand communication; otherwise - left-sided.

Sieve of Eratosthenes

To find all the prime numbers less than or equal to a given integer n by Eratosthenes' method:

1. Create a list of consecutive integers from 2 through n: (2, 3, 4, ..., n).
2. Initially, let p equal 2, the smallest prime number.
3. Enumerate the multiples of p by counting in increments of p from 2p to n, and mark them in the list (these will be 2p, 3p, 4p, ...; the p itself should not be marked).
4. Find the first number greater than p in the list that is not marked. If there was no such number, stop. Otherwise, let p now equal this new number (which is the next prime), and repeat from step 3.
5. When the algorithm terminates, the numbers remaining not marked in the list are all the primes below n.



Sieve of Eratosthenes

```
val primes = (toN:Int) => {
  def findPrimes(sieve:List[Int]):List[Int] = sieve match {
    case h::t => h::findPrimes(t filter (x=>x%h != 0))
    case Nil => Nil
  }
  findPrimes(List.range(2,toN))
}

primes(30)
```

Transformer methods (List)

```
List(1,2,3,4) map (x=>x*x)
List(1,2,3,4) filter (x=>x%2 == 0)
```

```
List(List(5,6), List(1,2,3)) flatten
List(List(5,6),List(1,2,3)) flatMap (x=> x tail)
```

map, filter, and reduce explained with emoji 😂

```
map([🐮, 🍠, 🐔, 🌽], cook)
=> [🍔, 🍟, 🍗, 🍿]

filter([🍔, 🍟, 🍗, 🍿], isVegetarian)
=> [🍟, 🍿]

reduce([🍔, 🍟, 🍗, 🍿], eat)
=> 💩
```

Transformer methods (List)

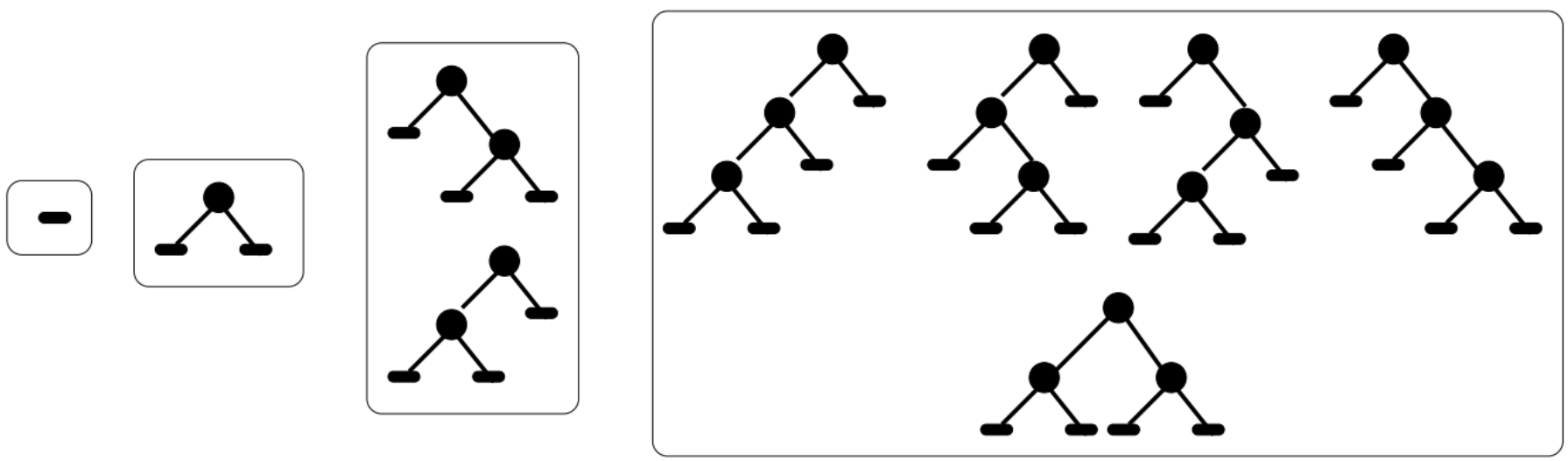
foldLeft applies the function f, in the expanded form, to each element of the list (from left to right) accumulating the result in acc (effectively - tail recursion):

```
(List(1,2,3,4) foldLeft 0) ((sum,x)=>sum+x)
(0 /: List(1,2,3,4)) ((sum,x)=>sum+x)
```

foldRight is a standard recursion:

```
(List(1,2,3,4) foldRight 0) ((x,sum)=>sum+x)
(List(1,2,3,4) :\ 0) ((x,sum)=>sum+x)
```

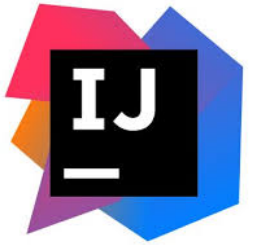
Algebraic data type (ADT)



Type definitions are aliases (synonyms)

```
type ParaIX[Param] = (Int,Param)
type ParaIF = ParaIX[Float]
```

A type is a set of certain values.
For example, the Boolean type consists of True and False.



How much value do we have for…

Boolean:

True & False

Day:

Monday or Tuesday or Wednesday or Thursday or Friday or Saturday or Sunday

RequestMethod:

GET or HEAD or POST or PUT or DELETE or TRACE or OPTIONS or CONNECT or PATH

UserRole:

ADMIN or CONSUMER or CONSULTANT

How much value do we have for …

Nothing?

Unit?

Boolean?

Byte?

String?

ADT:

- The concept of data representation with specific and limited values
- Usually a complex type consisting of simple types
- Algebraic data types are defined using type constructors and value constructors.
- Three types of representation:
  - Sum type
  - Product type
  - Hybrid types

Variants (Sum Type)

```
sealed trait Card
case object Club extends Card
case object Heart extends Card
case object Diamond extends Card
case object Spade extends Card
```

Hybrid Type (Sum of Product)

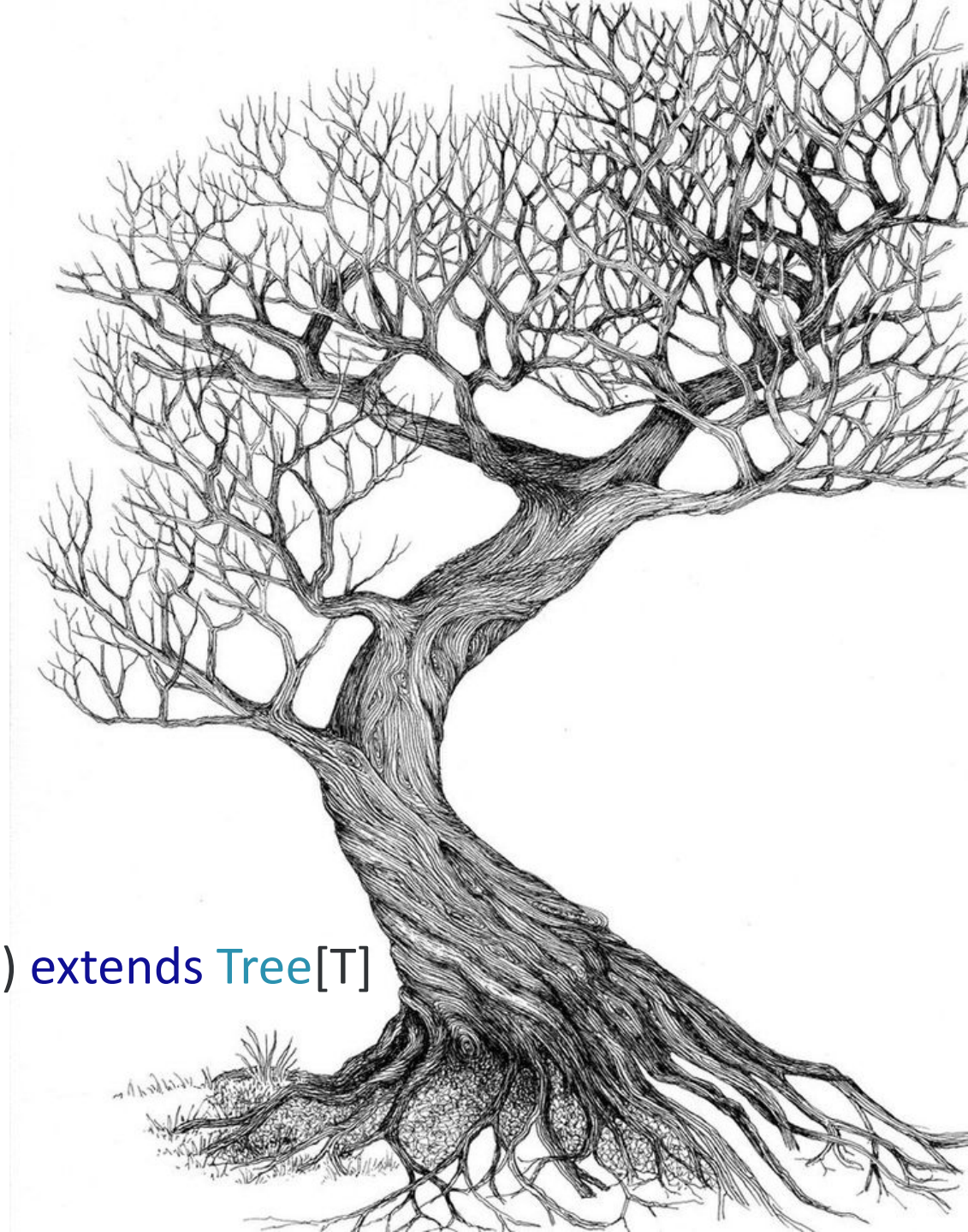
```
sealed trait Pet
case class Cat(name: String) extends Pet
case class Fish(name: String) extends Pet
case class Squid(name: String) extends Pet

def sayHi(p: Pet): String =
  p match {
    case Cat(n)      => "Meow " + n + "!"
    case Fish(n)  => "Hello fishy " + n + "."
    case Squid(n) => "Hi " + n + "."
  }
```

Tree

```haskell
-- Haskell
data Tree a = Leaf | Branch a (Tree a) (Tree a)
```

```scala
// Scala
sealed trait Tree[+T]
case object Leaf extends Tree[Nothing]
case class Branch[T](value: T, left: Tree[T], right: Tree[T]) extends Tree[T]
```



Option / Try

```
sealed trait Option[+A]
case object None extends Option[Nothing]
case class  Some[A](a: A) extends Option[A]
```

```
sealed trait Try[+A]
case class Success[A](a: A) extends Try[A]
case class Failure[A](t: Throwable) extends Try[A]
```

FP Principles

- Purity
- Immutability
- High Order
- Composition
- Currying

michal.szczepanik@pwr.edu.pl