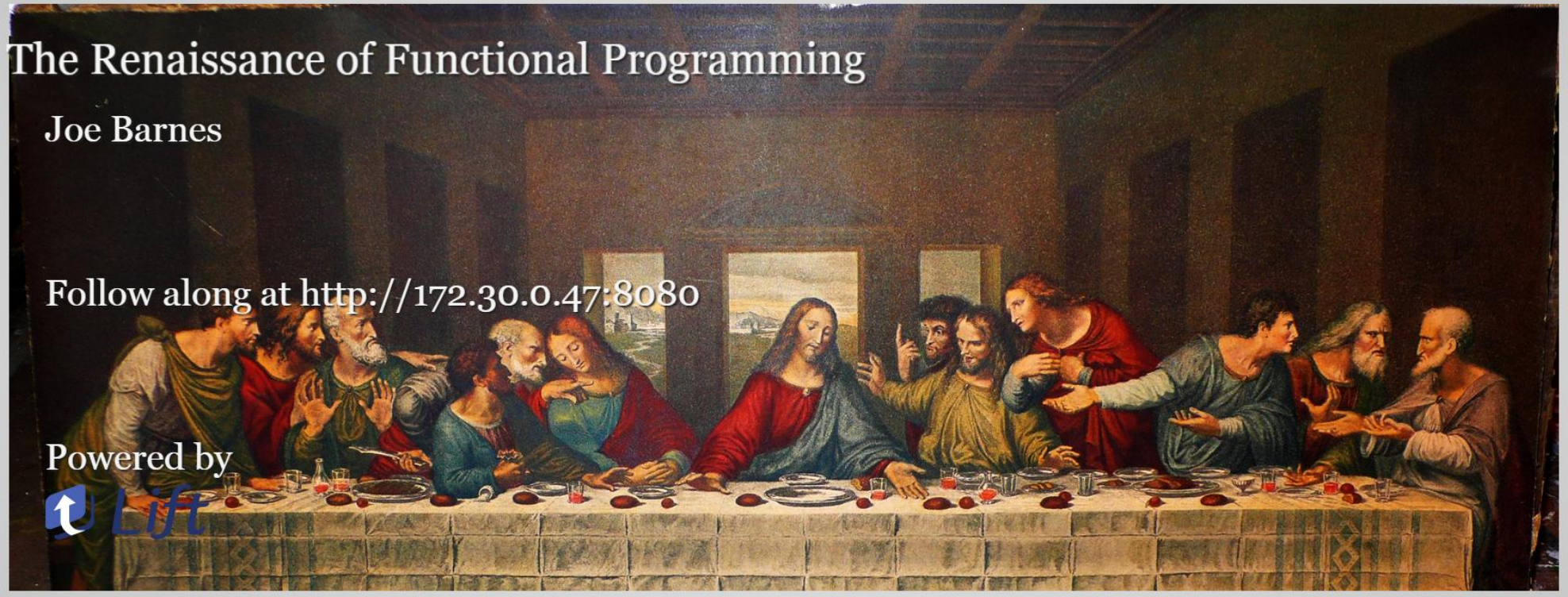
The Renaissance of Functional Programming

Joe Barnes

Follow along at http://172.30.0.47:8080

Powered by



- Influential trend in computing technology
- Programming with constrained power
- Fundamental difference of functional programming
- Code examples

So what's going on in computing these days?

Moore's law is the observation that, over the history of computing hardware, the speed of integrated circuits doubles approximately every two years.

ning

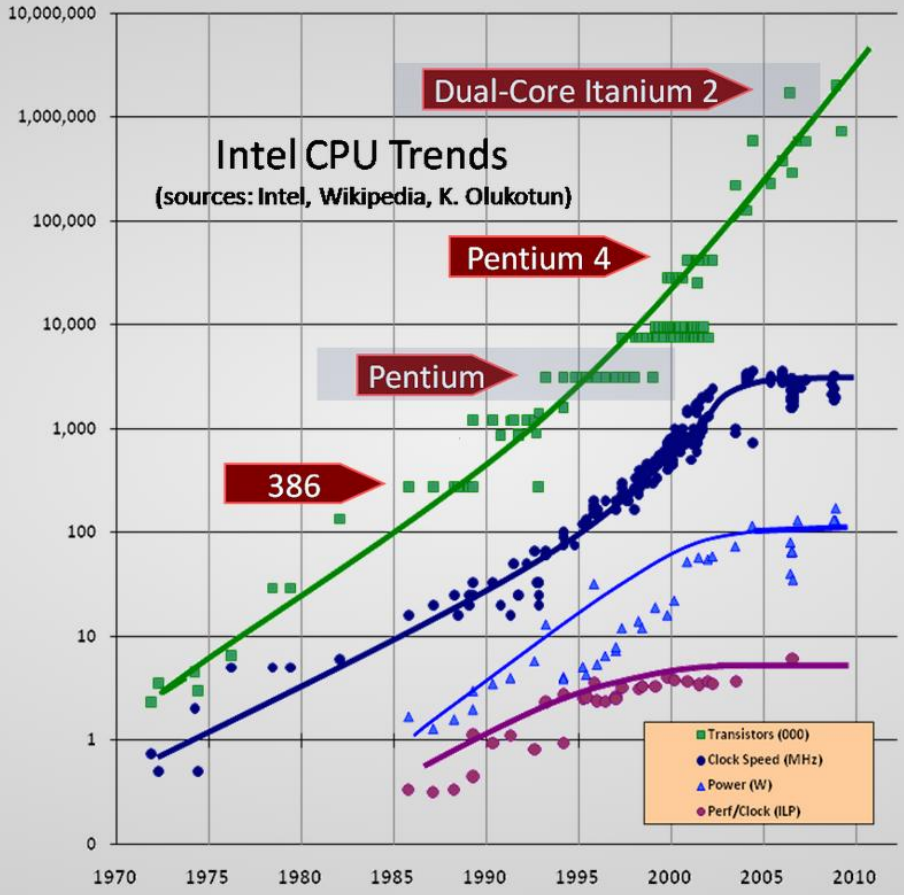
FALSE

Moore's law is the observation that, over the history of computing hardware, the speed of integrated circuits doubles approximately every two years.

Moore's law is the observation that, over the history of computing hardware, the speed of

Moore's law is the observation that, over the history of computing hardware, the ~~speed of~~ **number of transistors on** integrated circuits doubles approximately every two years.

That has mostly continued, but the speed up is over.


Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)
Dual-Core Itanium 2
Pentium 4
Pentium
386
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
1970
1975
1980
1985
1990
1995
2000
2005
2010
Transistors (000)
Clock Speed (MHz)
Power (W)
Perf/Clock (ILP)

"The free lunch is over!!!"

-Some C/C++ guy named Herb Sutter

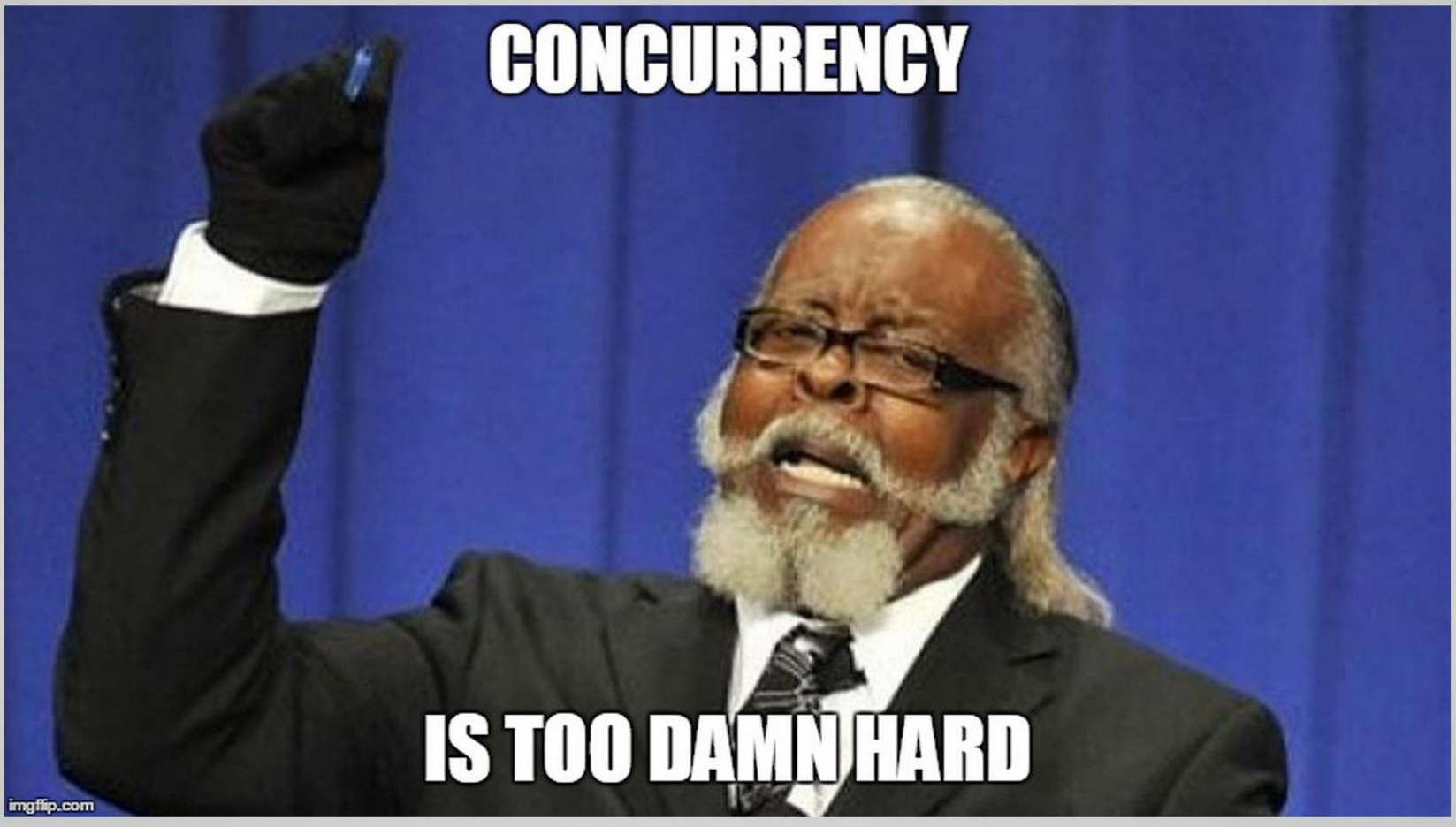
More cores, not more speed

"C++ has been designed for single thread programming, and parallel programming requires a revolutionary rather than evolutionary change. Two words: ***data races****."*

-Bartosz Milewski (another C++ guy, who happens to look like Tony Iommi)

*"**The programmer must ensure** read and write access to objects is properly coordinated (or "synchronized") between threads."*

-Java Concurrency | Wikipedia


CONCURRENCY
IS TOO DAMN HARD
imgflip.com

The developer bears the cognitive load of solving the business problem *and* juggling shared mutable state.

A **data race** occurs when two concurrent threads access a shared variable and when

- at least one access is a write and
- the threads use no explicit mechanism to prevent the accesses from being simultaneous.

Recipe

- Two concurrent threads
- Shared memory
- At least one write to a *variable*
- Mechanism not used

Recipe

- Two concurrent threads → *Here to stay*
- Shared memory
- At least one write to a *variable*
- Mechanism not used

Recipe

- Two concurrent threads → *Here to stay*
- Shared memory → *If not, incur wrath of OS overhead*
- At least one write to a *variable*
- Mechanism not used

Recipe

- Two concurrent threads → *Here to stay*
- Shared memory → *If not, incur wrath of OS overhead*
- At least one write to a *variable*
- Mechanism not used → *Programmer error, also here to stay*

Recipe

- Two concurrent threads → *Here to stay*
- Shared memory → *If not, incur wrath of OS overhead*
- At least one write to a *variable* → *Well there's an idea...*
- Mechanism not used → *Programmer error, also here to stay*

More specifically, eliminate writes after reads.

- Mark everything `const` / `final`
- No more reassigning names
- No more variables

two concurrent threads access a shared va
write and
icit mechanism to prevent the accesses fro

ognitive load of solving the business proble

mutable state.

More specifically, eliminate writes after reads.

- Mark everything `const` / `final`
- No more reassigning names
- No more variables

Revoking powers/freedoms is not a new thing in software.

1968 – Structured Programming

- Edsger Dijkstra writes *Go To Statement Considered Harmful*
- Revoked the `goto`

1968 – Structured Programming

- Edsger Dijkstra writes *Go To Statement Considered Harmful*
- Revoked the `goto`

1966 – Object Oriented Programming

- Ole-Johan Dahl and Kristen Nygaard create Simula 67
- Revoked the need for function pointers
- Modern OO constrains visibility/scope

- Revoked the `goto`

1966 – Object Oriented Programming

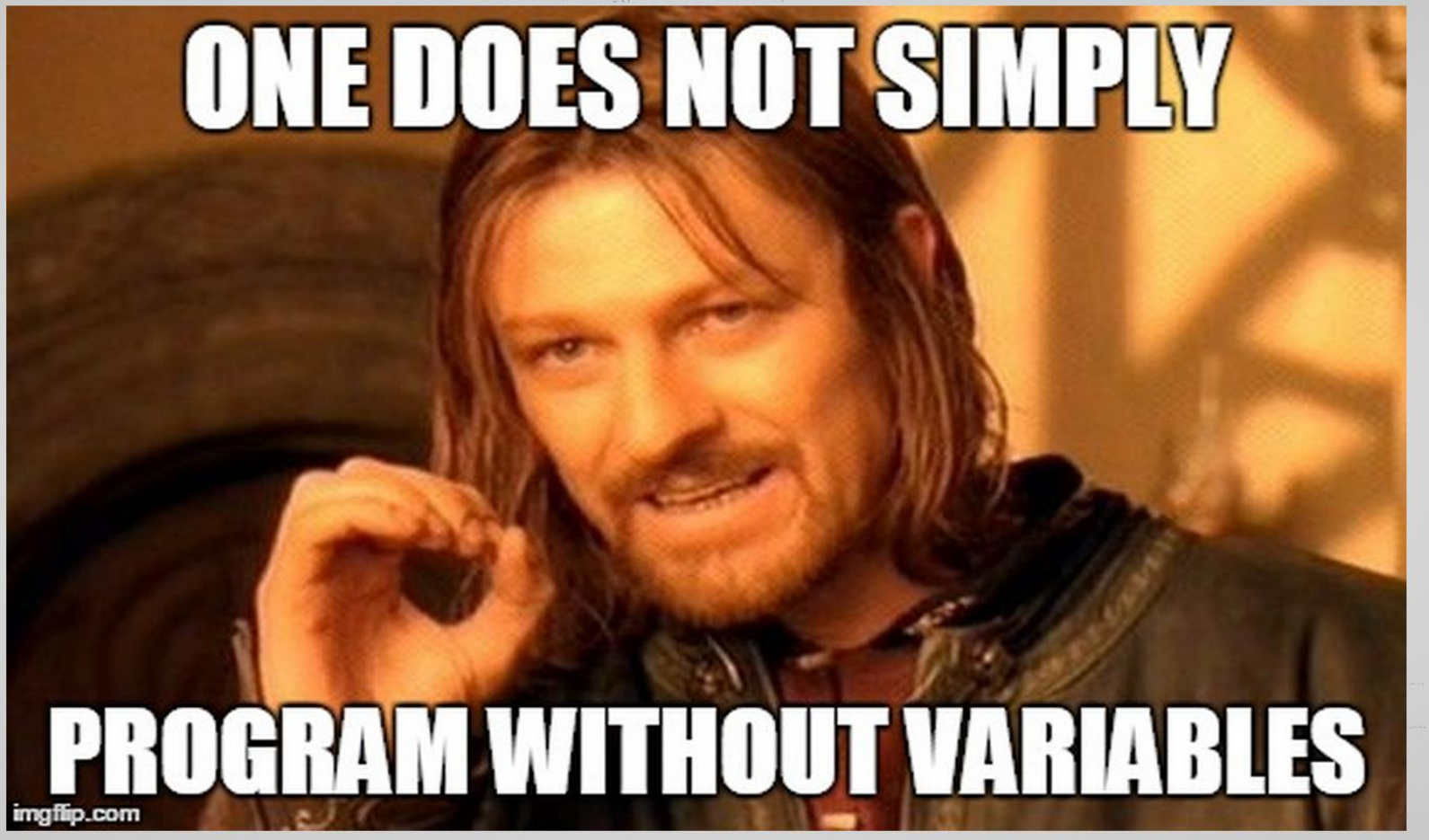
- Ole-Johan Dahl and Kristen Nygaard create Simula 67
- Revoked the need for function pointers
- Modern OO constrains visibility/scope

1957 – Functional Programming

- John McCarthy creates Lisp based on Alonzo Church's Lambda Calculus circa 1930's
- Revokes re-assignments and mutation (well, it never had them in the first place)

- Revoked the goto

1966 – Object Oriented Programming
- Ole-Johan Dahl and Kristen Nygaard create Simula 67



Even managers can do it.

Actually, I bet you've written something like this before...

```
$ find . -name *.java | xargs grep -l "function" | wc -l
```

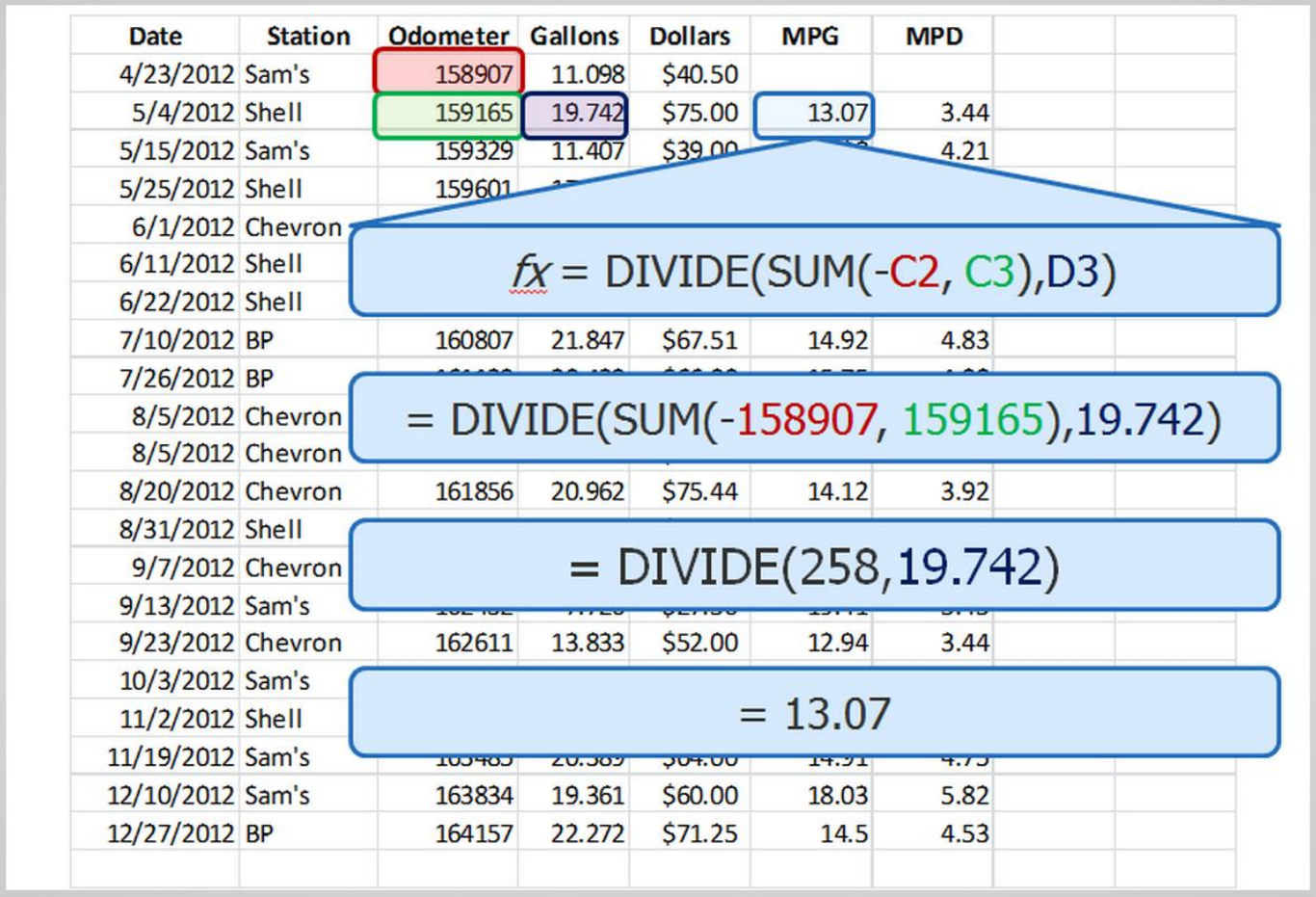
```
$ find . -name *.java | xargs grep -l "function" | wc -l
# A function returning a list of strings
```

```
$ find . -name *.java | xargs grep -l "function" | wc -l
# A function returning a list of strings
# A function filtering a list of strings
```

```
$ find . -name *.java | xargs grep -l "function" | wc -l
# A function returning a list of strings
# A function filtering a list of strings
# A function that returns the size of a list
```

Even managers can do it.

Date	Station	Odometer	Gallons	Dollars	MPG	MPD		
4/23/2012	Sam's	158907	11.098	$40.50				
5/4/2012	Shell	159165	19.742	$75.00	13.07	3.44		
5/15/2012	Sam's	159329	11.407	$39.00	14.38	4.21		
5/25/2012	Shell	159601	17.226	$62.00	15.79	4.39		
6/1/2012	Chevron	159790	12.289	$43.00	15.38	4.4		
6/11/2012	Shell	160146	22.36	$76.00	15.92	4.68		
6/22/2012	Shell	160481	21.521	$71.00	15.57	4.72		
7/10/2012	BP	160807	21.847	$67.51	14.92	4.83		
7/26/2012	BP	161129	20.439	$66.00	15.75	4.88		
8/5/2012	Chevron	161369	18.005	$63.00	13.33	3.81		
8/5/2012	Chevron	161560	10.861	$38.00	17.59	5.03		
8/20/2012	Chevron	161856	20.962	$75.44	14.12	3.92		
8/31/2012	Shell	162122	18.09	$68.00	14.7	3.91		
9/7/2012	Chevron	162282	11.971	$45.00	13.37	3.56		
9/13/2012	Sam's	162432	7.726	$27.50	19.41	5.45		
9/23/2012	Chevron	162611	13.833	$52.00	12.94	3.44		
10/3/2012	Sam's	162884	18.232	$63.25	14.97	4.32		
11/2/2012	Shell	163181	19.863	$69.50	14.95	4.27		
11/19/2012	Sam's	163485	20.389	$64.00	14.91	4.75		
12/10/2012	Sam's	163834	19.361	$60.00	18.03	5.82		
12/27/2012	BP	164157	22.272	$71.25	14.5	4.53		


Date
Station
Odometer
Gallons
Dollars
MPG
MPD
4/23/2012 Sam's 158907 11.098 $40.50
5/4/2012 Shell 159165 19.742 $75.00 13.07 3.44
5/15/2012 Sam's 159329 11.407 $39.00 4.21
5/25/2012 Shell 159601
6/1/2012 Chevron
6/11/2012 Shell
6/22/2012 Shell
7/10/2012 BP 160807 21.847 $67.51 14.92 4.83
7/26/2012 BP
8/5/2012 Chevron
8/5/2012 Chevron
8/20/2012 Chevron 161856 20.962 $75.44 14.12 3.92
8/31/2012 Shell
9/7/2012 Chevron
9/13/2012 Sam's
9/23/2012 Chevron 162611 13.833 $52.00 12.94 3.44
10/3/2012 Sam's
11/2/2012 Shell
11/19/2012 Sam's
12/10/2012 Sam's 163834 19.361 $60.00 18.03 5.82
12/27/2012 BP 164157 22.272 $71.25 14.5 4.53
fx = DIVIDE(SUM(-C2, C3),D3)
= DIVIDE(SUM(-158907, 159165),19.742)
= DIVIDE(258,19.742)
= 13.07

ered Harmful

*An expression is said to be **referentially** without changing the behavior of a prog the same effects and output on the same*

6/11/2012	Shell	160146	22.36	$76.00	15.92	4.68
6/22/2012	Shell	160481	21.521	$71.00	15.57	4.72
7/10/2012	BP	160807	21.847	$67.51	14.92	4.83
7/26/2012	BP	161129	20.439	$66.00	15.75	4.88
8/5/2012	Chevron	161369	18.005	$63.00	13.33	3.81

*An expression is said to be **referentially transparent** if it can be replaced with its value without changing the behavior of a program (in other words, yielding a program that has the same effects and output on the same input).*

-Somebody smart on Wikipedia

```
$ find . -name *.java | \
$ xargs grep -l "function" | \
$ wc -l
```

```
$ [Functions.java, Lambdas.java, Objects.java] | \
$ xargs grep -l "function" | \
$ wc -l
```

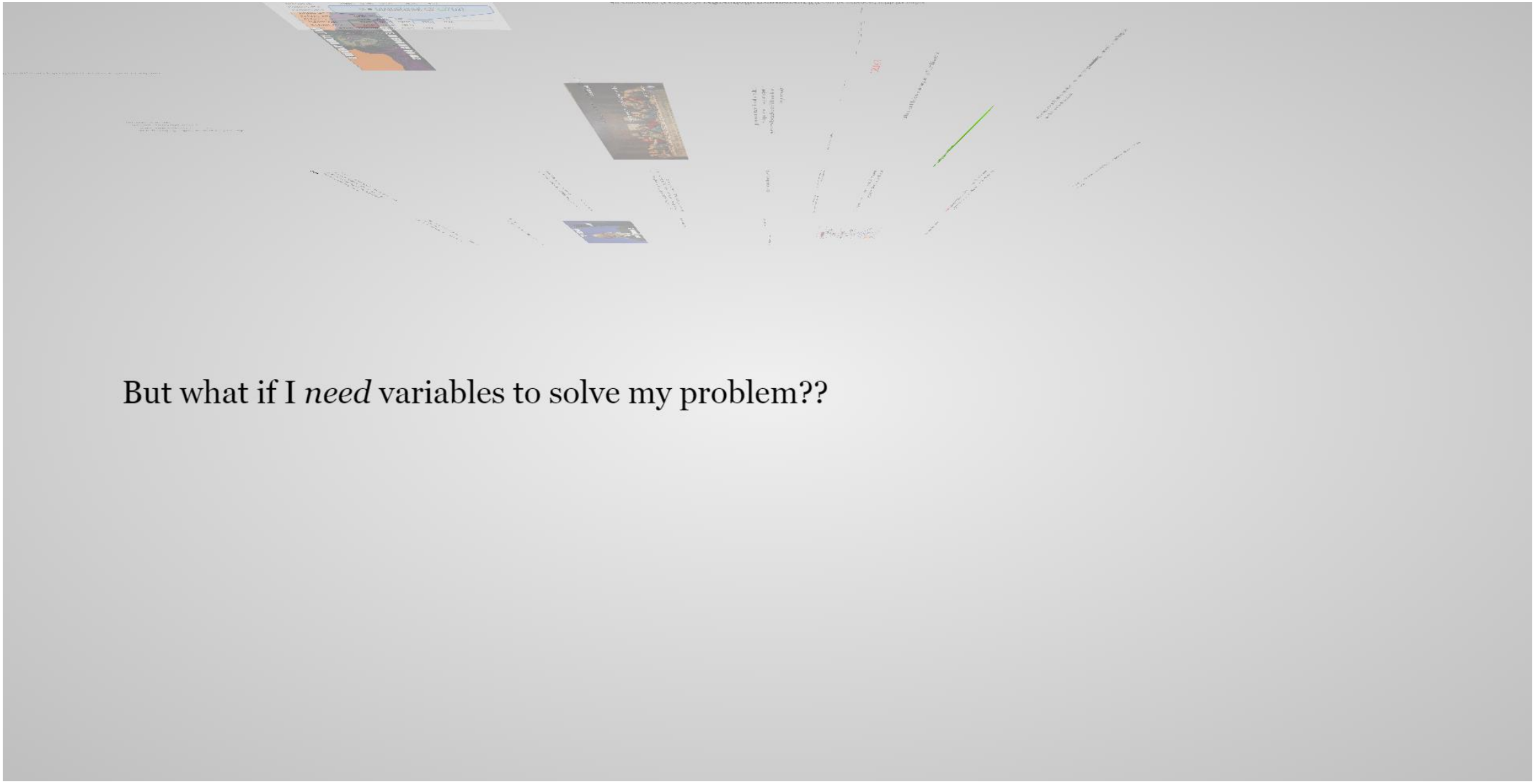
```
$ [Functions.java, Lambdas.java] | \
$ wc -l
```


$ 2

By giving ourselves a constraint, we gain a new power


THEY CAN TAKE OUR WRITES
BUT THEY'LL NEVER TAKE OUR FREEDOM
imgflip.com

Contrast this *expression-oriented* approach with the *statement-oriented* approach which describes computation in statements which mutate the state of the running program.



But what if I *need* variables to solve my problem??

```java
public static List<Integer> firstNEvens(int n) {
    List<Integer> evens = new ArrayList<Integer>();
    for(int i=1; i<=n; i++)
        evens.add(i*2);
    return evens;
}
```

```
public static IntStream firstNEvens(int n) {
    return IntStream.rangeClosed(1, n)
            .map((i) -> i * 2);
}
```

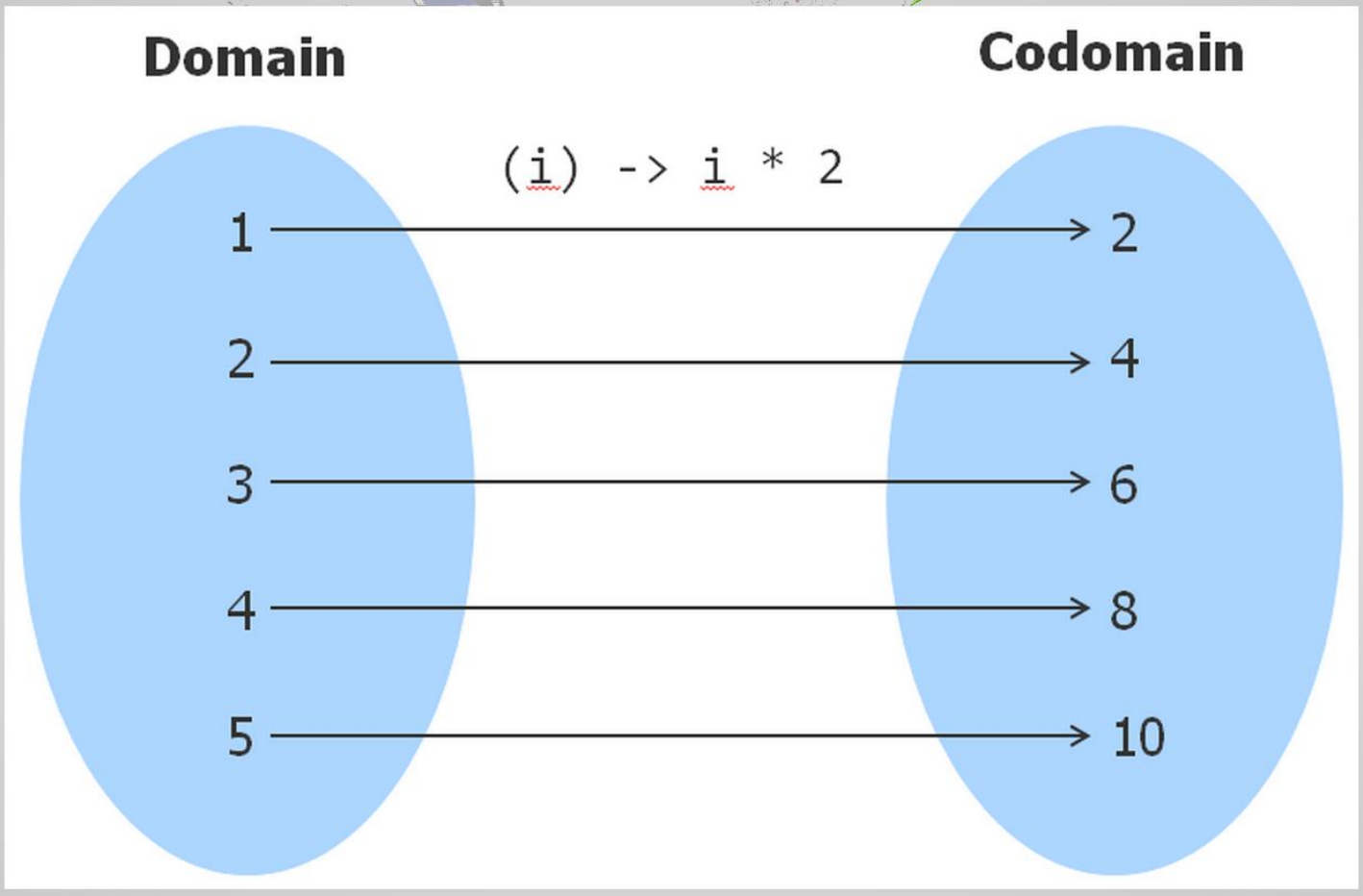

BUT...
AIN'T THAT A VARIABLE??
imgflip.com

```java
// Notice that n and i are never _reassigned_
public static IntStream firstNEvens(int n) {
    return IntStream.rangeClosed(1, n)
            .map((i) -> i * 2);
}
```

Strictly (i.e. *mathematically*) speaking, functions *never* mutate anything.

```
(i) -> i * 2
```

is an *expression* for *describing* a function.


Domain
Codomain
(i) -> i * 2
1
2
3
4
5
2
4
6
8
10

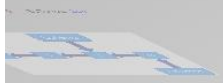
To be precise, a function is a subset of the cross-product of the domain and codomain.

It's just that `(i) -> i * 2` is a far-more robust representation than...

```
{ (1, 2), (2, 4), (3, 6), (4, 8), (5, 10), ... }
```

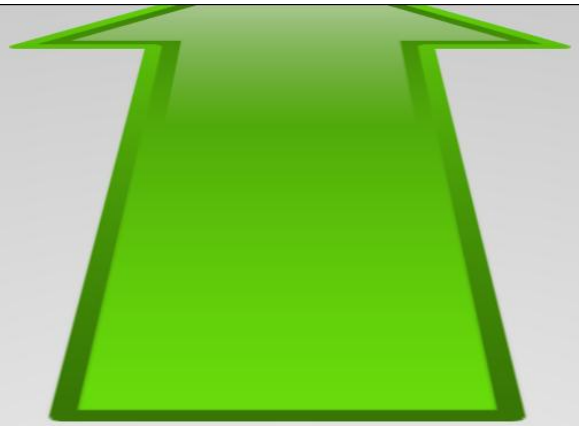
So that's it? Just slap `const` / `final` everywhere and use fancy libraries?

It doesn't sound like the *revolutionary change* Milewski wanted.

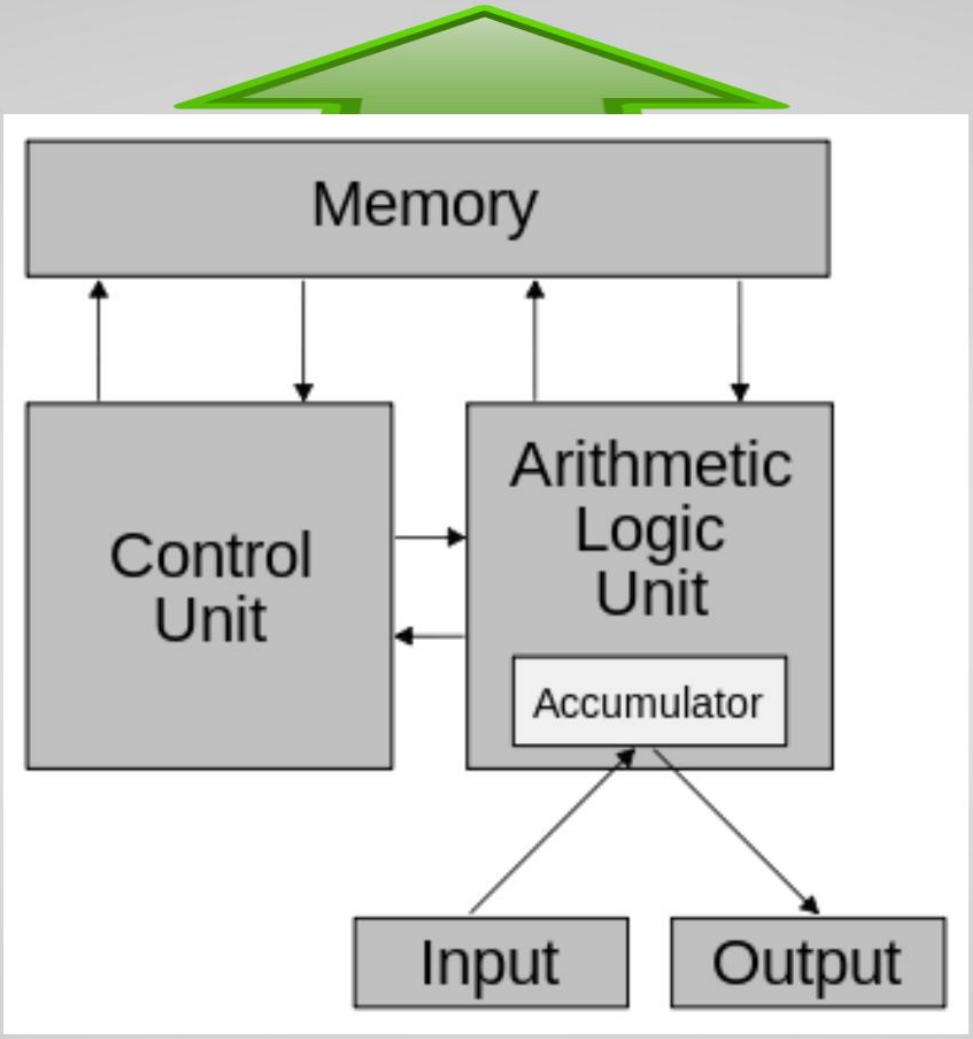


Otherwise we could just summarize functional programming as merely the logical conclusion of best practices.

Perhaps, but that conclusion is so strong it changes the fundamental design of our languages!

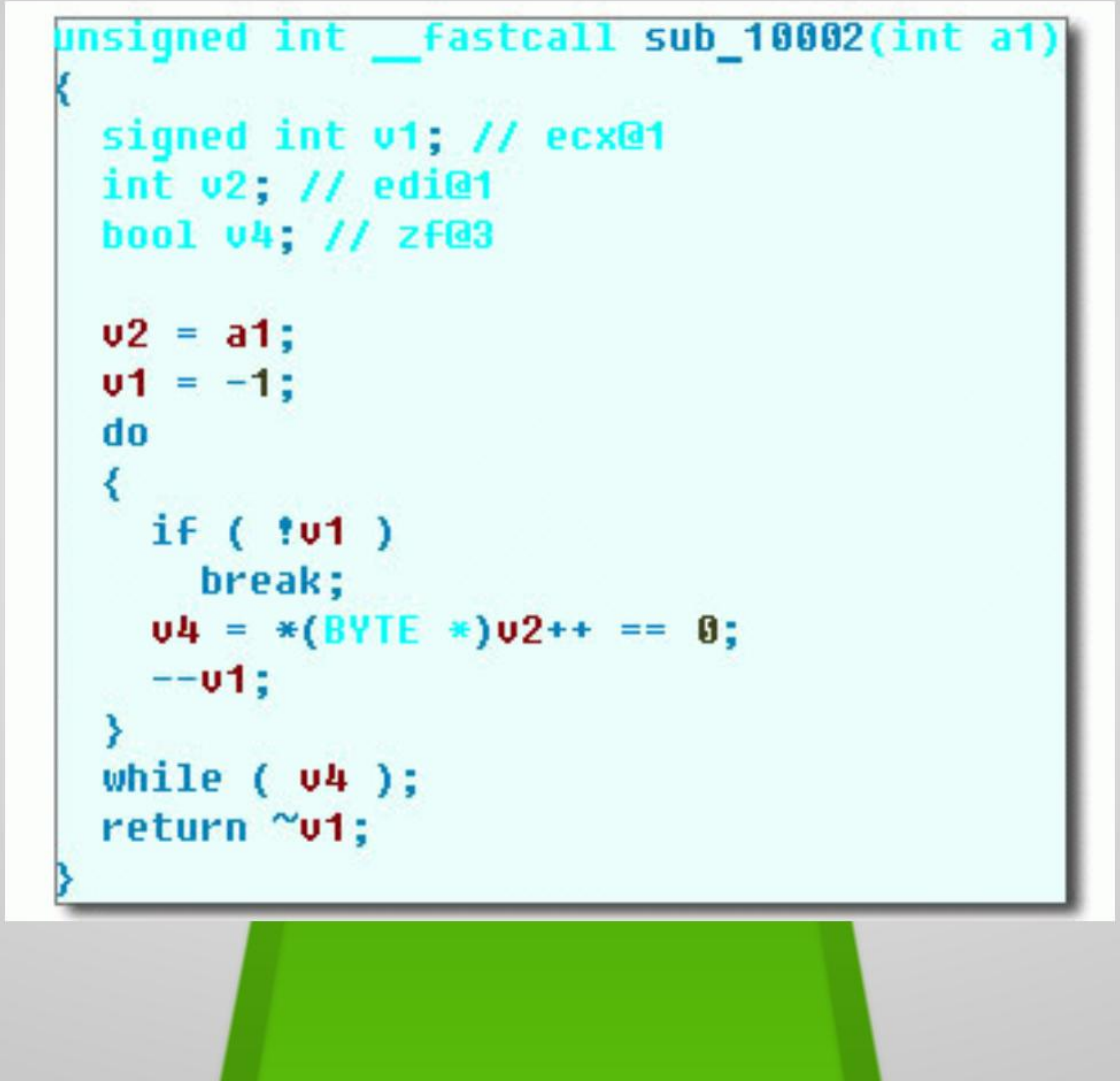


Consider the history of computing and the emergence of imperative languages


Memory
Control Unit
Arithmetic Logic Unit
Accumulator
Input
Output


1001001011011010101
0101100101001110100
1000101110000110110
1010000001010110101
1100101010110000101
1011001000110001000

```
simple_loop:
# parameter 1: %rdi
..B1.1:                                 # Preds ..B1.0
..___tag_value_simple_loop.1:           #2.1
        xorl      %eax, %eax            #3.19
        xorl      %edx, %edx            #5.8
        testq     %rdi, %rdi            #5.16
        jle       ..B1.5                # Prob 10% #5.16
                   #LOE rax rdx rbx rbp rdi r12 r13 r14 r15
..B1.3:                                 # Preds ..B1.1 ..B1.3
        addq      %rdx, %rax            #6.5
        addq      $1, %rdx              #5.19
        cmpq      %rdi, %rdx            #5.16
        jl        ..B1.3                # Prob 82%   #5.16
..B1.5:                                 # Preds ..B1.3 ..B1.1
        ret                             #8.10
        .align    2,0x90
```

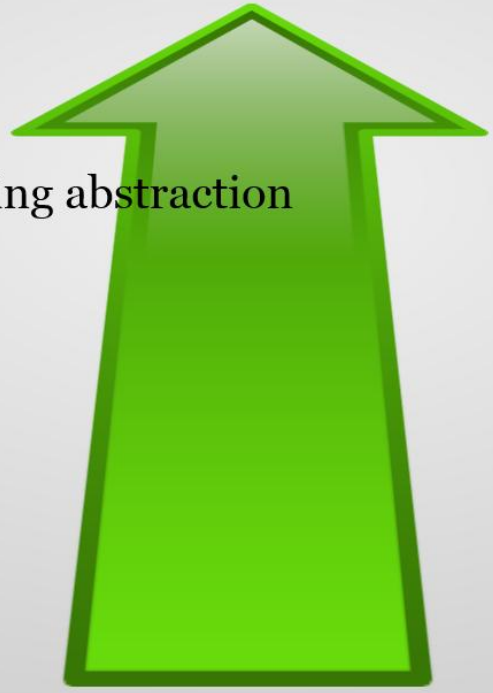

unsigned int __fastcall sub_10002(int a1)
{
signed int v1; // ecx@1
int v2; // edi@1
bool v4; // zf@3
v2 = a1;
v1 = -1;
do
{
if (!v1)
break;
v4 = *(BYTE *)v2++ == 0;
--v1;
}
while (v4);
return ~v1;
}

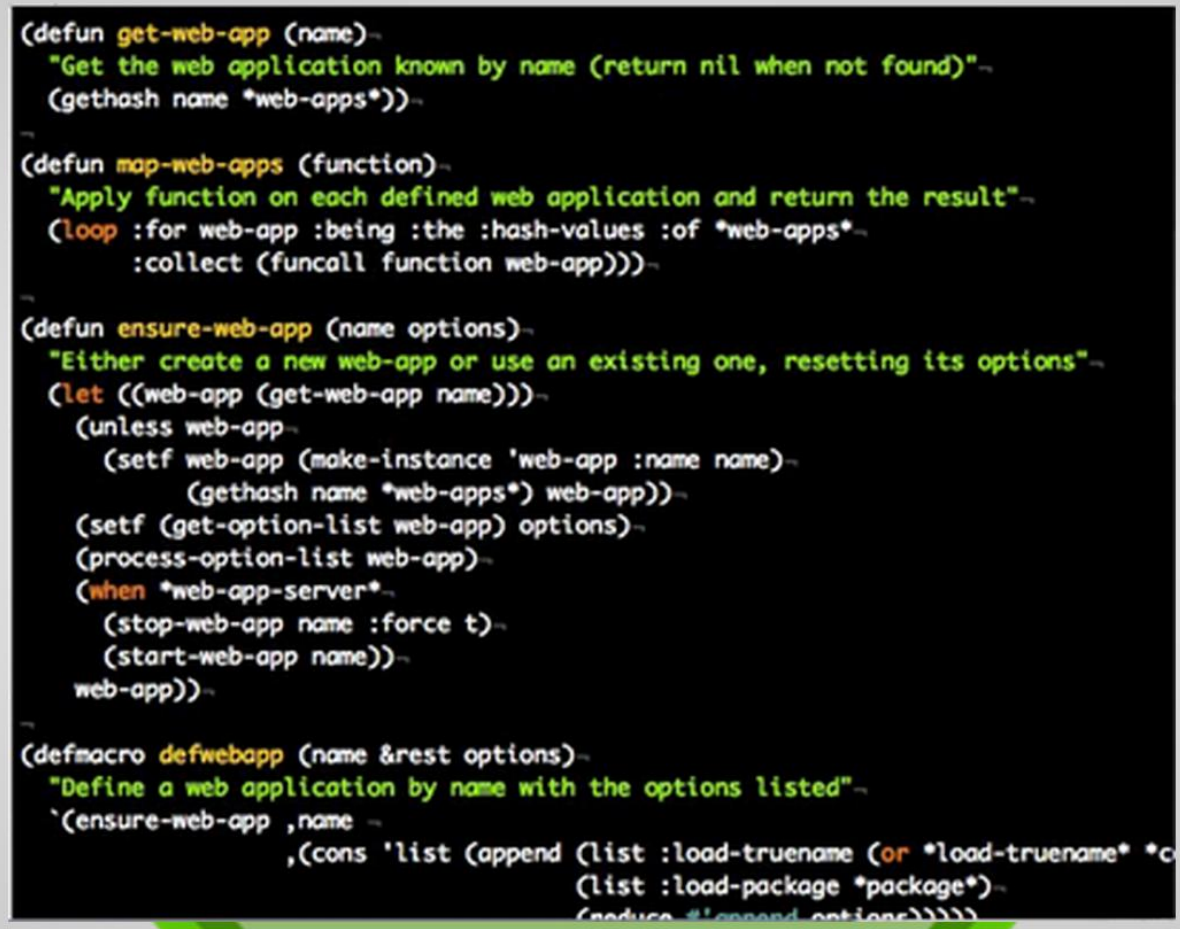
```cpp
#include "maxcpp5.h"

class Example : public MaxCpp5<Example> {
public:
    Example(t_symbol * sym, long ac, t_atom * av) {
        setupIO(2, 2); // inlets / outlets
    }
    ~Example() {}
    void bang(long inlet) {
        outlet_bang(m_outlet[0]);
    }
    void test(long inlet, t_symbol * s, long ac, t_atom * av) {
        outlet_anything(m_outlet[1], gensym("test"), ac, av);
    }
};

extern "C" int main(void) {
    // create a class with the given name:
    Example::makeMaxClass("example");
    REGISTER_METHOD(Example, bang);
    REGISTER_METHOD_GIMME(Example, test);
}
```

```java
/* Block comment */
import java.util.Date;
/**
 * Doc comment here for SomeClass
 * @version 1.0
 */
public class SomeClass { // some comment
  private String field = "Hello World";
  private double unusedField = 12345.67890;
  private UnknownType anotherString = "AnotherString";
  public SomeClass() {
   //TODO: something
   int localVar = "IntelliJ"; // Error, incompatible types
   System.out.println(anotherString + field + localVar);
   long time = Date.parse("1.2.3"); // Method is deprecated
  }
}
```

History of statement-oriented/imperative programming in a nutshell...

Dealing with difficulty by increasing abstraction





```
(defun get-web-app (name)
  "Get the web application known by name (return nil when not found)"
  (gethash name *web-apps*))

(defun map-web-apps (function)
  "Apply function on each defined web application and return the result"
  (loop :for web-app :being :the :hash-values :of *web-apps*
        :collect (funcall function web-app)))

(defun ensure-web-app (name options)
  "Either create a new web-app or use an existing one, resetting its options"
  (let ((web-app (get-web-app name)))
    (unless web-app
      (setf web-app (make-instance 'web-app :name name)
            (gethash name *web-apps*) web-app))
    (setf (get-option-list web-app) options)
    (process-option-list web-app)
    (when *web-app-server*
      (stop-web-app name :force t)
      (start-web-app name))
    web-app))

(defmacro defwebapp (name &rest options)
  "Define a web application by name with the options listed"
  `(ensure-web-app ,name
                   ,(cons 'list (append (list :load-truename (or *load-truename* *c
                                        (list :load-package *package*)
                                        (reduce #'append options)))))
```

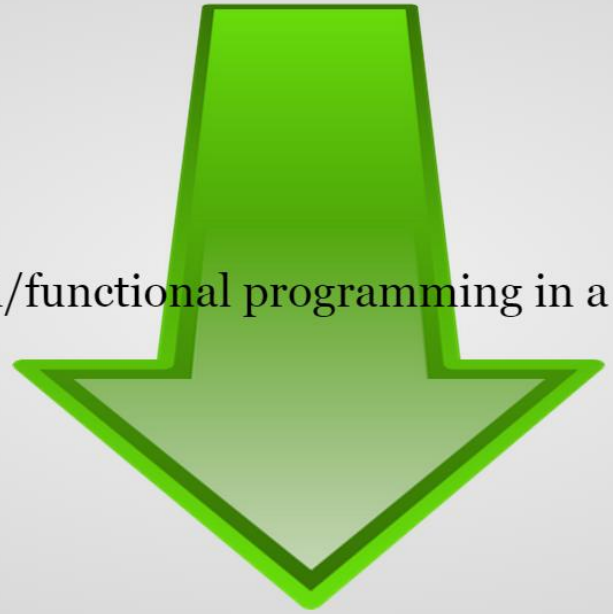
```
fun append (xs, ys) =
    if null xs
    then ys
    else (hd xs):: append (tl xs, ys)

fun map (f, xs) =
    case xs of
        [] => []
      | x :: xs' => (f x)::(map (f, xs'))

val a = map (increment, [4,8,12,16])
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

```haskell
-- Type annotation (optional)
fib :: Int -> Integer

-- With self-referencing data
fib n = fibs !! n
        where fibs = 0 : scanl (+) 1 fibs
        -- 0,1,1,2,3,5,...

-- Same, coded directly
fib n = fibs !! n
        where fibs = 0 : 1 : next fibs
              next (a : t@(b:_)) = (a+b) : next t

-- Similar idea, using zipWith
fib n = fibs !! n
        where fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

-- Using a generator function
fib n = fibs (0,1) !! n
        where fibs (a,b) = a : fibs (b,a+b)
```

```erlang
-module(primes).
-export([factor/1]).

% factors integers >=2 into its prime numbers.
factor(N) when is_integer(N), N>=2 ->
        factor(N, 2, []).

% factors N with the divisor D into its Factors.
factor(N, D, Factors) when D*D > N ->
        [N | Factors];
factor(N, D, Factors) when N rem D =:= 0 ->
        factor(N div D, D, [D|Factors]);
factor(N, D, Factors) ->
        factor(N, D+1, Factors).
```

```
package ListMethods;

object ListMethods {
  def main(args: Array[String]) {
    /// listcat.scala
    //  Scala List - prepend operator
    val oneTwo = List(1, 2)
    val threeFour = List(3, 4)
    val oneTwoThreeFour = oneTwo ::: threeFour  // prepend
    println(oneTwo + " and " + threeFour + " were not mutated.")
    println("This " + oneTwoThreeFour + " is a new List.")
```

History of expression-oriented/functional programming in a nutshell...



From theory down to practice

Functional programming is more than eliminating variables

It is a fundamentally different approach to programming language design

With functional programming being better suited to take advantage of modern hardware advances, we're seeing a steady uptake in languages which encourage the paradigm.

Really there's more to it than simplifying concurrency

Functional programming simply never had certain problems in the first place

From conception, functional programming is a simpler paradigm.

Consider the reassignment of names (variables) again.

```java
public static int factorial(int n) {
    int acc = 1;
    for(int i=1; i<=n; i++)
        acc *= i;
    return acc;
}
```

t/i	acc
0	1
1	1
2	2
3	6
4	24
5	120

The programmer must juggle both time and value

Less simple than only value

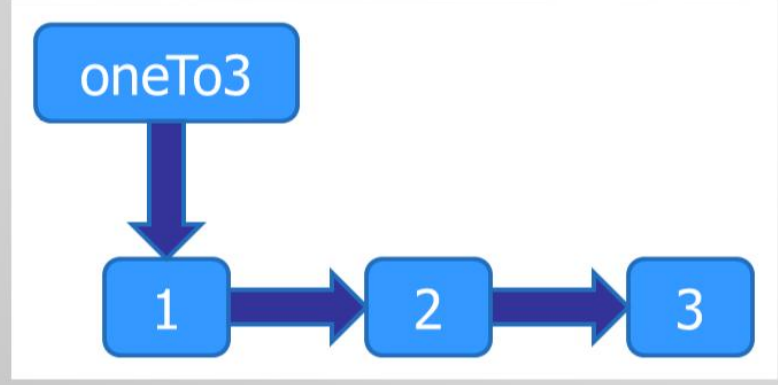
Much like how garbage collection frees developers from juggling resource allocation along with business logic.

Garbage collection has a cost, so what is the cost of immutability?

It depends...

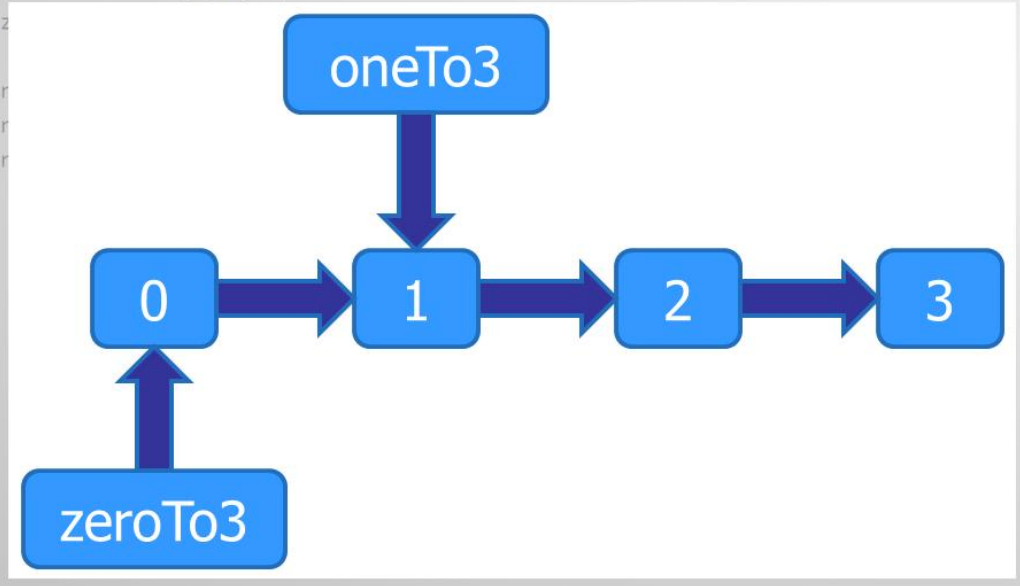
```
test("Prepend produces new instance") {
  val oneTo3  = List(1, 2, 3)
  val zeroTo3 = 0 :: oneTo3

  assert(oneTo3.size  == 3)
  assert(zeroTo3.size == 4)
  assert(oneTo3 != zeroTo3)
}
```

```scala
val oneTo3  = List(1, 2, 3)
```

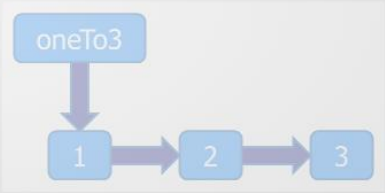


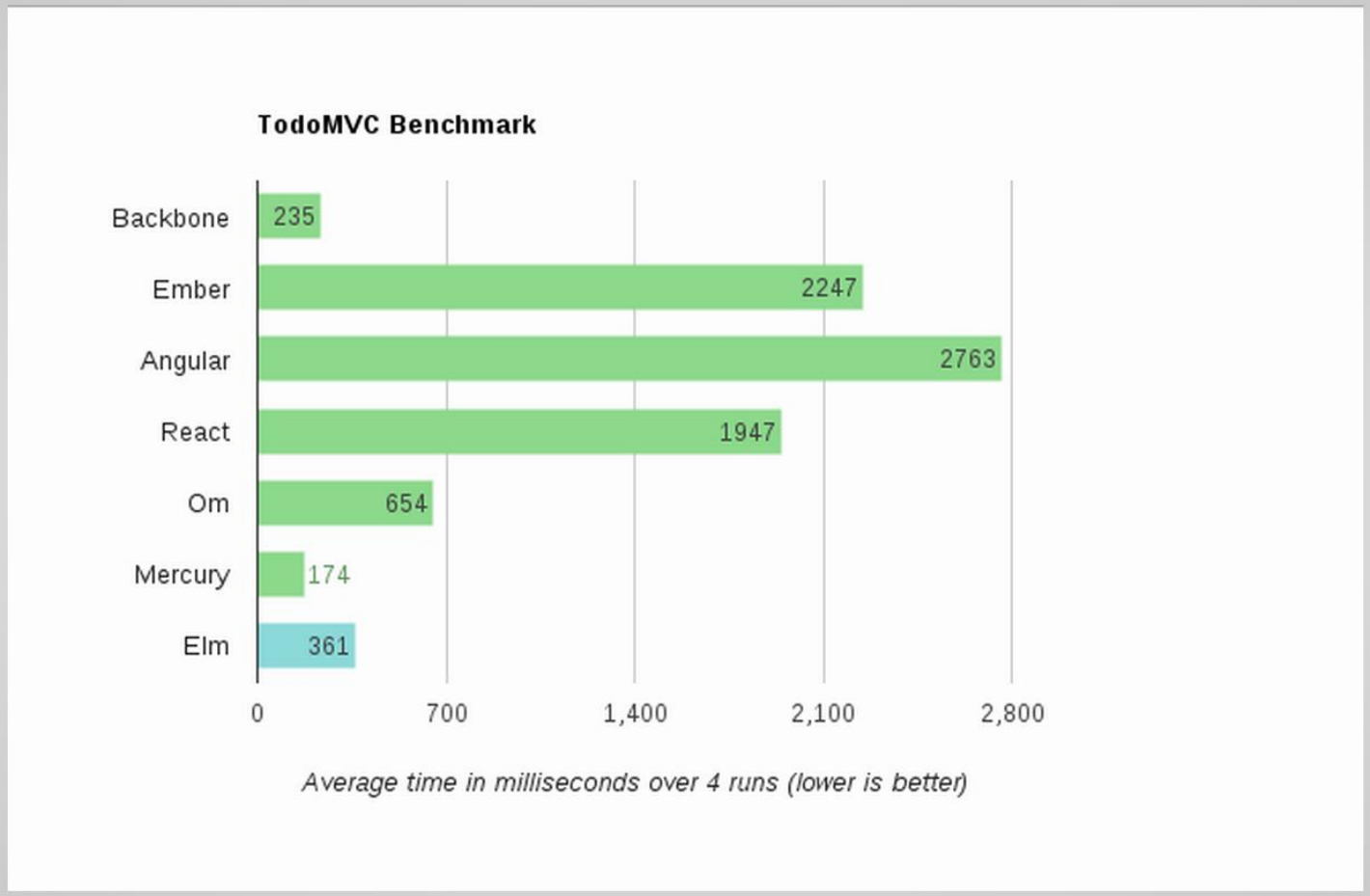
```
val zeroTo3 = 0 :: oneTo3
```

```
test("Prepend produces new instance") {
  val oneTo3  = List(1, 2, 3)
  val z

  asser
  asser
  asser
}
```




```
val oneTo3  = List(1, 2, 3)
```




TodoMVC Benchmark
Backbone
235
Ember
2247
Angular
2763
React
1947
Om
654
Mercury
174
Elm
361
0
700
1,400
2,100
2,800
Average time in milliseconds over 4 runs (lower is better)

Functional programming is far more than just eliminating mutation

It is programming at a higher-level of abstraction

The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

–Edsger W. Dijkstra

```java
public static boolean isPrime(int n) {
    boolean isPrime = true;
    for(int i=2; i<n; i++) {
        if(n % i == 0) {
            isPrime = false;
            break;
        }
    }
    return isPrime;
}
```

```
def isPrime(n:Int) = (2 to (n-1))
    .forall(i => n % i != 0)
```

```java
public static List<Integer> first(int n) {
    List<Integer> primes = new LinkedList<Integer>();
    for(int i=2; primes.size() < n; i++) {
        if(isPrime(i)) primes.add(i);
    }
    return primes;
}
```

```
def first(n:Int) = Stream.from(2)
  .filter(isPrime _)
  .take(n)
```

```
def first(n:Int, p:Int=>Boolean) = Stream.from(2)
  .filter(p)
  .take(n)

def first2(n:Int) = first(n, isPrime _)
```

```java
public static List<Integer> firstNEvens(int n) {
    List<Integer> evens = new ArrayList<Integer>();
    for(int i=1; i<=n; i++)
        evens.add(i*2);
    return evens;
}
```

```scala
def firstNEvens(n:Int) = (1 to n) map (_ * 2)

def firstNEvensPar(n:Int) = (1 to n).par map (_ * 2)
```

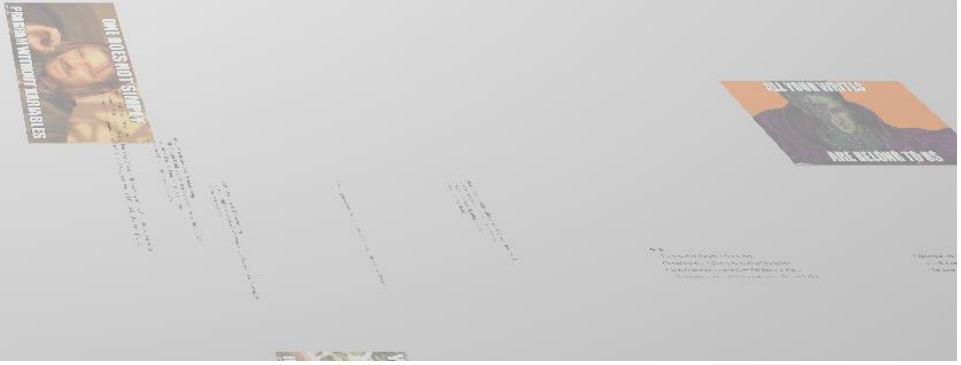
The higher level of abstraction coupled with greater constraints yields a code base that is

- Simpler to reason about
- Composeable

```scala
def ints(i:Int):String = i match {
  case 1 => "one"
  case 2 => "two"
  case n if n % 2 == 0 => s"$n is even"
  case other => s"$other is whatever"
}
```

```scala
val none  = Nil
val one   = 3 :: Nil
val three = 1 :: 2 :: 3 :: Nil

def lists(l:List[Int]) = l match {
  case Nil => "list is empty"
  case last :: Nil => s"$last is alone"
  case head :: tail => s"$head and ${tail.size} others"
}
```

```scala
val str = "12.3"
val num = str match {
  case f(x) => x
  case _ => 0
}
num shouldEqual 12.3f
```



```
val str = "eleven"
val num = str match {
  case f(x) => x
  case _ => 0
}
num shouldEqual 0f
```

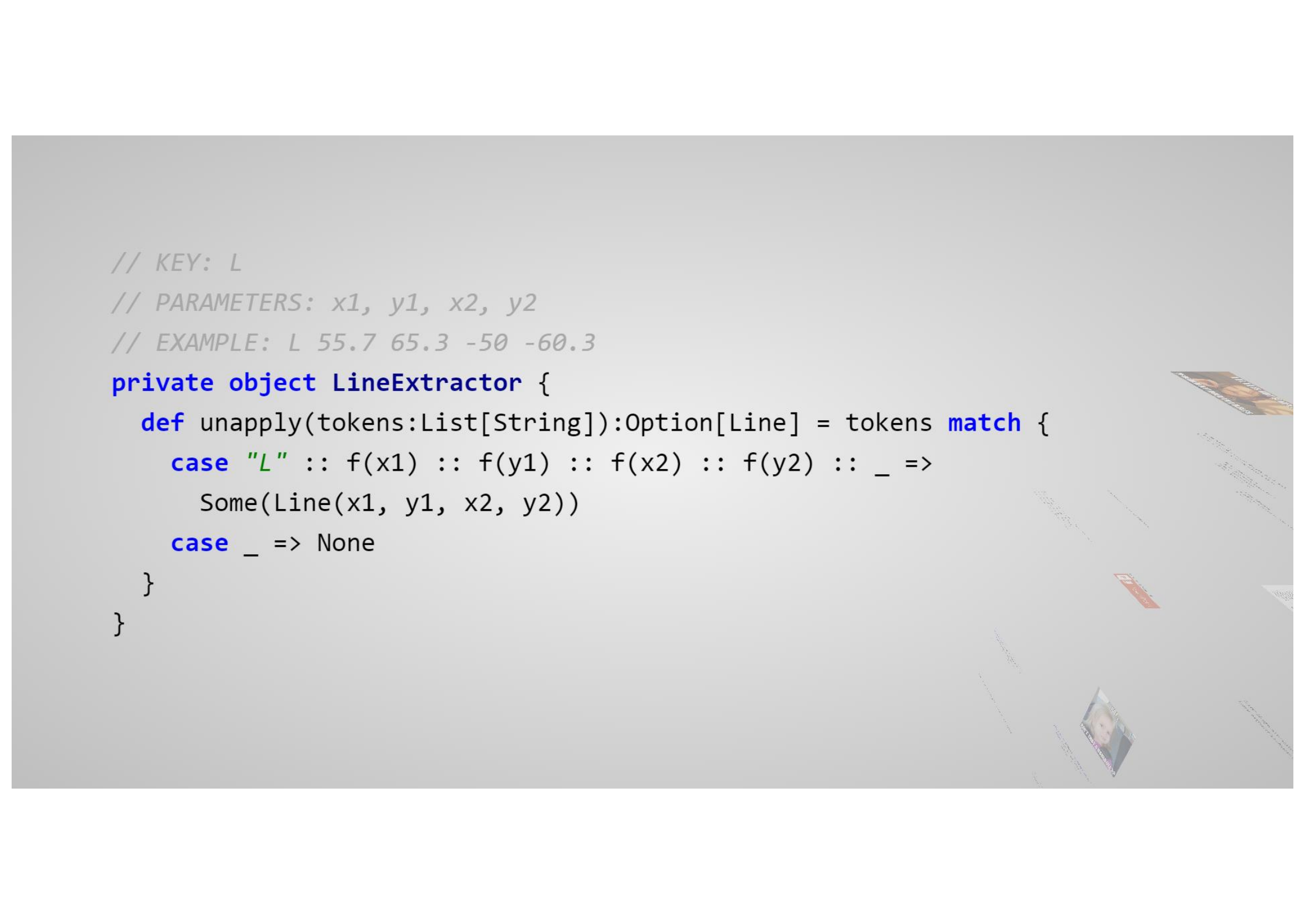
```scala
object f {
  def unapply(s: String): Option[Float] = try {
    Some(s.toFloat)
  } catch {
    case _:Exception => None
  }
}
```

```scala
object z {
  def unapply(s: String): Option[Int] =
    Try(s.toInt).toOption
}
```

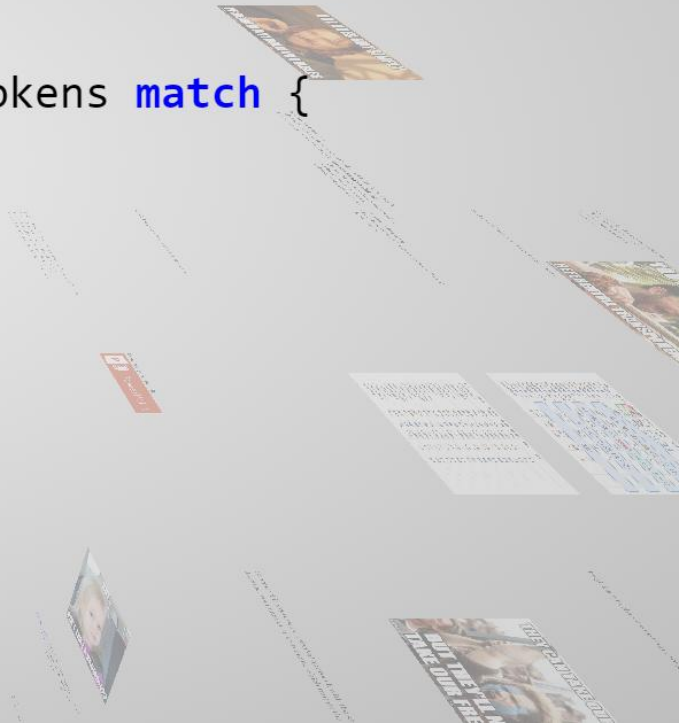
```
L 55.7 65.3 -50 -60.3
C 10 12.2 5.5
T -5 23.2 0 My text label
```

```
trait DrawingObject
case class Circle(x:Float, y:Float, r:Float) extends DrawingObject
case class Line(x1:Float, y1:Float, x2:Float, y2:Float) extends DrawingObject
case class Text(x:Float, y:Float, orientation:Int, text:String) extends DrawingOb
case class Unknown(line:String) extends DrawingObject
```

```
// KEY: L
// PARAMETERS: x1, y1, x2, y2
// EXAMPLE: L 55.7 65.3 -50 -60.3
private object LineExtractor {
  def unapply(tokens:List[String]):Option[Line] = tokens match {
    case "L" :: f(x1) :: f(y1) :: f(x2) :: f(y2) :: _ =>
      Some(Line(x1, y1, x2, y2))
    case _ => None
  }
}
```

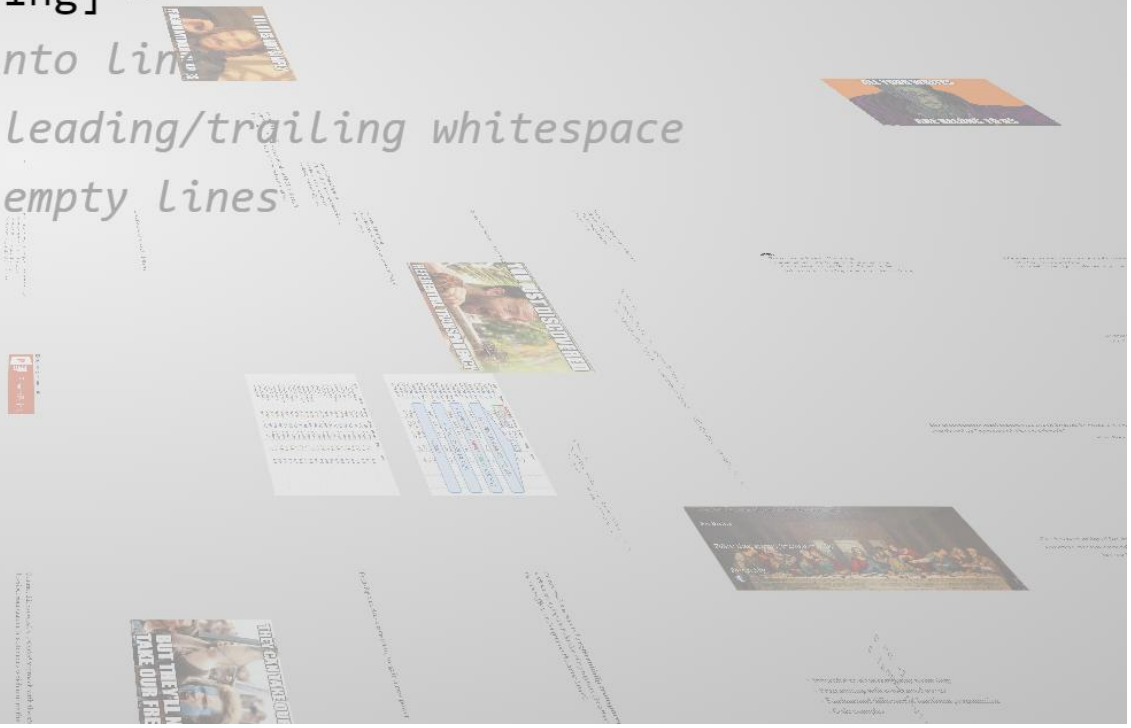


```
// KEY: C
// PARAMETERS: x, y, radius
// EXAMPLE: C 10 12.2 5.5
private object CircleExtractor {
  def unapply(tokens:List[String]):Option[Circle] = tokens match {
    case "C" :: f(x) :: f(y) :: f(r) :: _ =>
      Some(Circle(x, y, r))
    case _ => None
  }
}
```



```
// KEY: T
// PARAMETERS: x, y, orientation, text
//   orientation: 0 => L to R; 1 => Top to Bottom
// EXAMPLE: T -5 23.2 0 My text label
private object TextExtractor {
  def unapply(tokens:List[String]):Option[Text] = tokens match {
    case "T" :: f(x) :: f(y) :: z(orie) :: text =>
      Some(Text(x, y, orie, text mkString " "))
    case _ => None
  }
}
```

```
def cleanLines(file:String):List[String] =
  file.lines                // Split into lin
    .map(_.trim)            // Remove leading/trailing whitespace
    .filter(!_.isEmpty)     // Remove empty lines
    .toList
```



```
def parse(file:String):List[DrawingObject] =
    cleanLines(file).map { line =>
      line.split("""\s+""").toList match {
        case LineExtractor(line) => line
        case CircleExtractor(circle) => circle
        case TextExtractor(text) => text
        case unknown => Unknown(line)
      }
    }
```




A constraint at one level gives us freedom and power at a higher level.

-Some Scala guy named Rúnar Bjarnason

```
public static int someFn(List<Integer> list)
```

```
public static int size(List<Integer> list)
```

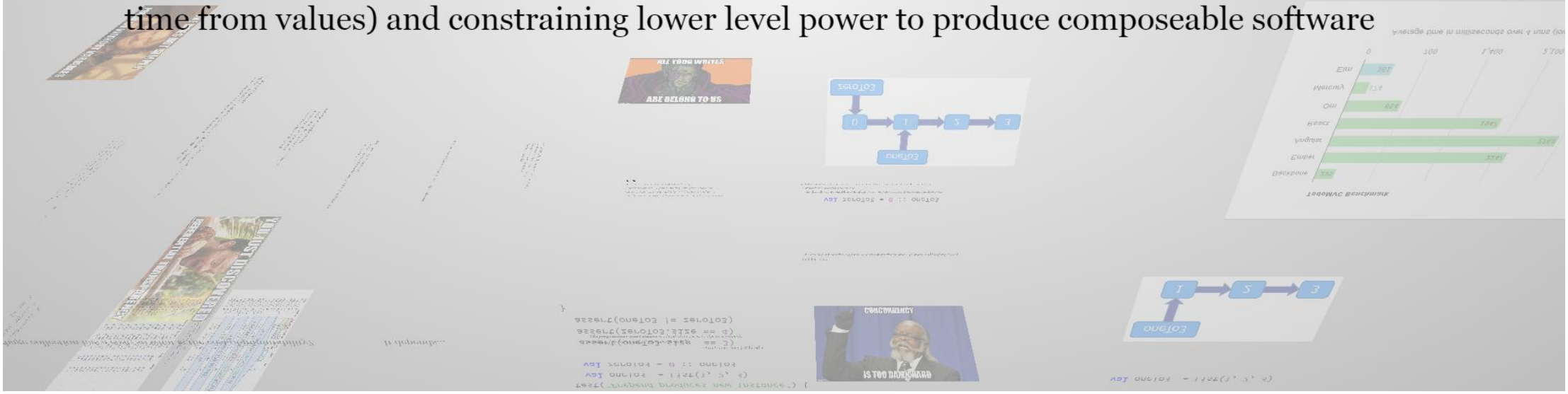
```
public static <A> int size(List<A> list)
```

`size(List<Integer>)` has more power available, but constrains the usages

`size(List<A>)` has constrained its power, offering more power to the caller

By constraining unneeded power at the lowest level (the language), we preserve the maximum possible power for the higher levels => Composition

Functional programming is steeped a culture of simplifying by separating concerns (such as time from values) and constraining lower level power to produce composeable software



It requires a tremendous amount of skill and effort to construct from polymers...

...but even a child can construct a spaceship when the polymers are constrained to the shape of Legos.


Join the Renaissance and have fun again!
Download slides
Presentation Source