# Functional Programming

Theory and Practice

By Alex Reustle

## SciCIG

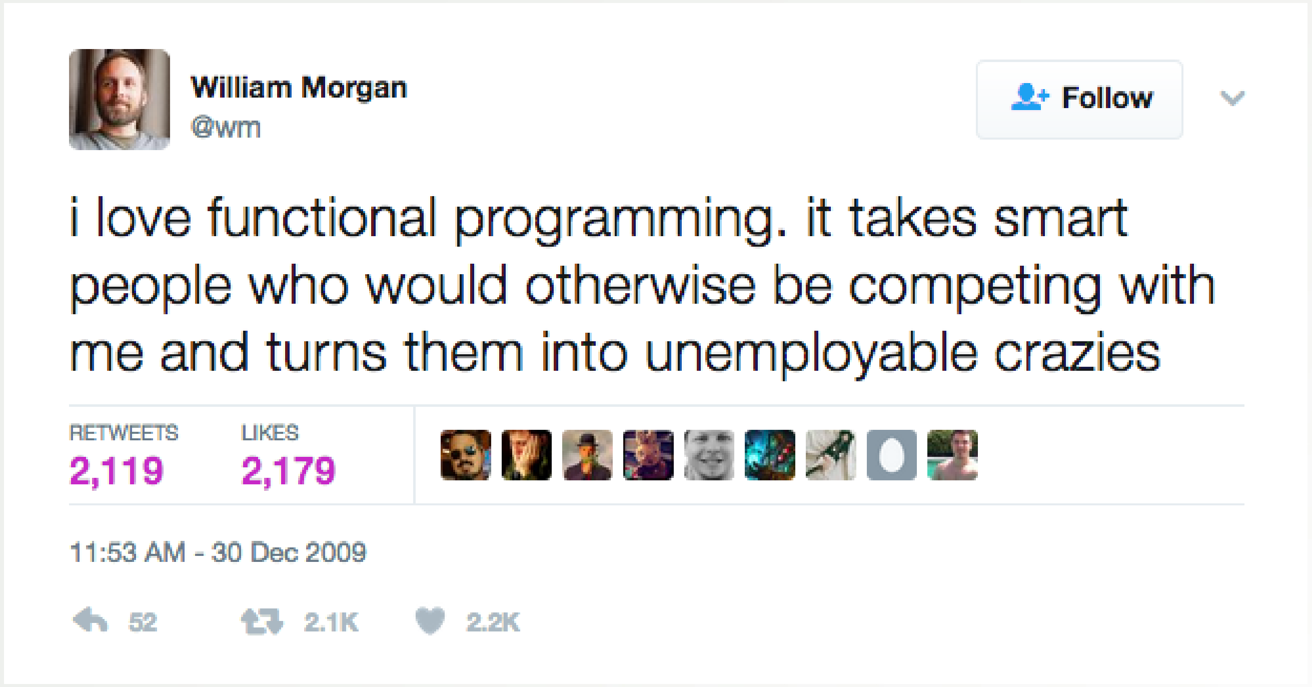
- Scientific Software Common Interest Group
- SESDA III+ contract
  - ( open to all! )
- https://spaces.gsfc.nasa.gov/display/SEDWIKI/SciCIG

- Past Topics
  - AstroPy
  - Git version control
  - Debugging
  - Docker Containers
  - Spatial Indexing

# Functional Programming Love

- *“No matter what language you work in, programming in a functional style provides benefits. You should do it whenever it is convenient, and you should think hard about the decision when it isn't convenient”*
  - John Carmack, Id software
- *“Haskell is faster than C++, more concise than Perl, more regular than Python, more flexible than Ruby, more typeful than C#, more robust than Java, and has absolutely nothing in common with PHP”*
  - Audrey Tang, Perl 6 Compiler project lead
- “*In programming, the opposite of functional is dysfunctional”*
  - *Eric Elliott,* Author of “Programming JavaScript Applications”

# Functional Programming Hate



William Morgan
@wm

Follow

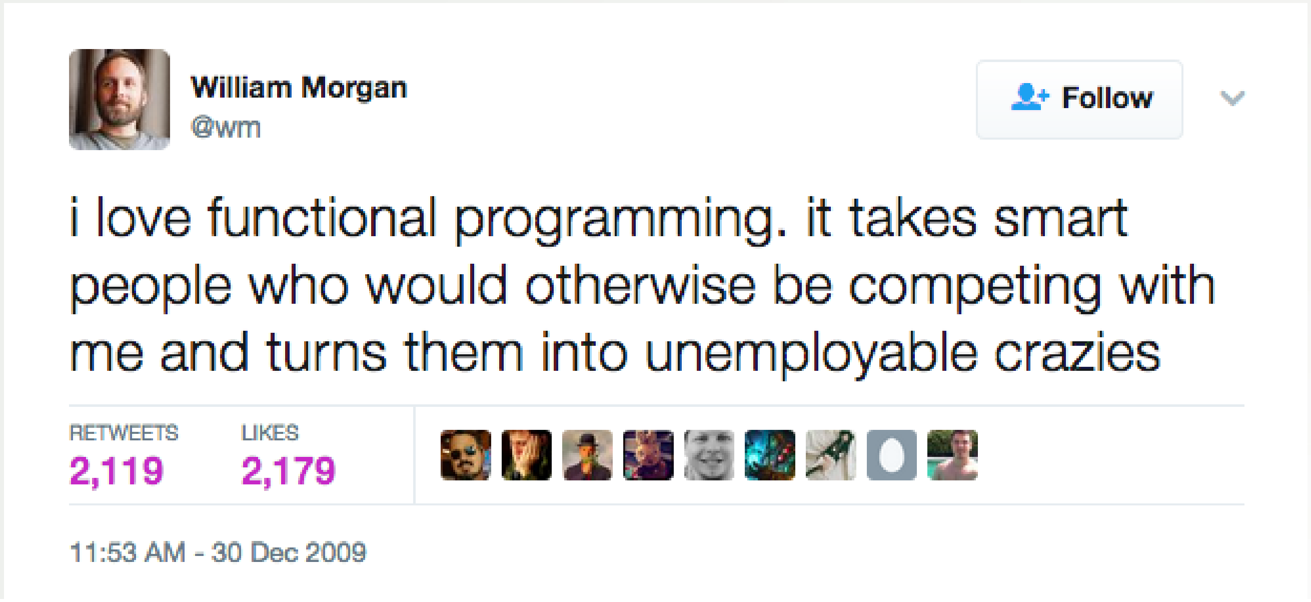
i love functional programming. it takes smart people who would otherwise be competing with me and turns them into unemployable crazies

RETWEETS 2,119 LIKES 2,179

11:53 AM - 30 Dec 2009

52 2.1K 2.2K

# Functional Programming Hate

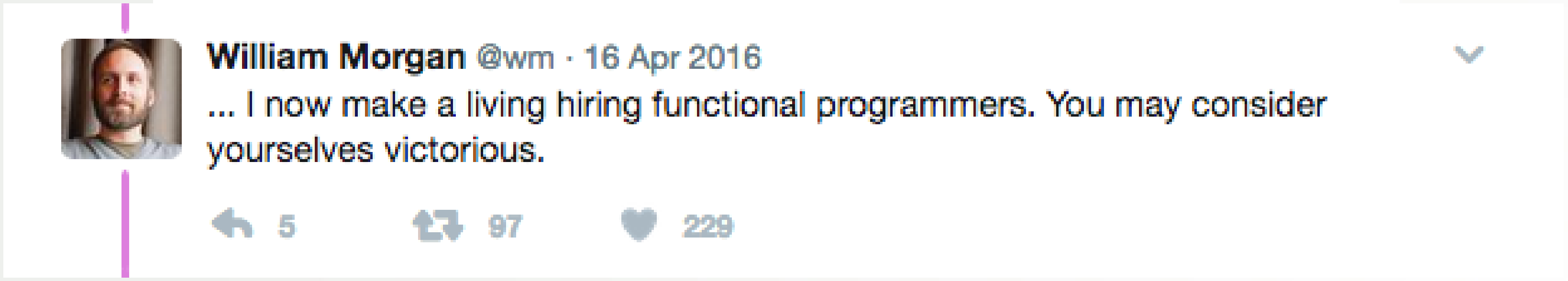


William Morgan
@wm

i love functional programming. it takes smart people who would otherwise be competing with me and turns them into unemployable crazies

RETWEETS 2,119 LIKES 2,179

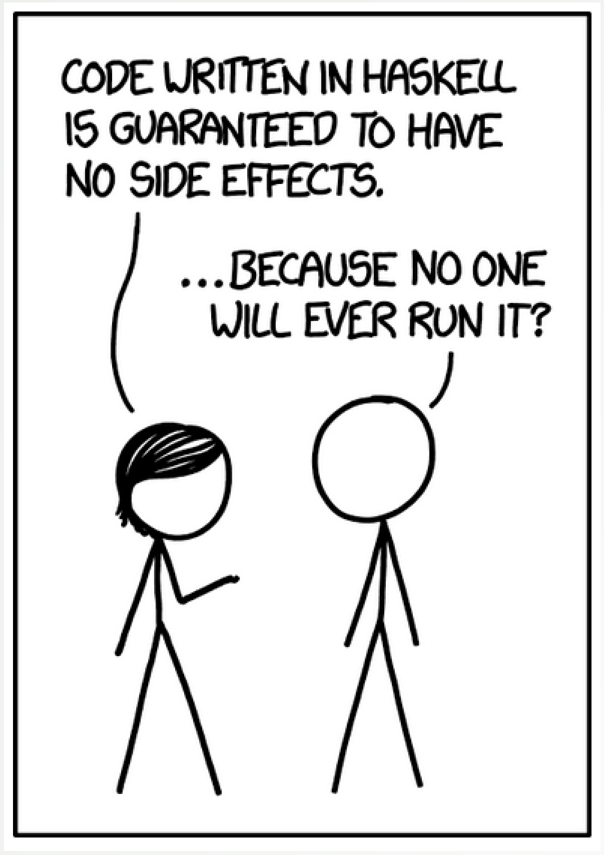
11:53 AM - 30 Dec 2009



William Morgan @wm · 16 Apr 2016
... I now make a living hiring functional programmers. You may consider yourselves victorious.

5 97 229

## Functional Programming Hate


CODE WRITTEN IN HASKELL IS GUARANTEED TO HAVE NO SIDE EFFECTS.
...BECAUSE NO ONE WILL EVER RUN IT?

## Functional Programming Languages

- Common Lisp
- Scheme
- Clojure
- Scala
- Mathematica
- Rust
- Erlang
- OCaml
- Haskell
- F#

- Popular Languages with Functional attributes:
  - C++11(/14/17/20)
  - Java8
  - Perl6
  - Python
  - Javascript

## Functional Programming Users

- AT&T – Haskell
- Akamai – Clojure
- Bank of America Merrill Lynch – Haskell
- Twitter – Scala
- Facebook – Haskell
- Emacs – Lisp
- DeTeXify – Haskell

# So What is Functional Programming???

*“...A programming paradigm that treats computation as the evaluation of* ***mathematical functions*** *and avoids changing-state and* ***mutable data****. It is a* **declarative programming** *paradigm, which means programming is done with expressions or declarations instead of statements.”* - Wikipedia

## So What is
## Functional Programming???

# Computability Theory

Alan Turing
(1912 – 1954)

Alonzo Church
(1903-1995)

Turing Machine

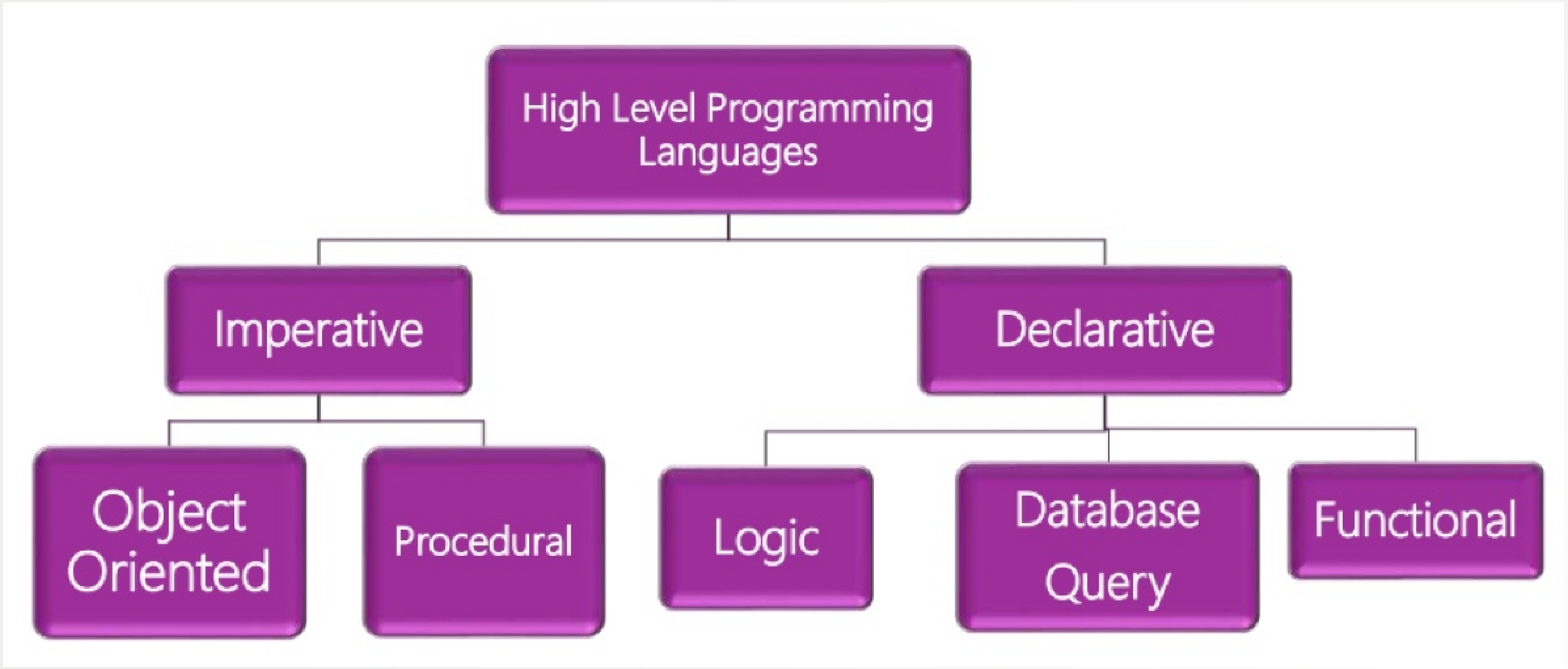
Lambda calculus

Two mathematical ways to ask questions about
“computability”

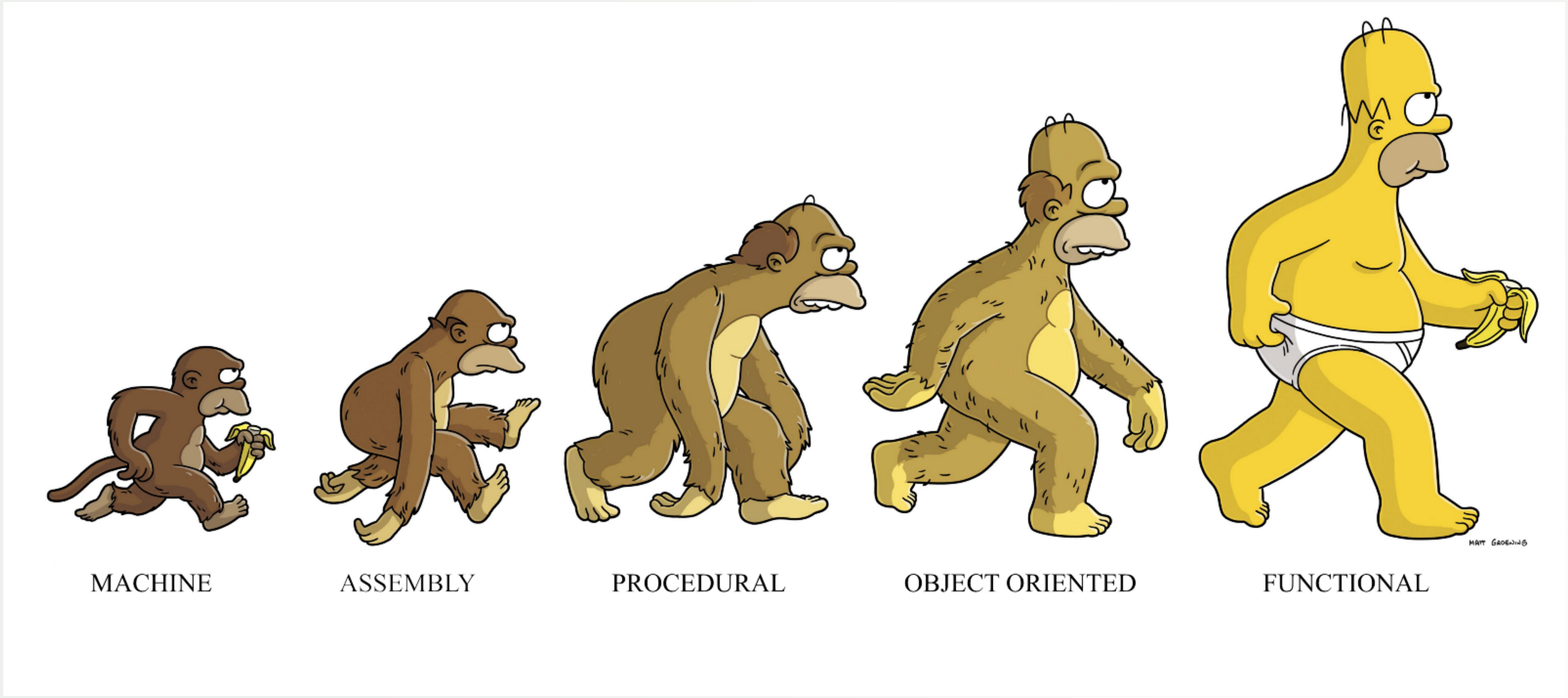
## No Really, What is Functional Programming?

- Declarative Programming
- Mathematical Functions
  - Pure, First Class, Higher Order
- Immutable State
- No Side Effects
- Strong Static Types
- Everything is a Function

# Hierarchy of Programming Languages



## Functional programmers see it differently



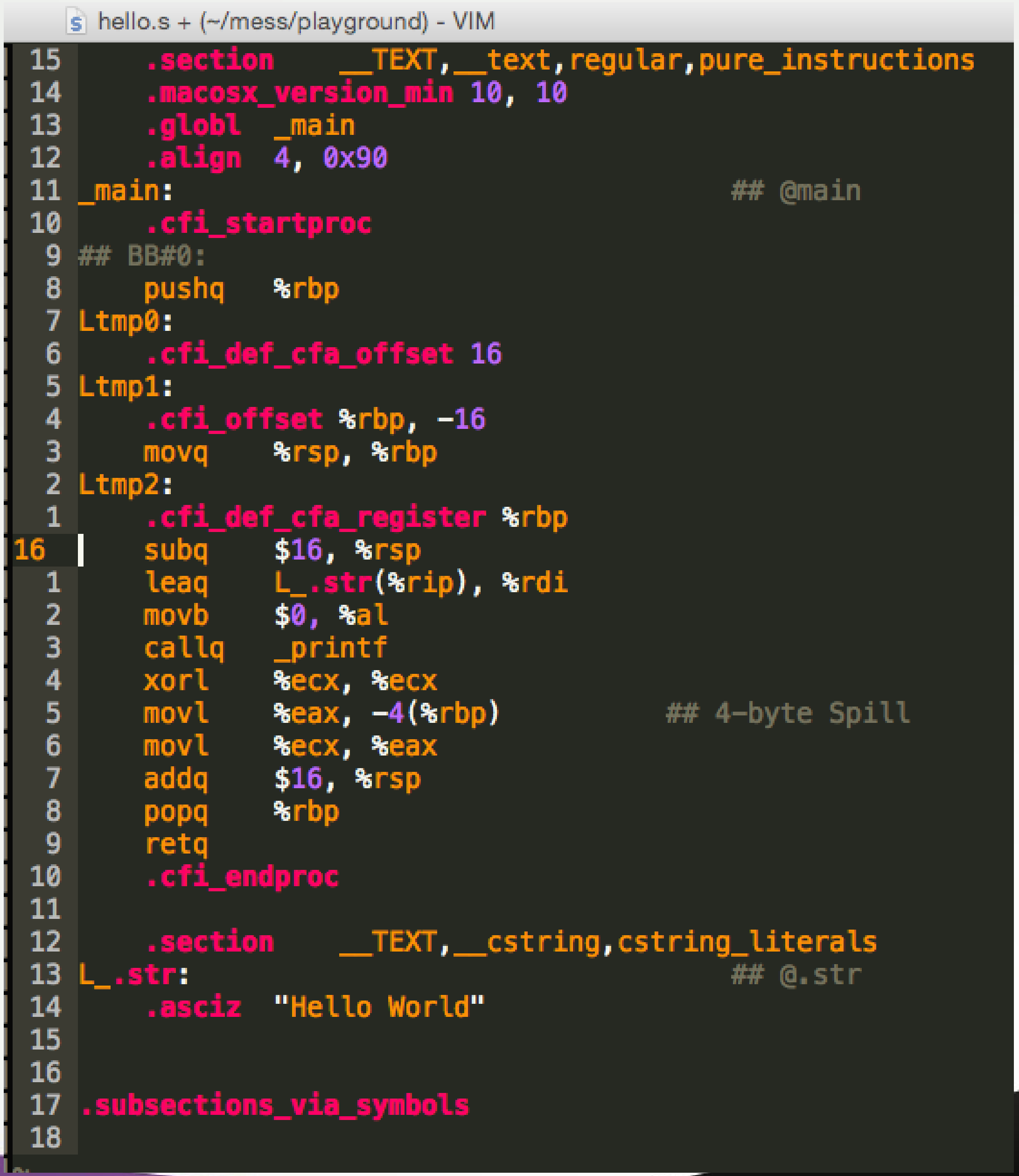
## Imperative vs Declarative

- Building blocks
  - Instructions
  - Statements
  - Control Loops
- Code is executed line-by-line, defined as a sequence of commands
- Program defines **HOW** to get result.

- Building blocks
  - Functions
  - Expressions
  - Recursion
- Code is executed by evaluating collections of functions on inputs
- Program defines **WHAT** result to get.

# Assembly Programming

- Imperative Style
- Direct mapping with machine code
- Has some ‘abstractions’ like functions (callq)
- This is “Hello World” assembled from C →
- At least it’s not pure pure Hexadecimal

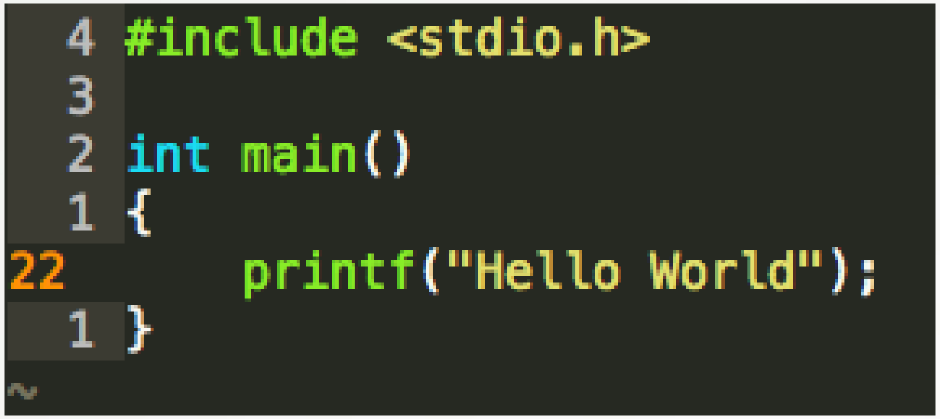


hello.s + (~/mess/playground) - VIM

```
	.section	__TEXT,__text,regular,pure_instructions
	.macosx_version_min 10, 10
	.globl	_main
	.align	4, 0x90
_main:                                  ## @main
	.cfi_startproc
## BB#0:
	pushq	%rbp
Ltmp0:
	.cfi_def_cfa_offset 16
Ltmp1:
	.cfi_offset %rbp, -16
	movq	%rsp, %rbp
Ltmp2:
	.cfi_def_cfa_register %rbp
	subq	$16, %rsp
	leaq	L_.str(%rip), %rdi
	movb	$0, %al
	callq	_printf
	xorl	%ecx, %ecx
	movl	%eax, -4(%rbp)          ## 4-byte Spill
	movl	%ecx, %eax
	addq	$16, %rsp
	popq	%rbp
	retq
	.cfi_endproc

	.section	__TEXT,__cstring,cstring_literals
L_.str:                                 ## @.str
	.asciz	"Hello World"


.subsections_via_symbols

```

## Procedural Programming

- Imperative Style
- Introduces new abstractions
  - Loops
- This is “Hello World” C source code →
- Why? Easier to reason about, avoids GOTOs.



```
#include <stdio.h>

int main()
{
        printf("Hello World");
}
~
```

## Object Oriented Programming

- Imperative style
- Introduces new abstractions
  - Objects
  - Scope
  - Classes
    - Private Data and Methods Together
- Class Inheritance
  - Polymorphism
    - Generics
- Why? Avoid shared data / global state. Strive toward Generic programming

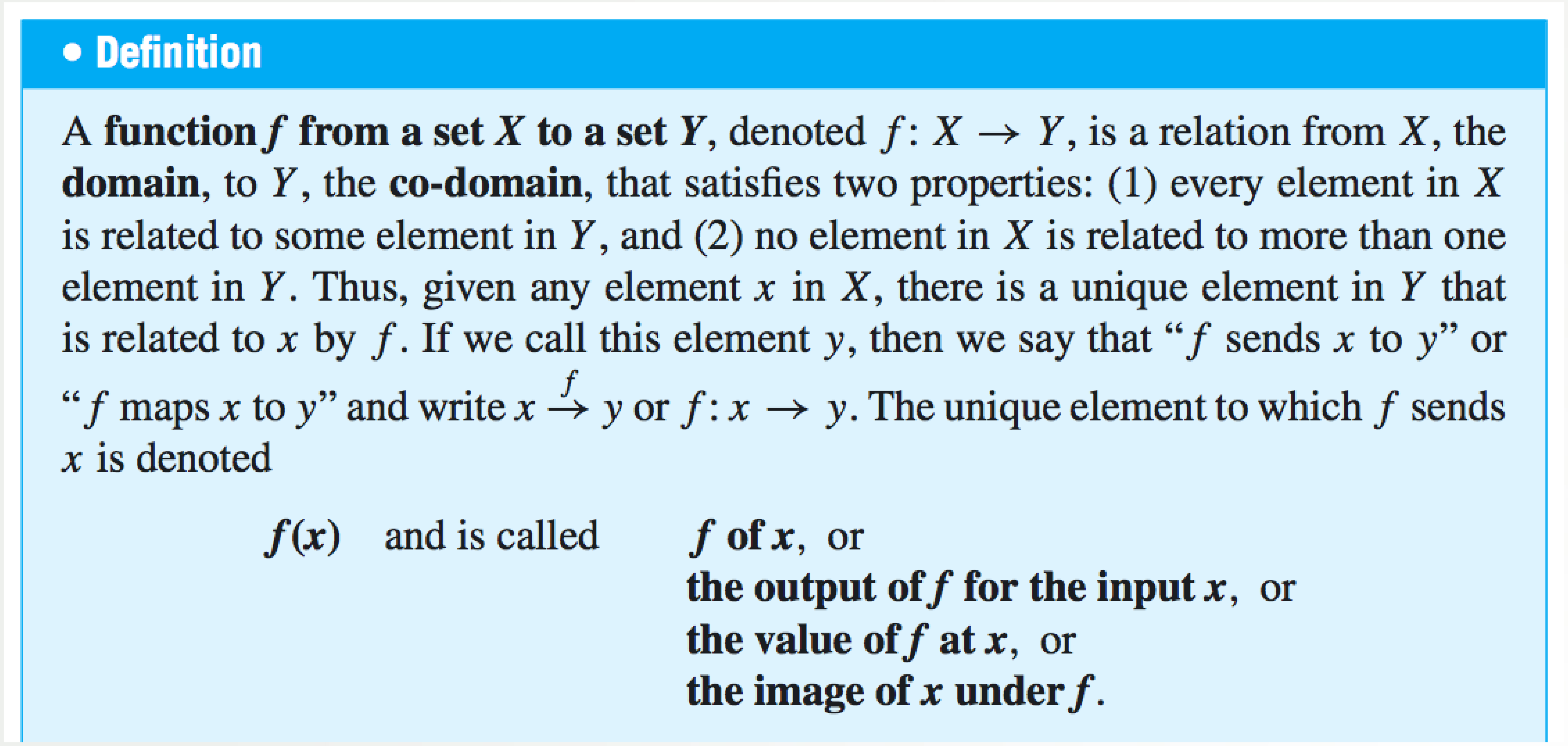
- *“Object Oriented Programming is an exceptionally bad idea that can only have originated in California”*
  - Edsger Dijkstra
- *“You probably know that arrogance, in computer science, is measured in nano-Dijkstras”*
  - Alan Kay

## Functional Programming

- Gave us Pure, First class, Higher Order Functions
- Allows total statelessness
  - No Side Effects
- Truly Generic programming
  - Define functions that can operate on  any type
    - Int
    - String
    - List
    - Other Functions
    - Itself

# What is a function?

- Stolen from Mathematics. A function is:



● **Definition**

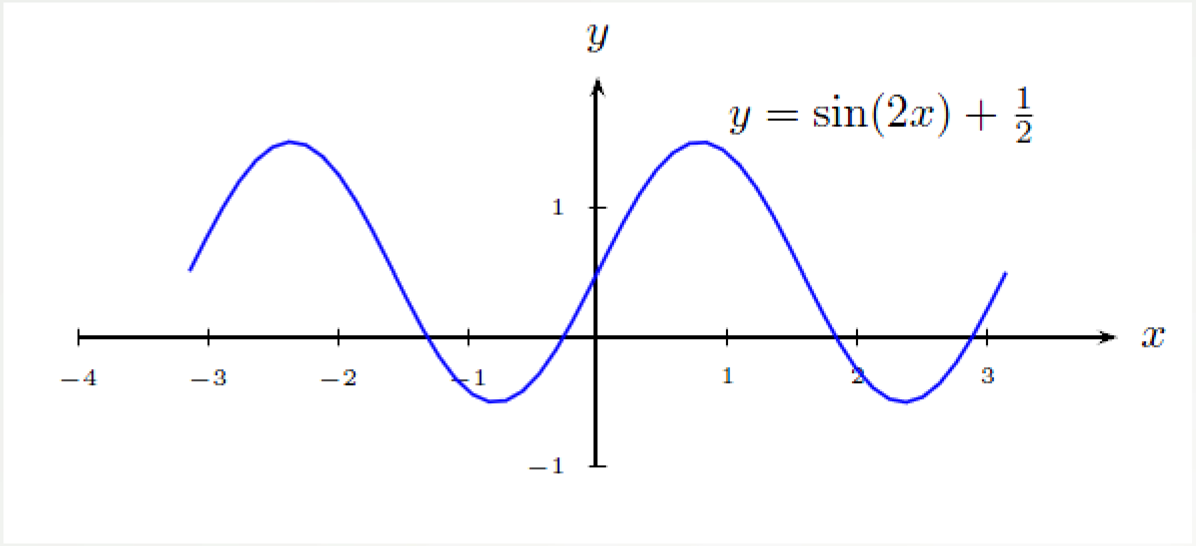
A **function $f$ from a set $X$ to a set $Y$**, denoted $f\colon X \to Y$, is a relation from $X$, the **domain**, to $Y$, the **co-domain**, that satisfies two properties: (1) every element in $X$ is related to some element in $Y$, and (2) no element in $X$ is related to more than one element in $Y$. Thus, given any element $x$ in $X$, there is a unique element in $Y$ that is related to $x$ by $f$. If we call this element $y$, then we say that "$f$ sends $x$ to $y$" or "$f$ maps $x$ to $y$" and write $x \xrightarrow{f} y$ or $f\colon x \to y$. The unique element to which $f$ sends $x$ is denoted

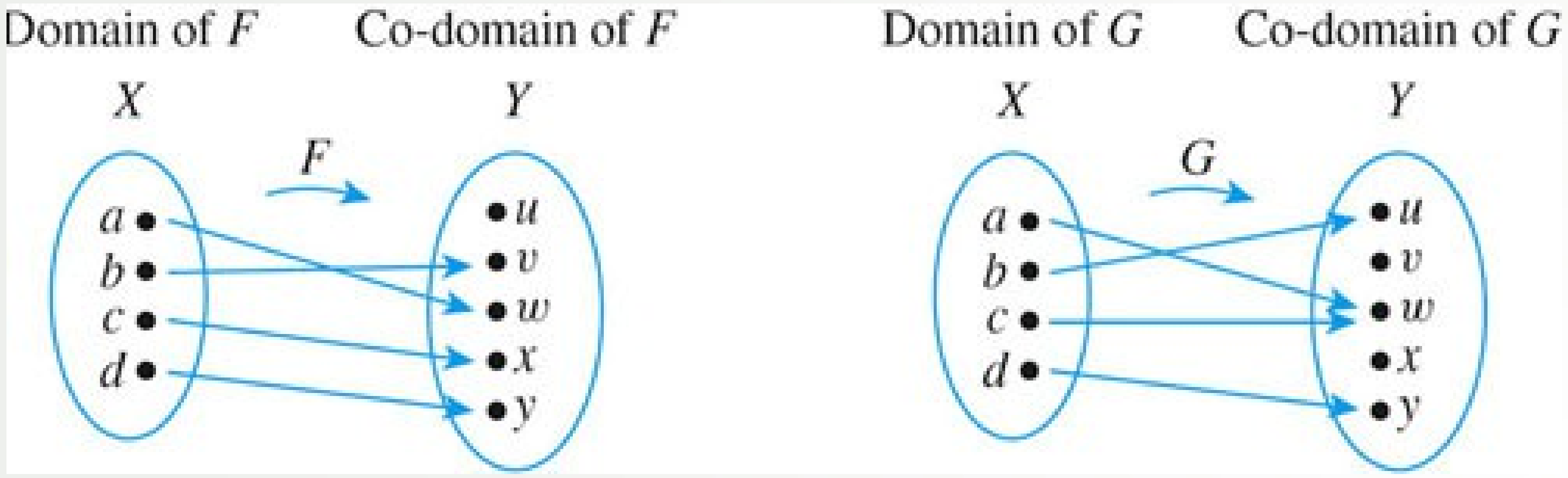
$f(x)$ and is called

**$f$ of $x$**, or
**the output of $f$ for the input $x$**, or
**the value of $f$ at $x$**, or
**the image of $x$ under $f$**.

## What is a function?

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ -x & \text{if } x < 0 \end{cases}$$





## What is a function?

- Stolen from Mathematics. A function is a total mapping from X to Y.
- No requirements on what X and Y must be
- No function memory
  - Idempotent
    - Functions always return the same output on the same input. Past inputs do not change the result of a function.
- No Side Effects

## Functional Programming Jargon

- Referential Transparency
- Pure Functions
- First Class Functions
- Higher Order Functions
- Immutability

## Referential Transparency

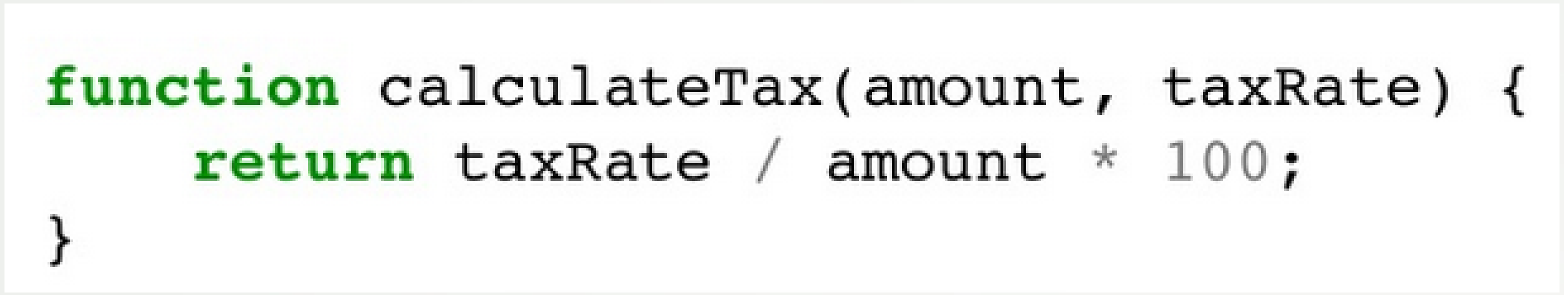
- Any function may be substituted with it’s result
  - If y=f(x) all f(x) may be replaced with y
- Why?
  - Optimization
  - Lazy Evaluation
  - Let compiler decide when to evaluate f(x)

## Pure Functions

- Behave like Mathematical Functions
- Result depends only on inputs
- Same inputs --> same outputs ALWAYS
- No Side Effects.
  - Does Not Change state of the machine somewhere else
  - Avoids Mystery bugs
  - Great for Concurrency

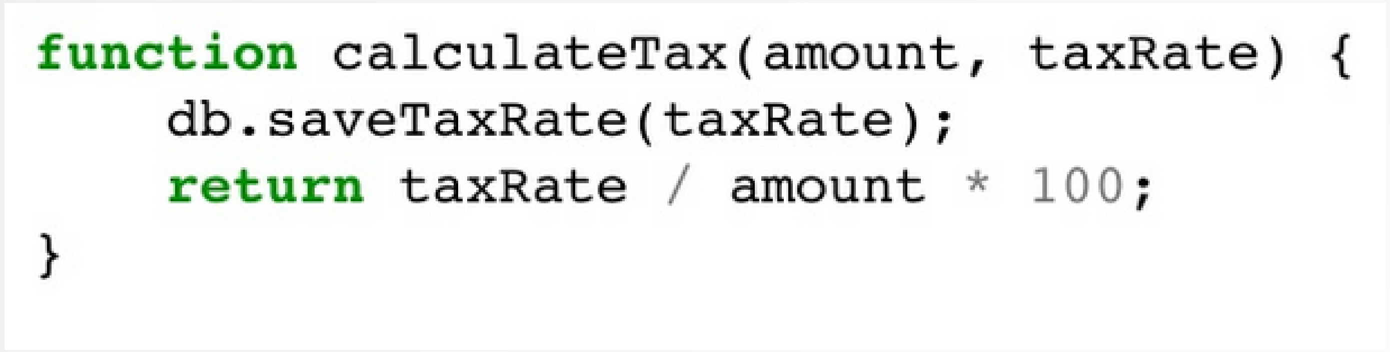
## Pure Functions

- Pure

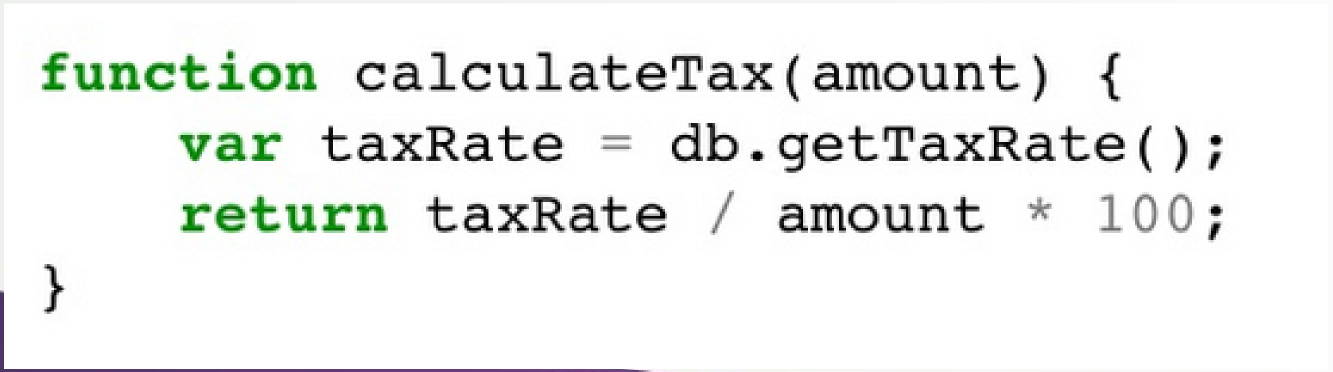


```
function calculateTax(amount, taxRate) {
    return taxRate / amount * 100;
}
```

- Not Pure



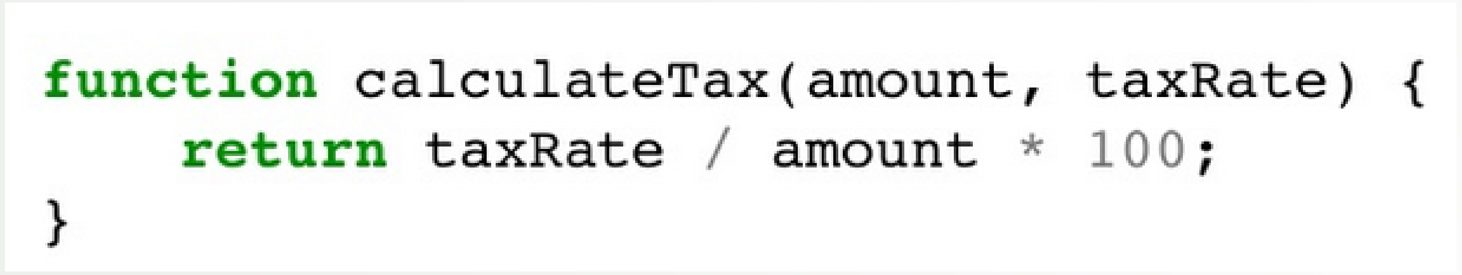
```
function calculateTax(amount, taxRate) {
    db.saveTaxRate(taxRate);
    return taxRate / amount * 100;
}
```



```
function calculateTax(amount) {
    var taxRate = db.getTaxRate();
    return taxRate / amount * 100;
}
```

## First Class Functions

- Functions are objects just like variables, literals, primitive types
- Functions have types, just like variables
- Function type is (input) → (output)



```
function calculateTax(amount, taxRate) {
    return taxRate / amount * 100;
}
```

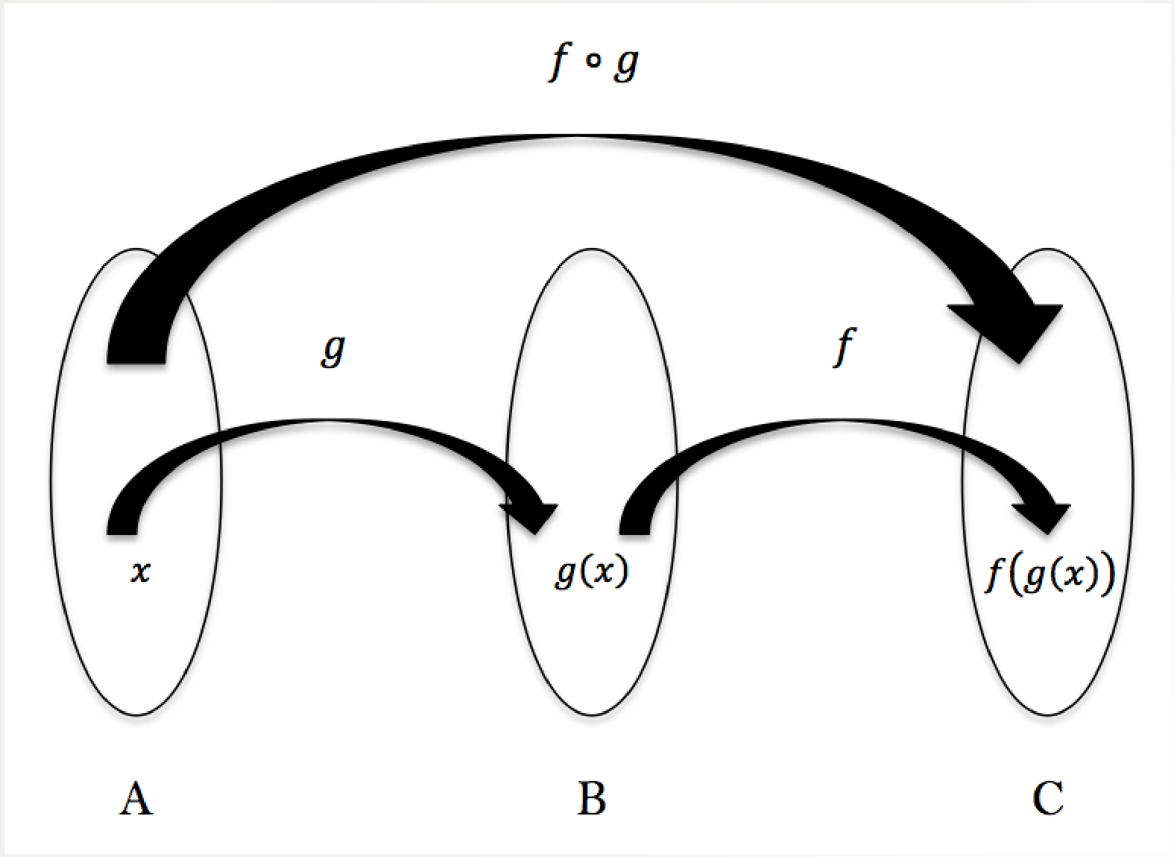
- Type: (int, int) → int
- ‘calculateTax’ can be passed around and reasoned about

## Higher Order Functions

- Functions can be <u>inputs</u>
- Functions can be <u>outputs</u>
- Derivative function takes a function and returns a function
  - Df/dx = derivative(f, x)

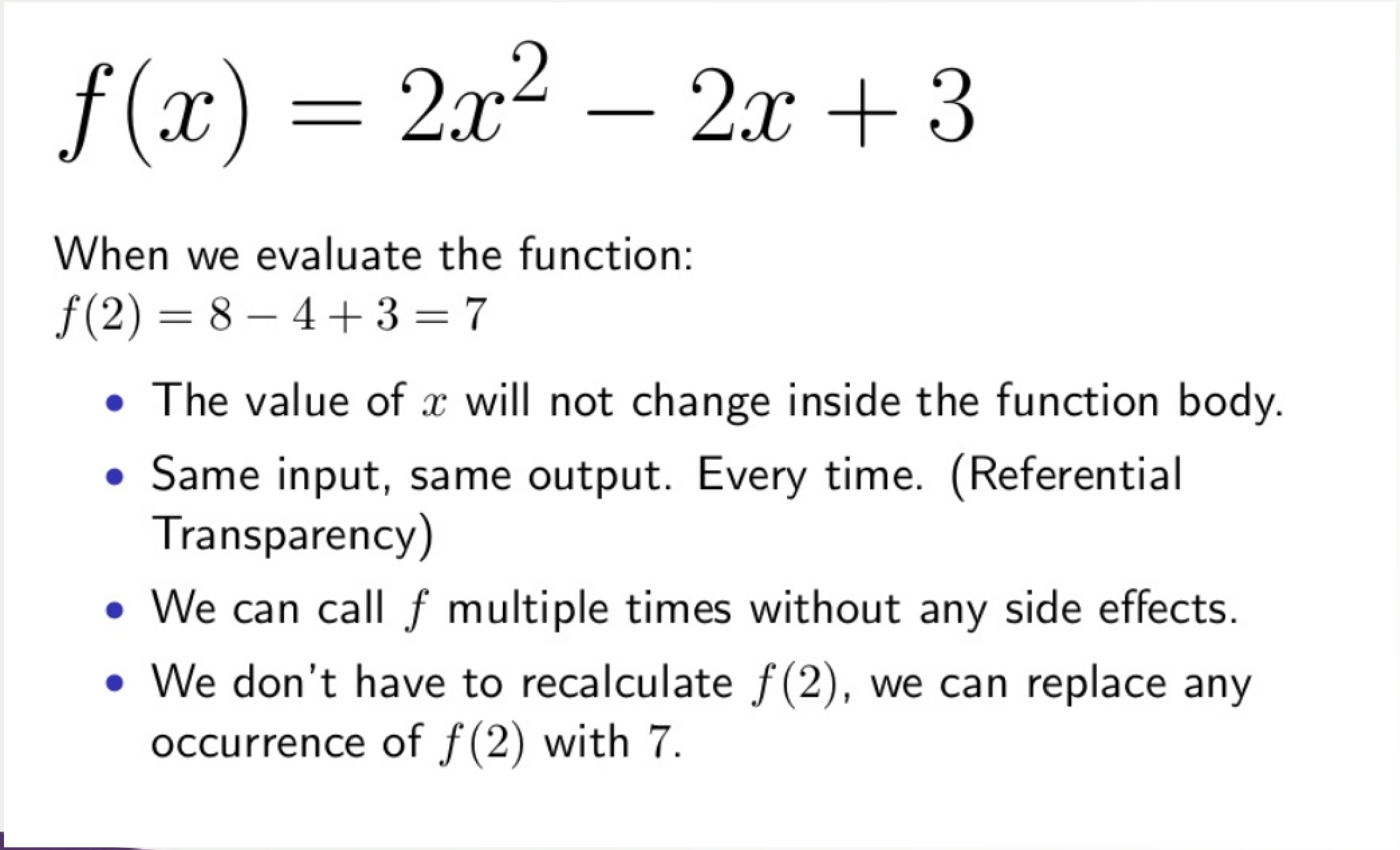
# Higher Order Functions

- Functions are compose-able



## Functions and Immutability

- Variables are constant, they cannot be mutated



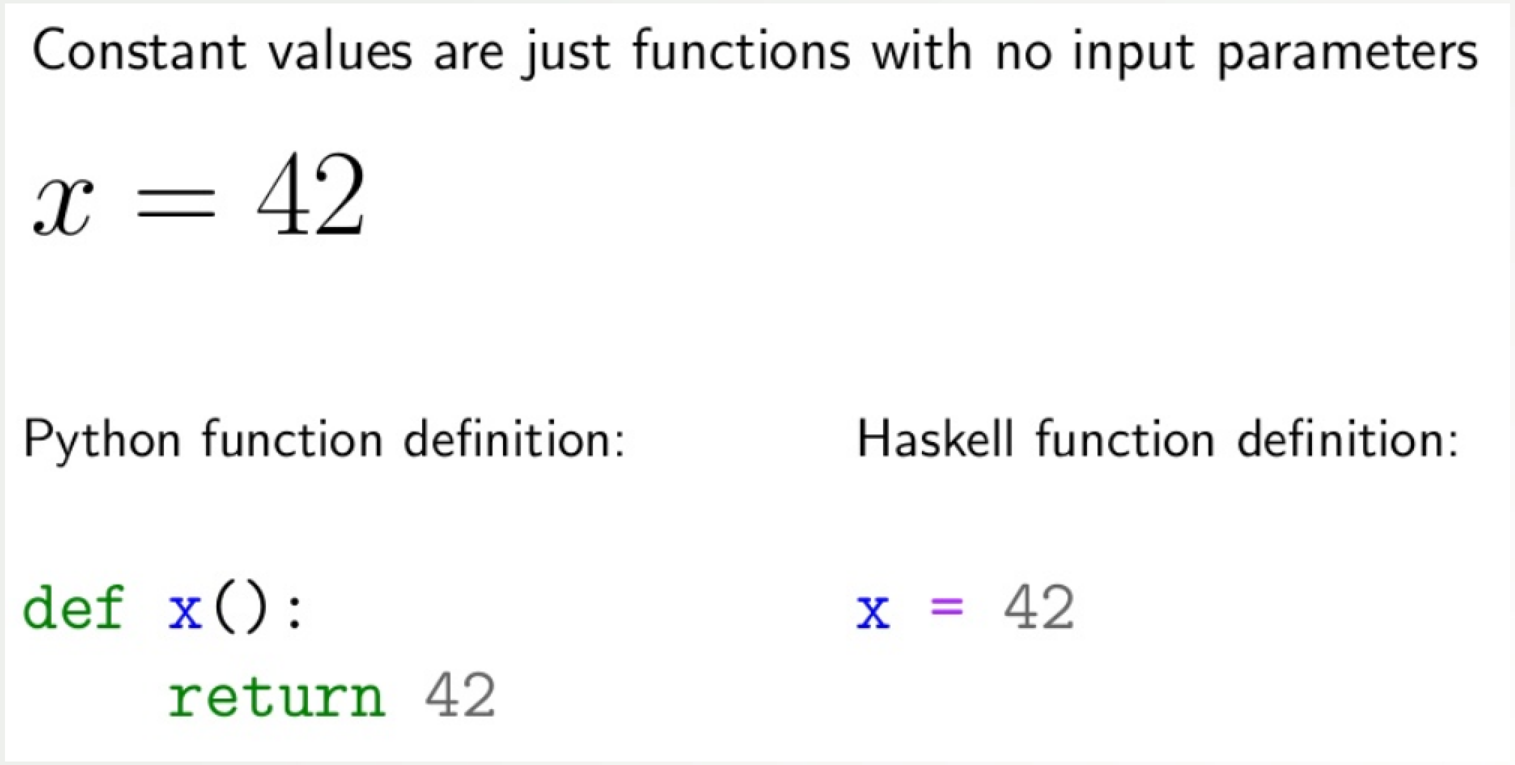
$$f(x) = 2x^2 - 2x + 3$$

When we evaluate the function:
$f(2) = 8 - 4 + 3 = 7$

- The value of $x$ will not change inside the function body.
- Same input, same output. Every time. (Referential Transparency)
- We can call $f$ multiple times without any side effects.
- We don't have to recalculate $f(2)$, we can replace any occurrence of $f(2)$ with 7.

# Functions and Immutability

- Variables are functions too.



Constant values are just functions with no input parameters

$$x = 42$$

Python function definition:

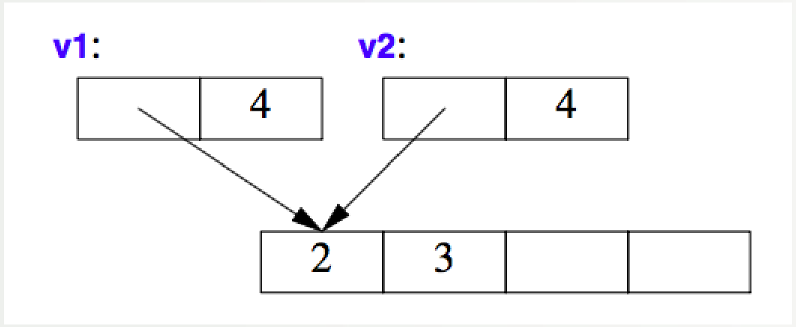
```python
def x():
    return 42
```

Haskell function definition:

```haskell
x = 42
```

## Functions and Immutability – WHY?

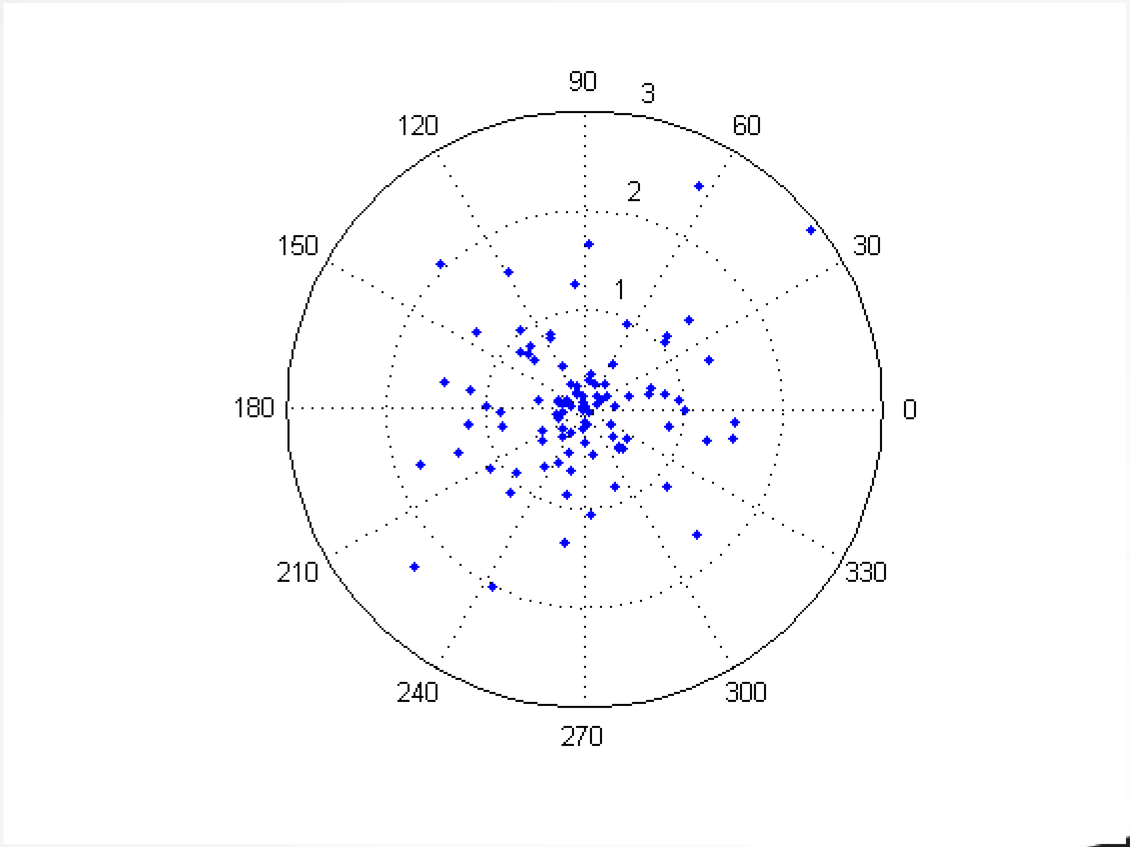
- Mutating shared objects can lead to subtle errors.
- Consider two variables that share a pointer to the same place in memory.



- *The true constant is change. Mutation hides change. Hidden change manifests chaos. Mutation attempts to erase history, but history cannot be truly erased. When the past is not preserved, it will come back to haunt you, manifesting as bugs in the program. Therefore, the wise embrace history.*

# Higher Order Function Example

- Sorting points in cartesian plane
- What Order to define?
  - Row Major?
  - Distance From Origin?
  - Polar Coordinates?



## Higher Order Function Example

- Comparator design pattern
- Define sort(compare(a,b), list) that takes a comparison function and a list of points.
- Sort() handles the ordering, depending on how compare orders them
- Compare() handles the ordering of 2 points, returns true if a > b.
- Programmer defines different compare() for different desired orderings.

# Higher Order Function Example

- Define sort(compare(a,b), [a,b,...]) that takes a comparison function and a list of points.
- Sort() is generic. It doesn’t need to know the type of the items in the list, or the items compare handles.
- What type is compare(a,b)? (any,any)→bool
- What type is sort(___,[any])? (___,[any] ) → [any]
- Sort type is: ((any,any)→bool, [list])→[list]

## What Functions Lack

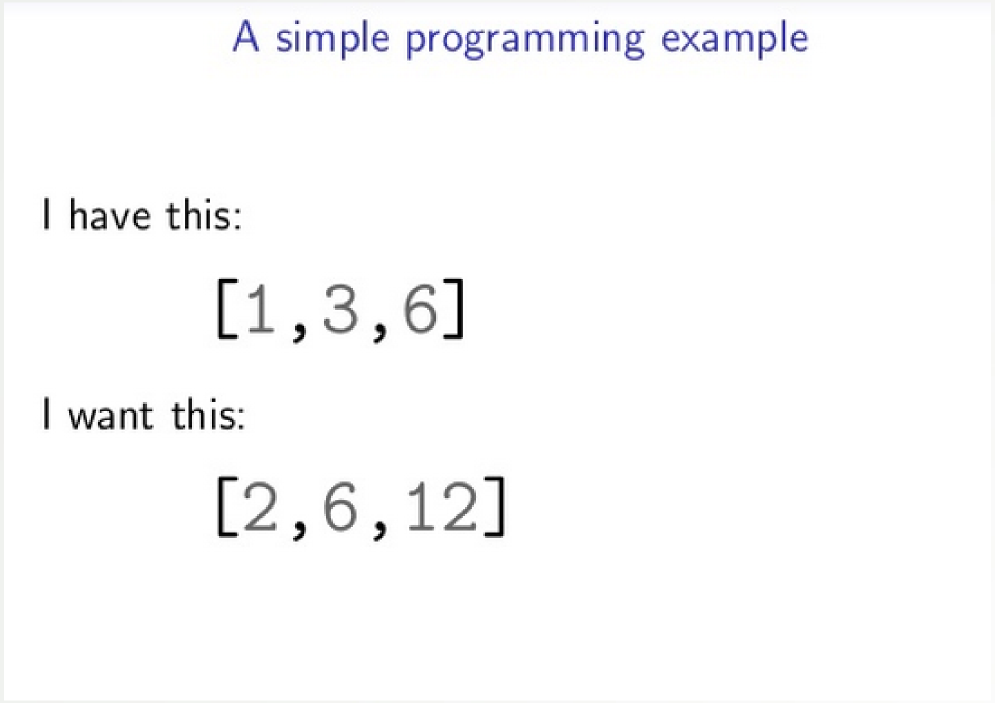
- Loops
  - For, While, Until, foreach, etc.
- Memory
  - No state
- Cannot change values
  - Immutable

## How do you program in this?

- Also lacks GOTO. You don’t need loops to declare results
- Recursion
- “Pattern Matching”
- Useful Standard Generic Functions
  - Map
  - Filter
  - Reduce,Fold,Aggregate

## Functional Programming Example

- Simple example: just double everything in a list



A simple programming example

I have this:

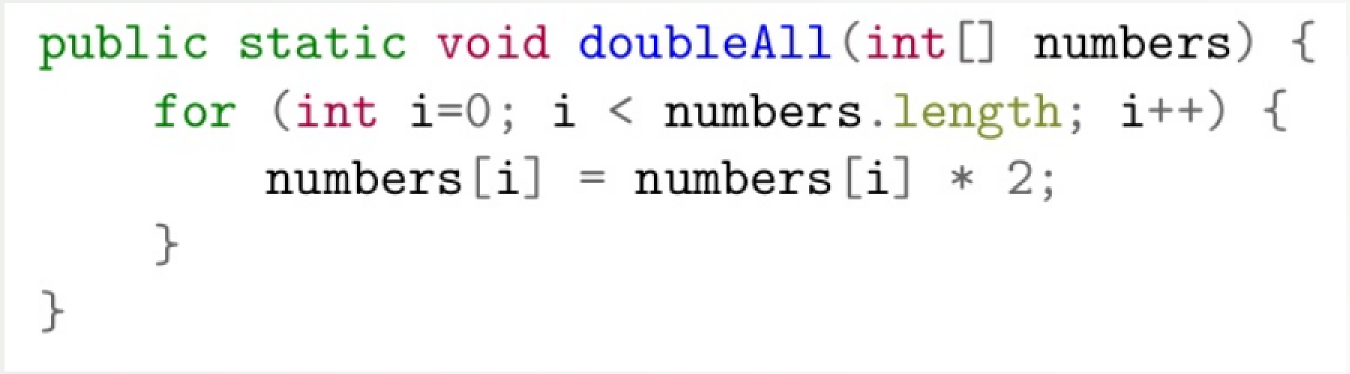
```
[1,3,6]
```

I want this:

```
[2,6,12]
```

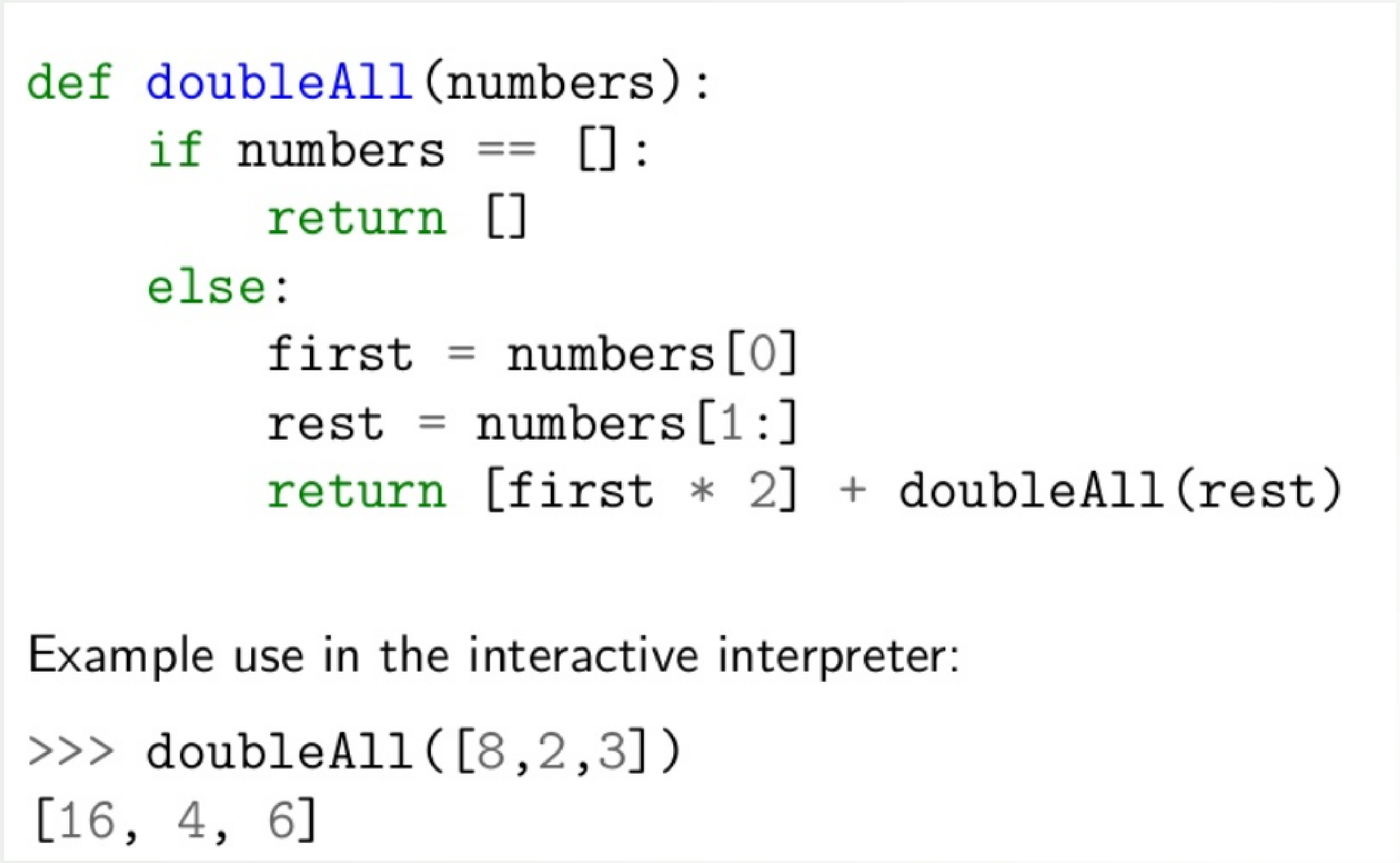
## Functional Programming Example

- Nope. Mutates both the input and the iterator index (i++), not allowed. SO FRUSTRATING



```
public static void doubleAll(int[] numbers) {
    for (int i=0; i < numbers.length; i++) {
        numbers[i] = numbers[i] * 2;
    }
}
```

# Recursive approach

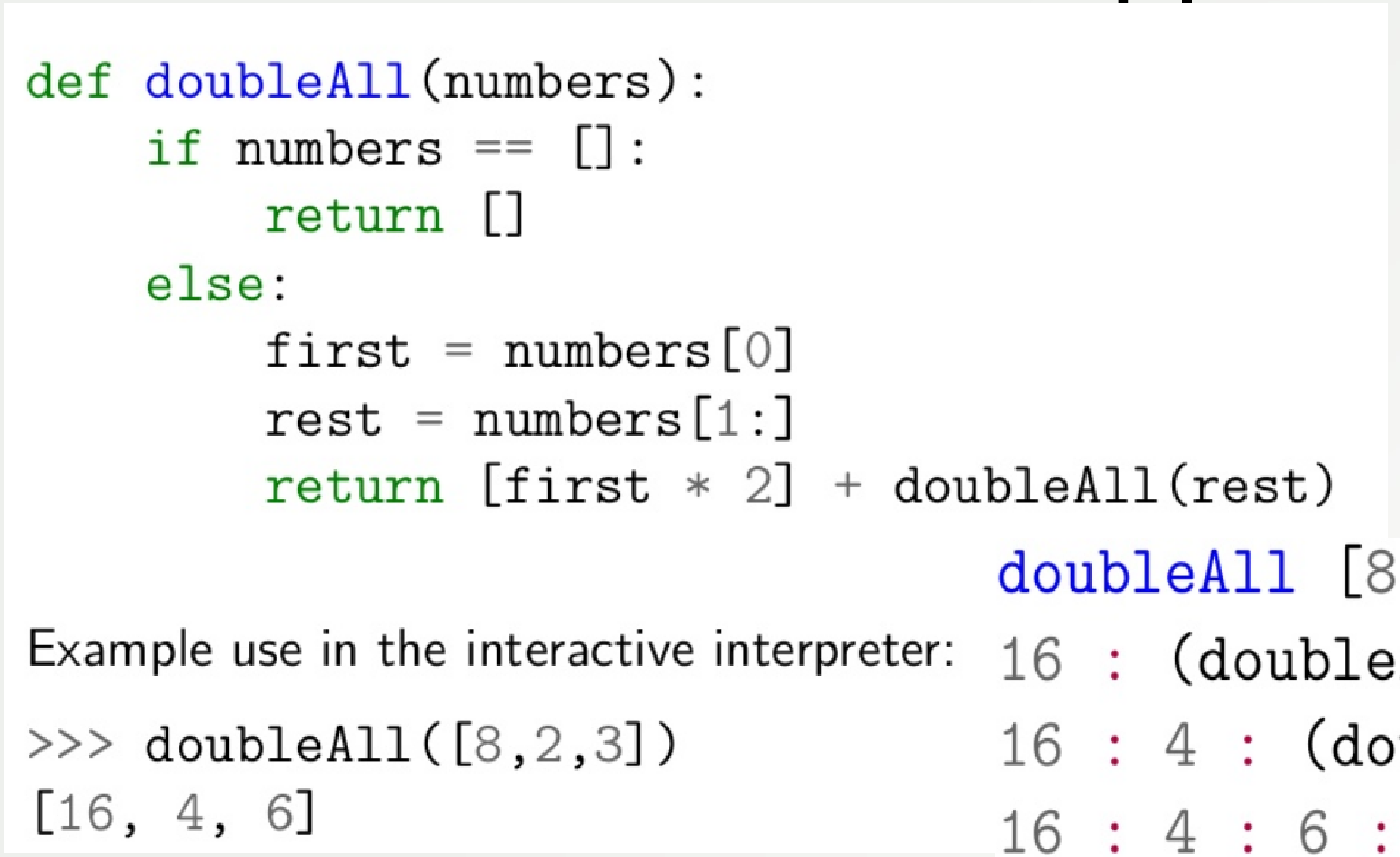


```
def doubleAll(numbers):
    if numbers == []:
        return []
    else:
        first = numbers[0]
        rest = numbers[1:]
        return [first * 2] + doubleAll(rest)
```

Example use in the interactive interpreter:

```
>>> doubleAll([8,2,3])
[16, 4, 6]
```

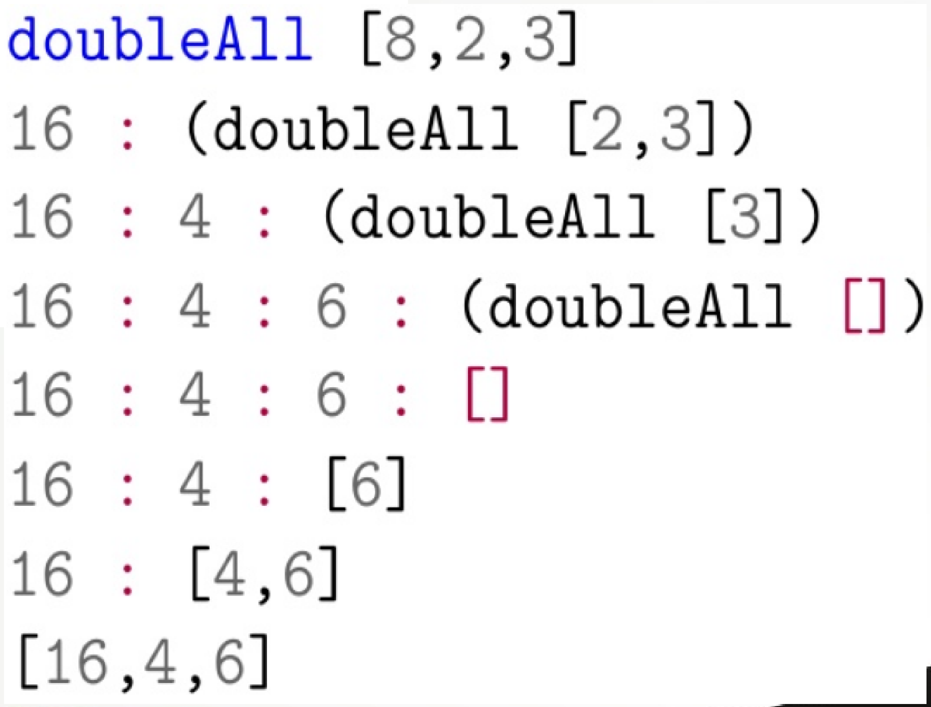
# Recursive approach



```
def doubleAll(numbers):
    if numbers == []:
        return []
    else:
        first = numbers[0]
        rest = numbers[1:]
        return [first * 2] + doubleAll(rest)
```

Example use in the interactive interpreter:

```
>>> doubleAll([8,2,3])
[16, 4, 6]
```



```
doubleAll [8,2,3]
16 : (doubleAll [2,3])
16 : 4 : (doubleAll [3])
16 : 4 : 6 : (doubleAll [])
16 : 4 : 6 : []
16 : 4 : [6]
16 : [4,6]
[16,4,6]
```

## Not your Grandmother’s Recursion

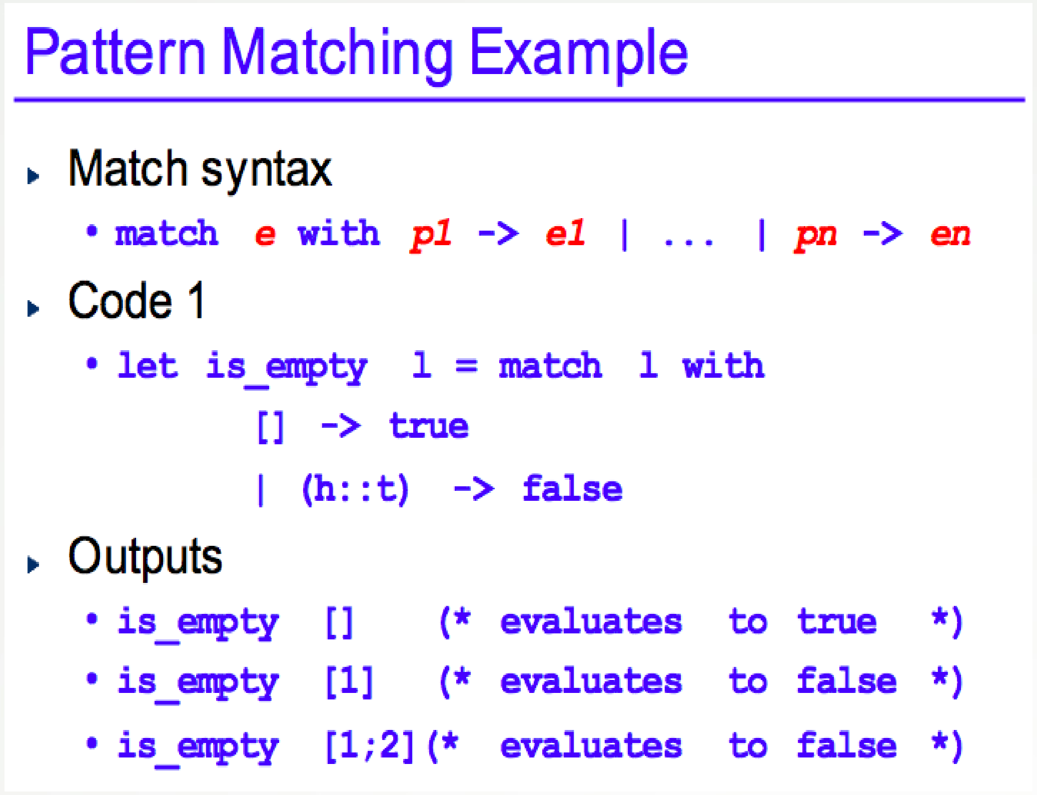
- FP compilers avoid recursive inefficiencies
  - Recursion is treated differently at Machine Code level than in C, C++ etc.
- Most Functions are inlined
- Runtime stack safety guaranteed
- Faster, Safer, Better

## Recursion’s other Benefits

- Looks like <u>Proof by Induction</u>
  - Easy to reason about
  - Easier to prove correct
- Code is smaller
  - Easier to visualize
  - Faster to write
  - Easier to maintain
- All Algorithms can be implemented recursively.

# Pattern Matching

- Glorified Case Switch
- Can reason about
  - Type
  - Structure
  - Content
- Custom Syntax
  - OCAML →



## Pattern Matching Example

- Match syntax
  - `match e with p1 -> e1 | ... | pn -> en`
- Code 1

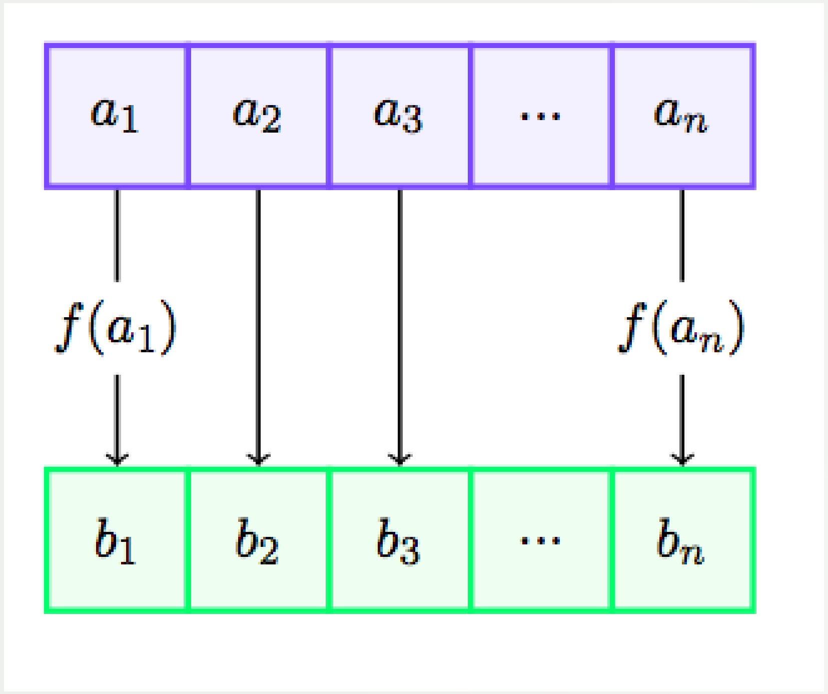
  -
    ```
    let is_empty  l = match  l with
             []  -> true
             |  (h::t)  -> false
    ```
- Outputs
  - `is_empty  []    (*  evaluates  to true   *)`
  - `is_empty  [1]   (*  evaluates  to false  *)`
  - `is_empty  [1;2] (*  evaluates  to false  *)`

## Basic Utility Functions

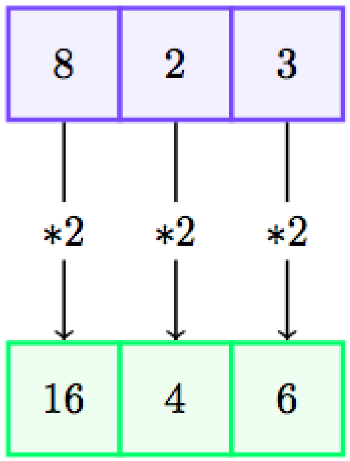
- Map
- Filter
- Fold / Reduce / Aggregate
- Lambdas

# Basic Utility Functions: Map

- Apply a function to every item in a list

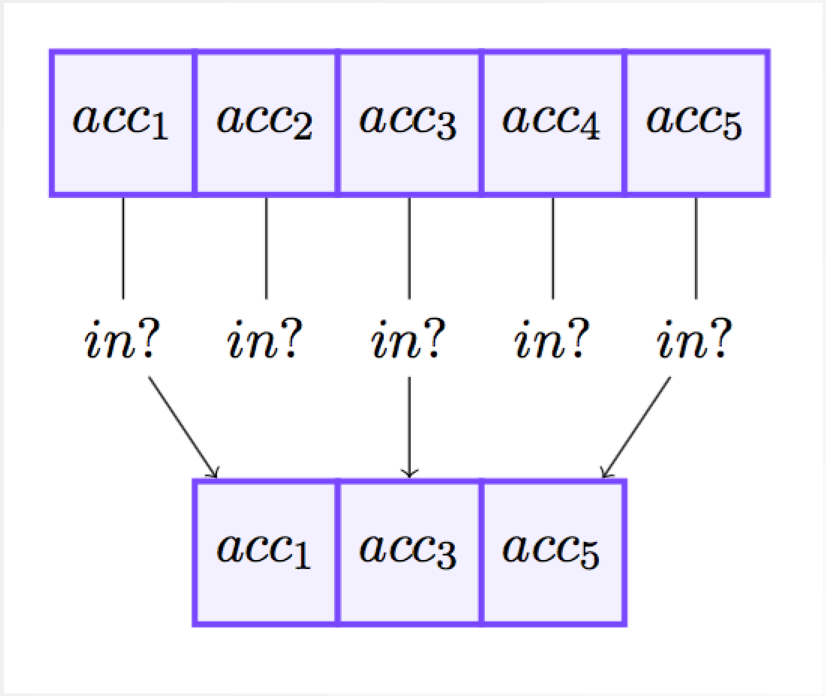


```haskell
doubleAll :: Num a => [a] -> [a]
doubleAll = map (*2)
```



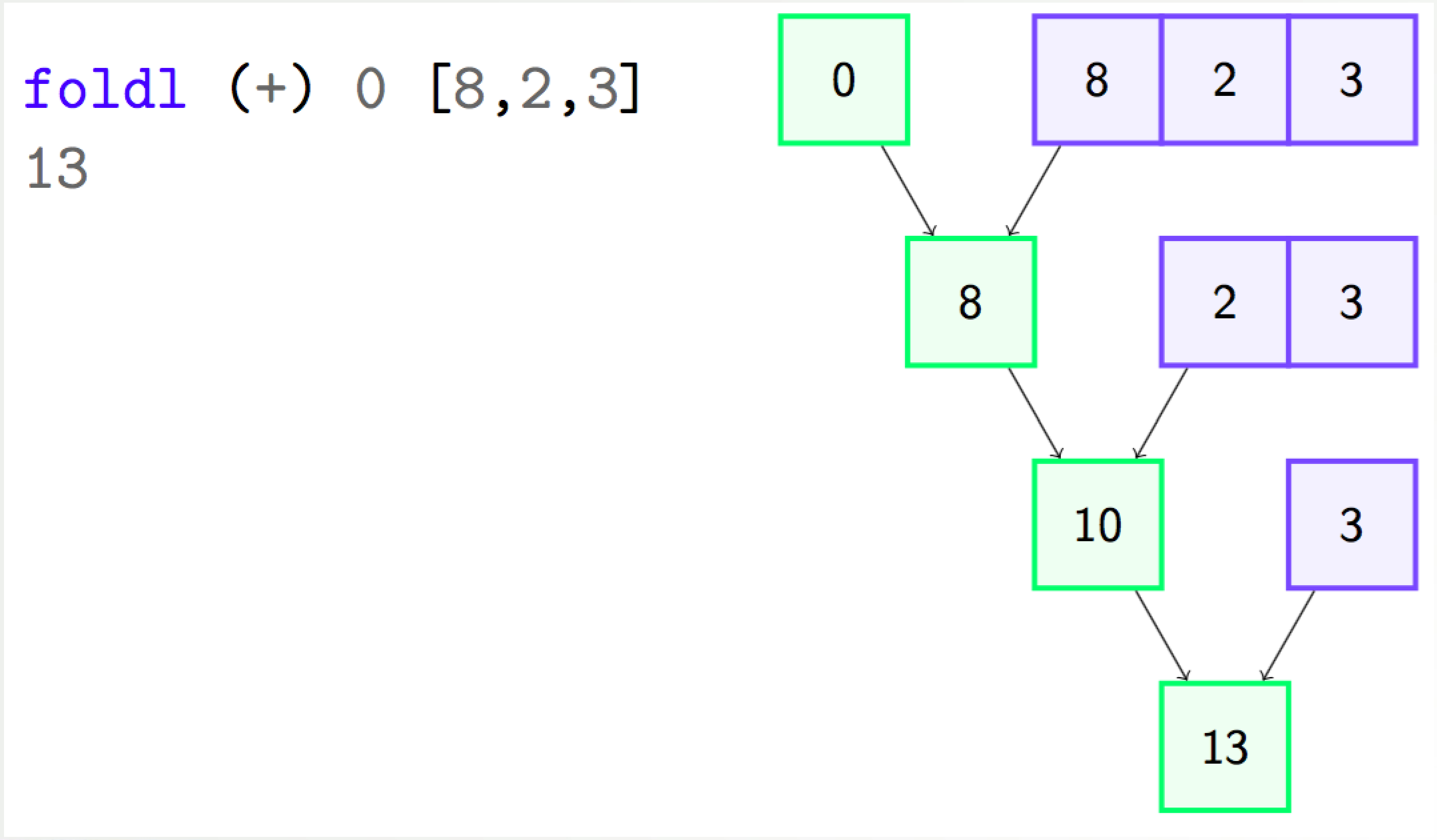
## Basic Utility Functions: Filter

- Return the items that pass a predicate



# Basic Utility Functions: Fold (Reduce)

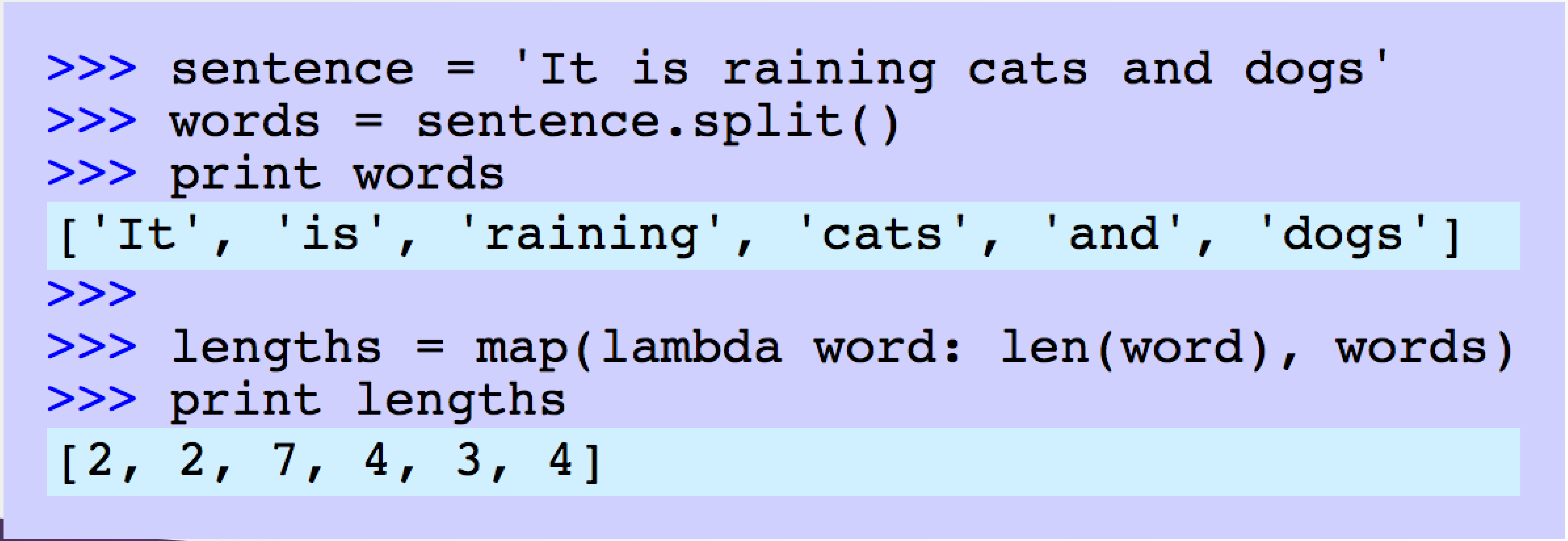
- Collapse all elements into single accumulator



```
foldl (+) 0 [8,2,3]
13
```

# Lambdas

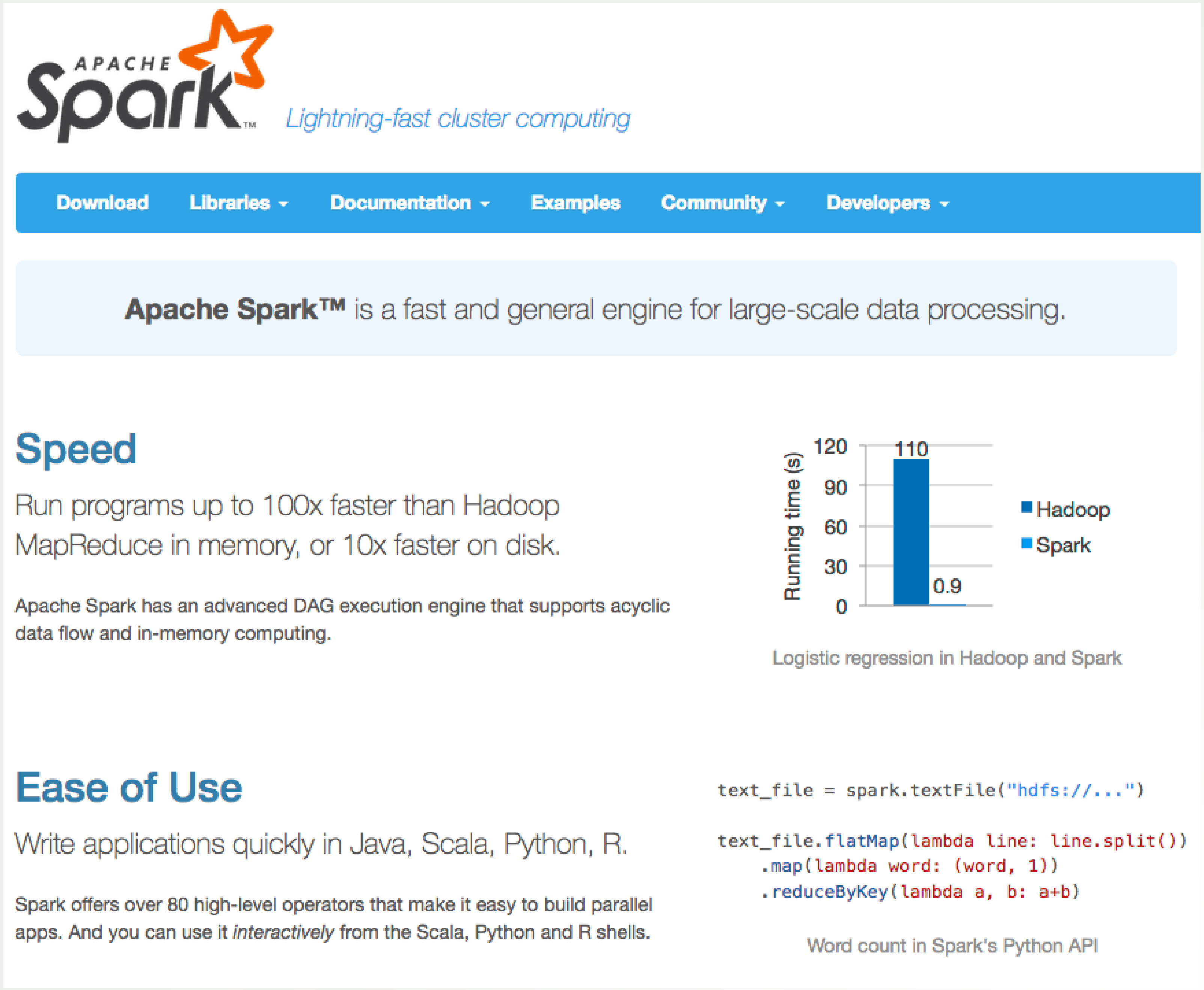
- Anonymous Functions (no callable name)
- Short throwaway when it’s not worth defining a separate function
- C++, Java, Python, Perl have all adopted



```
>>> sentence = 'It is raining cats and dogs'
>>> words = sentence.split()
>>> print words
['It', 'is', 'raining', 'cats', 'and', 'dogs']
>>>
>>> lengths = map(lambda word: len(word), words)
>>> print lengths
[2, 2, 7, 4, 3, 4]
```

# More Advanced Concepts

- Closures
  - Functions + Immutable Data
    - Similar to objects / Classes
- Currying
  - Partially filled Closures / Higher order functions
  - Don’t Pass all parameters at once
    - New interim function
- Monads
  - Chained functions



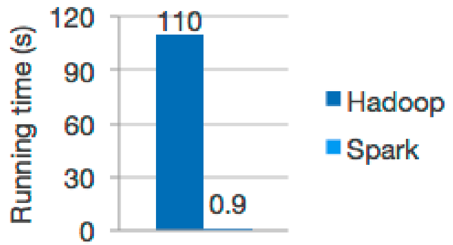
APACHE Spark™ *Lightning-fast cluster computing*

**Apache Spark™** is a fast and general engine for large-scale data processing.

## Speed

Run programs up to 100x faster than Hadoop MapReduce in memory, or 10x faster on disk.

Apache Spark has an advanced DAG execution engine that supports acyclic data flow and in-memory computing.



Logistic regression in Hadoop and Spark

## Ease of Use

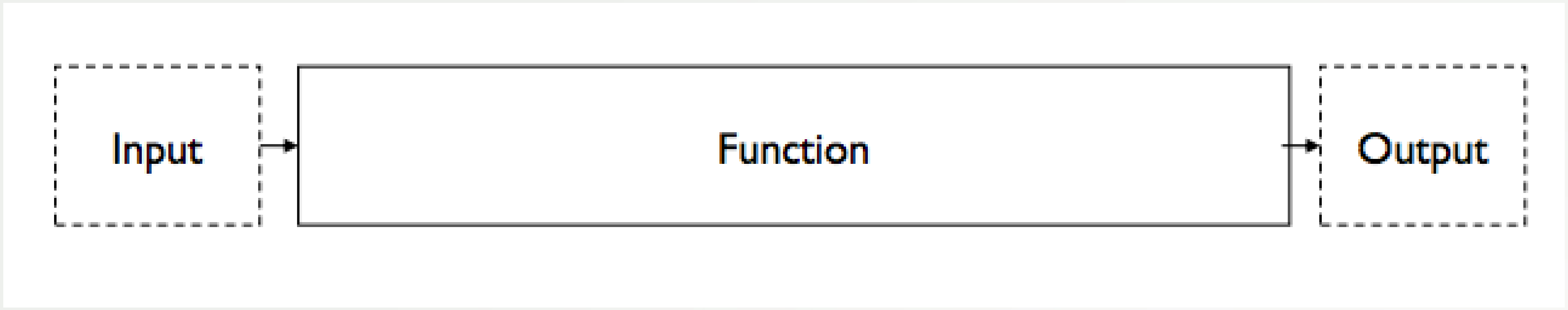
Write applications quickly in Java, Scala, Python, R.

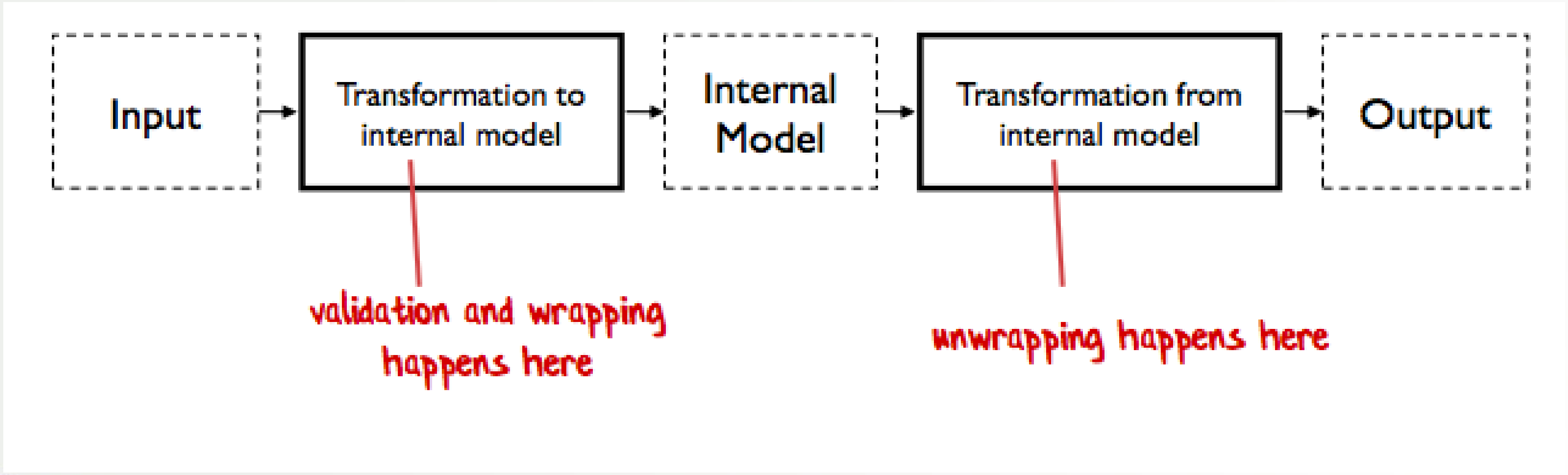
Spark offers over 80 high-level operators that make it easy to build parallel apps. And you can use it *interactively* from the Scala, Python and R shells.

```
text_file = spark.textFile("hdfs://...")

text_file.flatMap(lambda line: line.split())
    .map(lambda word: (word, 1))
    .reduceByKey(lambda a, b: a+b)
```

Word count in Spark's Python API

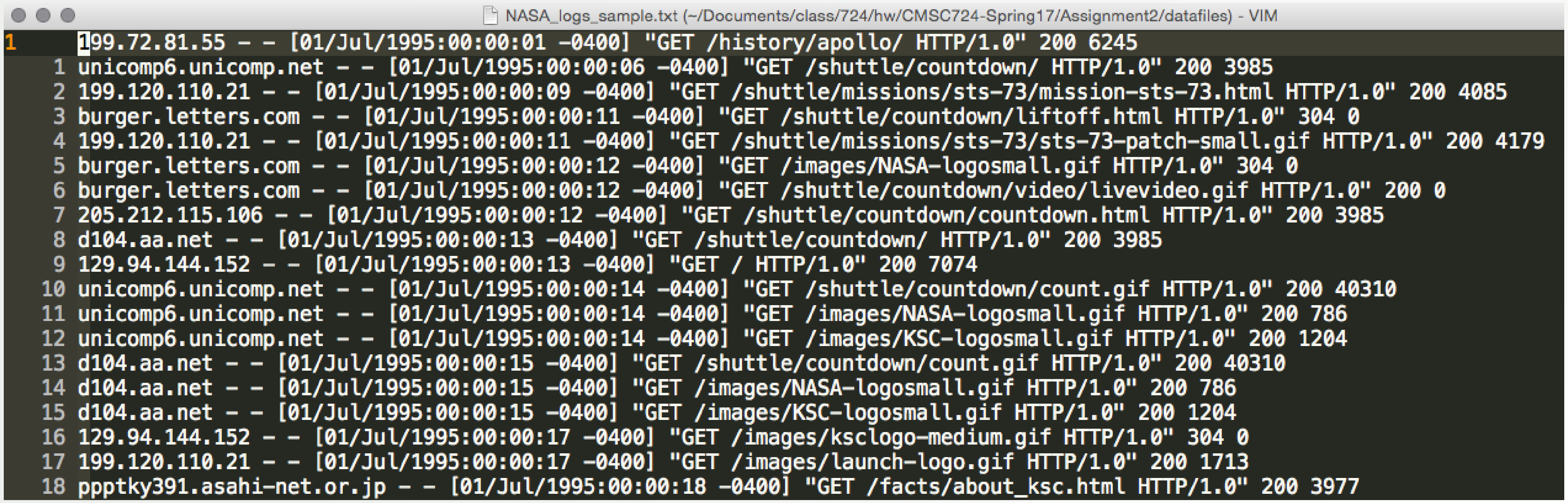
# F.P. as transformations




Input
Transformation to internal model
Internal Model
Transformation from internal model
Output
validation and wrapping happens here
unwrapping happens here

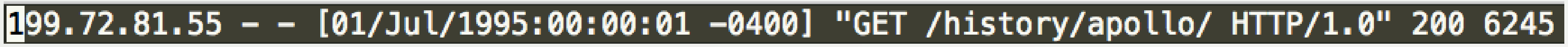
# NASA Spark Example

- Logfile of web traffic In July  1995
- Raw Text
- How many unique visitors on July 1st and 2nd

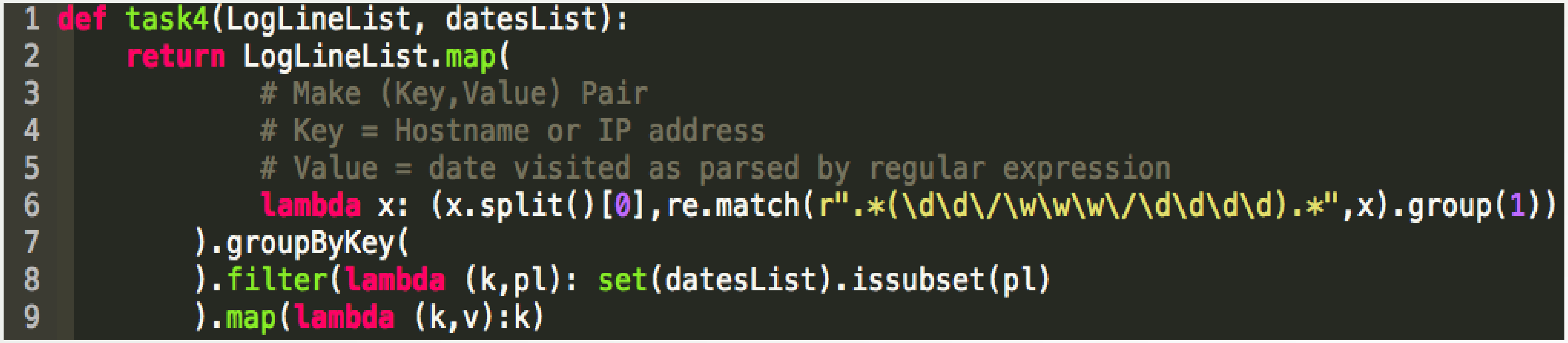

NASA_logs_sample.txt (~/Documents/class/724/hw/CMSC724-Spring17/Assignment2/datafiles) - VIM

```
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
unicomp6.unicomp.net - - [01/Jul/1995:00:00:06 -0400] "GET /shuttle/countdown/ HTTP/1.0" 200 3985
199.120.110.21 - - [01/Jul/1995:00:00:09 -0400] "GET /shuttle/missions/sts-73/mission-sts-73.html HTTP/1.0" 200 4085
burger.letters.com - - [01/Jul/1995:00:00:11 -0400] "GET /shuttle/countdown/liftoff.html HTTP/1.0" 304 0
199.120.110.21 - - [01/Jul/1995:00:00:11 -0400] "GET /shuttle/missions/sts-73/sts-73-patch-small.gif HTTP/1.0" 200 4179
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET /images/NASA-logosmall.gif HTTP/1.0" 304 0
burger.letters.com - - [01/Jul/1995:00:00:12 -0400] "GET /shuttle/countdown/video/livevideo.gif HTTP/1.0" 200 0
205.212.115.106 - - [01/Jul/1995:00:00:12 -0400] "GET /shuttle/countdown/countdown.html HTTP/1.0" 200 3985
d104.aa.net - - [01/Jul/1995:00:00:13 -0400] "GET /shuttle/countdown/ HTTP/1.0" 200 3985
129.94.144.152 - - [01/Jul/1995:00:00:13 -0400] "GET / HTTP/1.0" 200 7074
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400] "GET /shuttle/countdown/count.gif HTTP/1.0" 200 40310
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400] "GET /images/NASA-logosmall.gif HTTP/1.0" 200 786
unicomp6.unicomp.net - - [01/Jul/1995:00:00:14 -0400] "GET /images/KSC-logosmall.gif HTTP/1.0" 200 1204
d104.aa.net - - [01/Jul/1995:00:00:15 -0400] "GET /shuttle/countdown/count.gif HTTP/1.0" 200 40310
d104.aa.net - - [01/Jul/1995:00:00:15 -0400] "GET /images/NASA-logosmall.gif HTTP/1.0" 200 786
d104.aa.net - - [01/Jul/1995:00:00:15 -0400] "GET /images/KSC-logosmall.gif HTTP/1.0" 200 1204
129.94.144.152 - - [01/Jul/1995:00:00:17 -0400] "GET /images/ksclogo-medium.gif HTTP/1.0" 304 0
199.120.110.21 - - [01/Jul/1995:00:00:17 -0400] "GET /images/launch-logo.gif HTTP/1.0" 200 1713
ppptky391.asahi-net.or.jp - - [01/Jul/1995:00:00:18 -0400] "GET /facts/about_ksc.html HTTP/1.0" 200 3977
```

# NASA Spark Example

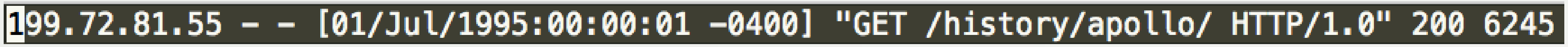


```
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
```

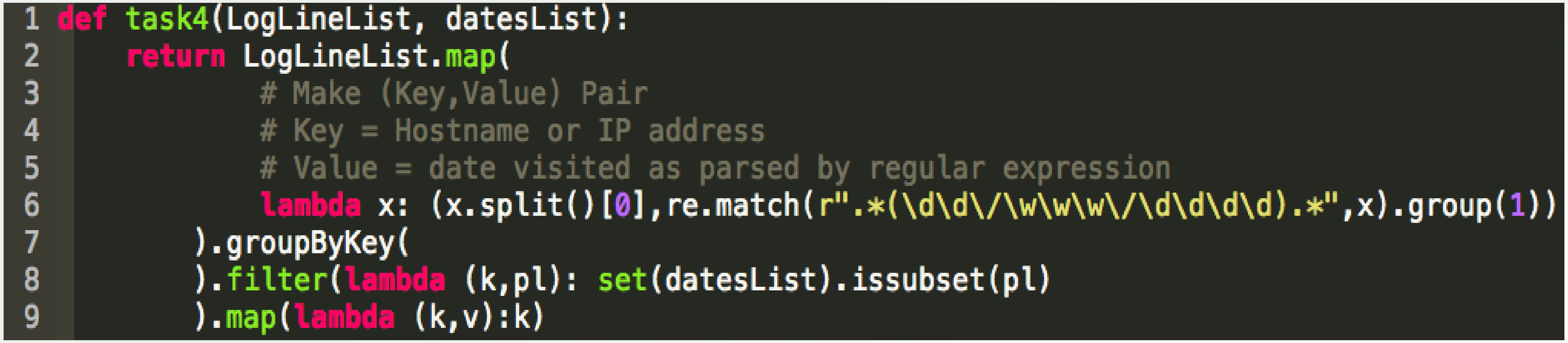


```python
def task4(LogLineList, datesList):
    return LogLineList.map(
            # Make (Key,Value) Pair
            # Key = Hostname or IP address
            # Value = date visited as parsed by regular expression
            lambda x: (x.split()[0],re.match(r".*(\d\d\/\w\w\w\/\d\d\d\d).*",x).group(1))
        ).groupByKey(
        ).filter(lambda (k,pl): set(datesList).issubset(pl)
        ).map(lambda (k,v):k)
```

# NASA Spark Example



```
199.72.81.55 - - [01/Jul/1995:00:00:01 -0400] "GET /history/apollo/ HTTP/1.0" 200 6245
```



```python
def task4(LogLineList, datesList):
    return LogLineList.map(
            # Make (Key,Value) Pair
            # Key = Hostname or IP address
            # Value = date visited as parsed by regular expression
            lambda x: (x.split()[0],re.match(r".*(\d\d\/\w\w\w\/\d\d\d\d).*",x).group(1))
        ).groupByKey(
        ).filter(lambda (k,pl): set(datesList).issubset(pl)
        ).map(lambda (k,v):k)
```

Run concurrently on hundreds or thousands of nodes

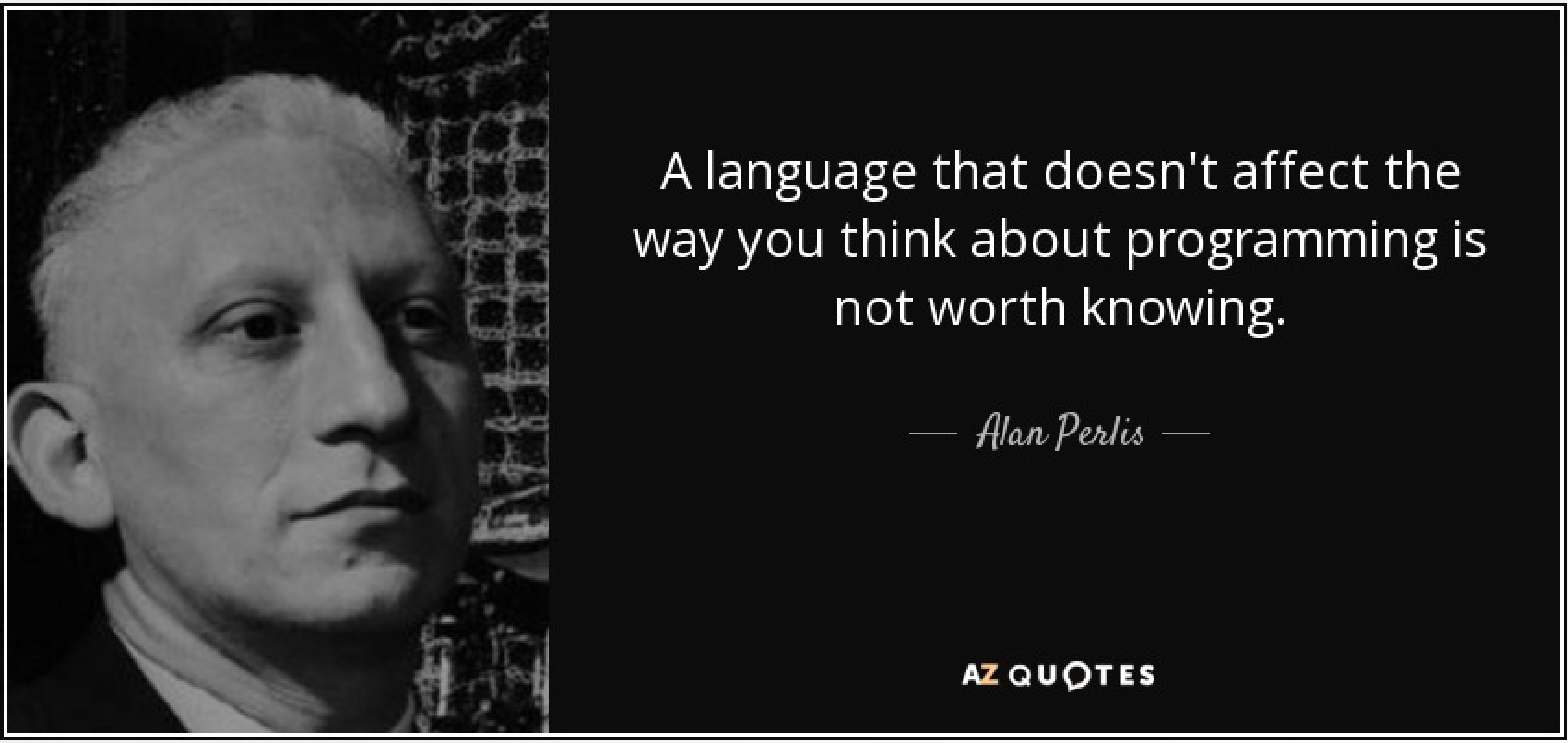
Vectorizable efficiency (SIMD commands)

Easy to debug

Is this correct?

## Functional Programming Tips

- Pure Functions over side effects.
- Write Small, generic Functions and Reuse.
- Don’t Sweat Immutable data, the compiler will pick an efficient computation strategy.
- Chain functions together to operate on intermediate results through function composition.
- Start today with Lambdas



A language that doesn't affect the way you think about programming is not worth knowing.

— Alan Perlis —

Winner of First Ever Turing Award
1966