Hacking With Monads

FUNCTIONAL PROGRAMMING FOR THE BLUE TEAM

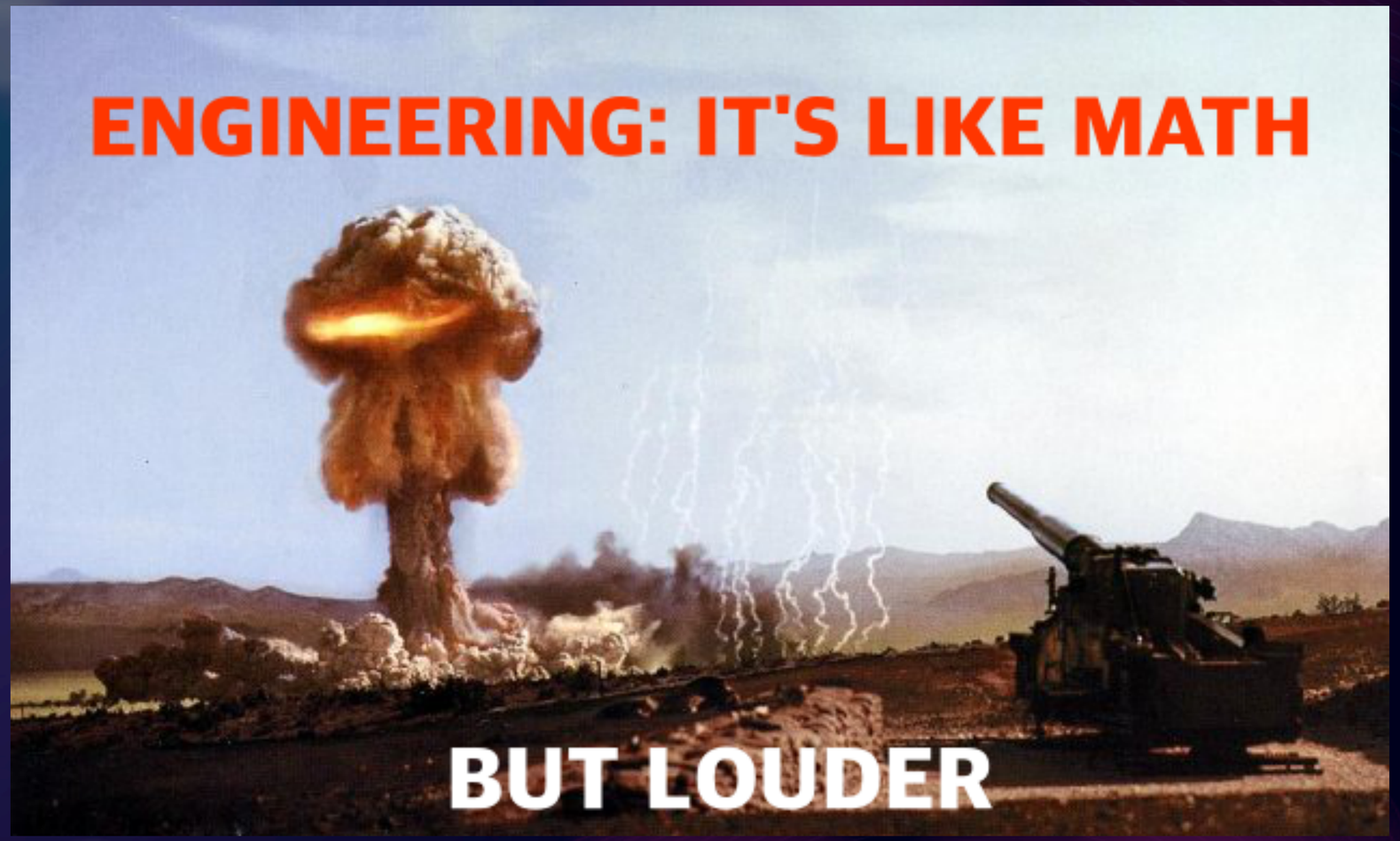
Functional Programming: Why?

There are two tribes of programming philosophy:

- Engineers
- Mathematicians

Engineers

- Built the hardware
- Focus on practical results
- Deal with real-world constraints
  - time
  - materials
  - tools
- Will create memes like this:



Mathematicians

- Created the language
- Focus on theoretical truth
- Deal with ideas and possibilities
  - proofs
  - equations
  - logic
- Will tell jokes like this:

A physicist, a biologist, and a mathematician are sitting in a cafe. Across the street is a house. As they sit talking, two people go into the house. Momentarily, three people come back out.

The physicist says, "our initial count must have been incorrect."
The biologist says, "they must have reproduced."

The mathematician says, "now... if one person goes back in, the house will be empty."

Programming Languages

From The Engineering Tribe:

- Fortran
- ALGOL
- COBOL
- Pascal
- C, C++, Java, et al.

Languages from the Engineering tribe are designed to let you, the programmer, tell the machine exactly what, how and when to do something.

Programming Languages

From The Mathematics Tribe:

- Lisp, Scheme
- ML
- Erlang
- Haskell
- Clojure

Languages from the Mathematics tribe will let you, the programmer, describe a system to the machine by defining and combining logical expressions:

Functions.

Thinking Mathematically

Mathematicians play by different rules than engineers.

The rest of us can say:

Let a = 10

A mathematician will be fine with the first statement.

and then, later:

But they will not tolerate the second one.

a = a + 10

Variables

To a mathematician, variables are just unknown values. We don't know what they are yet, so we represent them with symbols.

To a software engineer, variables are *locations for storing data*.

They exist in the real world.

Their contents can change.

Programs

Programs don't exist in mathematics.
Symbols exist.
Expressions exist.

Programs are real-world entities. They have behavior that changes based on their internal state.

A program, just like the computer it runs on, is a state machine.

State Machines

A state machine will change behavior based on its rules and input.

Programmers define those rules.

Failing to anticipate any input for any program state can create behavior that is unwanted or undefined.

Program Complexity

The complexity of a program arises from the total number of internal states it can have.

Managing complexity becomes harder as the total number of possible states rises.

This number rises as you add features.

What Have We Tried?

- 1930s: Turing machine and lambda calculus created
- 1950s: COBOL and Fortran introduced
- 1960s: Simula, the first object-oriented language
  Edsger Dijkstra writes "GOTO Considered Harmful"
- 1970s: Alan Kay and team, inspired by Simula, create and refine
  Smalltalk at Xerox Palo Alto Research Center

What Have We Tried?

- 1980s: 4th Generation Language builders try to eliminate programmers.
- 1995: *Design Patterns* by Gamma, Helm, Johnson, Vlissides
- 1995-2005: ZOMG, DESIGN PATTERNS
- 2005-06: We can't make the CPU faster without burning down your office, so here's multiple cores. Learn concurrency. Good luck. -- AMD and Intel

What Have We Tried?

- Object-oriented design tries to manage complexity by breaking apart a program into a community of miniature programs, each of which contains its own data and methods for doing its job.

- Objects in a system communicate by passing messages to each other. *This was Alan Kay's main idea,* but people focused on everything else.

- Multiple cores?? I'm supposed to do multithreaded everything now??

What Have We Tried?

- Objects were supposed to be simple, robust and easy to test.
- In reality, managing the complexity of objects turns out to be non-trivial.
- Data that can be changed can also be corrupted.
- Code turns out to be just another kind of data.

Side Effects

A side effect is any change in the state of your program or its environment that happens as a *byproduct* of your code's execution.

It's impossible to eliminate side effects altogether (if we did, our program would never be able to do anything.)

We need side effects. But we also need ways to handle them better.

Reducing Side Effects

Each part of your program that creates or manipulates state will become harder to understand and debug as complexity increases.

Every bit of code that creates or changes program state expands the potential attack surface of your program.

Minimizing the creation and manipulation of internal state is not trivial. But if you can do it, it will yield more predictable, more reliable code.

A Taste of Pure Functions

A function is an expression that describes a unit of logic in our program.
A pure function is one that does its job without creating any side effects.

```
factorial 0 = 1
factorial n = n * factorial(n - 1)
```

The above code computes a result from its input, and returns it, without affecting anything else. No matter what happens, it will always behave the same way.

A Word About Types

Over time, it's very common to grow accustomed to the meaning of "data type" as a flavor of data.

Strictly speaking, a type is a *set*.

The type *Integer* is the set of all integers.

The type *String* is the set of all possible combinations of characters, up to the maximum allowed string length.

A Word About Types

When we declare a variable: Var a : Int = 256

We are saying two things:

- a is a member of the set of all integers
- the value of a is 256

Functional Concept: Immutability

A variable or data structure that cannot be changed is one that cannot easily be corrupted.

Immutable Items in Python:

- int
- float
- bool
- str
- tuple
- frozenset

Mutable Items in Python:

- list
- set
- dict

Functional Concept: Immutability

One big caveat here: Everything in Python is an object.

This can be a rude awakening if you aren't aware of the implications.

If you say:

```
a = 'abcd'
```

and then say:

```
a = a + 'efgh'
```

and finally,

```
print(a)
```

you will see the output 'abcdefgh'.

Functional Concept: Immutability

How can we re-assign a value to an immutable object?

We didn't. We had a reference (a) that pointed to an immutable object ('abcd').

Then we created a new immutable object (a + 'efgh') and pointed our reference (a) at it.

Strings are immutable objects. References can be reassigned to point to some other object.

Or, to use another set of terms:

- a is a name.
- 'abcd' is an immutable object.
- a = 'abcd' creates a binding of the name a to the object 'abcd'.
- we can bind a to a new object with the statement a = a + 'efgh'.

Functional Concept: Monoid

Imagine a pair of rules that form a group.

add(a, b) = a + b where a and b are always integers

identity = 0

With these rules, the following things are true:

```
add(10, 10) == 20          or      10 `add` 10 == 20
add(10, identity) == 10    or      10 `add` `identity` == 10
```

Functional Concept: Monoid

Here's an example with strings. But why do we need the second of these two rules?

```
concat(a, b) = a + b
identity() = ''
```

```
'abc' `concat` 'def' == 'abcdef'
'abc' `concat` `identity` == 'abc'
```

The easiest answer is that whatever kind of data we are computing with, we need something that does the equivalent job of zero in addition, or one in multiplication, or an empty string in concatenation. Applying our monoid function to some value A and our identity function will always yield the same value A.

Functional Concept: Monoid

```
m :: a -> a
u :: a
```

m takes something of type a and returns something of type a.

u applied to any a returns that same a.

Monoids are the building blocks of functional programming. They are key to implementing features like recursion and iteration.

They are also excellent for building function pipelines, a practice of mixing and matching functions that is more commonly called *composition*.

Functional Concept: Monad

m :: a -> Ma

m takes an a, and returns some transformation of an a.

The monad transformation can be computation, but it doesn't have to be. It can be literally anything, including something that has side effects.

Monads are *the* tool that we use in functional programming to perform side effects. They allow a more controlled, rigorous way of working with the real world.

Functional Concept: Monad

Some of the most common uses of monads in a functional language:

- Print to the console
- Perform file or network I/O
- Interact with a database
- Control a device
- Basically, do anything that creates or maintains state somewhere

Functional Programming: Conclusions

Is it magic or sorcery? No.

Is it a perfect answer to the increasing complexity of software systems and the increasing inability of humans to design and build bug-free, vulnerability-free software? Probably not.

But it's a step forward, and the fact that it's showing up more and more in mainstream languages is evidence that it has merits.