

# Engineering a Compiler

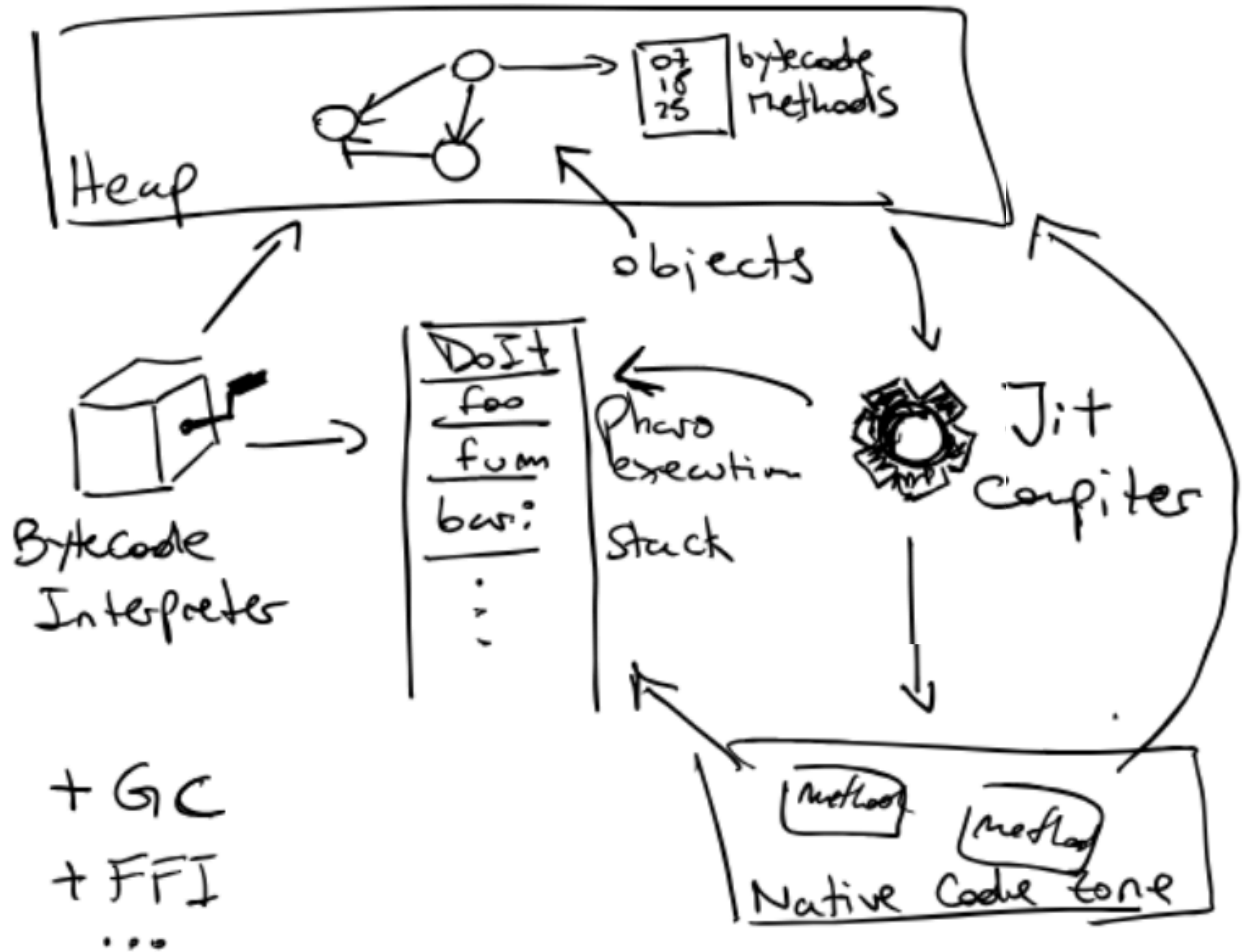
**Writing an ARMv8 backend for a JIT compiler in 2 days per week**



**Guille Polito**

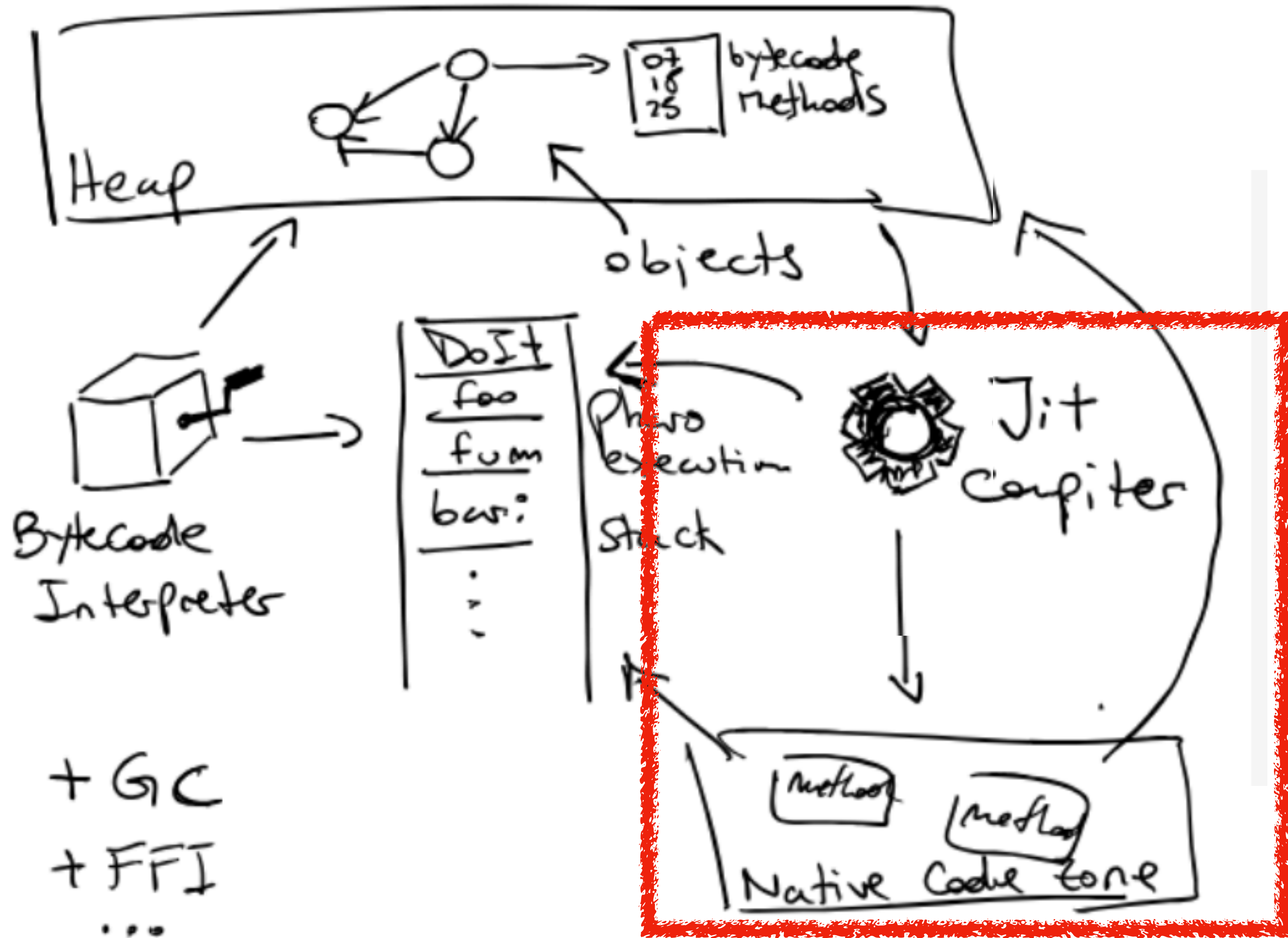
# Context

## The Pharo VM



# Context

## The Pharo VM



# Pharo VM JIT Compiler

## Overview

```
<76> push 1
<20> push 17
<B0> send #+
<7C> returnTop
```

bytecode

```
move r1 #1
move r2 #17
checkSmallInt r1
checkSmallInt r2
add r3 r1 r2
checkSmallInt r3
move r1 r3
ret
```

CogRTL - IR

```
move X5 #1
move X3 #17
test X5 #0x1
je 0x40
test X3 #0x1
je 0x32
add X0 X3 X5
test X0 #0x1
je 0x24
move X5 X0
ret
```

Machine Code

IR Generation  
"gen\*"

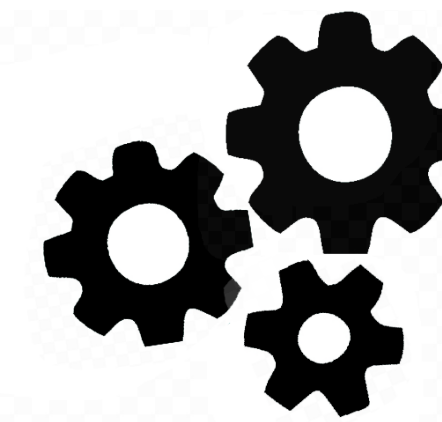
Machine Code Generation  
"Concretize"



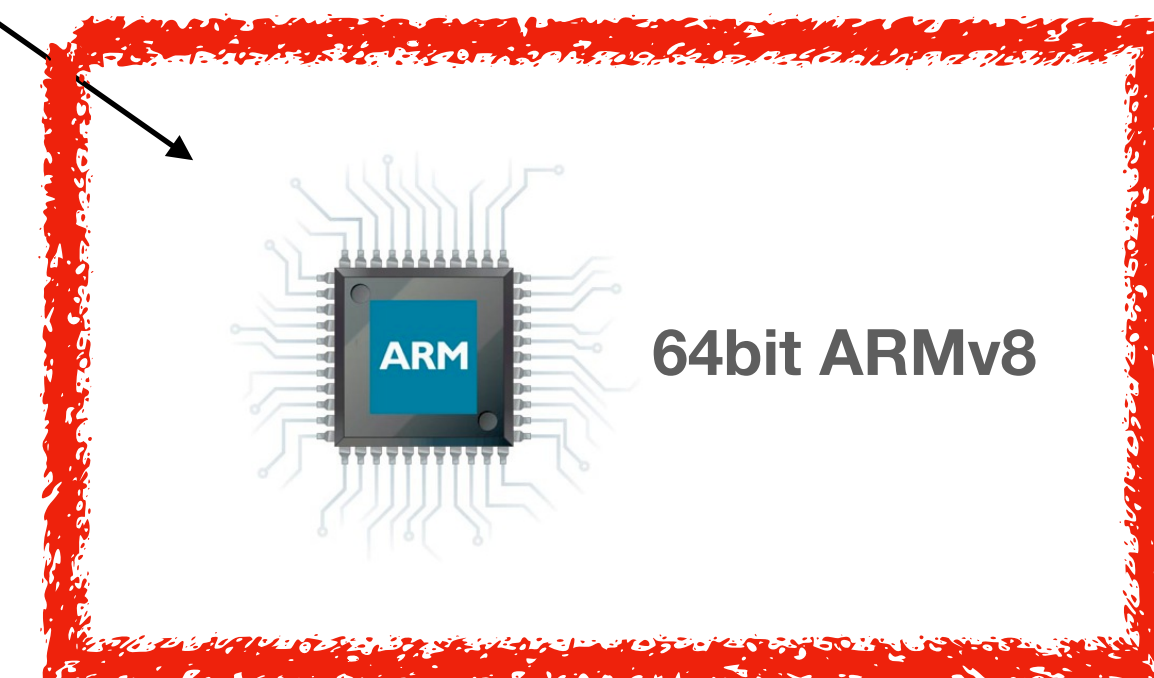
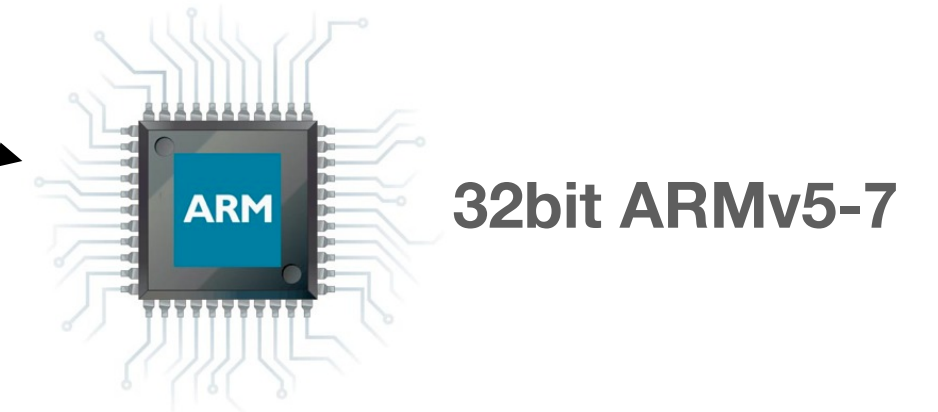
# Implementing an ARMv8 Backend

- ARMV8 is now pervasive:
  - New Apple Silicon
  - Raspberry Pi 4
  - Microsoft Surface Pro X
  - PineBook Pro
  - ...

```
move r1 #1
move r2 #17
checkSmallInt r1
checkSmallInt r2
add r3 r1 r2
checkSmallInt r3
move r1 r3
ret
```



Machine Code Generation  
“Concretize”



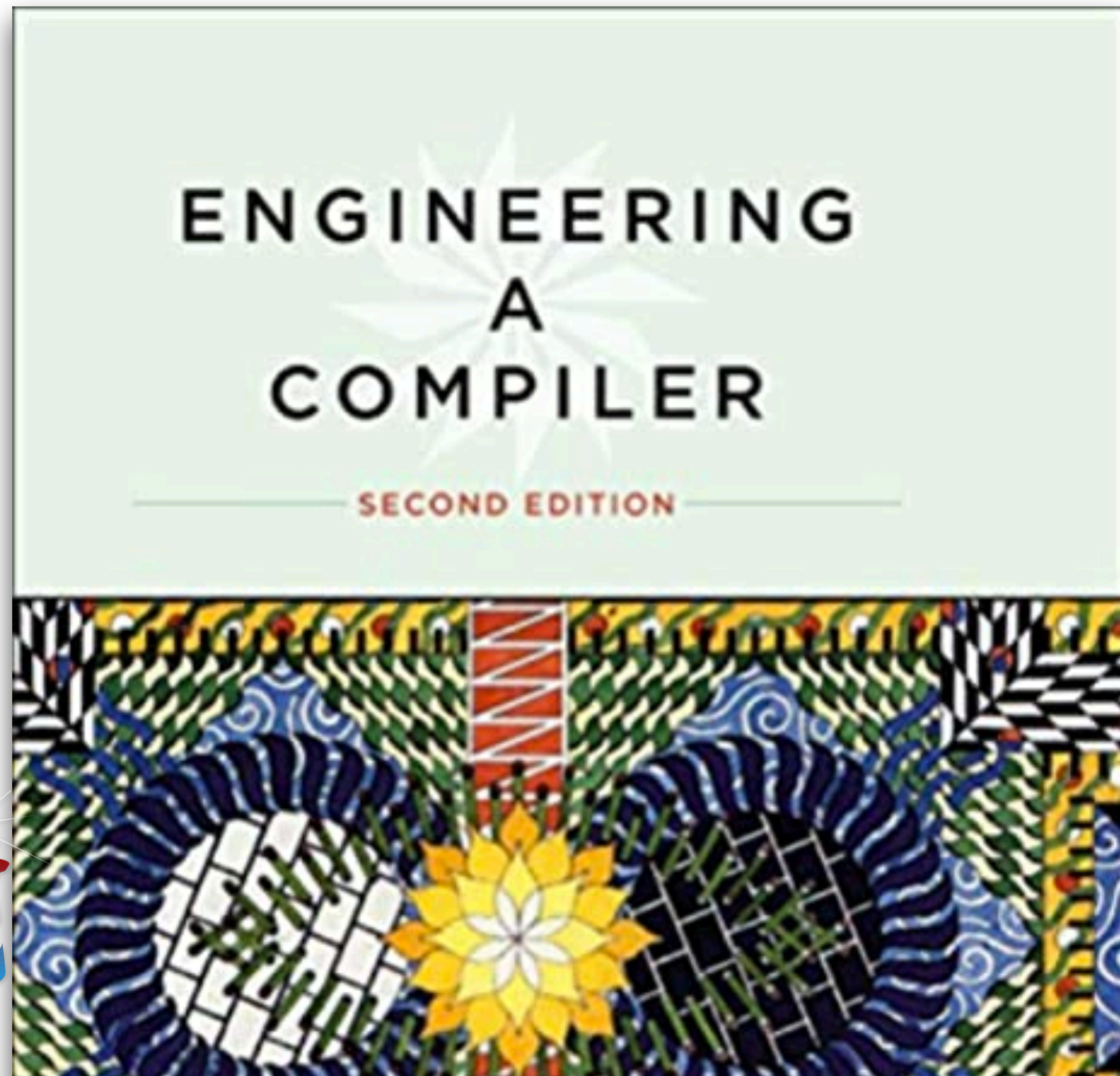
# Some other numbers to contextualise

- 255 bytecode (77 different) + ~340 primitives
- 146 different IR instructions
- Instruction patching (+runtime disassemble)
  - optimizations such as inline caches
  - unused generated machine code is garbage collected
    - (thus moved, and its callers need to be re-linked)

Lots of combinations!

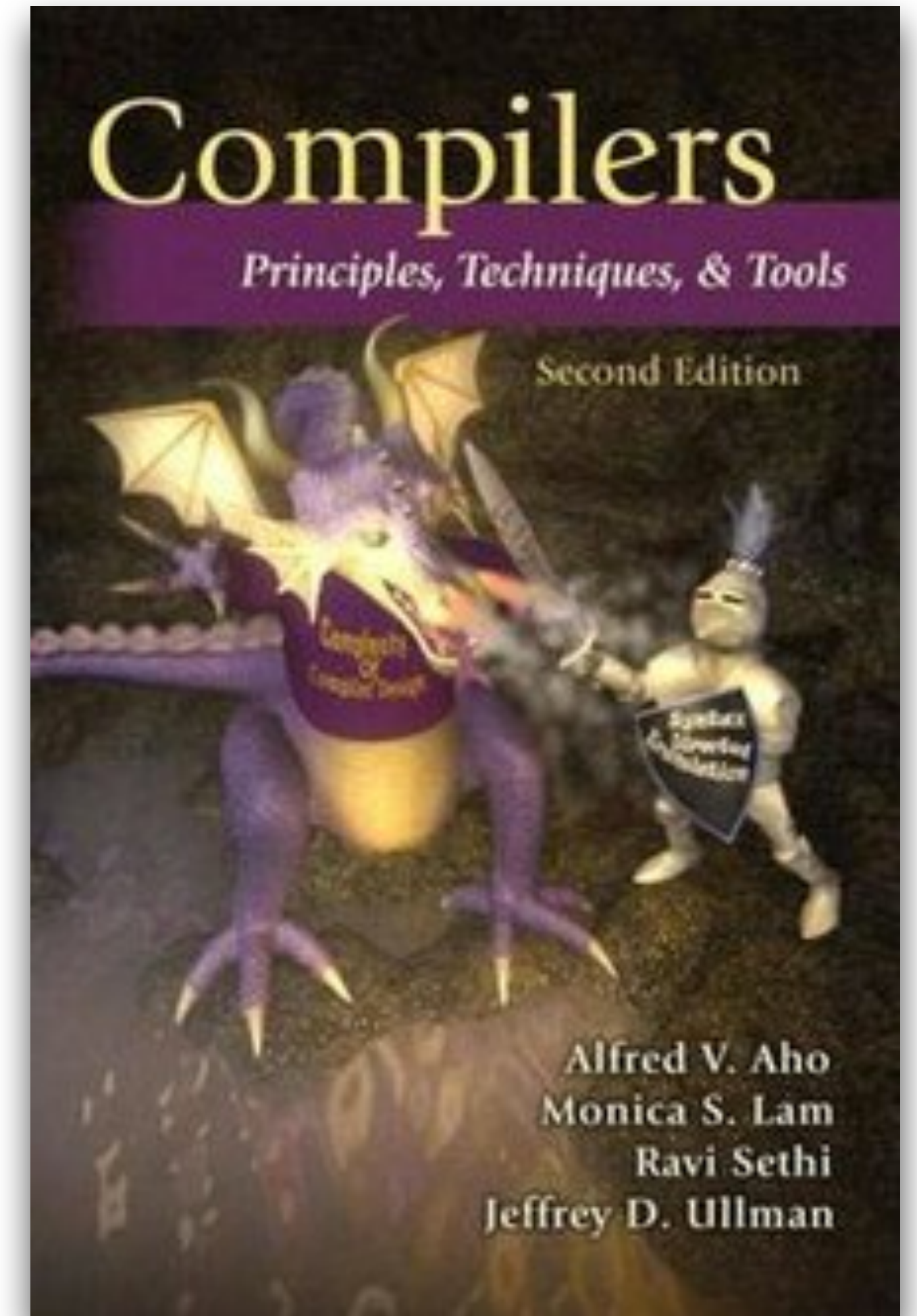
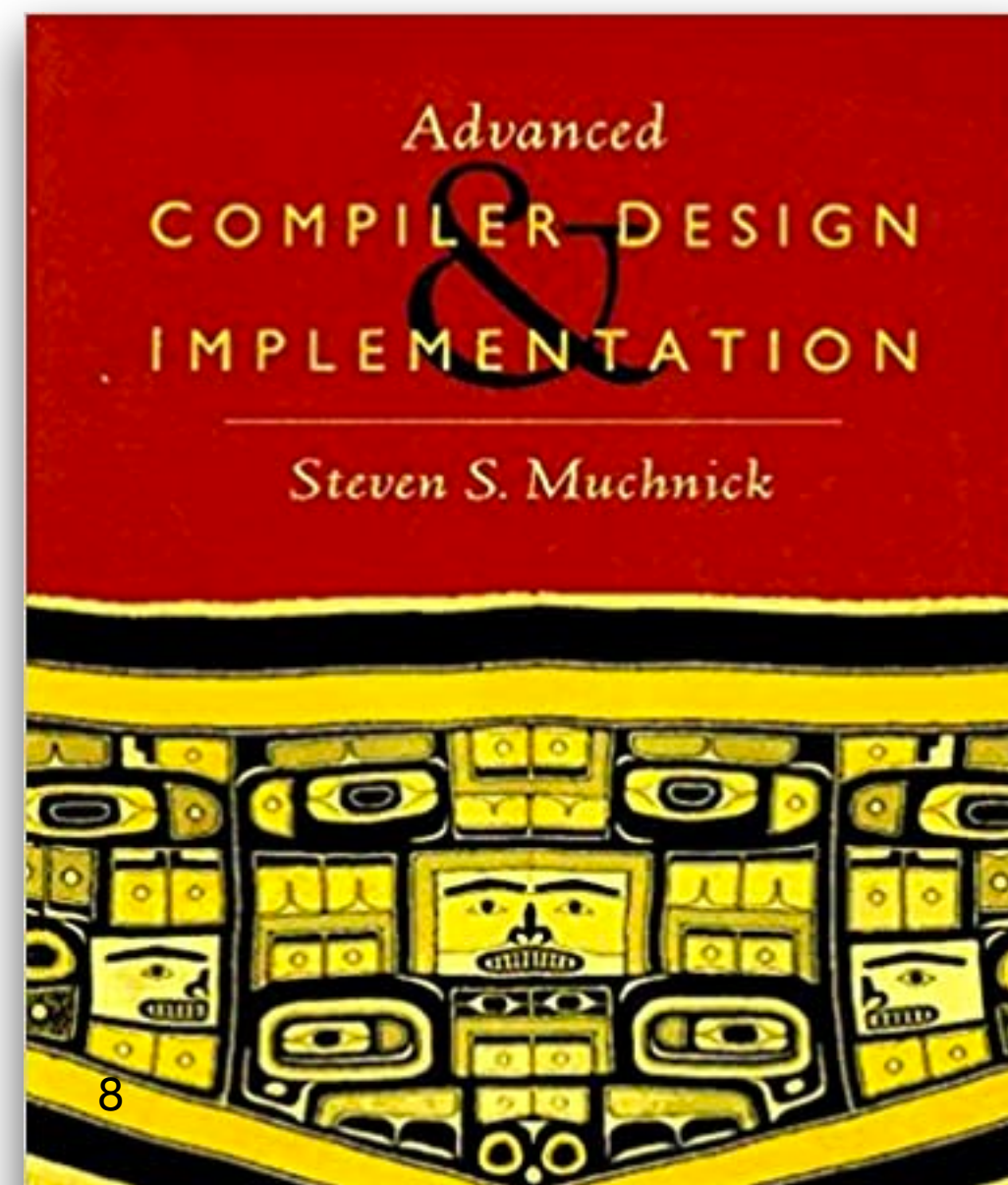
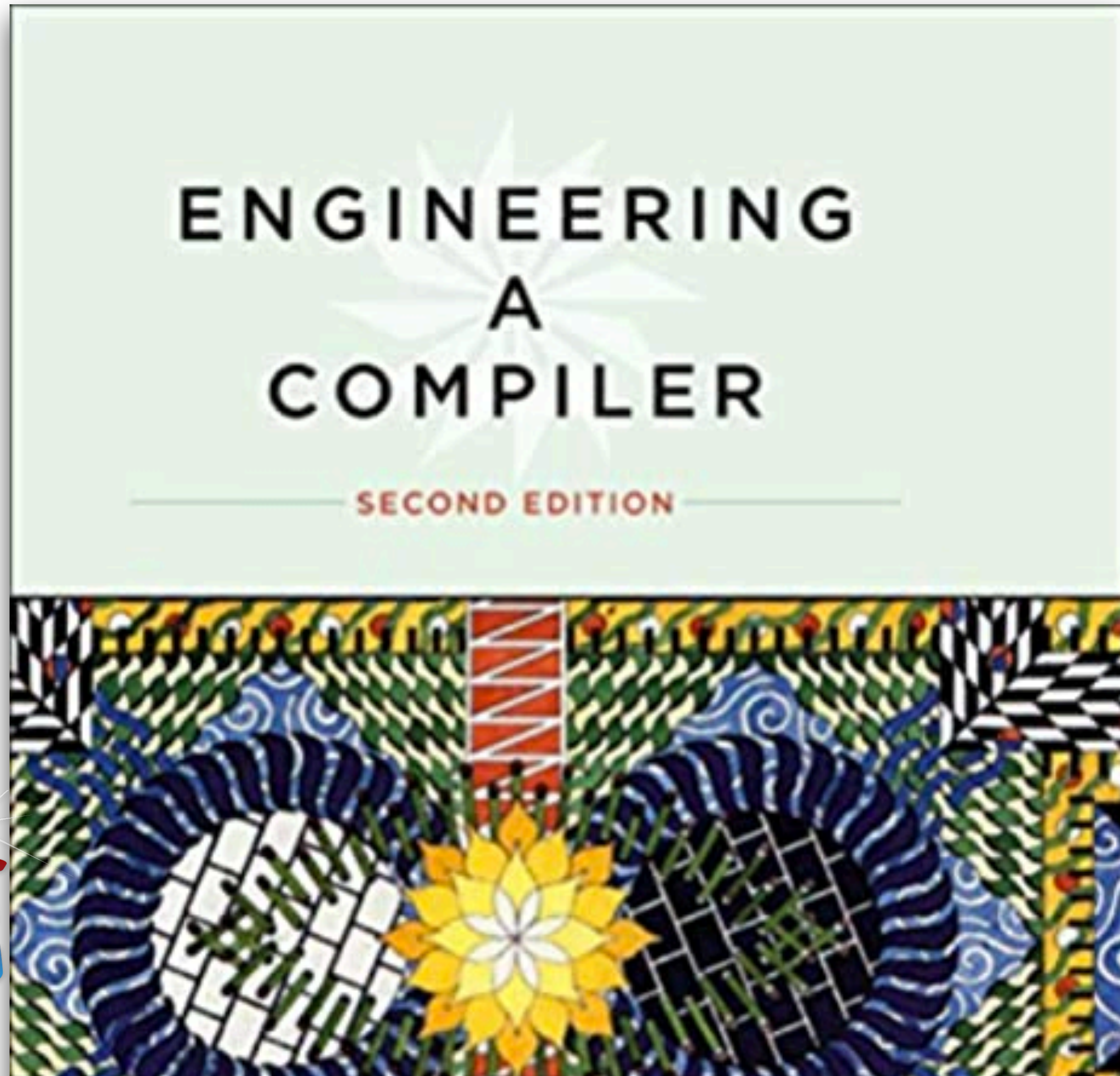


# Engineering a Compiler



# Engineering a Compiler

Good books, but centered on the *underlying algorithms*





# Engineering a Compiler

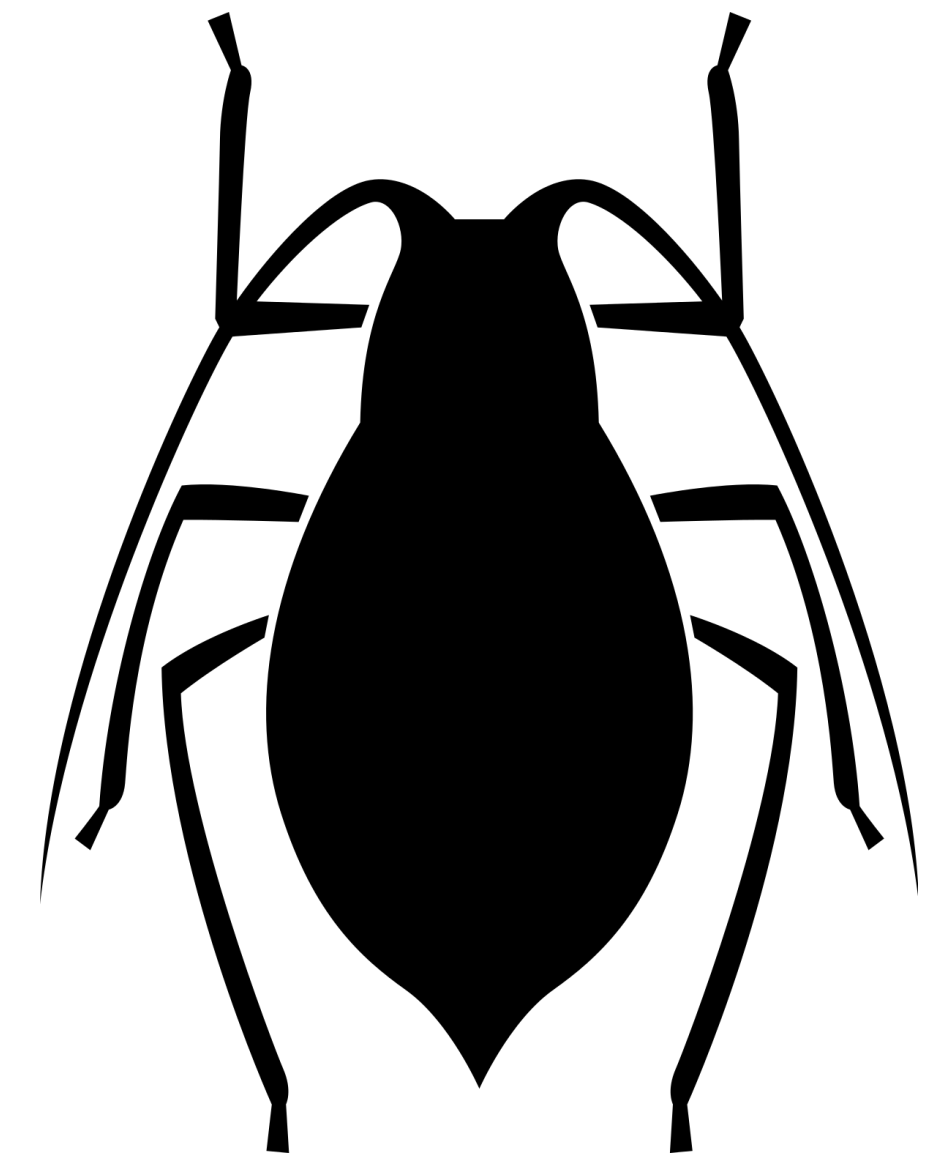
## Software Engineering



Testable



Incremental



Easy Debugging



# Engineering a Compiler

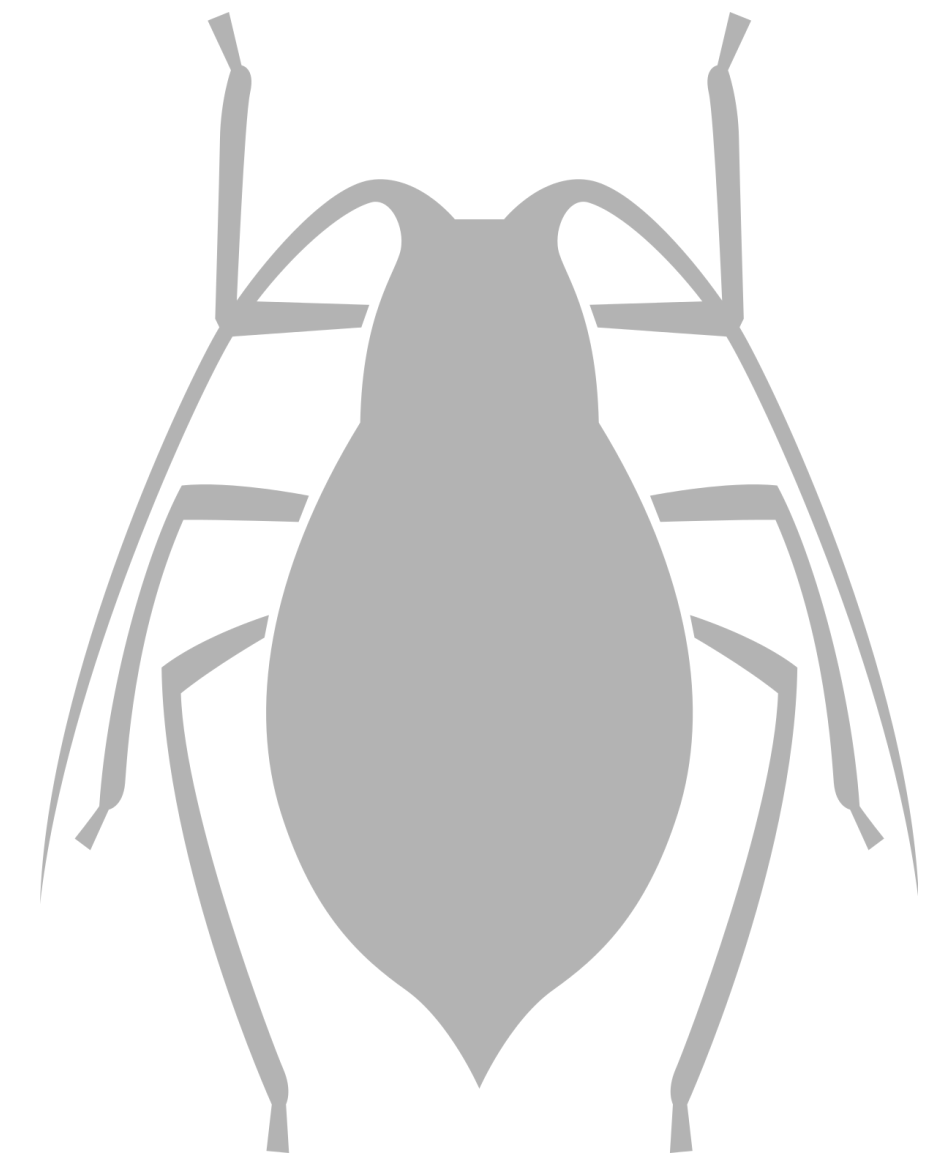
## Software Engineering



Testable



Incremental



Easy Debugging



# Testing a Compiler

## Insights: Black box testing

Black box testing

=> depend only on **observable behaviour**

=> **reusable** in different backends

=> more **resistant to changes** in the implementation

```
testPushConstantZeroBytecodePushesASmallIntegerZero
```

```
self compile: [ compiler genPushConstantZeroBytecode ].  
self runGeneratedCode.
```

```
self assert: self popAddress equals: (memory integerObjectOf: 0)
```

# Testing a Compiler

## Insights: Cross-compile / Cross-execute



<http://www.unicorn-engine.org>

Use a machine simulator

=> **hardware independent:** test and debug in any machine any backend

=> **parametrizable tests** run the same test with multiple backends

```
testPushConstantZeroBytecodePushesASmallIntegerZero
```

```
self compile: [ compiler genPushConstantZeroBytecode ].
```

```
self runGeneratedCode.
```

```
self assert: self popAddress equals: (memory integerObjectOf: 0)
```

# Intermezzo: Generating machine code

## To LLVM or not to LLVM

- LLVM is a compiler infrastructure: its own IR, its assemblers and disassemblers
- It even has a JIT module
- But
  - **Not observable:** not straightforward to extract the generated code to test it  
=> not testable!
  - **Lock-in:** it leaves **no control** on how machine code is generated, executed and managed  
=> not compatible with the rest of our infrastructure



# Intermezzo: Generating machine code

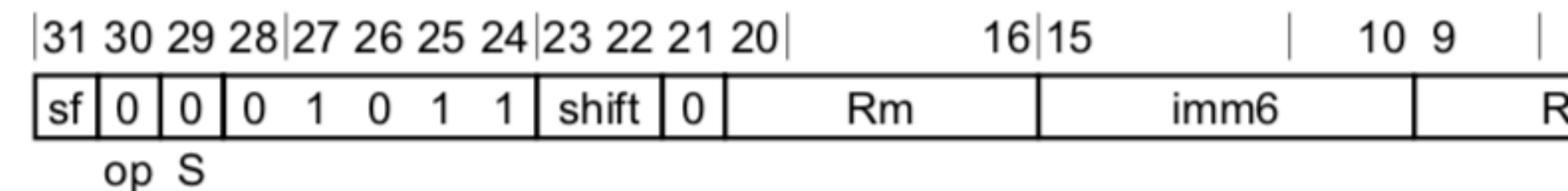
The backend **\*\*is\*\*** an *assembler*

Generates *machine code* from the intermediate representation

```
(is64Bits bitAnd: 1) << 31
  bitOr: ((subtractionFlag bitAnd: 1) << 30
  bitOr: ((setFlagsFlag bitAnd: 1) << 29
  bitOr: (2r01011 << 24
  bitOr: ((shiftType bitAnd: 2r11) << 22
  bitOr: ((rightRegister bitAnd: 2r11111) << 16
  bitOr: ((immediate6bitValue bitAnd: 2r111111) << 10
  bitOr: ((leftRegister bitAnd: 2r11111) << 5
  bitOr: (destinationRegister bitAnd: 2r11111))))))
```

## C6.2.5 ADD (shifted register)

Add (shifted register) adds a register value and an optionally-shifted register value to the destination register.



### 32-bit variant

Applies when `sf == 0`.

ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}

### 64-bit variant

Applies when `sf == 1`.



# Engineering a Compiler

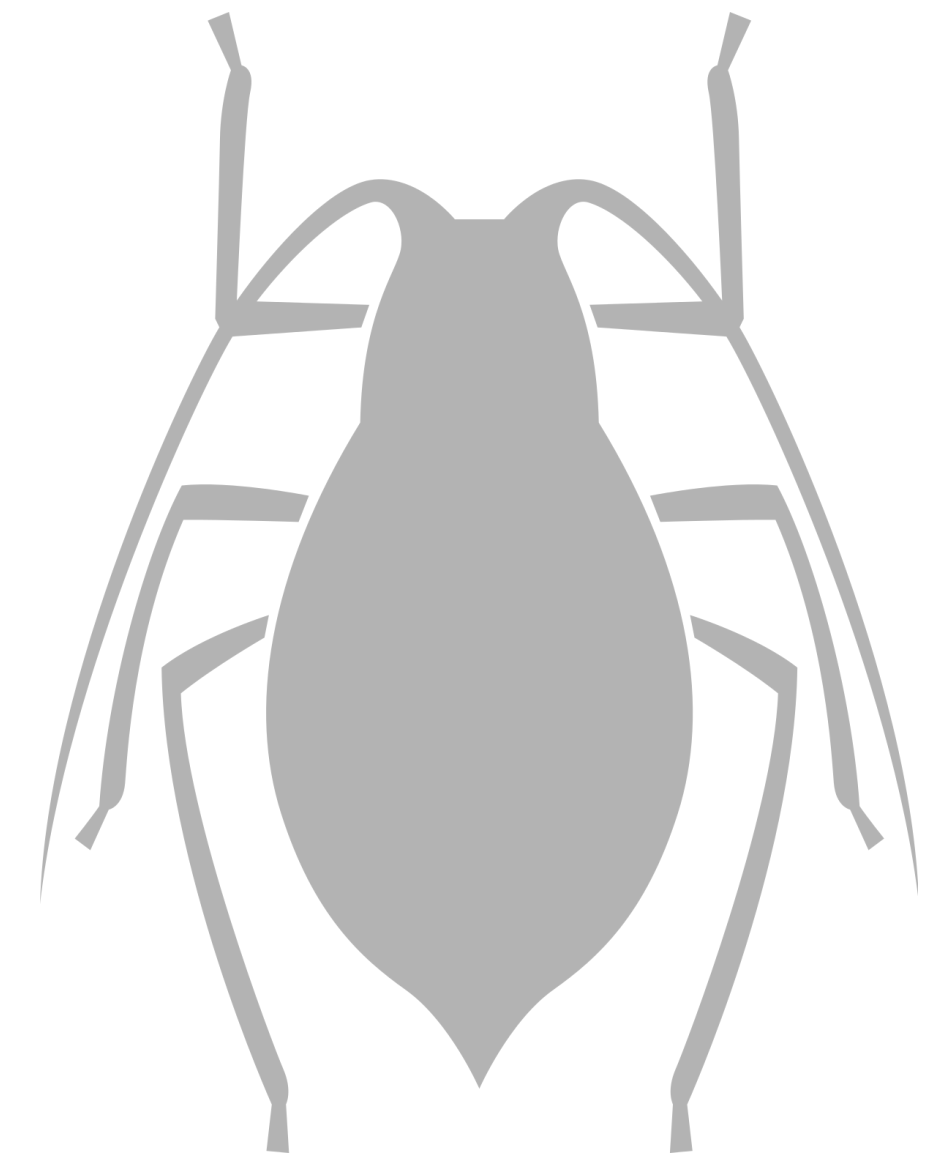
## Software Engineering



Testable



Incremental



Easy Debugging



# Incremental Compiler Engineering

## Insights: Start Small

- First: The **simplest** test you can write for the **simplest** functionality
- Second: The **next simplest** test you can write for the **next simplest** functionality

=> The first focus is in understanding **how to** better write the **tests**

### testPushConstantZeroBytecodePushesASmallIntegerZero

```
self compile: [ compiler genPushConstantZeroBytecode ].  
self runGeneratedCode.
```

```
self assert: self popAddress equals: (memory integerObjectOf: 0)
```



# Incremental Compiler Engineering

Insights: Invest in infrastructure

- Refactor
- Clean
- Create Reusable Components

```
testPushConstantZeroBytecodePushesASmallIntegerZero
```

```
self compile: [ compiler genPushConstantZeroBytecode ].
```

```
self runGeneratedCode.
```

```
self assert: self popAddress equals: (memory integerObjectOf: 0)
```

# Incremental Compiler Engineering

## Insights: Don't hesitate to step back

- If a test cannot be tamed with the current infrastructure
  - The infrastructure is not good!
  - Step back as soon as possible => **lose minutes, not weeks**
  - Choose a simpler test that could help you develop missing points



# Engineering a Compiler

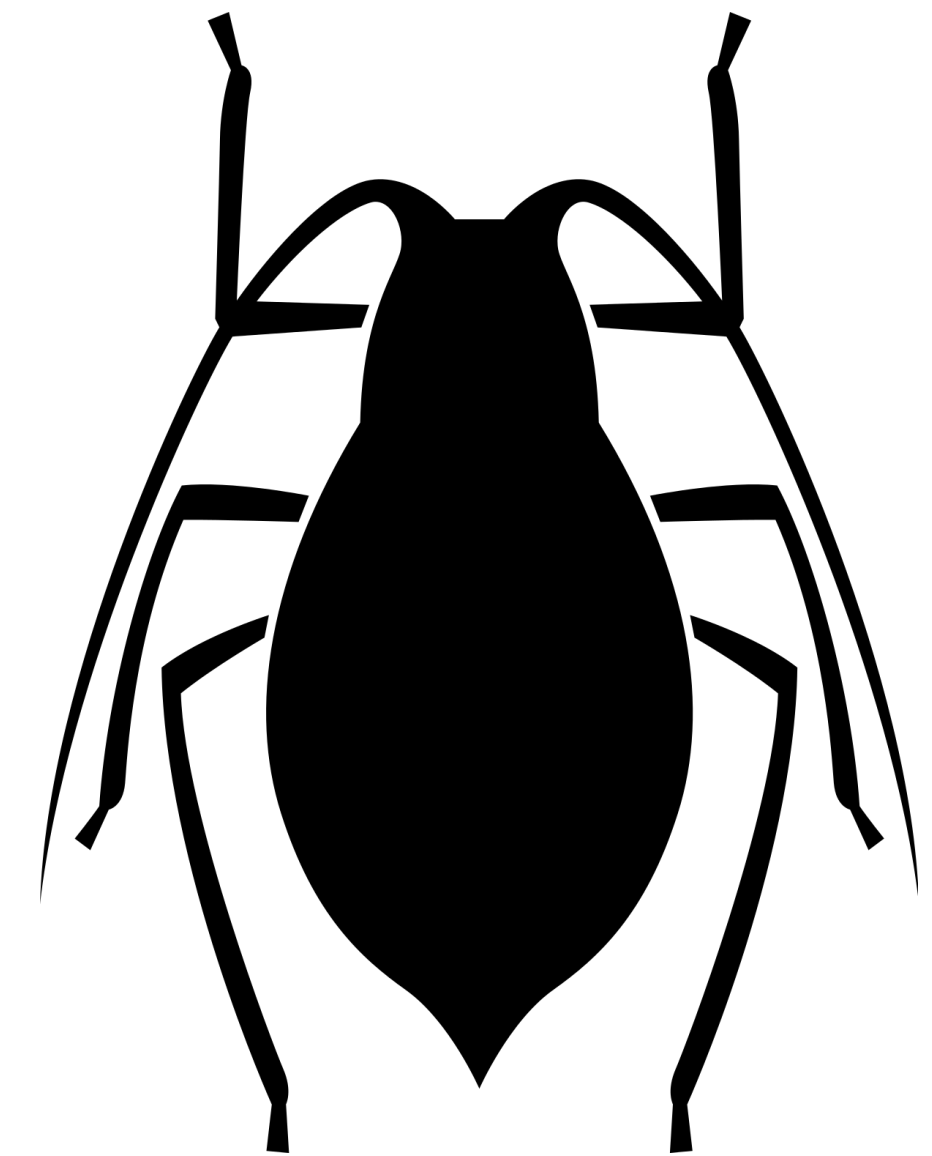
## Software Engineering



Testable



Incremental



Easy Debugging



# Debugging a compiler

## Insights: prototype your own tools

- Prototype **fast**
- Prototype based on **needs**, not desires

Examples:

- Machine debugger
- Bytecode-IR visualization
- Disassembler DSL

Address	ASM	Bytes	Register	Value	Register	Value
16r1000000	mov x1, x24	#[225 3 24 170]	lr	'16r1001000	SP	16r1002FF8
16r1000004	ldr x24, #36	#[56 1 0 88]	pc	'16rBADF00	FP	16r1003000
16r1000008	ldr x16, #40	#[80 1 0 88]	sp	'16r1012F80		16r1003008
16r100000C	str x29, [x16]	#[29 2 0 249]	fp	'16r1003000		16r1003010
16r1000010	ldr x16, #40	#[80 1 0 88]	x28	vmStackPointe'16r1002FF8		16r1003018
16r1000014	mov x17, sp	#[241 3 0 145]				16r1003020
16r1000018	str x17, [x16]	#[17 2 0 249]				16r1003028
16r100001C	mov x24, x1	#[248 3 1 170]				16r1003030
16r1000020	ret	#[192 3 95 214]				16r1003038
16r1000024	nop	#[31 32 3 213]				16r1003040
16r1000028	.inst undefined 16rFFFFFFB5	#[88 251 255 255]				16r1003048
16r100002C	.inst undefined 16r7FFFFFFF	#[255 255 255 127]				16r1003050
16r1000030	.inst undefined 16r1012FBC	#[176 47 1 1]				16r1003058
16r1000034	udf #0	#[0 0 0 0]				16r1003060
16r1000038	.inst undefined 16r1012FB8	#[184 47 1 1]				16r1003068
16r100003C	udf #0	#[0 0 0 0]				16r1003070
16r1000040	udf #0	#[0 0 0 0]				16r1003078
16r1000044	udf #0	#[0 0 0 0]				16r1003080
16r1000048	udf #0	#[0 0 0 0]				16r1003088
16r100004C	udf #0	#[0 0 0 0]				16r1003090
16r1000050	udf #0	#[0 0 0 0]				16r1003098
16r1000054	udf #0	#[0 0 0 0]				16r10030A0
16r1000058	udf #0	#[0 0 0 0]				16r10030A8
16r100005C	udf #0	#[0 0 0 0]				16r10030B0

Jump to

Step

Disassemble at PC



# Debugging a compiler

## Insights: Get real execution feedback

- Simulators are cheap, but not 100% trustworthy
- Full execution (simulated and real HW)
  - more expensive to run
  - cannot unit-test it (less controllable)
- Turn failures into tests
  - If you can reproduce it in a test, **you understand the bug**
  - **Fix with the aid of the test:**
    - => the test is faster to run
    - => and easier to debug than the real execution



# Engineering a Compiler

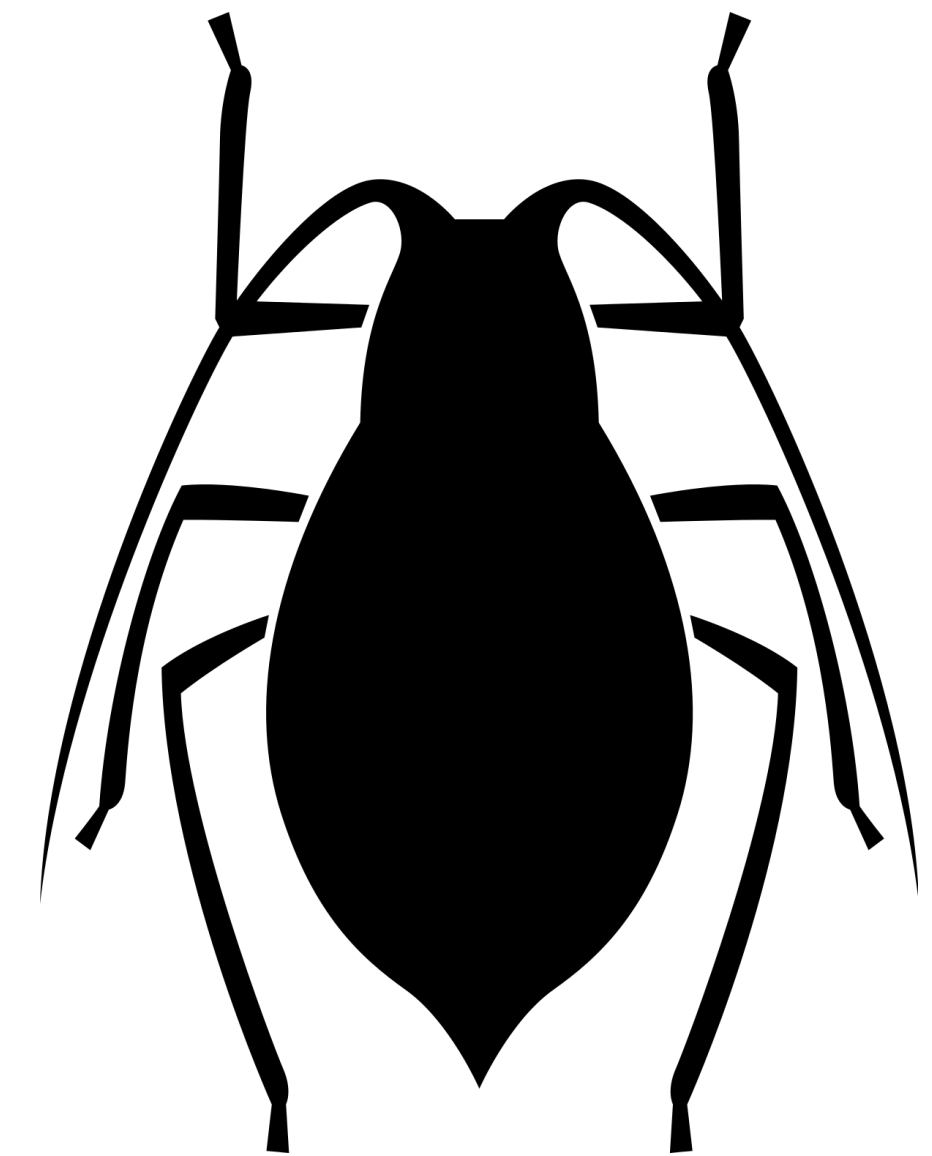
## Software Engineering



Testable



Incremental



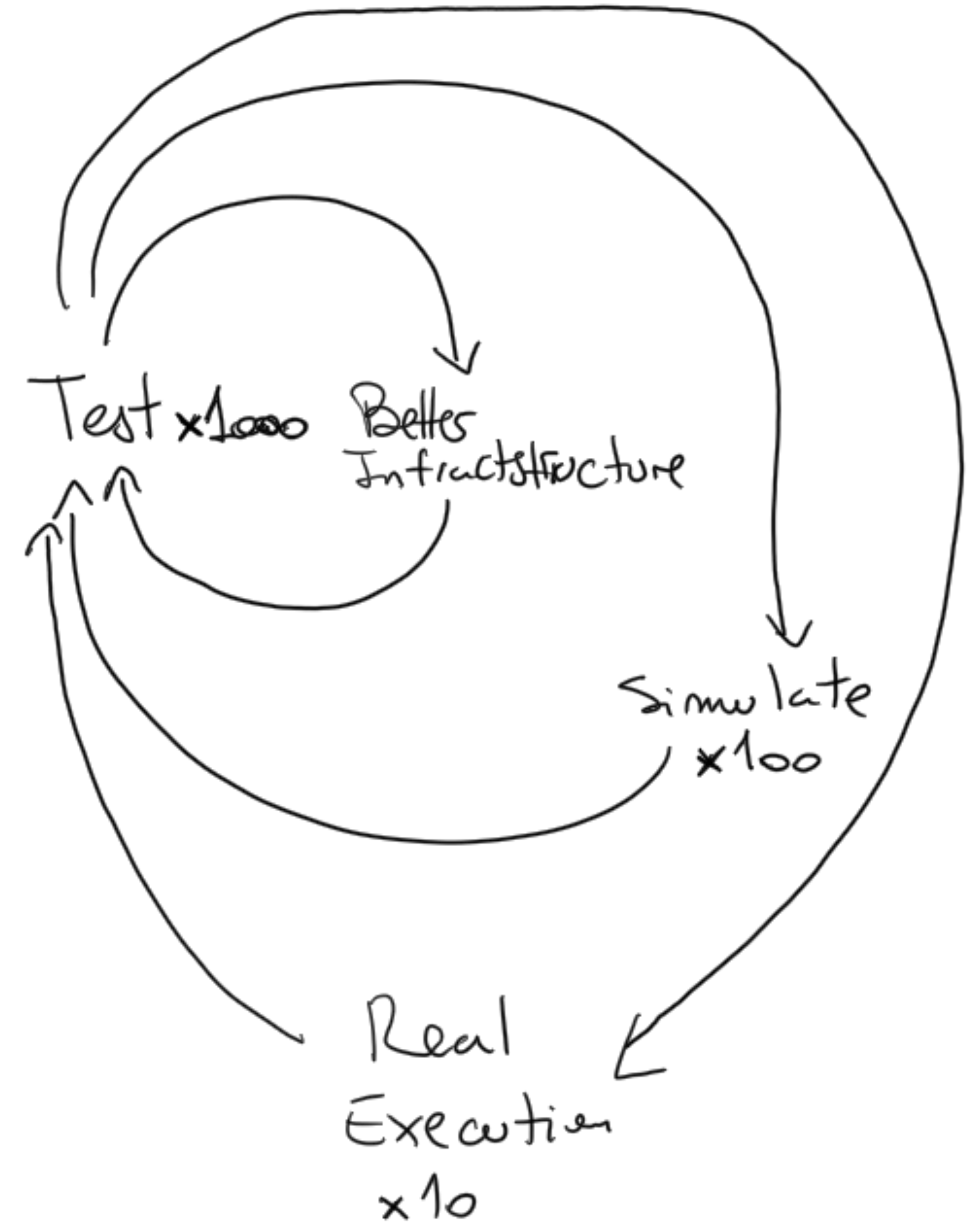
Easy Debugging



# Software Engineering a Compiler

## Our final workflow

- Test as much as you can
- Enhance your infrastructure as much too
- Simulate the execution, less than you run tests
- Run the real app, less than you simulate
- Turn failures into tests



# Software Engineering a Compiler

## Some Final Ideas/Insights

### Tests are code

- **Design** your tests
- **Refactor** your tests
- Make them **modular and reusable**

