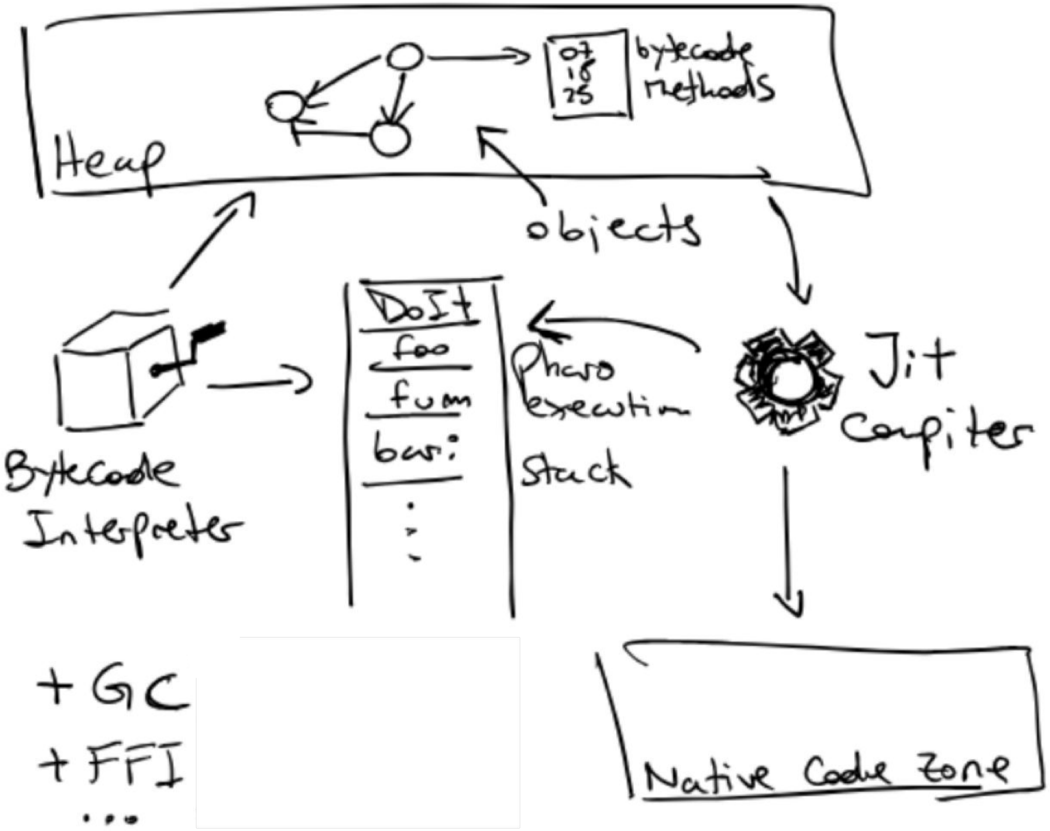
Interpreters, compilers

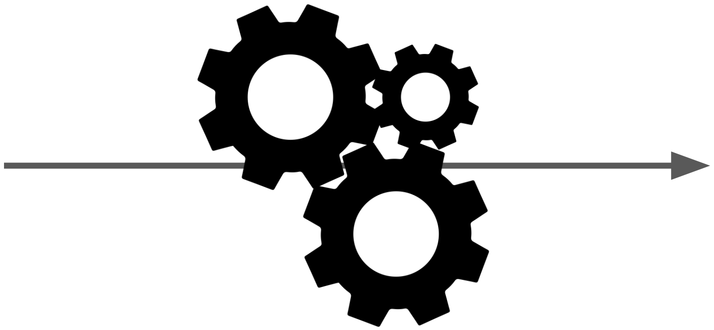
and how I learned to cook thanks to Guille & Pablo

Nico Rainhart
RMoD - September 2022

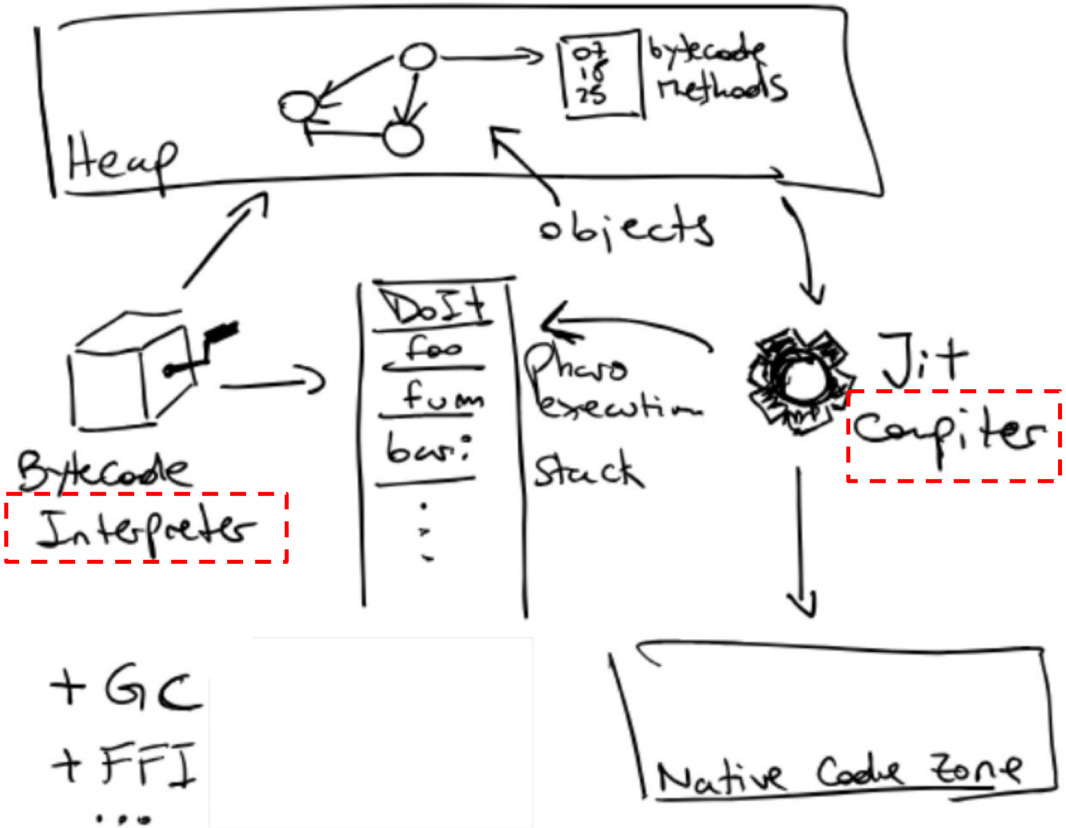

Heap
07
18
25
bytecode methods
objects
DoIt
foo
fum
bar:
Pharo execution stack
Jit Compiler
Bytecode Interpreter
+ GC
+ FFI
...
Native Code Zone

Our goal

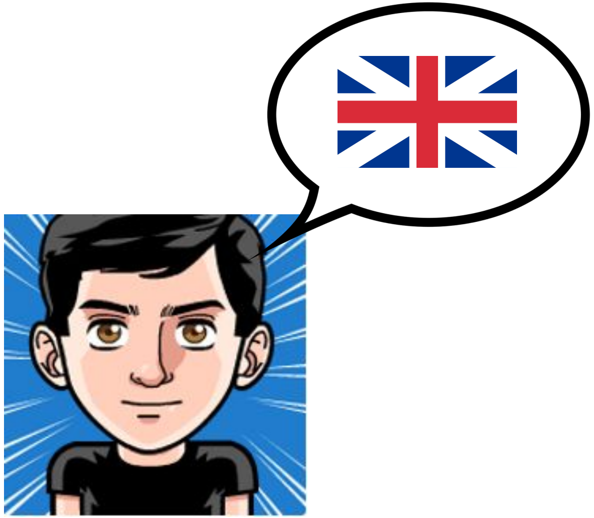
```
a := 1.
condition ifTrue: [
   a := a + 6.
].
^ a + 2.
```



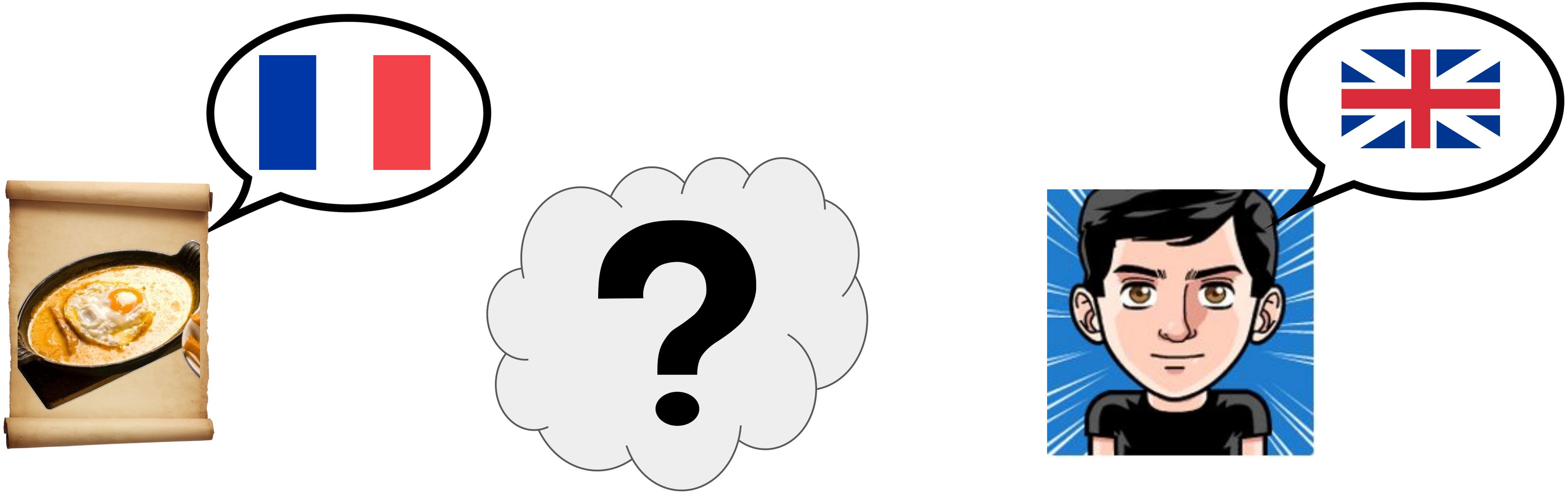
9


Heap
07
18
25
bytecode
methods
objects
DoIt
foo
fum
bar:
Pharo
execution
Stack
Jit
Compiler
Bytecode
Interpreter
+ GC
+ FFI
...
Native Code Zone

Running a program is like cooking a welsh…

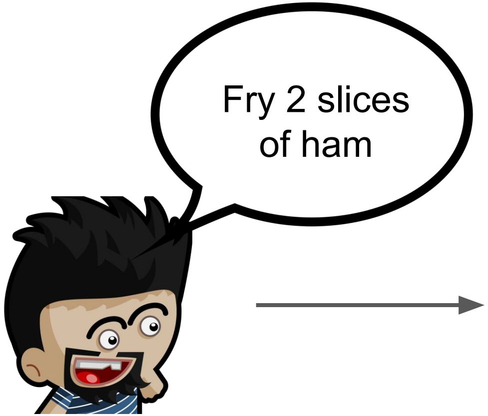


Running a program is like cooking a welsh…

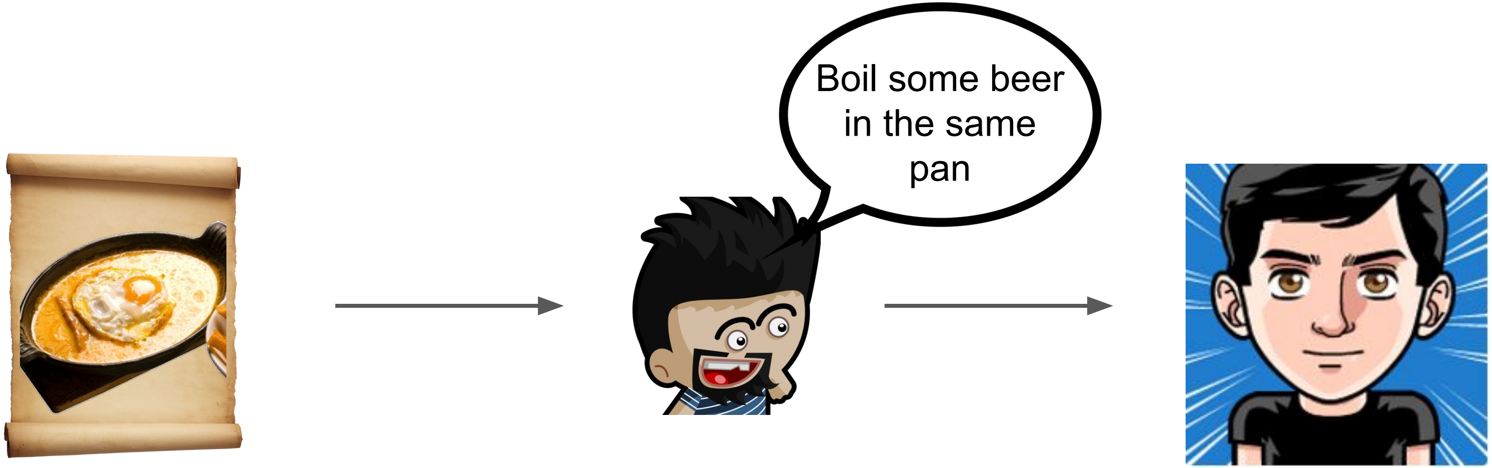


Running a program is like cooking a welsh…

Running a program is like cooking a welsh…

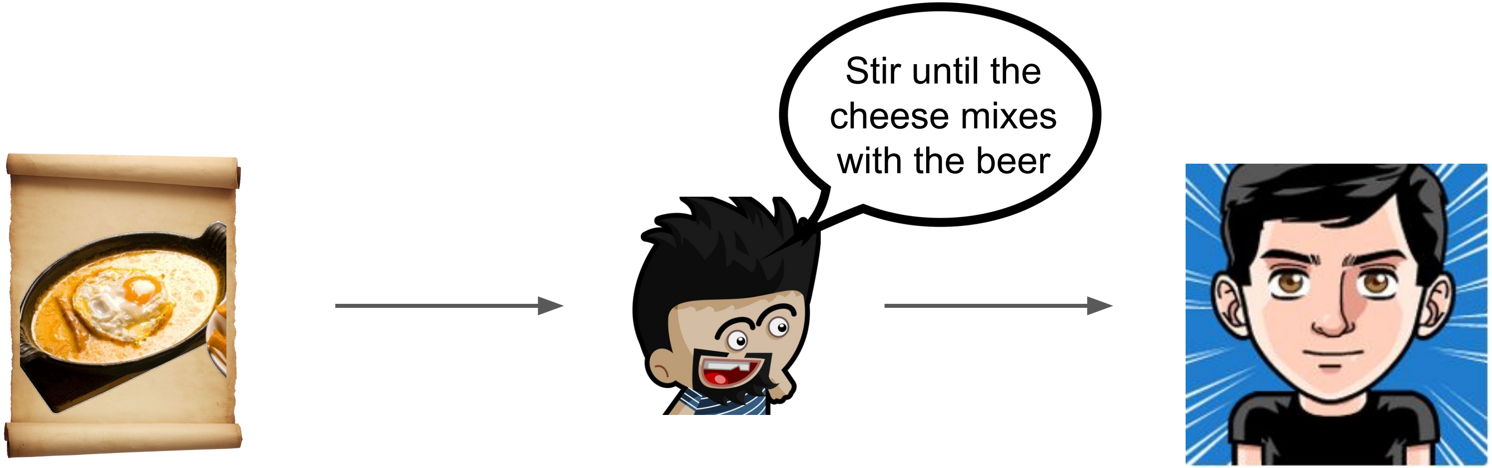


Running a program is like cooking a welsh…

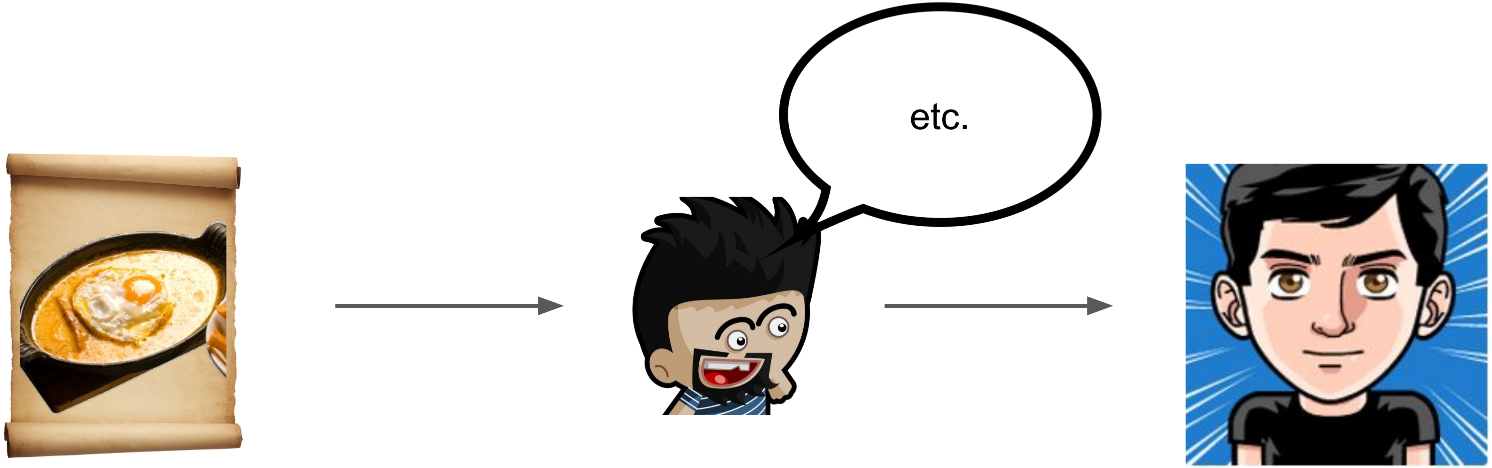


Running a program is like cooking a welsh…

Running a program is like cooking a welsh…



Running a program is like cooking a welsh…

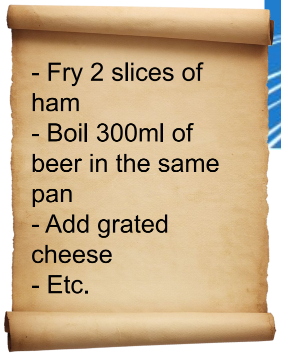


Running a program is like cooking a welsh…

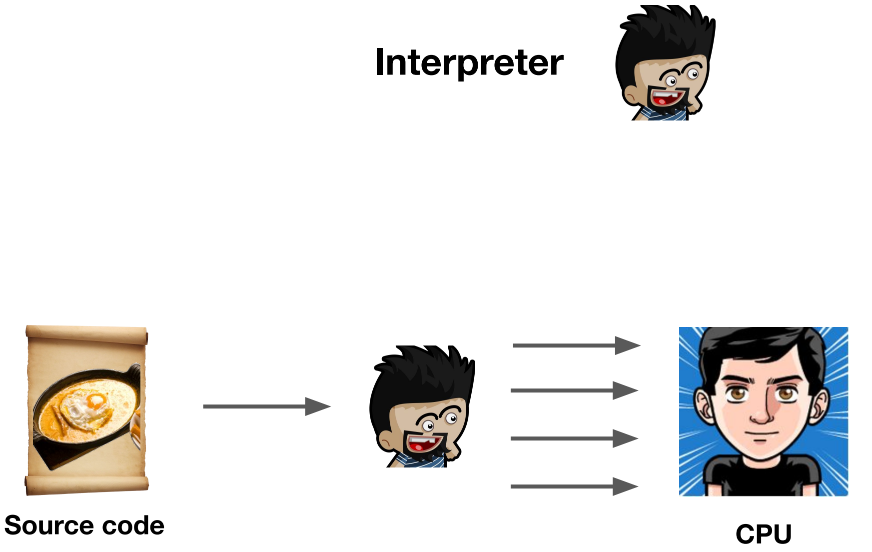
Running a program is like cooking a welsh…

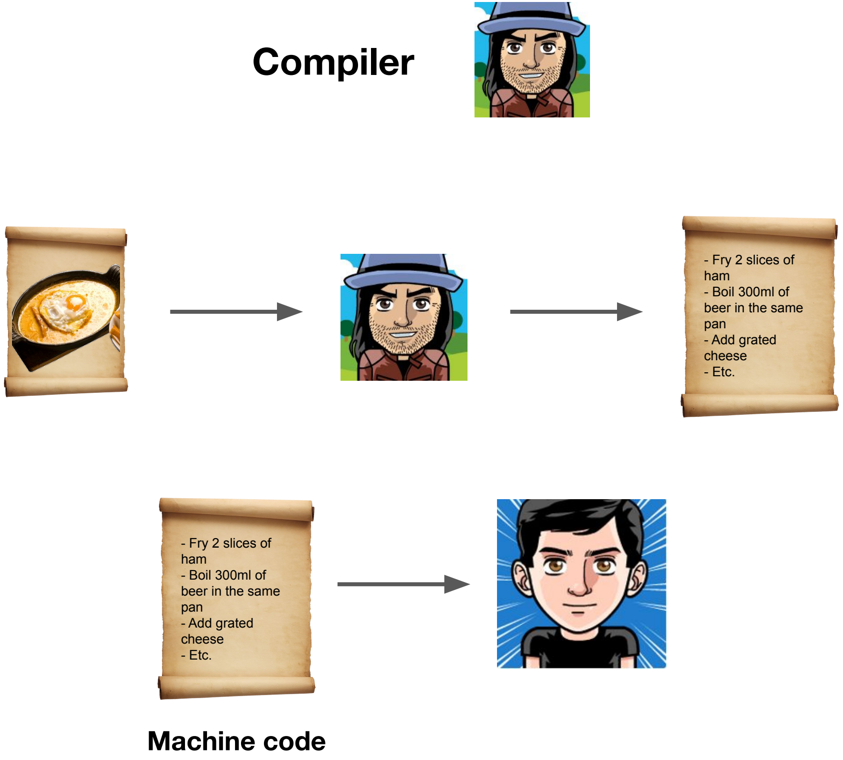
Running a program is like cooking a welsh…

Running a program is like cooking a welsh…



Two different strategies

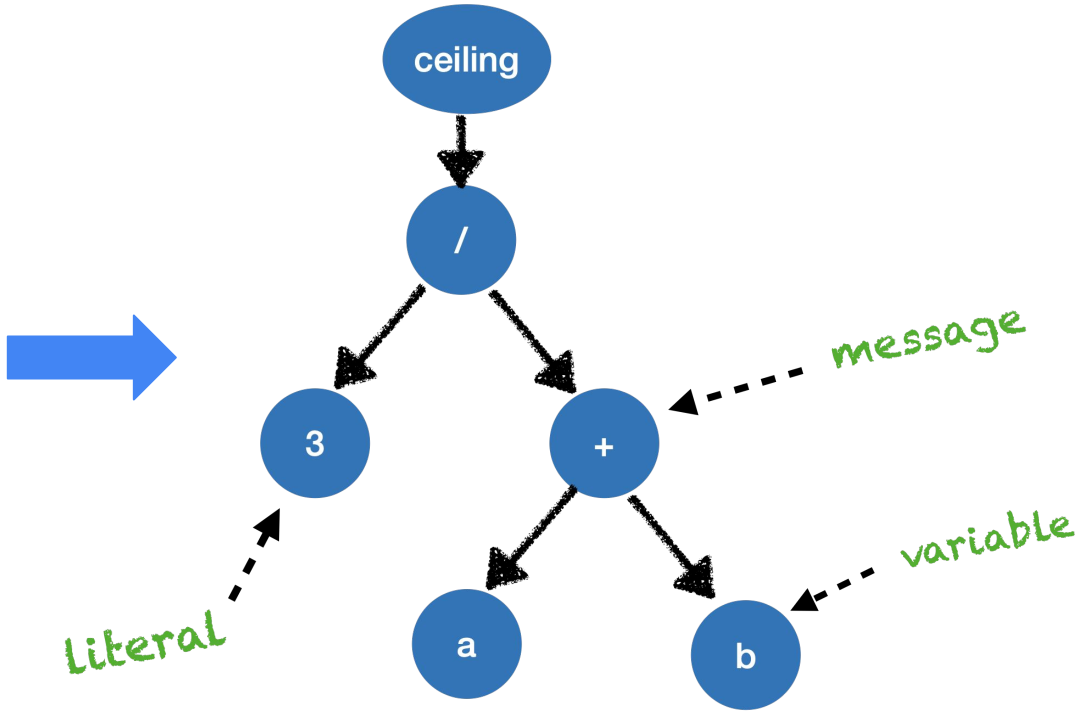




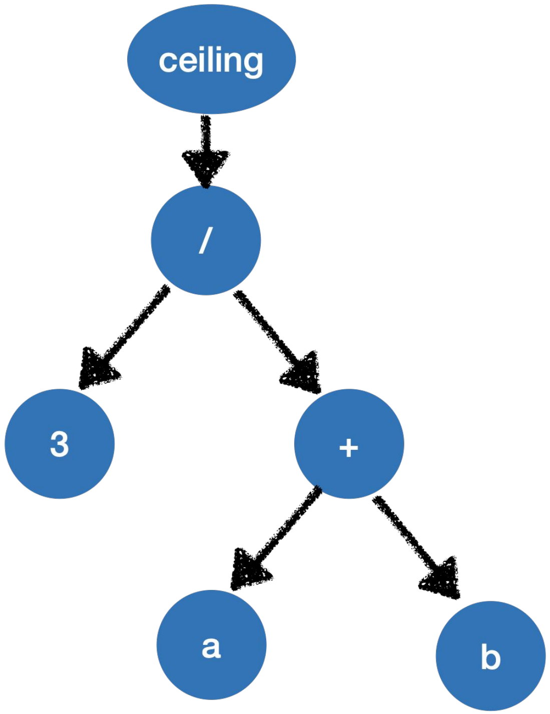
Interpreter

Parsing

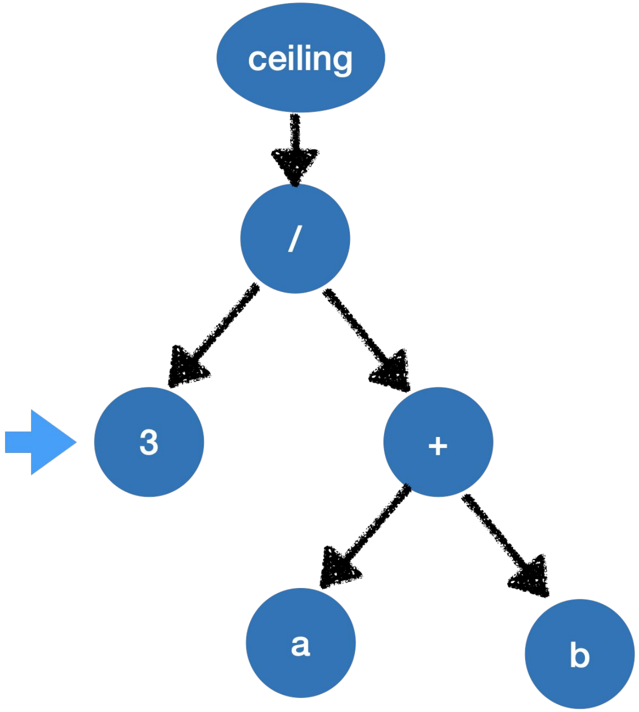
(3 / (a + b)) ceiling

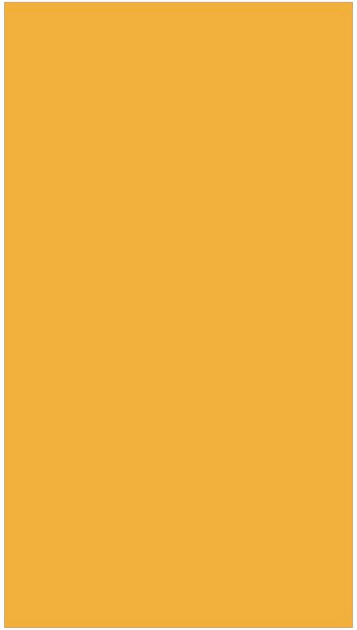


Interpreting the AST

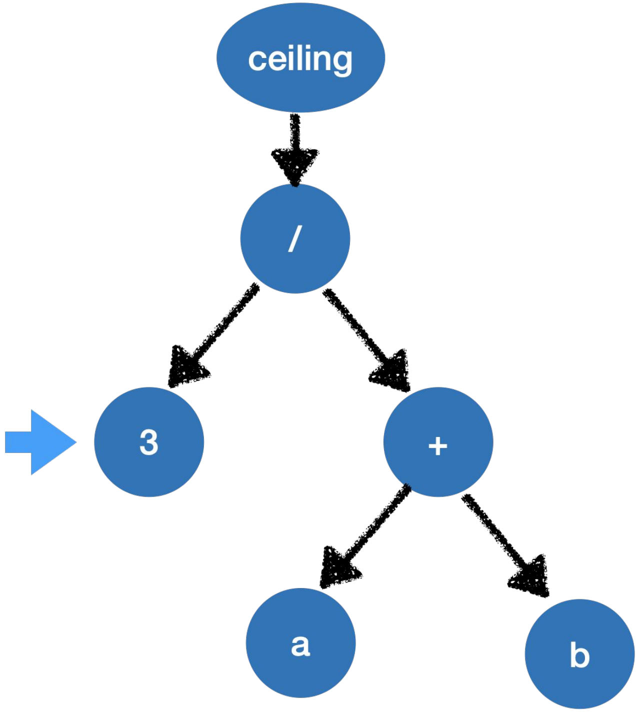


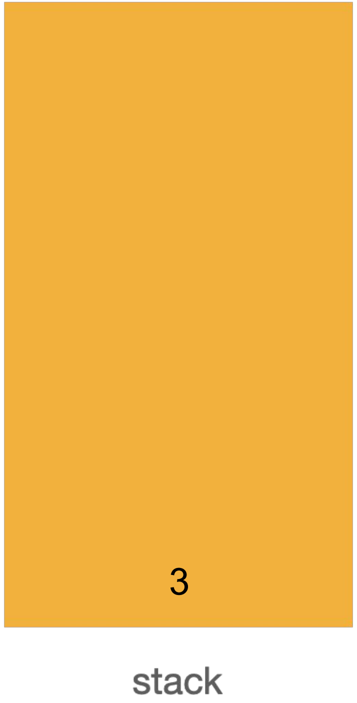
Interpreting the AST



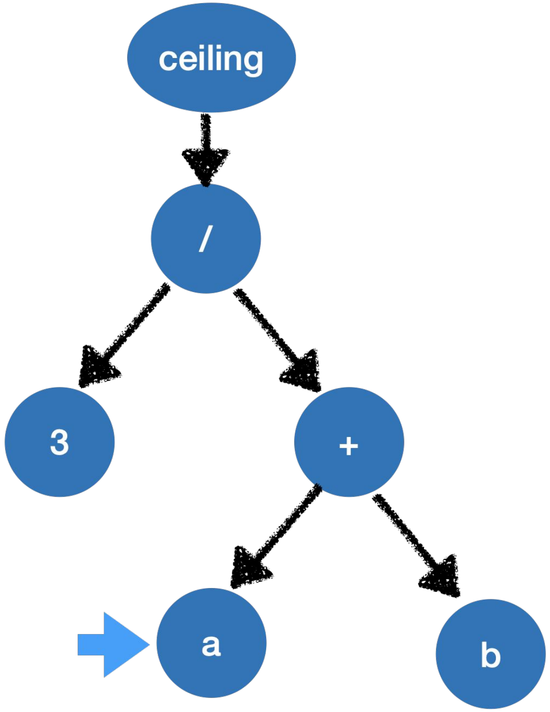


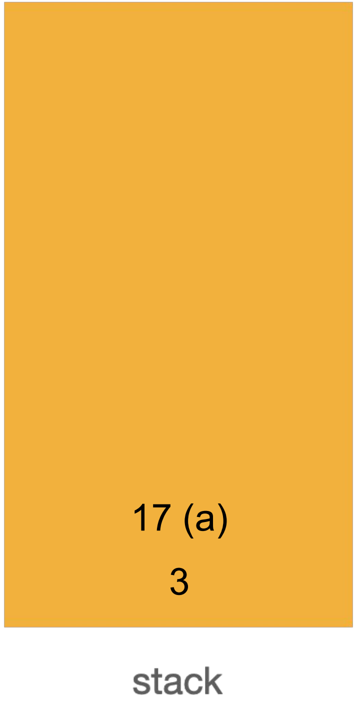
Interpreting the AST



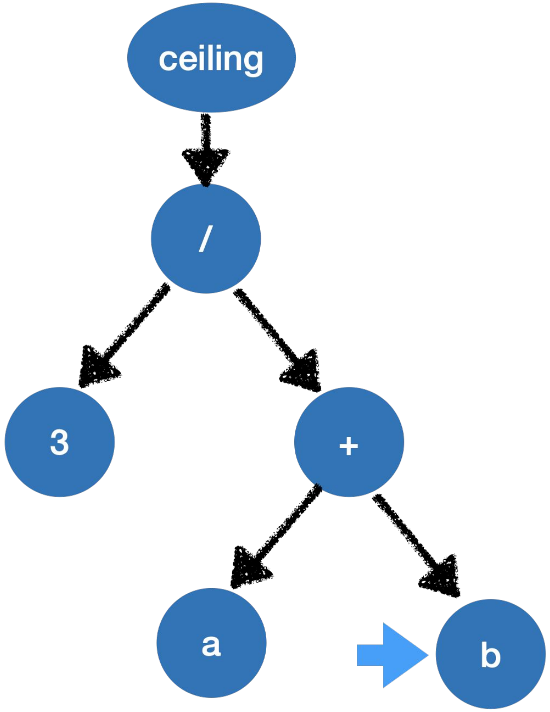


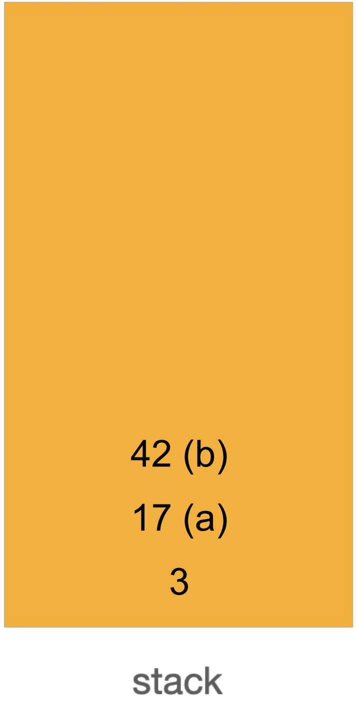
Interpreting the AST



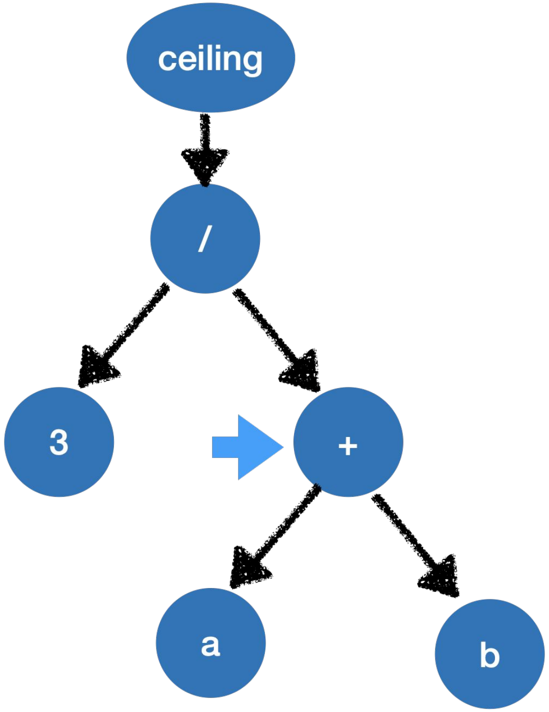


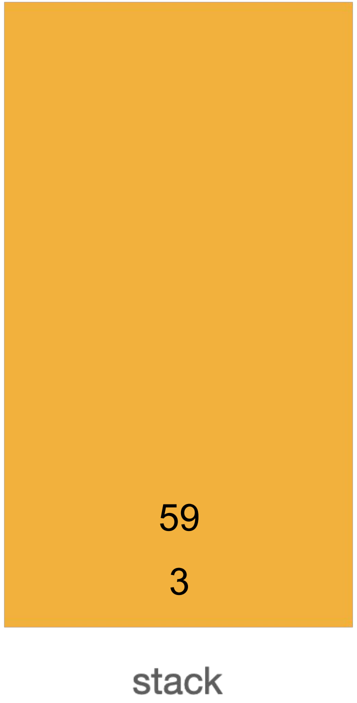
Interpreting the AST



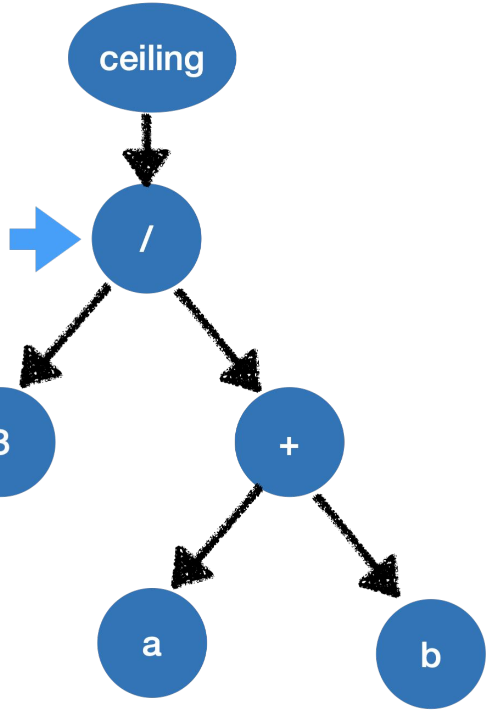


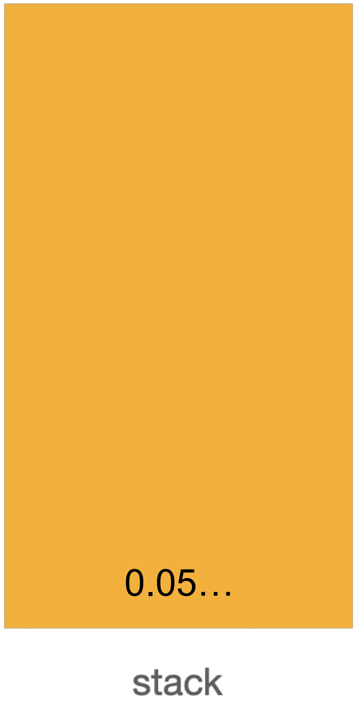
Interpreting the AST



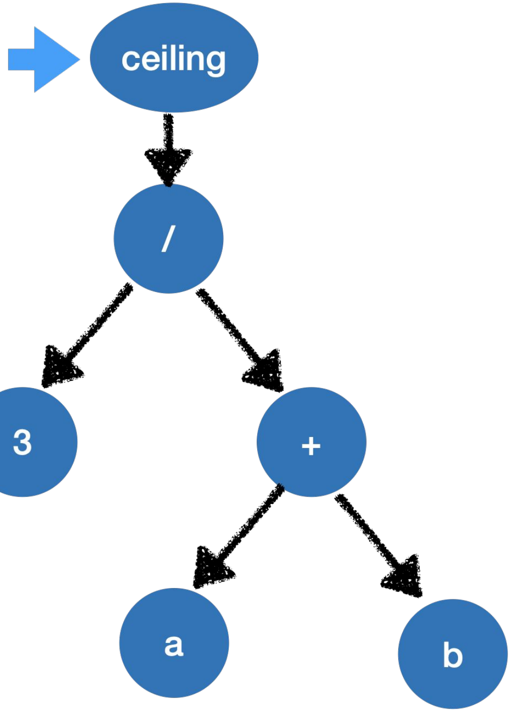


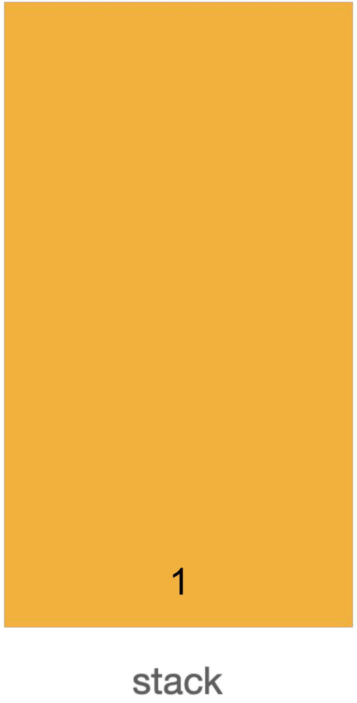
Interpreting the AST





Interpreting the AST

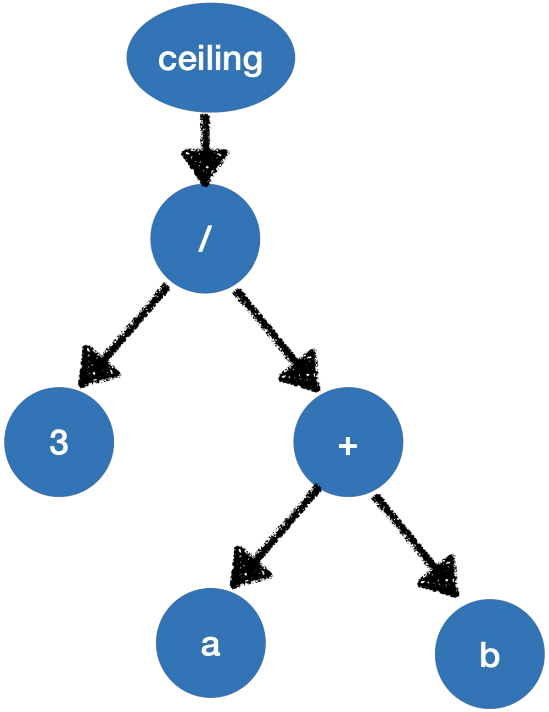




Are we done?

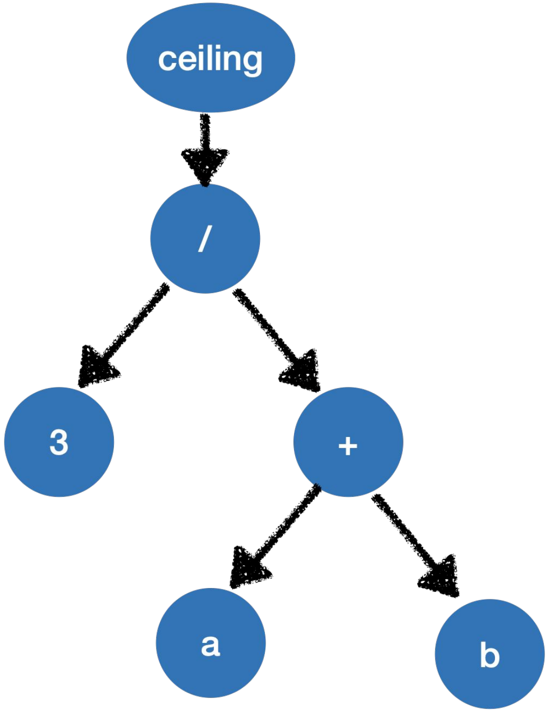
```
a := 1.
for (condition) {
   a := a + 6.
}
^ a + 2.
```

Compilation




```
FADD d0, d0, d1
FMOV d1, #3
FDIV d0, d1, d0
FRINTP d0, d0
RET
```

Why don't we just compile?

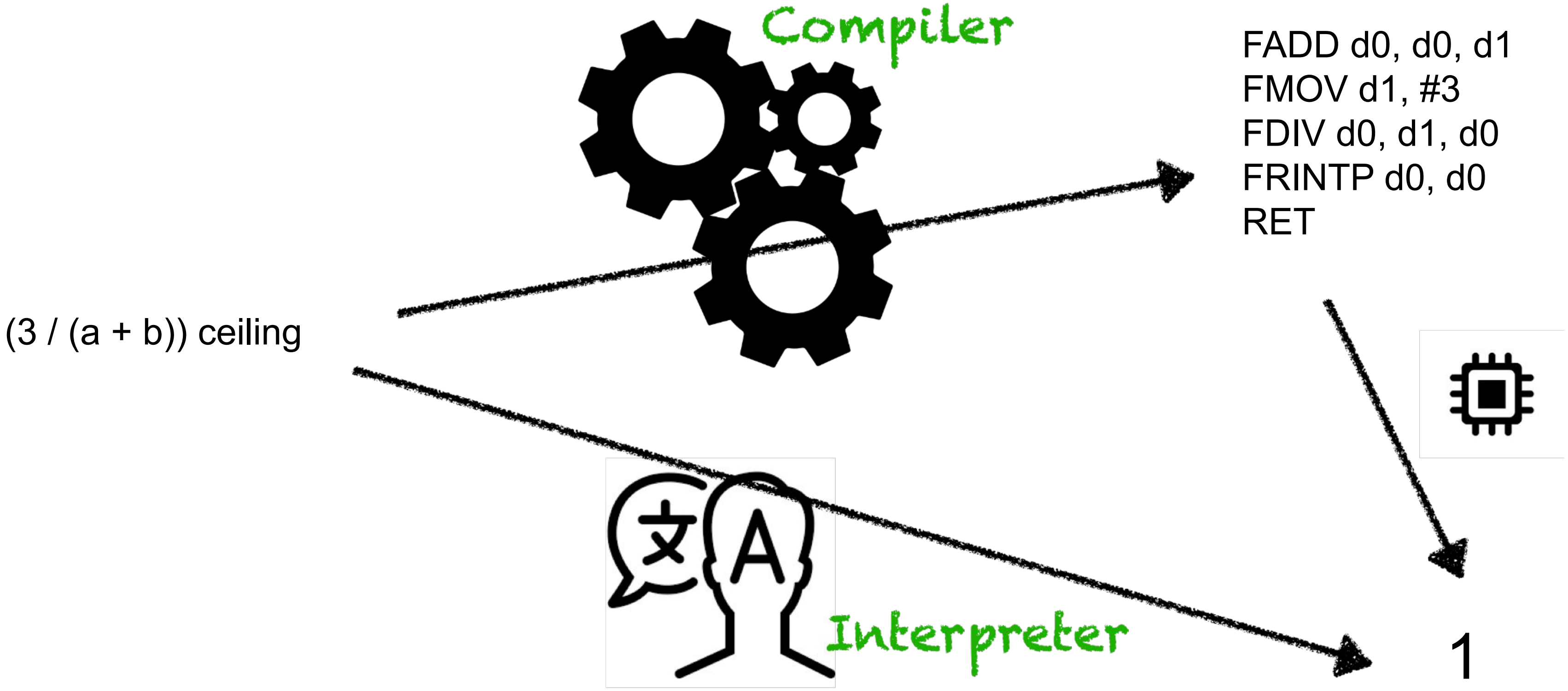


ADDSD xmm0, xmm1
MOVSD xmm1, #3
DIVSD xmm1, xmm0
(...)

FADD d0, d0, d1
FMOV d1, #3
FDIV d0, d1, d0
FRINTP d0, d0
RET

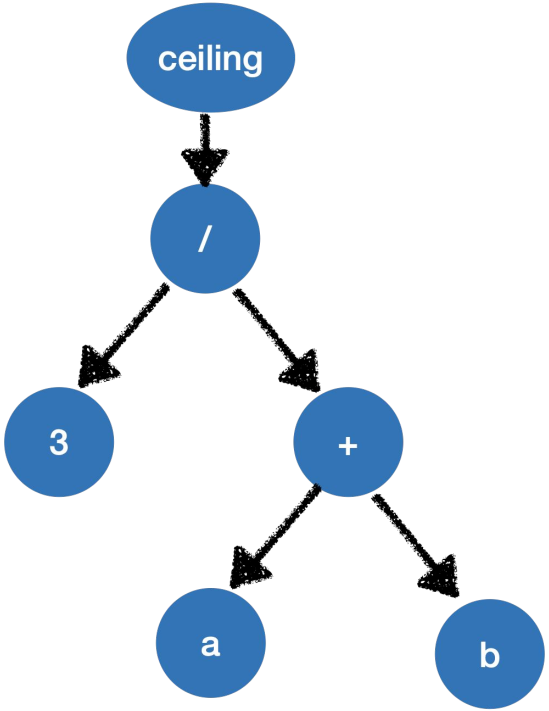
FADD.D ft1, fa0, fa1
FDIV.D fa0, ft0, ft1
CALL ceil@plt
(...)

Interpreter vs compiler



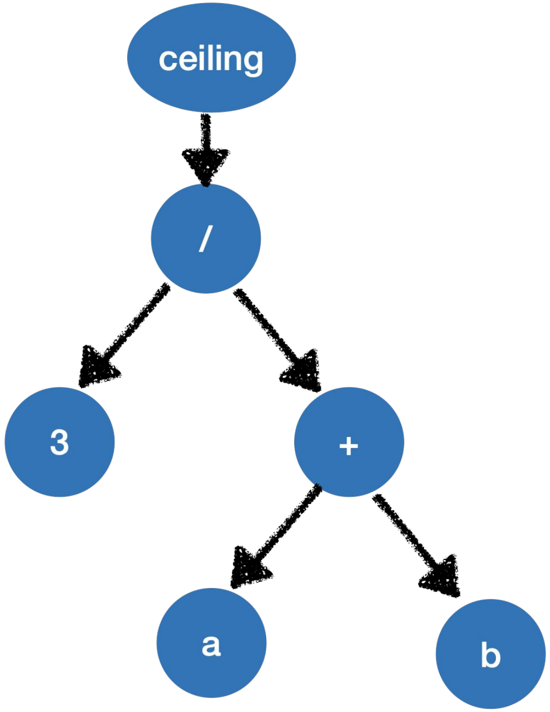
Can we combine both strategies?

Bytecode



FADD d0, d0, d1
FMOV d1, #3
FDIV d0, d1, d0
FRINTP d0, d0
RET

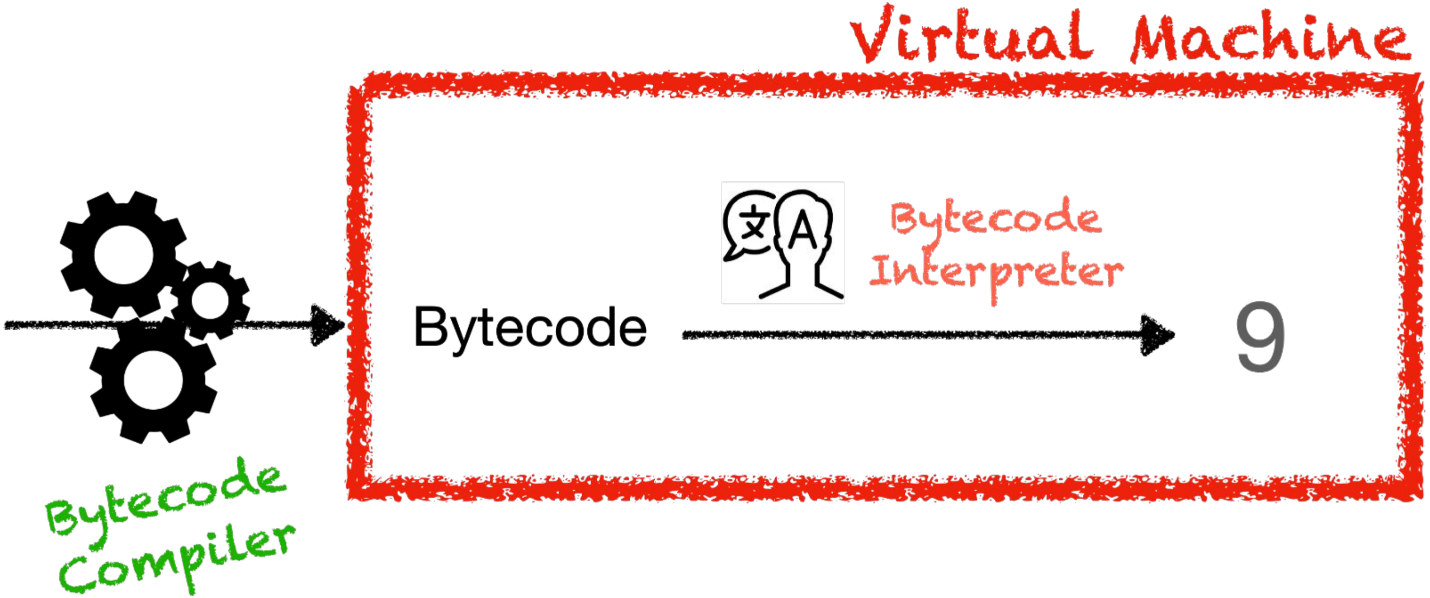
Bytecode



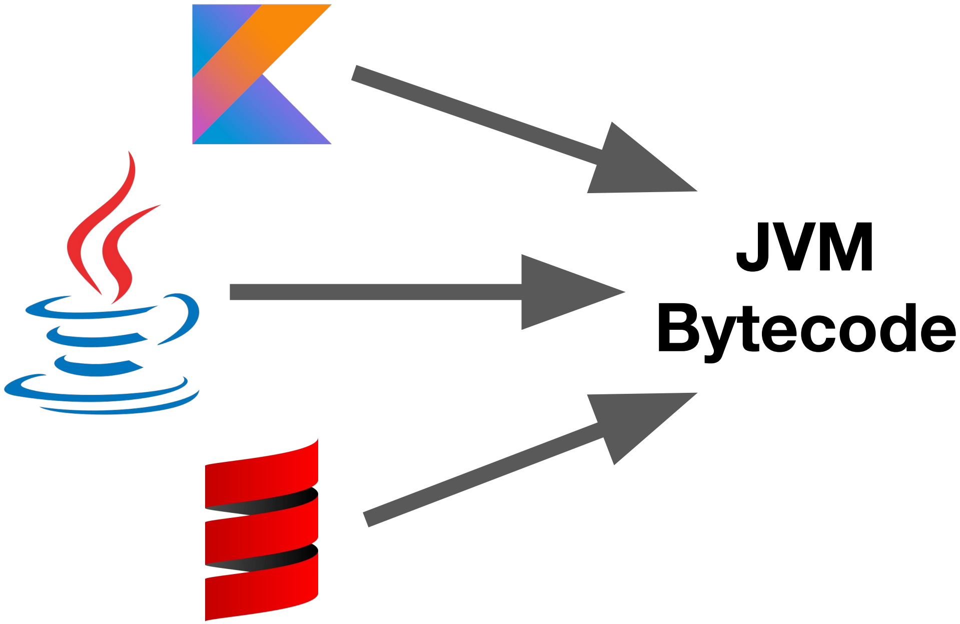
push 3	(17)
push a	(32)
push b	(33)
send +	(55)
send /	(56)
send ceiling	(48)

Bytecode

```
a := 1;
if (condition) {
  a := a + 6;
}
return a + 2;
```



Bytecode as compilation target

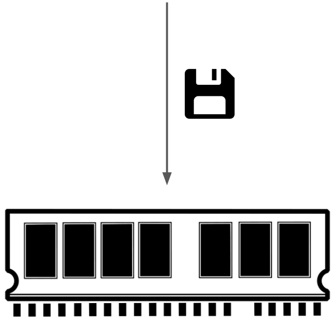


Can we go even further? => JIT compilation

```
someOperationBetween: a and: b
   ^ (3 / (a + b)) ceiling

arraySum: anArray
   sum := 0.
   a := 5
   1 to: anArray size do:
      [ :b | sum := sum + someOperationBetween: a and: b ].
   ^ sum
```

```
push 3
push a
push b
send +
send /
send ceiling
```

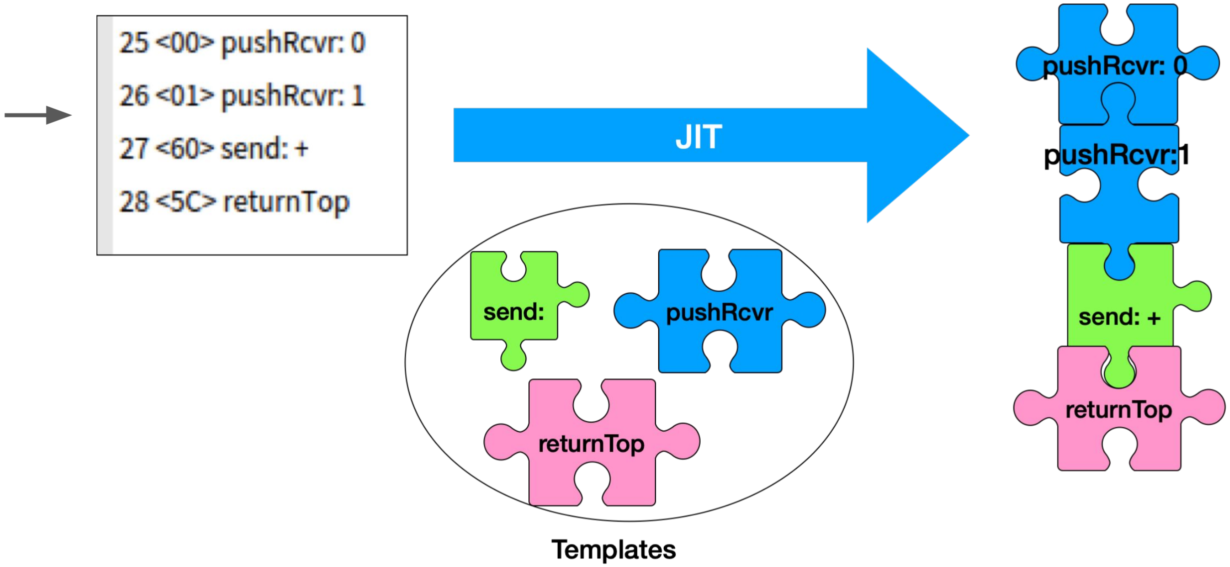
```
FADD d0, d0, d1
FMOV d1, #3
FDIV d0, d1, d0
FRINTP d0, d0
RET
```



Code cache

Just In Time compilers

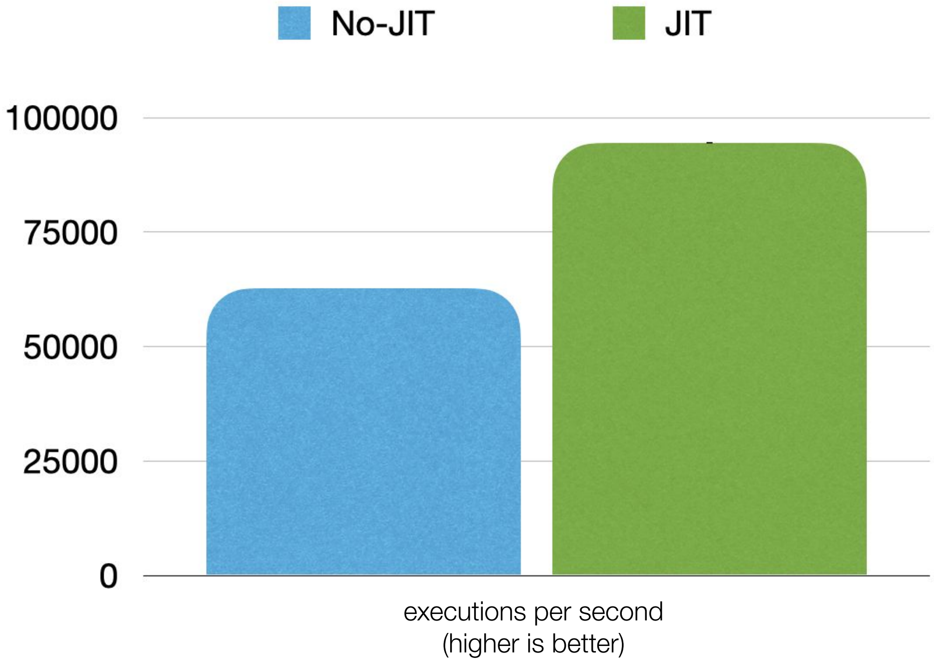
Baseline compiler



Optimizing compiler

Just In Time compilers

[100 factorial] bench



Just In Time compilers

Baseline compiler

Optimizing compiler →

```
a := 30.
b := a * 4.

(a > 10) ifTrue: [
   b := b - 10.
].
^ b * (60 / a)
```

Just In Time compilers

Baseline compiler

Optimizing compiler →

```
a := 30.
b := 30 * 4.

(30 > 10) ifTrue: [
   b := b - 10.
].
^ b * (60 / 30)
```

Constant propagation

Just In Time compilers

Baseline compiler

Optimizing compiler →

```
a := 30.
b := 120.

(true) ifTrue: [
   b := b - 10.
].
^ b * 2
```

Constant folding

Just In Time compilers

Baseline compiler

Optimizing compiler →

```
~~a := 30.~~
b := 120.

(true) ifTrue: [
   b := b - 10.
].
^ b * 2
```

Dead code elimination

Just In Time compilers

Baseline compiler

Optimizing compiler →

```
b := 120.

(true) ifTrue: [
   b := b - 10.
].
^ b * 2
```

Method inlining

Just In Time compilers

Baseline compiler

Optimizing compiler →

```
b := 110.

^ b * 2
```

Constant propagation + folding

Just In Time compilers

Baseline compiler

Optimizing compiler → ^ 220

Constant propagation + folding

Just In Time compilers

Baseline compiler

Optimizing compiler →

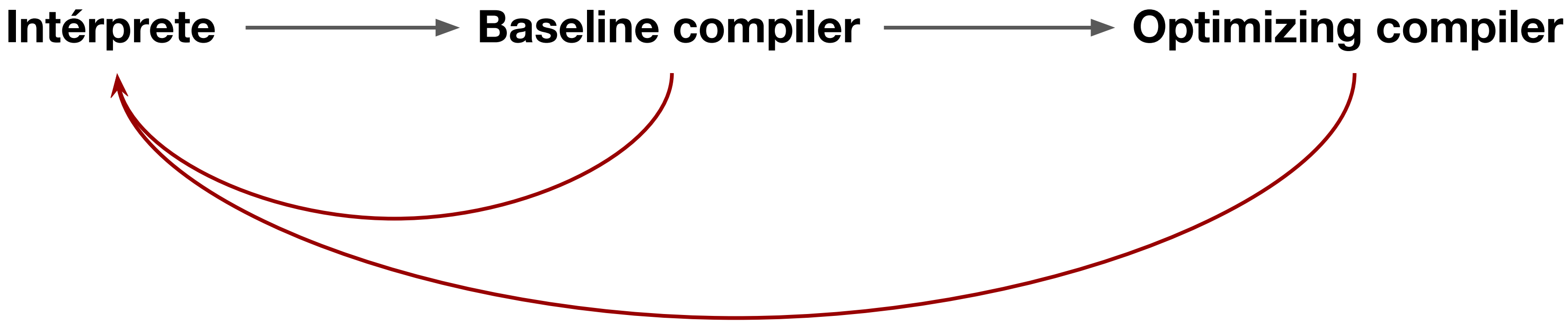
Speculative optimizations

```
push 3
push a
push b
send +
send /
send ceiling
```

⇒

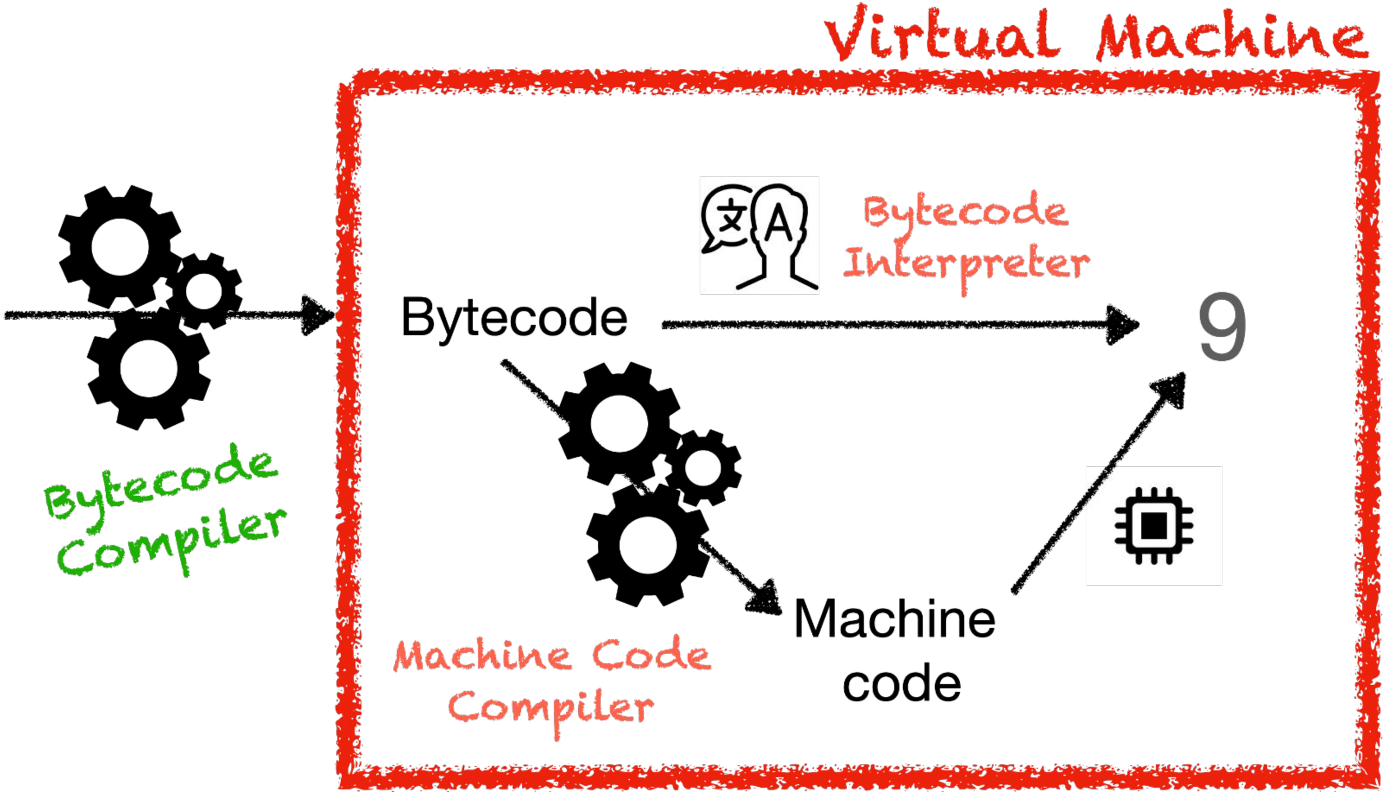
```
if (a or b are not Float)
  ^ deoptimize()

FADD d0, d0, d1
FMOV d1, #3
FDIV d0, d1, d0
FRINTP d0, d0
RET
```

Just In Time compilers



Final architecture

```
a := 1;
if (condition) {
  a := a + 6;
}
return a + 2;
```

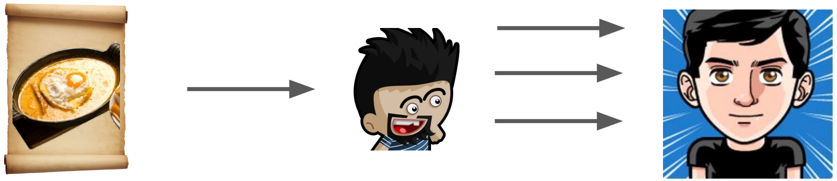


Recap

Interpreters

AST interpreter

Bytecode interpreter



Compilers

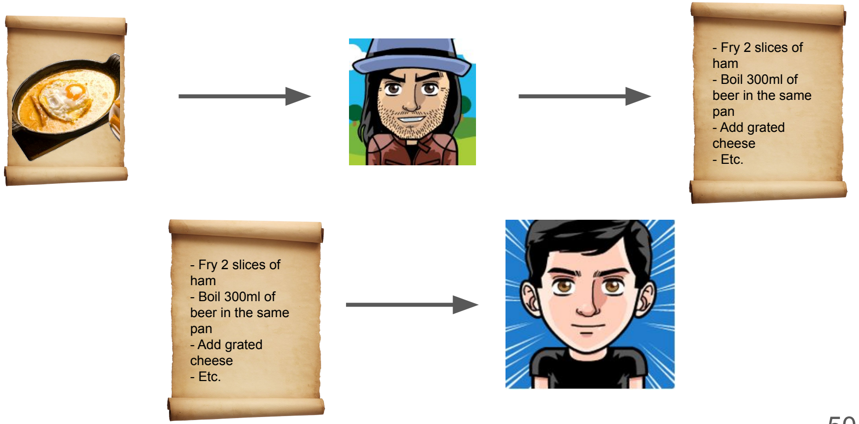
Compiling to machine code (ahead-of-time)

Compiling to bytecode

Compiling to machine code (just-in-time)

Baseline compilers

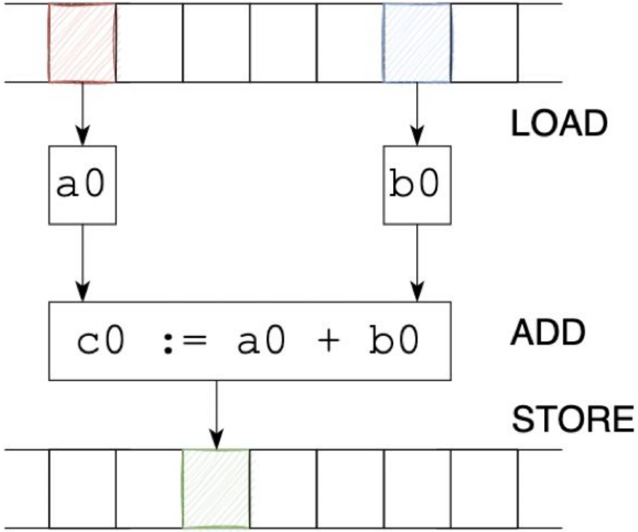
Optimizing compilers



Bonus:
What have I been doing?

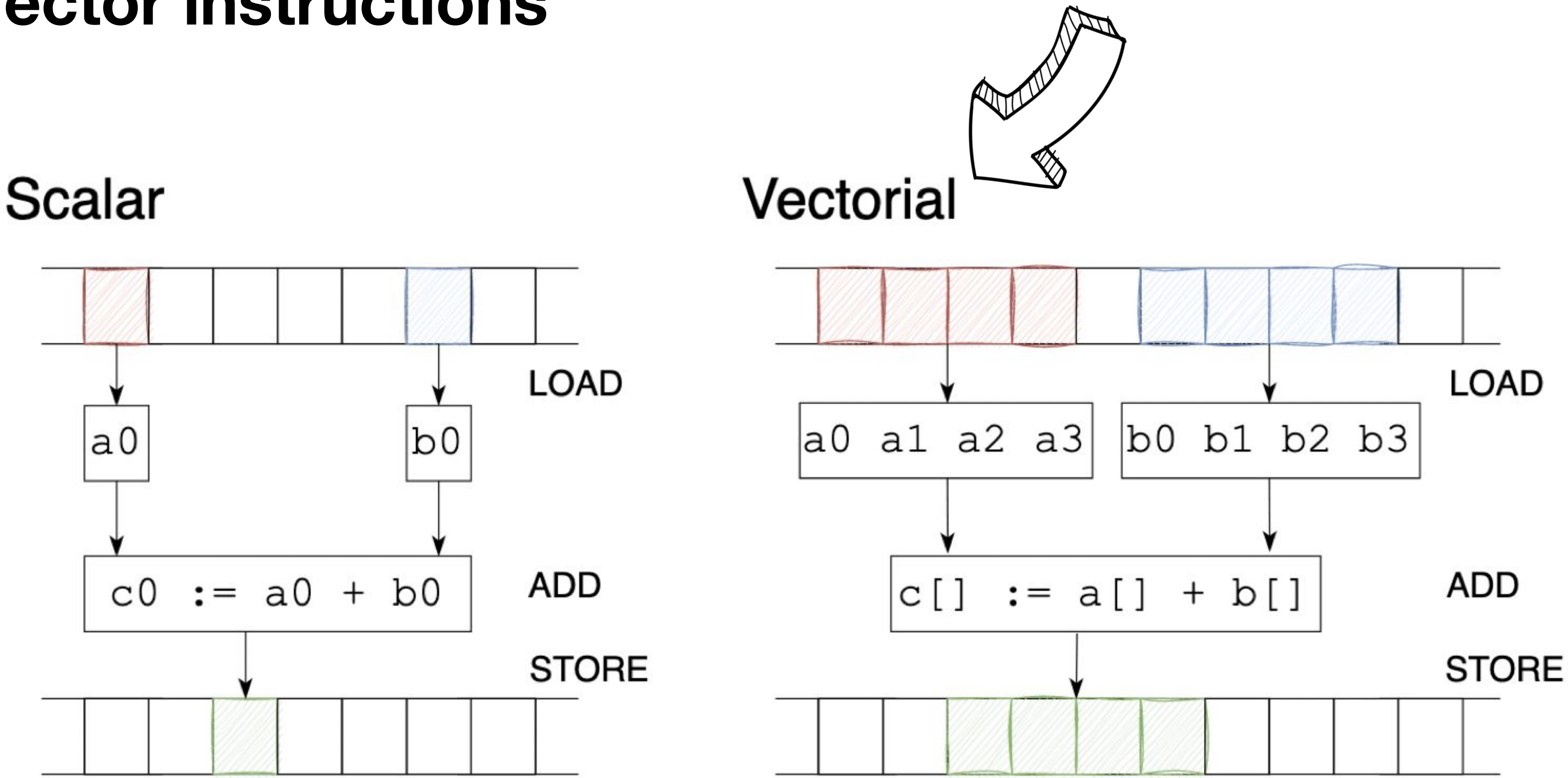
Loops are always a problem…

```
addArrays: term1 with: term2 intoArray: result

    ^ 1 to: term1 size do: [ :i |
      result at: i put: (term1 at: i) + (term2 at: i) ]
```




```
_scalarArrayAdd:
    mov x9, #0

check:
    cmp x9, x0
    b.eq exit

loop:
    ldr x10, [x1, x9, lsl #3]
    ldr x11, [x2, x9, lsl #3]
    add x12, x10, x11
    str x12, [x3, x9, lsl #3]
    add x9, x9, #1
    b check

exit:
    ret
```

Vector instructions



Vector instructions

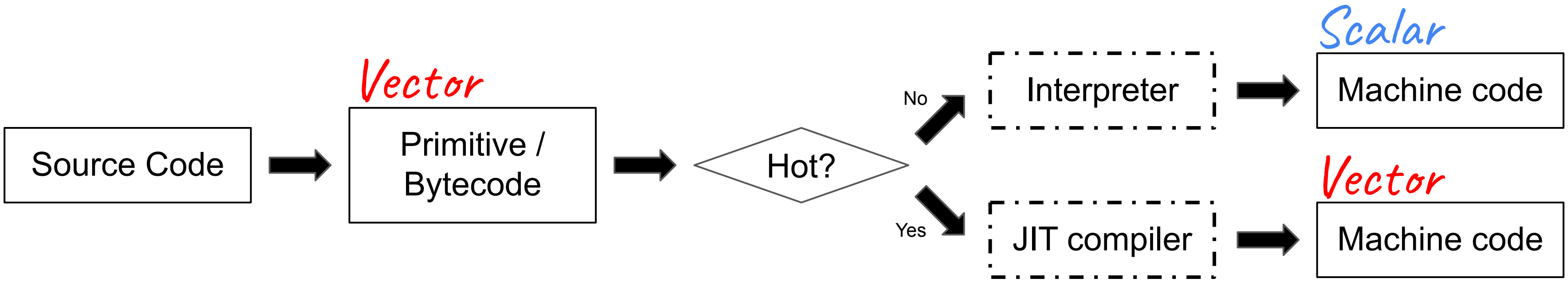
```
_scalarArrayAdd:
    mov x9, #0

check:
    cmp x9, x0
    b.eq exit

loop:
    ldr x10, [x1, x9, lsl #3]
    ldr x11, [x2, x9, lsl #3]
    add x12, x10, x11
    str x12, [x3, x9, lsl #3]
    add x9, x9, #1
    b check

exit:
    ret
```

```
_vectorialArrayAdd:
    lsr x0, x0, #2

loop:
    cmp x0, 0
    b.eq exit

    ld1 {v1.4s}, [x1], #16
    ld1 {v2.4s}, [x2], #16
    add v3.4s, v1.4s, v2.4s
    st1 {v3.4s}, [x3], #16

    sub x0, x0, #1
    b loop

exit:
    ret
```

SIMD Design Space

- VM Primitives
- Vectorized Bytecode

How are vector instructions generated in Pharo?



SIMD Design Space

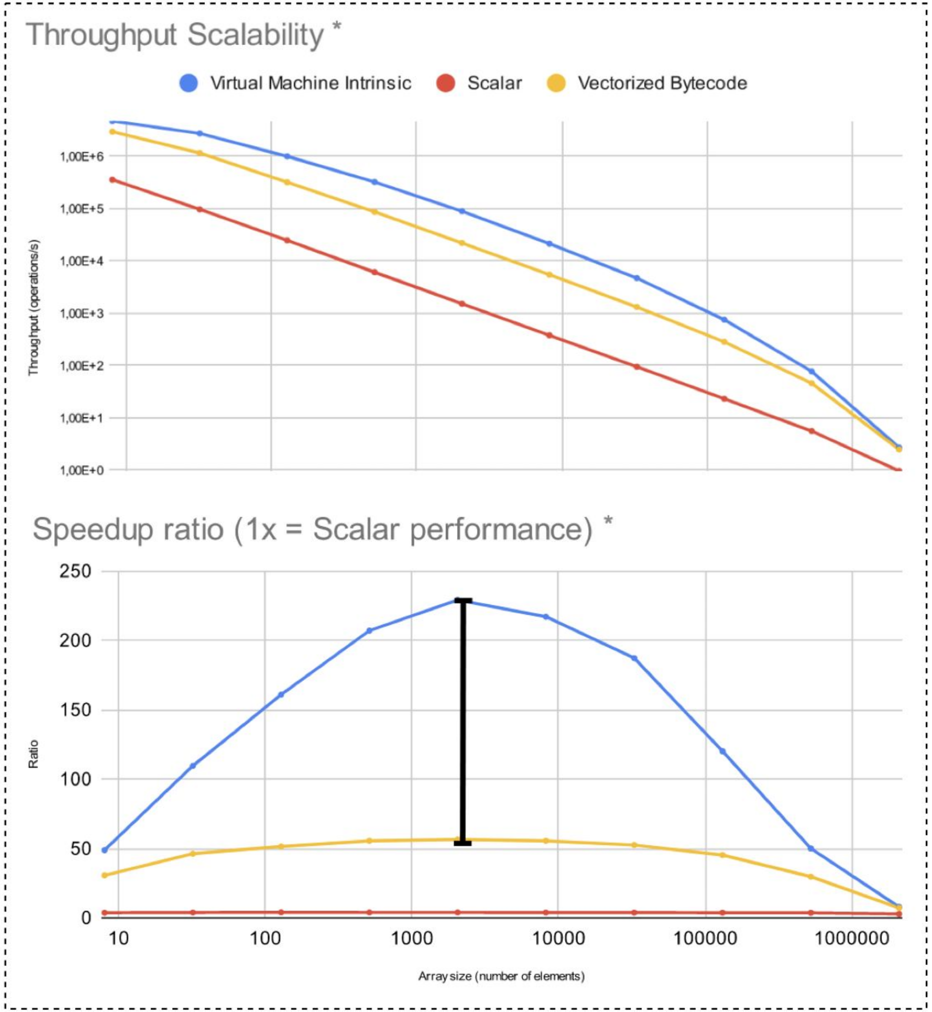
- VM Primitives
  - **Specialized**
  - Faster, less checks
- Vectorized Bytecode

SIMD Design Space

- VM Primitives
  - **Specialized**
  - Faster, less checks
- Vectorized Bytecode
  - **Composable**
  - Safe at the expense of speed

SIMD Design Space

- VM Primitives
  - **Specialized**
  - Faster, less checks
- Vectorized Bytecode
  - **Composable**
  - Safe at the expense of speed



What do we have today?

Optimized primitives for specific operations

- Object initialization → 2x faster with vector instructions

Arithmetic operations on arrays → testbed for primitives vs bytecodes

Open research

Can we have the best of both worlds?

- Composability
- Performance

Performant vectorized bytecode

Thanks!

Nico Rainhart
RMoD - September 2022