Introducing the Cogit JIT compiler

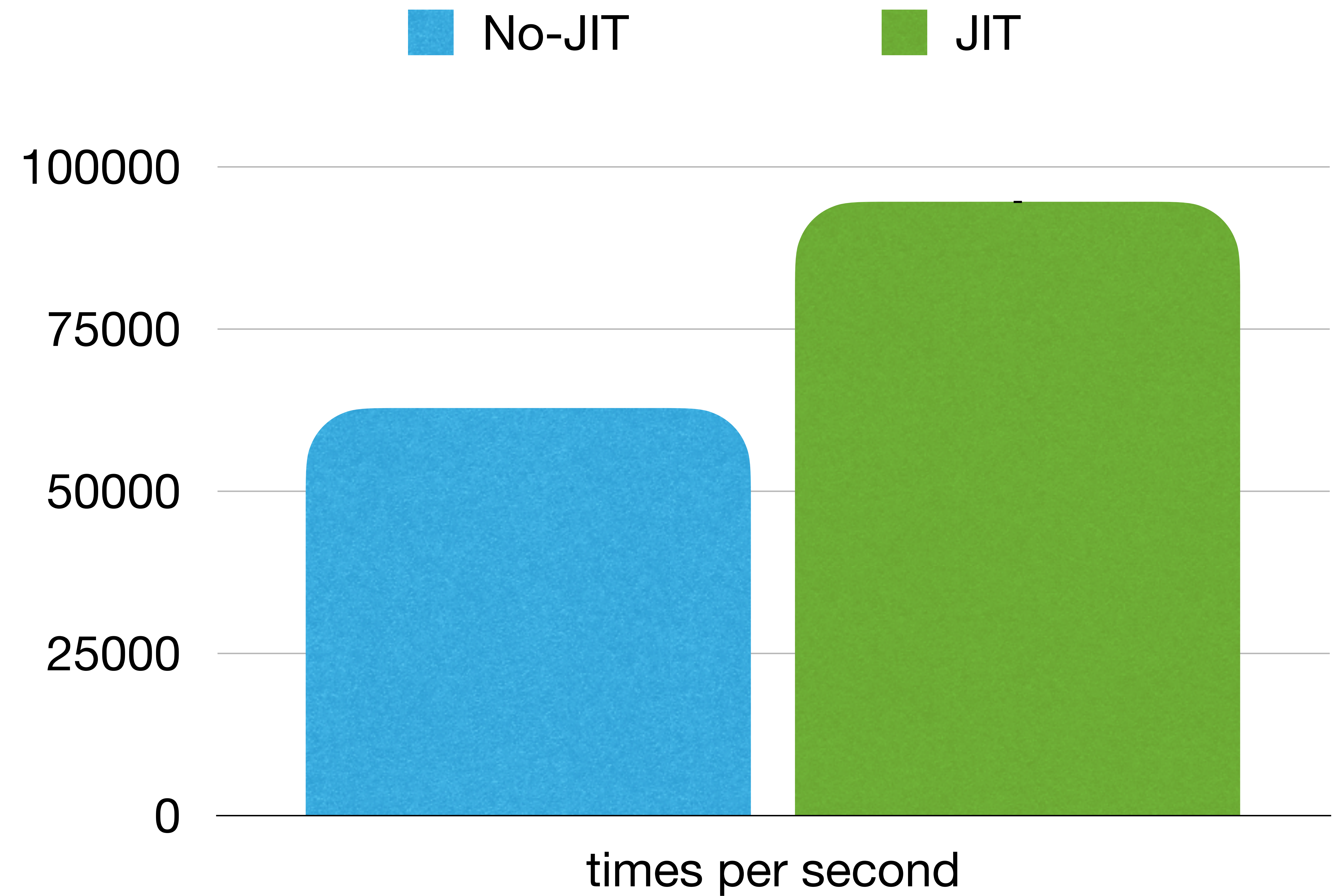
Context: What is the Cogit JIT compiler
(pre-condition: see the Compiler 101 presentation)

- A compiler => translates bytecode to machine code
- Comes with the Pharo VM
- JIT = Just In Time
  - Compilation at run-time
  - Compiles and optimizes when it finds a *hot* method
  - Look for balance between time spent compiling and executing

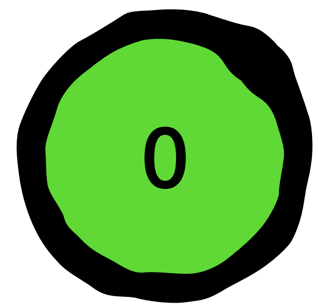
Why a JIT
(higher is better)

`[100 factorial] bench`

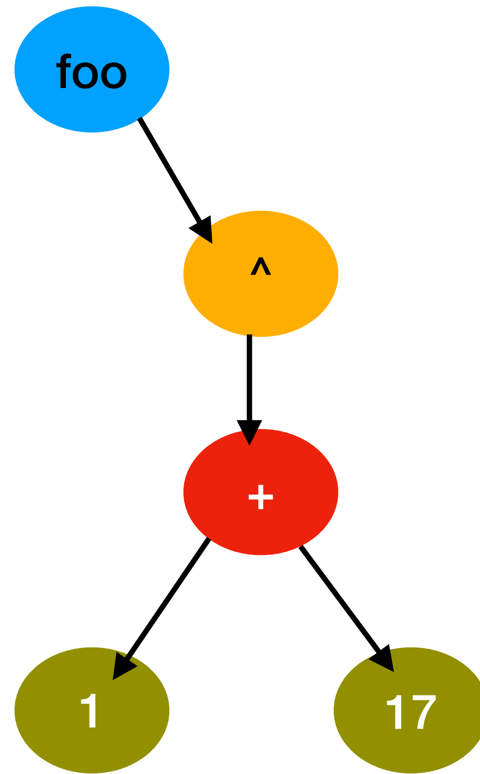
*this is not a serious benchmark, this is just illustrative



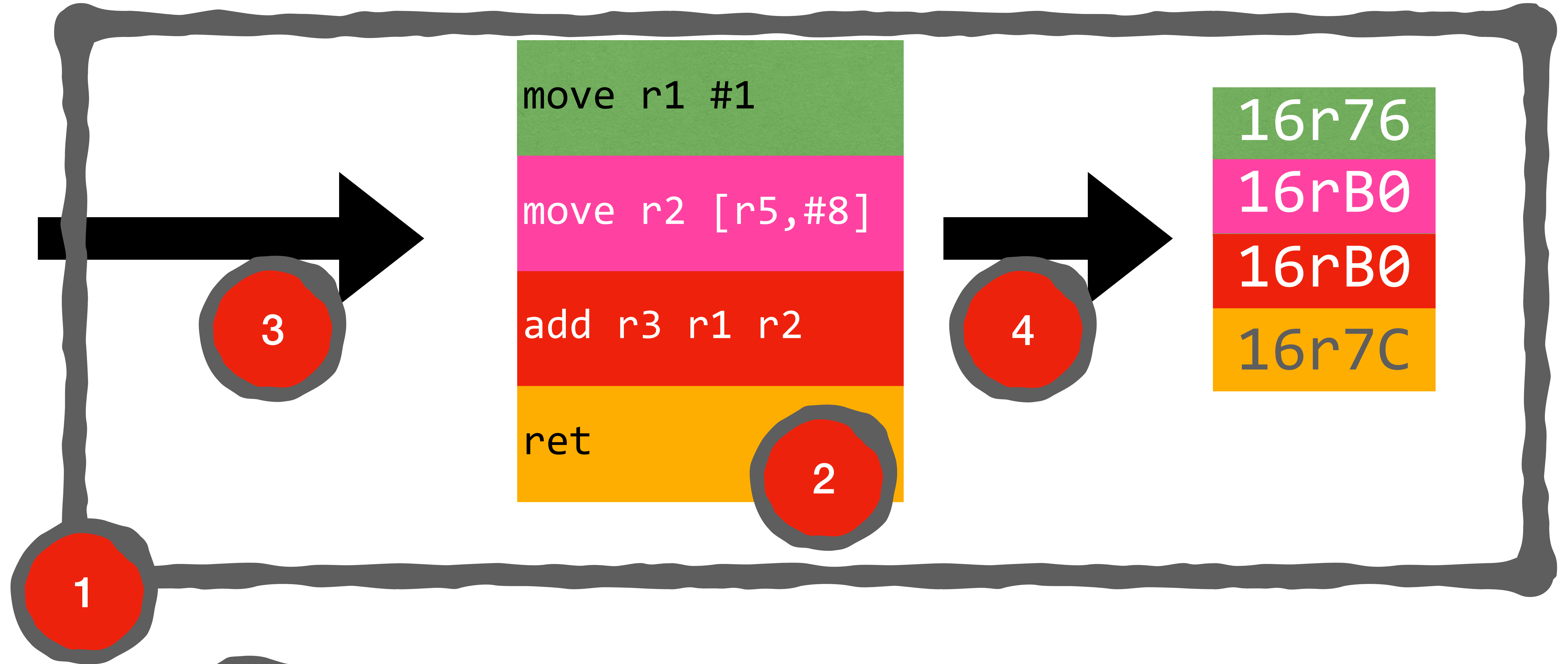
Plan: The Cogit Architecture

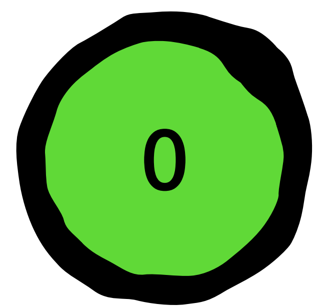



```
MyClass >> foo
    ^ 1 + 17
```

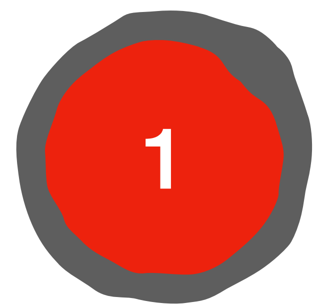



```
push 1
push 17
send +
returnTop
```

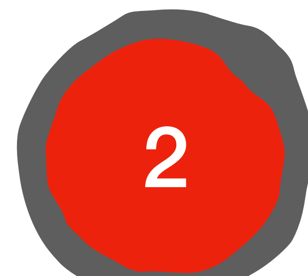




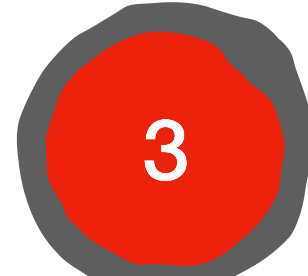
Bytecode Interpretation



The Cogit Architecture



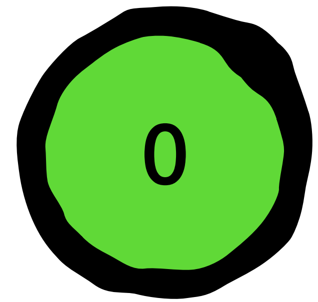
The CogRTL Intermediate Representation



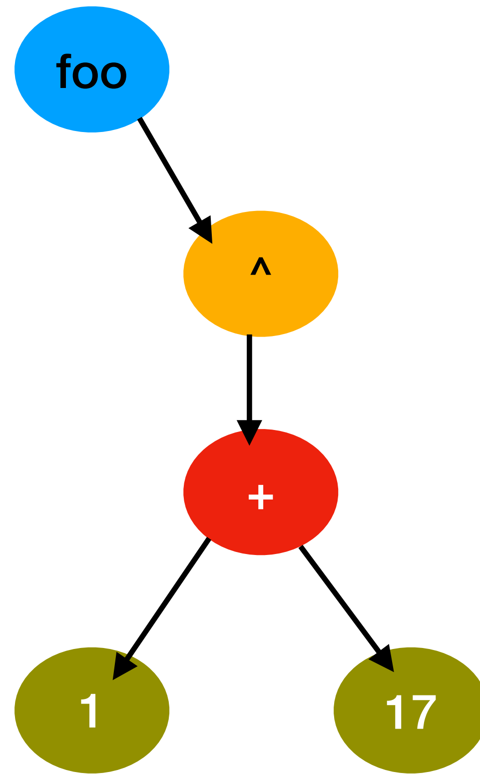
Bytecode-to-IR translation

IR-to-machine code translation

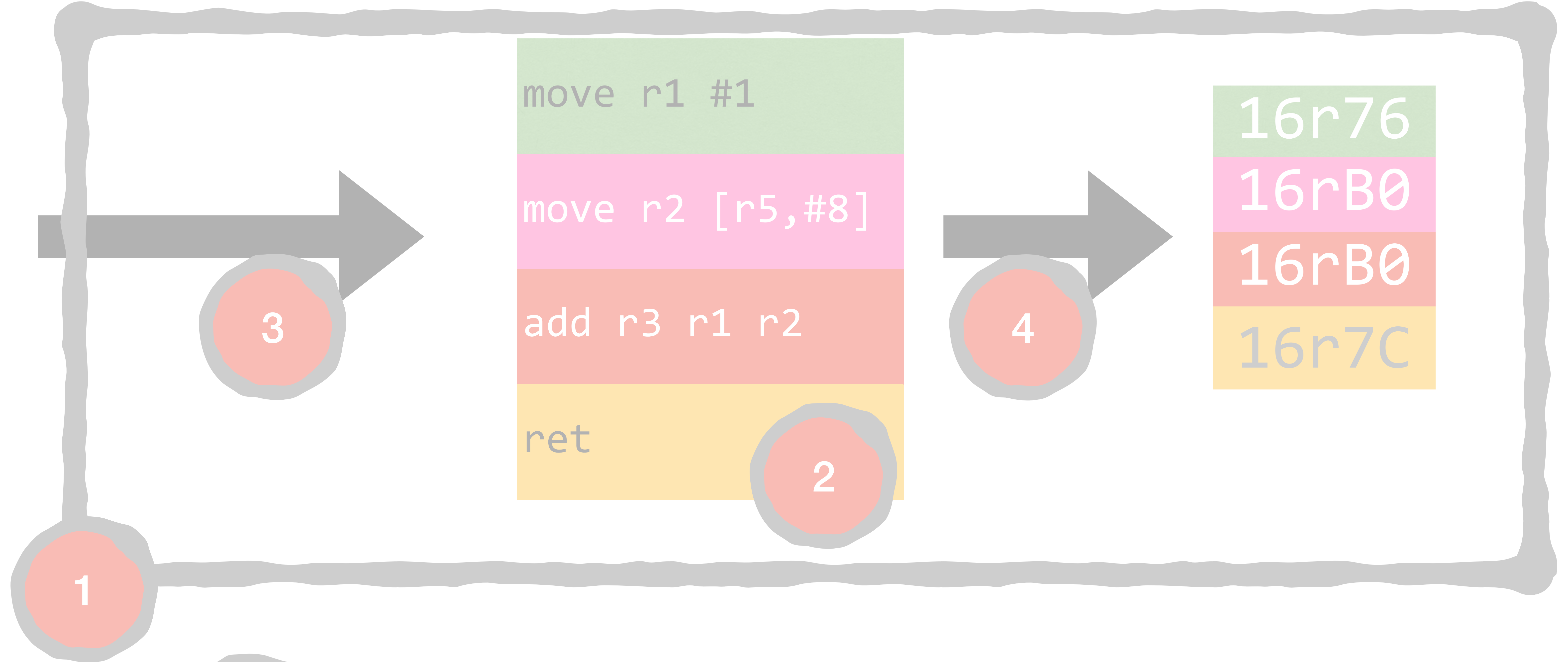
Bytecode interpretation

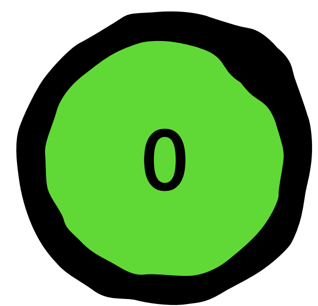



```
MyClass >> foo
    ^ 1 + 17
```

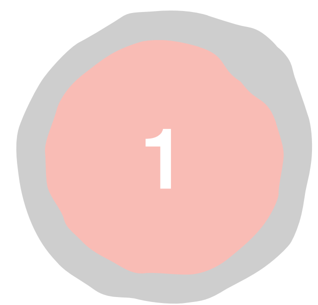



```
push 1
push 17
send +
returnTop
```

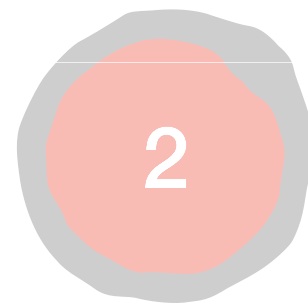




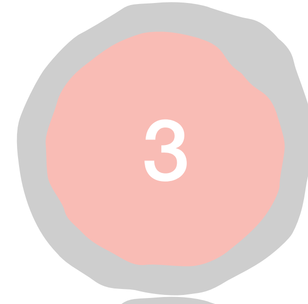
Bytecode Interpretation



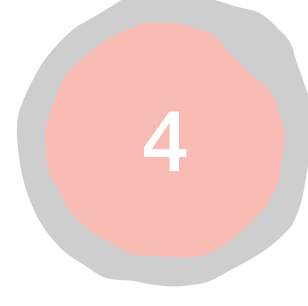
The Cogit Architecture



The CogRTL Intermediate Representation

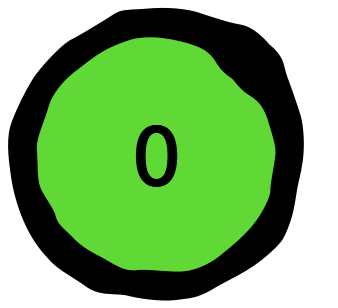


Bytecode-to-IR translation

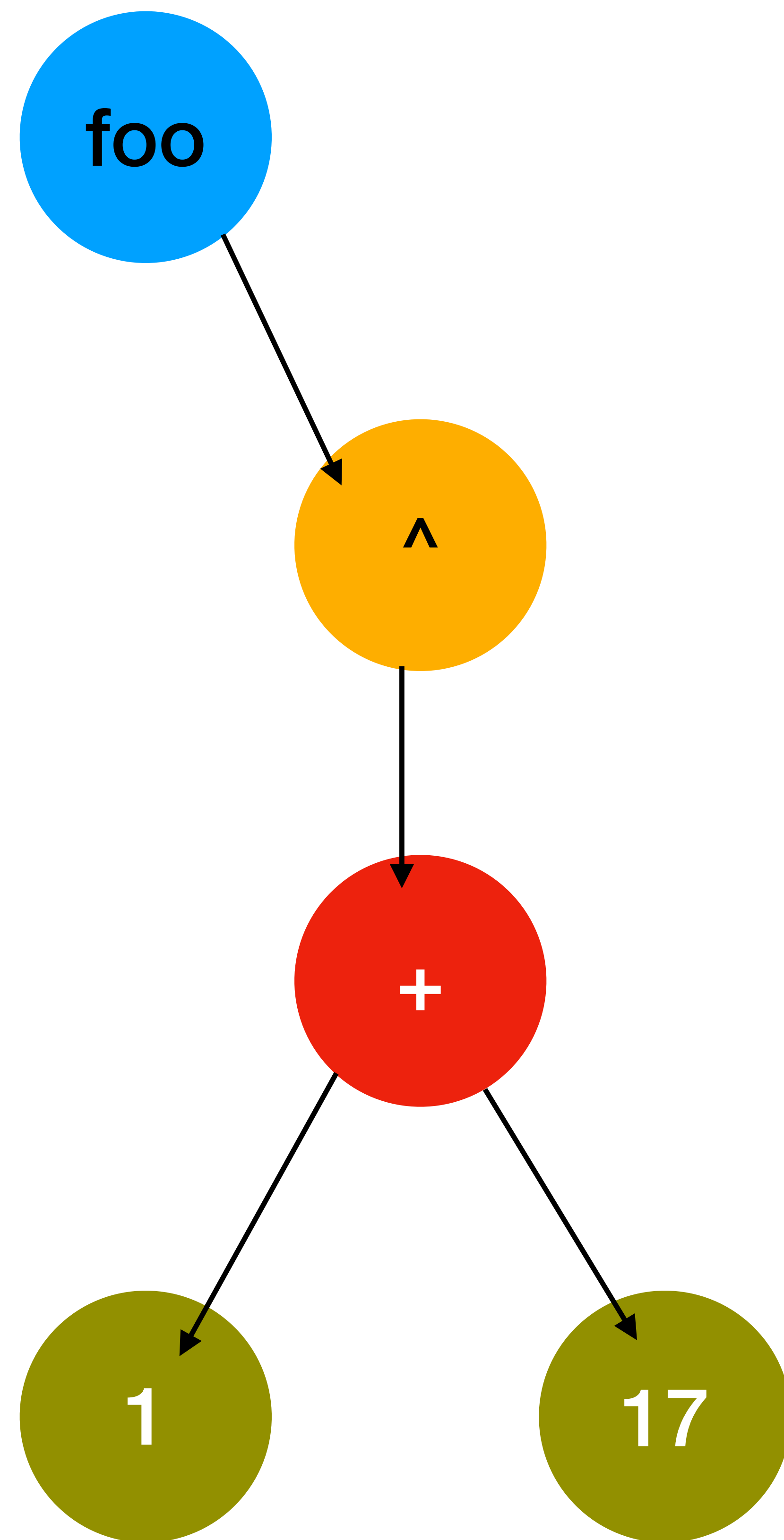


IR-to-machine code translation

Reminding the bytecode




```
MyClass >> foo
    ^ 1 + 17
```




```
push 1
push 17
send +
returnTop
```

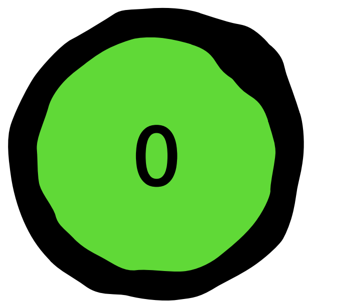
source code

AST

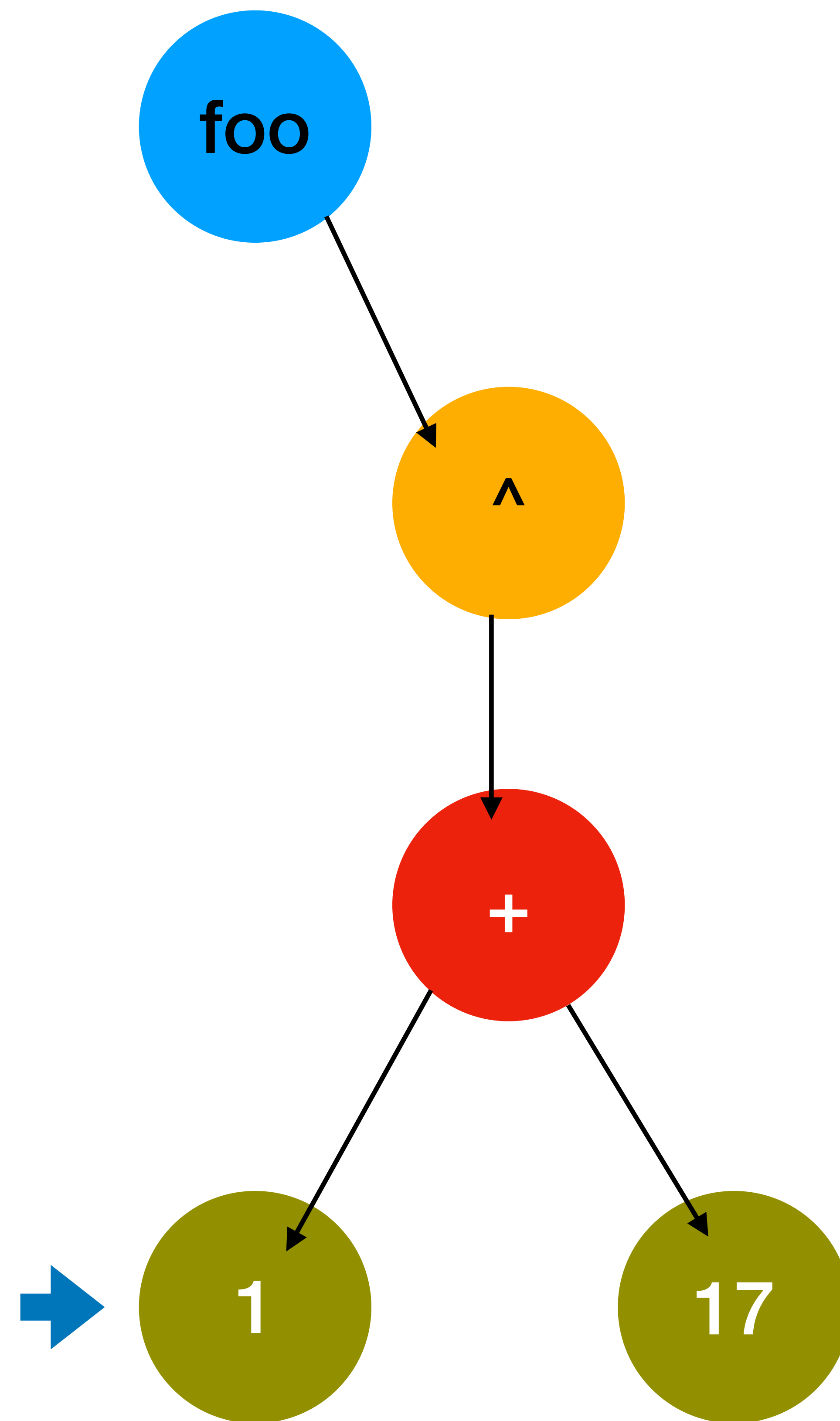
bytecode

value stack

Reminding the bytecode




```
MyClass >> foo
    ^ 1 + 17
```




```
push 1
push 17
send +
returnTop
```

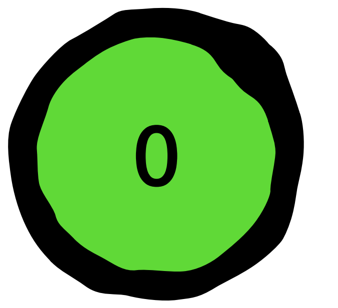
source code

AST

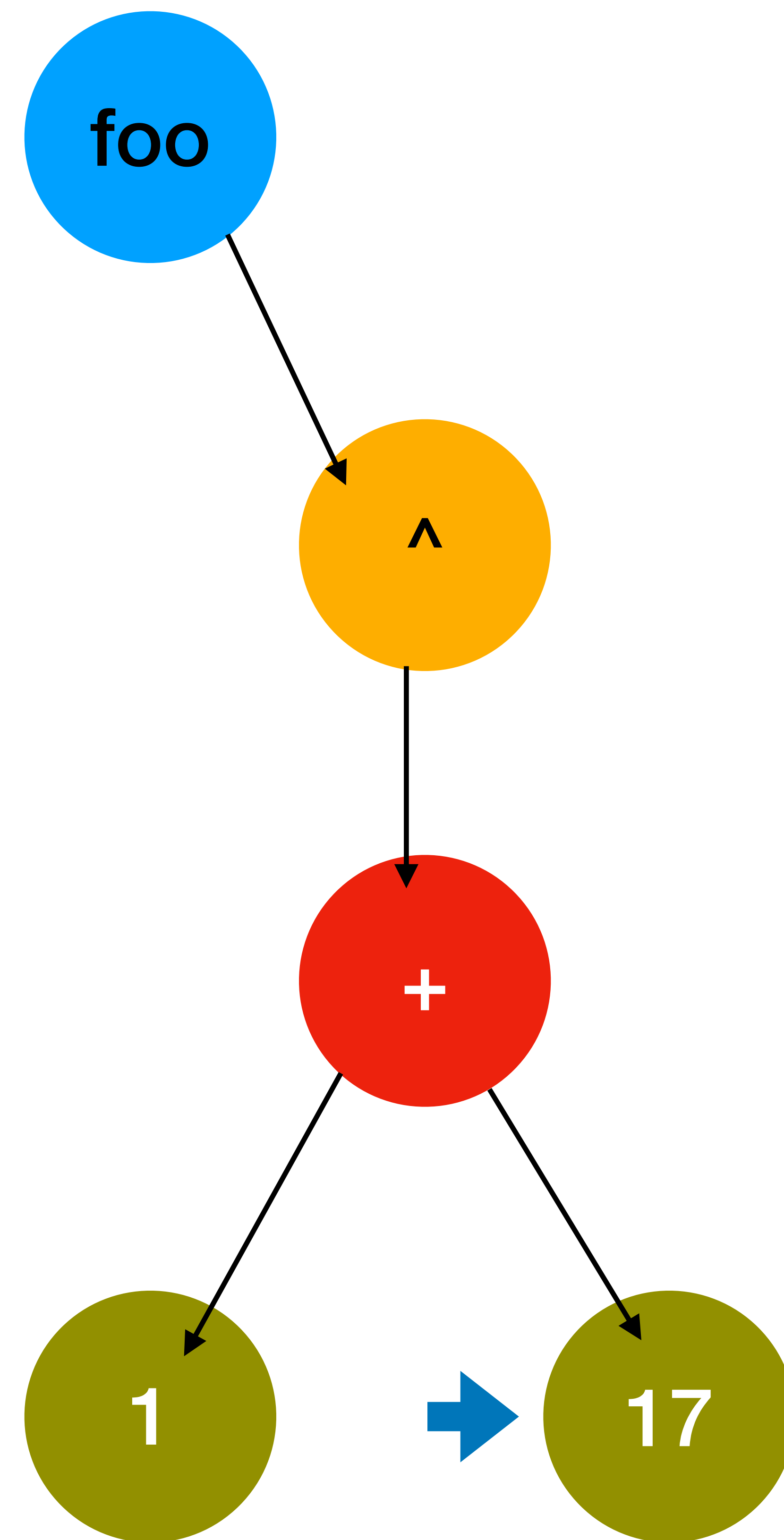
bytecode

value stack

Reminding the bytecode




```
MyClass >> foo
    ^ 1 + 17
```




```
push 1
push 17
send +
returnTop
```

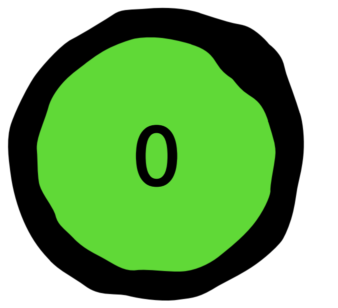
source code

AST

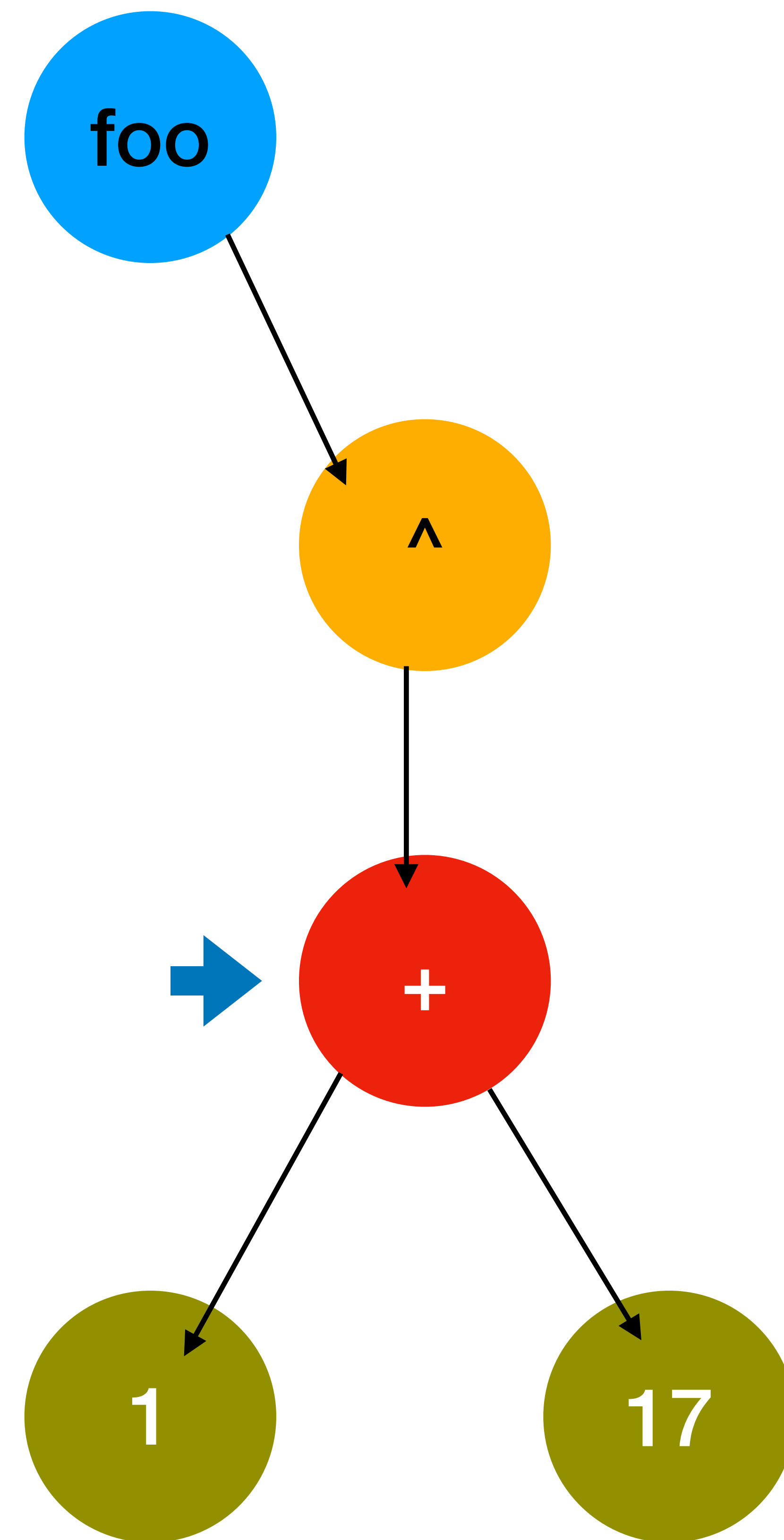
bytecode

value stack

Reminding the bytecode




```
MyClass >> foo
    ^ 1 + 17
```




```
push 1
push 17
send +
returnTop
```

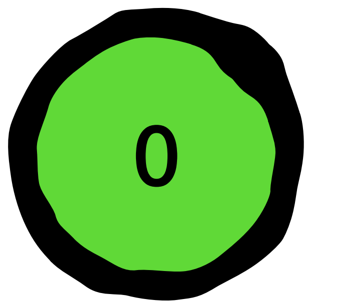
source code

AST

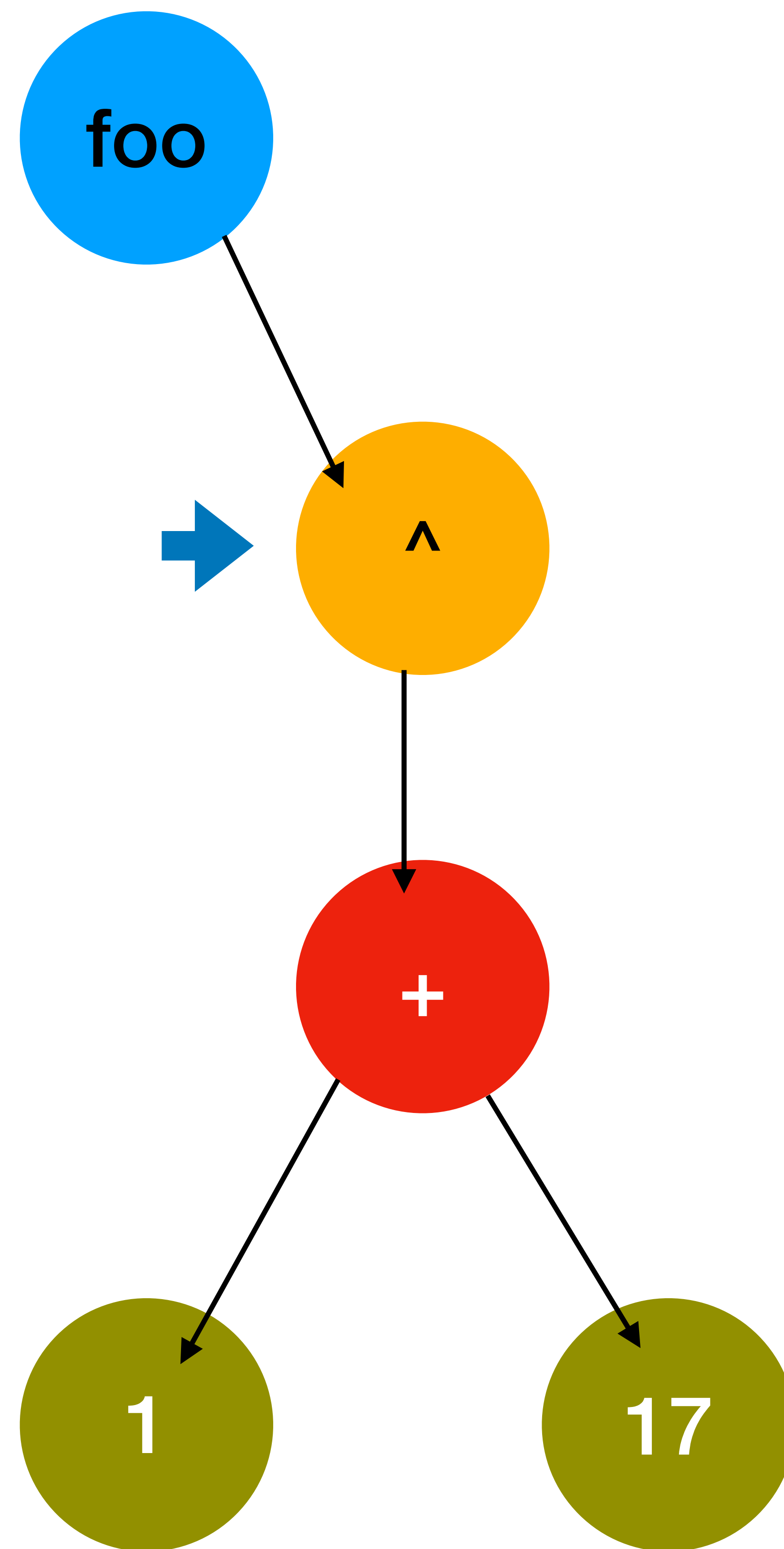
bytecode

value stack

Reminding the bytecode




```
MyClass >> foo
  ^ 1 + 17
```




```
push 1
push 17
send +
returnTop
```

source code

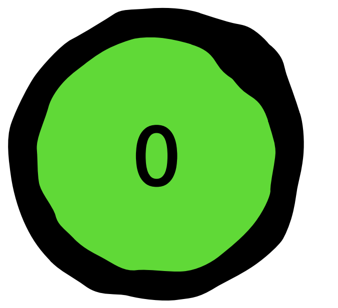
AST

bytecode

value stack

The Bytecode Interpreter

A table maps bytecode to functions




```
25 <76> pushConstant: 1
26 <20> pushConstant: 17
27 <B0> send: +
28 <7C> returnTop
```

bytecode

17
1

value stack

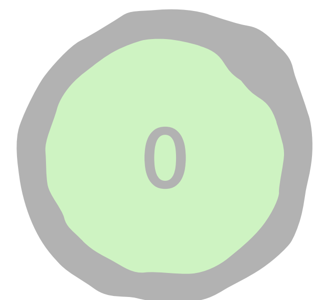
0x76	
0x20	
0xB0	
...	

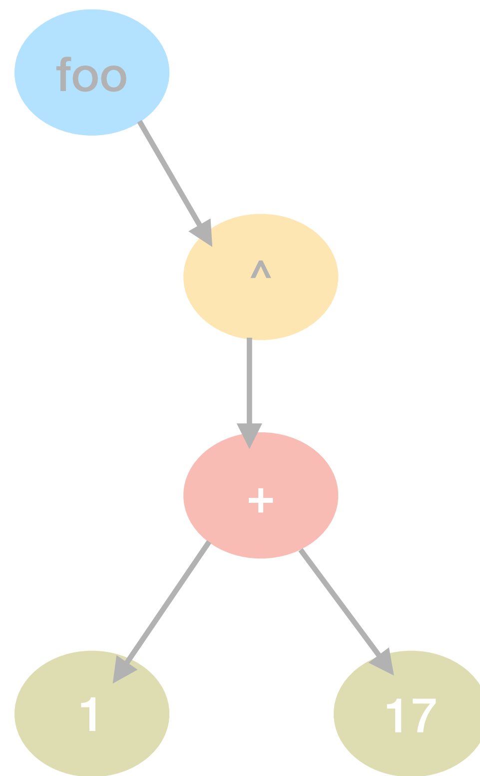
```
pushConstantOneBytecode

   self fetchNextBytecode.
   self internalPush: ConstOne.
```

```
pushLiteralConstantBytecode
   self fetchNextBytecode.
   self pushLiteralConstant:
      (currentBytecode bitAnd: 16r1F)
```

```
bytecodePrimAdd
   | rcvr arg result |
   rcvr := self internalStackValue: 1.
   arg := self internalStackValue: 0.
   (objectMemory areIntegers: rcvr and: arg) ifTrue: [
      result := (objectMemory integerValueOf: rcvr) +
         (objectMemory integerValueOf: arg).
      (objectMemory isIntegerValue: result) ifTrue:
         [self
            internalPop: 2
            thenPush: (objectMemory integerObjectOf: result)].
   …
   self successful ifTrue: [^ self fetchNextBytecode ]].
```

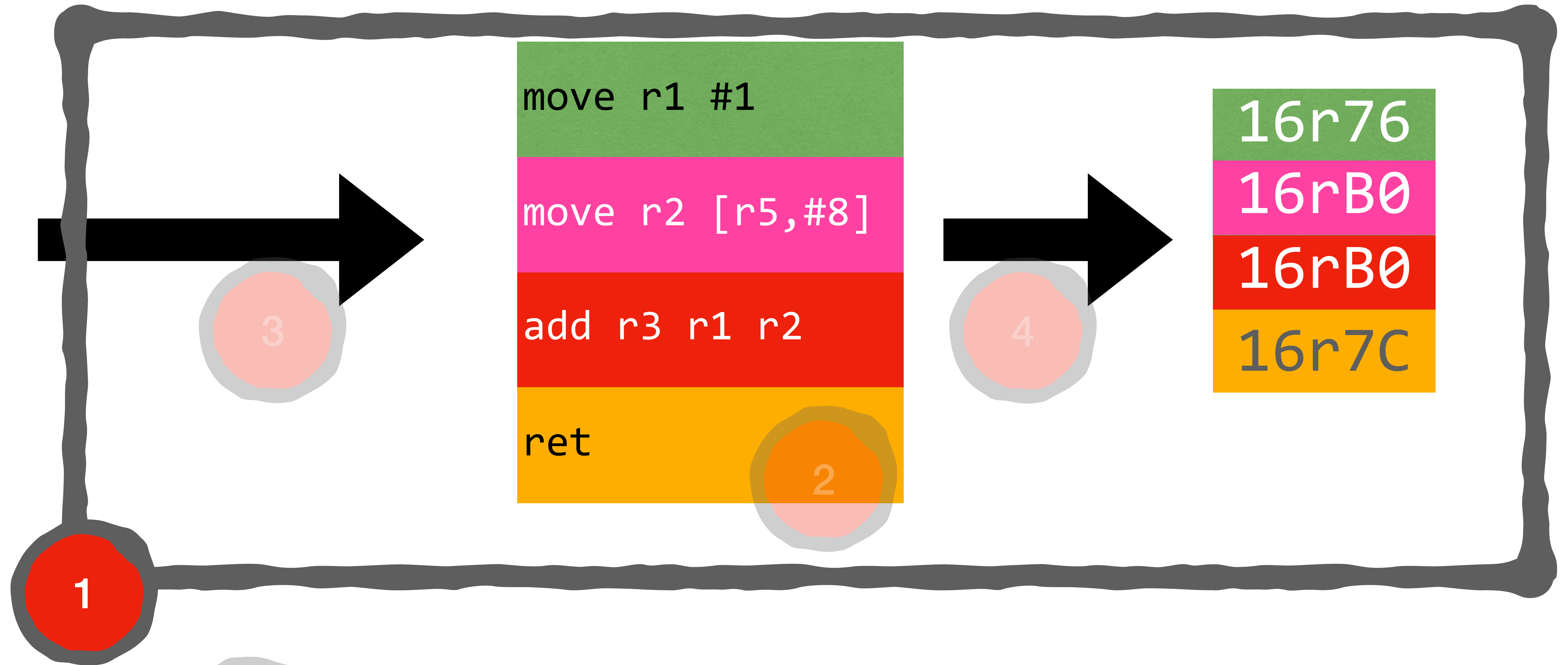
The Cogit Architecture

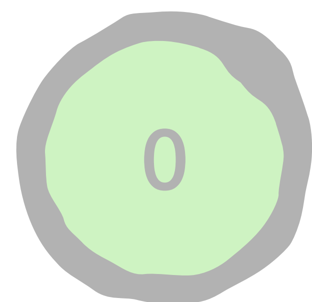




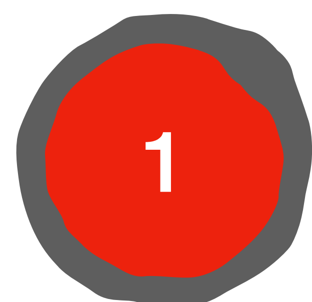
MyClass >> foo
^ 1 + 17

push 1
push 17
send +
returnTop

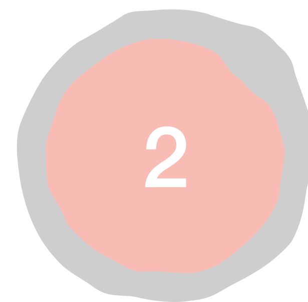




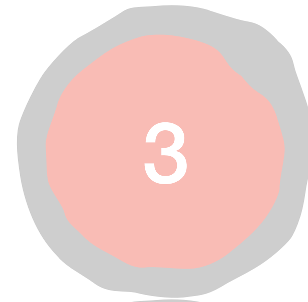
Bytecode Interpretation



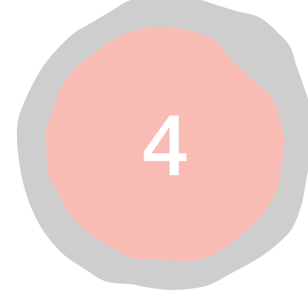
The Cogit Architecture



The CogRTL Intermediate Representation

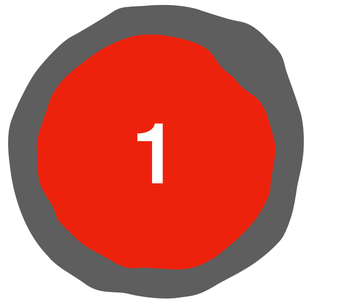


Bytecode-to-IR translation



IR-to-machine code translation

The Cogit Compiler



- Template based compiler: there is a template for each bytecode/primitive
- Each template generates a piece of intermediate representation (IR)
- Each IR instruction is translated to machine code

The Cogit Architecture

by example (illustrative, this is no real code below :))

```
<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

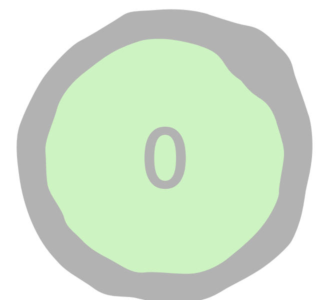
```
move X5 #1
move X3 #17
test X5 #0x1
je 0x40
test X3 #0x1
je 0x32
add X0 X3 X5
test X0 #0x1
je 0x24
move X5 X0
ret
```

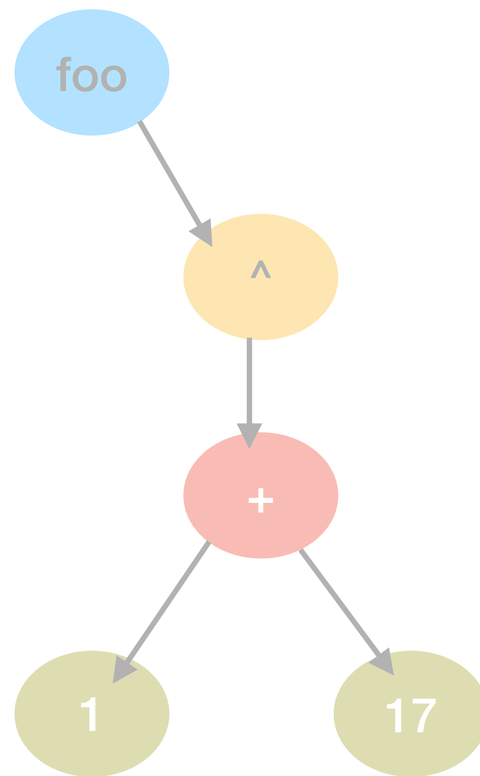
Machine Code

IR Generation
“gen*”

Machine Code Generation
“Concretize”

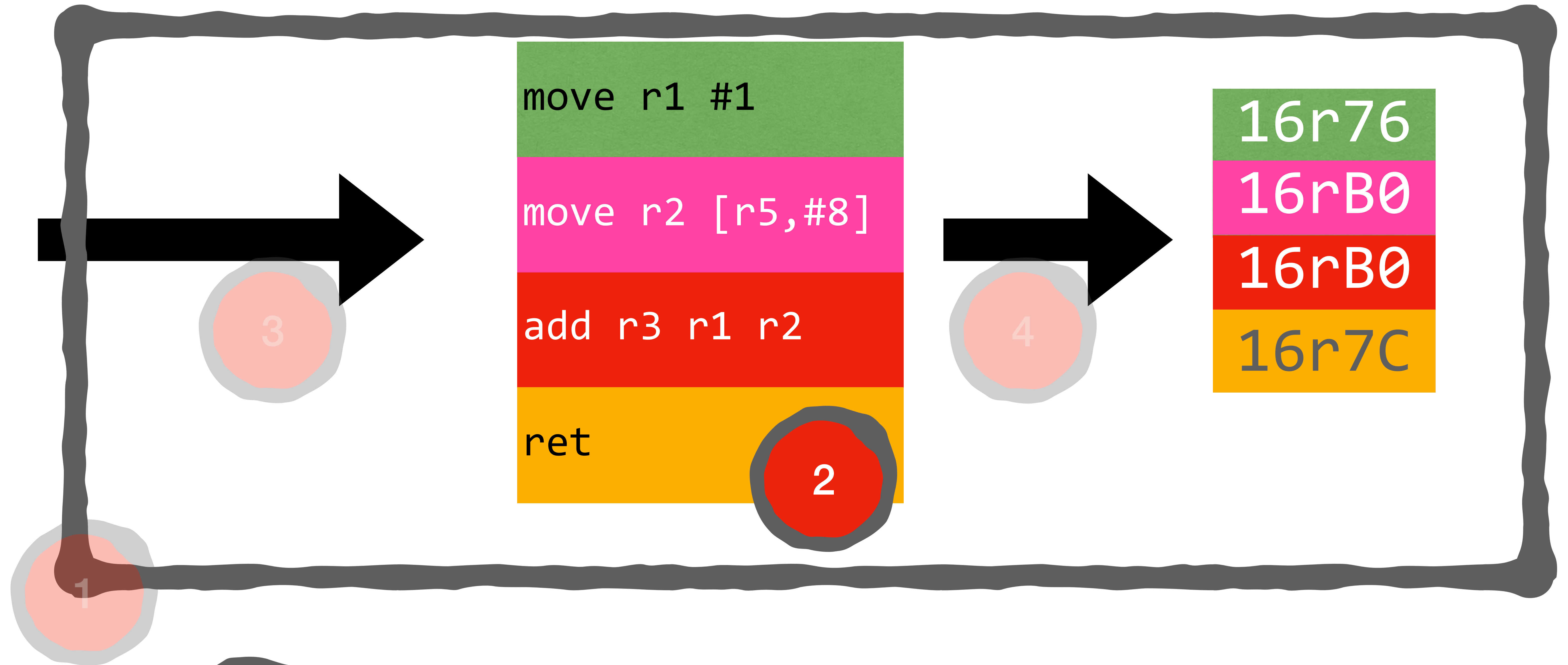
The CogRTL IR

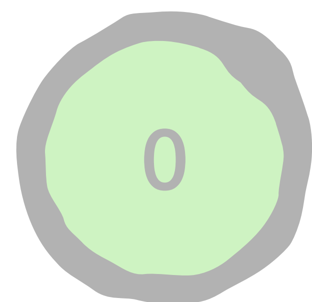




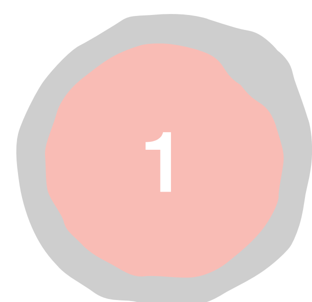
MyClass >> foo
^ 1 + 17

push 1
push 17
send +
returnTop

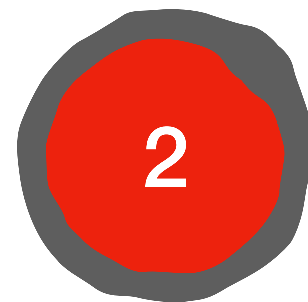




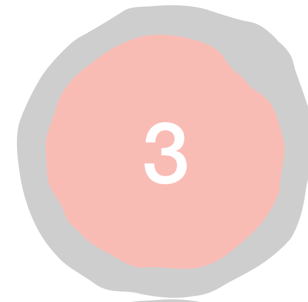
Bytecode Interpretation



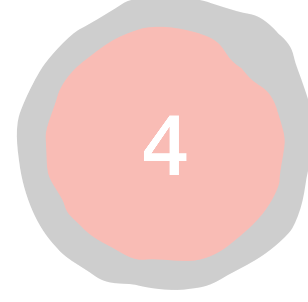
The Cogit Architecture



The CogRTL Intermediate Representation

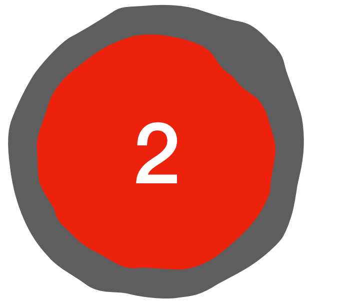


Bytecode-to-IR translation



IR-to-machine code translation

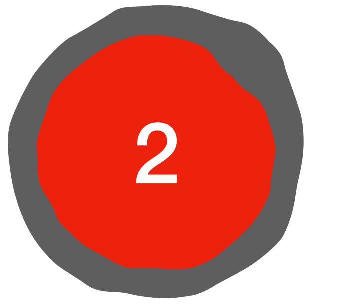
Cog RTL Intermediate Representation



- Linear Intermediate Representation
- Array of Tuples with operation code (opcode), args, and meta-data
- Created through factory methods
- Conventions: C means constant, R means register …

```
self PushR: R1.
self MoveCq: 17 R: R2.
self AddR: R1 R: R2.
```

Cog RTL is a 2-address code IR



- Operations have 2 operands
- Destination is implicitly the second operand

```
AddRR R1 R2   =>   R2 := R1 + R2
```

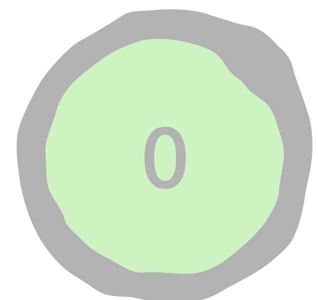
Cog RTL Registers

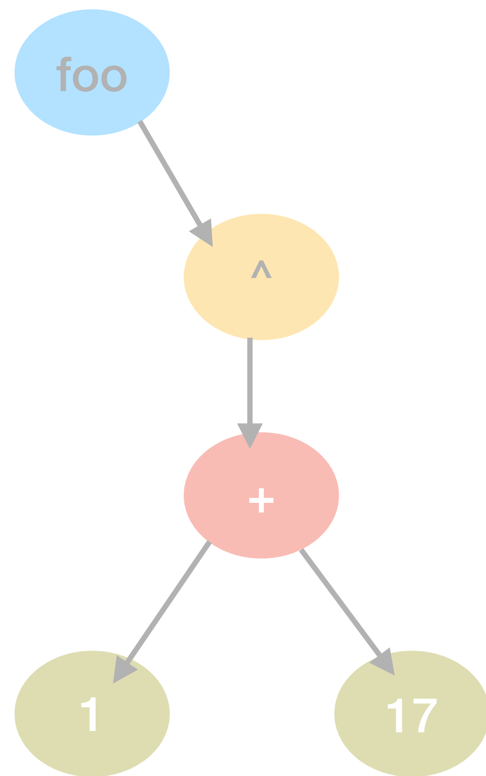
2

- CogRTL has fixed virtual registers
  - `ReceiverResultReg`
  - `Arg0Reg`

- When the VM is built, each virtual register is bound to a concrete register
  - e.g., `ReceiverResultReg = X5` in ArmV8

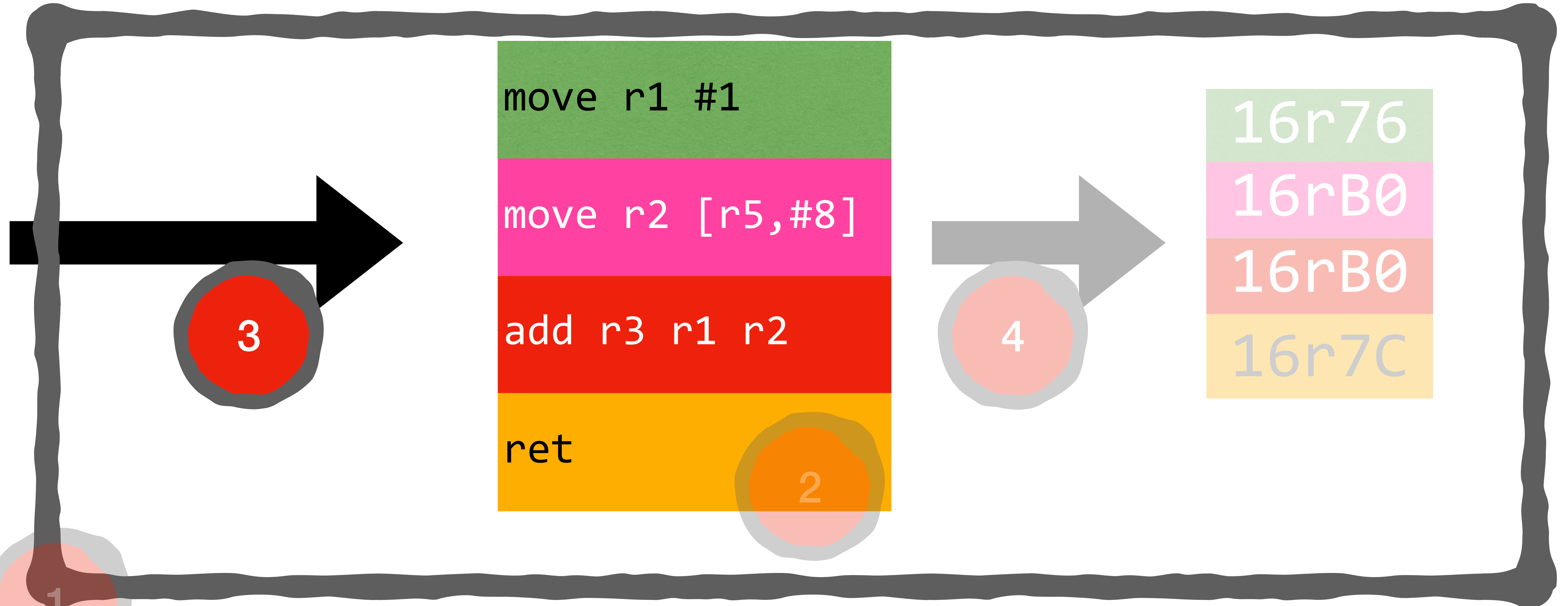
Bytecode-to-IR translation

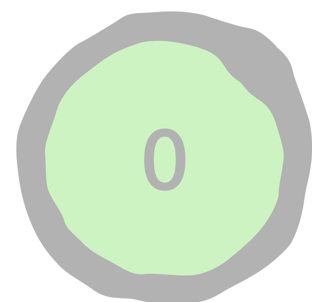




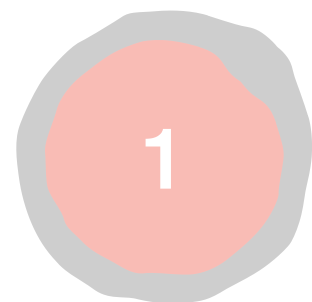

```
MyClass >> foo
    ^ 1 + 17
```

```
push 1
push 17
send +
returnTop
```

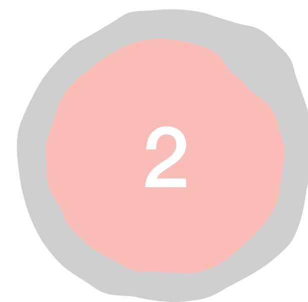




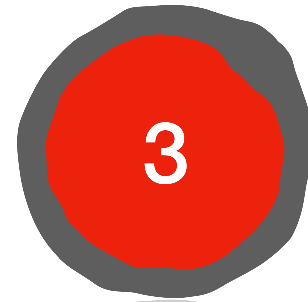
Bytecode Interpretation



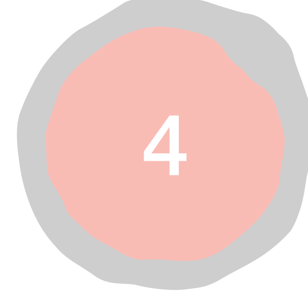
The Cogit Architecture



The CogRTL Intermediate Representation



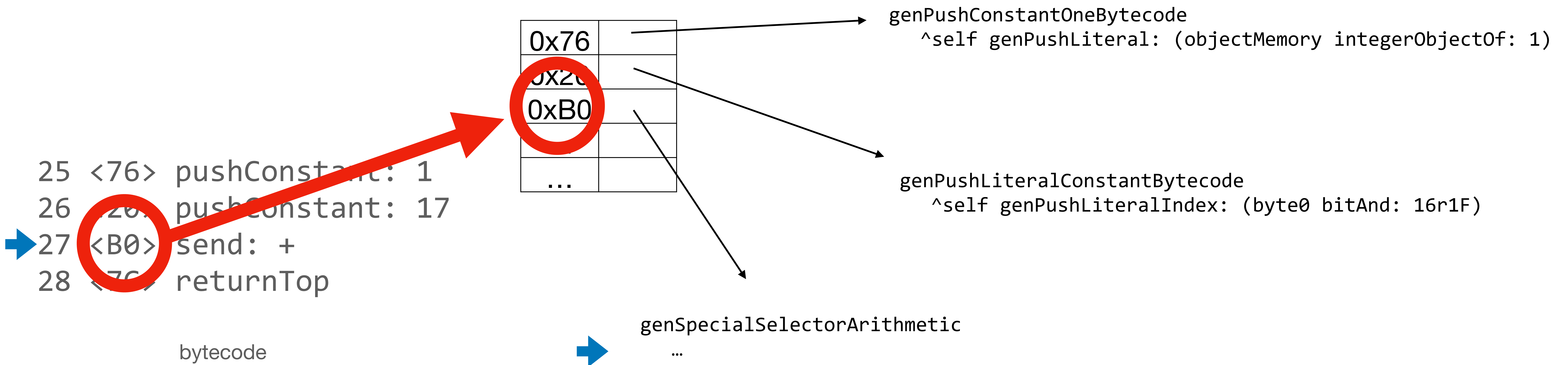
Bytecode-to-IR translation



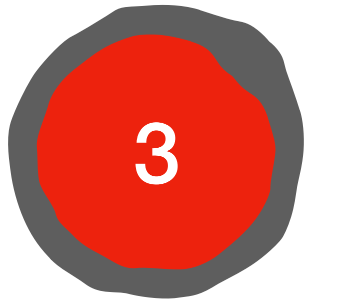
IR-to-machine code translation

Bytecode-to-IR translation

(the basic template schema)

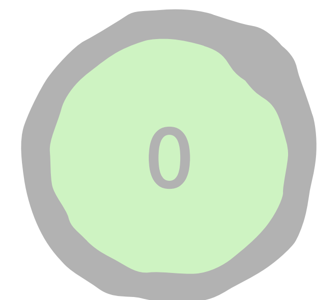


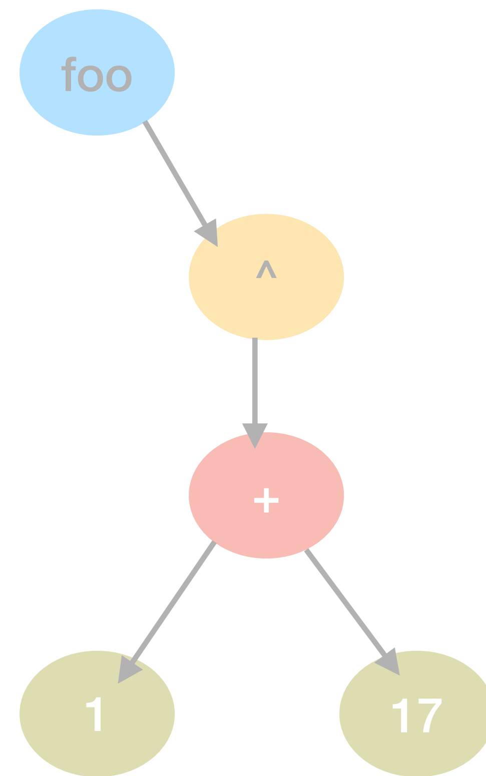
Example: push temporary template




```
genPushTemporaryVariable: index
  self
    MoveMw: (self frameOffsetOfTemporary: index)
    r: FPReg
    R: TempReg.
  self PushR: TempReg.
  ^0
```

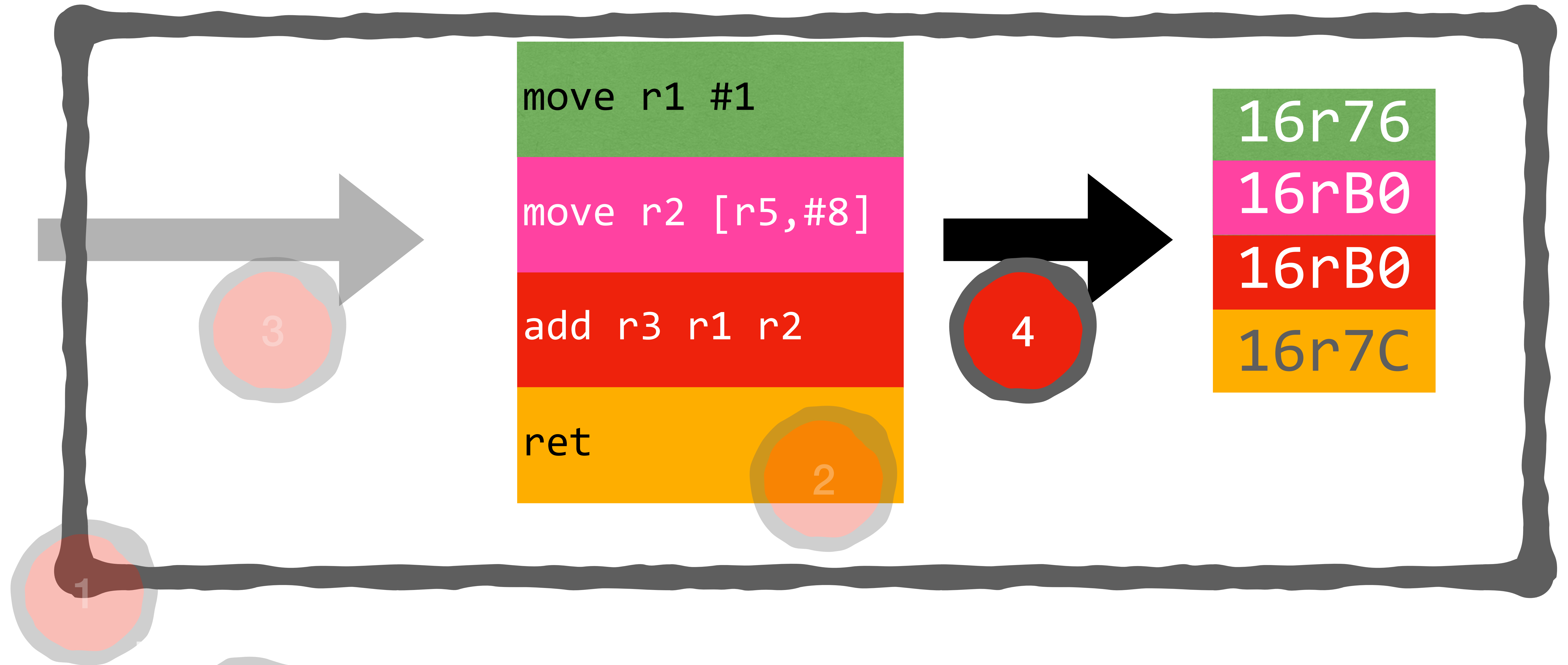
IR-to-machine code translation

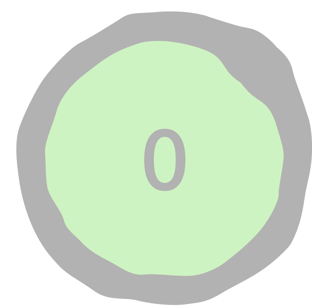




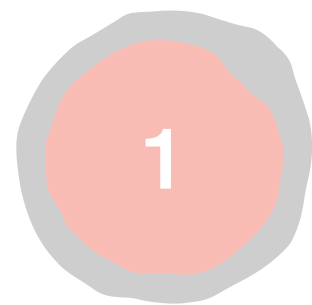
MyClass >> foo
^ 1 + 17

```
push 1
push 17
send +
returnTop
```

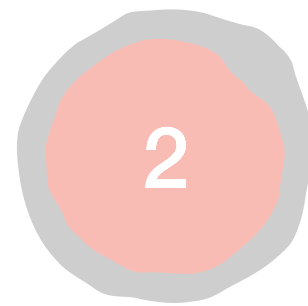




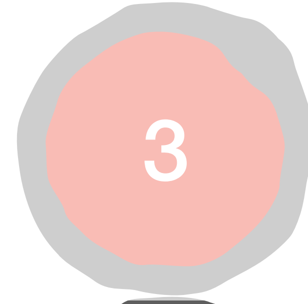
Bytecode Interpretation



The Cogit Architecture



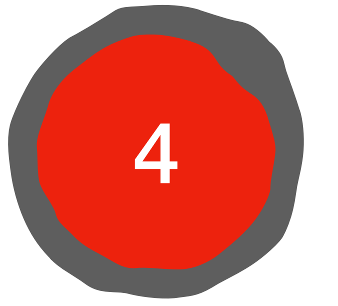
The CogRTL Intermediate Representation



Bytecode-to-IR translation

IR-to-machine code translation

IR-to-machine code translation



The concretize Step

- For each IR instruction generate the corresponding machine code
- Remember: we generate binary code, not assembly!

The Concretize Step

4

```
switch (opcode) { … case AddRR: self concretizeAddRR  }
```

```
concretizeAddRR
  | leftRegister rightRegister destinationRegister |
  leftRegister := operands at: 0.
  destinationRegister := rightRegister := operands at: 1.

  bytes :=  self
    addsSize: 1
    leftRegister: leftRegister
    shiftedRightRegister: rightRegister
    shiftType: 0
    shiftOffset: 0
    destinationRegister: destinationRegister.

  self machineCodeAt: 0 put: bytes.
  ^ machineCodeSize := 4
```

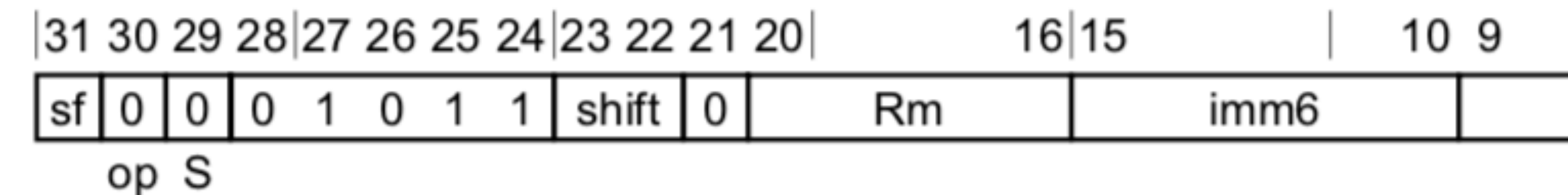
Writing Machine Code

Just some bit manipulations, following the manual

```
(is64Bits bitAnd: 1) << 31
	bitOr: ((substractionFlag 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
destination register.



32-bit variant

Applies when `sf == 0`.

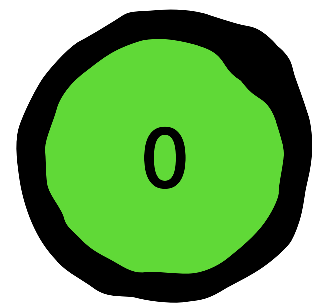
```
ADD <Wd>, <Wn>, <Wm>{, <shift> #<amount>}
```

64-bit variant

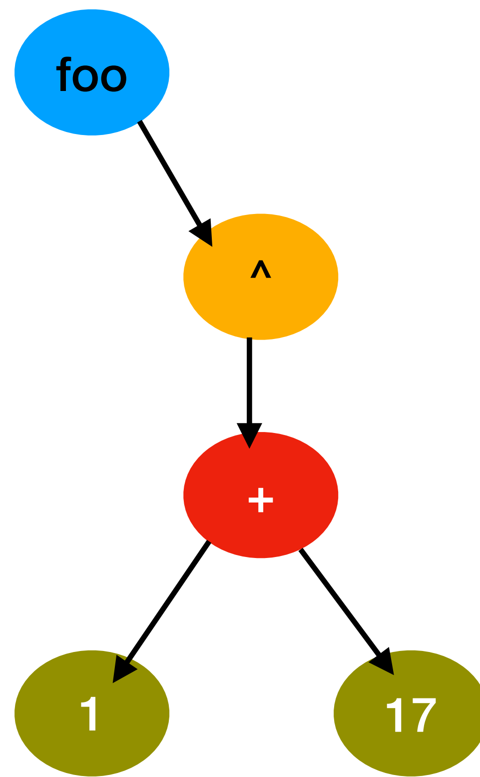
Applies when `sf == 1`.

```
ADD <Xd>, <Xn>, <Xm>{, <shift> #<amount>}
```

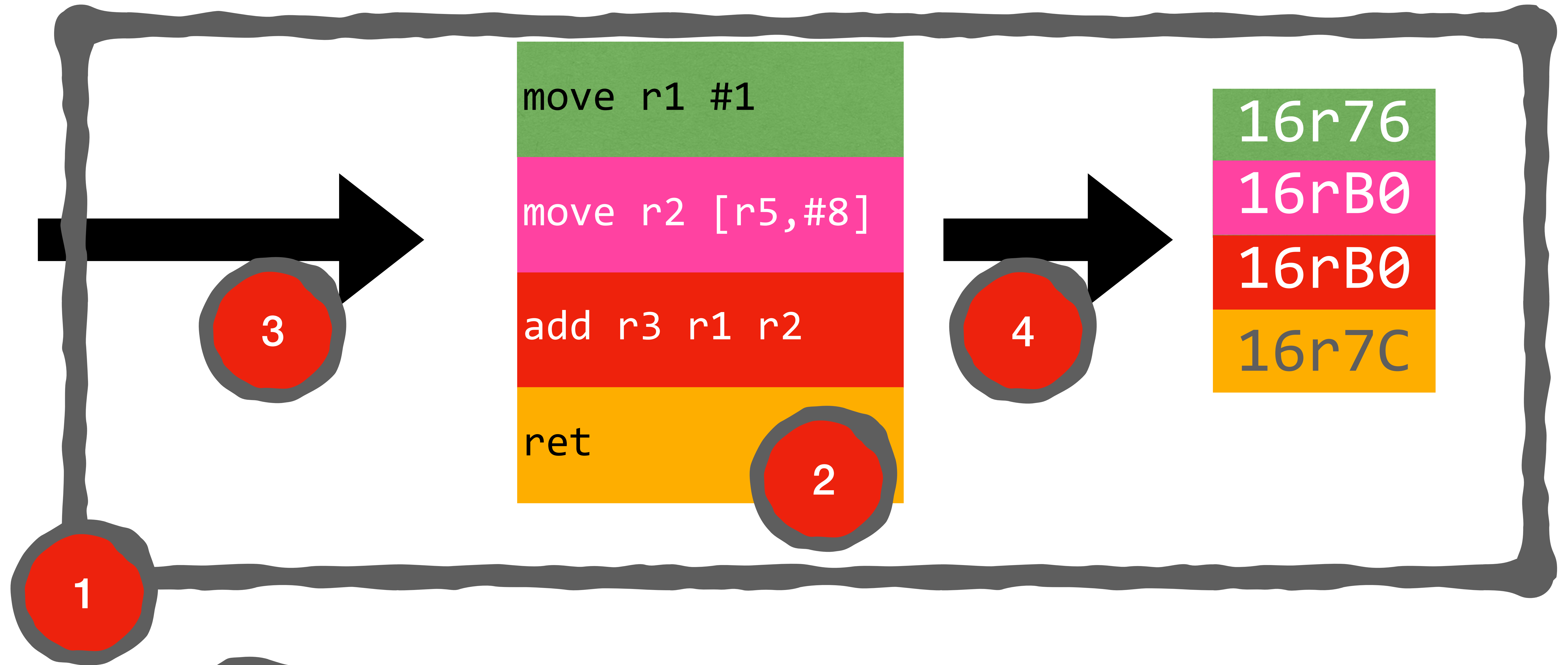
The Cogit Architecture

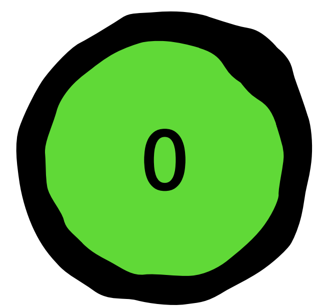



```
MyClass >> foo
   ^ 1 + 17
```

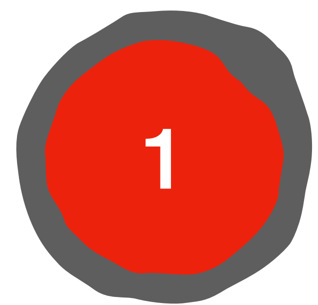



```
push 1
push 17
send +
returnTop
```

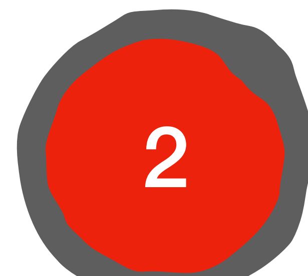




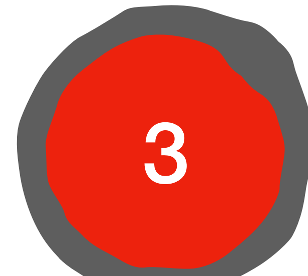
Bytecode Interpretation



The Cogit Architecture



The CogRTL Intermediate Representation



Bytecode-to-IR translation

IR-to-machine code translation