Compiling Bytecode

What is a compiler

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


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

A program that translates a program in a *source* language to a *target* language

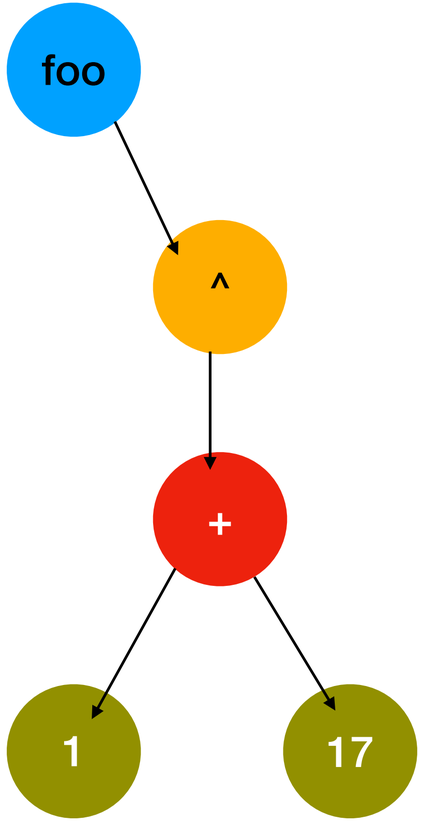
Overview of a compiler internals

An example with bytecode

Source code

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

Parse



Intermediate Representation

generate

Target Code

```
16r76
16r20
16rB0
16r7C
```

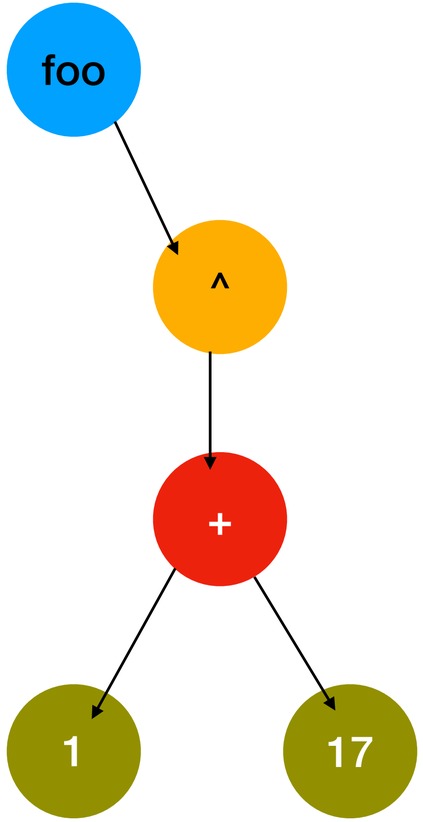
Example 1: The old Pharo Compiler

AST as intermediate representation

Source code

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

Parse



Intermediate Representation

generate

Target Code

```
16r76
16r20
16rB0
16r7C
```

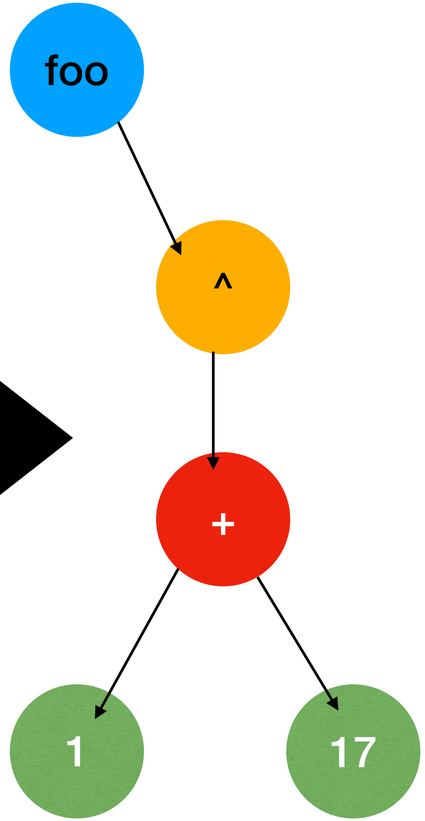
Example 2: The Opal Compiler

Introducing linear representations

Source code

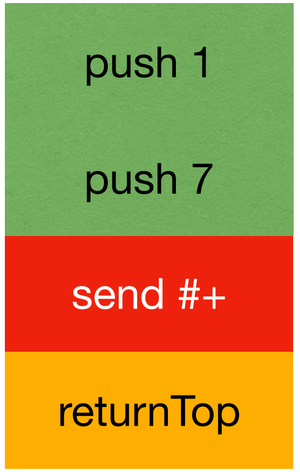
```
MyClass >> foo
    ^ 1 + 17
```

Parse



Intermediate Representation 1: ASTs

generate



Intermediate Representation 2
Linear Bytecode-like IR

generate

Target Code

What is the target code?

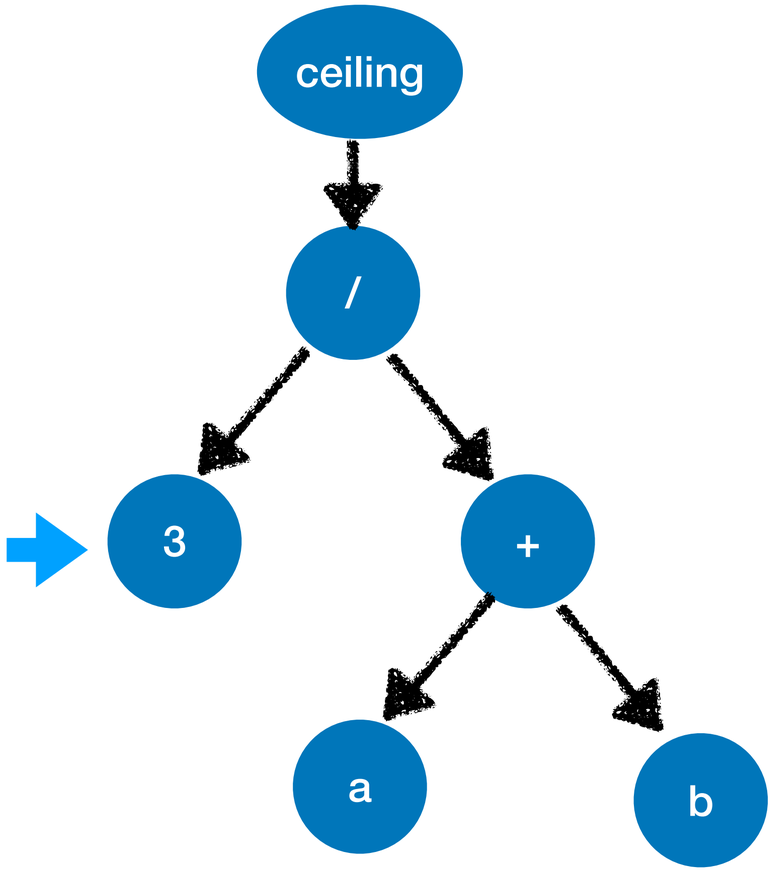
- Another programming language => we talk about transpilation
  e.g., Pharo to C translation

- Some binary code for a virtual machine
  e.g., the Pharo bytecode

- Some binary code for a real machine
  e.g., machine code for x86, or ARMv8

Generating stack based code

Same traversal order as the interpreter



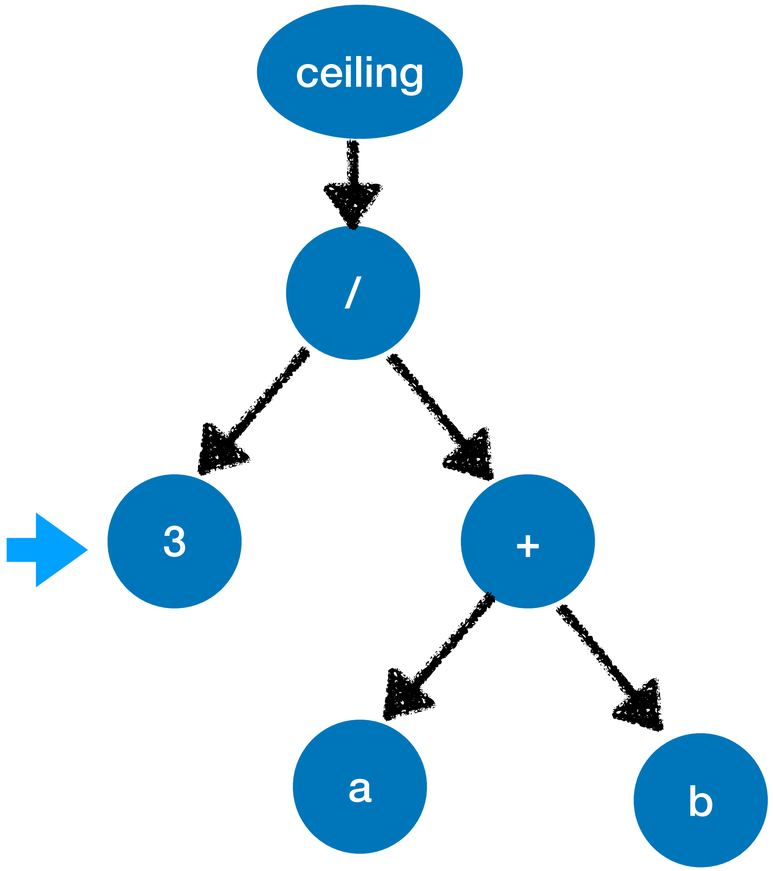

```
(3 / (a + b)) . ceiling()
```

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

the stack

Generating stack based code

Same traversal order as the interpreter



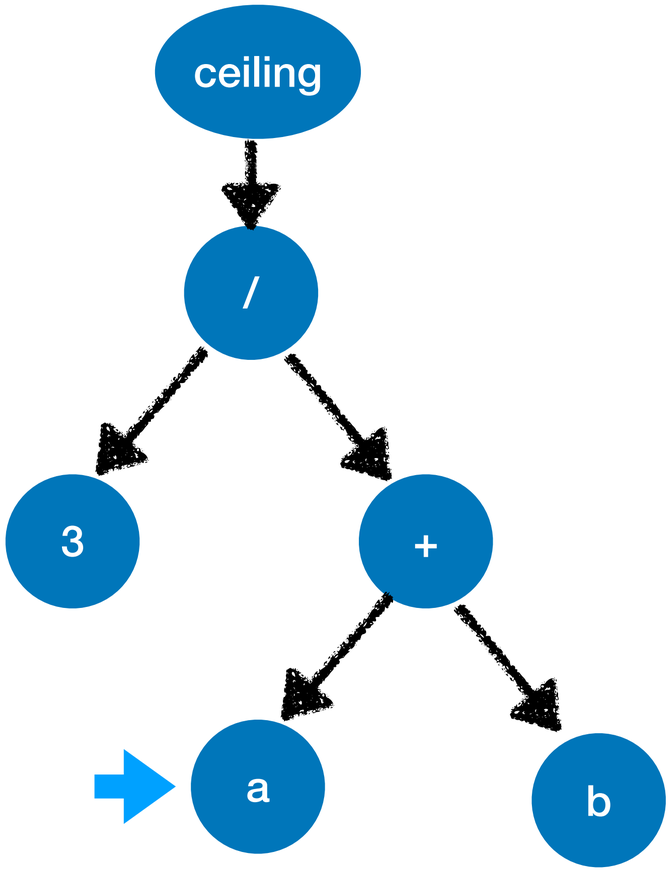

```
(3 / (a + b)) . ceiling()
```

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

the stack

Generating stack based code

Same traversal order as the interpreter



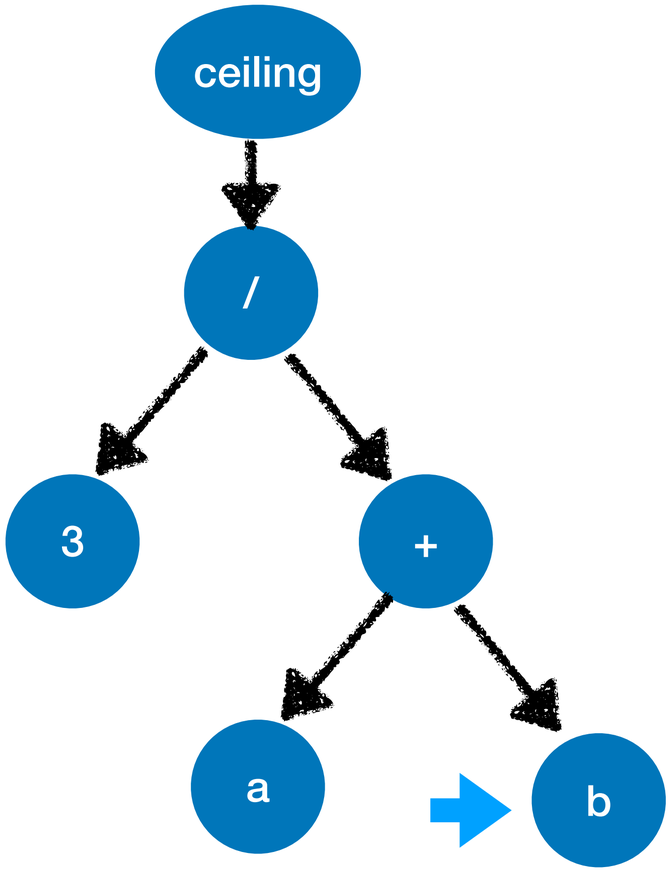

```
(3 / (a + b)) . ceiling()
```

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

the stack

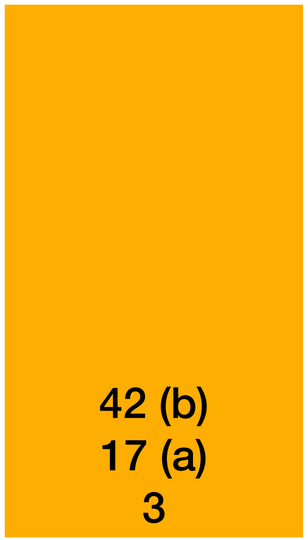
Generating stack based code

Same traversal order as the interpreter




```
(3 / (a + b)) . ceiling()
```

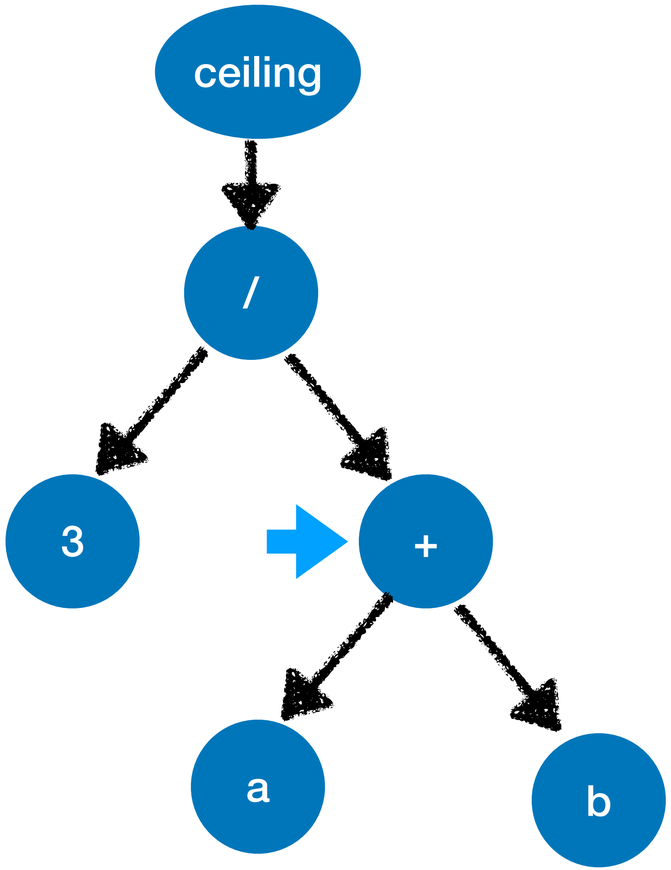
```
push 3
push a
push b
send +
send /
send ceiling
```



the stack

Generating stack based code

Same traversal order as the interpreter



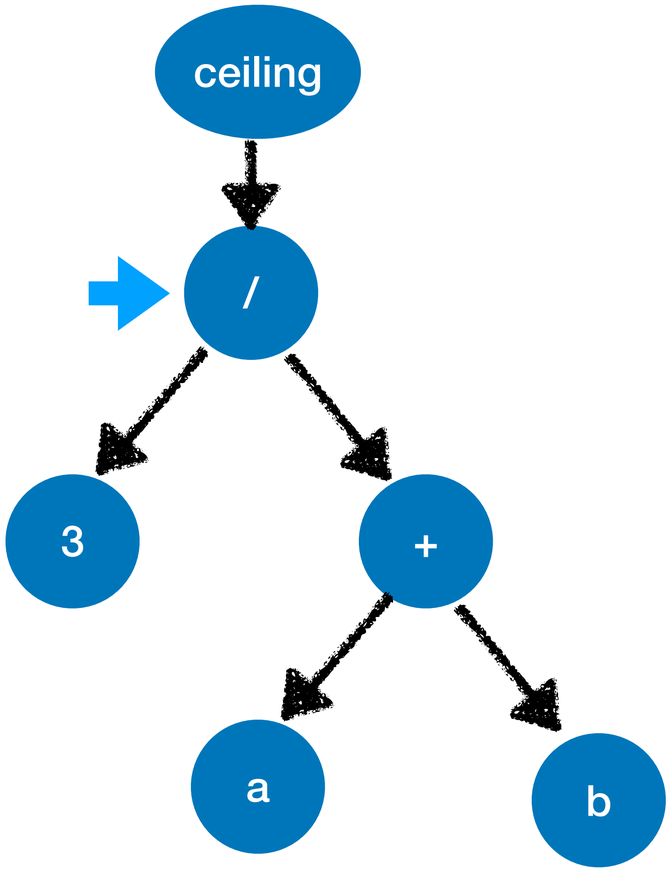

```
(3 / (a + b)) . ceiling()
```

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

the stack

Generating stack based code

Same traversal order as the interpreter



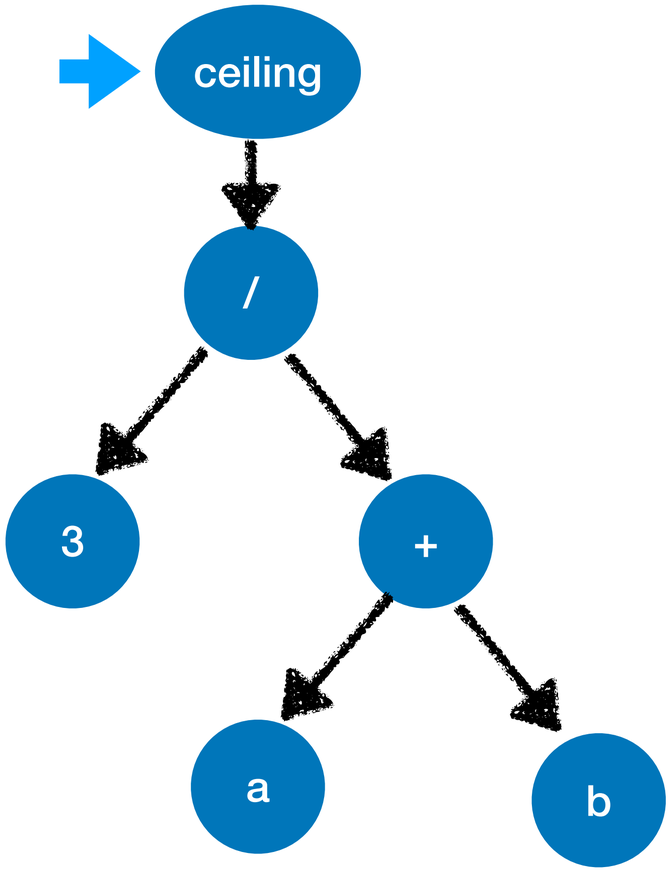

```
(3 / (a + b)) . ceiling()
```

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

the stack

Generating stack based code

Same traversal order as the interpreter




```
(3 / (a + b)) . ceiling()
```

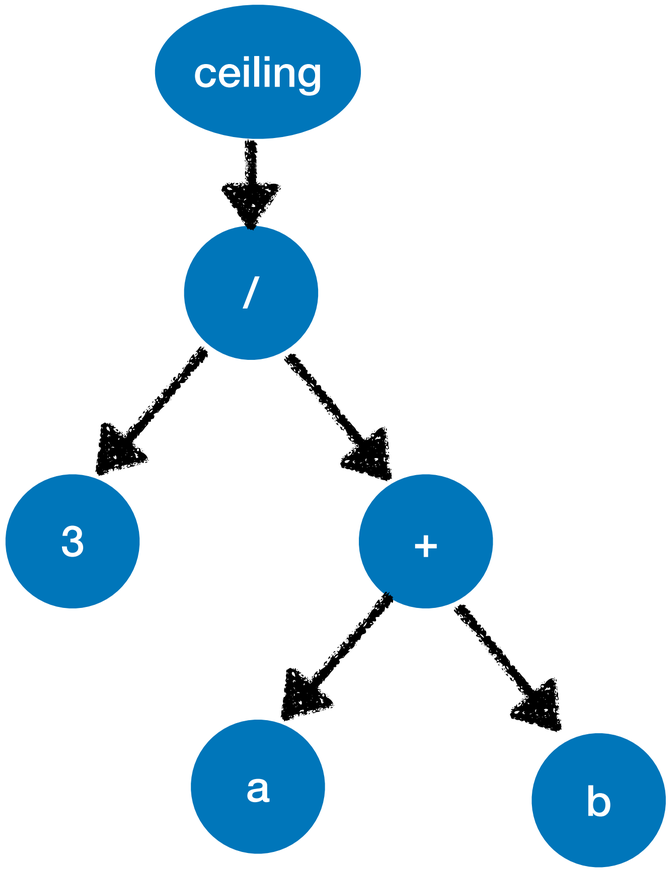
```
push 3
push a
push b
send +
send /
send ceiling
```

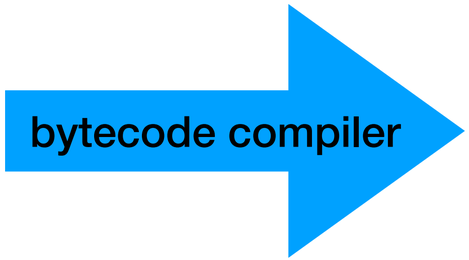
the stack

Bytecode

Stack-based linear code

```
(3 / (a + b)) . ceiling()
```






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

A simple bytecode compiler

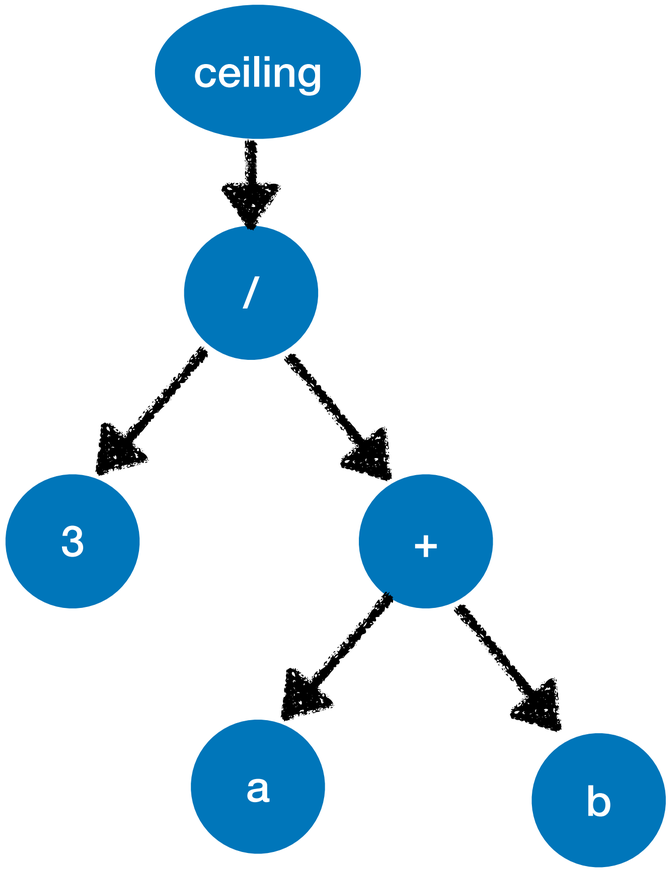
Generating code directly

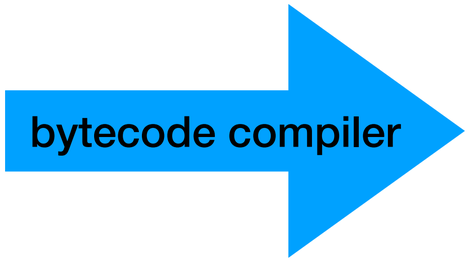
- Case analysis per type of node
- Each node encodes one or more instructions objects
- E.g.,

```
Compiler >> visitLiteralNode: aLiteralNode
        self generatePushLiteral: aLiteralNodeValue
```

A simple bytecode compiler

Generating code directly





17

32

33

55

56

48

opcodes

A simple bytecode compiler

Generating code directly

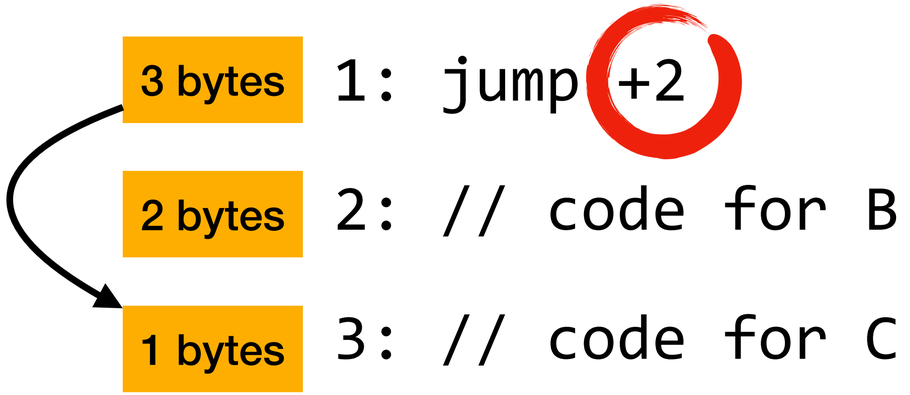
- Needs auxiliary data structures to resolve pc-relative instructions (e.g., jumps)

```
1: jumpTo 3

2: // code for B

3: // code for C
```

offset needs to know the sizes of all intermediate instructions

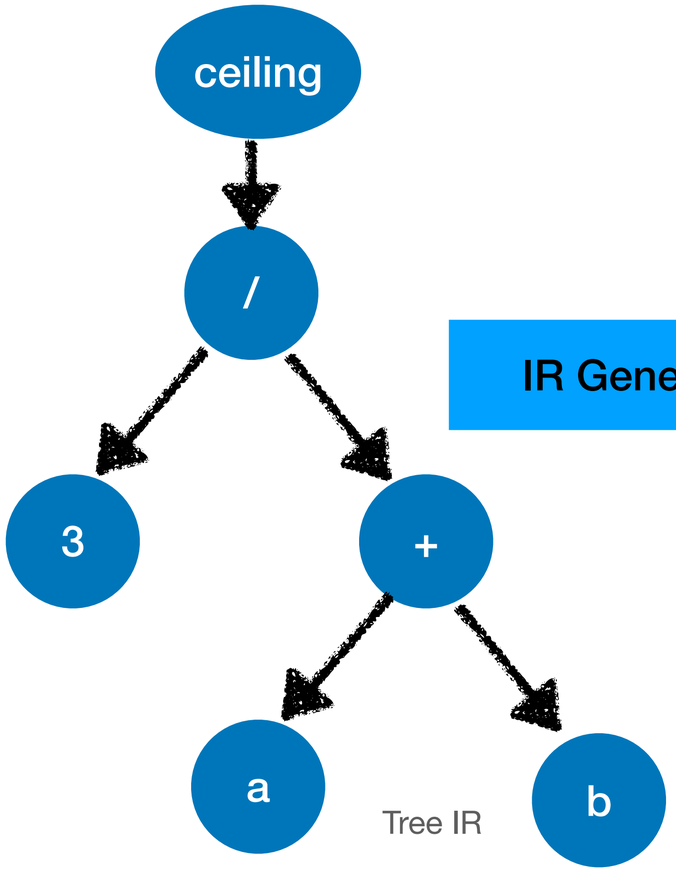


Using an Intermediate Representation (IR)

- Case analysis per type of node
- Each node generates one or more instruction ***objects***
- Extra passes on the IR do code transformations and code generation
- E.g.,

```
Compiler >> visitLiteralNode: aLiteralNode
    instructions add: (IRPushLiteral value: aLiteralNode value)
```

Using an Intermediate Representation (IR)



Tree IR

IR Generation

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

Linear IR

Code Generation

```
17
32
33
55
56
48
```

opcodes

Using an Intermediate Representation (IR)

Generating code directly

- IR objects can have references between them to represent dependencies
- They can also store extra properties (size, usages, parameters…)

- A Builder is a good abstraction to help in creating it!

Compiling Control Flow

Example if

```
if (cond) {
  // True case
} else {
  // False case
}
```

```
// Generate condition
ifFalse := builder jumpIfFalse.
// Generate True case
jumpToEnd := builder jump.
ifFalse target: builder label.
// Generate False case
jumpToEnd target: builder label.
```

Compiling Control Flow

Example loop

```
while (cond) {
  // Loop Body
}
```

```
loopHead := builder label.

// Generate condition

exitJump := builder jumpIfFalse.

// Generate Loop Body

builder jump: loopHead.

exitJump target: builder label.
```

Conclusion

- Generating Stack-Based bytecode requires same traversal as interpreting it

- Compilers can use one or many Intermediate Representations (IRs)
- IRs help in doing manipulations to the code before code generation
- Compiling control flow structures is eased with builder and special instructions such as labels