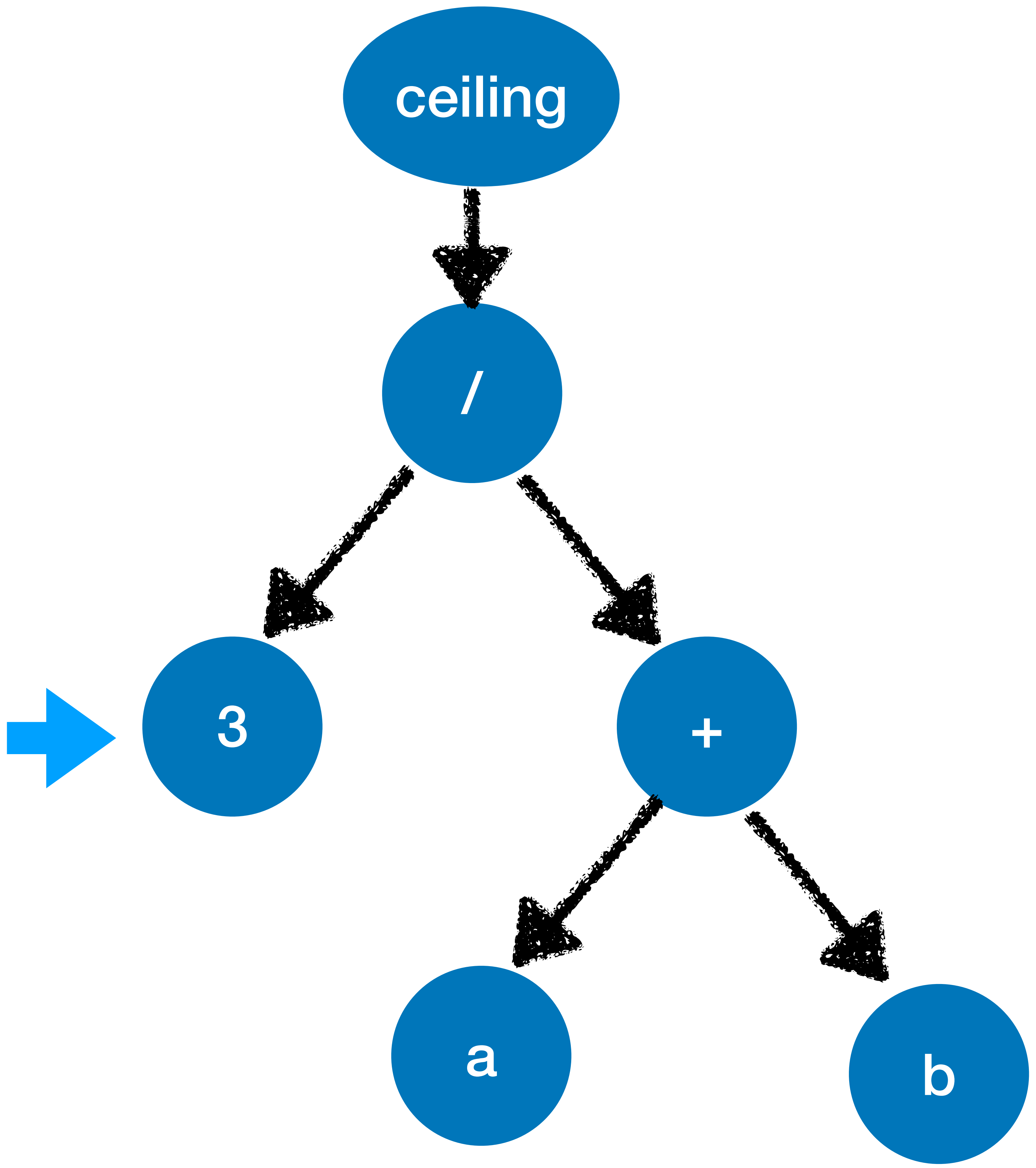
Stack-based Bytecode Design

Stack-Based AST interpreters

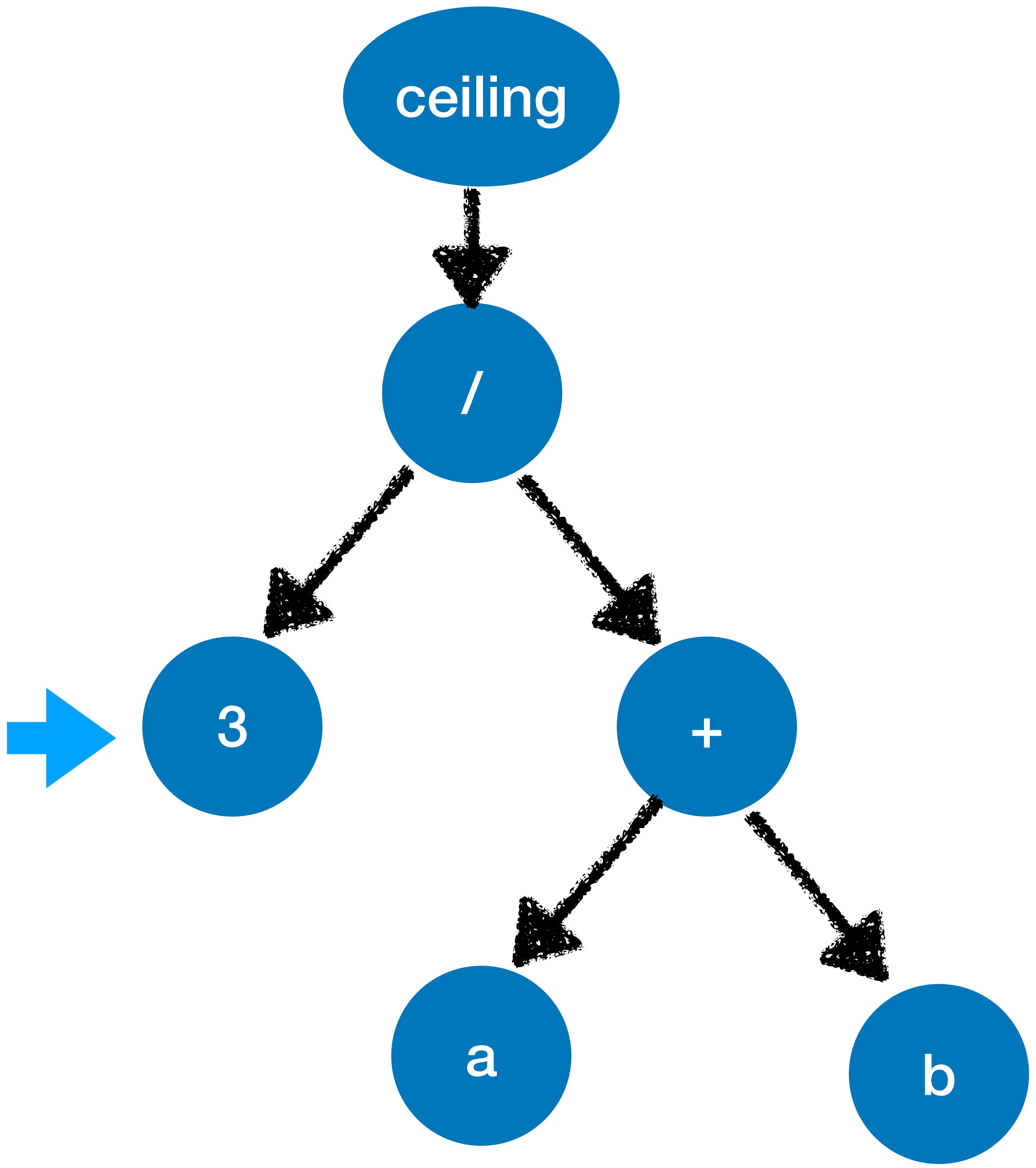
Sharing state through an implicit stack - example




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

Stack-Based AST interpreters

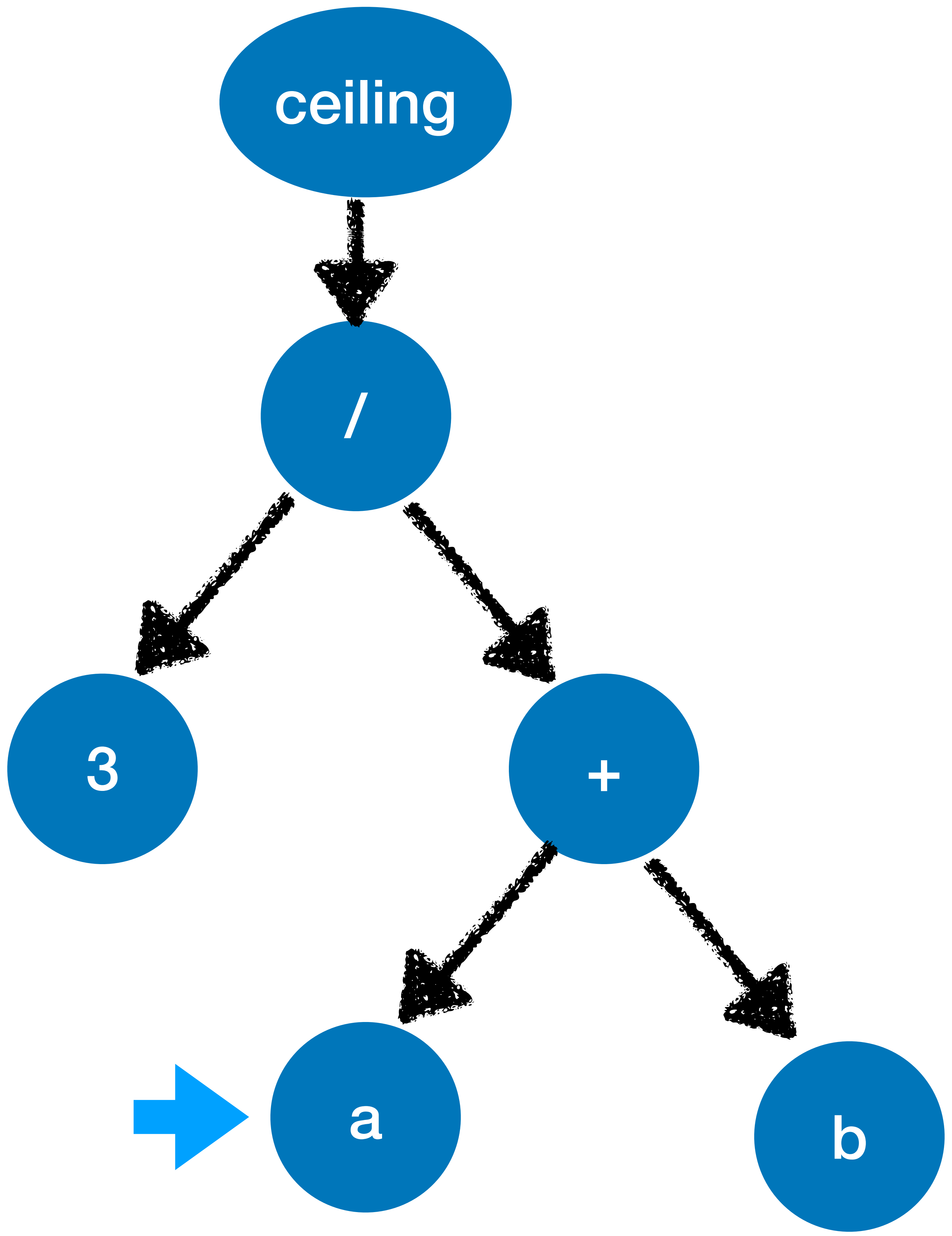
Sharing state through an implicit stack - example



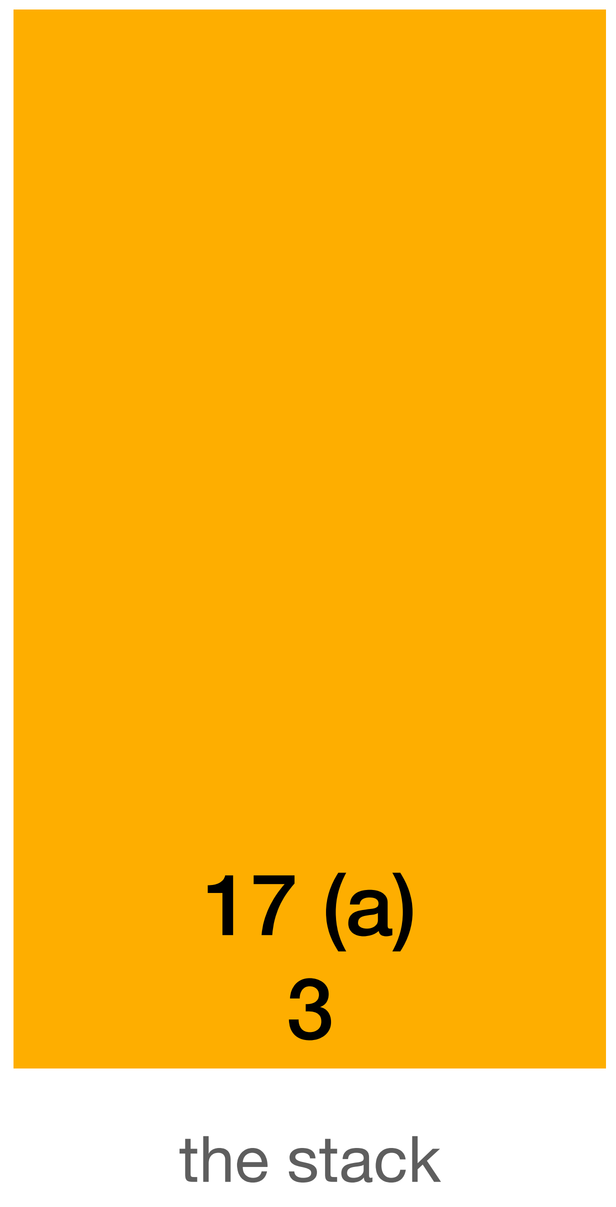

```
(3 / (a + b)) . ceiling()
```

Stack-Based AST interpreters

Sharing state through an implicit stack - example

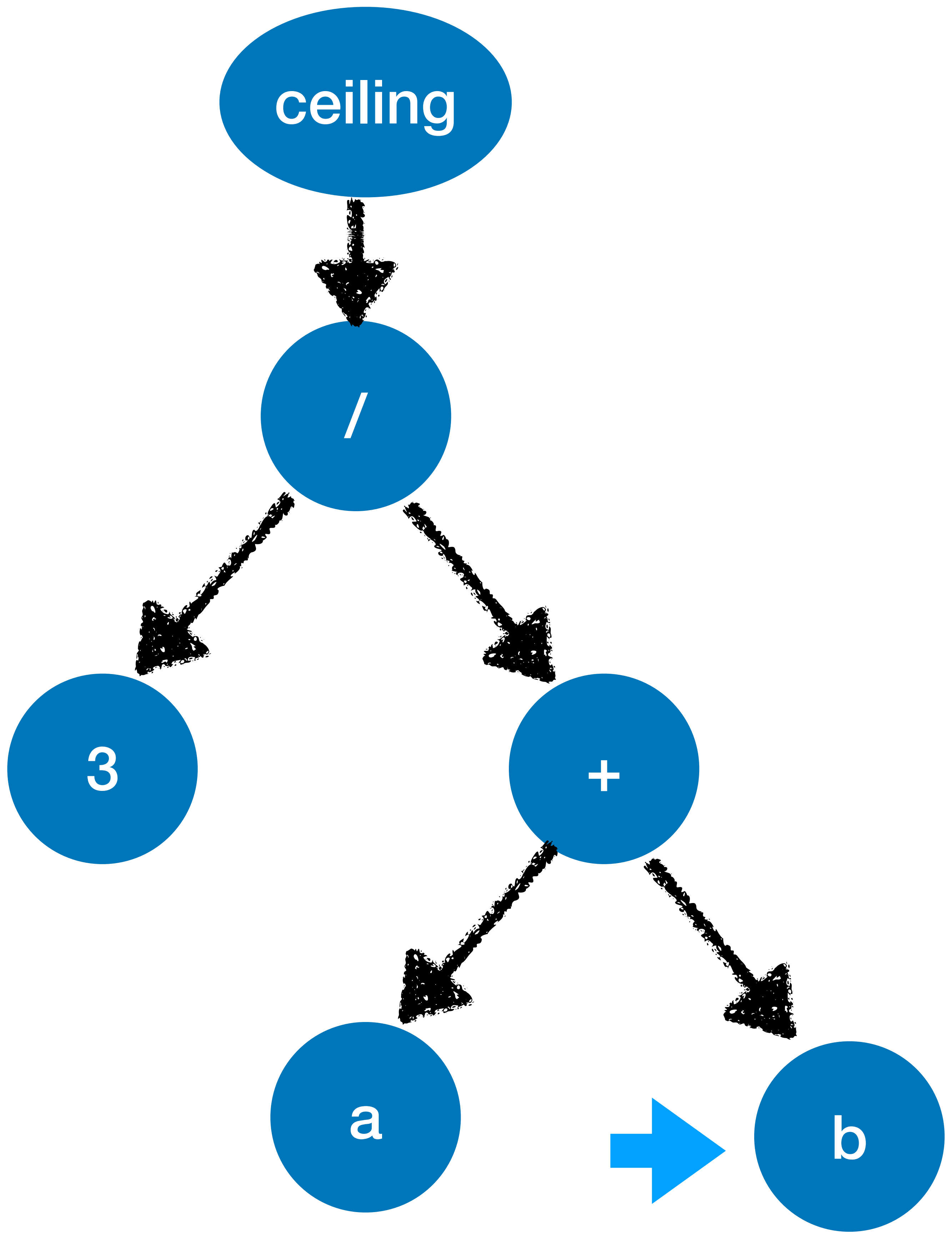



```
(3 / (a + b)) . ceiling()
```

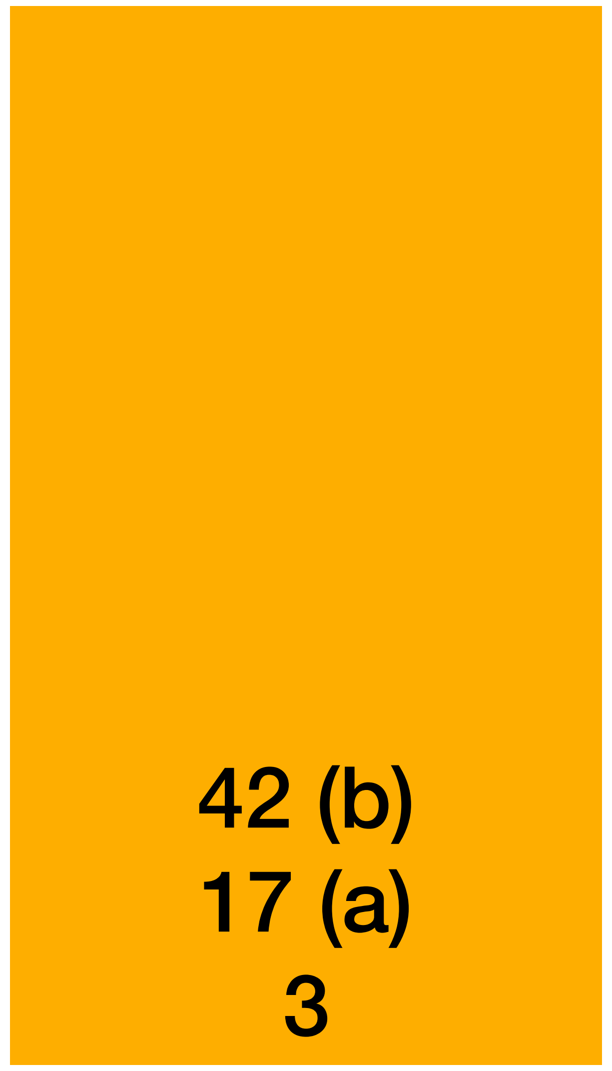


Stack-Based AST interpreters

Sharing state through an implicit stack - example



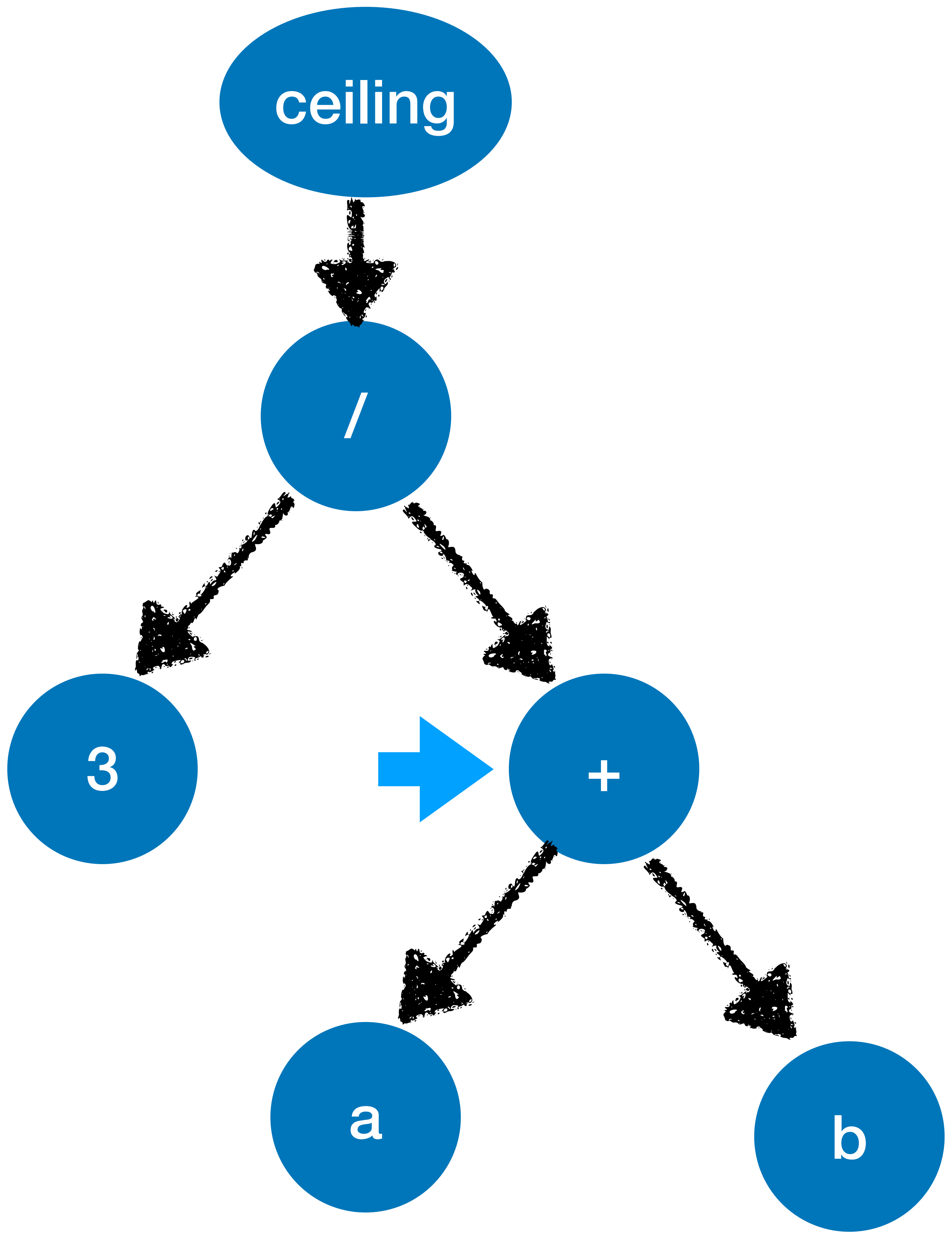

```
(3 / (a + b)) . ceiling()
```



the stack

Stack-Based AST interpreters

Sharing state through an implicit stack - example

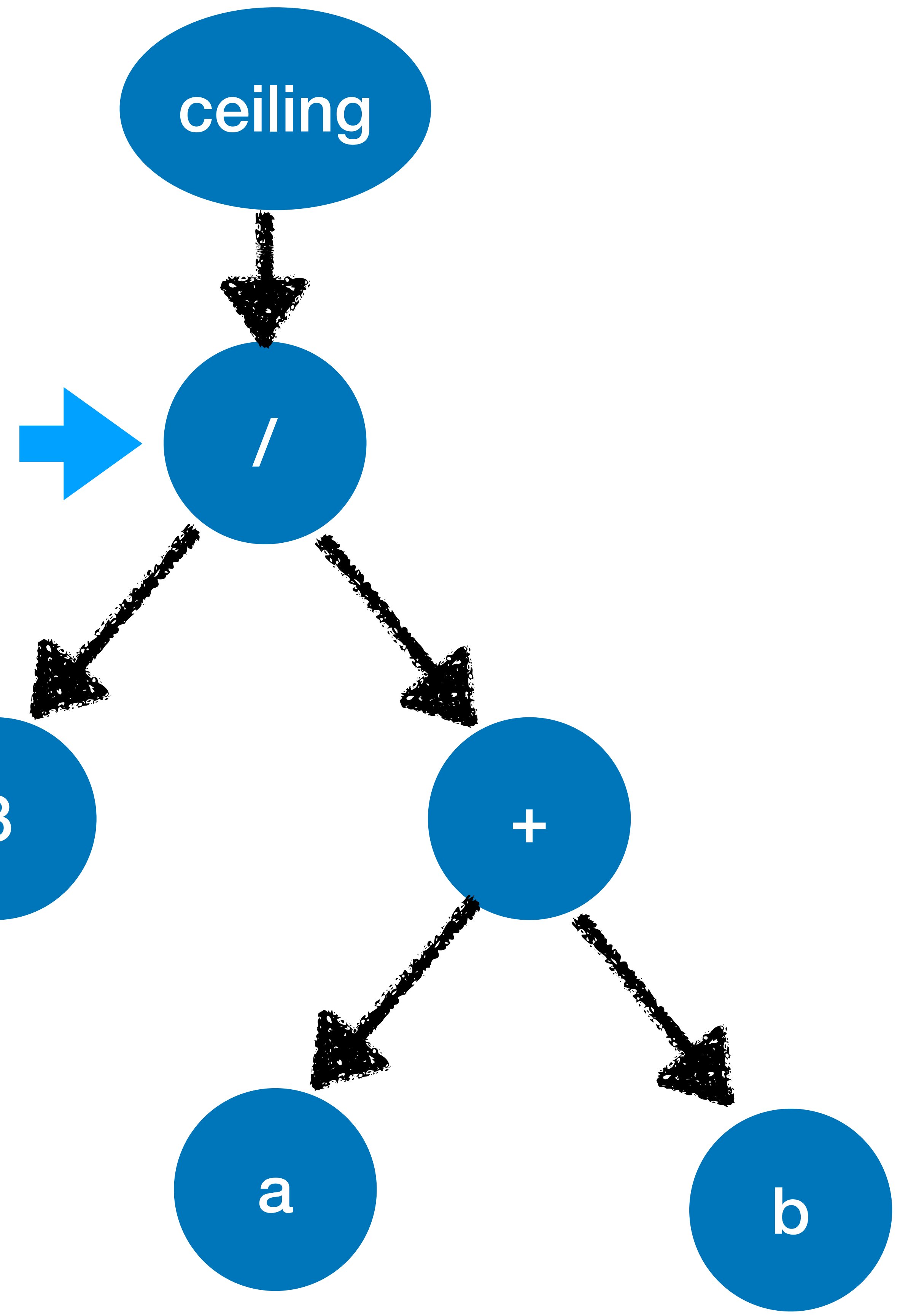



```
(3 / (a + b)) . ceiling()
```

the stack

Stack-Based AST interpreters

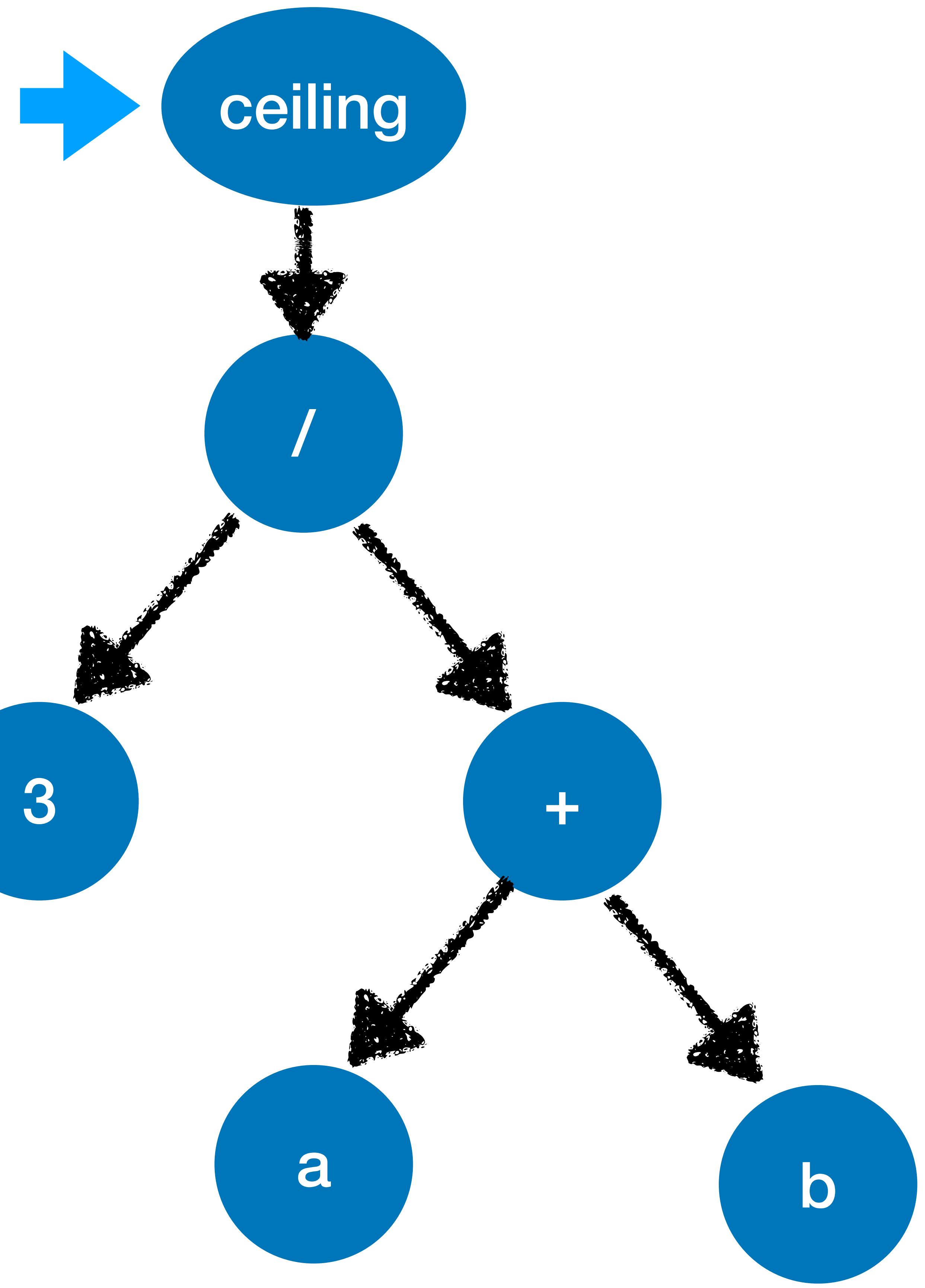
Sharing state through an implicit stack - example



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

Stack-Based AST interpreters

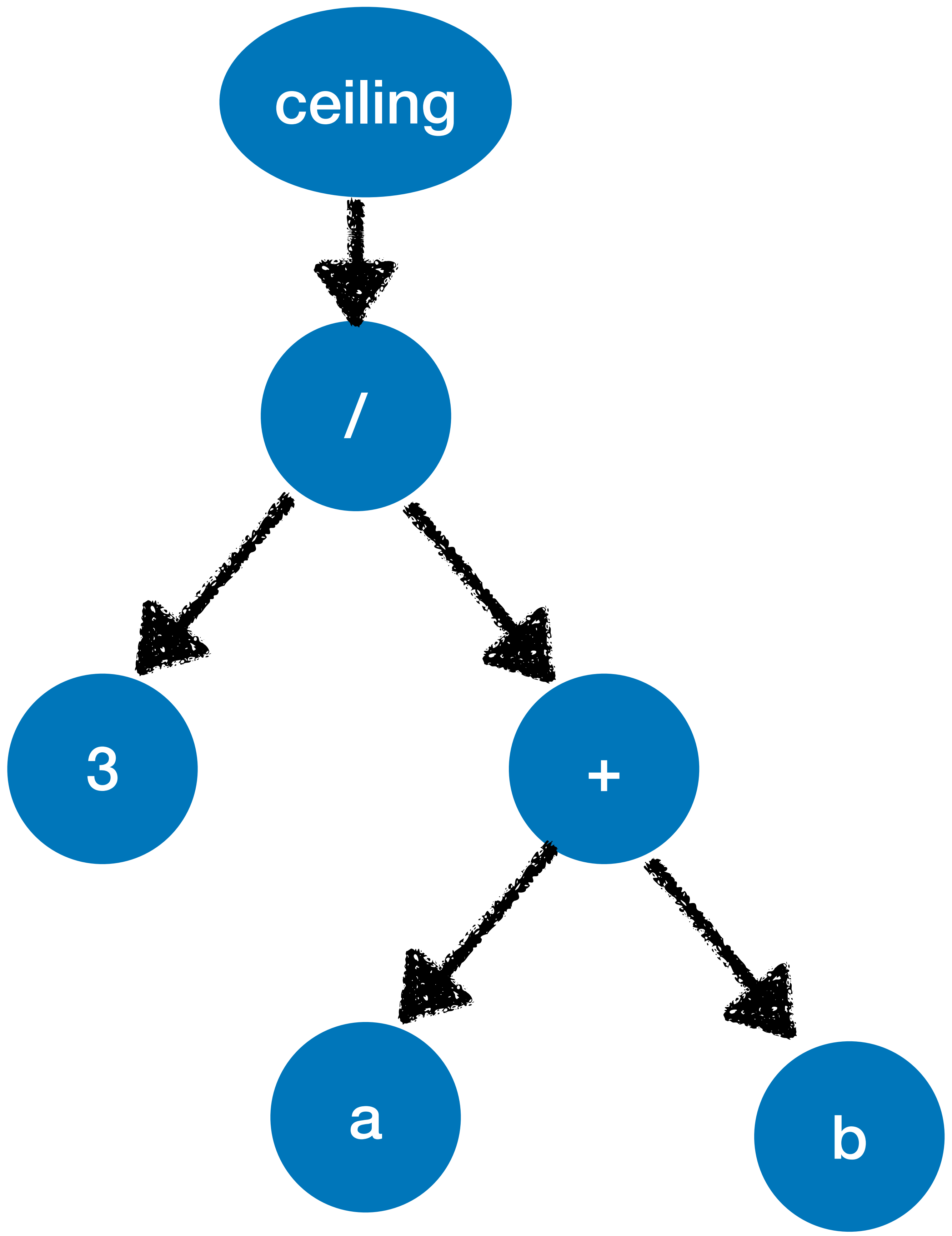
Sharing state through an implicit stack - example




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

Stack-Based AST interpreters

Sharing state through an implicit stack - example

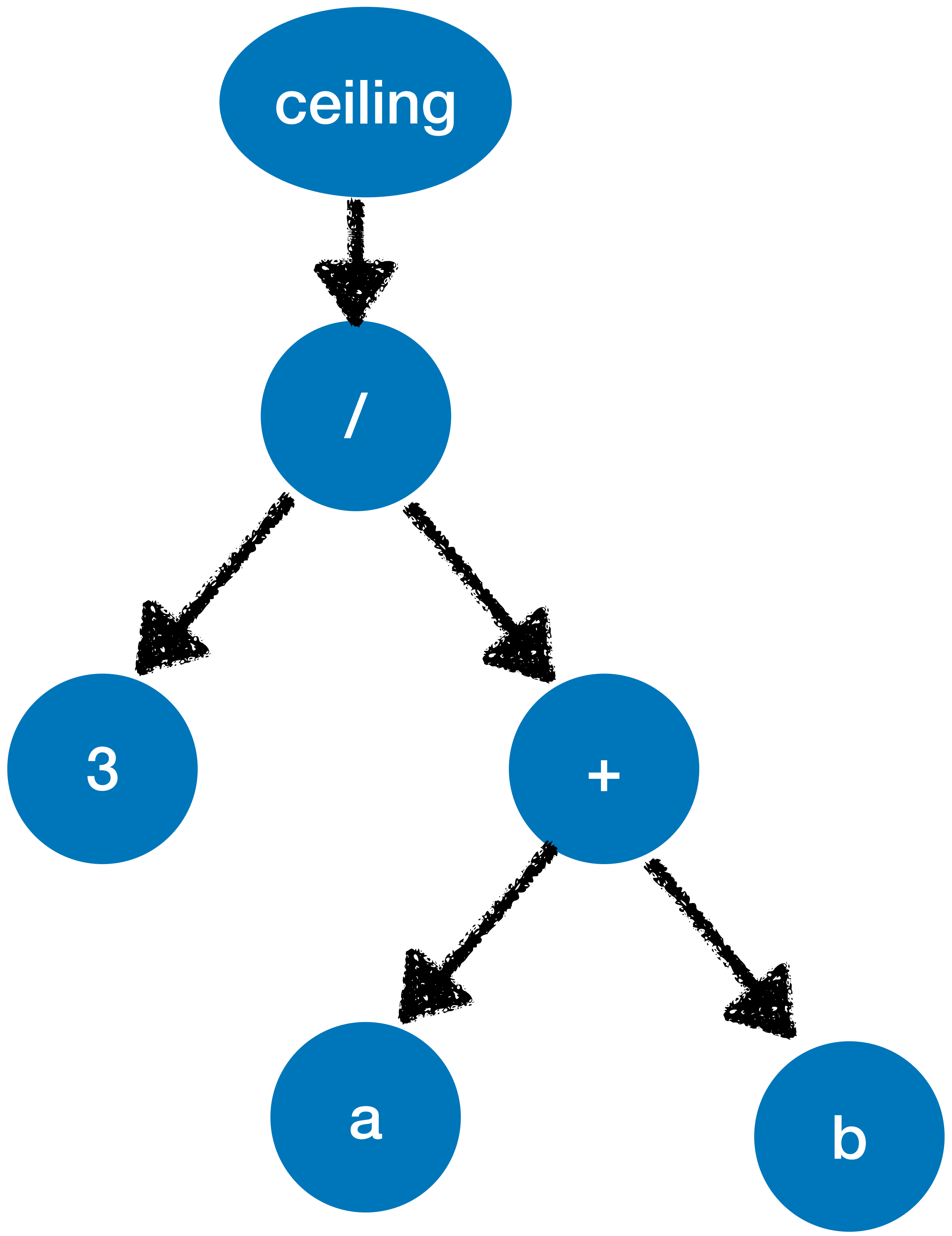



```
(3 / (a + b)) . ceiling()
```

- Post-order depth-first traversal of the code
- Share state through implicit stack
- Each operation pops arguments and pushes their result

Stack-Based AST interpreters

Some disadvantages



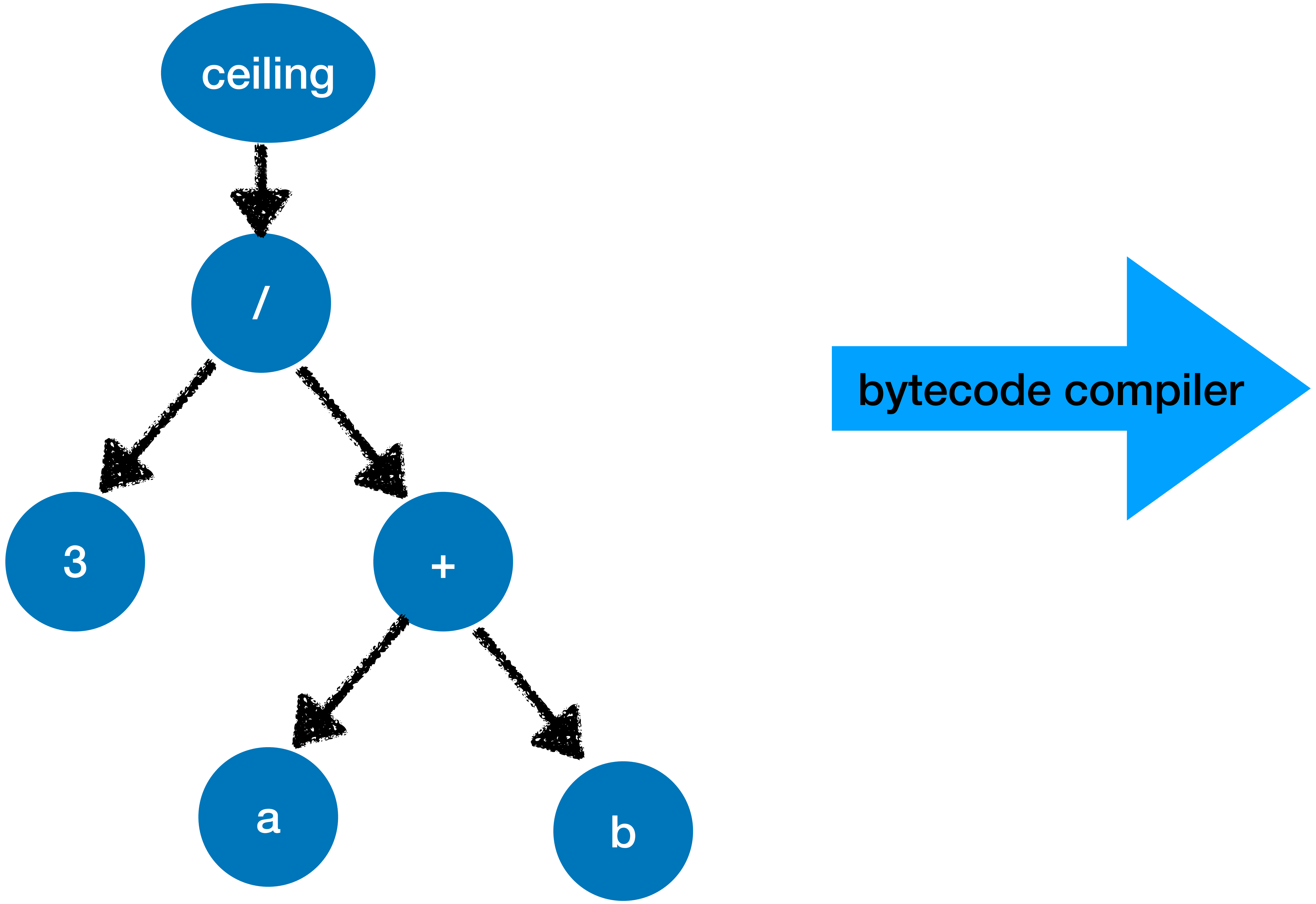

```
(3 / (a + b)) . ceiling()
```

- You need an AST implementation!
- “Fat” representation
- “Decoding instructions” is expensive: Execution needs to jump here and there between nodes

Bytecode

Stack-based linear code

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




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

Bytecode

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

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

- Each operation produces or consumes values into/from the stack
- Compact linear representation
- Execution “falls” from one instruction to the next one

Bytecode

An example

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

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

the stack

Bytecode

An example

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

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

the stack

Bytecode

An example

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

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

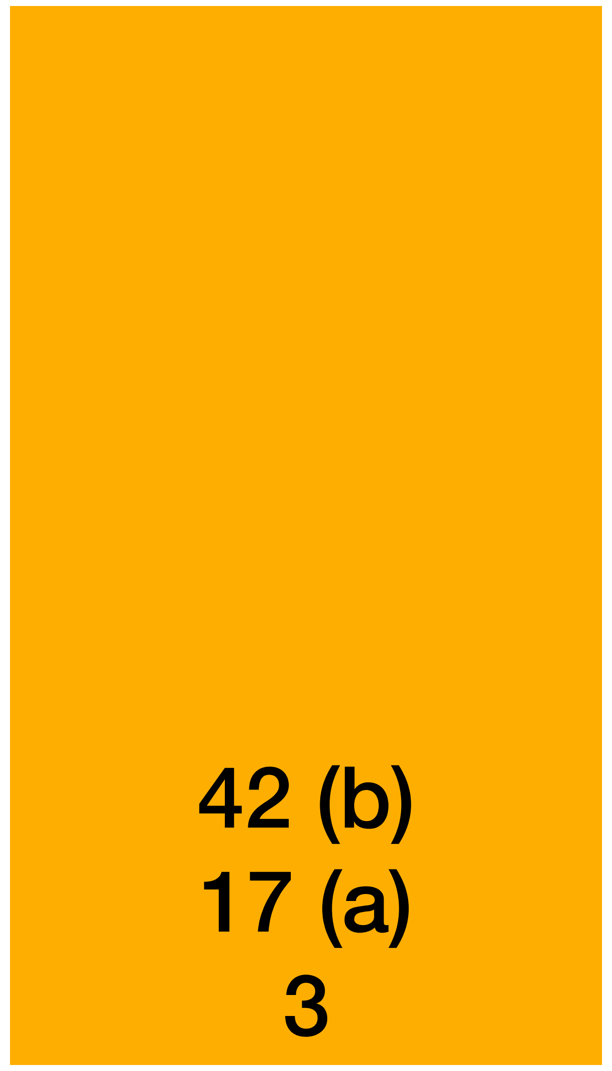
the stack

Bytecode

An example

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

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



the stack

Bytecode

An example

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

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

the stack

Bytecode

An example

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

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

the stack

Bytecode
An example

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

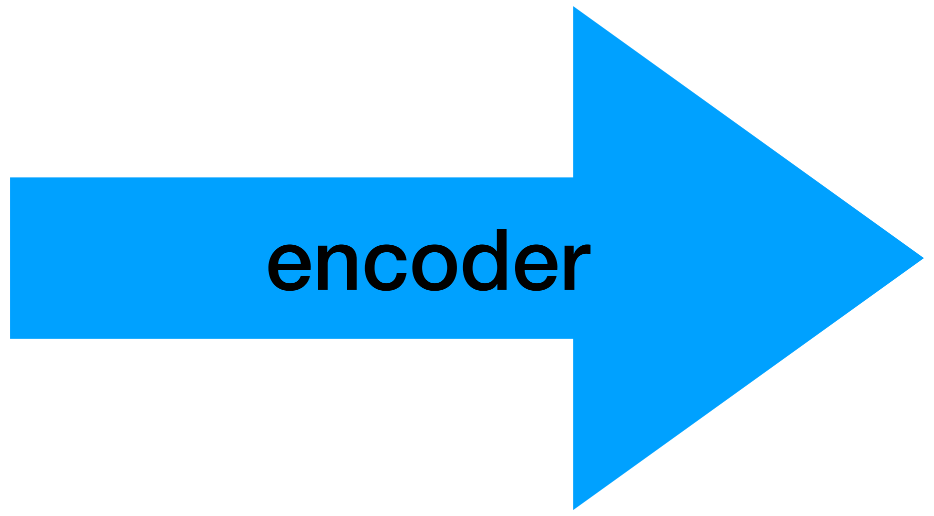
```
push 3
push a
push b
send +
send /
send ceiling
```

the stack

Binary Bytecode Representation

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

conceptual bytecodes




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

opcodes

Encoding bytecodes

- Each kind of bytecode will have an *opcode* or “operation code”
- Opcodes may be of fixed size (e.g., all 1 byte) or variable size (e.g., all different)
- Important! They must be non-ambiguous => the bytecode interpreter should be able to determine what to do from an opcode

Representing Control Flow

- Conditionals and loops alter the order of execution
- Both can be represented with two kind of instructions:
  - **conditional jumps:** move the program counter to some other point
  - **unconditional jumps:** move the program counter if some condition is met

- Jumps may have an absolute program counter to jump to, or a relative offset
- Loops are generally modelled with backward jumps, or *“backjumps”*

Representing Control Flow

Example if

```
if (cond) {
  //A
} else {
  //B
}
//C
```

```
1: push cond
2: jumpIfFalseTo 5
3: // code for A
4: jumpTo 6
5: // code for B
6: // code for C
```

Representing Control Flow

Example loop

```
while (cond) {
  //A
}
// B
```

```
1: push cond

2: jumpIfFalseTo 5

3: // code for A

4: jumpTo 1

5: // code for B
```

Bytecode Families

- Many bytecodes will do *the same* but with different parametrization
- E.g., push constant 1, push constant 2, push constant ‘my string’

- This means we could have an operation “push constant” with a parameter
- These can be encoded as
  - one byte for the opcode and one byte for the parameter or;
  - one byte that has the opcode (5 bits) and the parameter (3 bits)

Generic vs Specific Bytecodes

- Case: reading a variable should look it up in the scope chain
    - But! this lookup can be pre-computed at compile-time
- Option 1: have a generic bytecode “read variable”
    - then let the interpreter lookup variables at runtime
- Option 2: have many specific bytecodes
    - read local, read field/instance variable, read global …
    - decide what opcode to use at compile time

There is more than the bytecode

Meta-data

- We need a binary format including class declarations, method declarations…
- The “bytecode” will be only inside the methods
- This means meta-data needs to be encoded too
  - (and be non-ambiguous)

```
beginClass Person
  beginMethod sumThree
    pushConstant 1
    pushConstant 2
    send +
    returnTop
  endMethod
  …
endClass
```

There is more than the bytecode

Literals

- Literals in the code need to be encoded somehow
- One possibility:
  - put all the literals in a table
  - have a bytecode *pushLiteral indexInTheTable*
- Literal tables can be stored per method, per class, per file…

```
literal table
1 “my String”
2 42.75007

…
beginMethod foo
 pushLiteral 1
 send size
 pushLiteral 2
 send +
 returnTop
endMethod
```

Common bytecode optimisations

- Common long bytecodes could have shorter versions
  - Compact bytecodes and literals tables
  - e.g., `pushTrue` instead of `pushConstant true`
- Common sequences can have a special combined bytecode
  - Compact size of methods + less bytecode fetch overhead
  - e.g., a bytecode *storeAndPop* combining (`store, pop`) sequence
  - e.g., a bytecode *returnTrue* combining (`push true, returnTop`) sequence

Conclusion

- Bytecode is generally used to represent a stack-based linear code
- Compact and linear representation
- Execution falls through
- Except for conditionals and loops that modify the control-flow
  => rely on jumps
- Different designs lead play with complexity to achieve compactness, speed…
- Moreover, in general bytecode needs to have associated meta-data