AST Interpreters 2

Stack Management

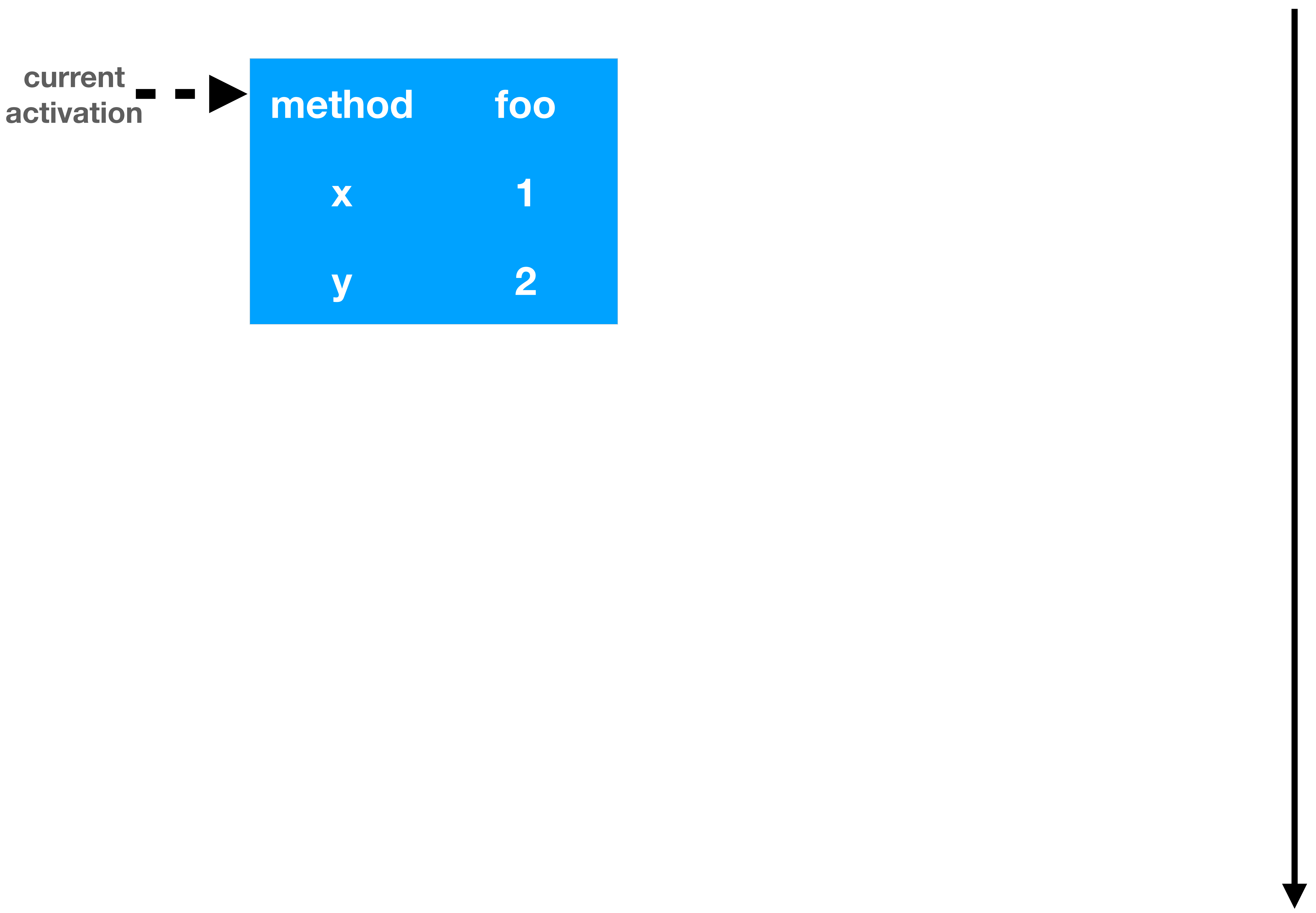
Pablo Tesone - Guille Polito - Fundación Uqbar

Method activations have state

Example

```
foo(1)

function foo(init){
 var x = init;
 var y = x + 1;
 bar(y);
}

function bar(what){
 if (what == 10){
  return;
 }
 foo(what)
}
```

1) execution suspended here



current activation

method	foo
x	1
y	2

drawing convention: stacks grow down

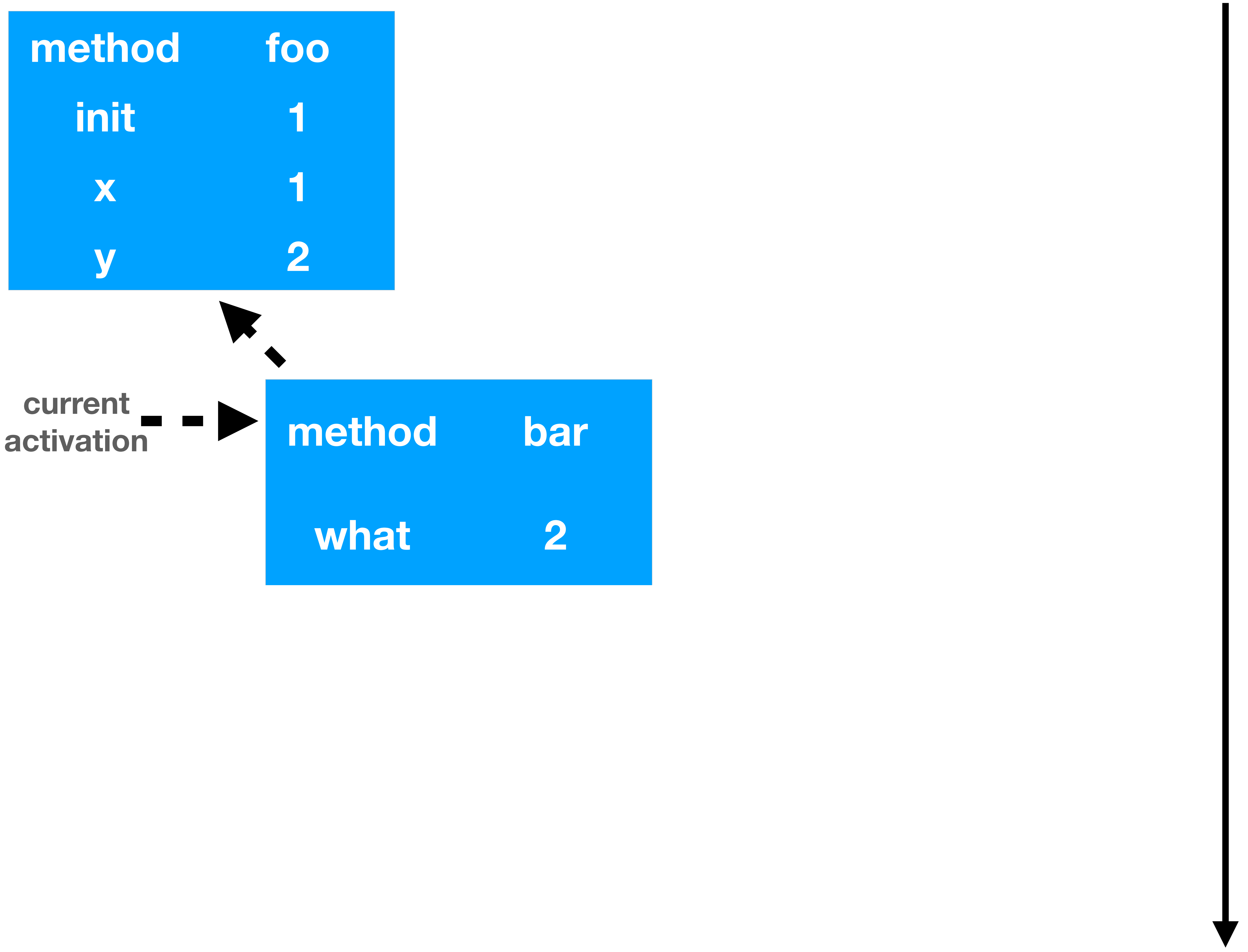
Activations chain form the “stack trace”

Example

```
foo(1)

function foo(init){
 var x = init;
 var y = x + 1;
 bar(y);
}

function bar(what){
 if (what == 10){
  return;
 }
 foo(what)
}
```

1) execution suspended here

2) execution suspended here



method	foo
init	1
x	1
y	2

current activation

method	bar
what	2

drawing convention: stacks grow down

Method Activations to Recursion

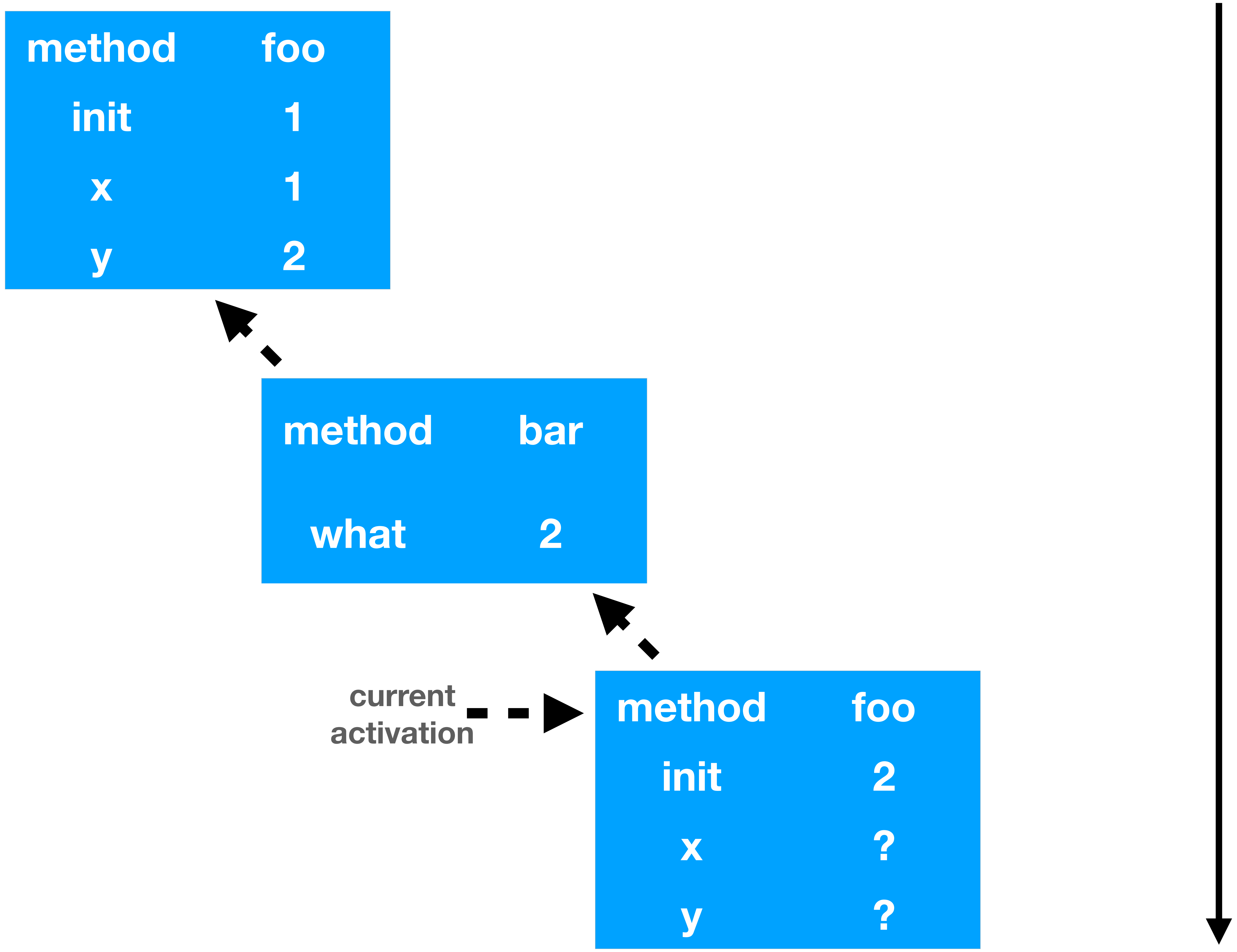
Example

```
foo(1)

function foo(init){
 var x = init;
 var y = x + 1;
 bar(y);
}

function bar(what){
 if (what == 10){
  return;
 }
 foo(what)
}
```

3) execution suspended here → `var x = init;`

1) execution suspended here → `bar(y);`

2) execution suspended here → `foo(what)`



method	foo
init	1
x	1
y	2

method	bar
what	2

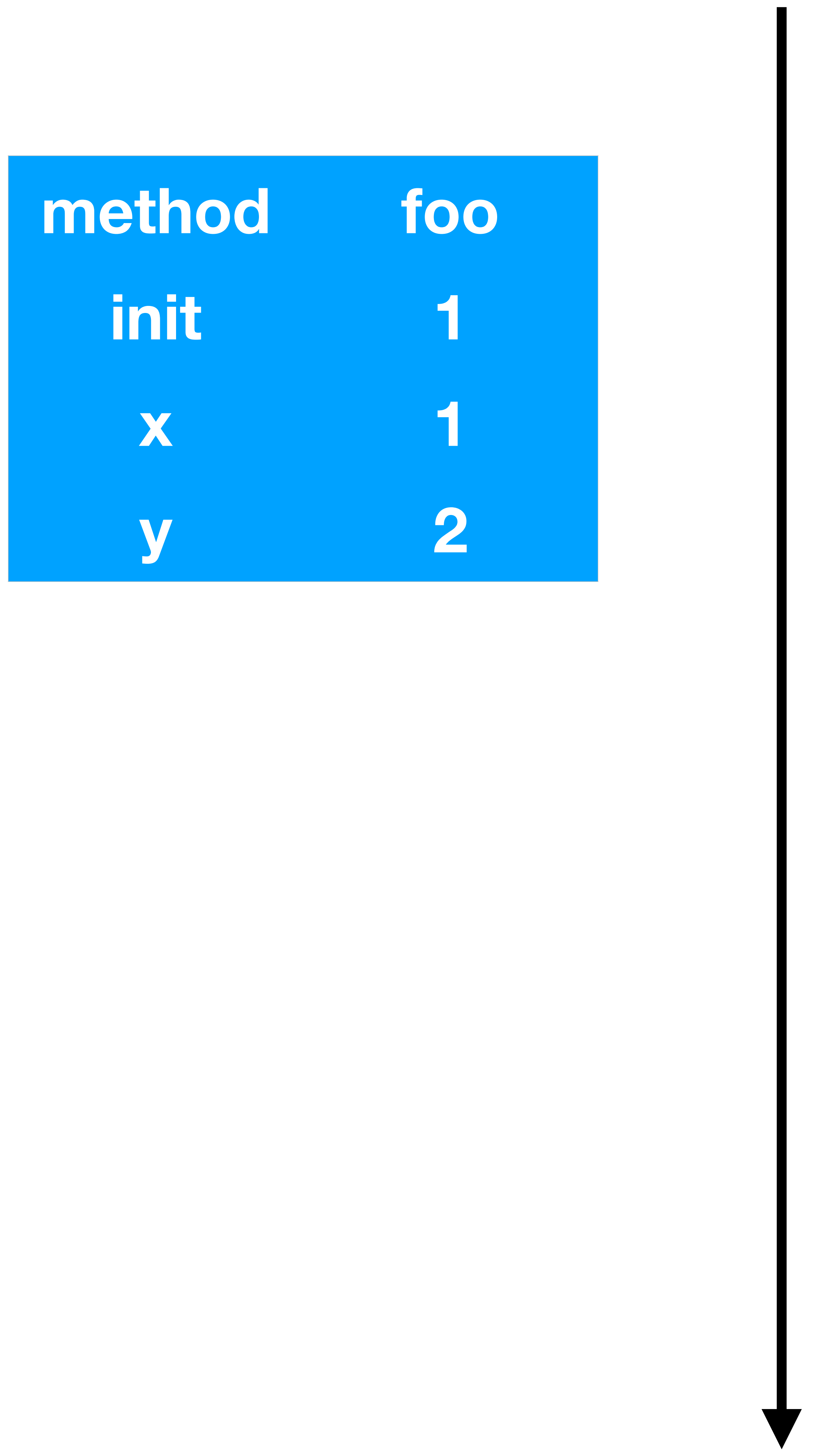
method	foo
init	2
x	?
y	?

current activation

drawing convention: stacks grow down

What to put on a method activation?

- Execution state, debugging information…
    - receiver, temporary variables
    - intermediate values, subexpressions
        - e.g., `result = n * (factorial(n-1))`
    - the program counter
    - the method being executed
    - exception handling data, meta-data, flags…
    - whatever your language needs to be executed :)



method	foo
init	1
x	1
y	2

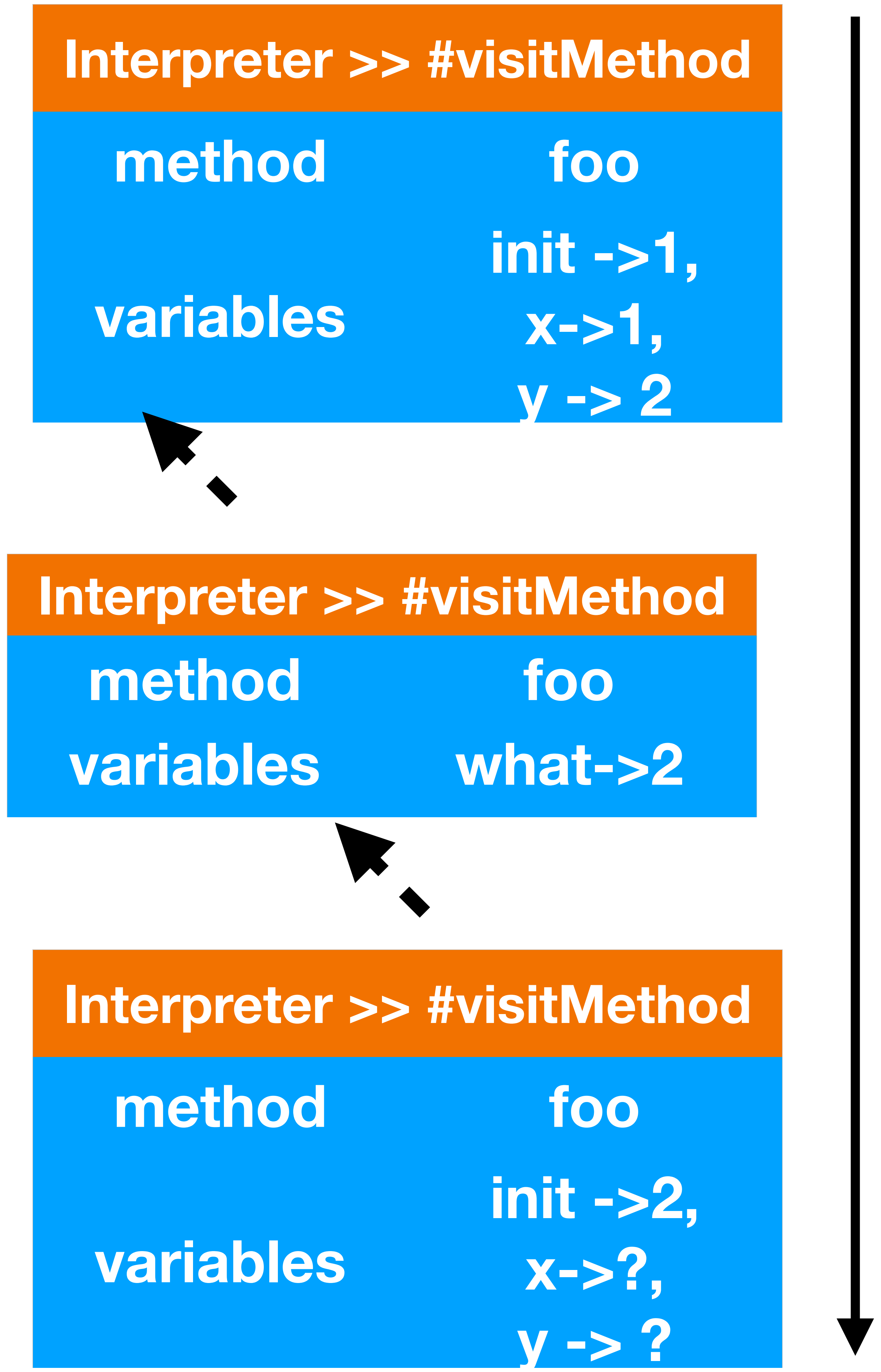
drawing convention:
stacks grow down

Call Stack Implementations

Using Host Language Stack

- Simple Implementation
- State stored in local variables in the interpreter
- We use the same existing stack
- We keep the state in local variables
- We depend on the host language
- Difficult / impossible to manage
- We need a recursive implementation
- Limits interesting features: exceptions, ensure blocks, reification, non-local return

```
foo(1)
function foo(init){
 var x = init;
 var y = x + 1;
 bar(y);
}

function bar(what){
 if (what == 10){
  return;
 }
 foo(what)
}
```



Interpreter >> #visitMethod	
method	foo
variables	init ->1, x->1, y -> 2

Interpreter >> #visitMethod	
method	foo
variables	what->2

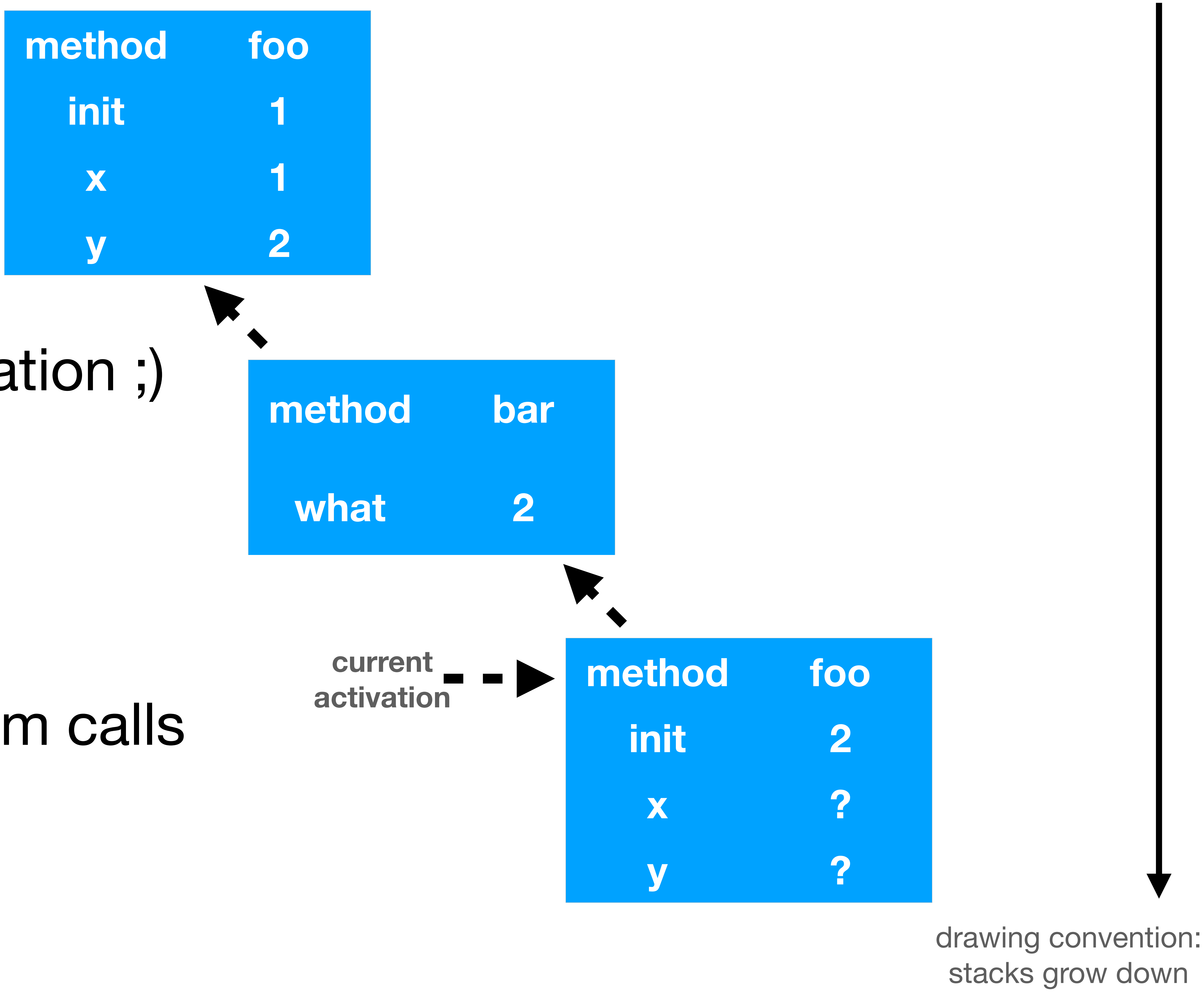
Interpreter >> #visitMethod	
method	foo
variables	init ->2, x->?, y -> ?

drawing convention:
stacks grow down

Call Stack Implementations

Heap allocated

- using malloc or new
- easy to understand and manage
  - so very good for a first implementation ;)
  - e.g., using a linked list
- cons: could be very slow
  - de-allocation requires GC or system calls
  - poor locality

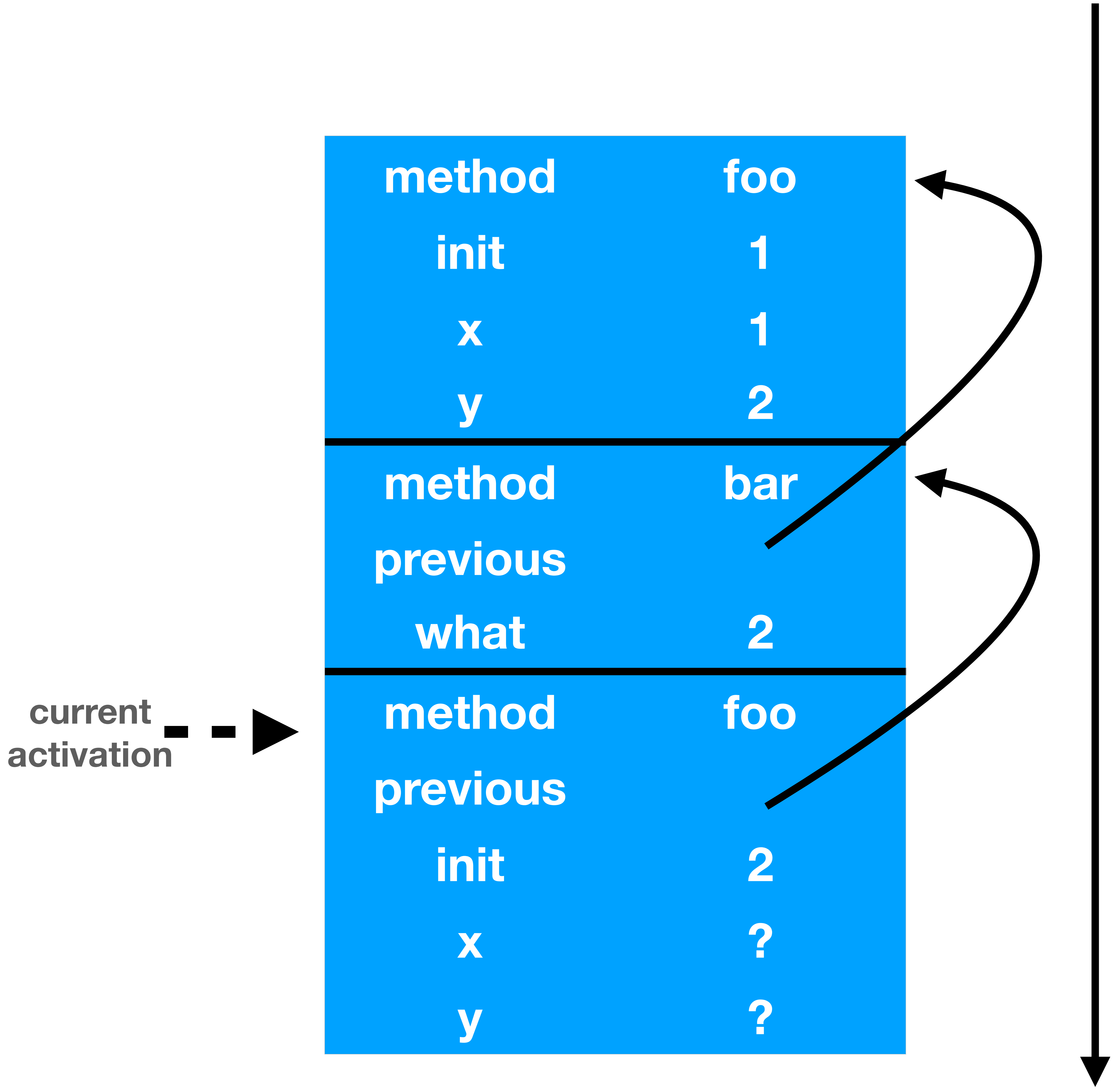


drawing convention:
stacks grow down

Call Stack Implementations

Contiguous Stack

- single contiguous chunk of memory
- activations are implicit!
  - the abstraction is now hidden
- but it is fast
  - deallocation is just moving one pointer
  - great locality

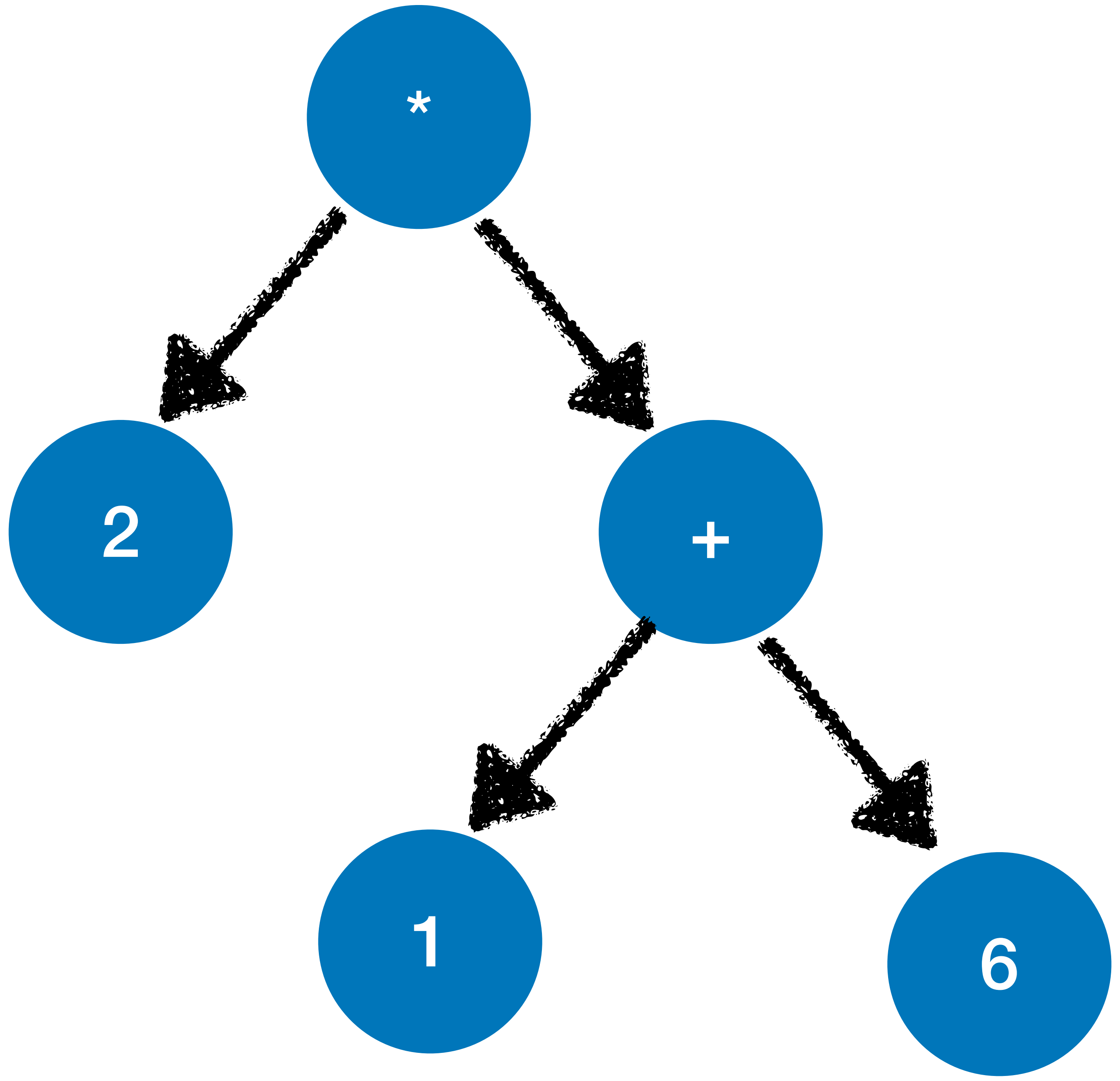


method	foo
init	1
x	1
y	2
method	bar
previous	
what	2
method	foo
previous	
init	2
x	?
y	?

drawing convention:
stacks grow down

Managing subexpressions

- The result of subexpressions need to be stored somewhere!
  - e.g., `2 * (1 + 6)`
- Two main options appear:
  - hold them in interpreter variables
  - hold them in an operand stack

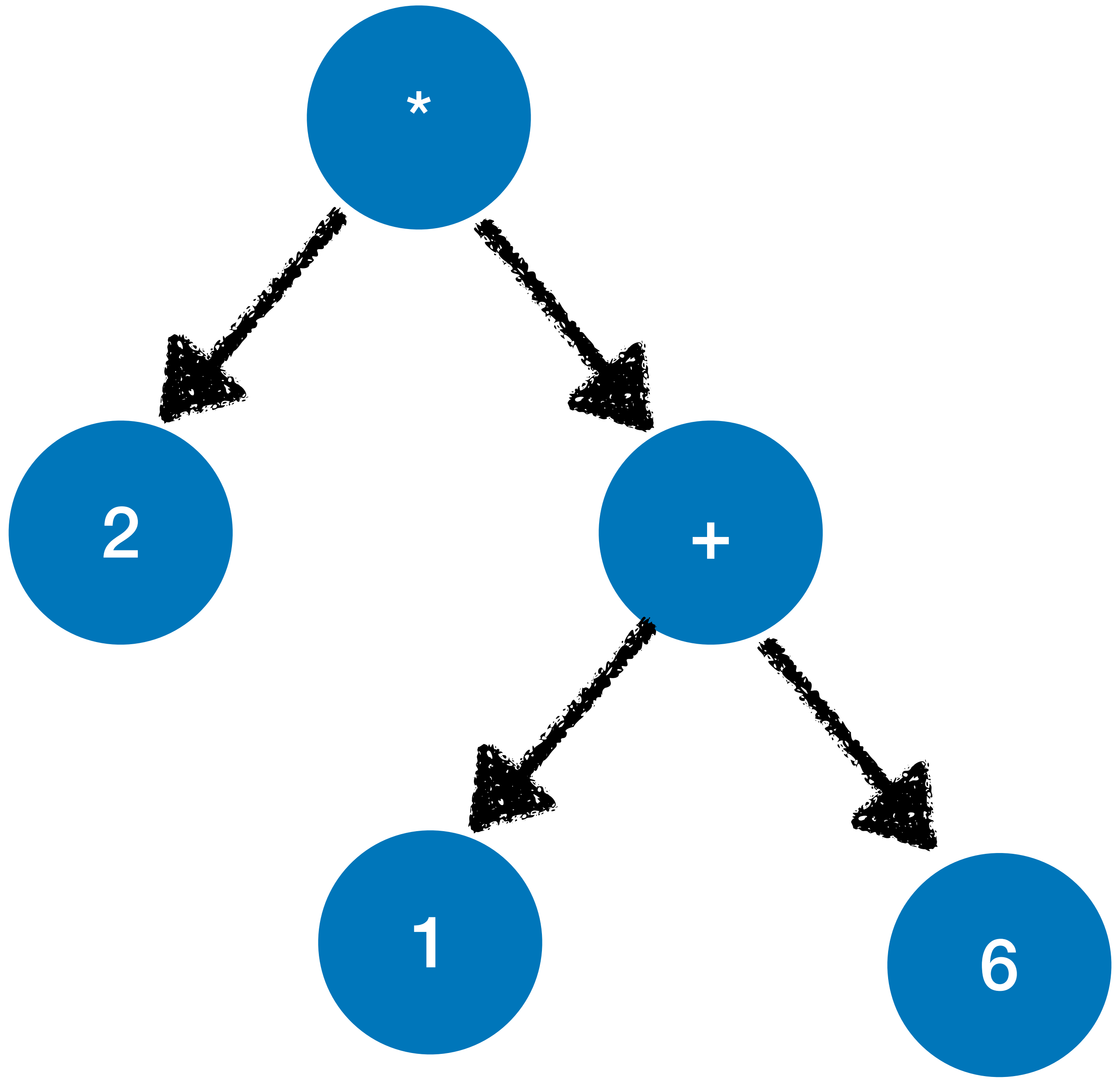


Subexpressions in interpreter variables

The interpreter (host) stack

- Subexpression results stored in interpreter temps

```
visitMultiplication: aMultiplication
  | leftOperand rightOperand |
  leftOperand := self visit: aMultiplication left.
  rightOperand := self visit: aMultiplication right.
  ^ leftOperand * rightOperand
```



- Simple solution, works with recursive implementations

Subexpressions in a stack

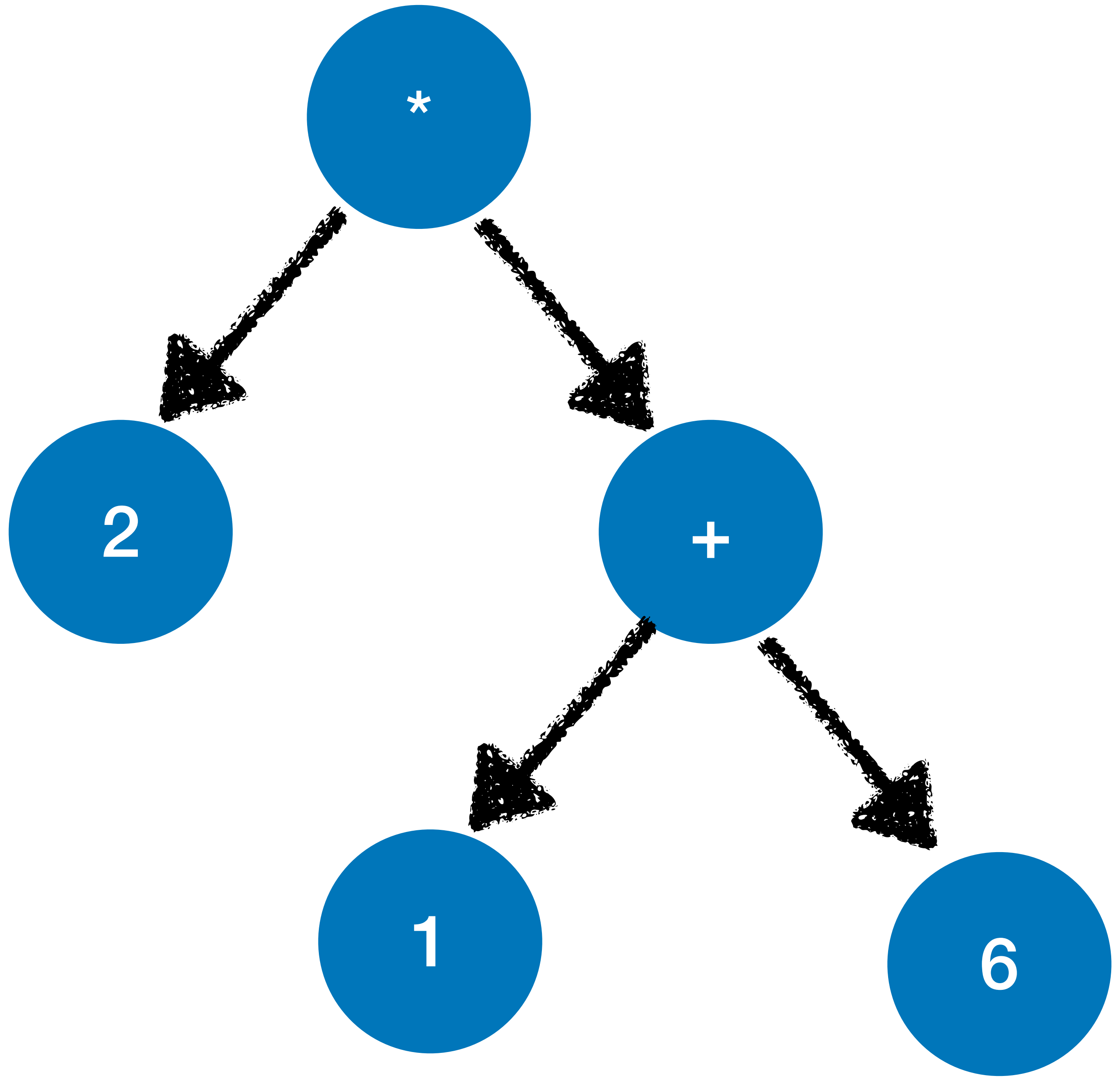
The operand stack

- Subexpression results stored in a stack per activation

1) execution suspended here

```
visitMultiplication: aMultiplication
  self visit: aMultiplication left.
  self visit: aMultiplication right.
  ^ self pop * self pop
```

- Works with non-recursive implementations
- Simplify debugger implementation



method	foo
init	1
t1	1
t2	2
stack-0	2
stack-1	7

Conclusion

- Method activations are organised in a stack
- They store the program execution’s state, and any other required meta-data
- Different designs lead to simpler, complex, faster or slow implementations
- Particular attention needs to be taken with the results of subexpressions!