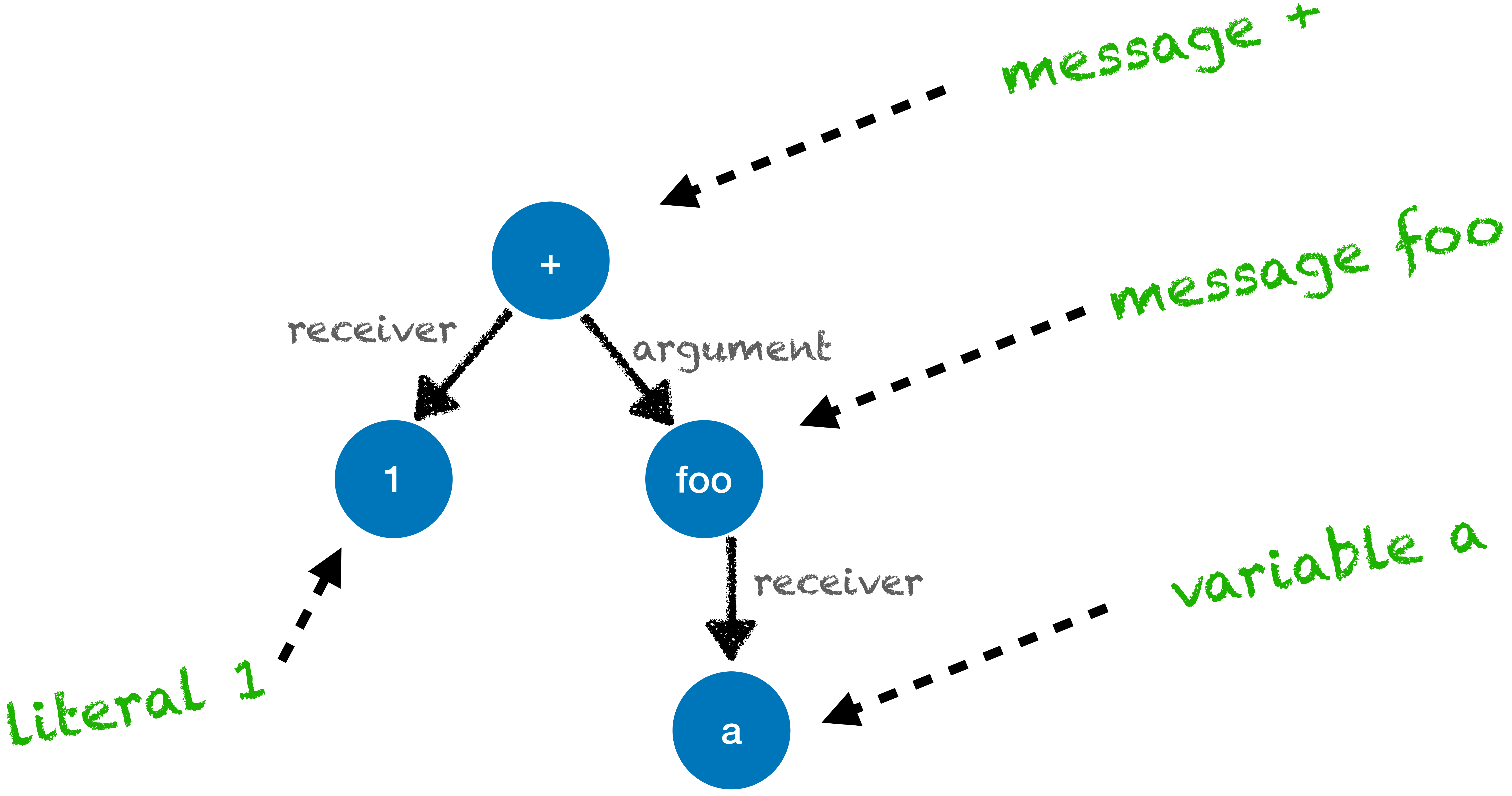
AST Interpreters

Guille Polito - META

Reminding ASTs

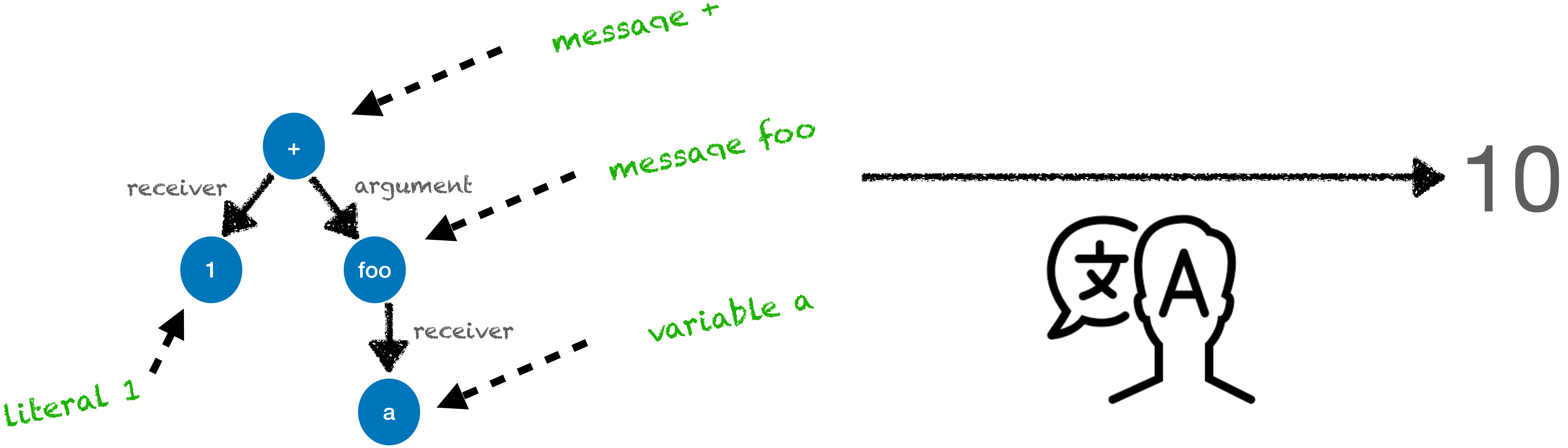
Example

```
1 + a foo
```



AST Interpreters

- A program that takes ASTs and evaluates them to some value



Why AST interpreters?

- ASTs are simple to manipulate

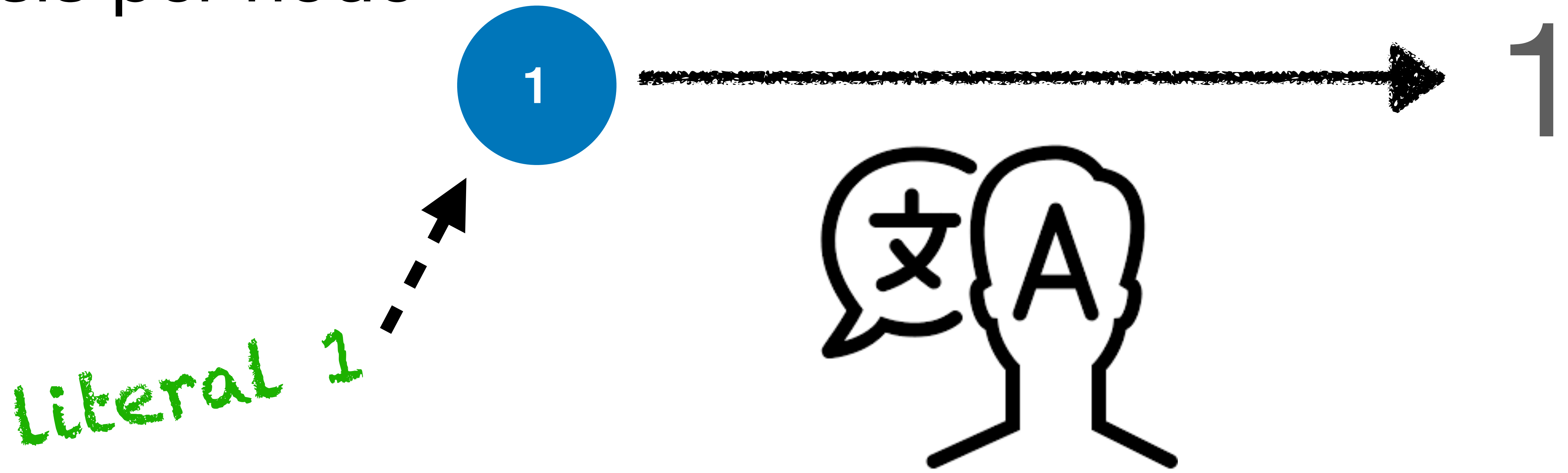
  => AST interpreters are easy to write

- AST interpreters can have many shapes
  - Evaluator: *executes* the program and returns its result
  - Abstract interpreters / symbolic executors:
    - do approximate executions on “mock” values
  - Compilers can be build as interpreters!

Adding Semantics to the Syntax

Example of an evaluator

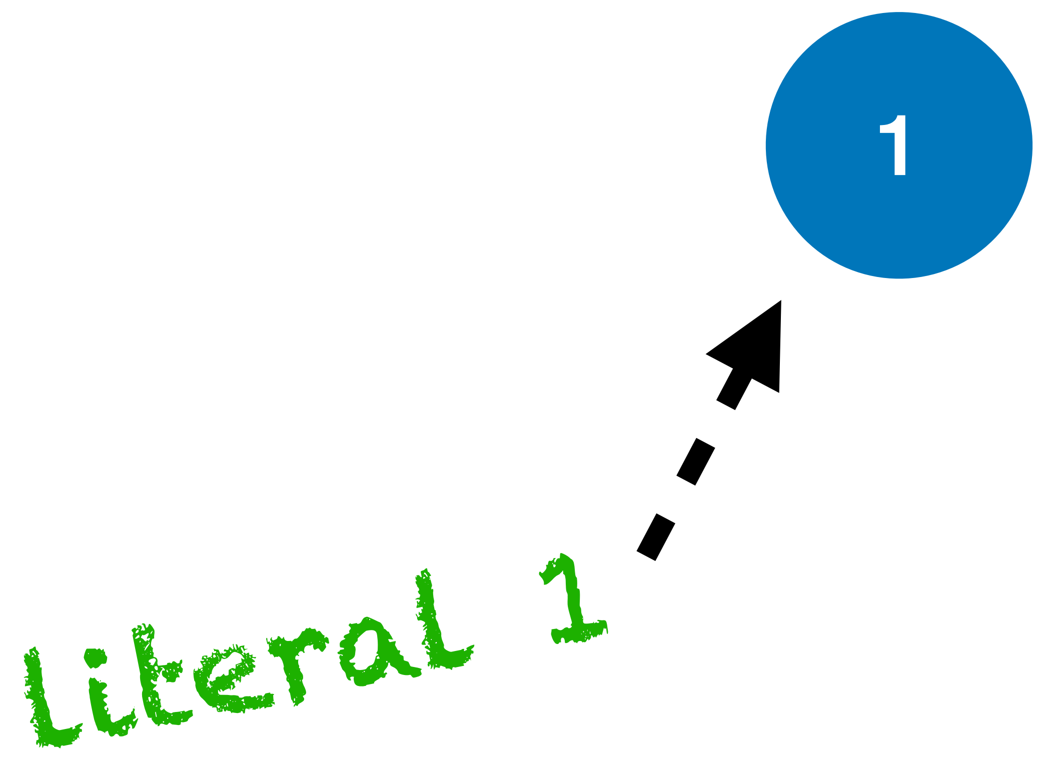
- AST nodes do not have semantics attached
- It is the interpreter that says what to do with each node
- E.g., in an evaluator each node is reduced to a value
- The interpreter does case analysis per node
  - using, e.g., a visitor pattern



Evaluating Literals

- The value of a literal node is the parsed value

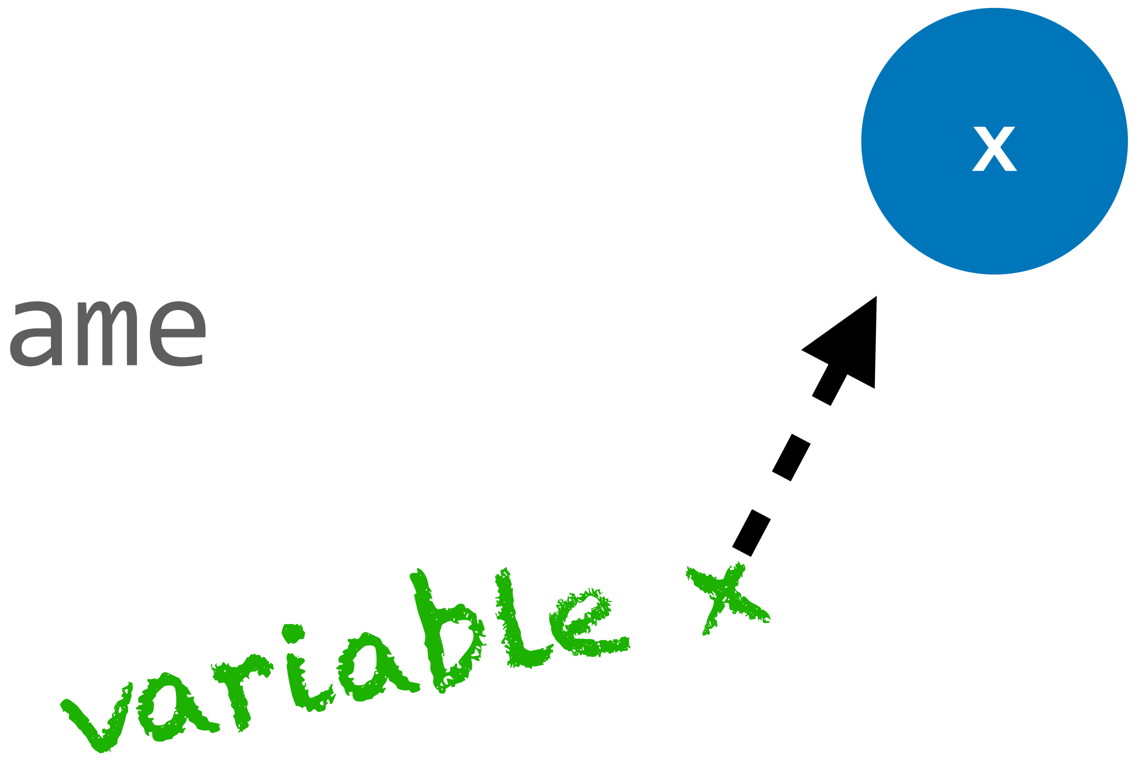
```
visitLiteralNode: aNode
  ^ aNode value
```



Evaluating Variables

- The value of a variable node is the value stored in some memory location
- E.g., the value of instance variable #x has to fetch it from the receiver object

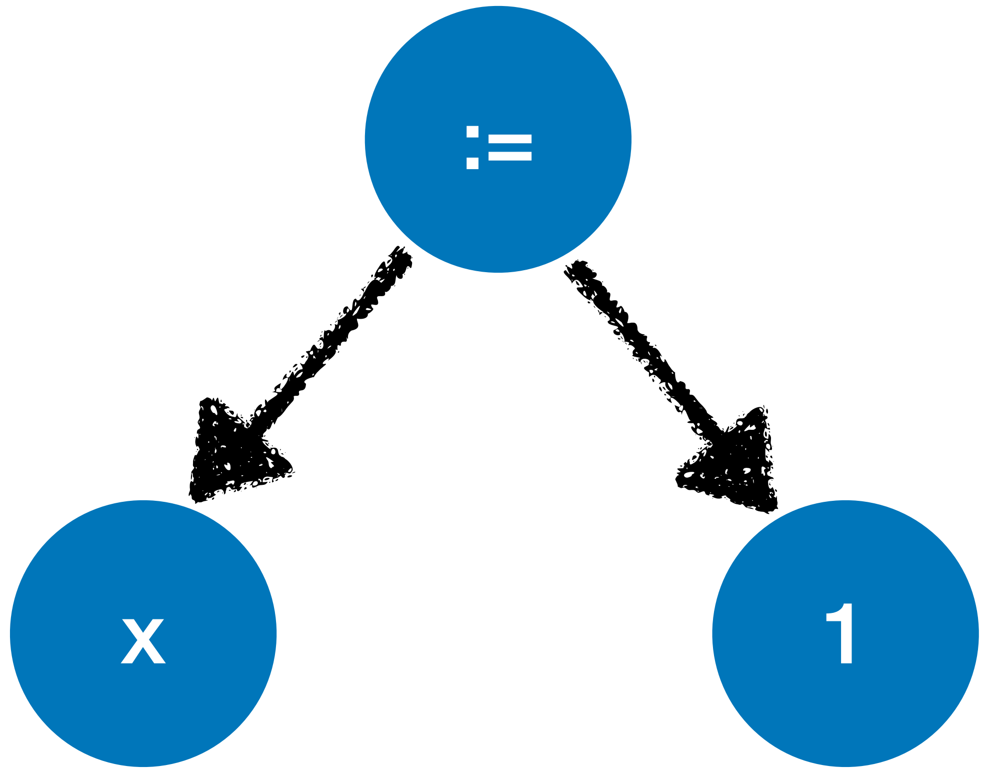
```
visitVariableNode: aNode
  ^ receiver instVarNamed: aNode name
```



Evaluating Assignment

- An assigment has an effect! It stores the evaluation of the RHS on the LHS
- It also has a value: its value is the value stored

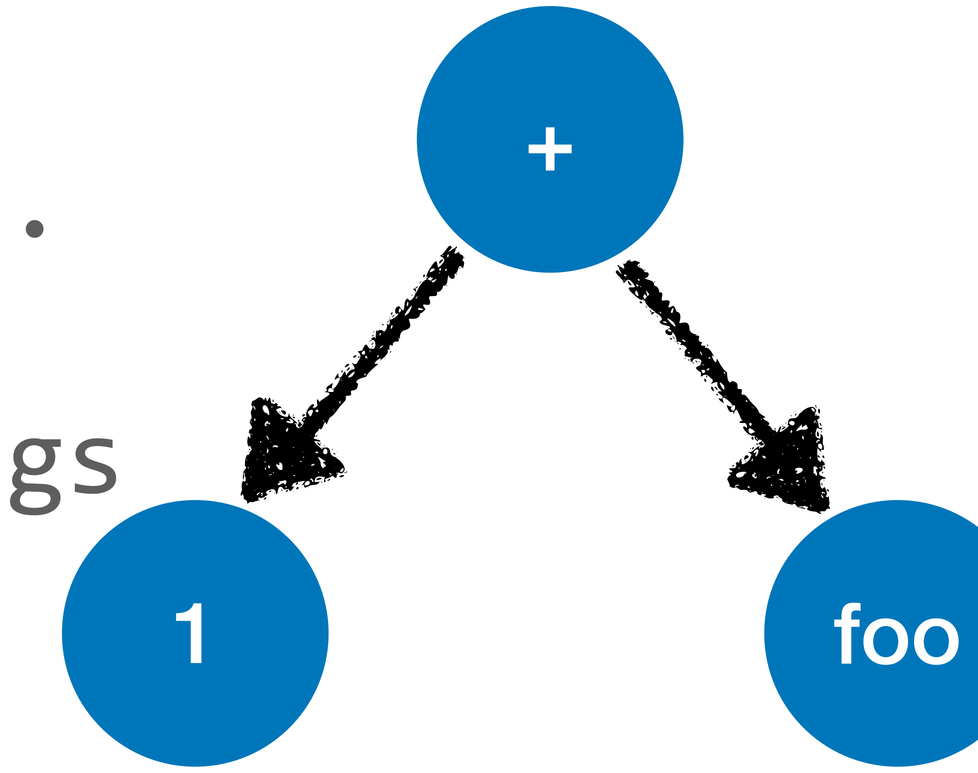
```
visitAssignmentNode: aNode
  ^ receiver
      instVarNamed: aNode variable name
      put: (aNode value acceptVisitor: self)
```



Evaluating Messages

- The value of a message node is the value returned by the fact of invoking a method
- Given the receiver, we must *lookup* the method corresponding to the selector
- Then evaluate that method using the receiver as *self*
- E.g., the value of instance variable #x has to fetch it from the receiver object

```
visitMessageNode: aNode
  receiver := aNode receiver acceptVisitor: self.
  args := aNode arguments collect: [:aNd | aNd acceptVisitor: self].
  method := self lookup: aNode selector in: receiver class.
  ^ self evaluateMethod: method withReceiver: receiver withArgs: args
```



Implementing the method lookup

Recursive definition

```
lookup: aSymbol fromClass: aClass

   (aClass includesSelector: aSymbol)
      ifTrue: [ ^ (aClass compiledMethodAt: aSymbol) ast ].

   ^ aClass superclass
      ifNil: [ nil ]
      ifNotNil: [ self lookup: aSymbol fromClass: aClass superclass ]
```

Questions? Refresh with the MOOC or your OOP course

Exercising

- First extending the Pharo AST visitor
- Second creating our own

Let us practice: Building a Message counter

```
testMessageCount

  “Point >> sideOf: otherPoint
  | side |
  side := (self crossProduct: otherPoint) sign.
  ^ { #right . #center . #left } at: side + 2   "


  | ast counting |
  ast := RBParser parseMethod: (Point >> #sideOf:) sourceCode.
  counting := CountingInterpreter new.
  ast acceptVisitor: counting.
  self assert: counting numberOfMessages equals: 4.
```

Reusing the Pharo Visitor

```
RBProgramNodeVisitor subclass: #CountingInterpreter
  instanceVariableNames: 'count'
  classVariableNames: ''
  package: 'myBecher-MetaASTVisitor'
```

Initialization

```
CountingInterpreter >> initialize

  super initialize.
  count := 0.
```

Now counting messages

```
CountingVisitor >> visitMessageNode: aMessageNode
  super visitMessageNode: aMessageNode.
  count := count + 1.
```

Thinking

The Pharo visitor implements the visit of the AST nodes.

Let us do the visitor from scratch

- A bit more difficult but you can learn more

Let us practice: Building a Message counter

```
testMessageCount2

  “Point >> sideOf: otherPoint
  | side |
  side := (self crossProduct: otherPoint) sign.
  ^ { #right . #center . #left } at: side + 2   "


  | ast counting |
  ast := RBParser parseMethod: (Point >> #sideOf:) sourceCode.
  counting := CountingManualInterpreter new.
  ast acceptVisitor: counting.
  self assert: counting numberOfMessages equals: 4.
```

Building a Pharo Visitor

```
Object subclass: #CountingManualInterpreter
  instanceVariableNames: 'count'
  classVariableNames: ''
  package: 'myBecher-MetaASTVisitor'
```

visitMethodNode:

```
visitMethodNode: aMethodNode

  aMethodNode statements do: [ :each | each acceptVisitor: self ]
```

visitMethodNode:

```
visitMethodNode: aMethodNode
  aMethodNode statements do: [ :each | each acceptVisitor: self ]
```

```
visitMessageNode: aRBMessageNode
   count := count + 1.
   aRBMessageNode receiver acceptVisitor: self.
   aRBMessageNode arguments do: [ :each | each acceptVisitor: self ]
```

visitAssignmentNode:

```
visitAssignmentNode: anAssignmentNode
  anAssignmentNode value acceptVisitor: self


visitVariableNode: aRBVariableNode
  ^ self

visitSelfNode: aRBMethodNode

  ^ self

visitLiteralValueNode: aRBLiteralValueNode
   ^ self
```

visitAssignmentNode:

```
visitArrayNode: aRBArrayNode
  ^ self


visitSelfNode: aRBMethodNode
  ^ self

visitLiteralValueNode: aRBLiteralValueNode
    ^ self
```

Preparing the exam

- Redo the Counter interpreters in both forms.
- Pay attention the manual visitor should be enhanced
- Write a visitor + tests to
    - Exo1: Determine whether a method is using self.
    - Exo2: Determine whether a method is not assigning any of its instance variable.