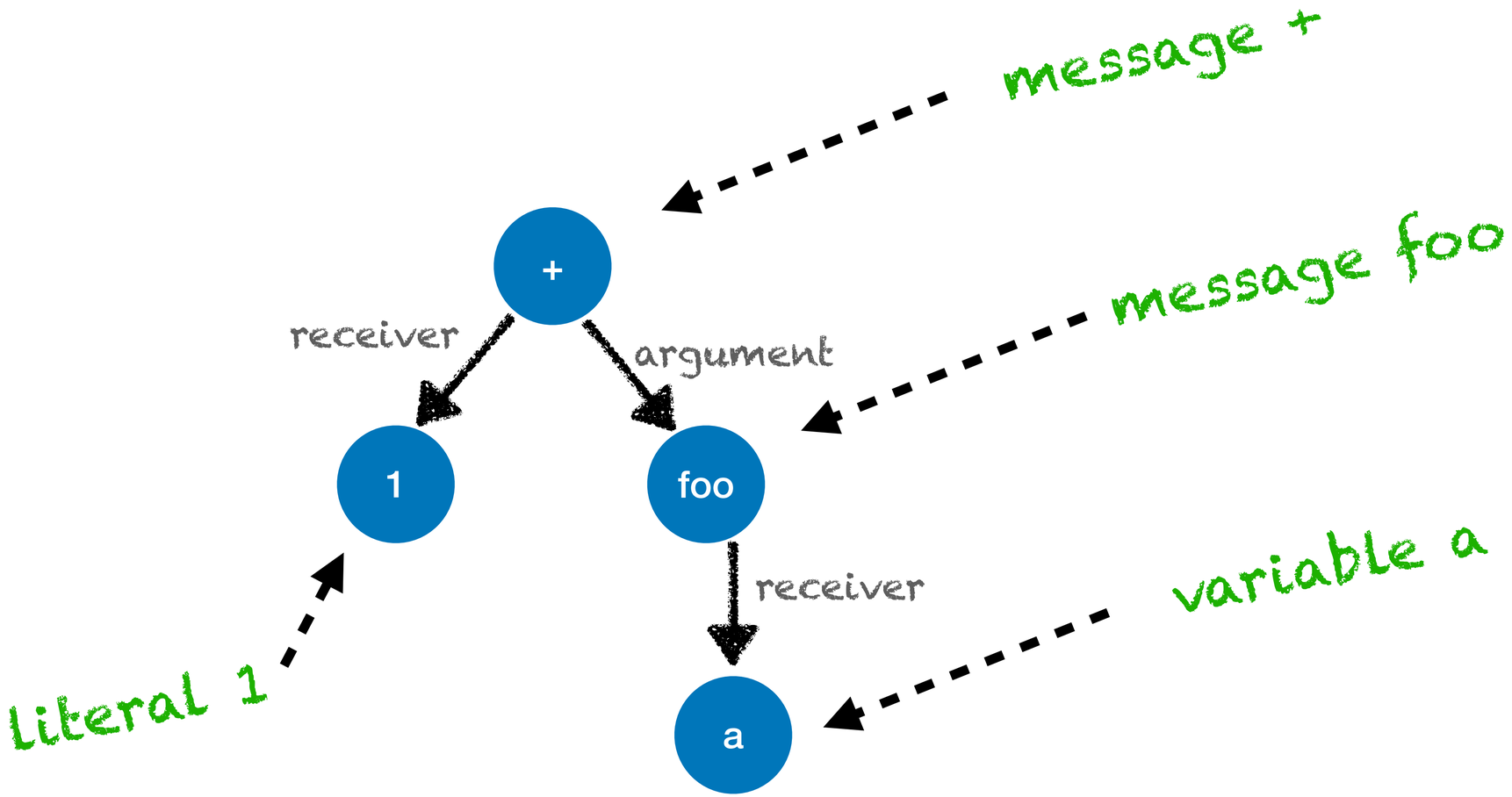
AST Interpreters 101

Guille Polito - META

Reminding ASTs

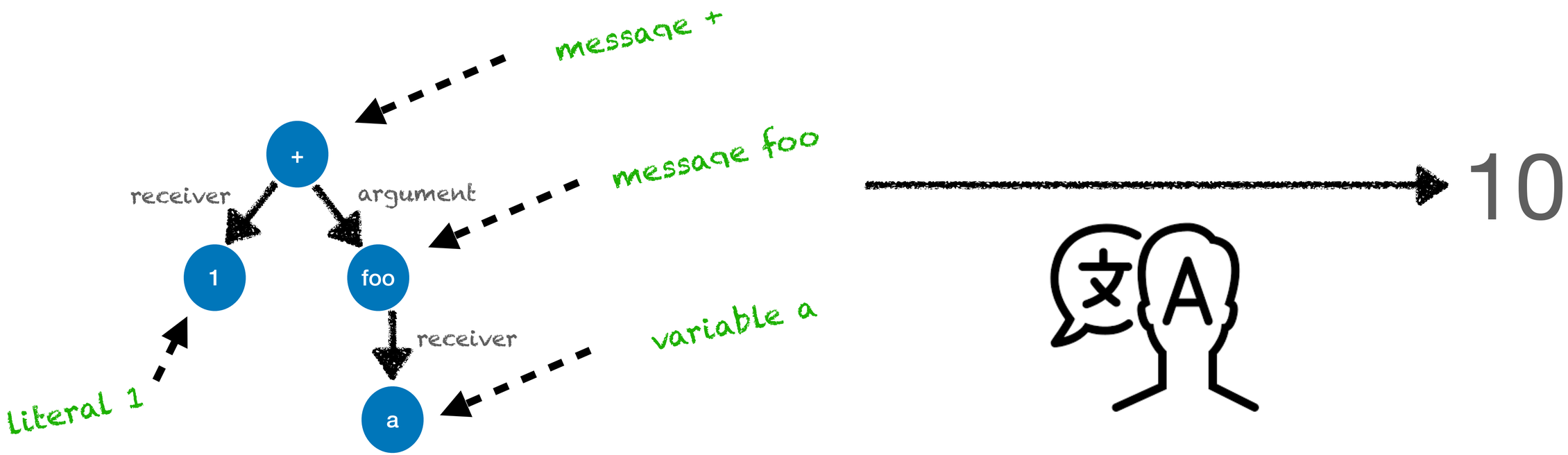
Example

```
1 + a foo
```



AST Interpreters

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



Why AST interpreters?

- ASTs are simple to manipulate

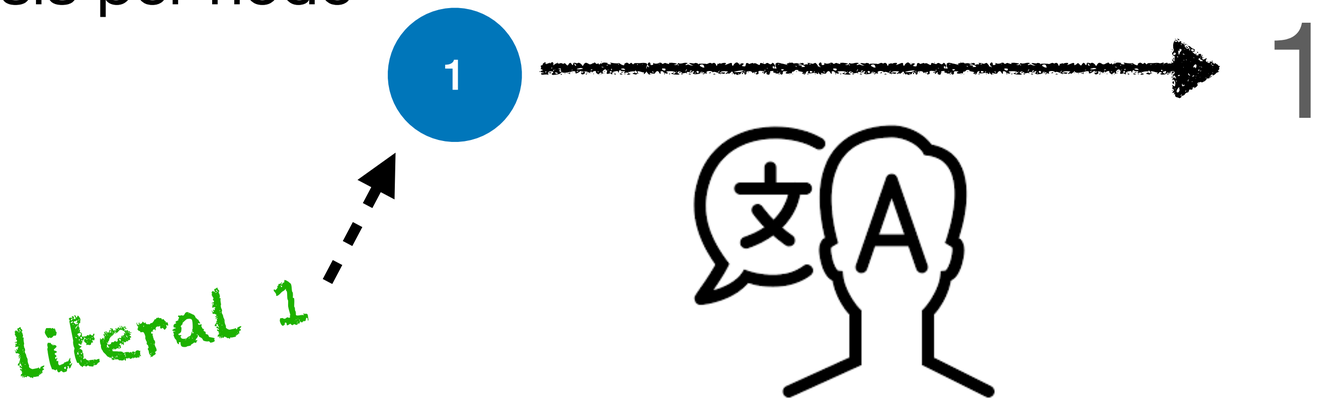
  => AST interpreters are easy to write

- 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

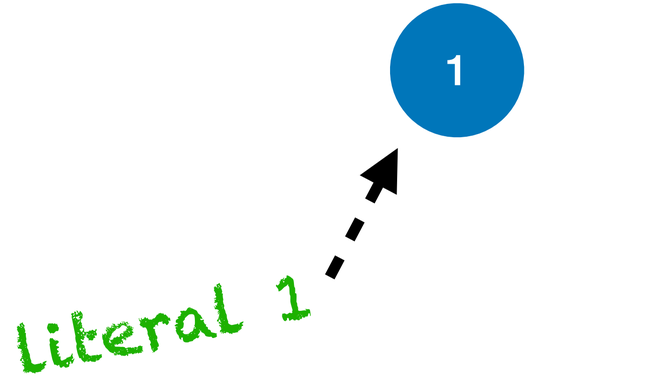
- AST nodes do not have semantics attached
- It is the interpreter that says what to do with each node
- E.g., in an evaluator each node is reduced to a value
- The interpreter does case analysis per node
  - using, e.g., a visitor pattern



Evaluating Literals

- The value of a literal node is the parsed value

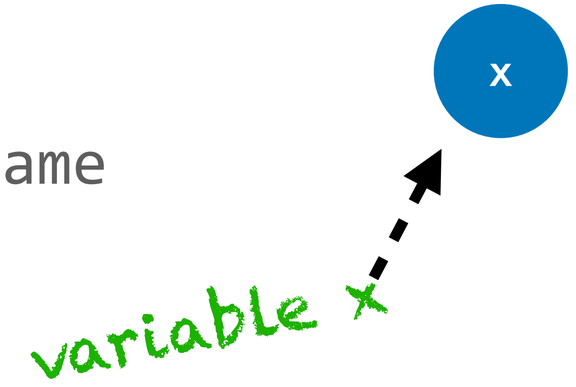
```
visitLiteralNode: aNode
  ^ aNode value
```



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

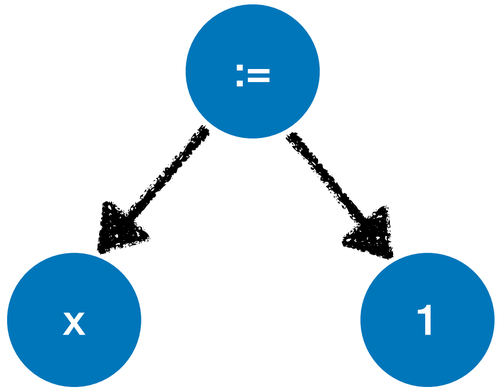
```
visitVariableNode: aNode
 ^ receiver instVarNamed: aNode name
```



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

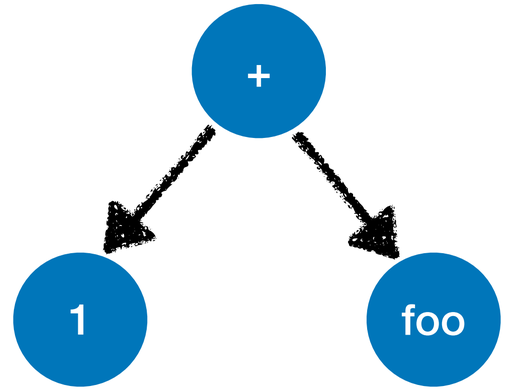
```
visitAssignmentNode: aNode
 ^ receiver
   instVarNamed: aNode variable name
   put: (aNode value acceptVisitor: self)
```



Evaluating Messages

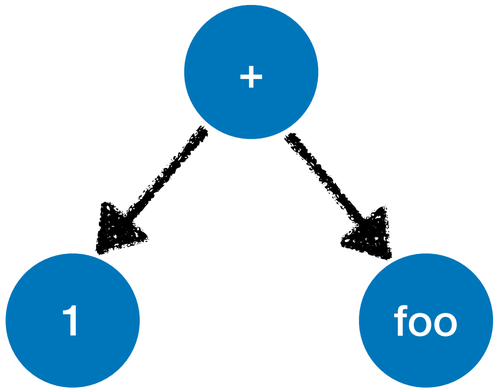
- The value of a message node is the value returned by the fact of invoking a method
- Given the receiver, we must *lookup* the method corresponding to the selector
- Then evaluate that method using the receiver as *self*
- E.g., the value of instance variable #x has to fetch it from the receiver object

```
visitMessageNode: aNode
 receiver := aNode receiver accept: self.
 method := self lookup: aNode selector in: receiver.
 ^ self evaluateMethod: method withReceiver: receiver
```



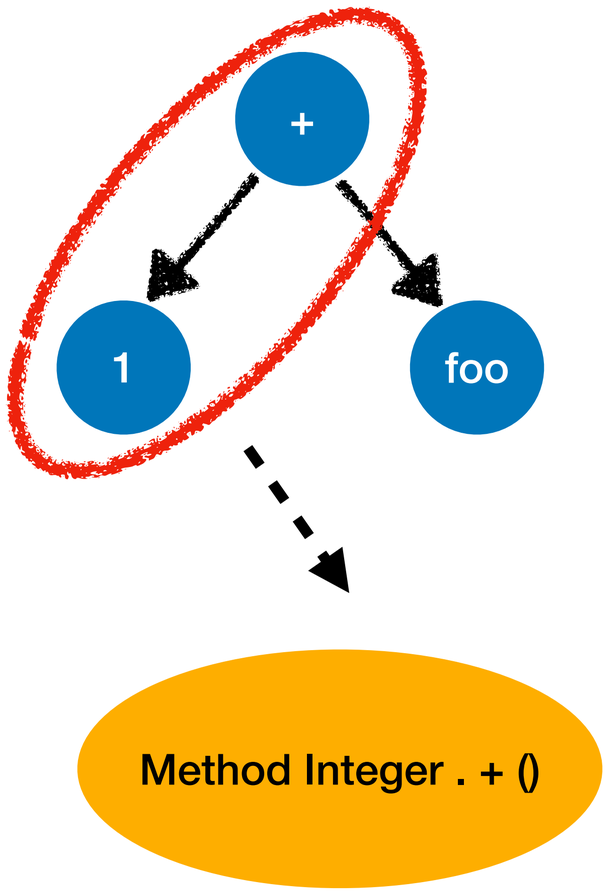
Evaluating Messages

Example



Evaluating Messages

Example



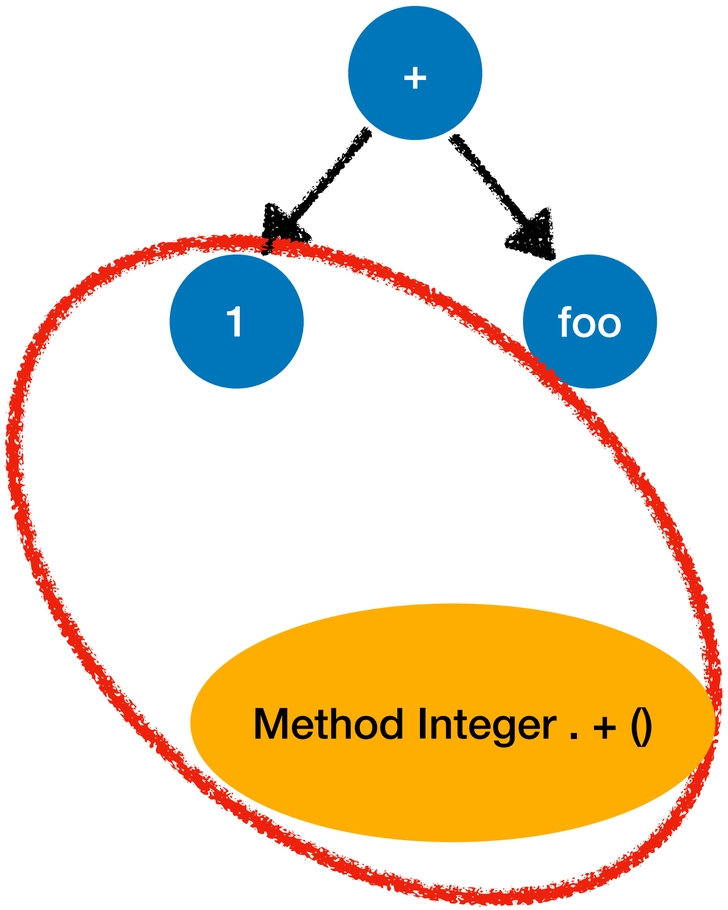
1. Lookup

    1.1. Get the receiver’s class (in class based language)

    1.2. Look in the hierarchy until you find a method that matches the signature of the message

Evaluating Messages

Example



2. Apply

    2.1. Get the method you found

    2.2. Execute it on the receiver (a.k.a. method activation)

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

Method activation

- execute the new method binding:
  - the receiver as the *self/this* special variable
  - each argument (the values) with each parameter (the variables)
  - at the end of the evaluation of the method, return the result to the caller

Native Methods

- Some language implementations include **native methods**:
  - special methods that have an implementation in the *implementation language.*
  - For example, when implementing Pharo in C, C is the implementation language
  - Typically used for special behaviours (low-level arithmetics, object allocation…)

Conclusion

- An AST interpreter does a case analysis per node
- In object oriented languages, the juice is in the message sends
- Each message send activates a new method
- Special ***native*** methods implement low-level behaviour in the implementation language, because it usually cannot be expressed in the interpreted language