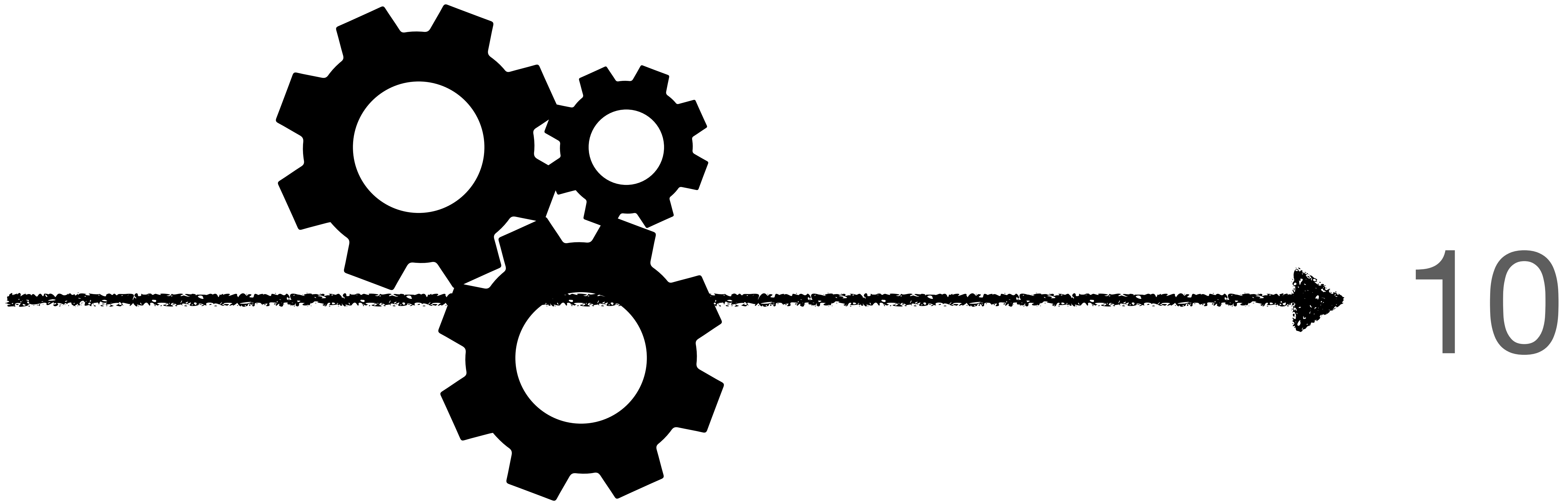
Abstract Syntax Trees

And code representations

Guille Polito - META 2021

Executing Code

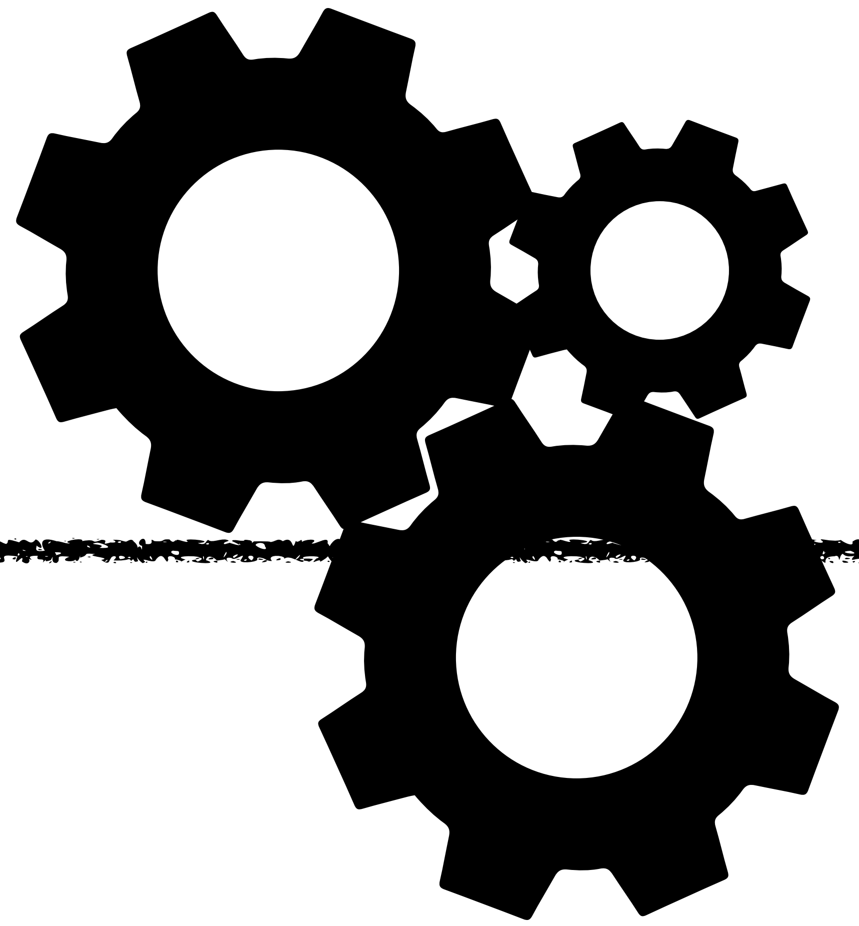
```
a := 1;
if (condition){
  a := a + 7;
}
return a + 2;
```



This is text!

Executing Code

```
a := 1;
if (condition){
  a := a + 7;
}
return a + 2;
```

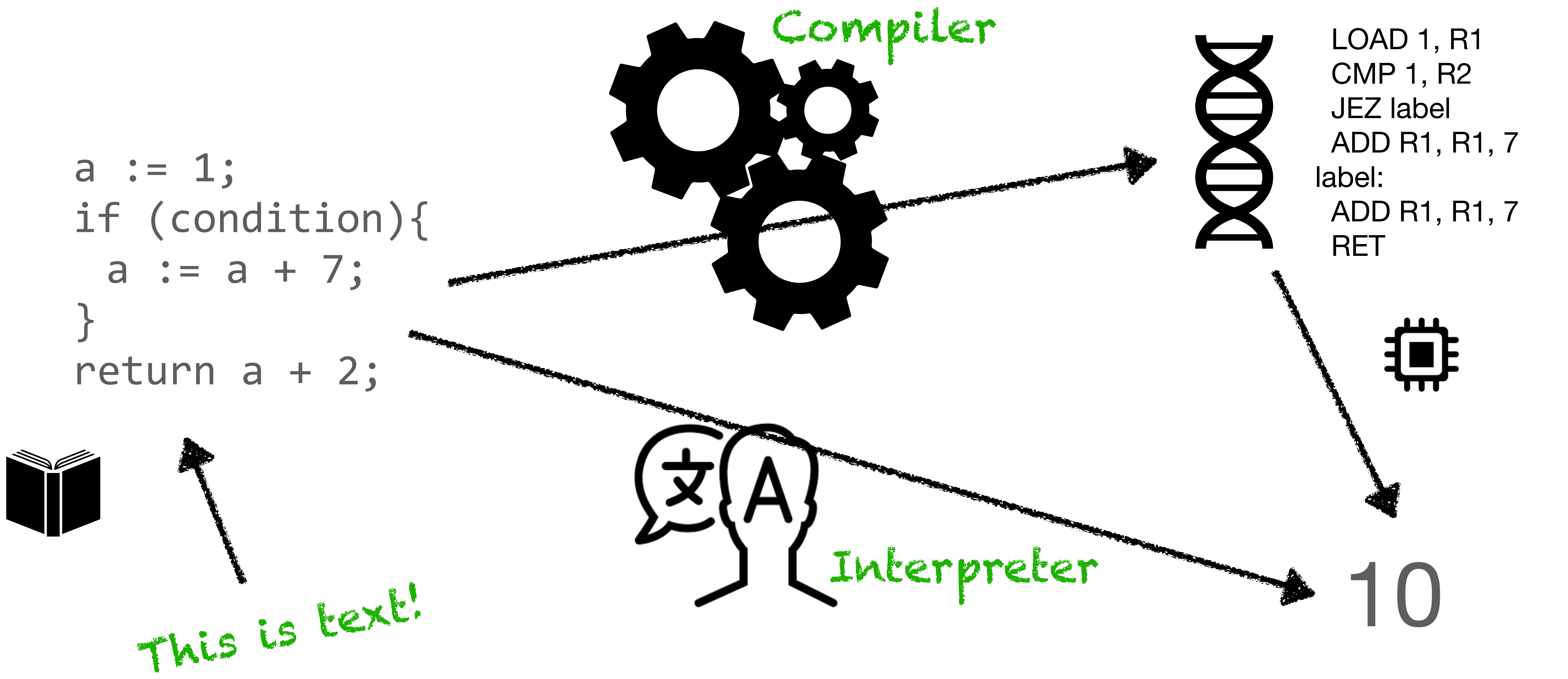


10

This is text!

???????

Compilers vs Interpreters



Modern Languages

Use both compilers AND interpreters!

```
a := 1;
if (condition){
  a := a + 7;
}
return a + 2;
```

Basics of Interpreters and Compilers

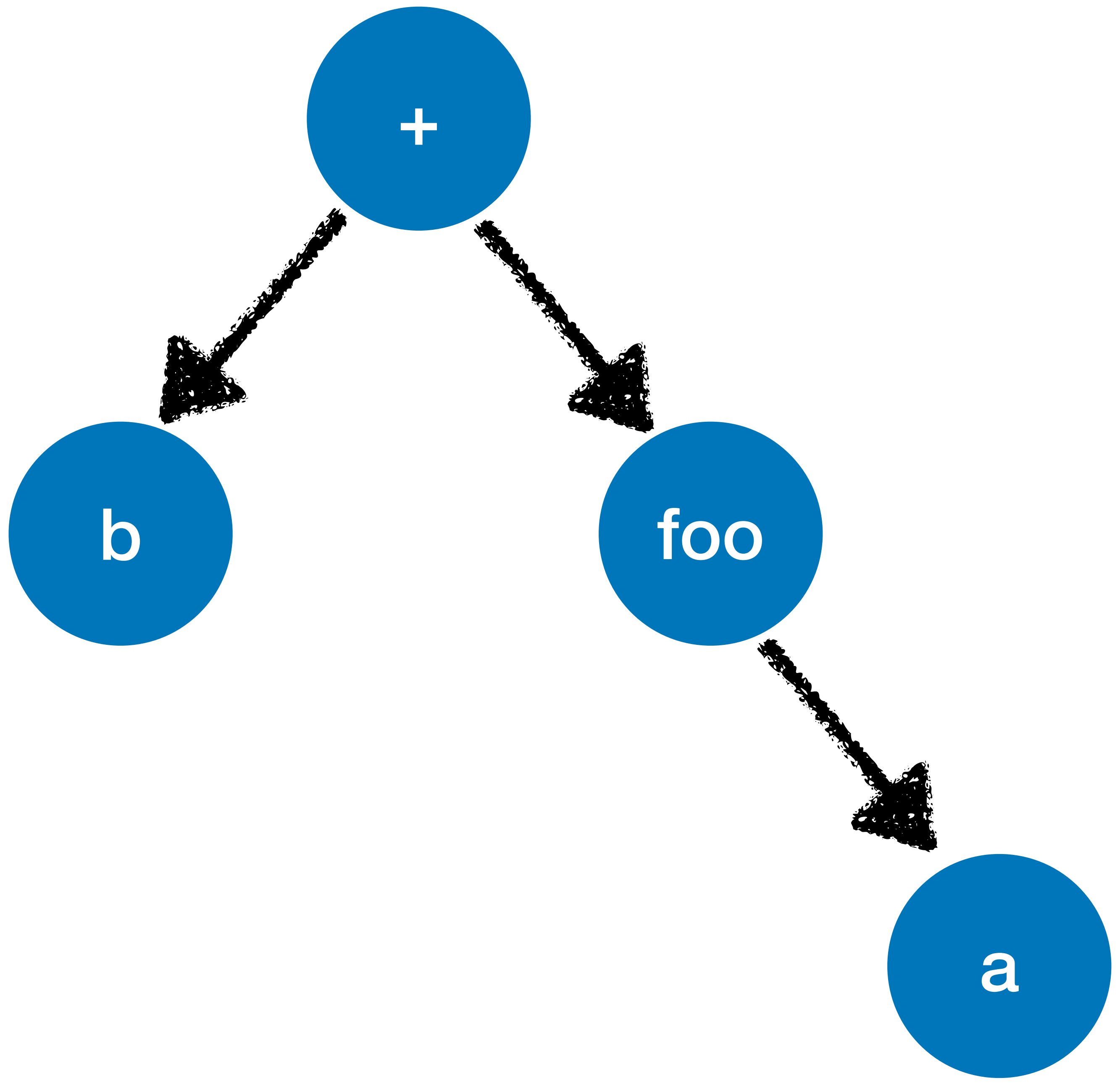
- Interpreters and compilers **are programs**
- They take data as input (the program to execute)
- The manipulate it using some data structures
- They output the result (if an interpreter) or code (if a compiler)

Data structures to represent code

- Lists
- Trees

```
b + foo(a);
```

LOAD R1, b
MOV R2, R1
LOAD R1, a
CALL FOO
ADD R1, R1, b



There are also graphs,
but they are for an advanced course
(or an internship ;))

Data structures to represent code

Lists

- Closer to “machine” code
- Simple to manipulate
- Relations between instructions become implicit
  - e.g., how many arguments does foo have?
  - e.g., Answer => sometimes, we need to see foo’s code
  - These become “conventions"

```
b + foo(a);
```

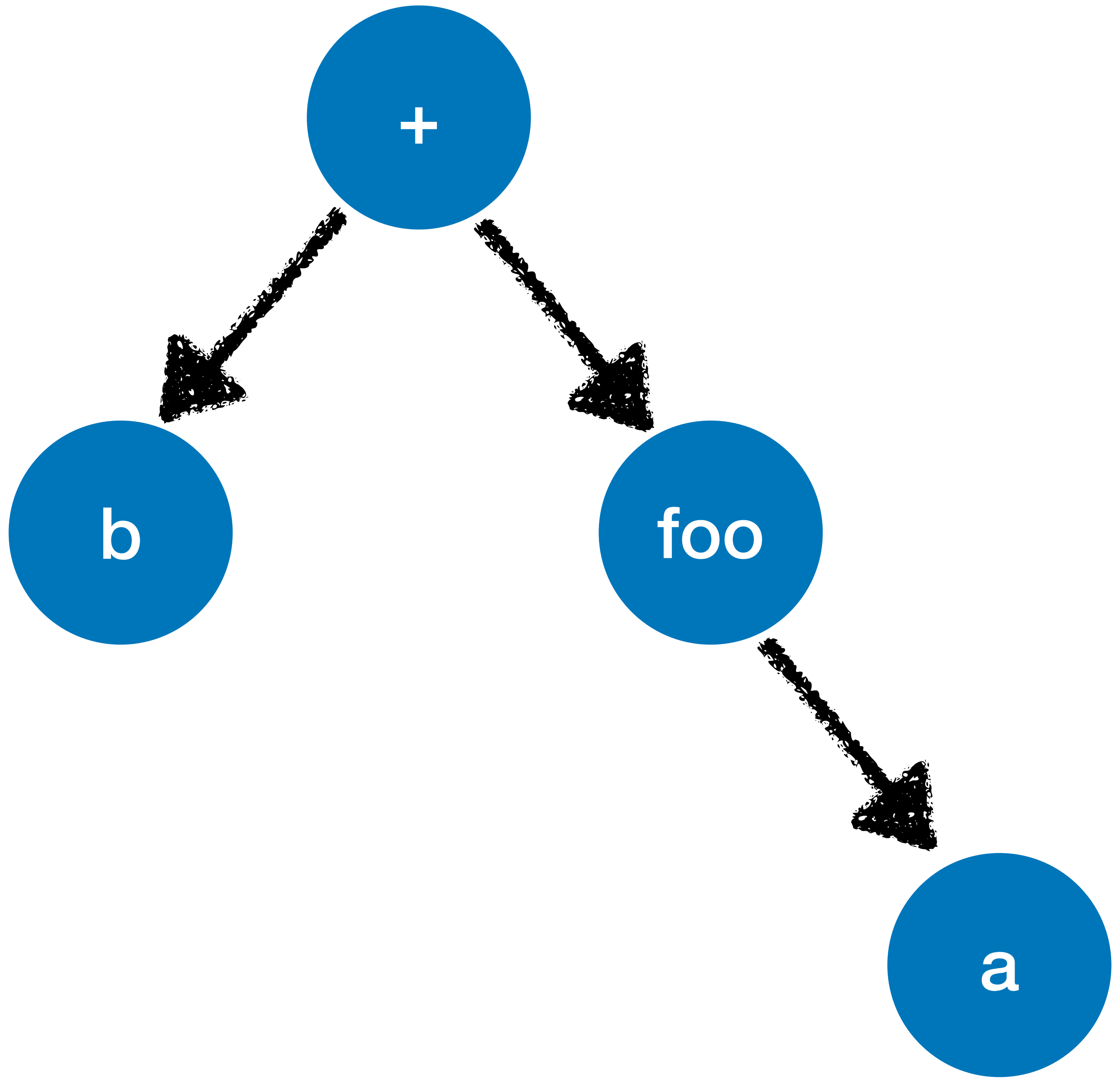
LOAD R1, b
MOV R2, R1
LOAD R1, a
CALL foo
ADD R1, R1, b

Data structures to represent code

Trees

- Closer to source code
- Often produced by a parser
- Relations are explicit
  - e.g., how many arguments does foo have?
  - e.g., Answer => look at foo's children!

```
b + foo(a);
```



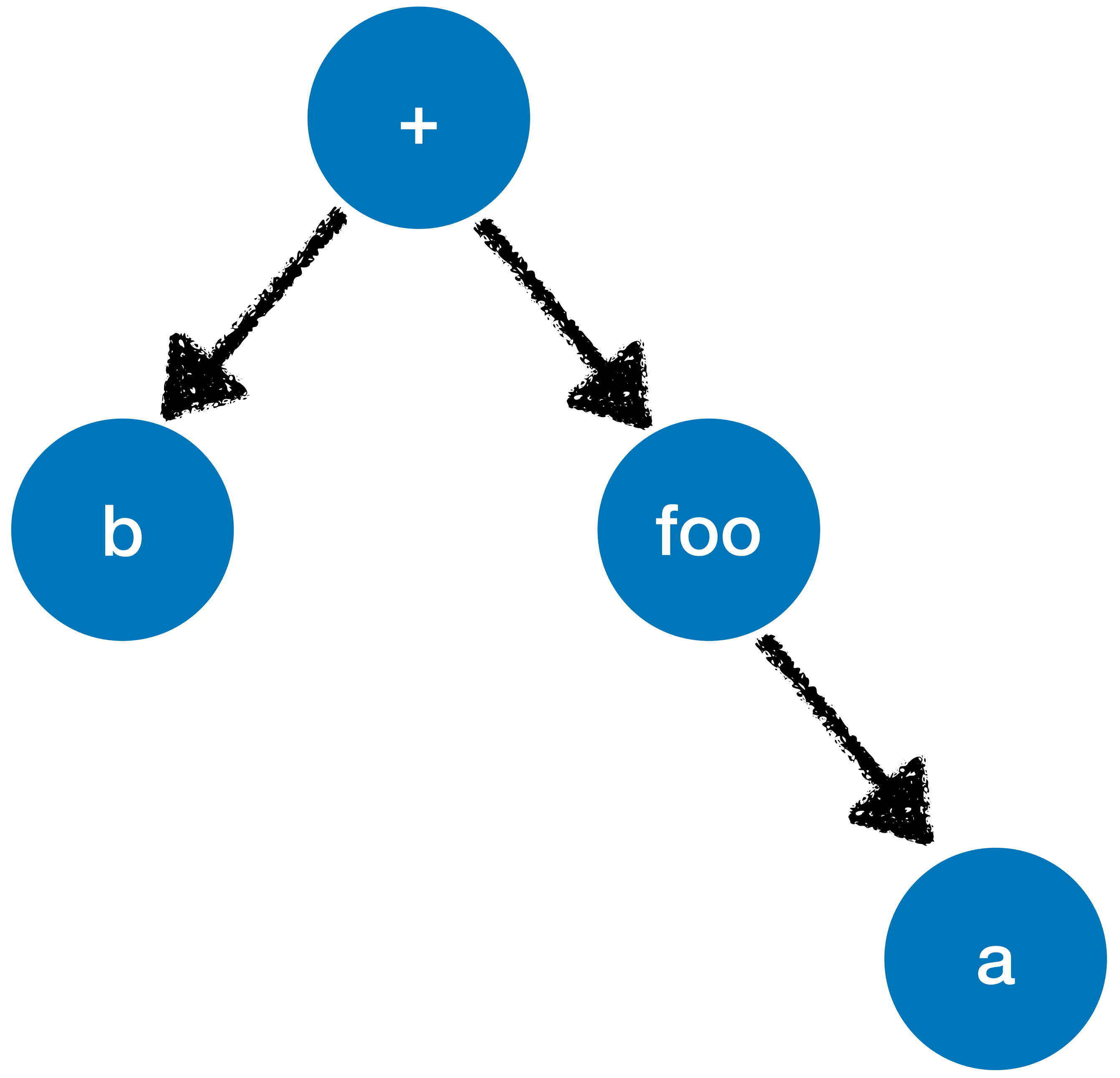
Abstract Syntax Trees (ASTs)

- Trees representing code
- Abstract, because they do not represent ALL elements in the grammar
  - i.e., parentheses, statement finalisers, indentation are **not** in the tree

```
b + foo(a);

b + (foo(a));

b +
  foo(a)
```



Pharo AST's

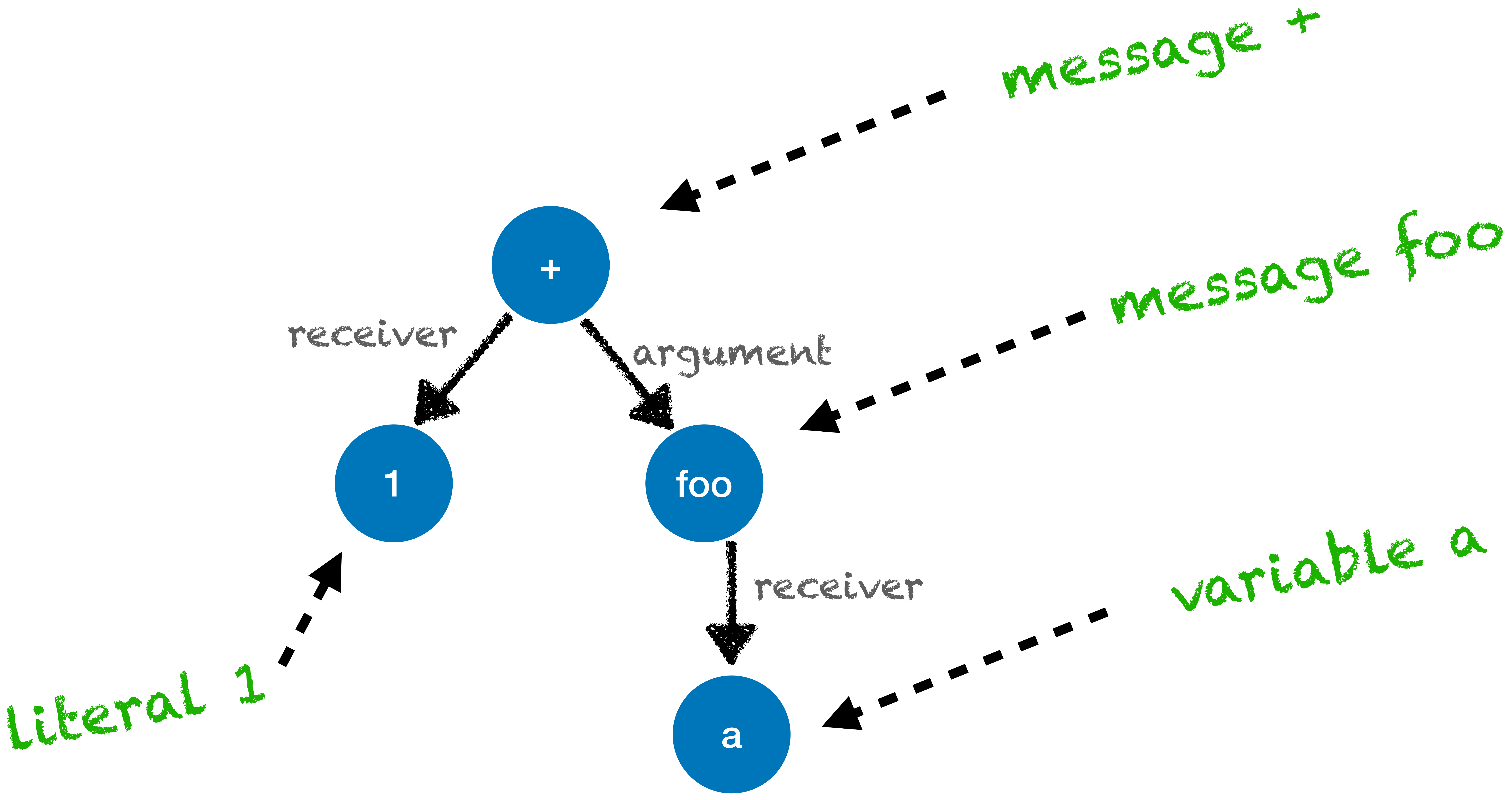
Revisiting the syntax

- The tree is made of nodes
- Roughly one node per syntactic element
  - message-sends
  - variables (locals, globals, instance)
  - literals (integers, booleans…)
  - blocks ([…])
- precedence is in the tree!

Pharo AST's

Example

```
1 + a foo
```

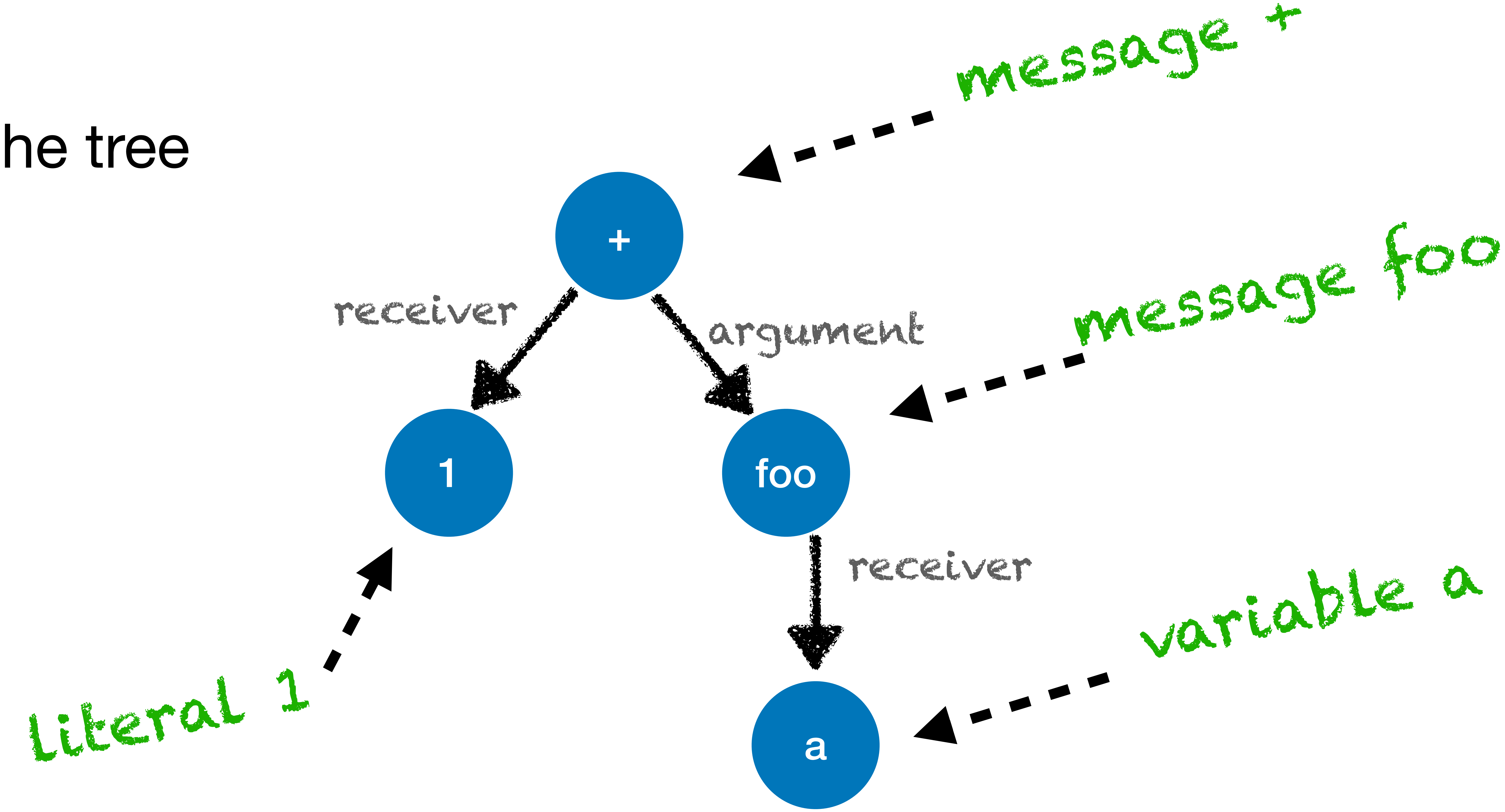


Pharo AST's

Precedence is in the tree!

- Executed first => lower in the tree
- Unary < binary < keyword
- Thus unary is lower

```
1 + a foo
```

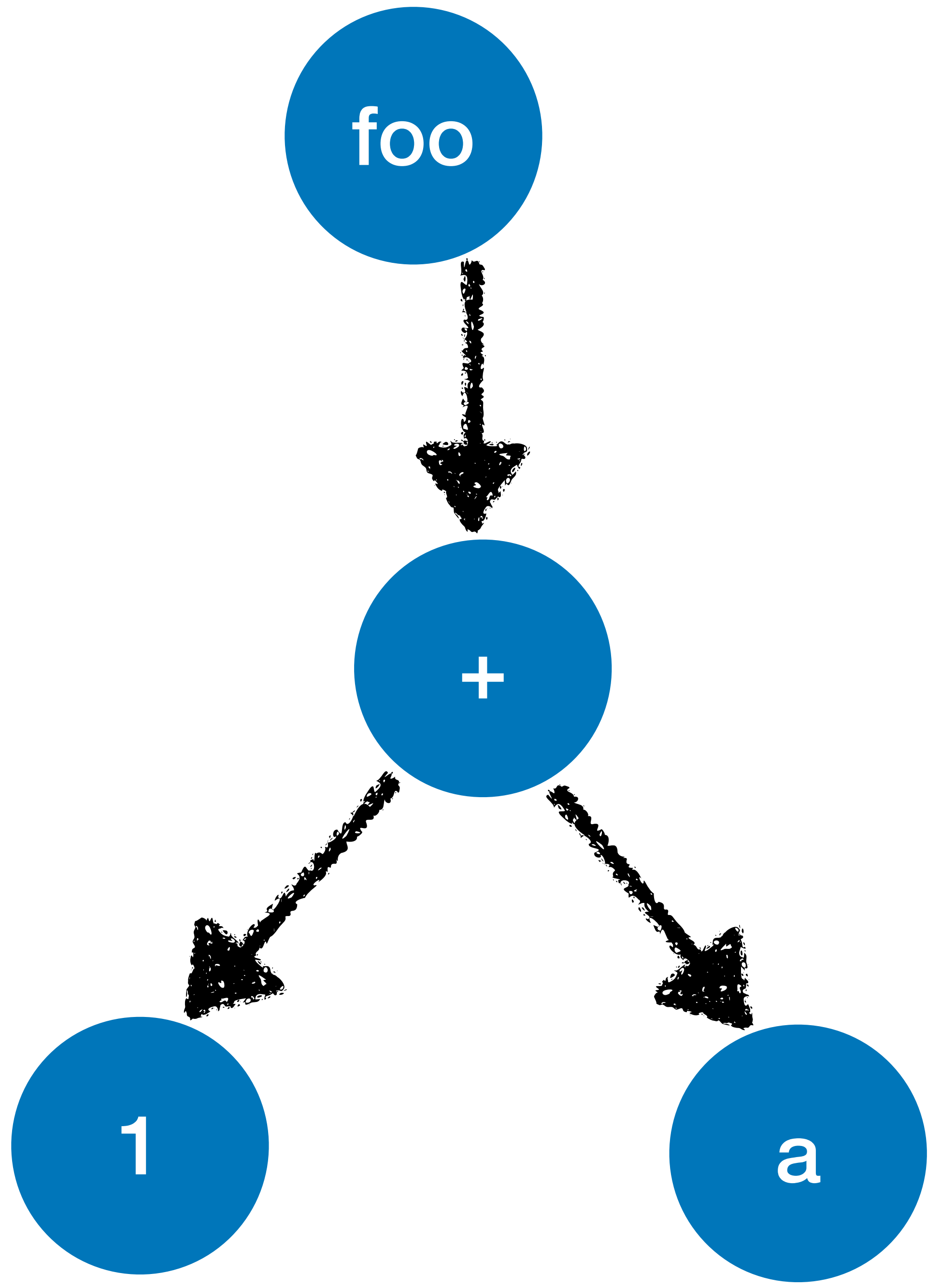


Pharo AST's

Precedence is in the tree, example 2

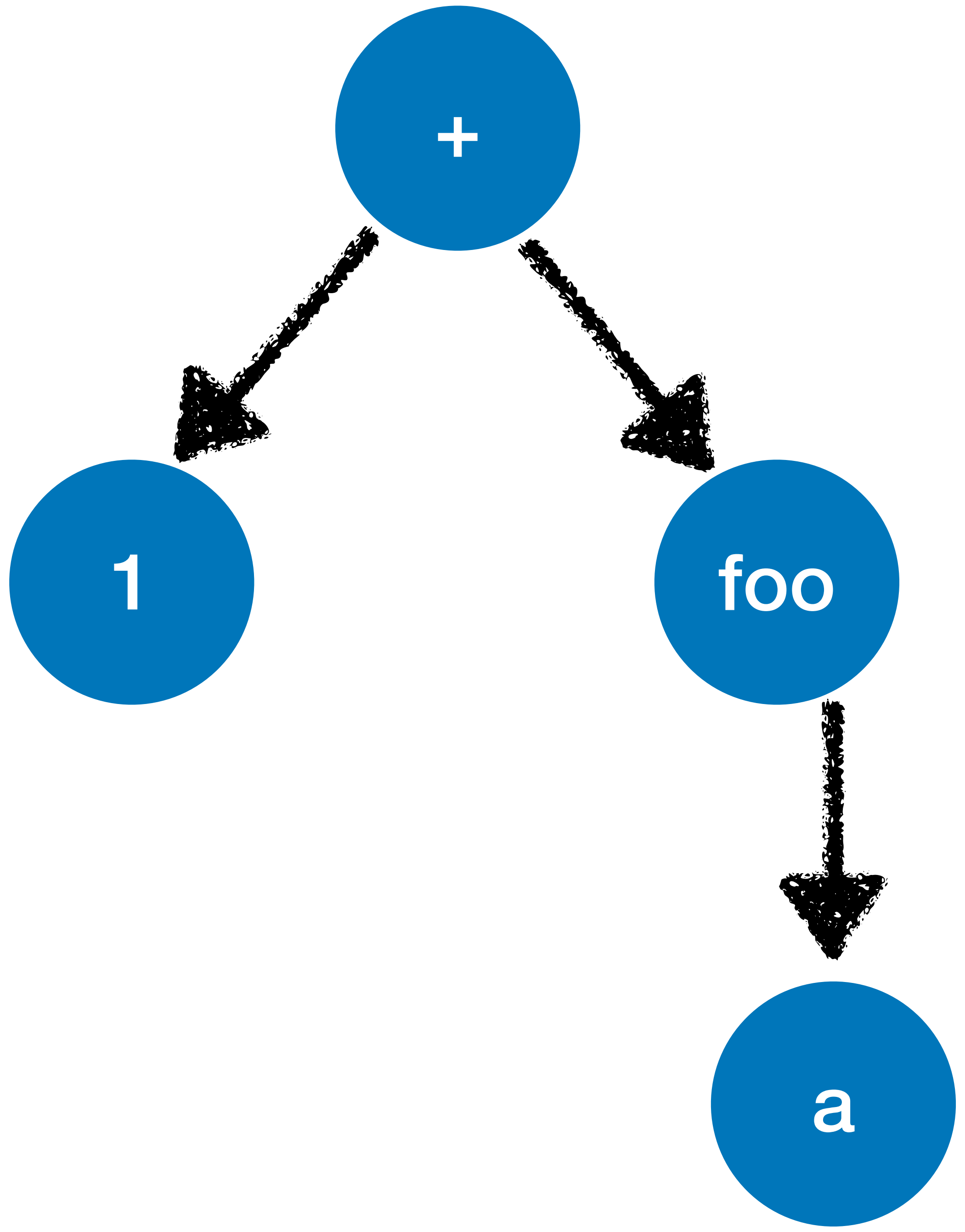
- Executed first => lower in the tree
- Parenthesis < unary < binary < keyword
- Thus parenthesis is lower

```
(1 + a) foo
```

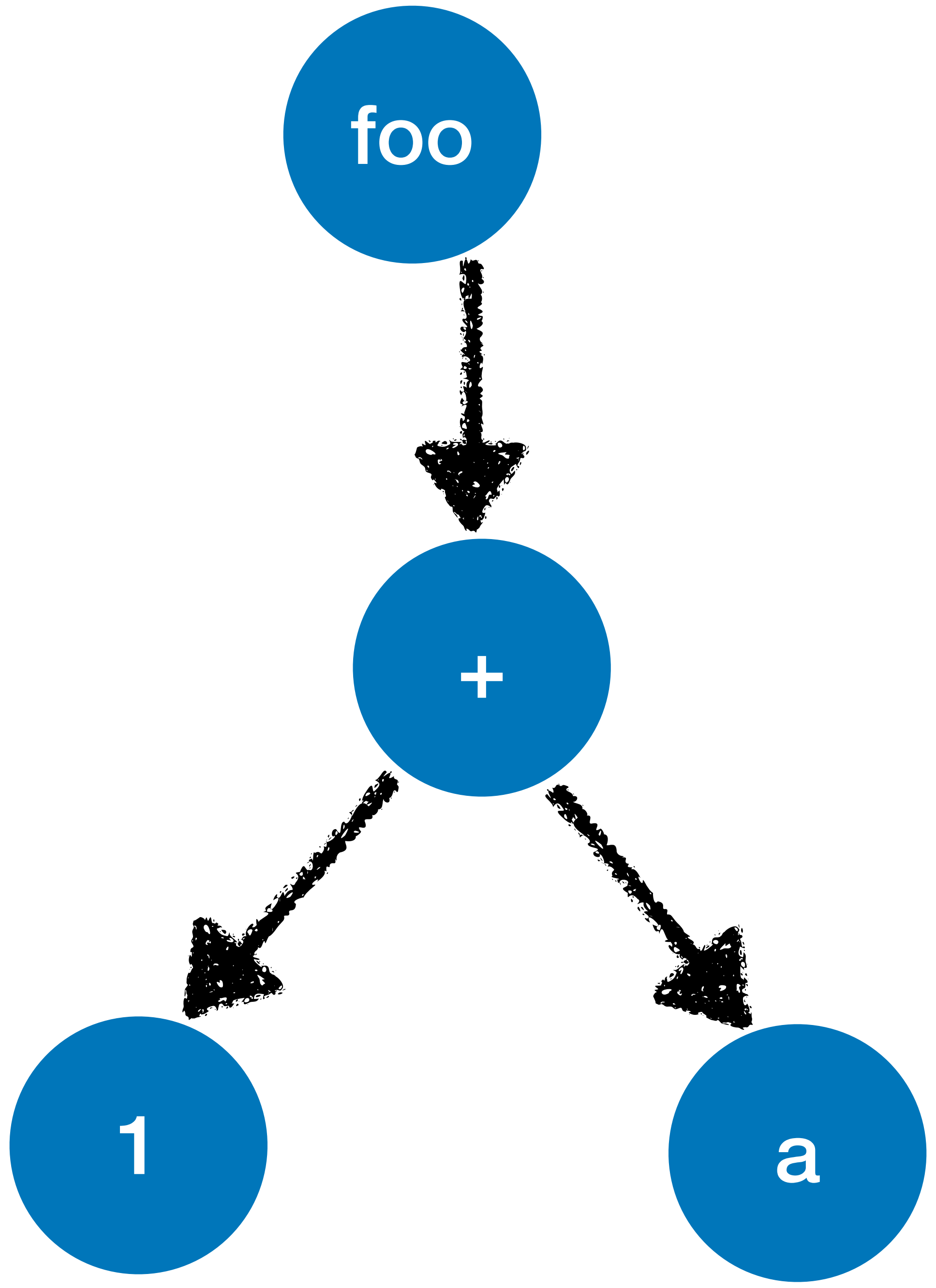


Comparing Precedence

`1 + a foo`



`(1 + a) foo`



Basic AST Manipulation

- `aNode children`
- `aNode allChildren`
- `aNode nodesDo: aBlock`
- `aNode methodNode`
- `aNode isLiteralNode (isVariable, isMessage…)`

Conclusion

- Code can have many representations (with plus and cons)
- ASTs are trees representing code
  - Each node is a syntactic element
  - Relation between nodes show dependencies
  - Precedence is explicit in the tree
    => the lower in the tree, the higher the precedence