Tag-Based Modules in Genetic Programming

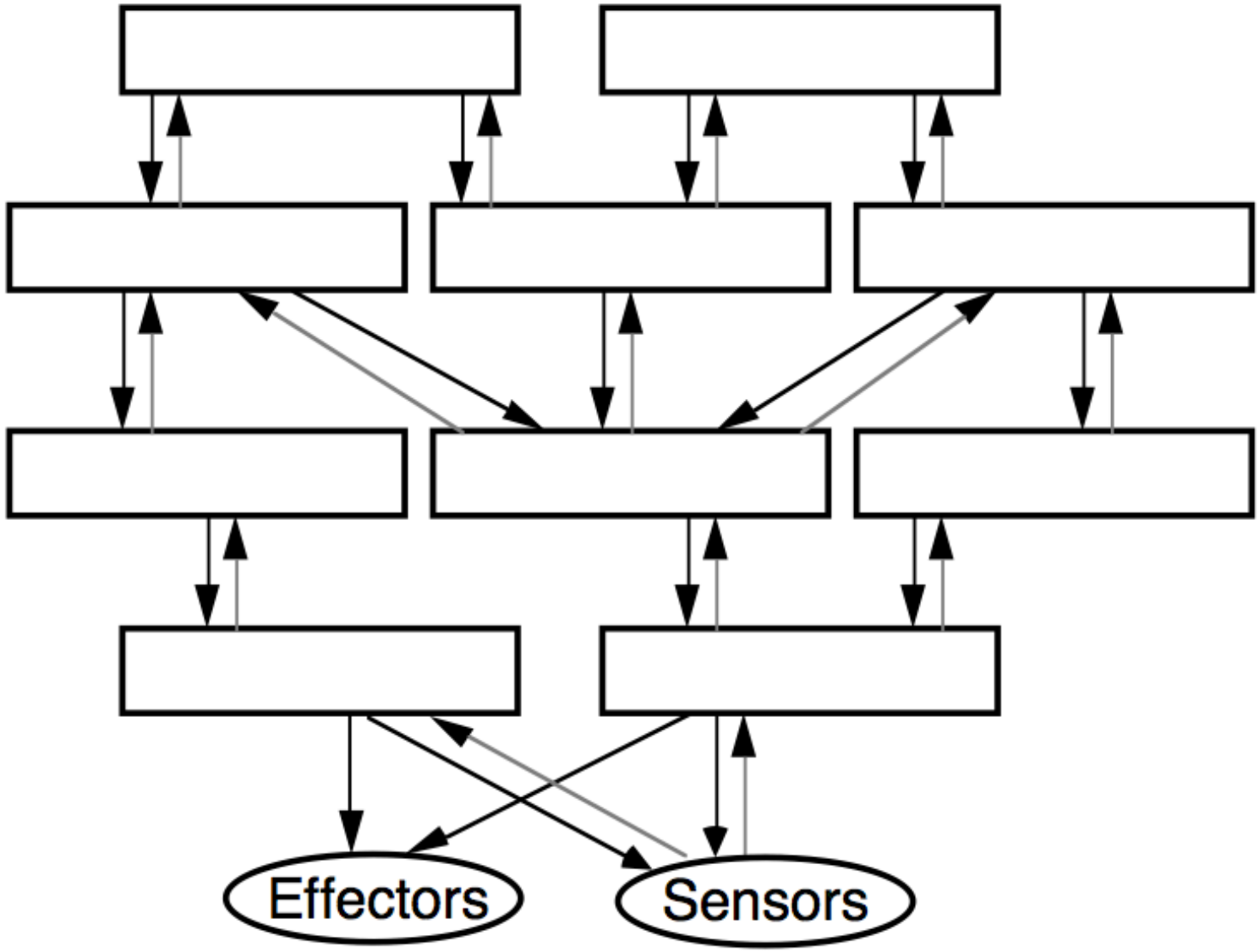
Lee Spector, Brian Martin, Kyle Harrington & Thomas Helmuth

Cognitive Science, Hampshire College
Computer Science, Brandeis University
Computer Science, University of Massachusetts, Amherst

Outline

- Modularity in GP
- Tags and Tag-based modularity in GP
- Push and PushGP
- Modules, names, and tags in PushGP
- Tag-based modularity in traditional GP
- Results/Conclusions

Modularity is Everywhere



Modules in GP

- Automatically-defined functions (Koza)
- Automatically-defined macros (Spector)
- Architecture-altering operations (Koza)
- Module acquisition/encapsulation systems (Kinnear, Roberts, many others)
- In Push: code-manipulation instructions that build/execute modules as programs run

We will return to this later!

ADFs

- All programs in the population have the same, pre-specified architecture
- Genetic operators respect that architecture
-
```
(progn (defn adf0 (arg0 arg1) ...)
       (defn adf1 (arg0 arg1 arg2) ...)
  (.... (adf1 ...) (adf0 ...) ...))
```
- Complicated, brittle, limited...
- Architecture-altering operations: more so

Tags

- Roots in John Holland's work on principles of complex adaptive systems
- Applied in models of the evolution of altruism, with agents having tags and tag-difference thresholds for donation
- A tag is *an initially meaningless identifier that can come to have meaning through the matches in which it participates*
- Matches may be inexact

Tag-based Modules in GP

- Add mechanisms for tagging code
- Add mechanisms for retrieving/branching to code with closest matching tag
- As long as any code has been tagged, all branches go somewhere
- Number of tagged modules can grow incrementally over evolutionary time

Push

- Stack-based postfix language with one stack per type
- Types include: integer, float, Boolean, name, code, exec, vector, matrix, quantum gate, [add more as needed]
- Missing argument? NOOP
- Trivial syntax:
  program → instruction | literal | (program*)

Why Push?

- Multiple data types
- User-defined procedures & functions
- User-defined macros & control structures
- User-defined representations
- Dynamic definition & redefinition
- All of the above provided without any mechanisms beyond the stack-based execution architecture

And I won't even mention

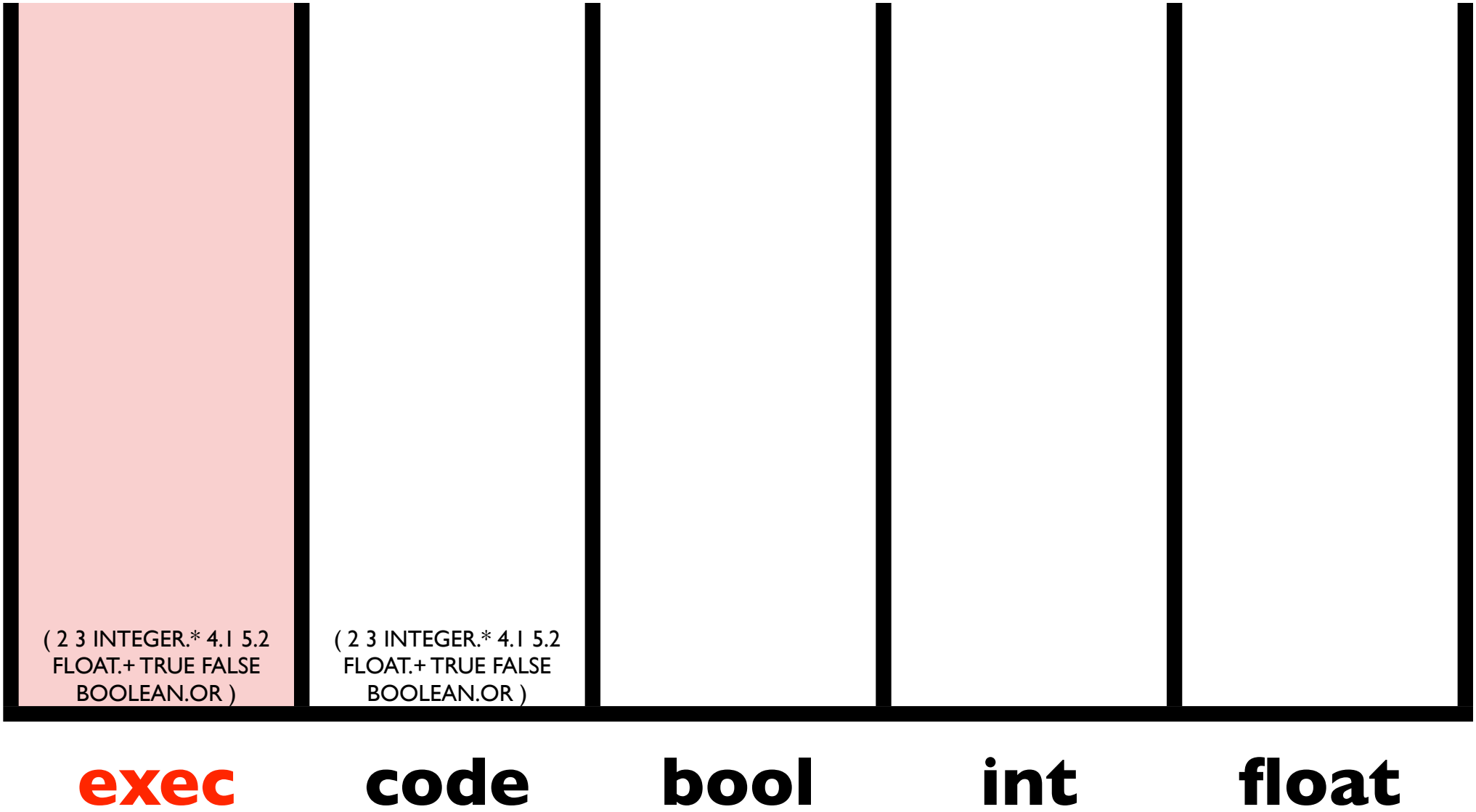
- Automatic simplification
- Autoconstructive evolution
- Iterators and combinators
- Code self reference
- Ontogenetic programming
- etc. See http://hampshire.edu/lspector/push.html

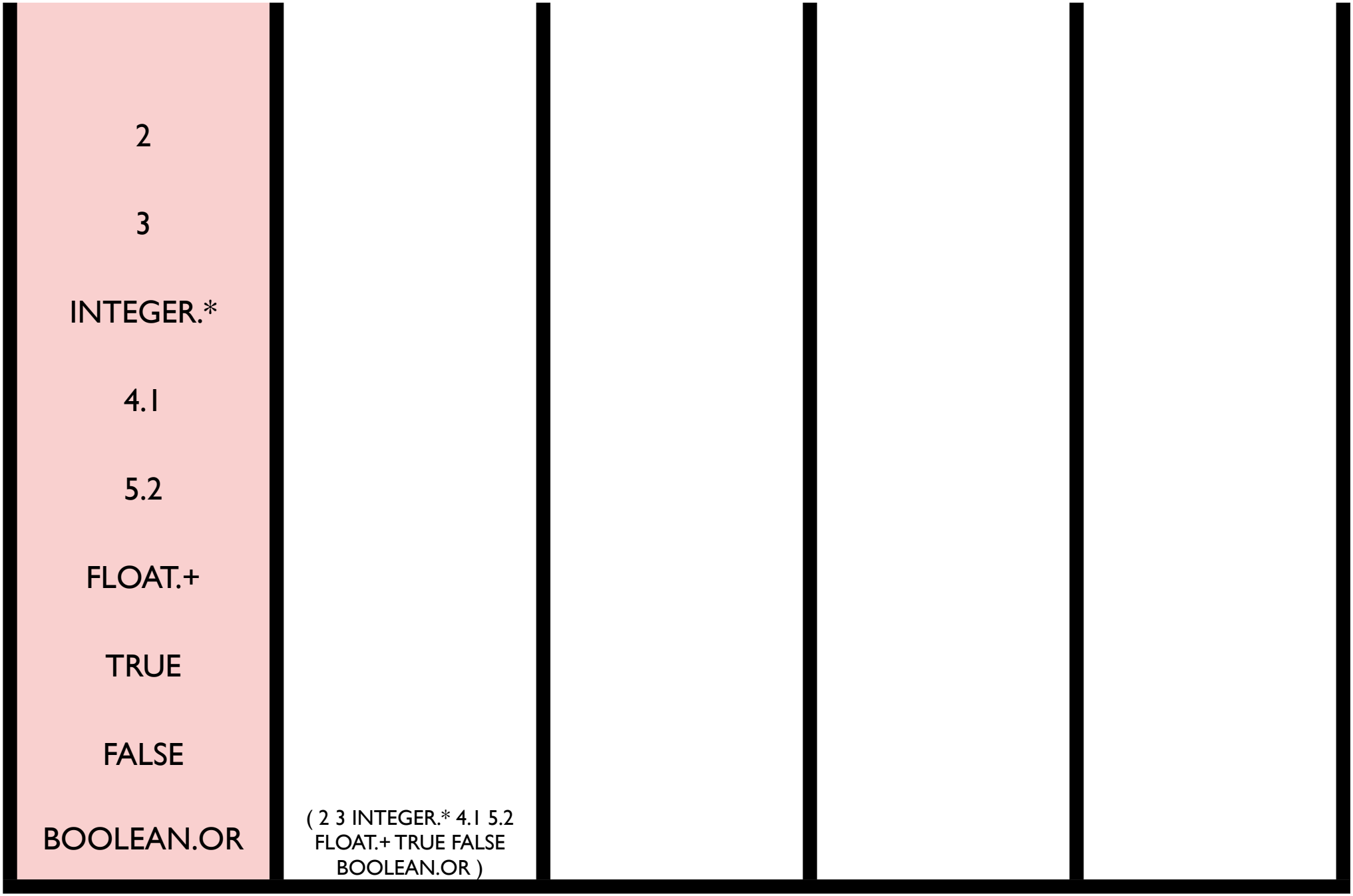
Push(3) Semantics

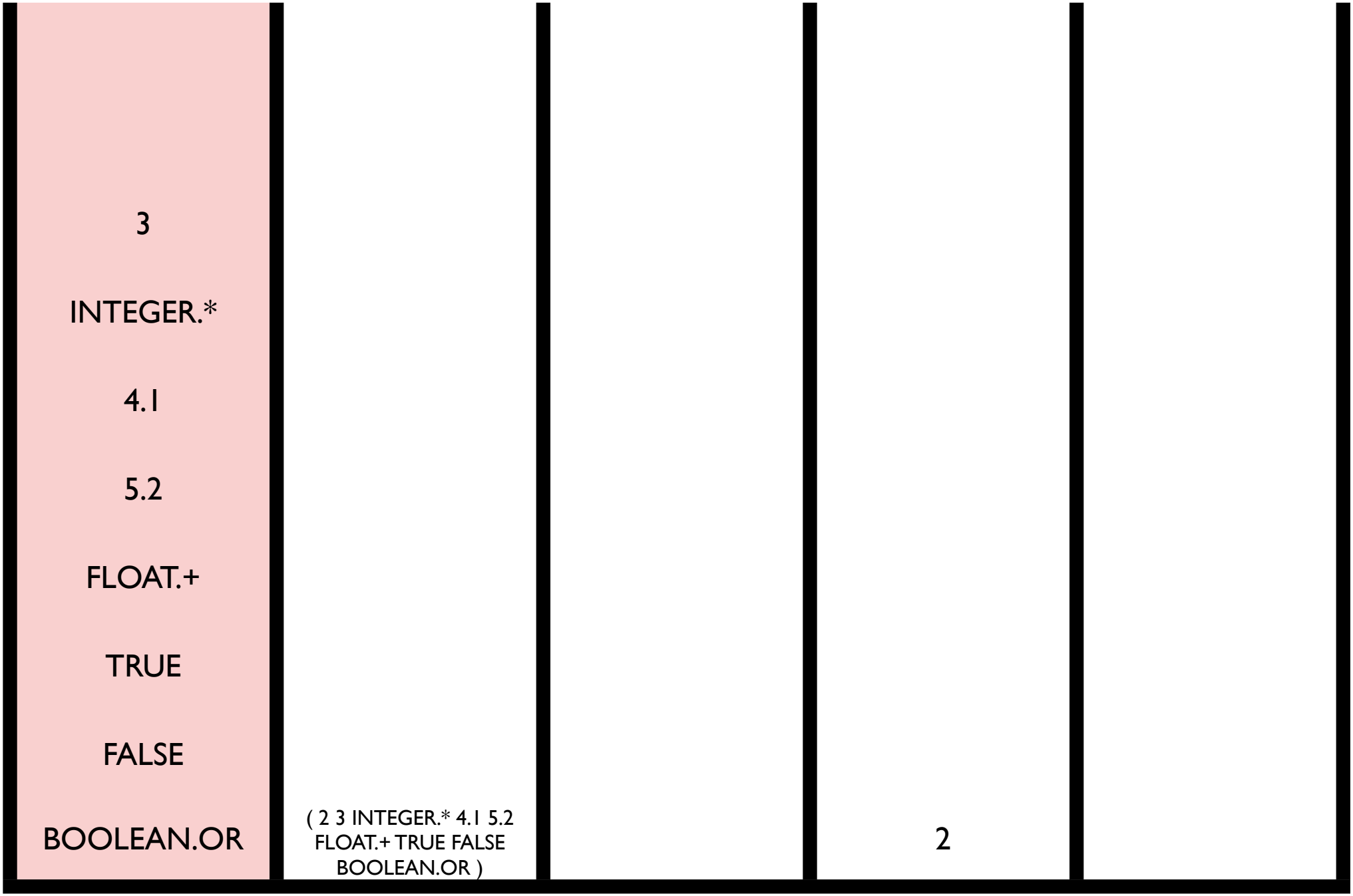
- To execute program P:

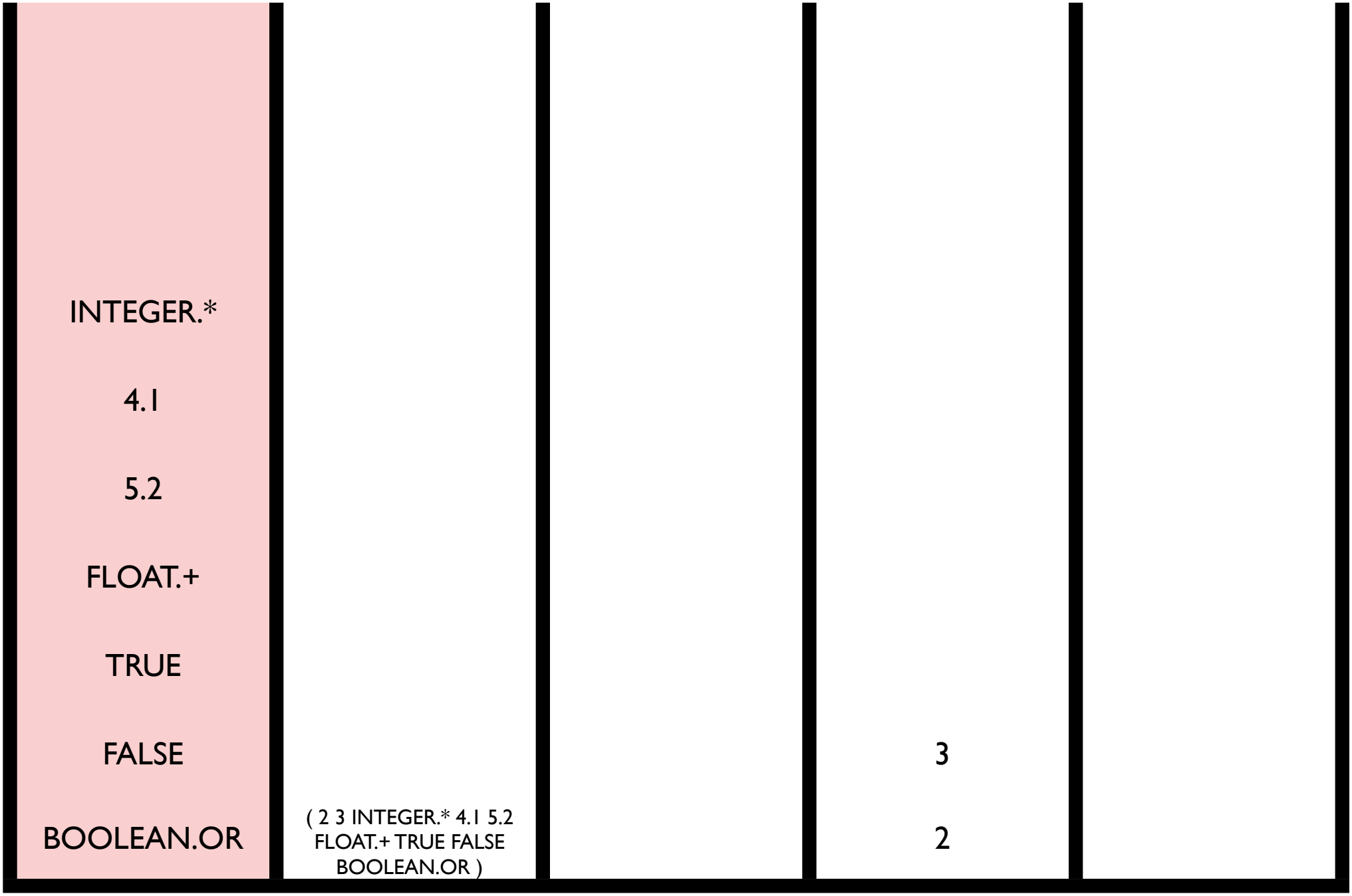
  1. Push P onto the `EXEC` stack.
  2. While the `EXEC` stack is not empty, pop and process the top element of the `EXEC` stack, E:
     (a) If E is an instruction: execute E (accessing whatever stacks are required).
     (b) If E is a literal: push E onto the appropriate stack.
     (c) If E is a list: push each element of E onto the `EXEC` stack, in reverse order.

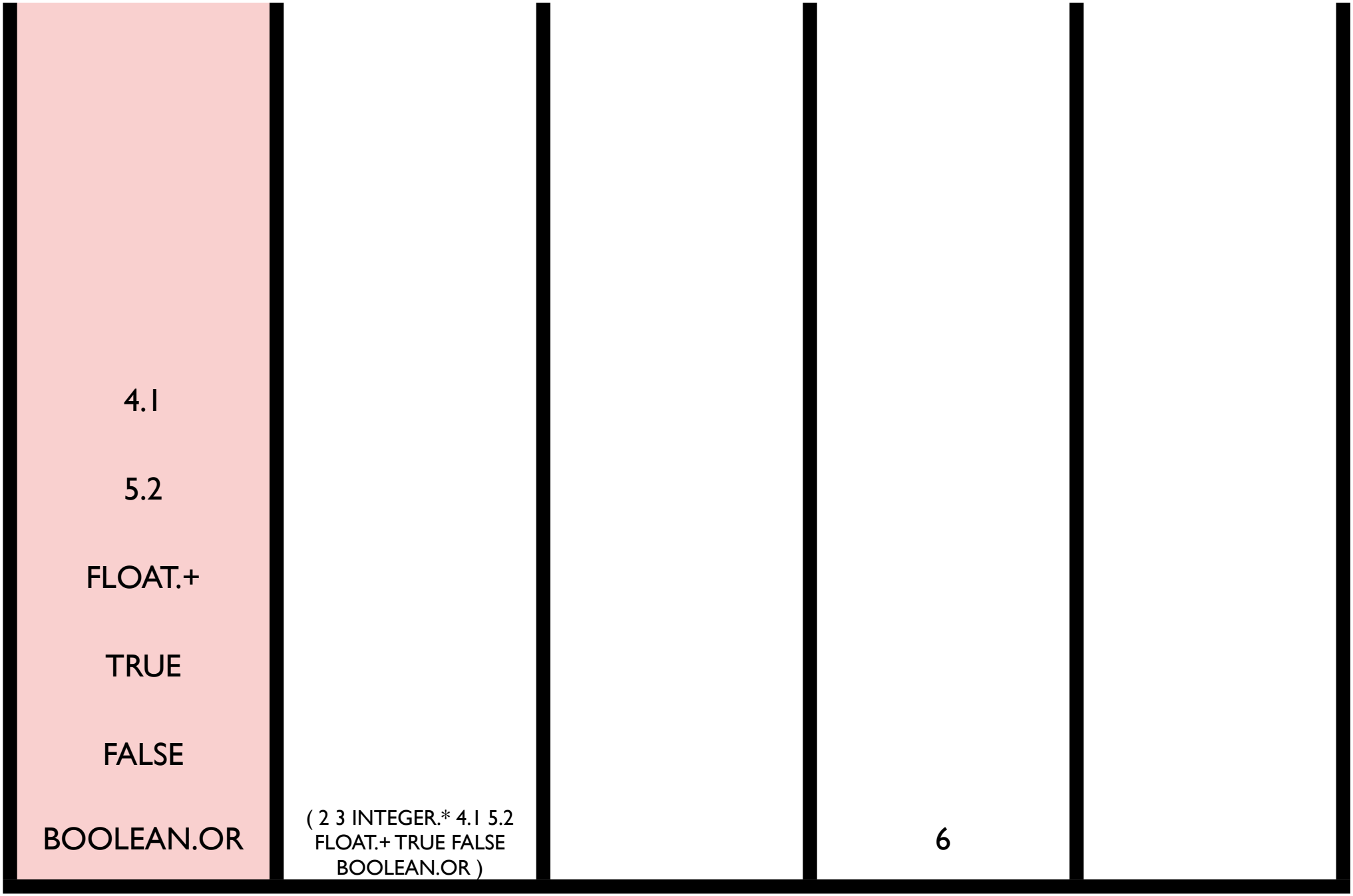
```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+
TRUE FALSE BOOLEAN.OR )
```

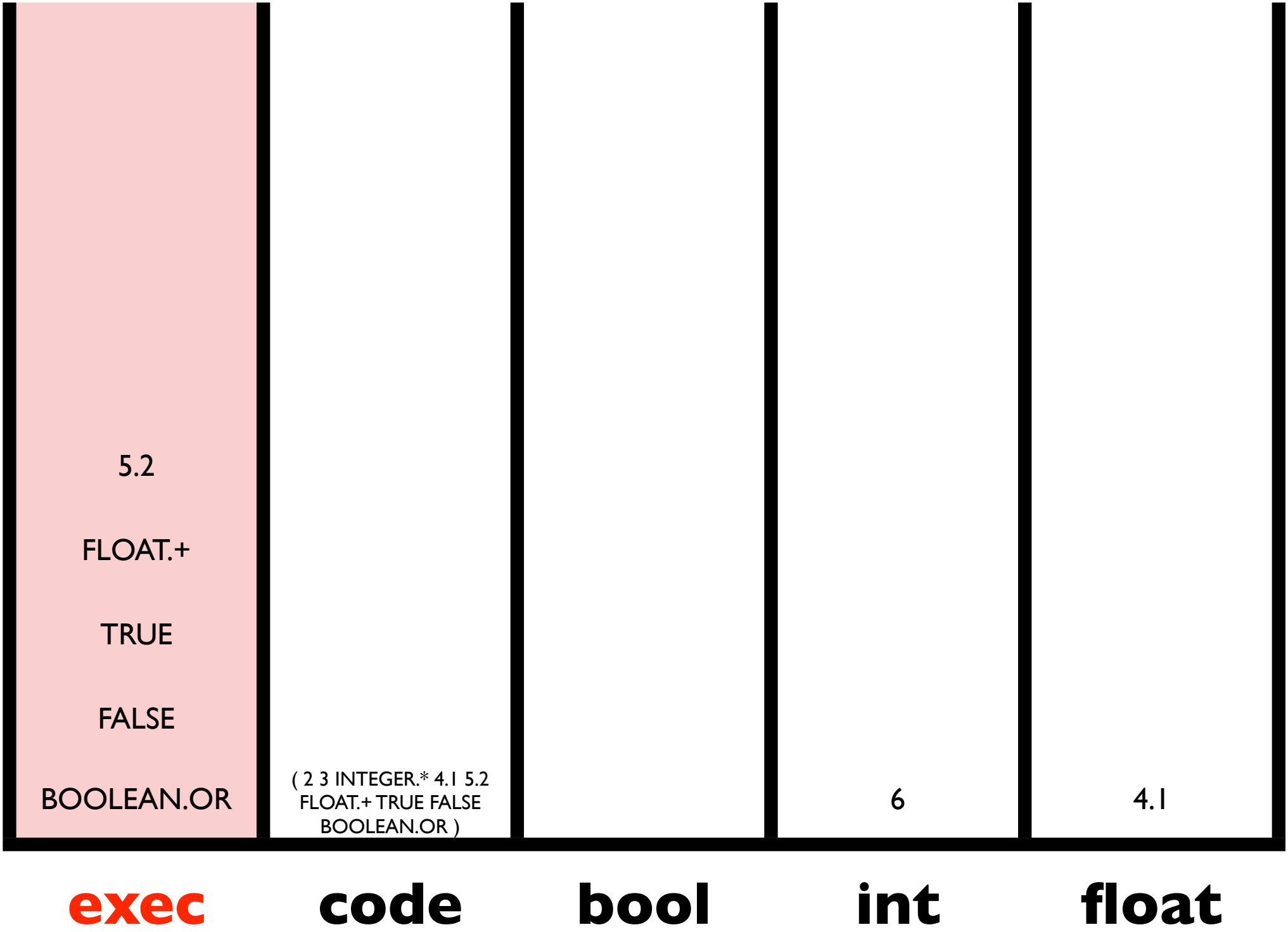


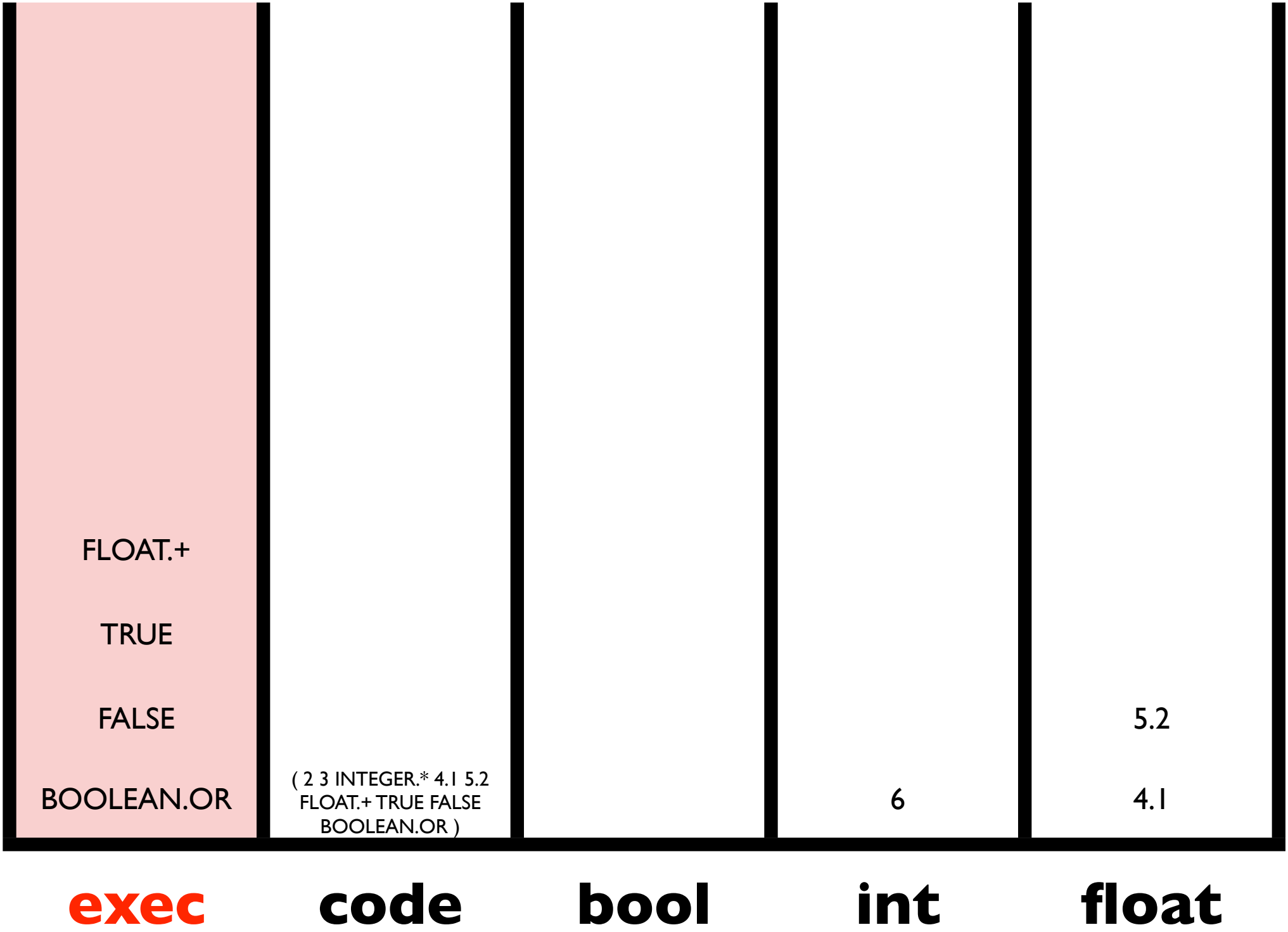

2
3
INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
exec
code
bool
int
float

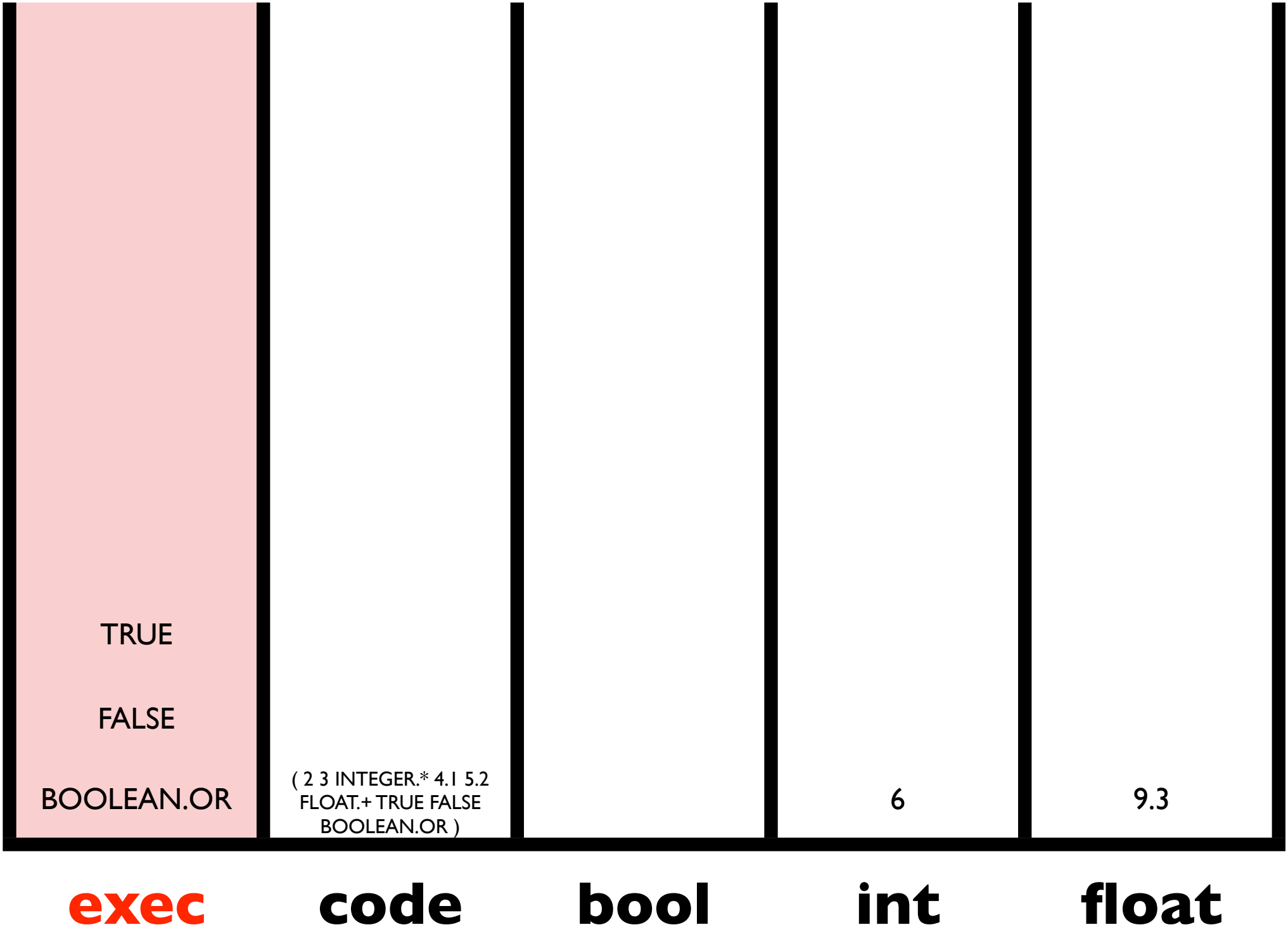

3
INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
2
exec
code
bool
int
float

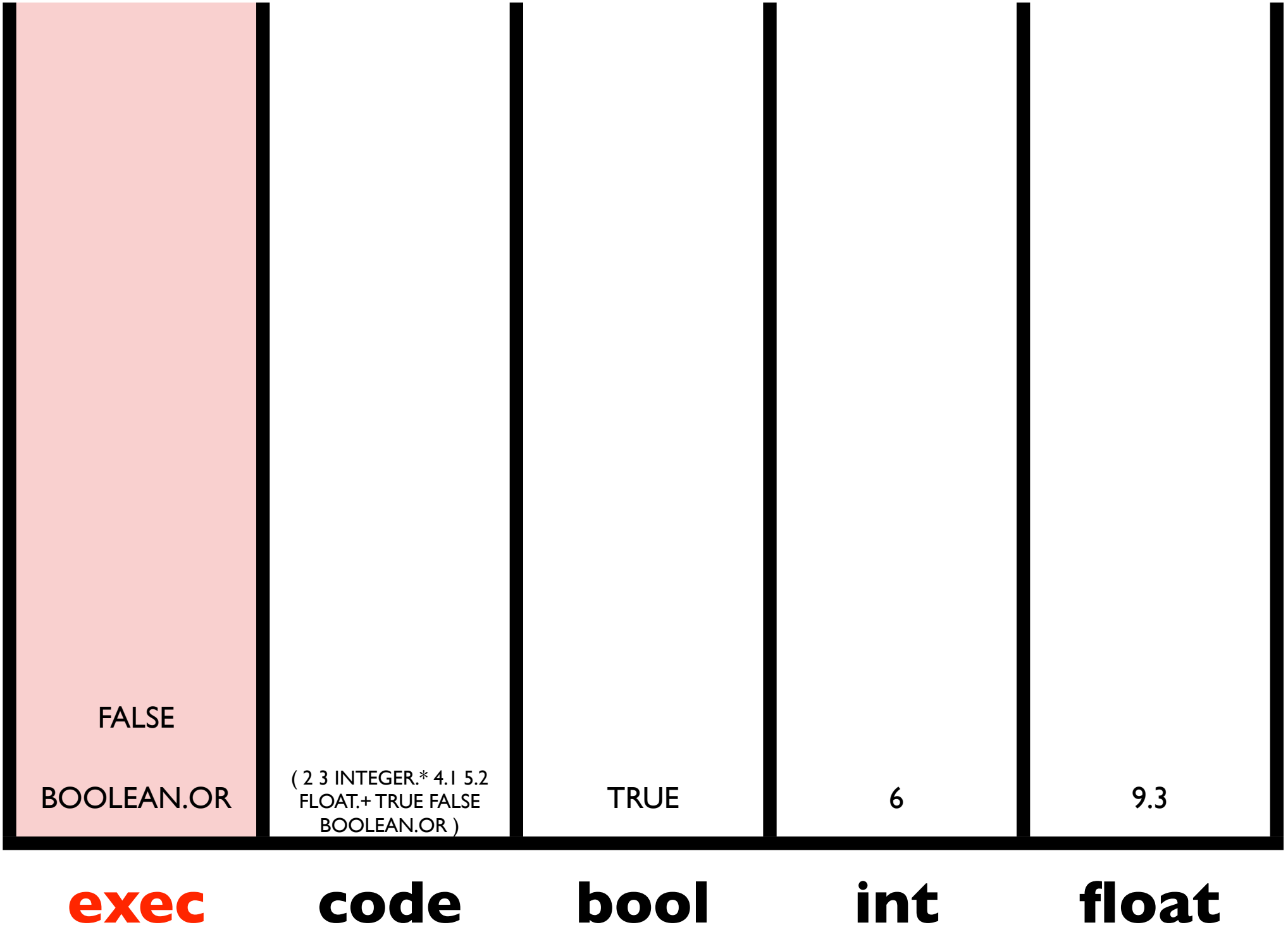

INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
3
2
exec
code
bool
int
float

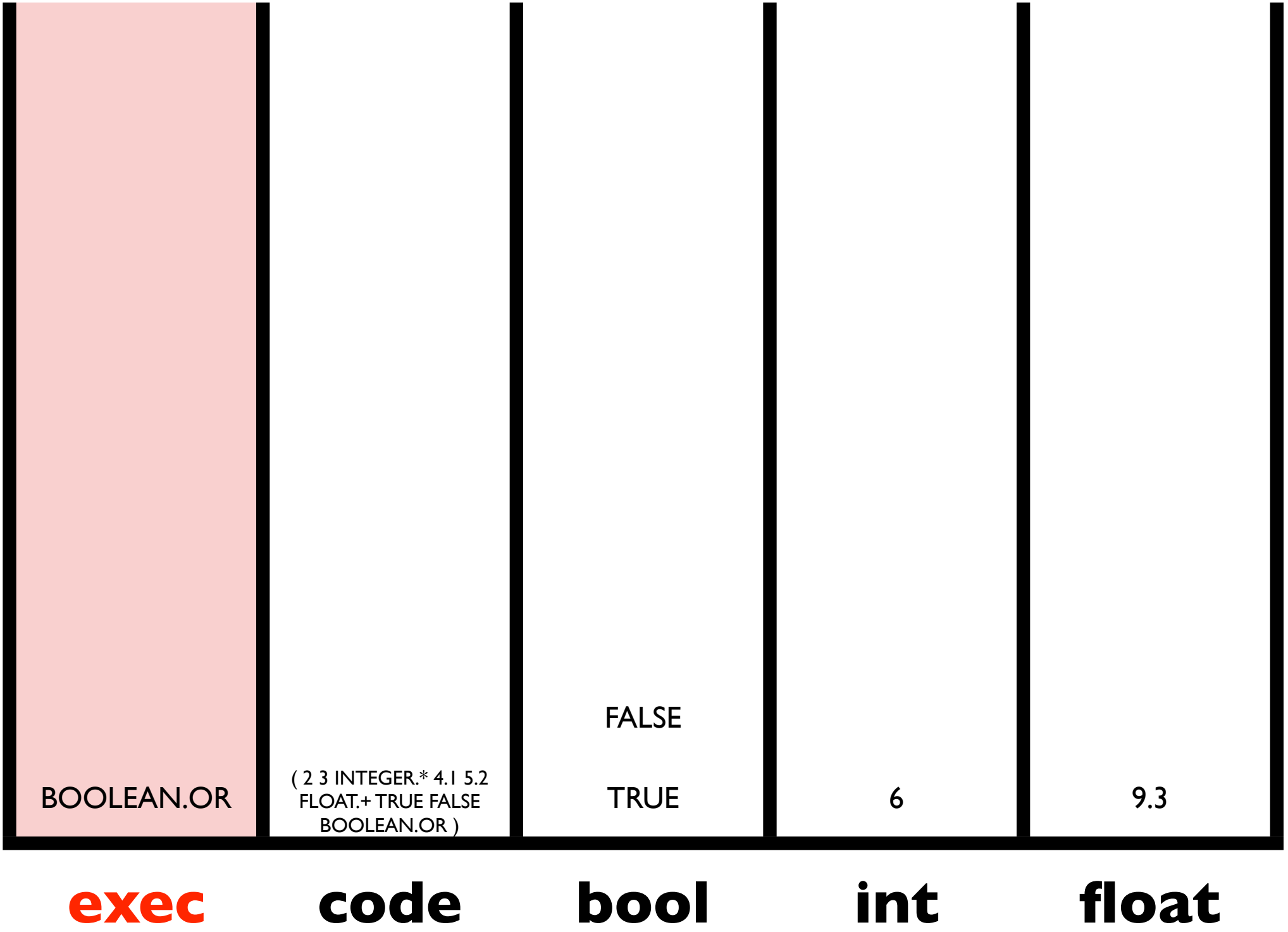

4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
6
exec
code
bool
int
float

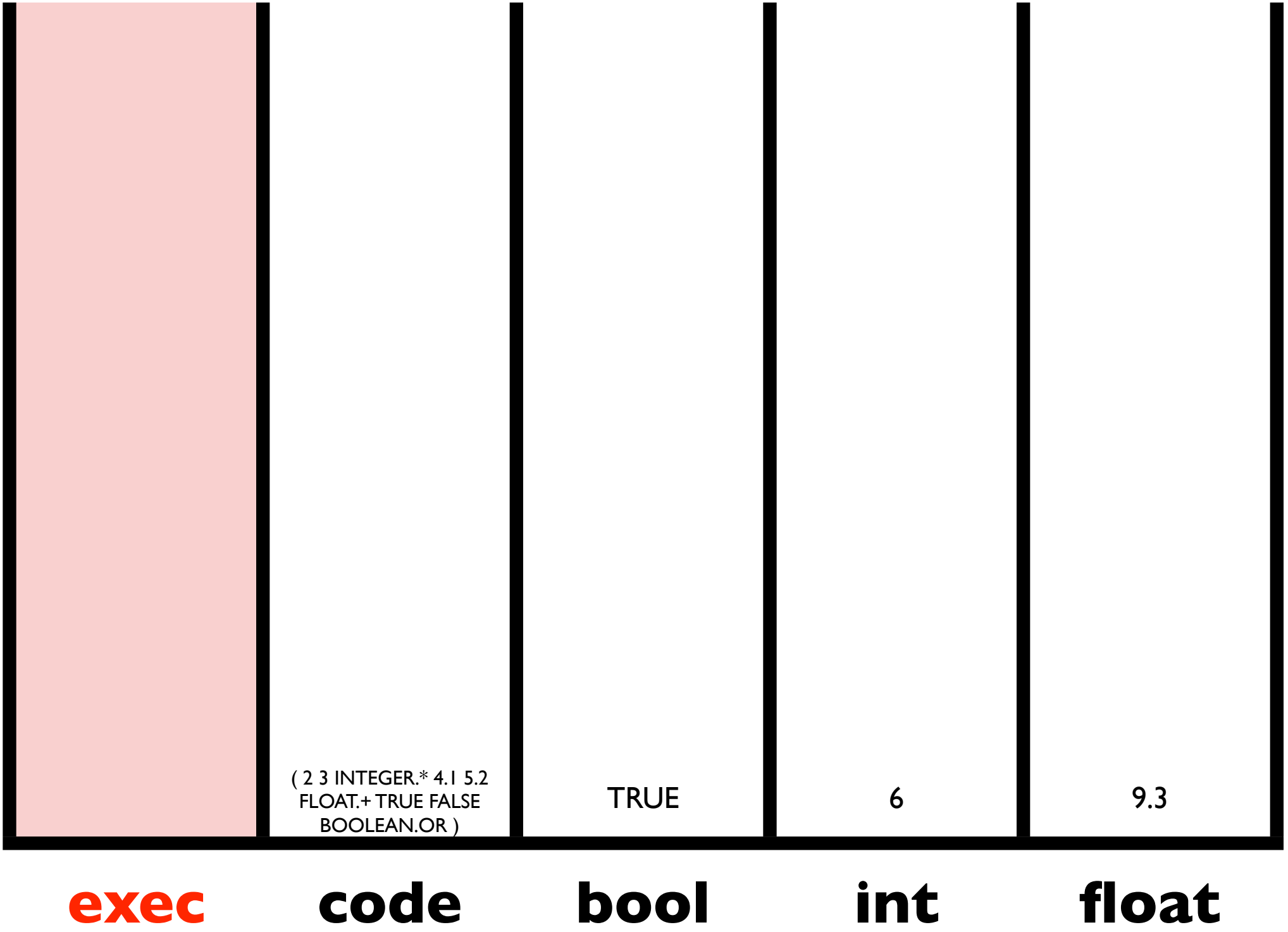

5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
6
4.1
exec
code
bool
int
float


FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
6
5.2
4.1
exec
code
bool
int
float


TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
6
9.3
exec
code
bool
int
float


FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
TRUE
6
9.3
exec
code
bool
int
float


BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
FALSE
TRUE
6
9.3
exec
code
bool
int
float


(2 3 INTEGER.* 4.1 5.2
FLOAT.+ TRUE FALSE
BOOLEAN.OR)
TRUE
6
9.3
exec
code
bool
int
float

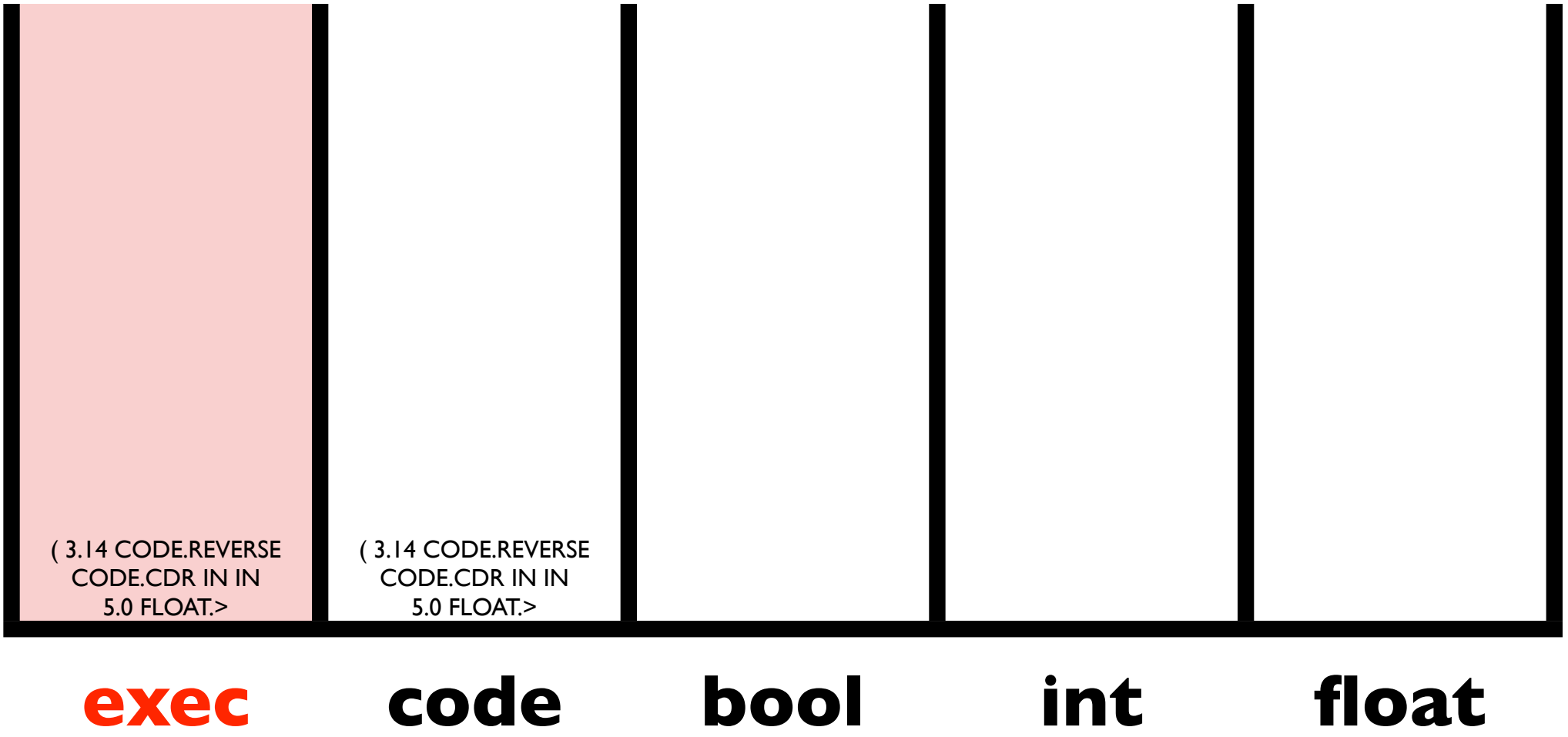
Same Results

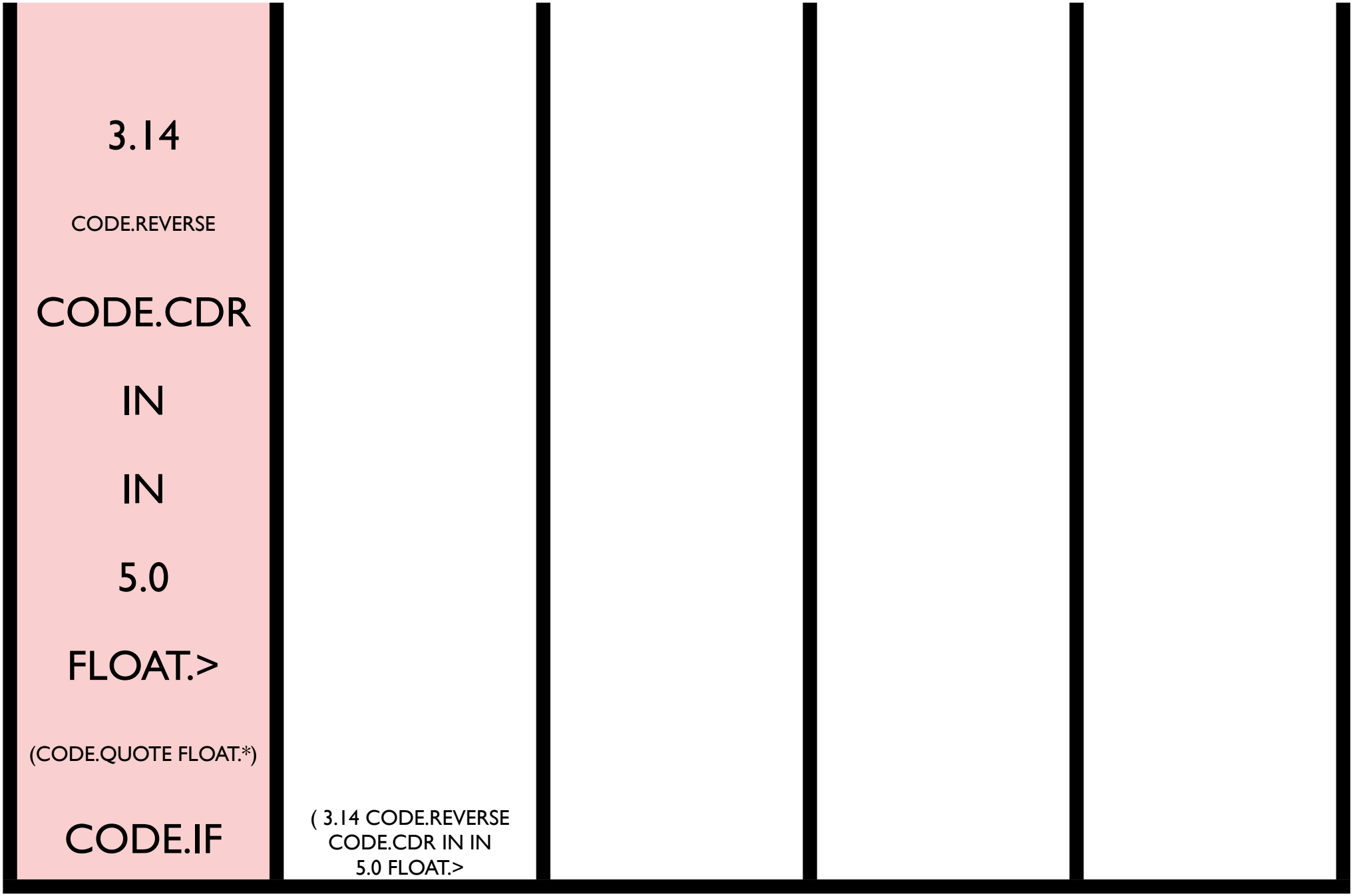
```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+
    TRUE FALSE BOOLEAN.OR )
```

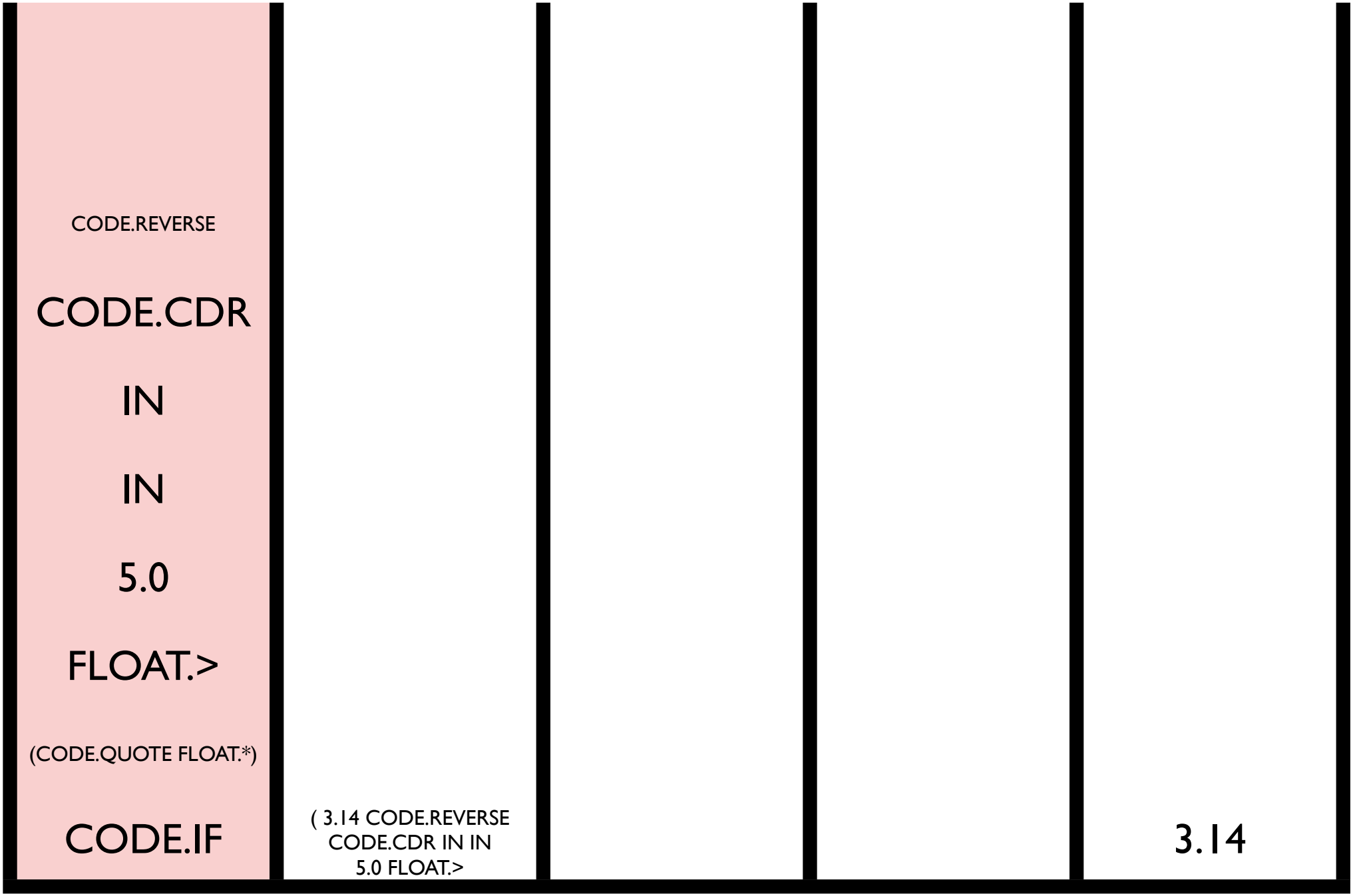
```
( 2 BOOLEAN.AND 4.1 TRUE INTEGER./ FALSE
  3 5.2 BOOLEAN.OR INTEGER.* FLOAT.+ )
```

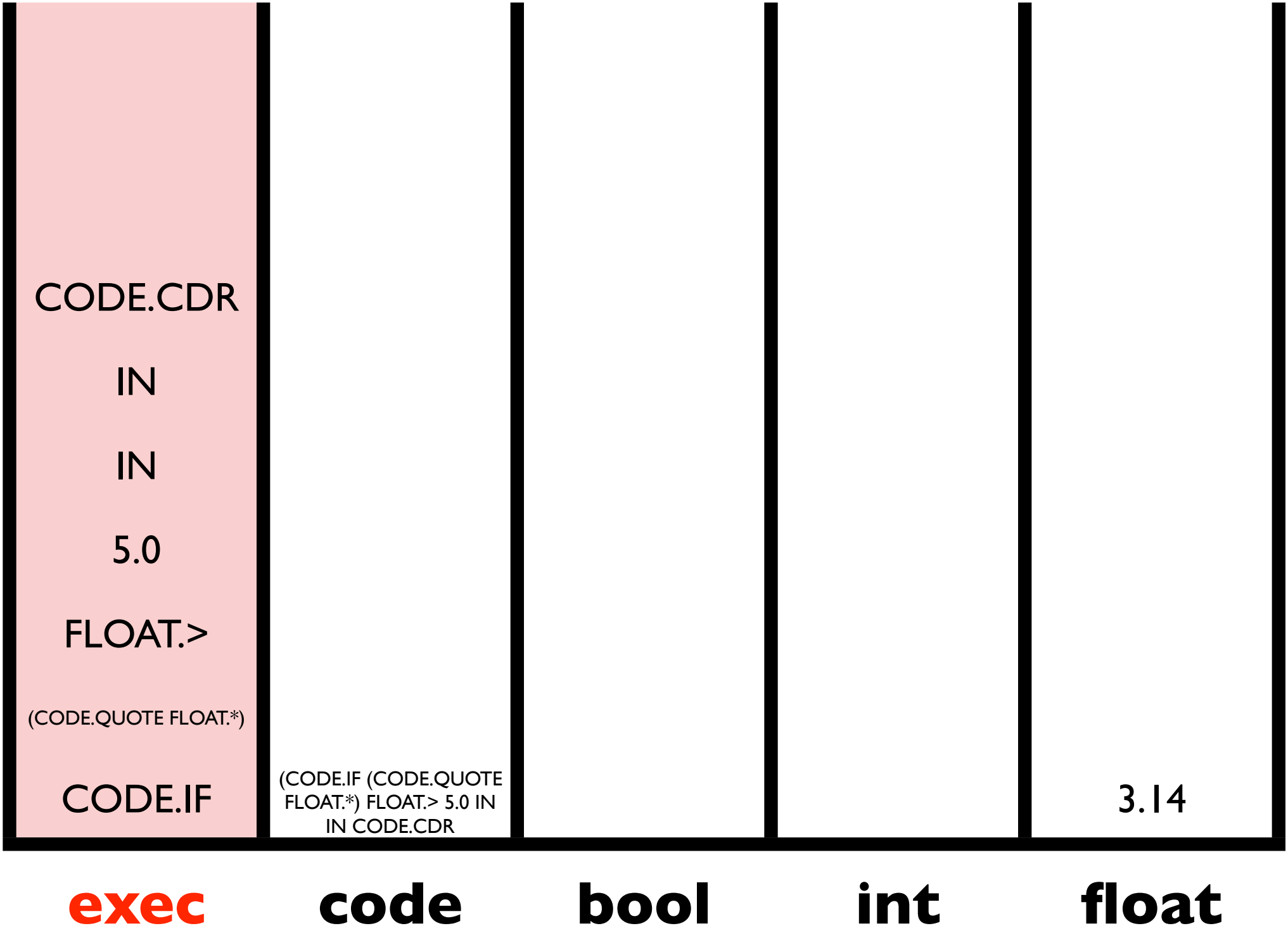
```
( 3.14 CODE.REVERSE CODE.CDR IN IN 5.0
FLOAT.> (CODE.QUOTE FLOAT.*) CODE.IF )
```

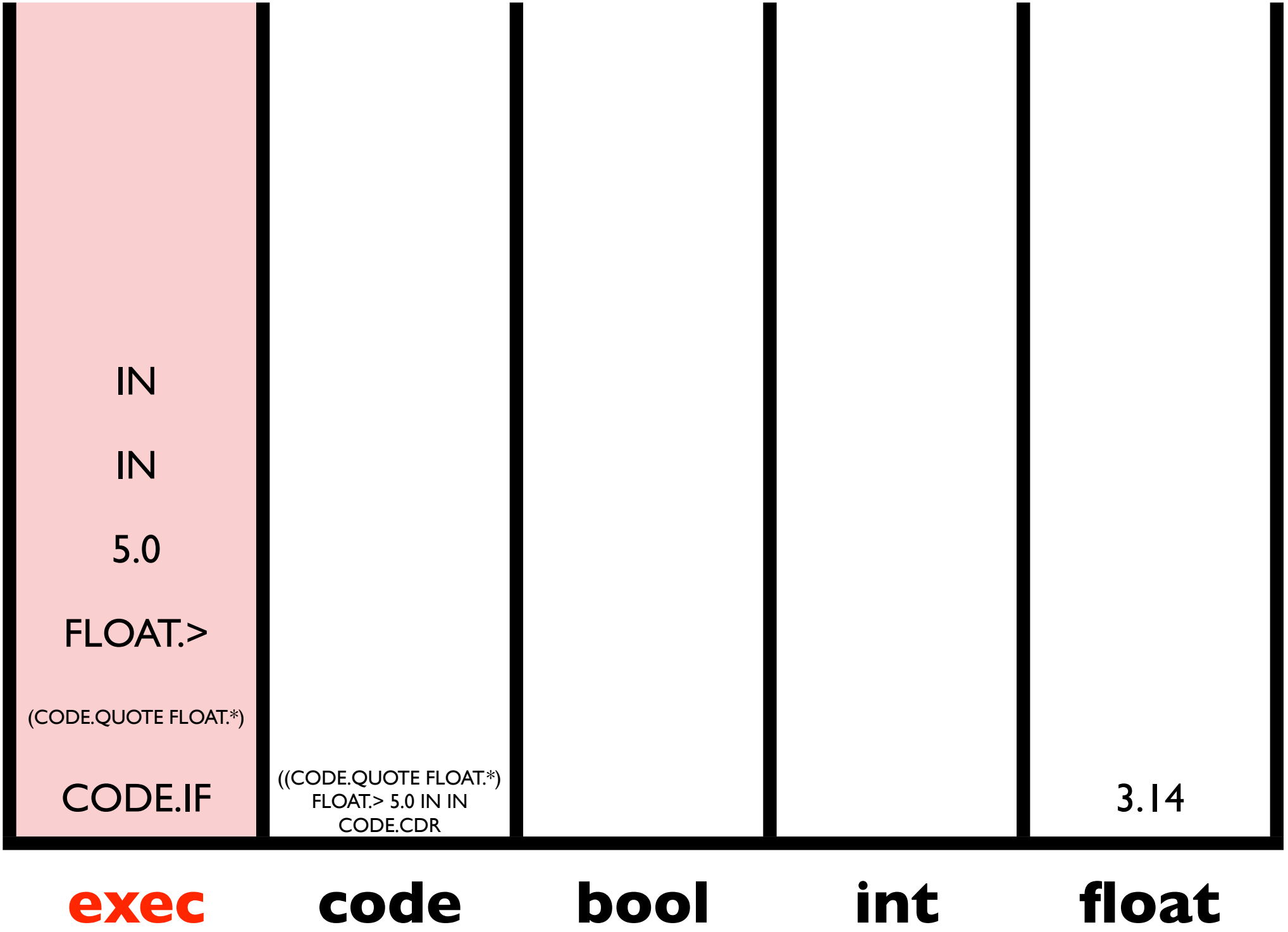
IN=4.0

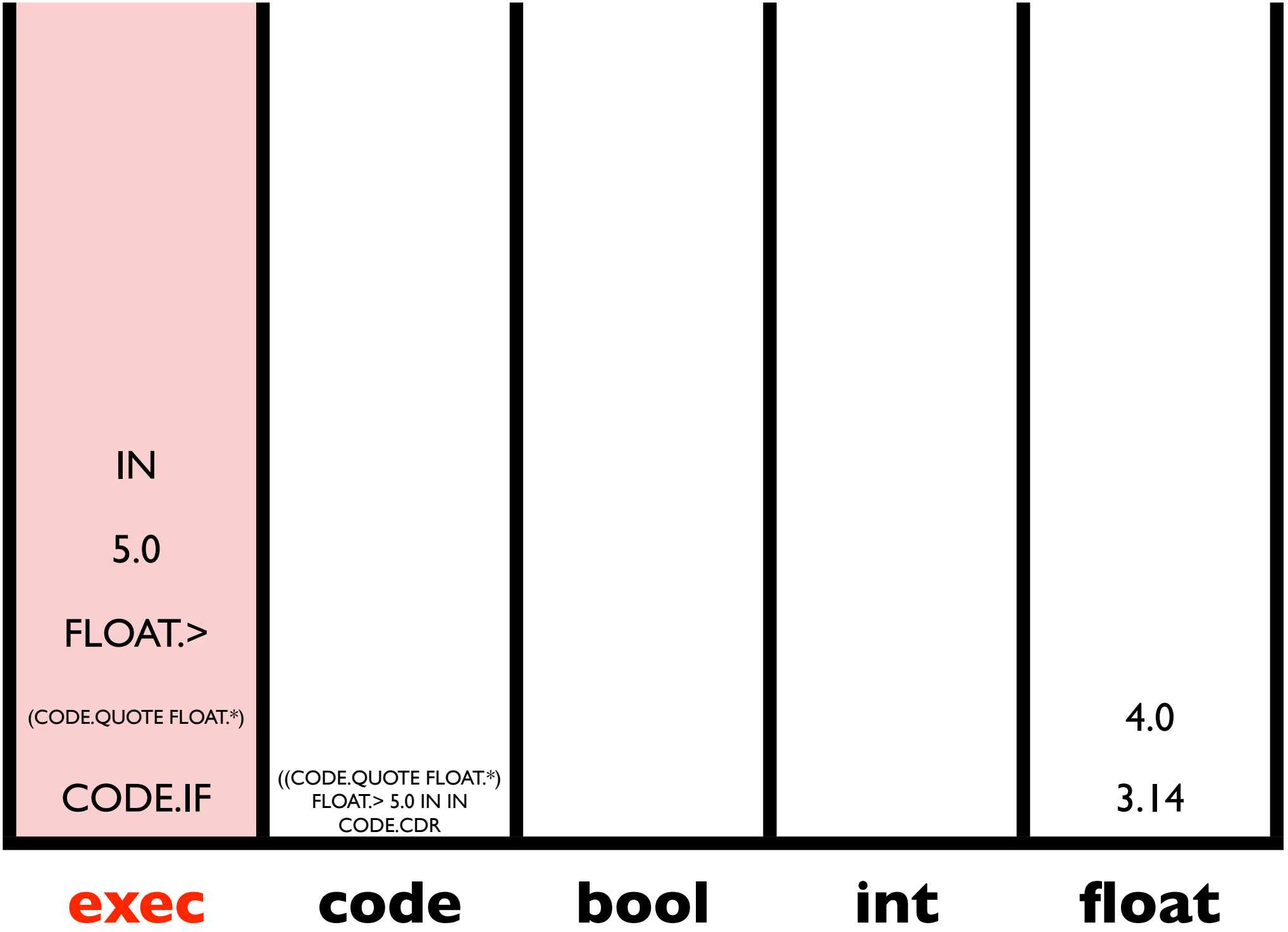


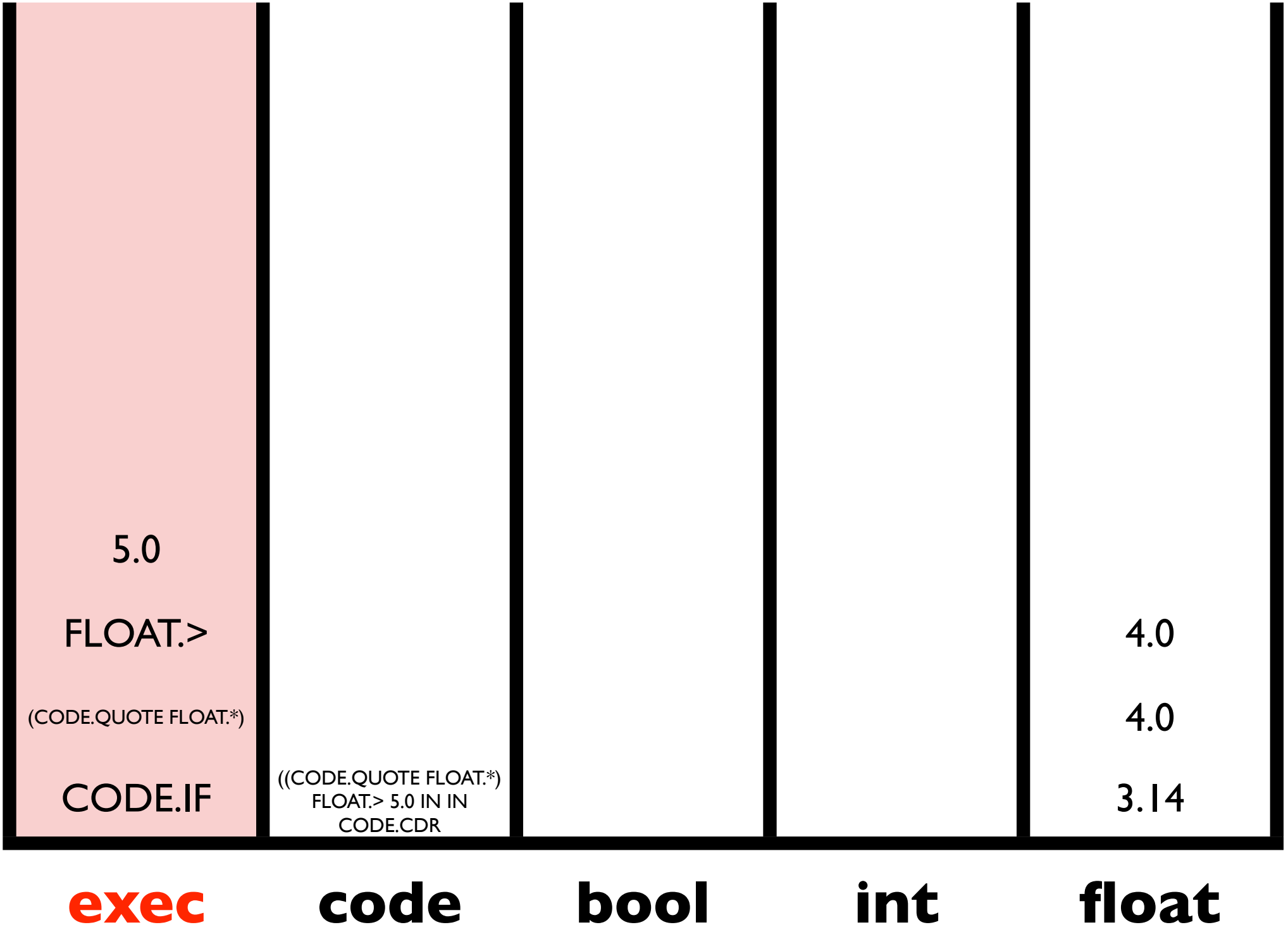

3.14
CODE.REVERSE
CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
(3.14 CODE.REVERSE
CODE.CDR IN IN
5.0 FLOAT.>
exec
code
bool
int
float

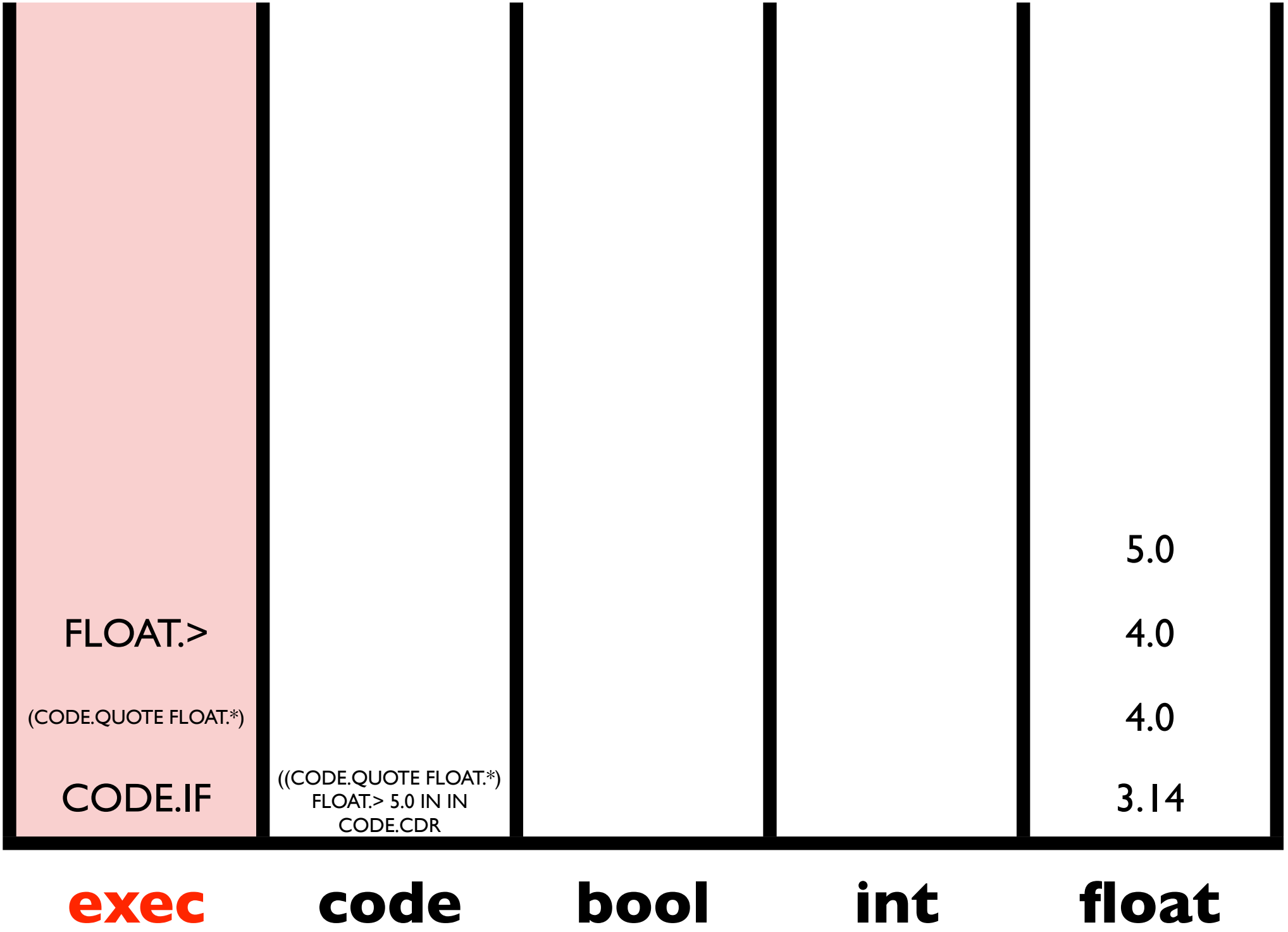

CODE.REVERSE
CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
(3.14 CODE.REVERSE CODE.CDR IN IN 5.0 FLOAT.>
3.14
exec
code
bool
int
float

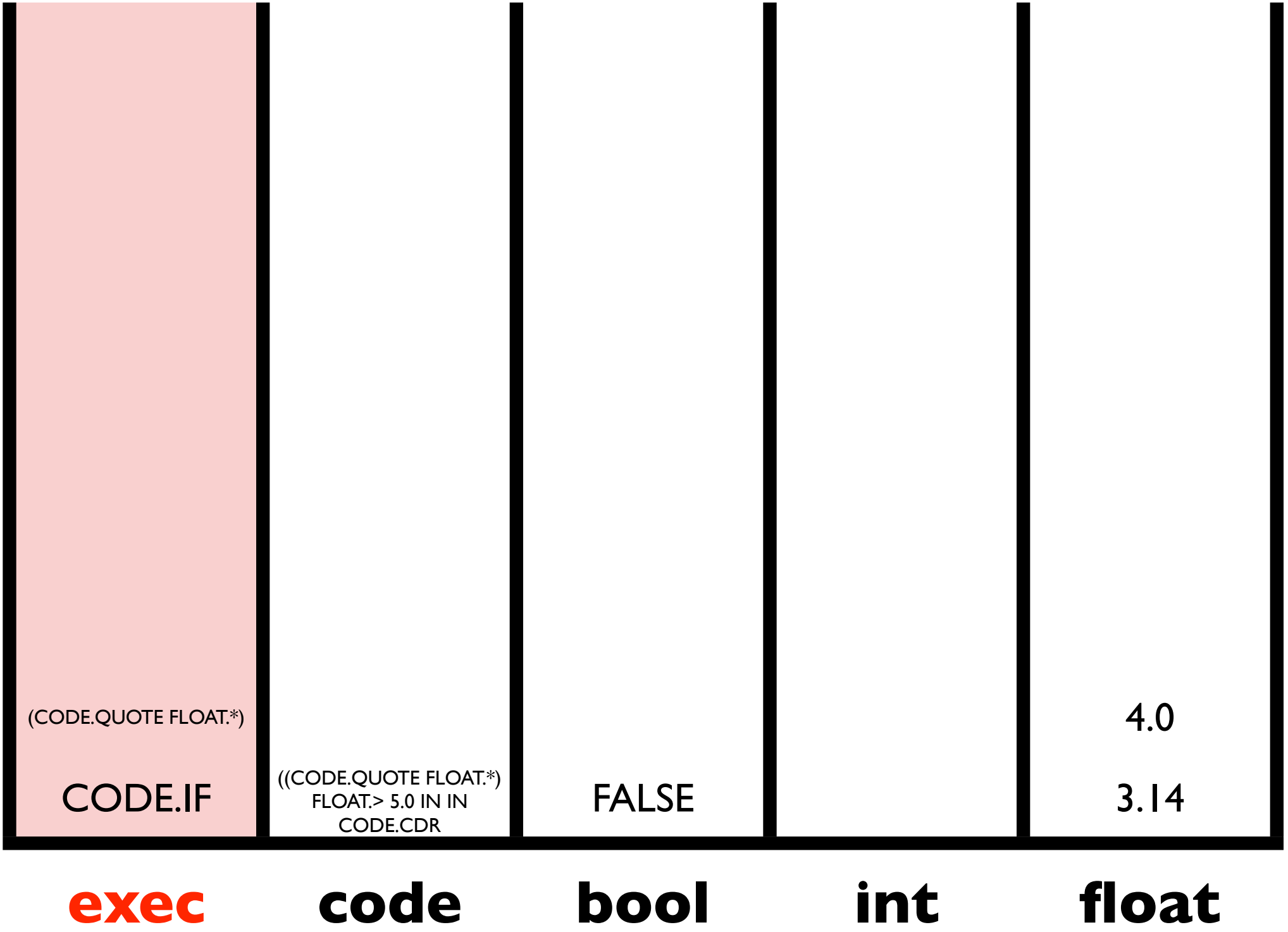

CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
(CODE.IF (CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
3.14
exec
code
bool
int
float

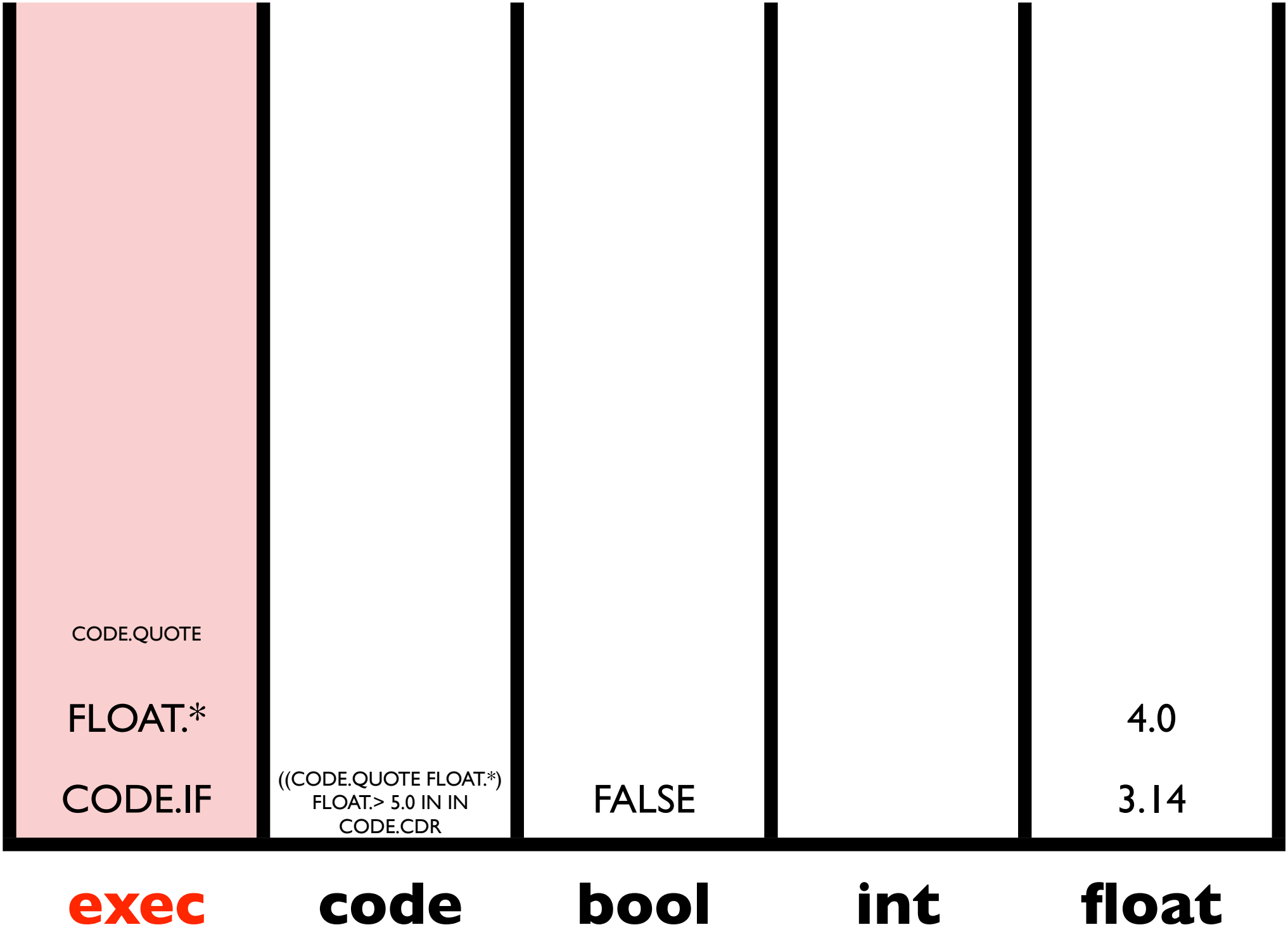

IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
3.14
exec
code
bool
int
float

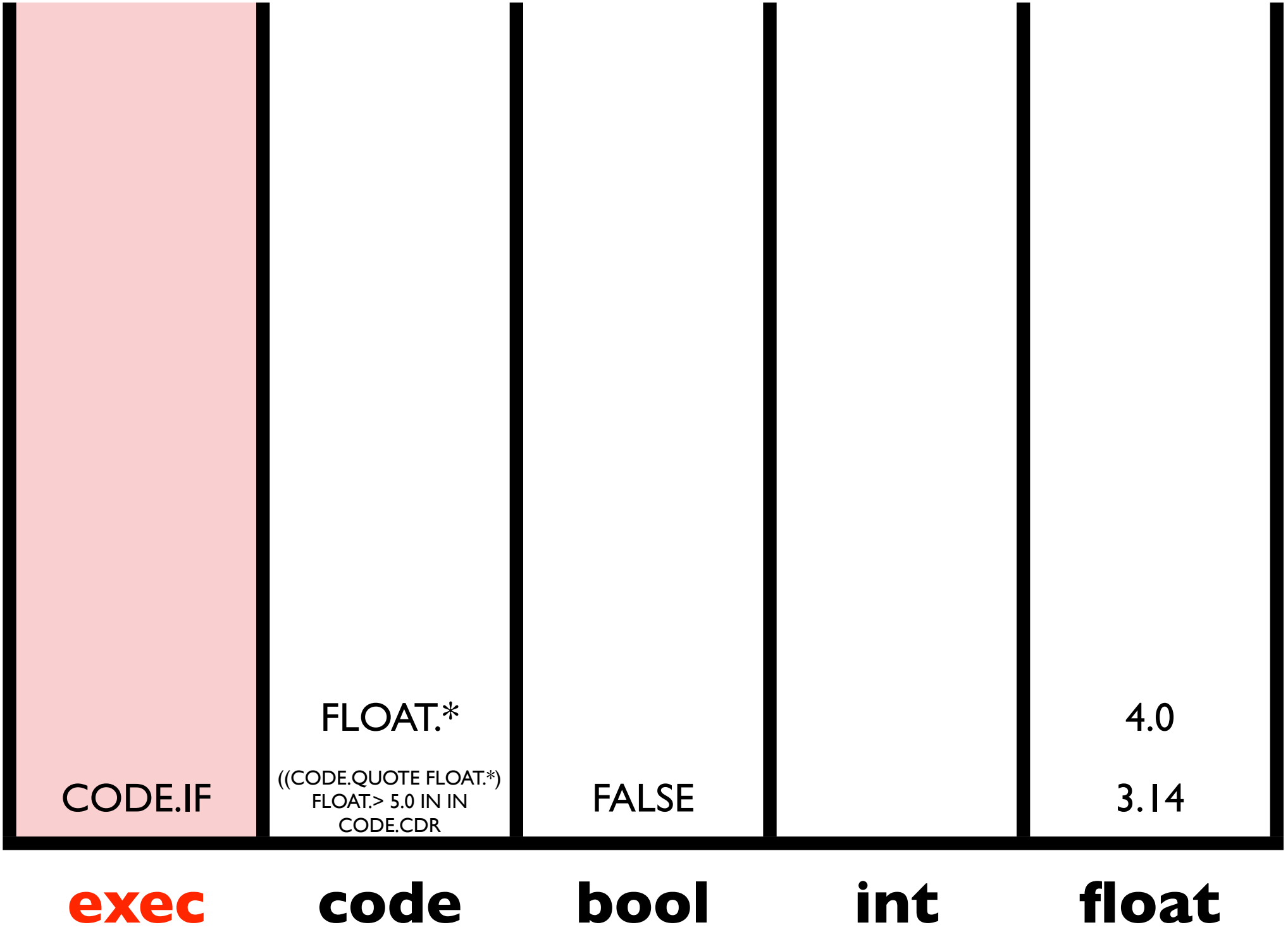

IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
4.0
3.14
exec
code
bool
int
float

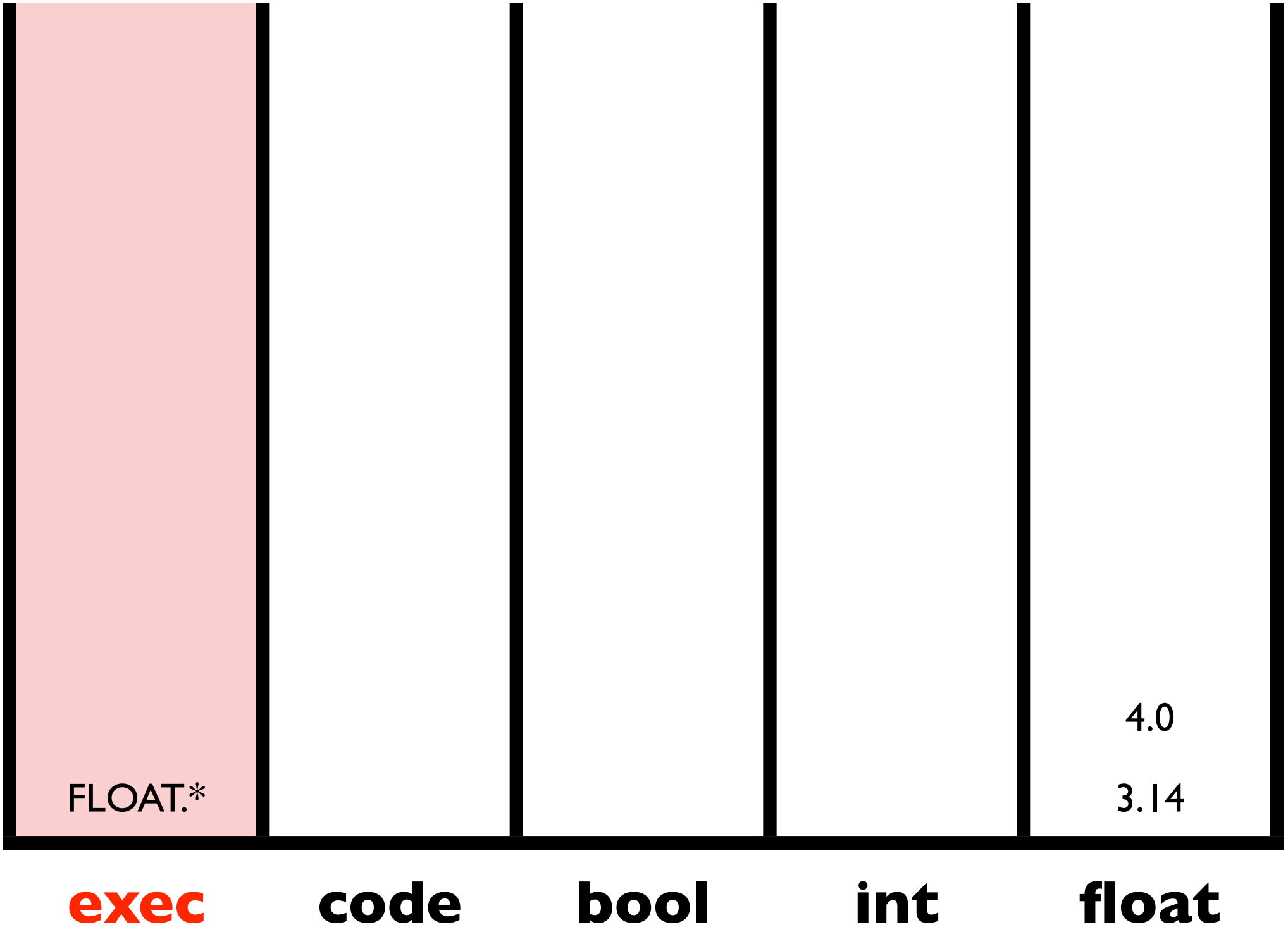

5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
4.0
4.0
3.14
exec
code
bool
int
float

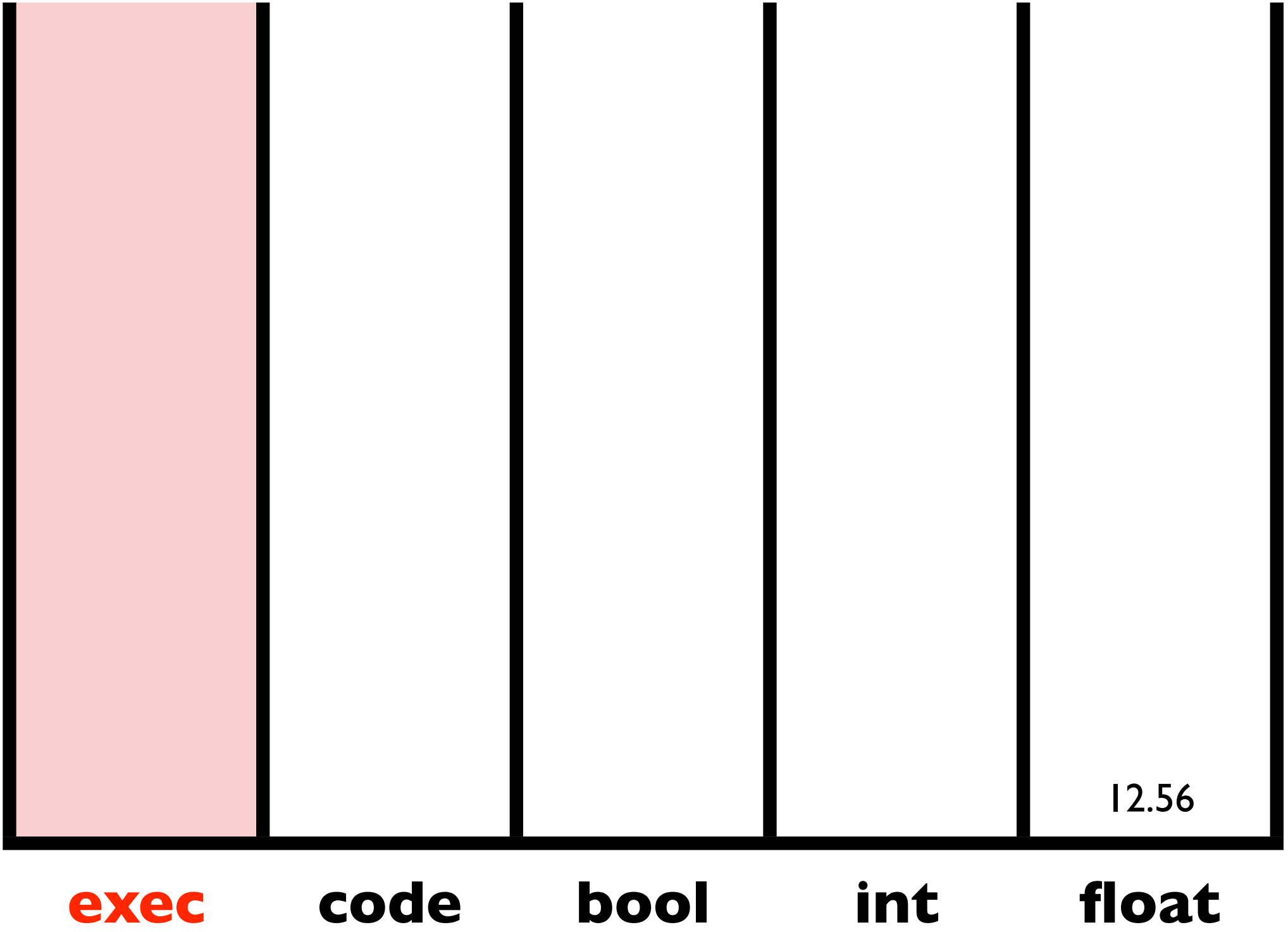

FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
5.0
4.0
4.0
3.14
exec
code
bool
int
float


(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
FALSE
4.0
3.14
exec
code
bool
int
float

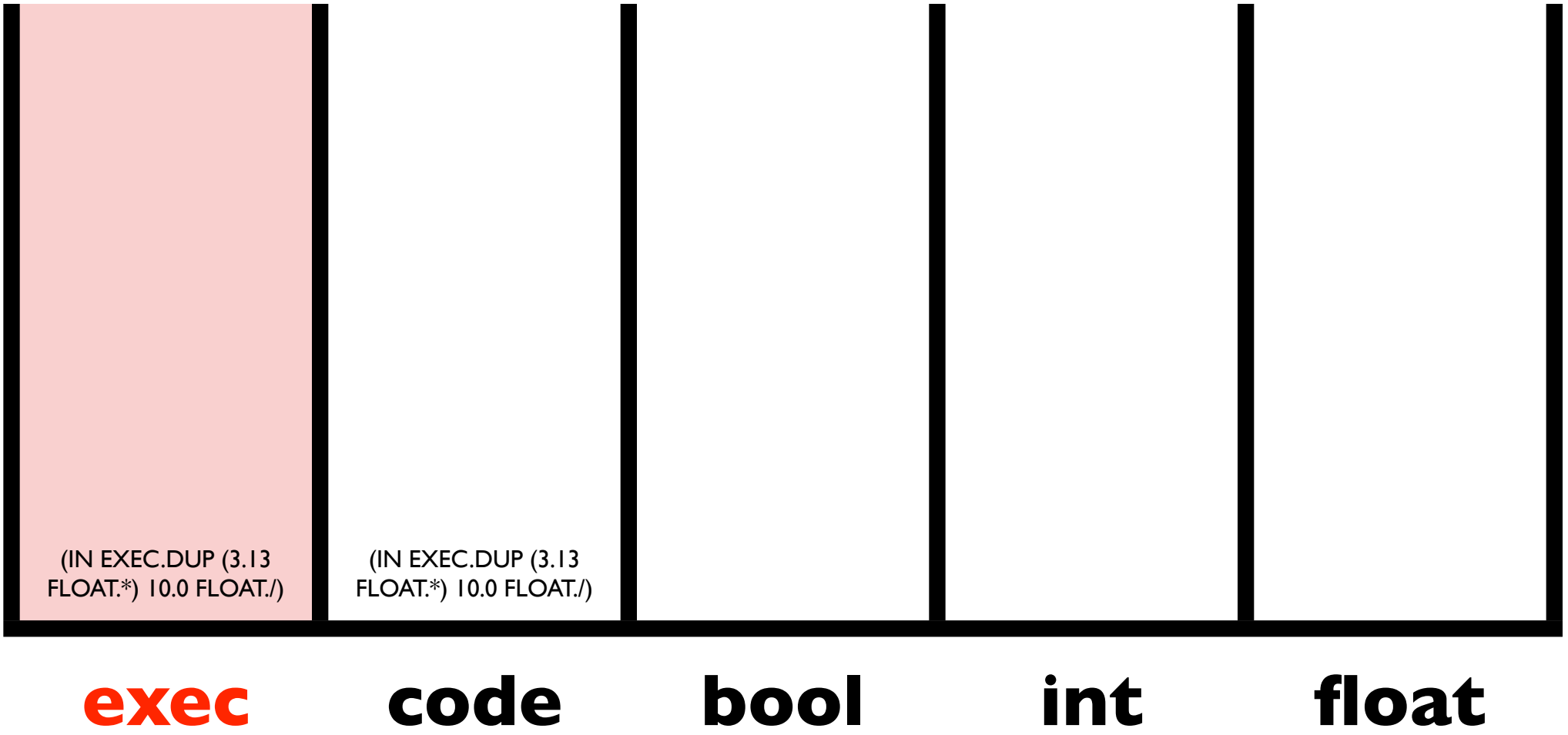

CODE.QUOTE
FLOAT.*
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
FALSE
4.0
3.14
exec
code
bool
int
float

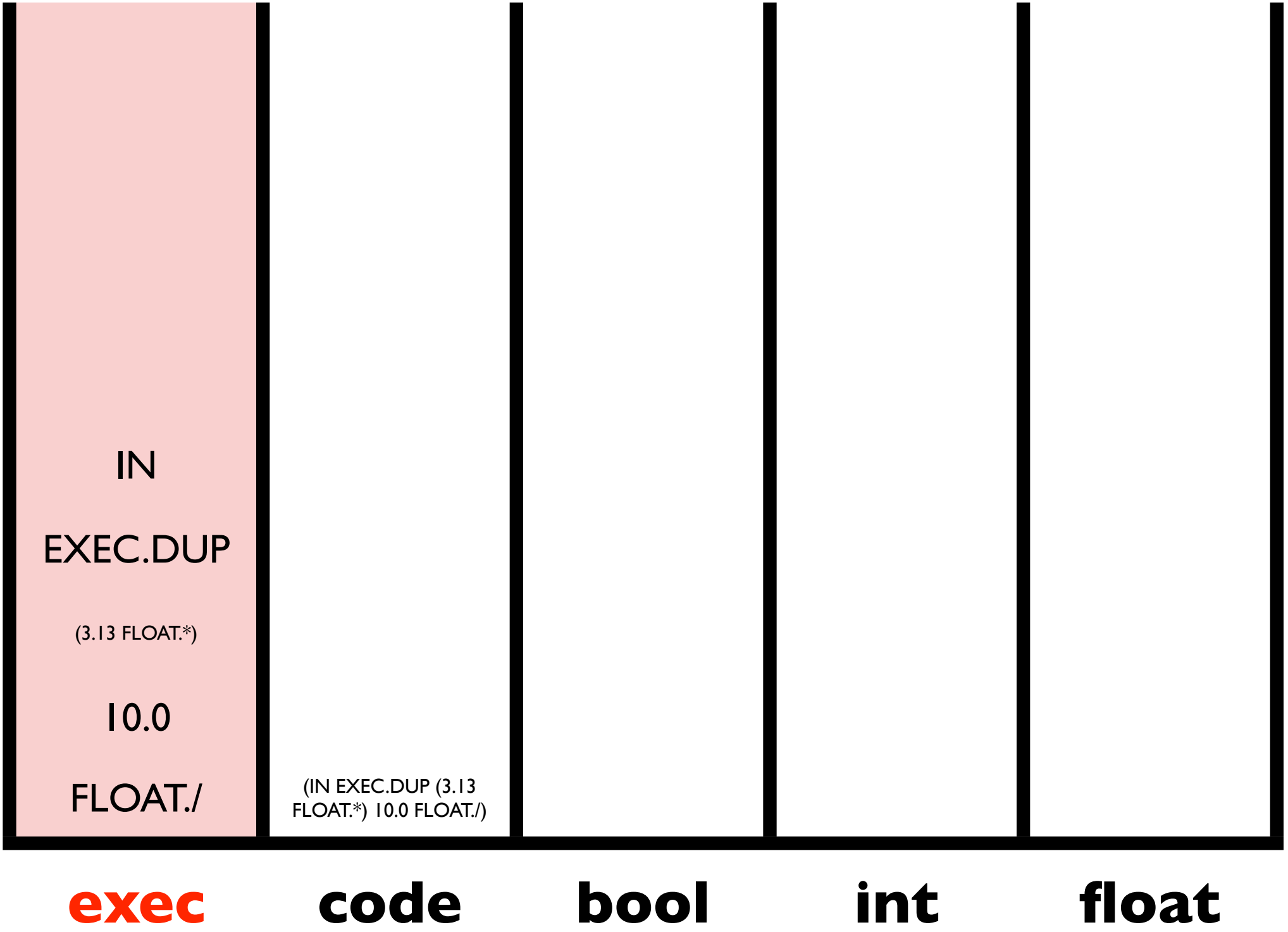

CODE.IF
FLOAT.*
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
FALSE
4.0
3.14
exec
code
bool
int
float

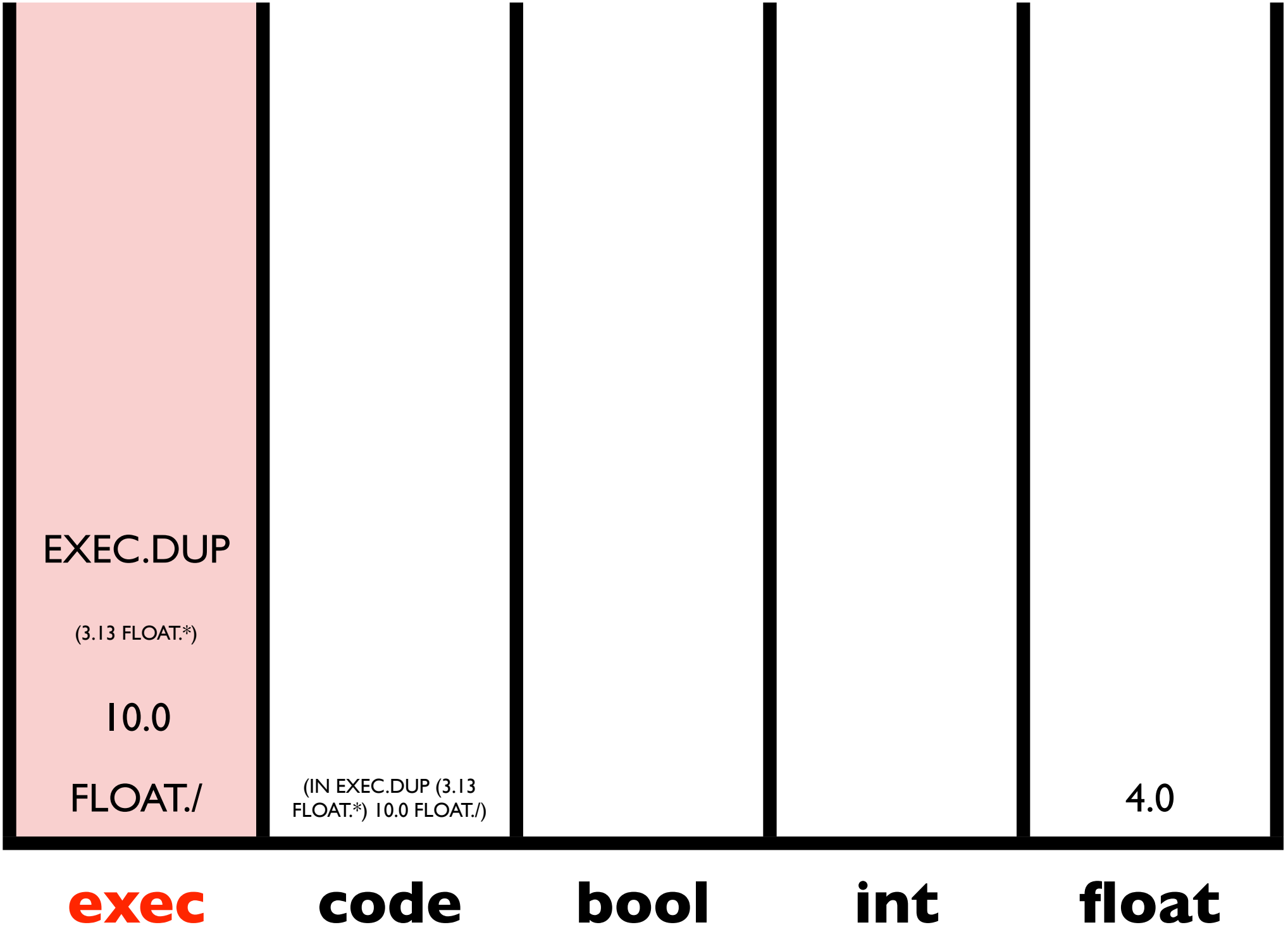

FLOAT.*
4.0
3.14
exec
code
bool
int
float

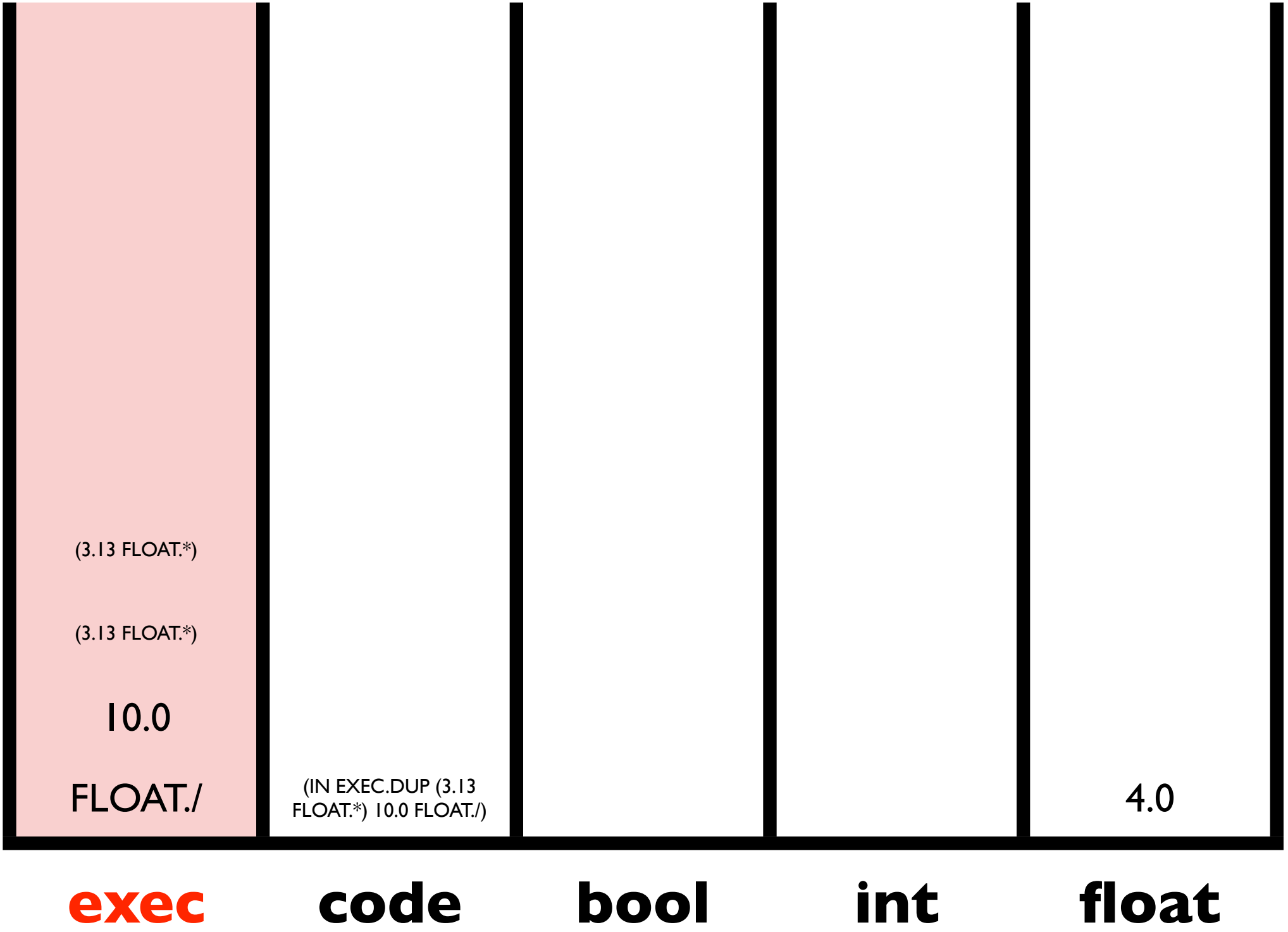

12.56
exec
code
bool
int
float

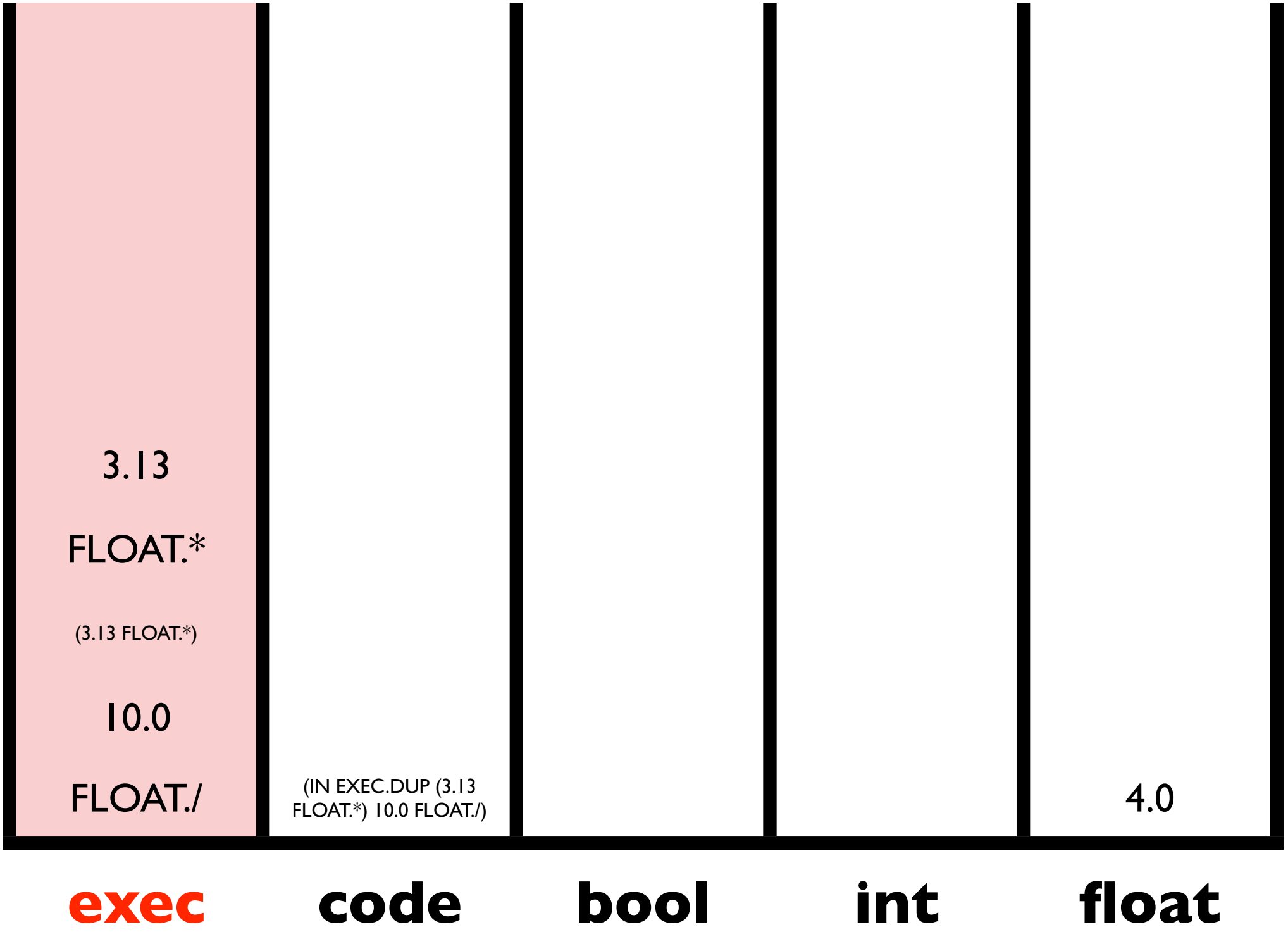
```
(IN EXEC.DUP (3.13 FLOAT.*)
10.0 FLOAT./)

IN=4.0
```

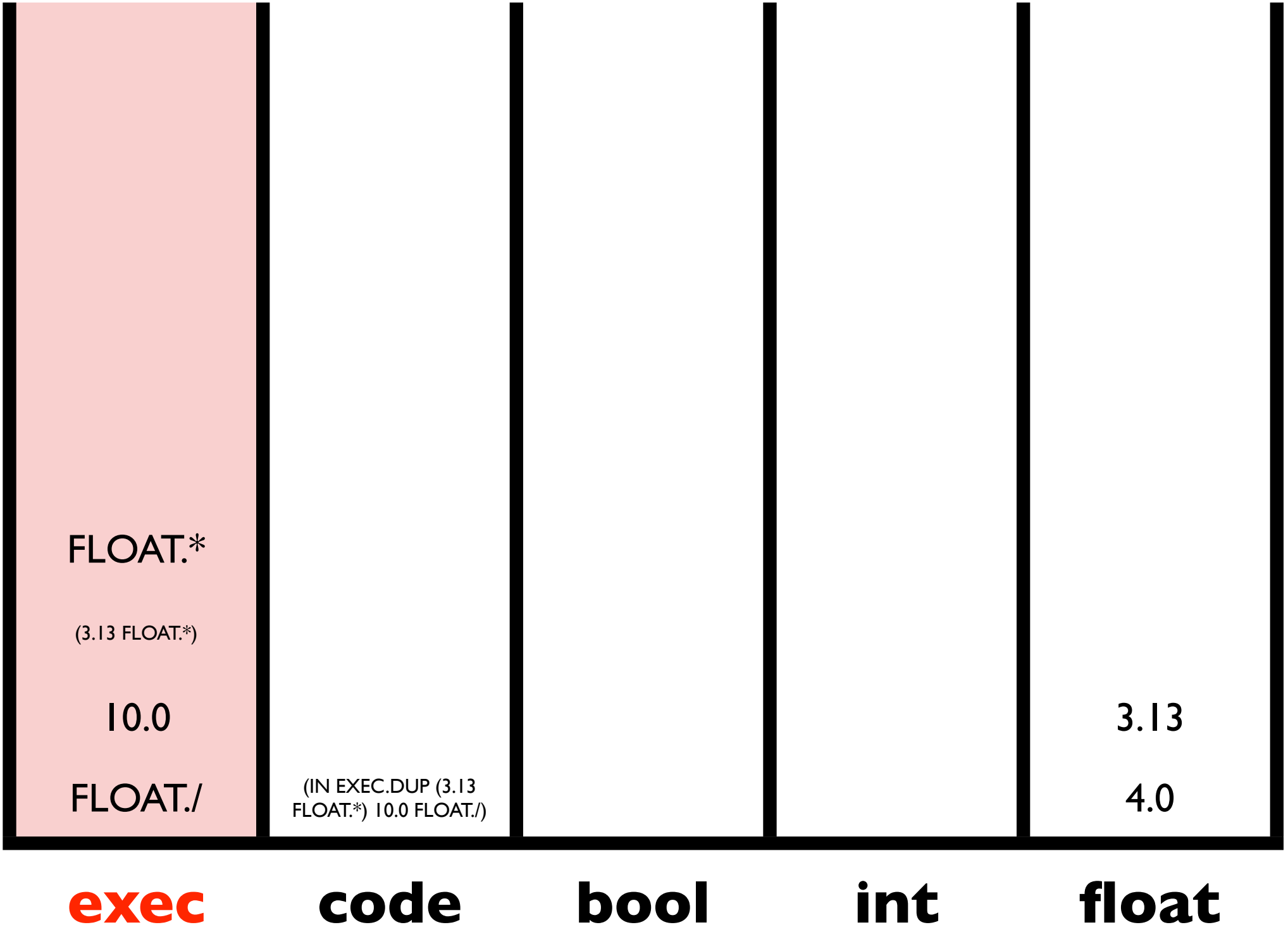


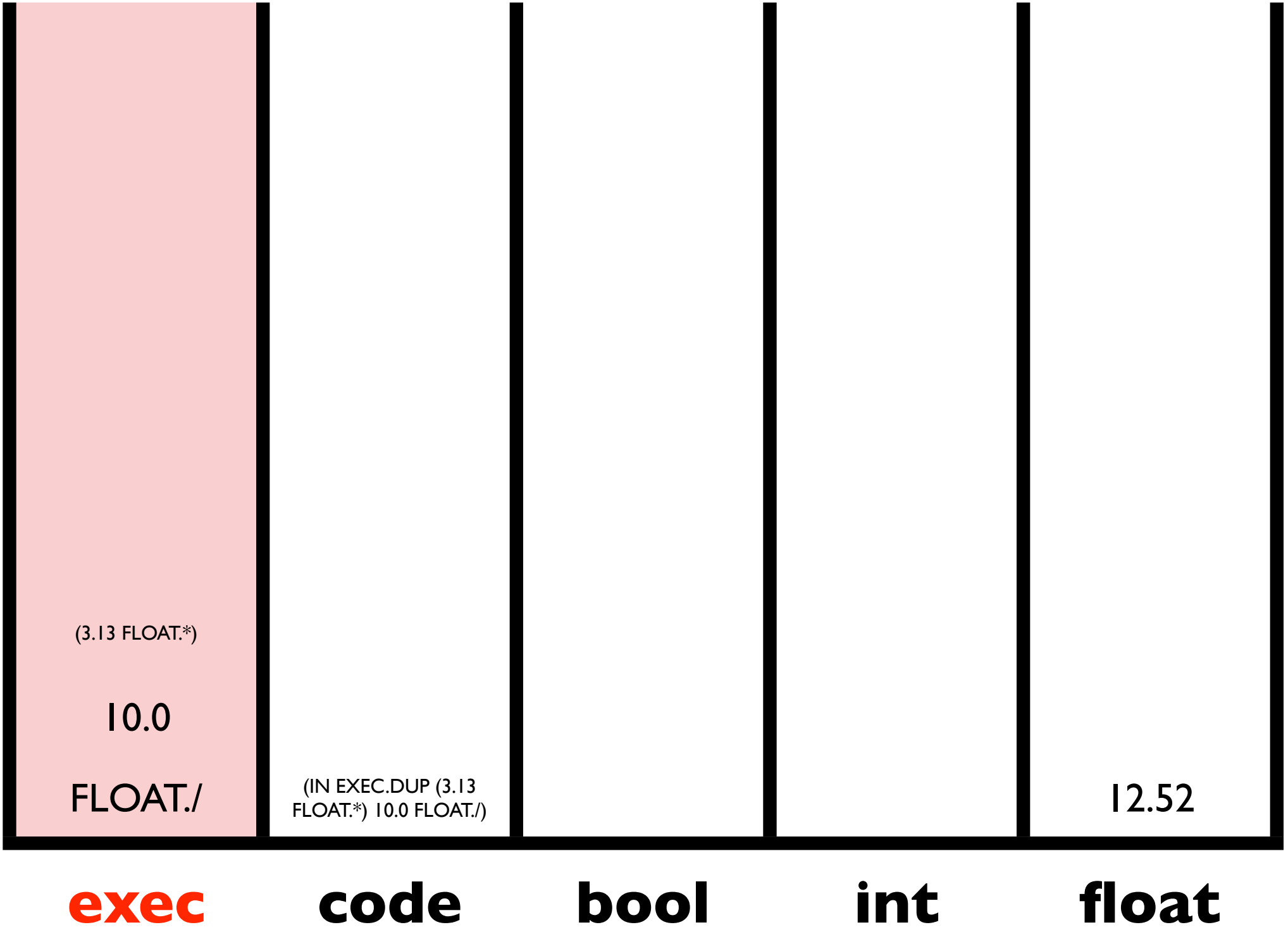

IN
EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
exec
code
bool
int
float

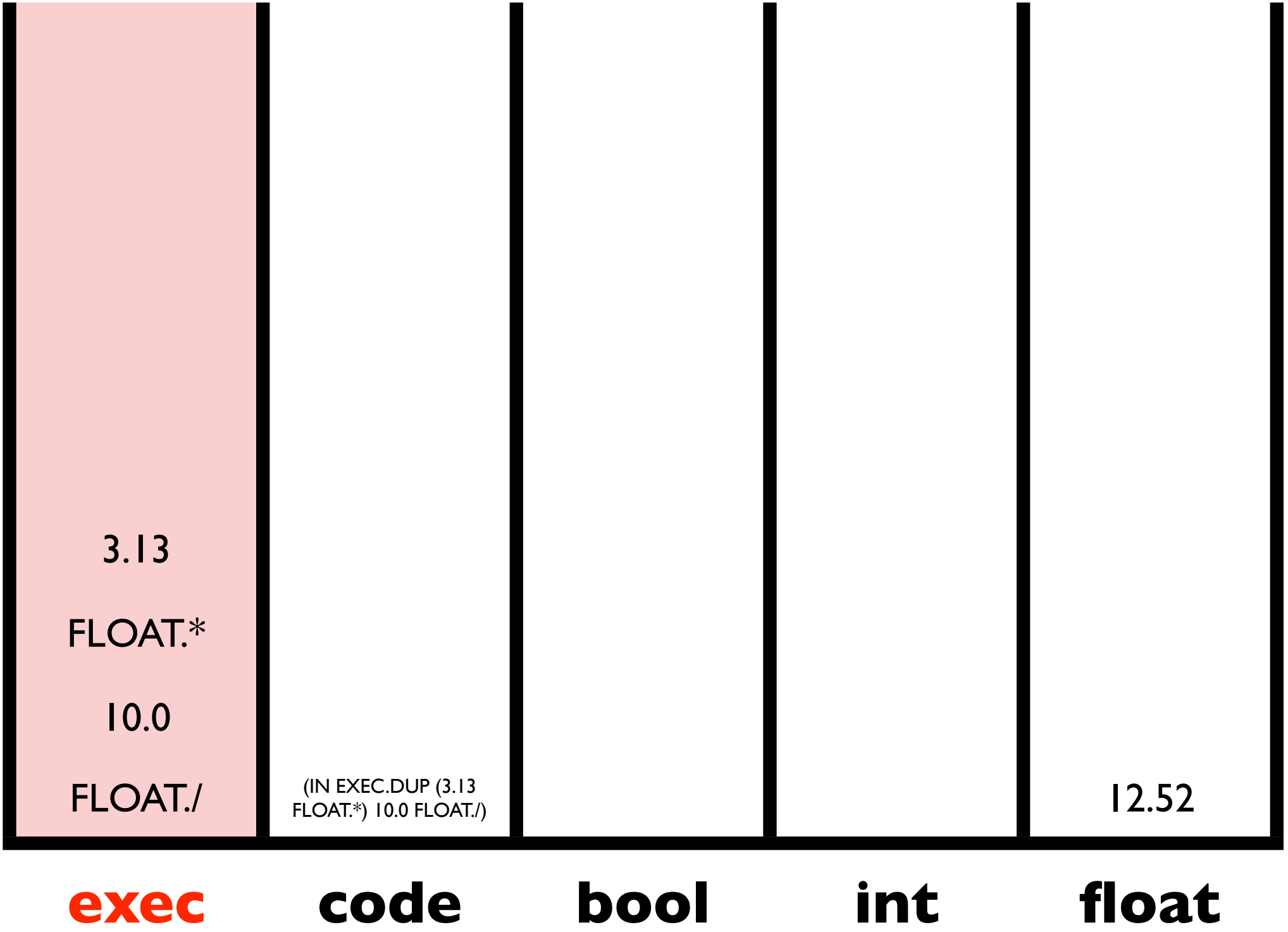

EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

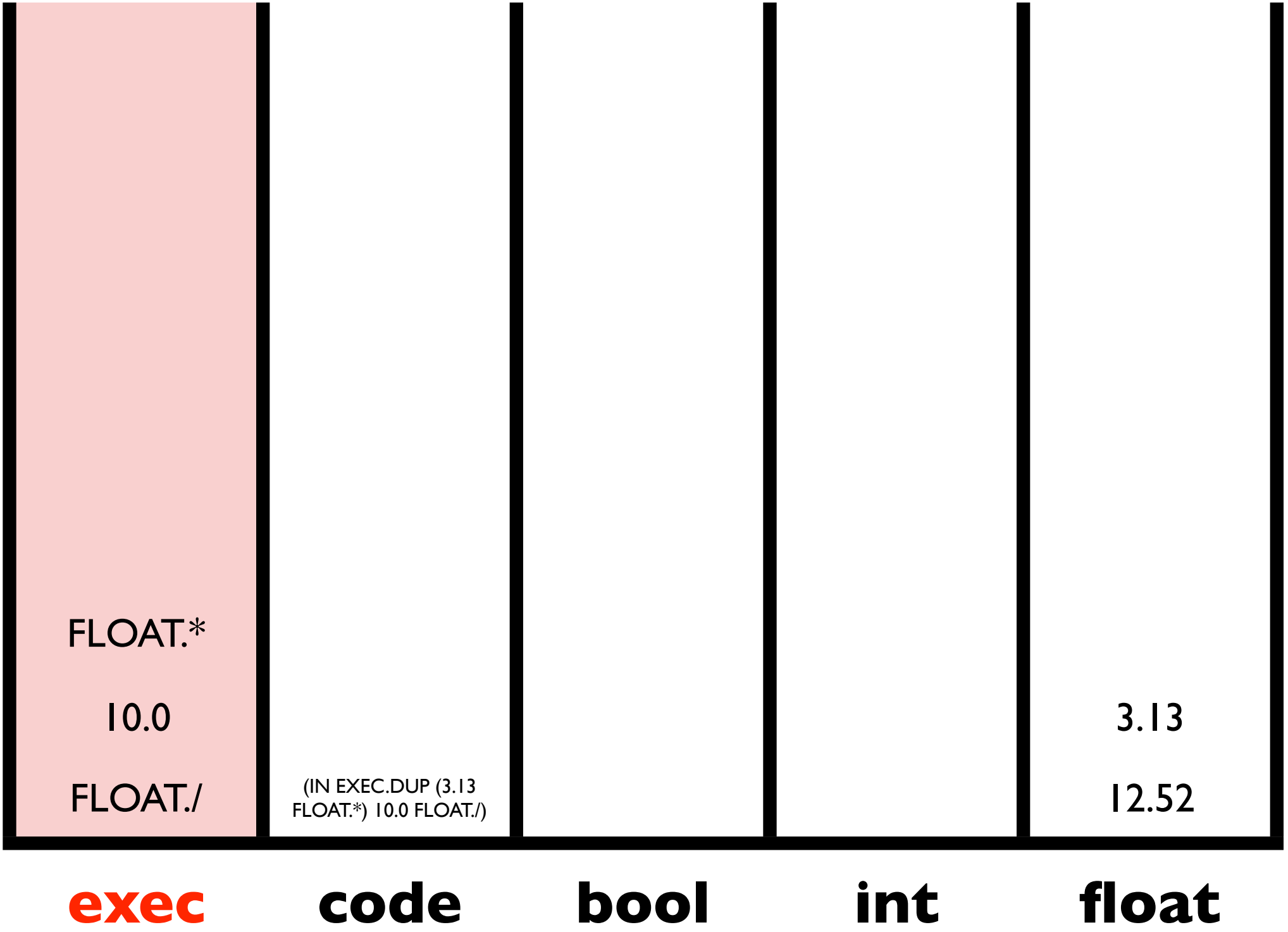

(3.13 FLOAT.*)
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

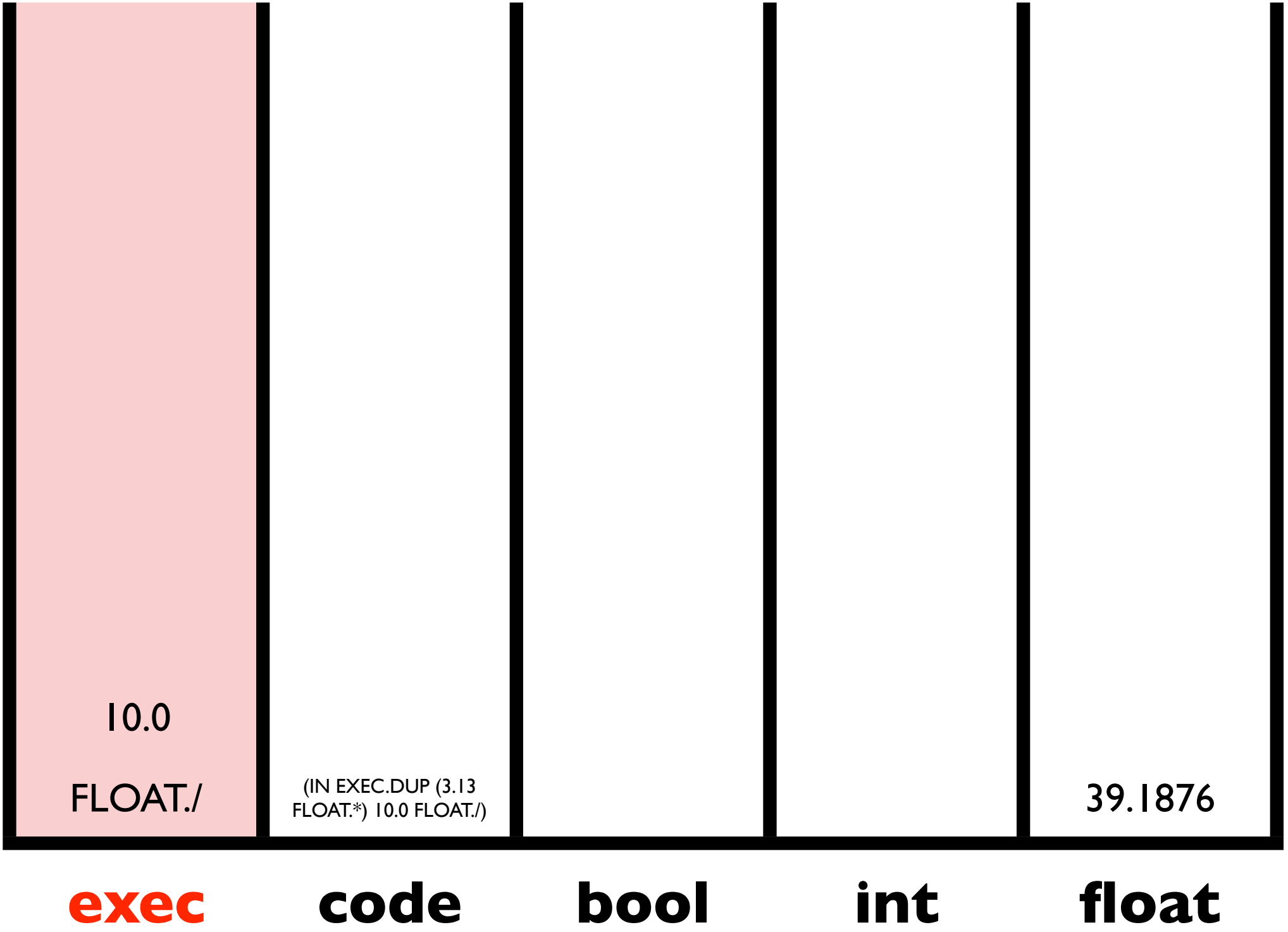

3.13
FLOAT.*
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

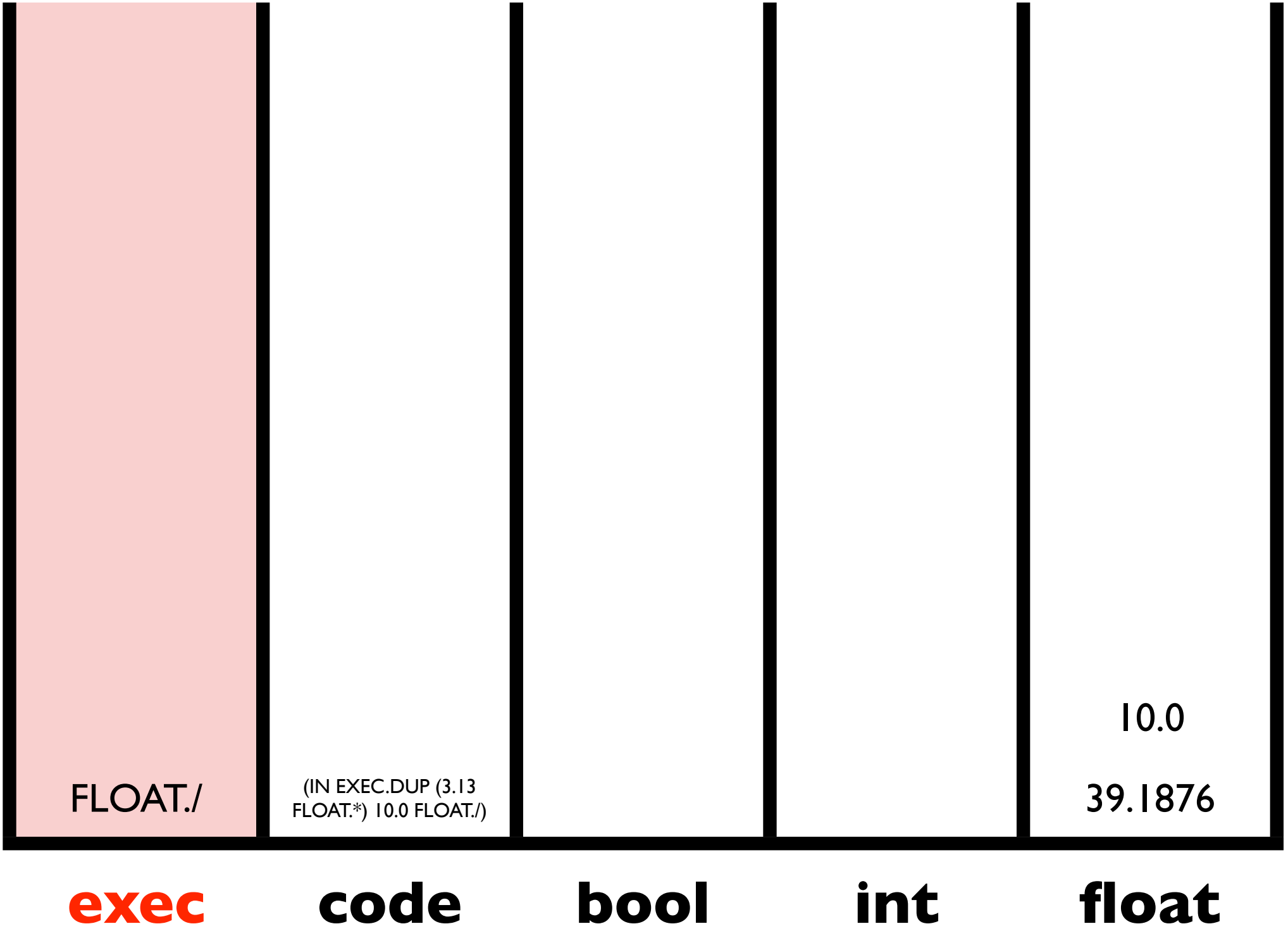

FLOAT.*
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
3.13
4.0
exec
code
bool
int
float

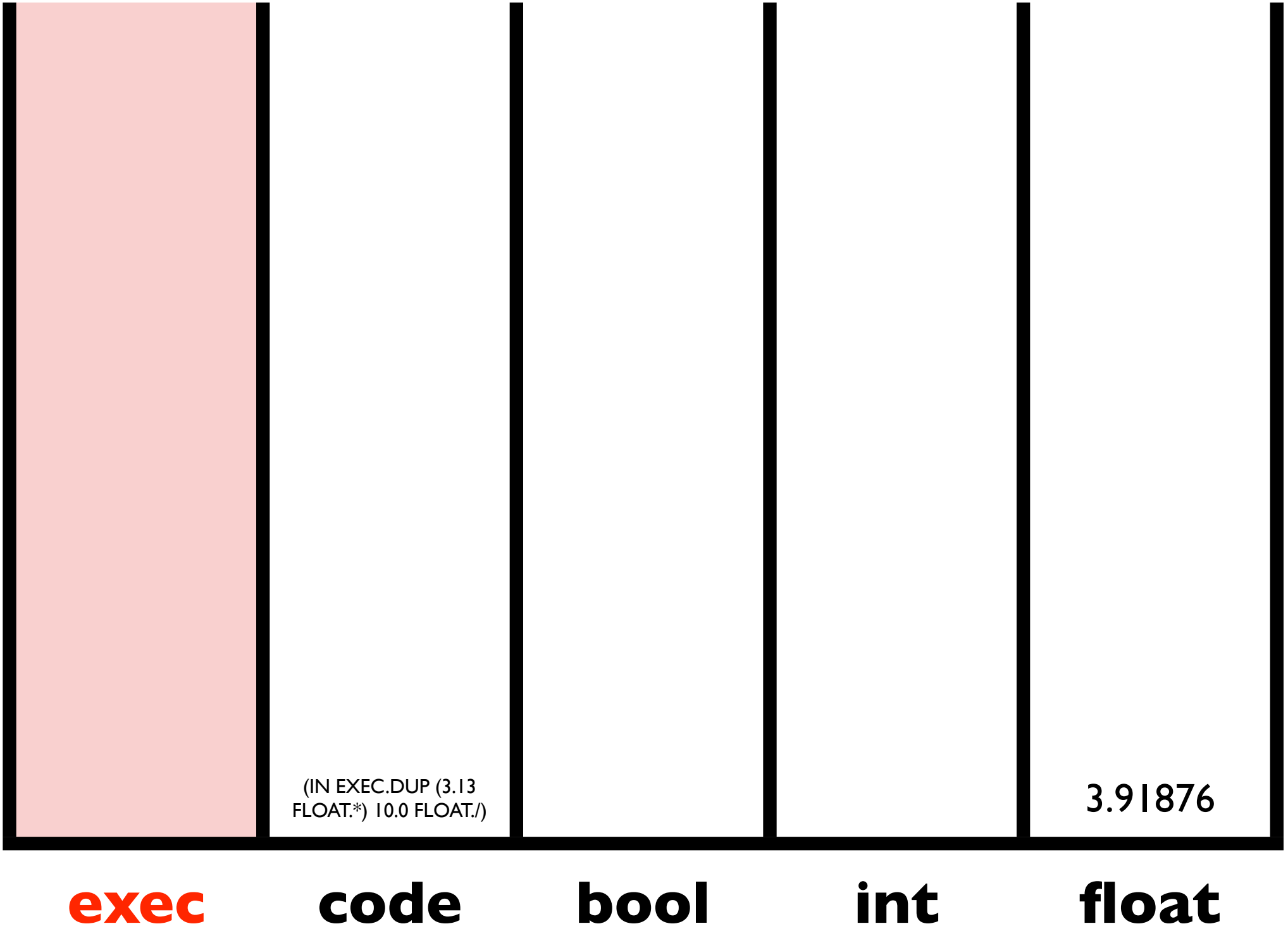

(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
12.52
exec
code
bool
int
float


3.13
FLOAT.*
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
12.52
exec
code
bool
int
float


FLOAT.*
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
3.13
12.52
exec
code
bool
int
float


10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
39.1876
exec
code
bool
int
float


FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
10.0
39.1876
exec
code
bool
int
float


(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
3.91876
exec
code
bool
int
float

Modules in Push

- Transform/execute code as data: Works, emerges, but stack-based module reference won't scale well
- Execution stack manipulation:

  ```
  (3 exec.dup (1 integer.+))
  ```

  More parsimonious, but same scaling issue
- Named modules:

  ```
  (plus1 exec.define (1 integer.+)) ... plus1
  ```

  Coordinating definitions/references is tricky ***and this never arises in evolution!***

Modularity
Ackley and Van Belle

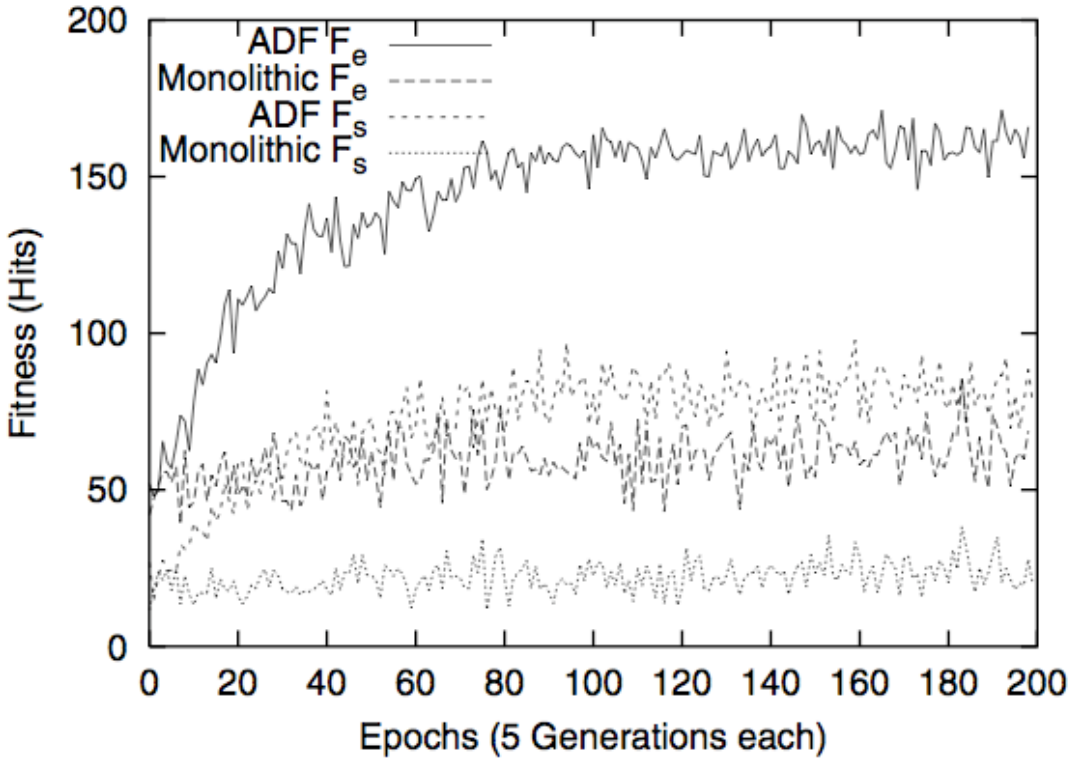
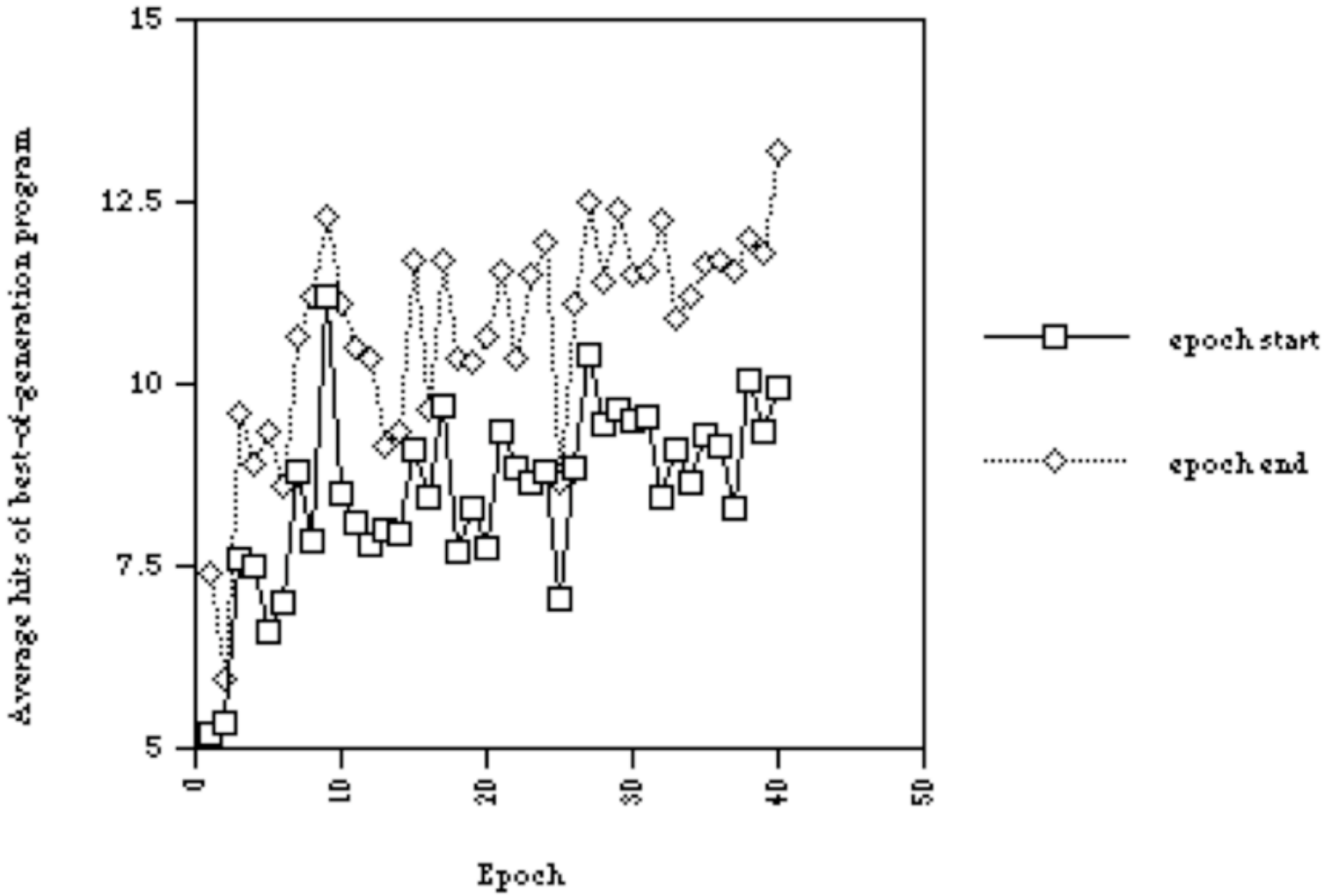


Figure 2: Average fitness values at the start (F_s) and end (F_e) of each epoch when regressing to $y = A\sin(Ax)$. A is selected at the start of each epoch uniformly from the range $[0,6)$.

Code-as-data Modularity in Push

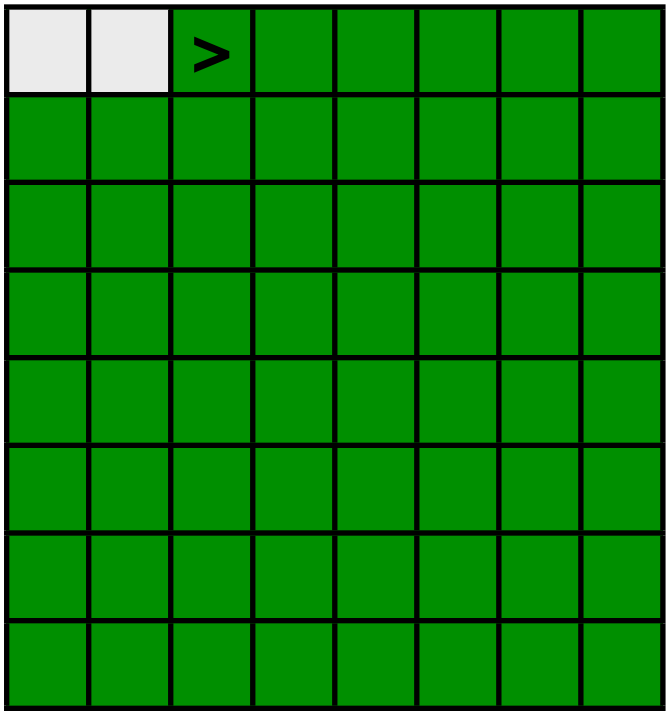


Tags in Push

- Tags are integers embedded in instruction names
- Instructions like tag.exec.123 tag values
- Instructions like tagged.456 recall values by *closest matching* tag
- If a single value has been tagged then all tag references will recall (and execute) values
- The number of tagged values can grow incrementally over evolutionary time

Lawnmower Problem

- Used by Koza to demonstrate utility of ADFs for scaling GP up to larger problems

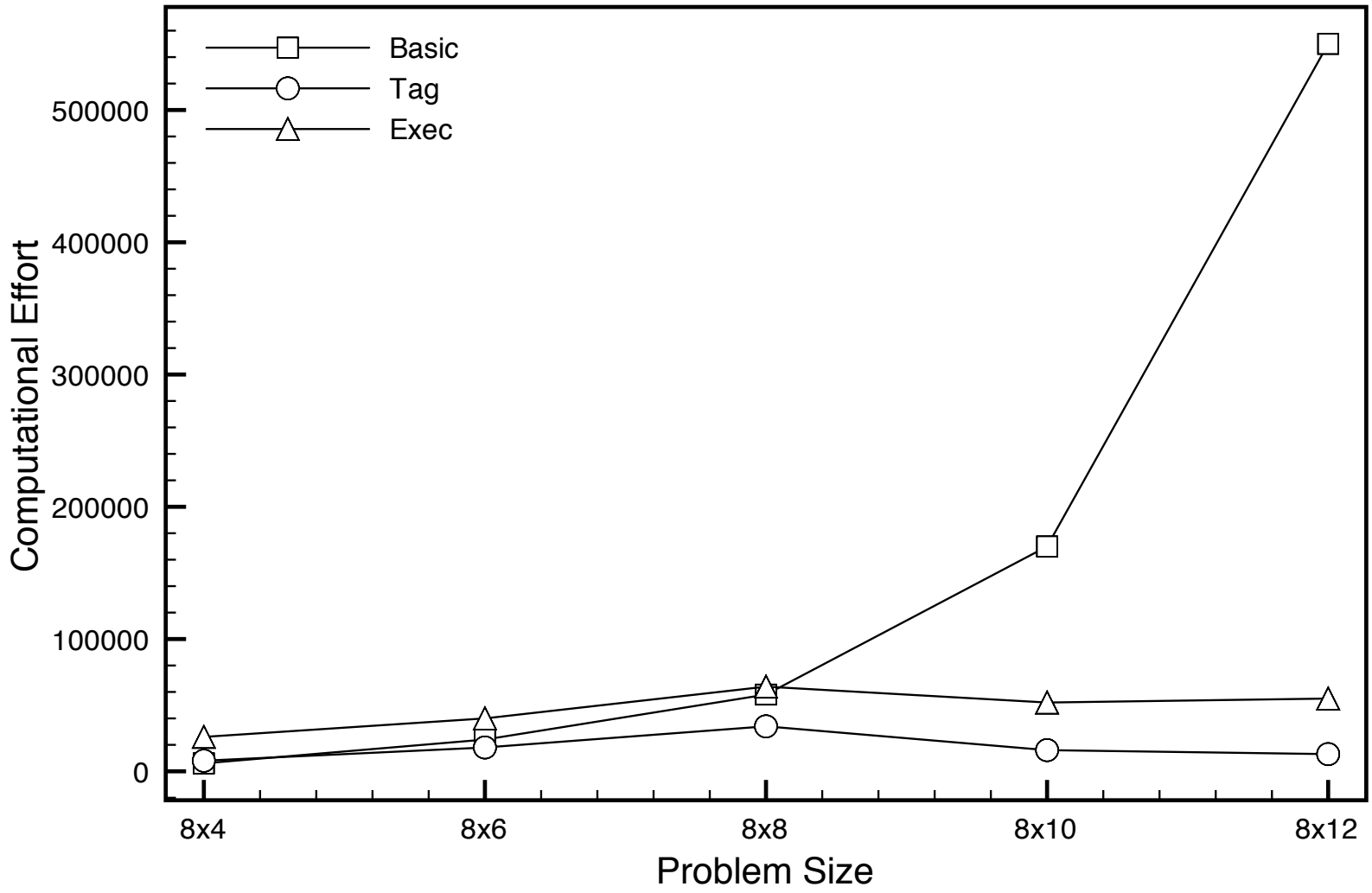


Lawnmower Instructions

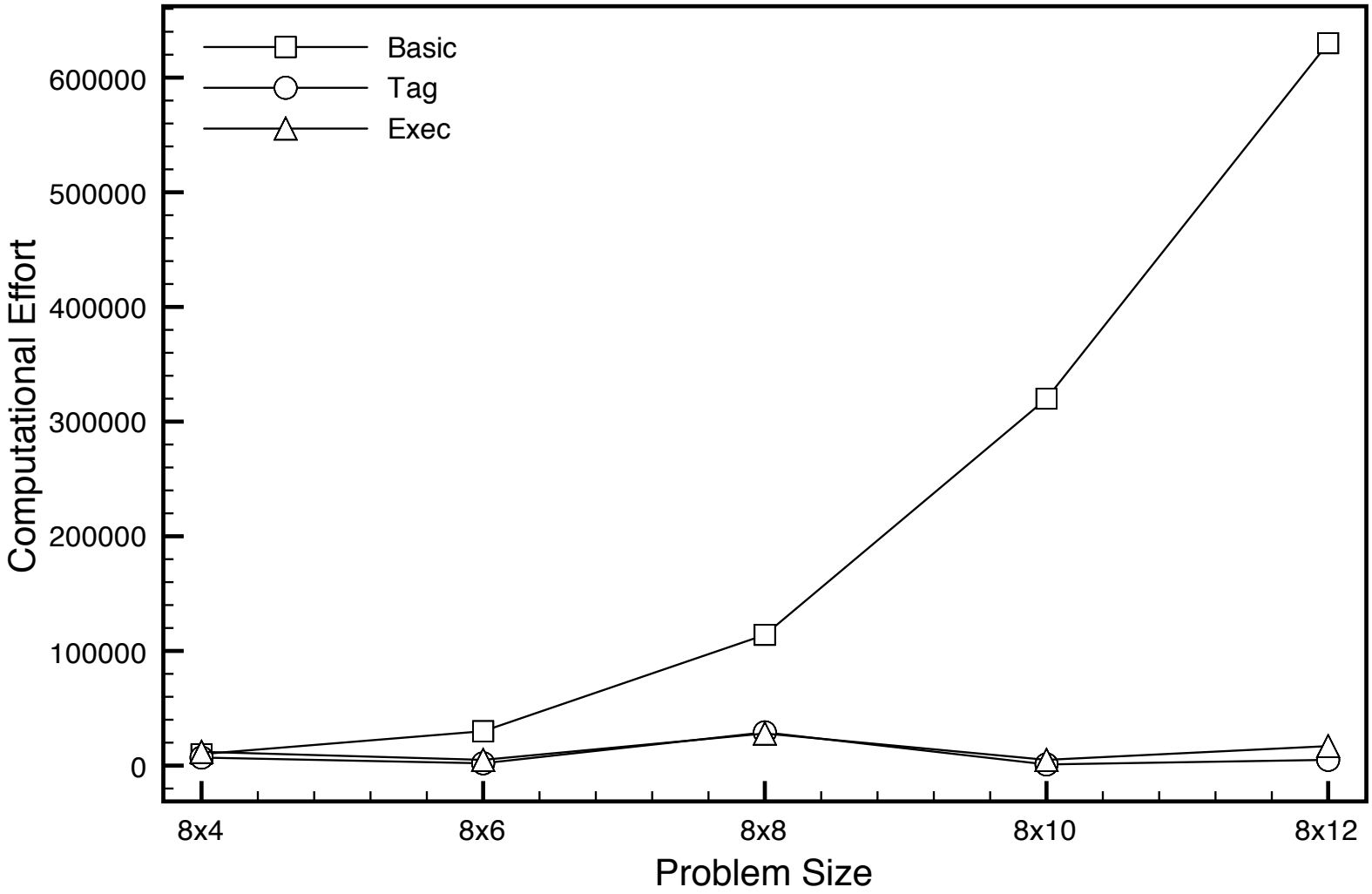
Condition	Instructions
Basic	left, mow, v8a, frog, $\mathcal{R}_{v8}$
Tag	left, mow, v8a, frog, $\mathcal{R}_{v8}$, tag.exec.[1000], tagged.[1000]
Exec	left, mow, v8a, frog, $\mathcal{R}_{v8}$, exec.dup, exec.pop, exec.rot, exec.swap, exec.k, exec.s, exec.y

Lawnmower Effort*

* with frog=noop bug



Lawnmower Effort

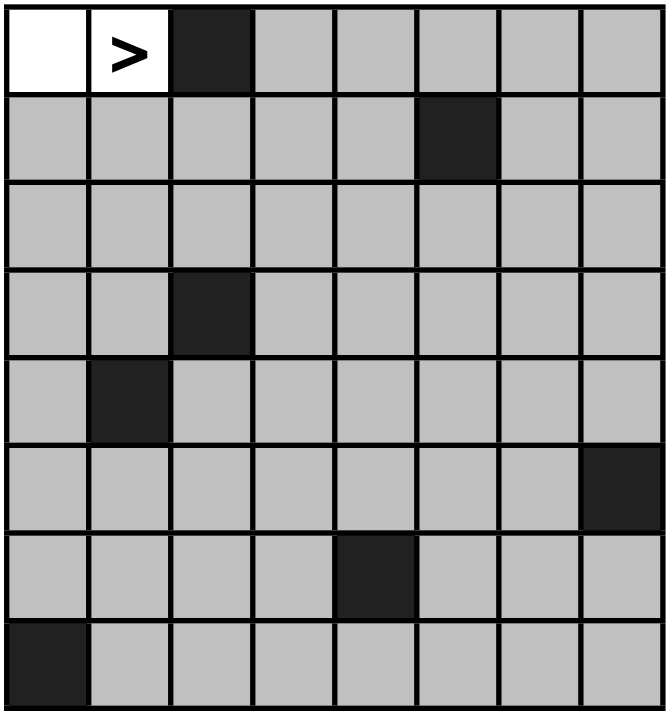


Lawnmower Effort

	problem size				
	8x4	8x6	8x8	8x10	8x12
instr set					
basic	10000	30000	114000	320000	630000
tag	7000	2000	29000	<1000	5000
exec	12000	5000	28000	5000	17000

Dirt-Sensing, Obstacle-Avoiding Robot Problem

Like the lawnmower problem but harder and less uniform

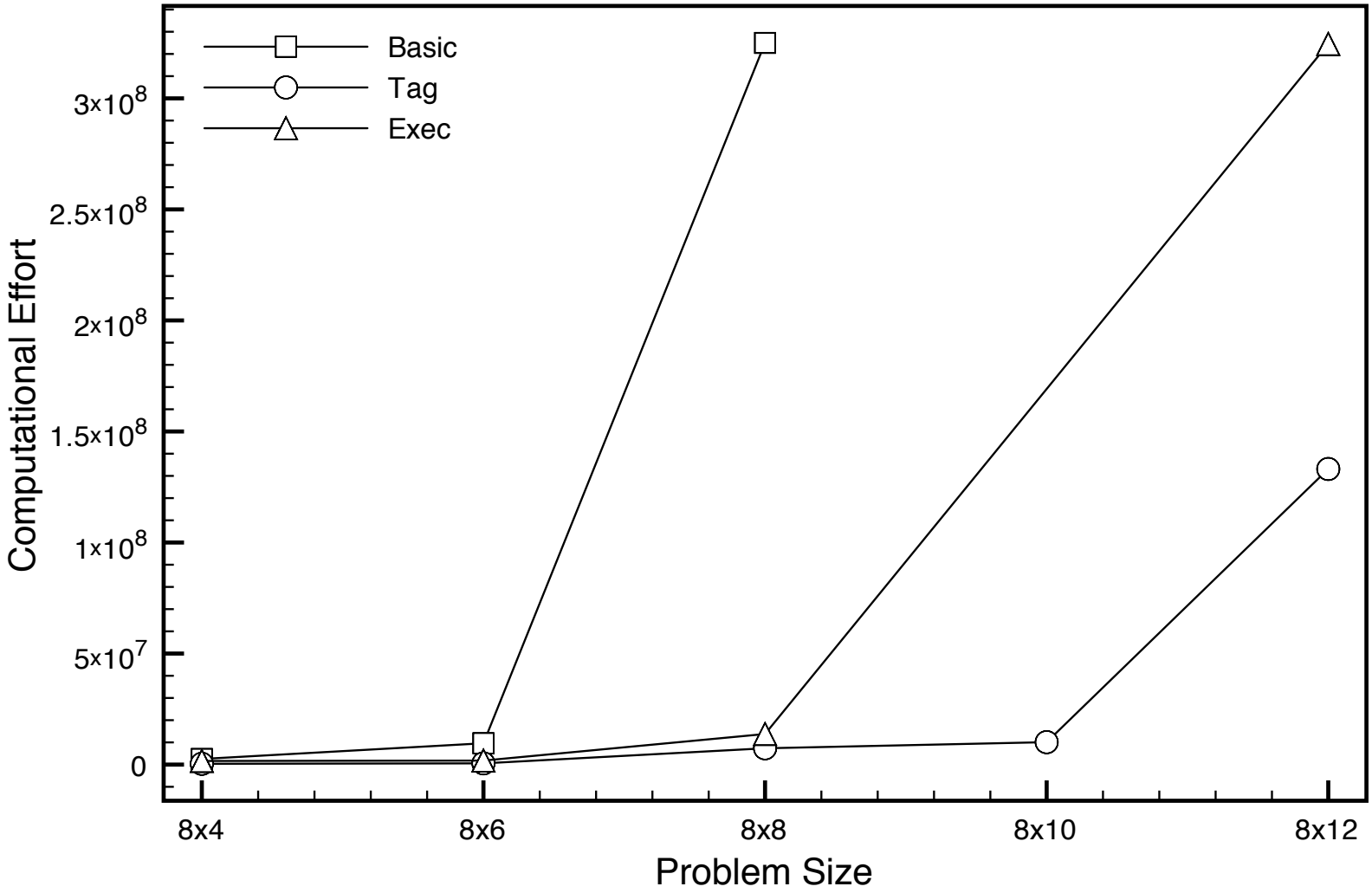


DSOAR Instructions

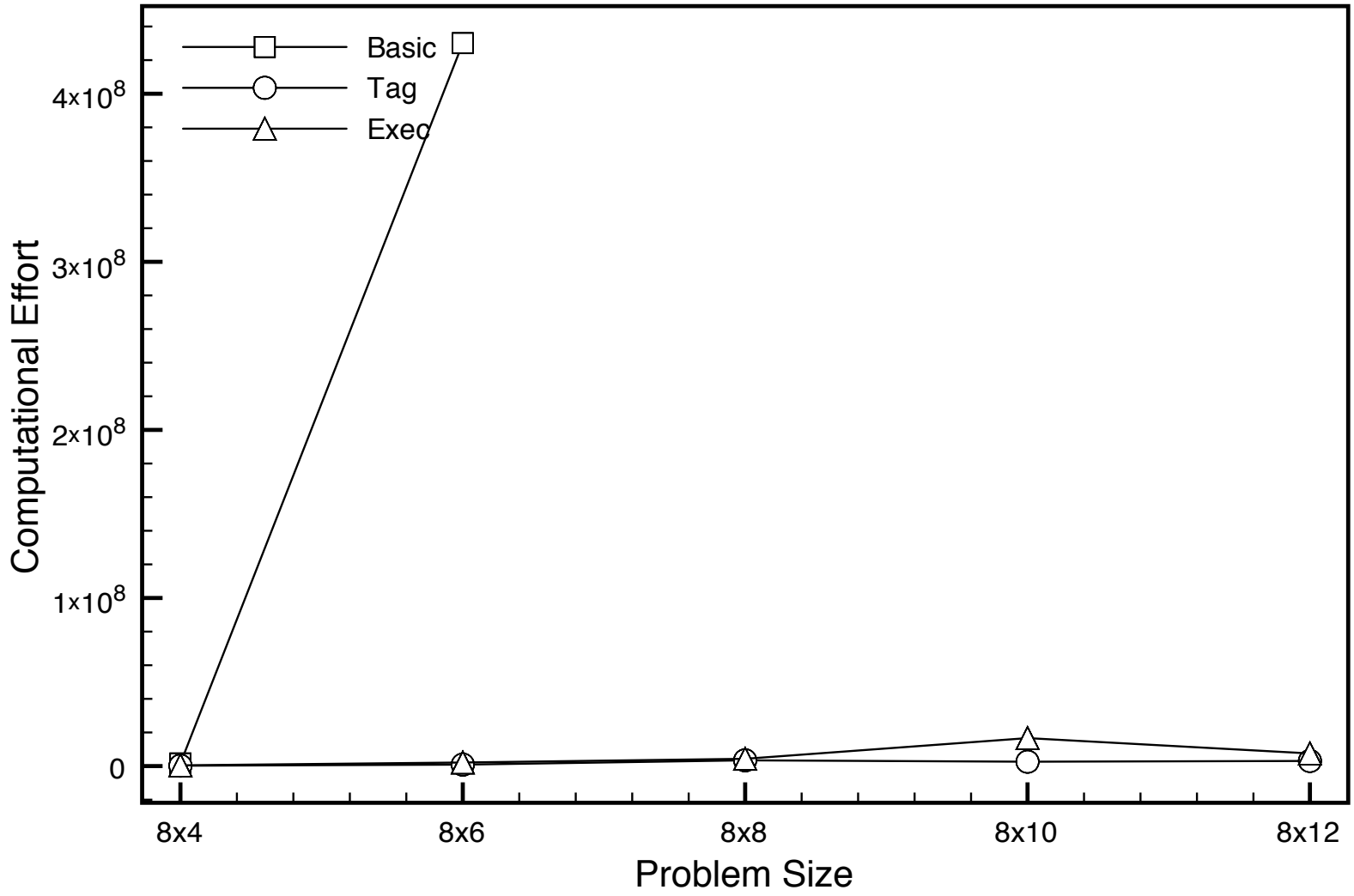
Condition	Instructions
Basic	if-dirty, if-obstacle, left, mop, v8a, frog, $\mathcal{R}_{v8}$
Tag	if-dirty, if-obstacle, left, mop, v8a, frog, $\mathcal{R}_{v8}$, tag.exec.[1000], tagged.[1000]
Exec	if-dirty, if-obstacle, left, mop, v8a, frog, $\mathcal{R}_{v8}$, exec.dup, exec.pop, exec.rot, exec.swap, exec.k, exec.s, exec.y

DSOAR Effort*

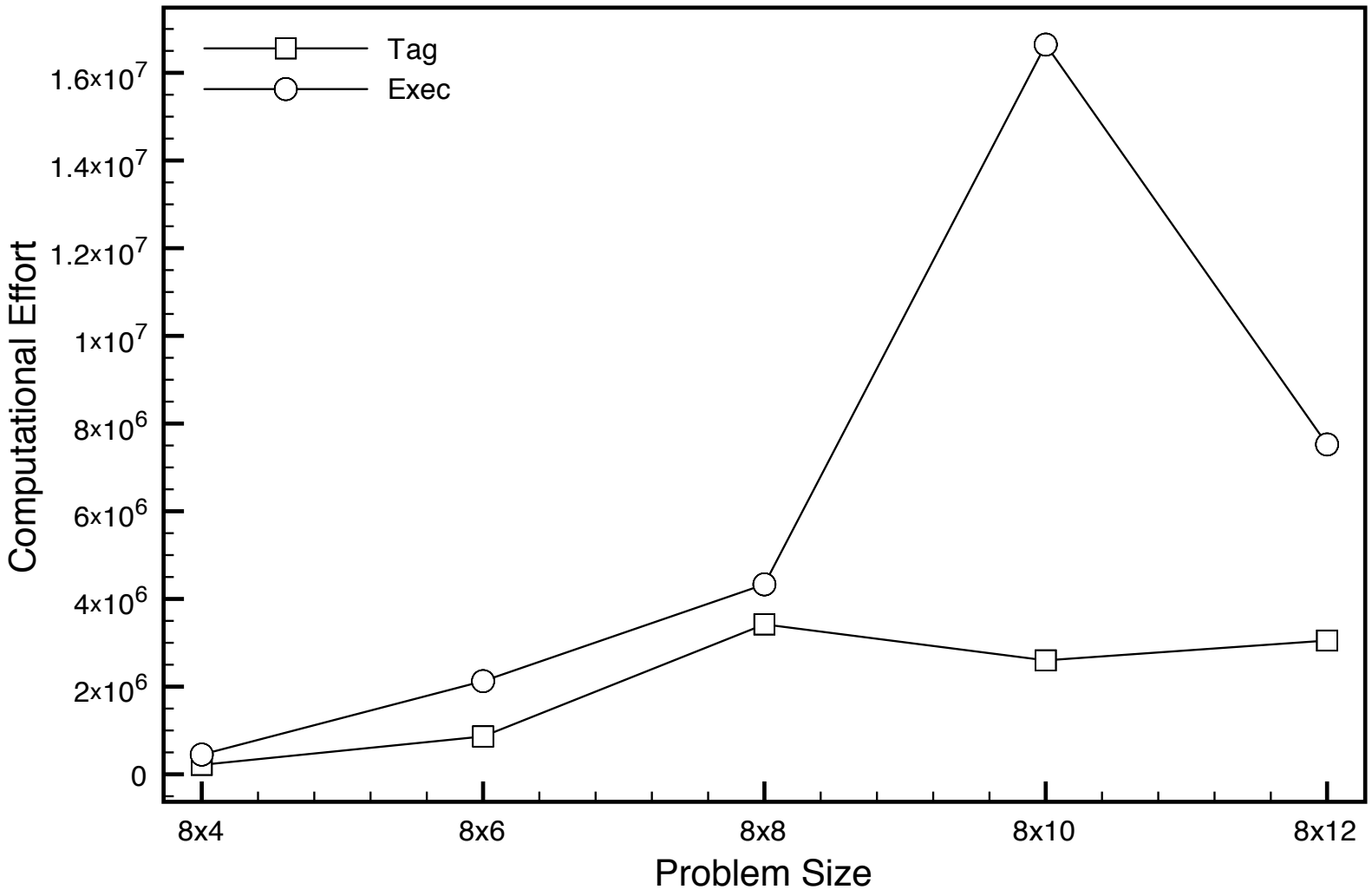
* with frog=noop bug



DSOAR Effort



DSOAR Effort



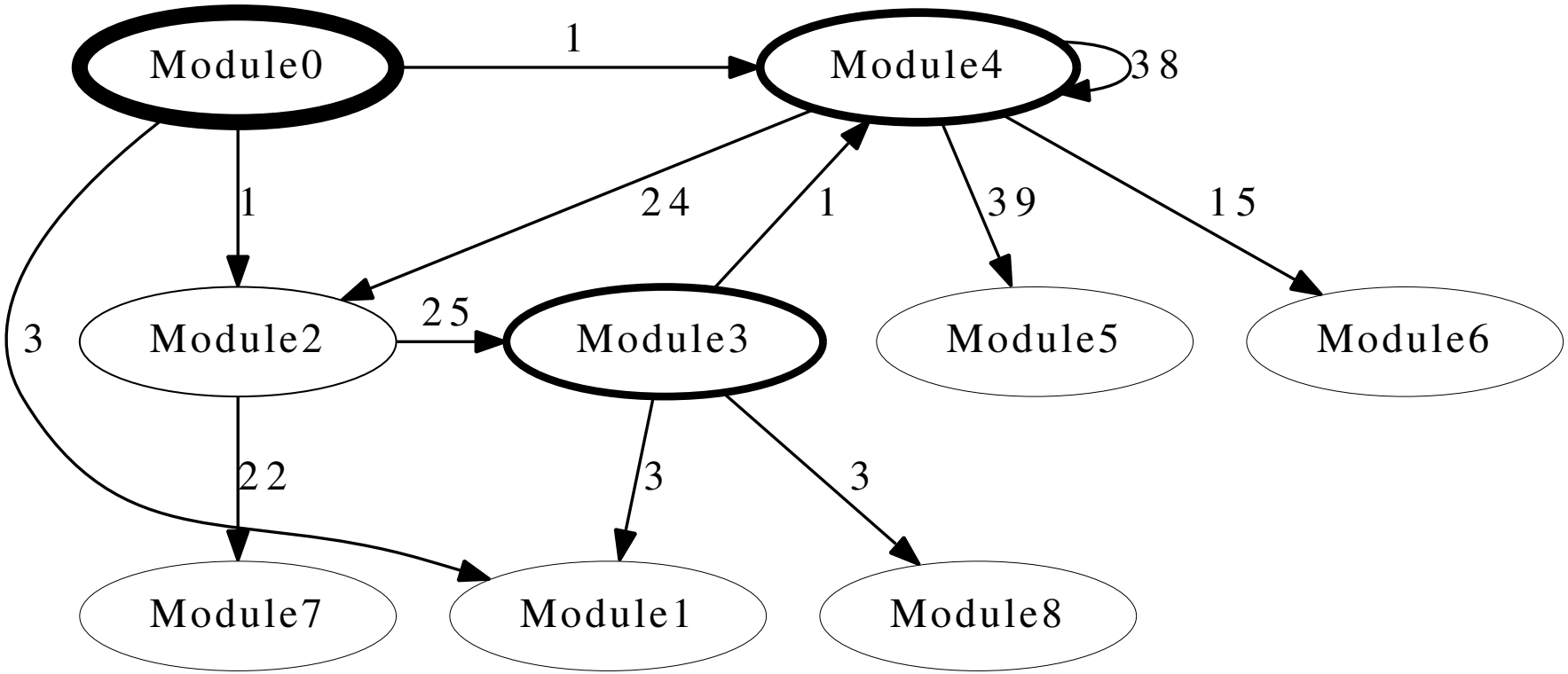
DSOAR Effort

	problem size				
	8x4	8x6	8x8	8x10	8x12
instr set					
basic	1584000	430083000	inf	inf	inf
tag	216000	864000	3420000	2599000	3051000
exec	450000	2125000	4332000	16644000	7524000

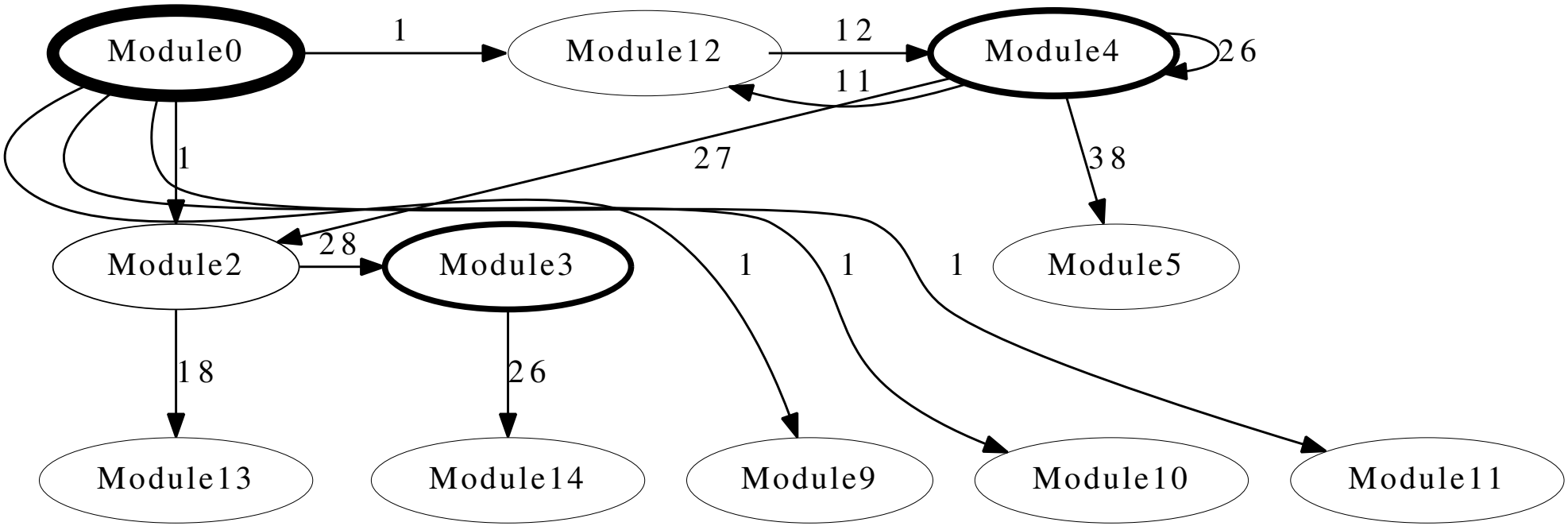
More data, source code, etc, at:

http://hampshire.edu/lspector/tags-gecco-2011

Evolved DSOAR Architecture (in one environment)



Evolved DSOAR Architecture (in another environment)



Tags in S-Expressions

- A simple form:
  (progn (tag-123 (+ a b)) tagged-034)
- Must do something about endless recursion
- Must do something about return values
- Must do something fancy to support modules with arguments, particularly arguments of multiple types.

Future Work

- Tags in s-expression-based GP
- Tag usage over evolutionary time
- No-pop tagging in PushGP
- Tags in autoconstructive evolution
- Applications, application, applications

Conclusions

- Execution stack manipulation supports the evolution of modular programs in many situations
- Tag-based modules are more effective in complex, non-uniform problem environments
- Tag-based modules may help to evolve complex software and solutions to unsolved problems in the future