Genetic Programming and Tag-Based Modularity

Lee Spector

based in part on work with

Brian Martin, Kyle Harrington & Thomas Helmuth

Cognitive Science, Hampshire College

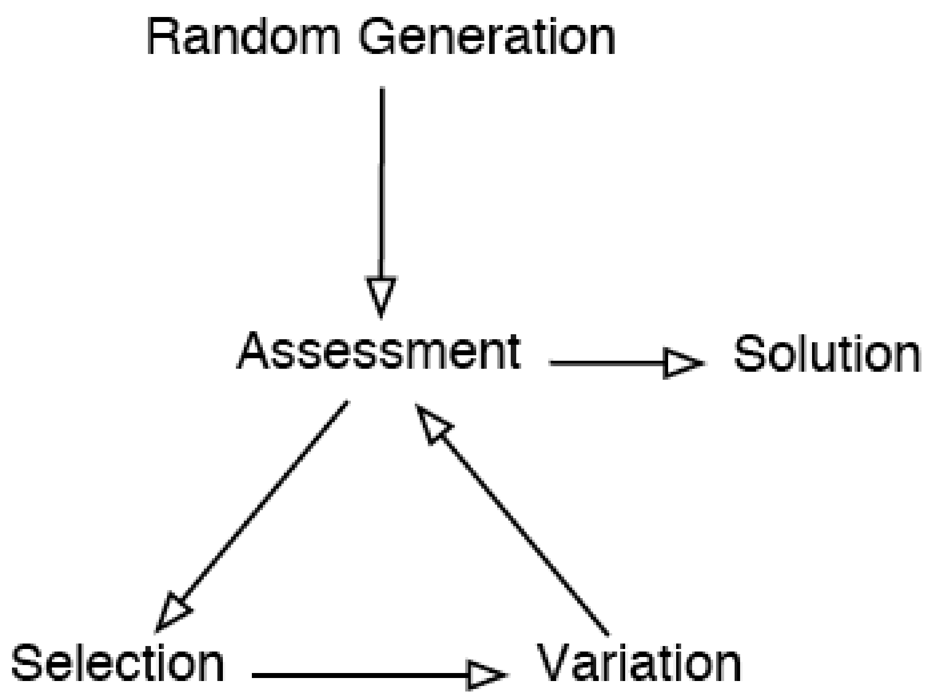
Computer Science, Brandeis University

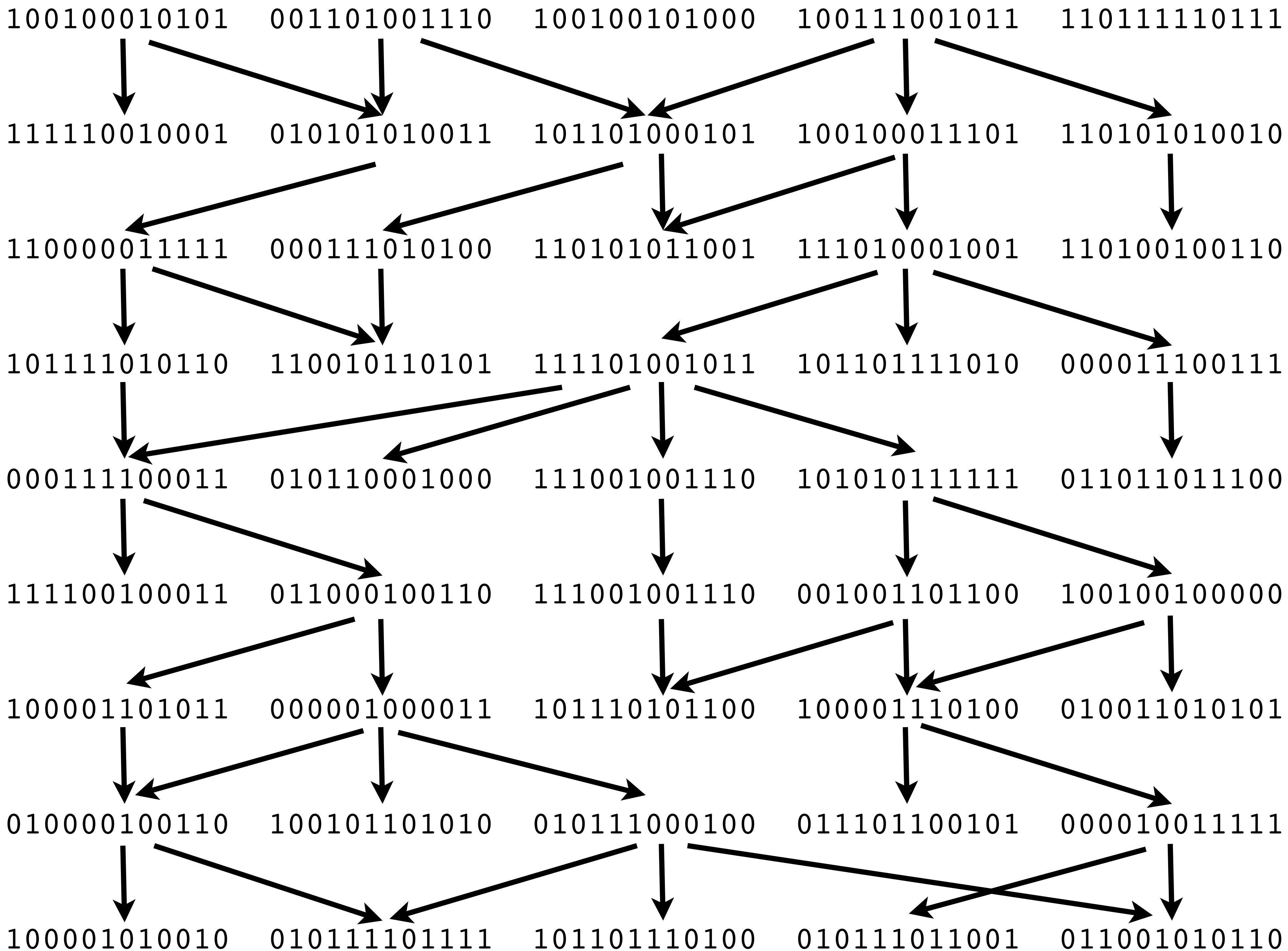
Computer Science, University of Massachusetts, Amherst

Outline

- Genetic Programming (GP)
- Push and PushGP
- Modularity in GP
- Tags and Tag-based modularity
- Results

Evolutionary Computation



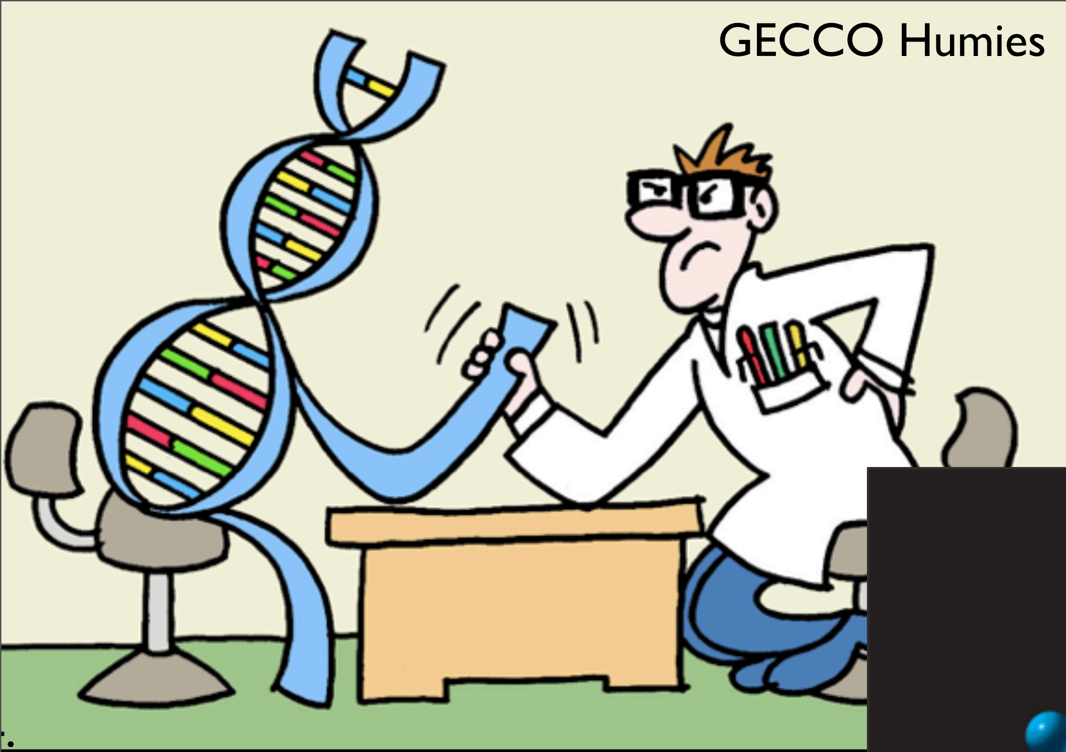

100100010101 001101001110 100100101000 100111001011 110111110111
111110010001 010101010011 101101000101 100100011101 110101010010
110000011111 000111010100 110101011001 111010001001 110100100110
101111010110 110010110101 111101001011 101101111010 000011100111
000111100011 010110001000 111001001110 101010111111 011011011100
111100100011 011000100110 111001001110 001001101100 100100100000
100001101011 000001000011 101110101100 100001110100 010011010101
010000100110 100101101010 010111000100 011101100101 000010011111
100001010010 010111101111 101101110100 010111011001 011001010110

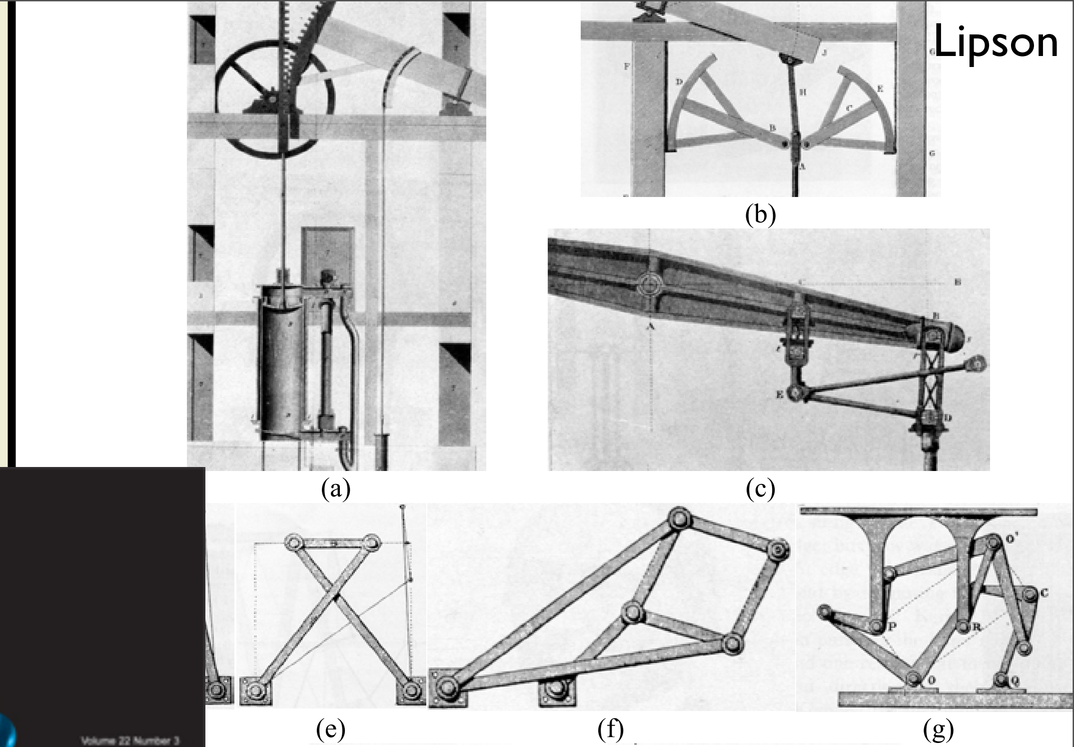
Traditional Genetic Algorithms

- Interesting dynamics
- Rarely solve interesting hard problems

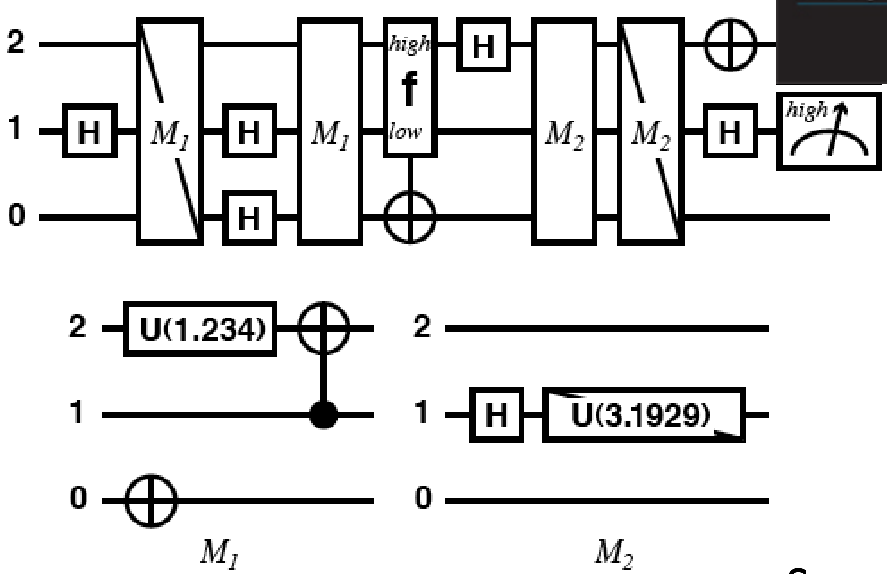
Genetic Programming

- Evolutionary computing to produce executable computer programs.
- Programs are tested by executing them.

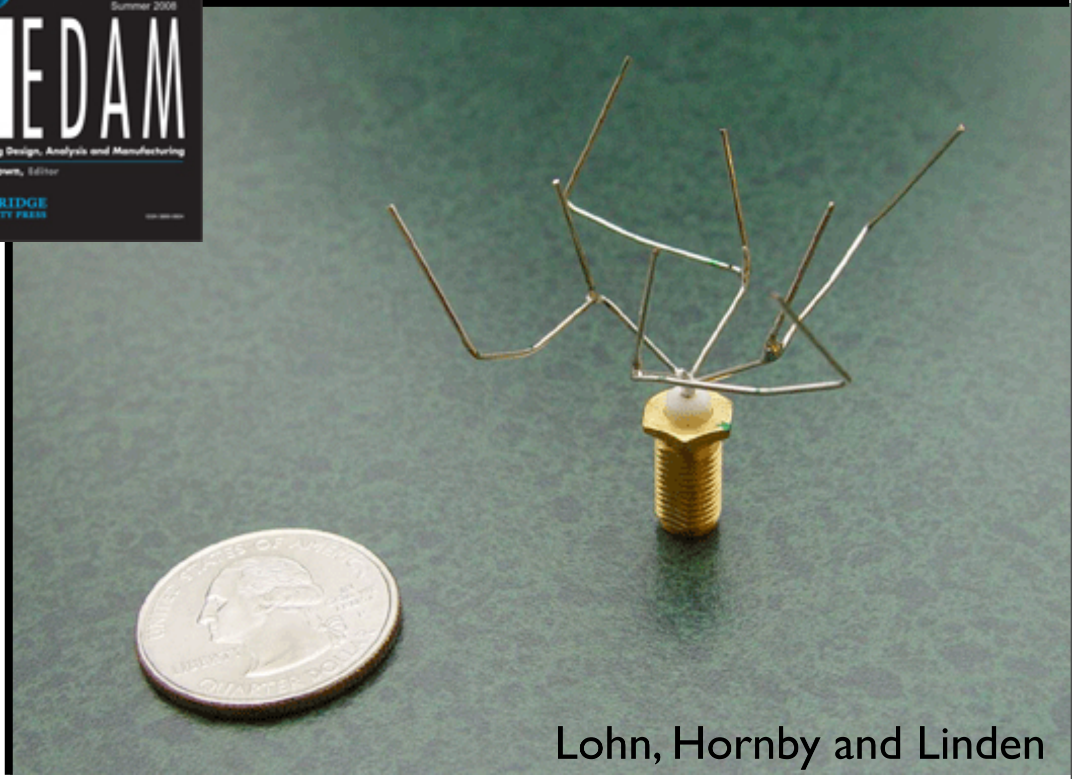

GECCO Humies


Lipson
(a)
(b)
(c)
(e)
(f)
(g)


Volume 22 Number 3
Summer 2008
AI EDAM
Artificial Intelligence for Engineering Design, Analysis and Manufacturing
David C. Brown, Editor
CAMBRIDGE
UNIVERSITY PRESS


2
1
0
H
M1
H
H
M1
high
f
low
H
M2
M2
H
high
U(1.234)
U(3.1929)
M1
M2

Spector


Lohn, Hornby and Linden

Evolution, the Designer

"Darwinian evolution is itself a designer worthy of significant respect, if not religious devotion." *Boston Globe* OpEd, Aug 29, 2005

WHAT WOULD DARWIN SAY? | LEE SPECTOR

The Boston Globe

And now, digital evolution

By Lee Spector | August 29, 2005

RECENT developments in computer science provide new perspective on "intelligent design," the view that life's complexity could only have arisen through the hand of an intelligent designer. These developments show that complex and useful designs can indeed emerge from random Darwinian processes.

Program Representations

- Lisp-style symbolic expressions (Koza, ...).
- Purely functional/lambda expressions (Walsh, Yu, ...).
- Linear sequences of machine/byte code (Nordin et al., ...).
- Artificial assembly-like languages (Ray, Adami, ...).
- Stack-based languages (Perkis, Spector, Stoffel, Tchernev, ...).
- Graph-structured programs (Teller, Globus, ...).
- Object hierarchies (Bruce, Abbott, Schmutter, Lucas, ...)
- Fuzzy rule systems (Tunstel, Jamshidi, ...)
- Logic programs (Osborn, Charif, Lamas, Dubossarsky, ...).
- Strings, grammar-mapped to arbitrary languages (O'Neill, Ryan, ...).

Mutating Lisp

```
(+ (* X Y)
   (+ 4 (- Z 23)))
```

(+ ***(* X Y)***
   (+ 4 (- Z 23)))

```
(+ (- (+ 2 2) Z)
   (+ 4 (- Z 23)))
```

Recombining Lisp

Parent 1: `(+ (* X Y)`
`(+ 4 (- Z 23)))`

Parent 2: `(- (* 17 (+ 2 X))`
`(* (- (* 2 Z) 1)`
`(+ 14 (/ Y X))))`

Child 1: `(+ (- (* 2 Z) 1)`
`(+ 4 (- Z 23)))`

Child 2: `(- (* 17 (+ 2 X))`
`(* (* X Y)`
`(+ 14 (/ Y X))))`

Symbolic Regression

Given a set of data points, evolve a program that produces *y* from *x*.

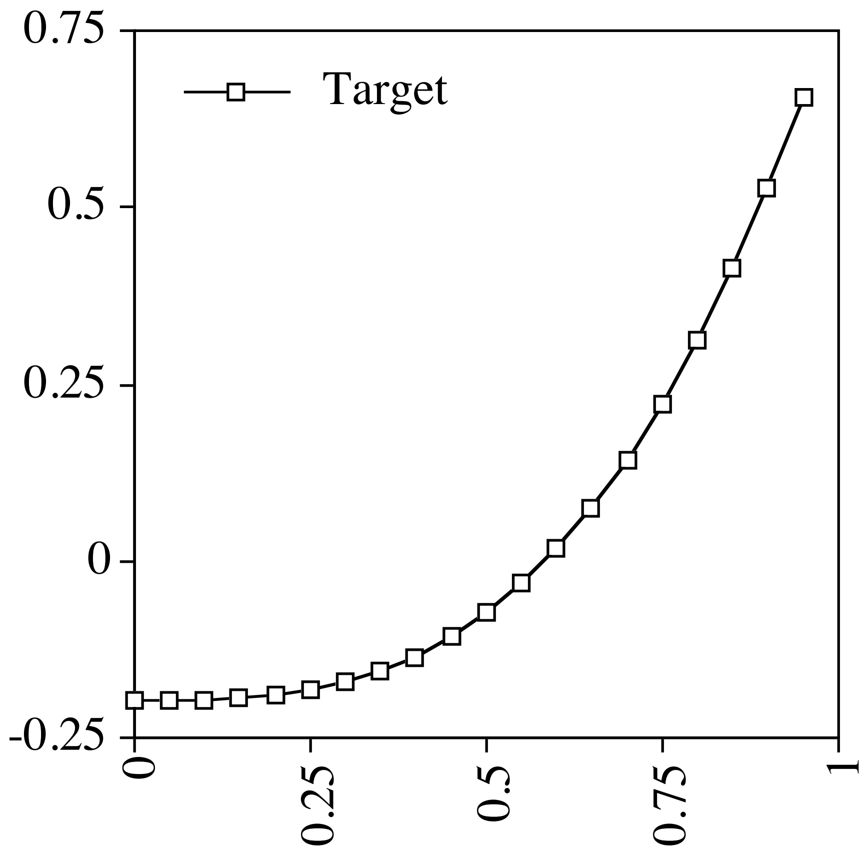
Primordial ooze: +, -, *, %, *x*, 0.1

Fitness = error (smaller is better)

GP Parameters

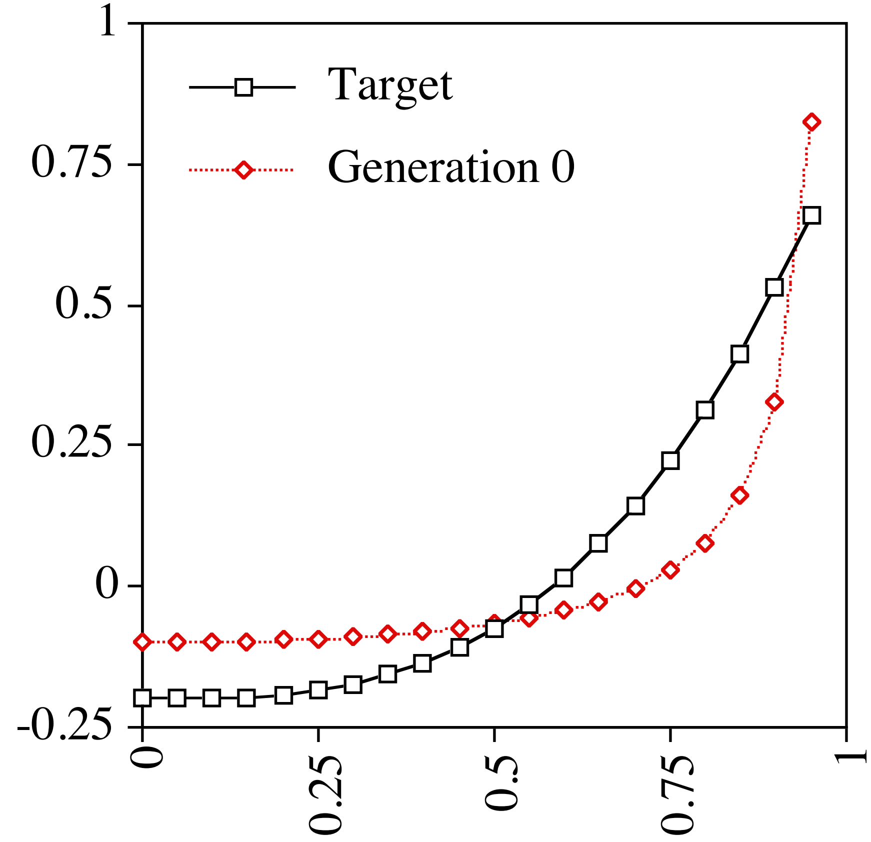
Maximum number of Generations: 51
Size of Population: 1000
Maximum depth of new individuals: 6
Maximum depth of new subtrees for mutants: 4
Maximum depth of individuals after crossover: 17
Fitness-proportionate reproduction fraction: 0.1
Crossover at any point fraction: 0.3
Crossover at function points fraction: 0.5
Selection method: FITNESS-PROPORTIONATE
Generation method: RAMPED-HALF-AND-HALF
Randomizer seed: 1.2

Evolving $y = x^3 - 0.2$



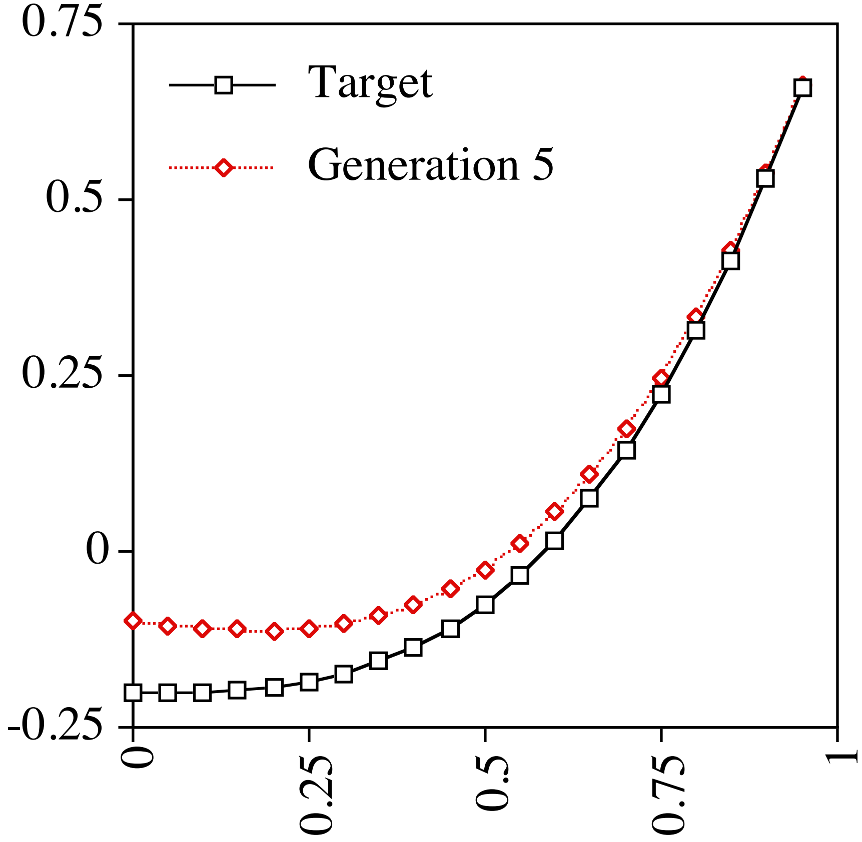
Best Program, Gen 0

```
(- (% (* 0.1
         (* X X))
      (- (% 0.1 0.1)
         (* X X)))
   0.1)
```



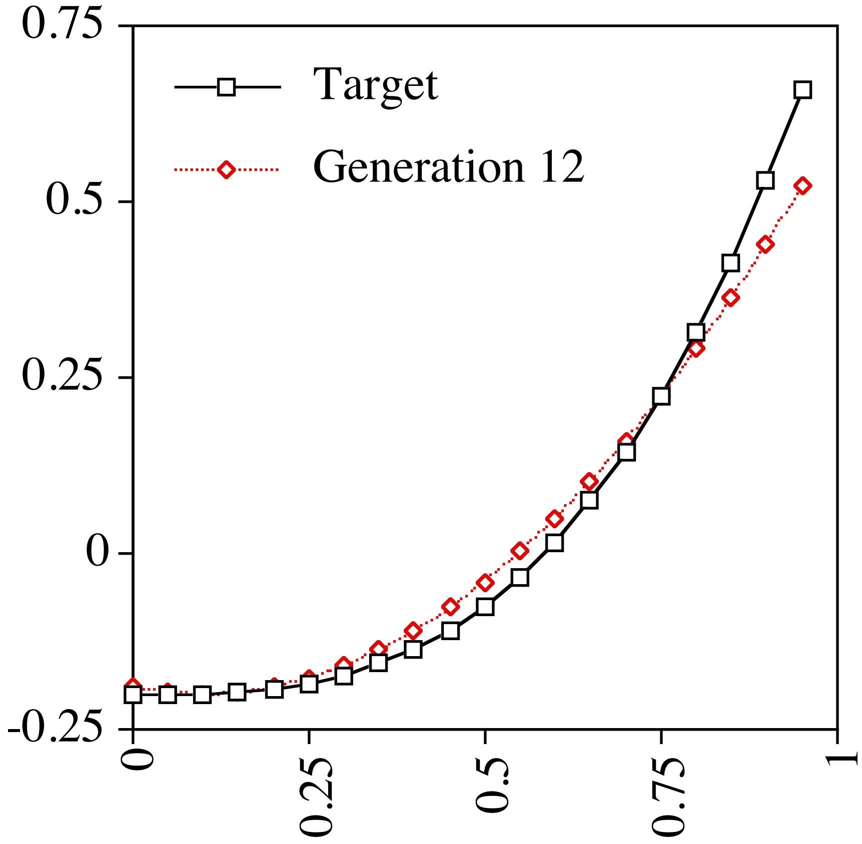
Best Program, Gen 5

```
(- (* (* (% X 0.1)
         (* 0.1 X))
      (- X
         (% 0.1 X)))
   0.1)
```



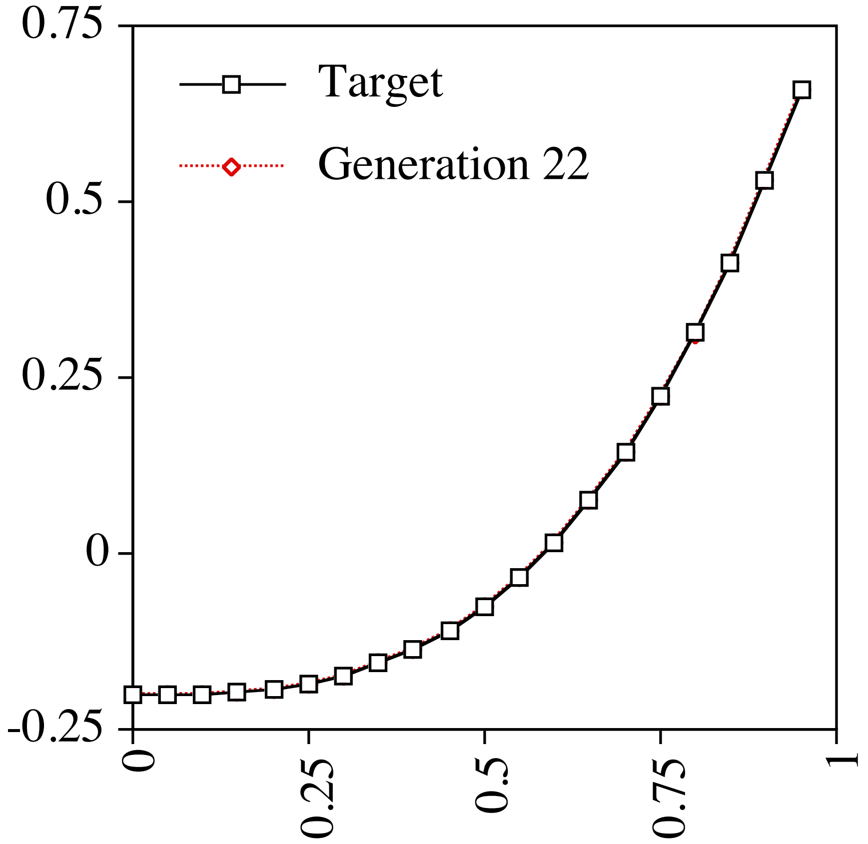
Best Program, Gen 12

```
(+ (- (- 0.1
         (- 0.1
            (- (* X X)
               (+ 0.1
                  (- 0.1
                     (* 0.1
                        0.1))))))
      (* X
         (* (% 0.1
               (% (* (* (- 0.1 0.1)
                        (+ X
                           (- 0.1 0.1)))
                     X)
                  (+ X (+ (- X 0.1)
                          (* X X)))))
            (+ 0.1 (+ 0.1 X)))))
   (* X X))
```



Best Program, Gen 22

```
(- (- (* X (* X X))
      0.1)
   0.1)
```



Genetic Programming for Finite Algebras

Lee Spector
Cognitive Science
Hampshire College
Amherst, MA 01002
lspector@hampshire.edu

David M. Clark
Mathematics
SUNY New Paltz
New Paltz, NY 12561
clarkd@newpaltz.edu

Ian Lindsay
Hampshire College
Amherst, MA 01002
iml04@hampshire.edu

Bradford Barr
Hampshire College
Amherst, MA 01002
bradford.barr@gmail.com

Jon Klein
Hampshire College
Amherst, MA 01002
jk@artificial.com

Humies 2008

GOLD MEDAL

Everybody's Favorite Finite Algebra

$$\text{Boolean algebra, } \mathbf{B} := \langle \{0,1\}, \wedge, \vee, \neg \rangle$$

$$\begin{array}{c|cc} \wedge & 0 & 1 \\ \hline 0 & 0 & 0 \\ 1 & 0 & 1 \end{array} \qquad \begin{array}{c|cc} \vee & 0 & 1 \\ \hline 0 & 0 & 1 \\ 1 & 1 & 1 \end{array} \qquad \begin{array}{c|c} & \neg \\ \hline 0 & 1 \\ 1 & 0 \end{array}$$

Primal: every possible operation can be expressed by a term using only (and not even) $\wedge$, $\vee$, and $\neg$.

Bigger Finite Algebras

- Have applications in many areas of science, engineering, mathematics
- Can be *much* harder to analyze/understand
- Number of terms grows astronomically with size of underlying set

Goal

- Find terms that have certain special properties
- *Discriminator* terms, determine primality

$$t^A(x,y,z) = \begin{cases} x \text{ if } x \neq y \\ z \text{ if } x = y \end{cases}$$

- *Mal'cev*, *majority*, and *Pixley* terms
- For decades there was no way to produce these terms in general, short of exhaustive search
- Current best methods produce enormous terms

Algebras Explored

$\mathbf{A}_1$ $*$	0	1	2
0	2	1	2
1	1	0	0
2	0	0	1

$\mathbf{A}_2$ $*$	0	1	2
0	2	0	2
1	1	0	2
2	1	2	1

$\mathbf{A}_3$ $*$	0	1	2
0	1	0	1
1	1	2	0
2	0	0	0

$\mathbf{A}_4$ $*$	0	1	2
0	1	0	1
1	0	2	0
2	0	1	0

$\mathbf{A}_5$ $*$	0	1	2
0	1	0	2
1	1	2	0
2	0	1	0

$\mathbf{B}_1$ $*$	0	1	2	3
0	1	3	1	0
1	3	2	0	1
2	0	1	3	1
3	1	0	2	0

Results

- Discriminators for A_1, A_2, A_3, A_4, A_5
- Mal'cev and majority terms for B_1
- Example Mal'cev term for B_1:

$$((((((((x*(y*x))*x)*z)*(z*x))*((x*(z*(x*(z*y)))))*z))*z)$$
$$*z)*(z*((((x*(((z*z)*x)*(z*x)))*x)*y)*(((y*(z*(z*y))))*$$
$$(((y*y)*x)*z))*(x*(((z*z)*x)*(z*(x*(z*y))))))))))$$

Significance, Time

	Uninformed Search Expected Time (Trials)
3 element algebras	
Mal'cev	5 seconds ($3^{15} \approx 10^7$)
Pixley/majority	1 hour ($3^{21} \approx 10^{10}$)
discriminator	1 month ($3^{27} \approx 10^{13}$)
4 element algebras	
Mal'cev	10^3 years ($4^{28} \approx 10^{17}$)
Pixley/majority	10^{10} years ($4^{40} \approx 10^{24}$)
discriminator	10^{24} years ($4^{64} \approx 10^{38}$)

Significance, Time

	Uninformed Search Expected Time (Trials)	GP Time
3 element algebras		
Mal'cev	5 seconds ($3^{15} \approx 10^7$)	1 minute
Pixley/majority	1 hour ($3^{21} \approx 10^{10}$)	3 minutes
discriminator	1 month ($3^{27} \approx 10^{13}$)	5 minutes
4 element algebras		
Mal'cev	10^3 years ($4^{28} \approx 10^{17}$)	30 minutes
Pixley/majority	10^{10} years ($4^{40} \approx 10^{24}$)	2 hours
discriminator	10^{24} years ($4^{64} \approx 10^{38}$)	?

Significance, Size

Term Type	Primality Theorem
Mal'cev	$10,060,219$
Majority	$6,847,499$
Pixley	$1,257,556,499$
Discriminator	$12,575,109$

(for A_I)

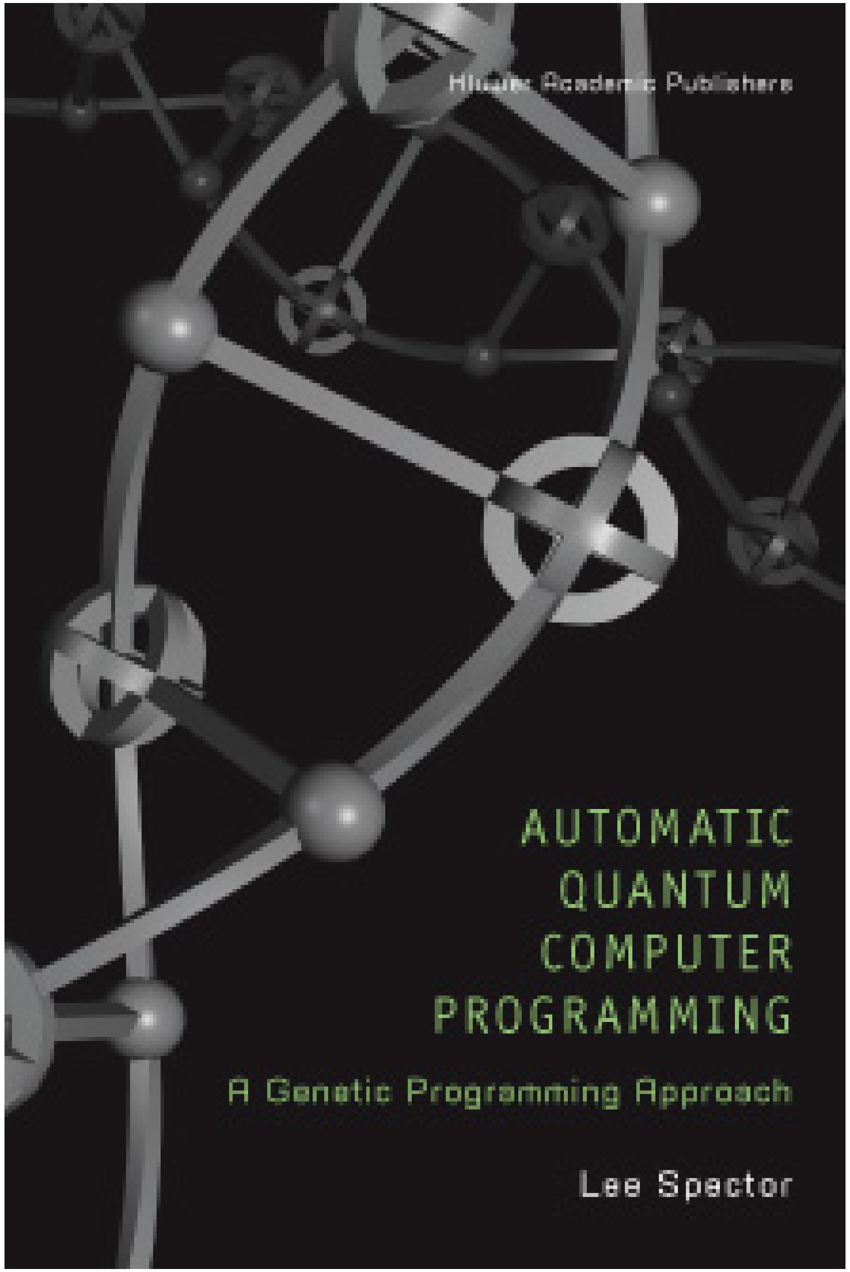
Significance, Size

Term Type	Primality Theorem	GP
Mal'cev	$10,060,219$	12
Majority	$6,847,499$	49
Pixley	$1,257,556,499$	59
Discriminator	$12,575,109$	39

(for A_I)

Human Competitive?

- Rather: human-**WHOMPING!**
- *Outperforms* humans *and all other known methods* on significant problems, providing benefits of *several orders of magnitude* with respect to search speed and result size
- Because there were no prior methods for generating practical terms in practical amounts of time, GP has provided the first solution to a previously open problem in the field



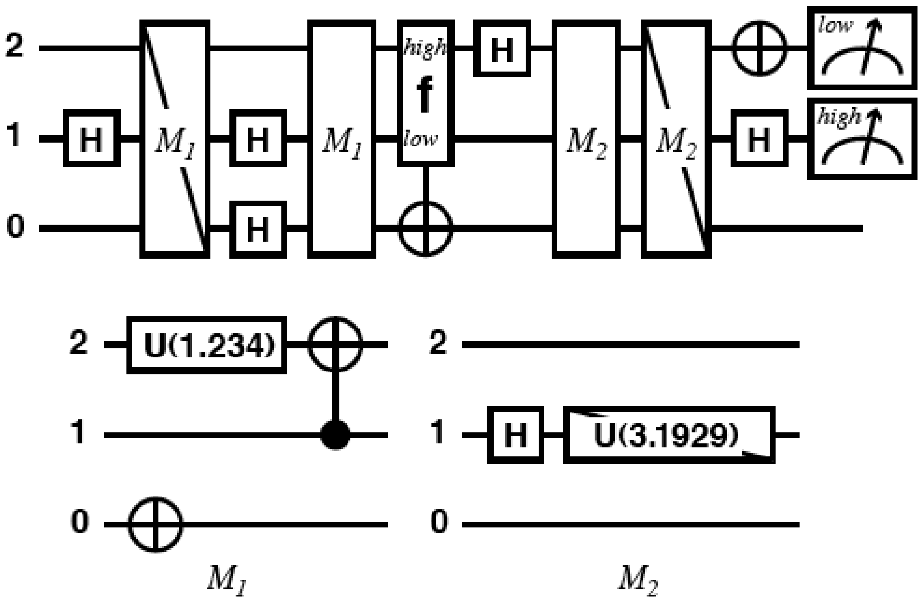


Figure 8.7. A gate array diagram for an evolved version of Grover's database search algorithm for a 4-item database. The full gate array is shown at the top, with M_1 and M_2 standing for the smaller gate arrays shown at the bottom. A diagonal line through a gate symbol indicates that the matrix for the gate is transposed. The "f" gate is the oracle.

Humies 2004 GOLD MEDAL

Expressive Languages

- Strongly typed genetic programming
- Automatically defined functions
- Automatically defined macros
- Architecture-altering operations
- Developmental genetic programming

Expressive Languages

- Strongly typed genetic programming
- Automatically defined functions
- Automatically defined macros
- Architecture-altering operations
- Developmental genetic programming
- Push provides all of the above and more, all without any mechanisms beyond the stack-based execution architecture

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

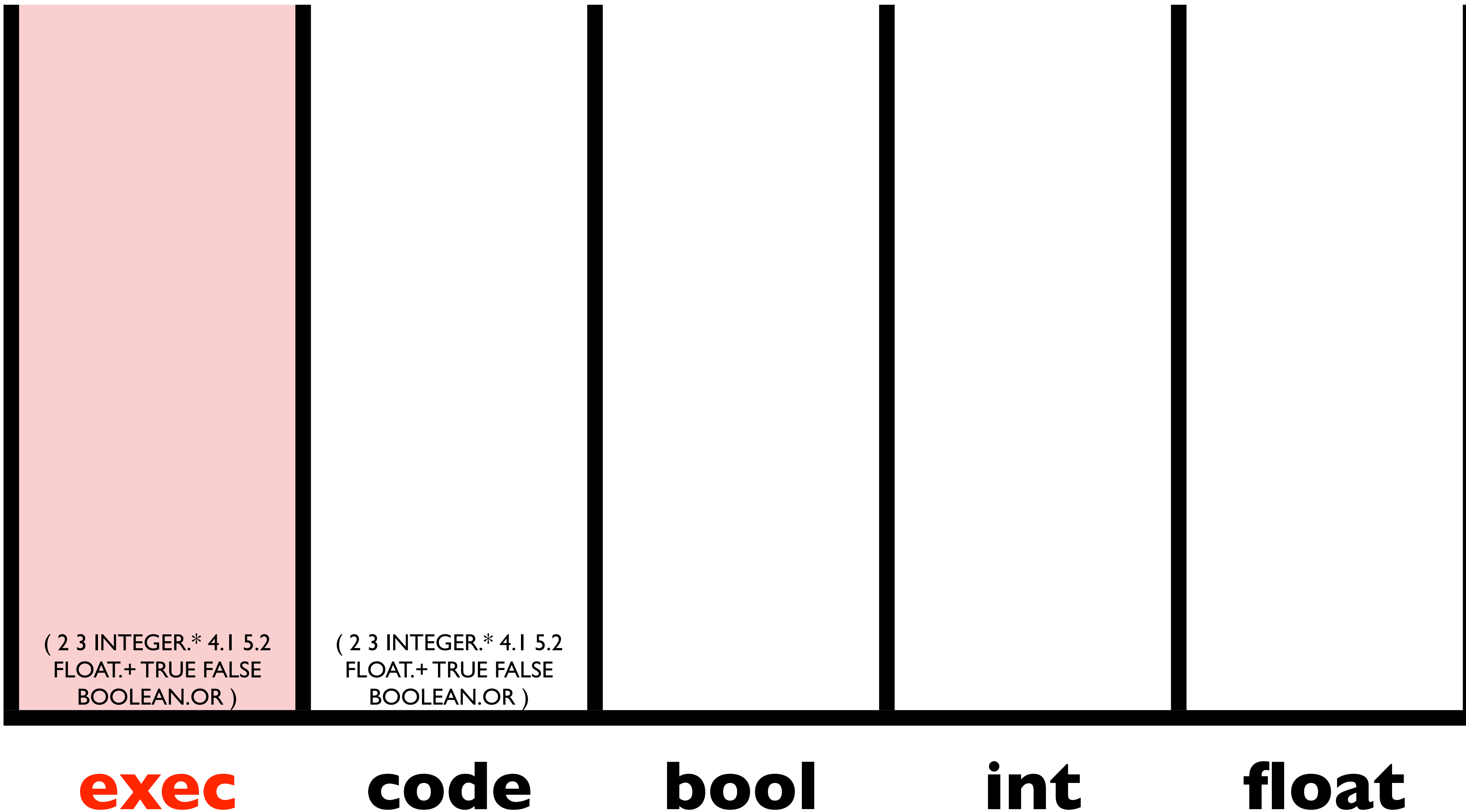
- Stack-based postfix language with one stack per type
- Types include: integer, float, Boolean, name, code, exec, vector, matrix, quantum gate, [add more as needed]
- Missing argument? NOOP
- Trivial syntax:
  program → instruction | literal | (program*)

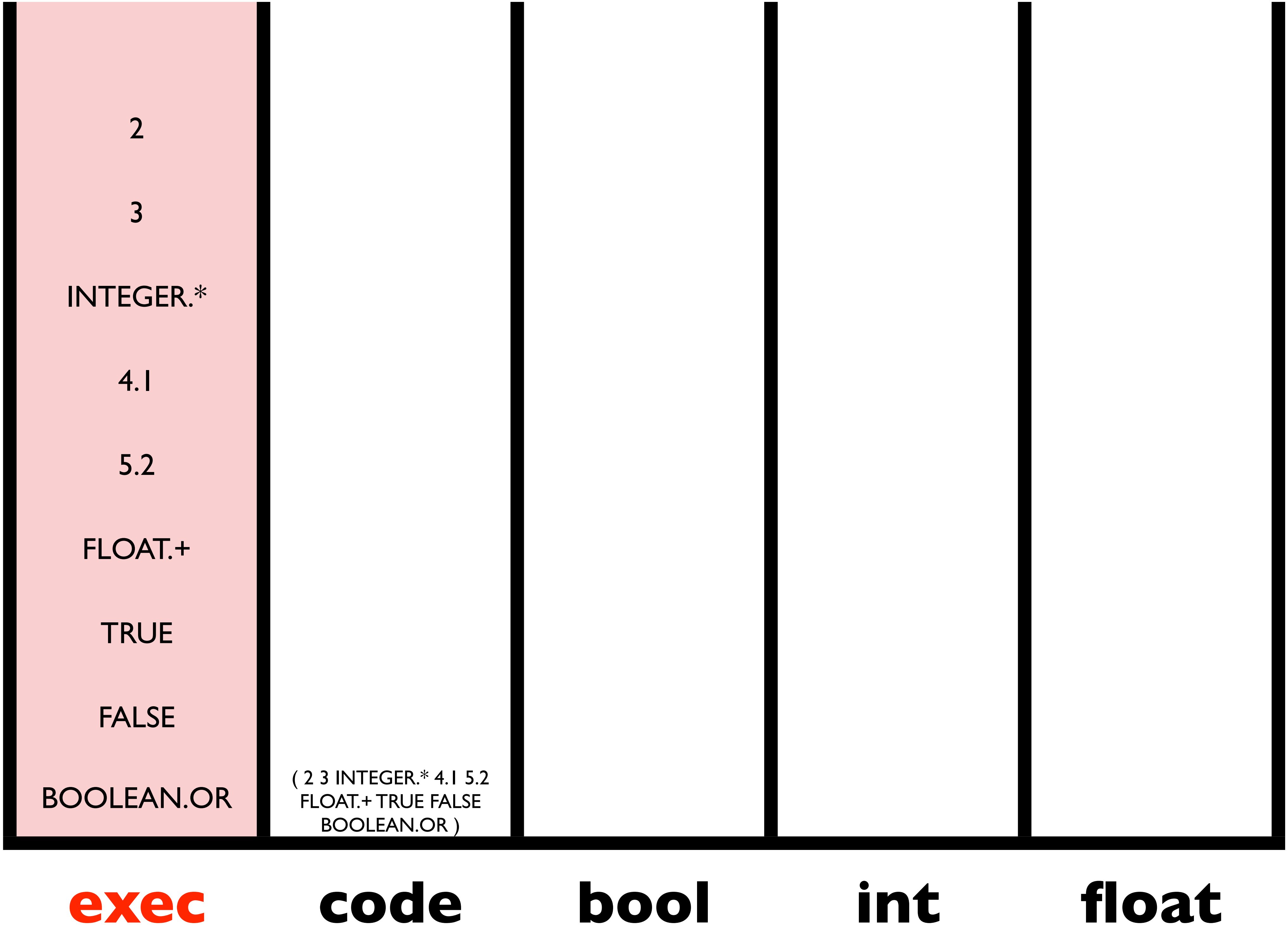
Push(3) Semantics

- To execute program P:

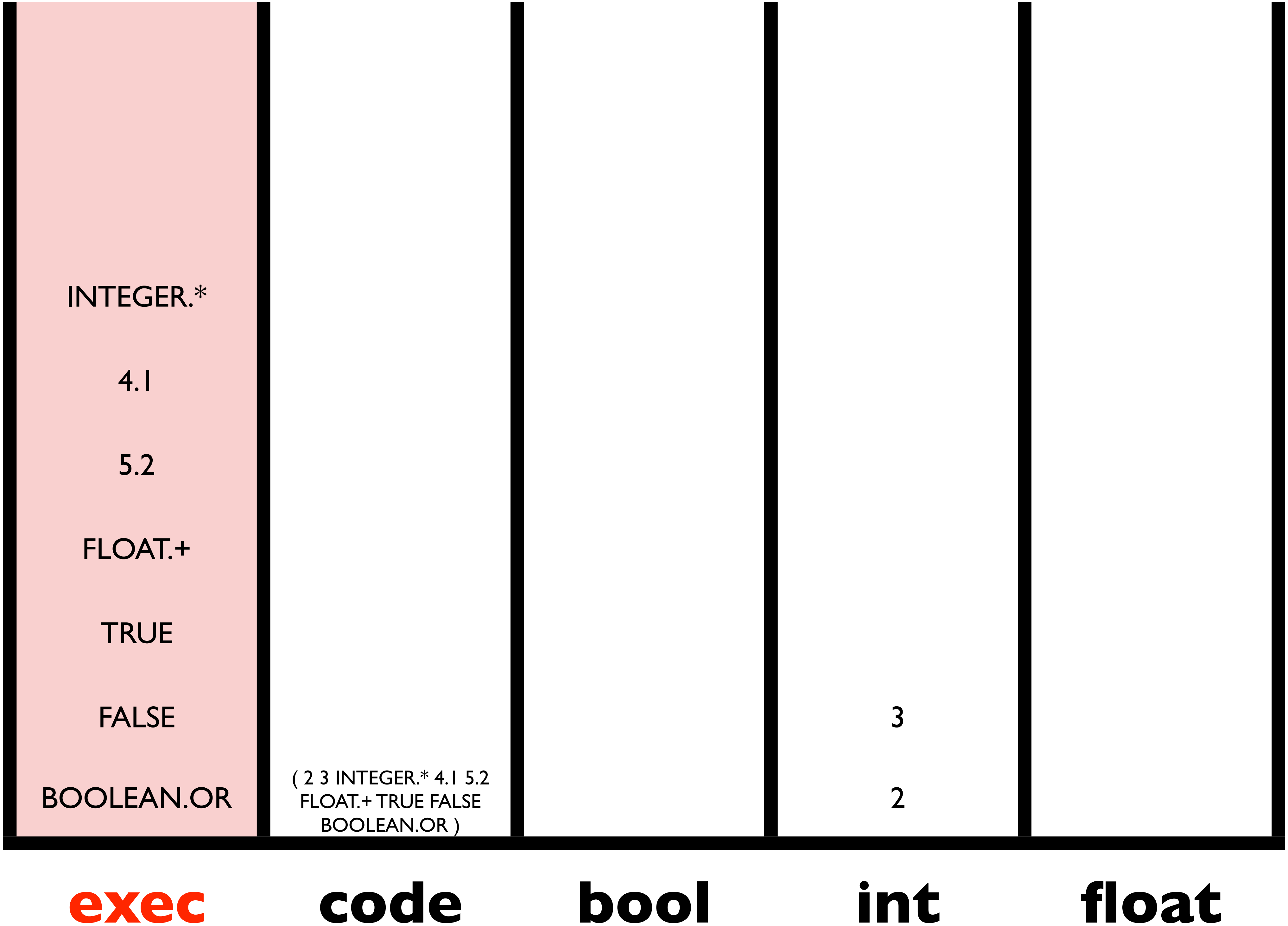
1. Push P onto the `EXEC` stack.
2. While the `EXEC` stack is not empty, pop and process the top element of the `EXEC` stack, E:
   (a) If E is an instruction: execute E (accessing whatever stacks are required).
   (b) If E is a literal: push E onto the appropriate stack.
   (c) If E is a list: push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

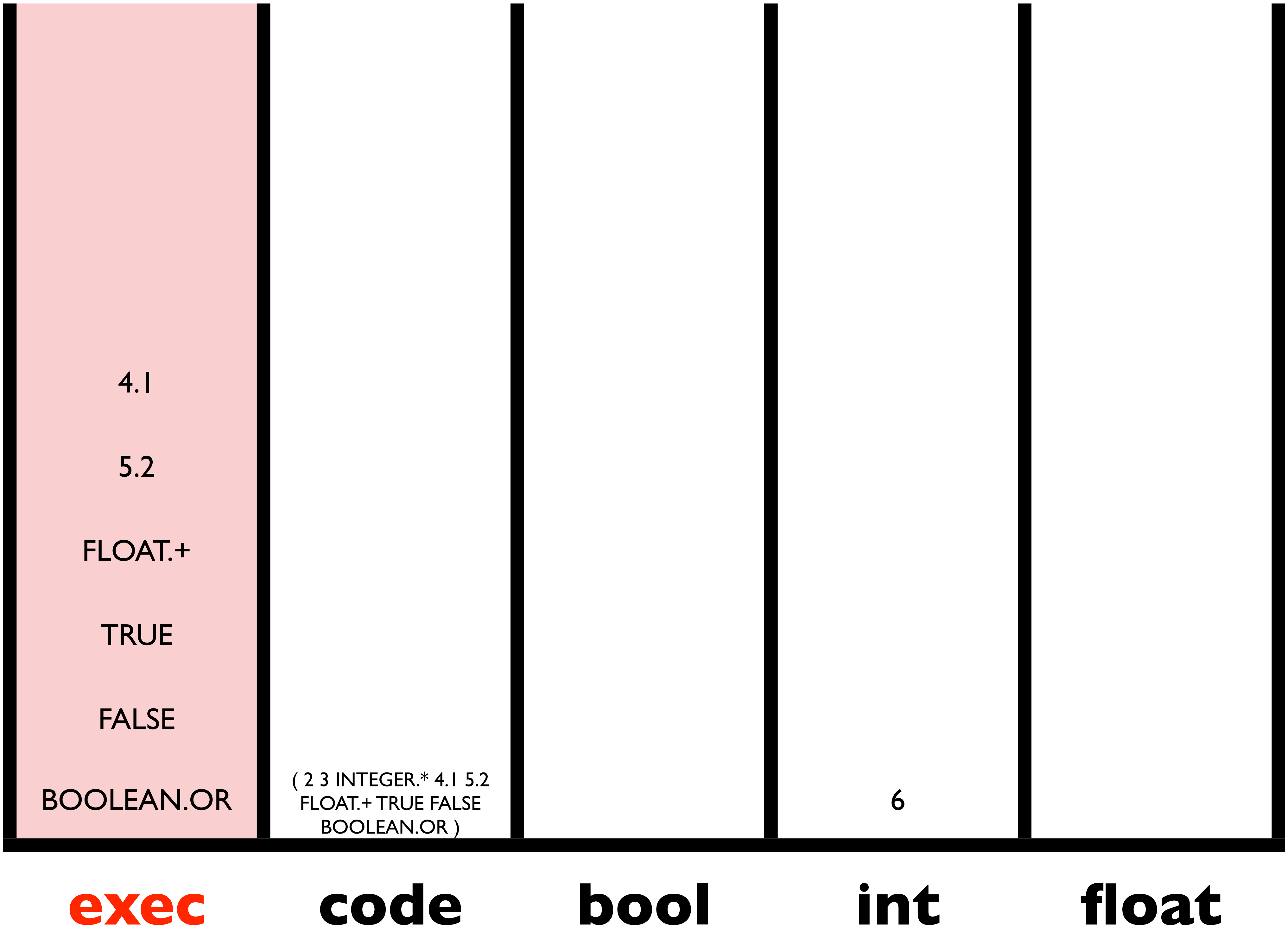
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)

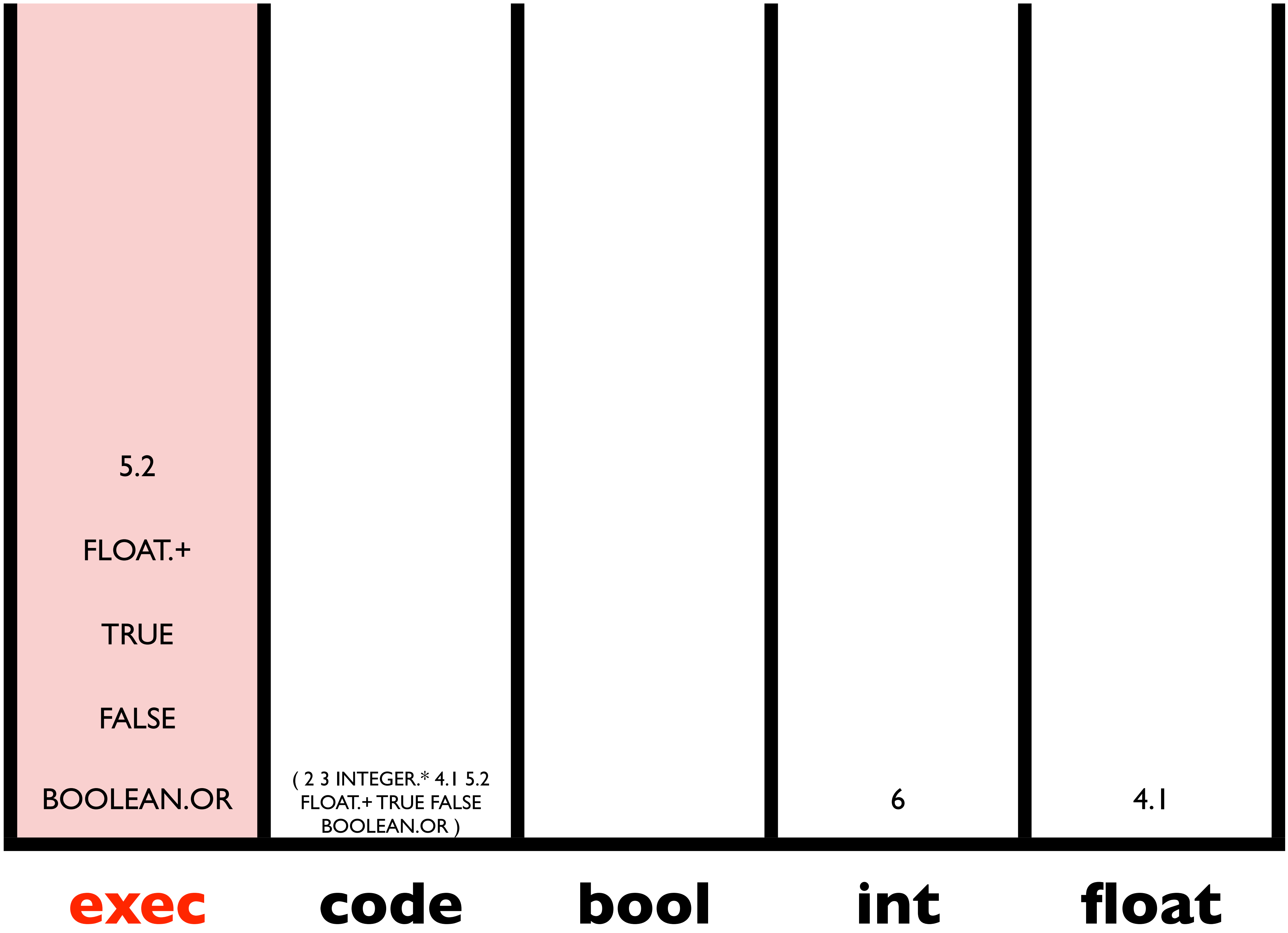



2
3
INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
exec
code
bool
int
float

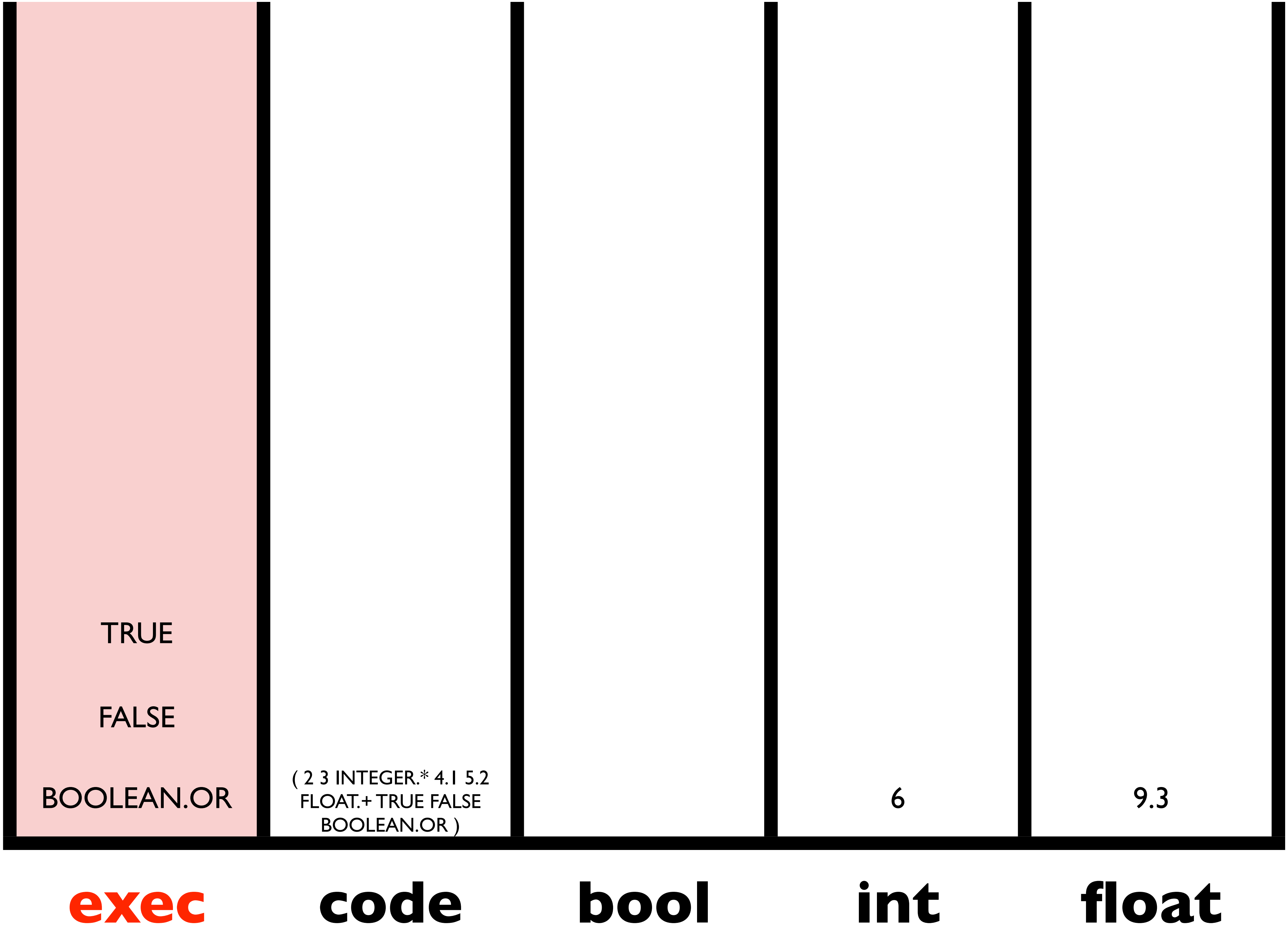
exec	code	bool	int	float
3				
INTEGER.*				
4.1				
5.2				
FLOAT.+				
TRUE				
FALSE				
BOOLEAN.OR	(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)		2	

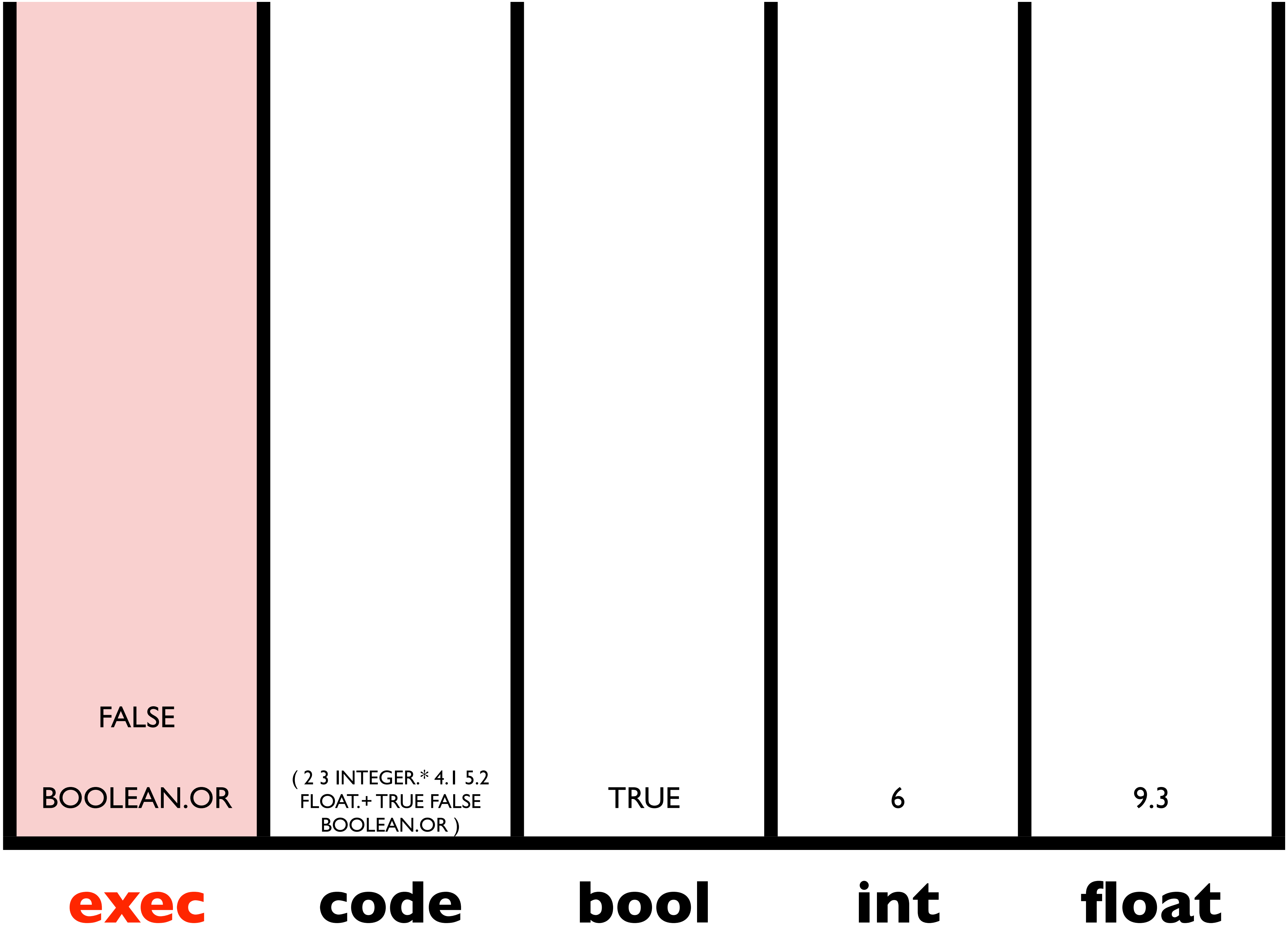

INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
3
2
exec
code
bool
int
float

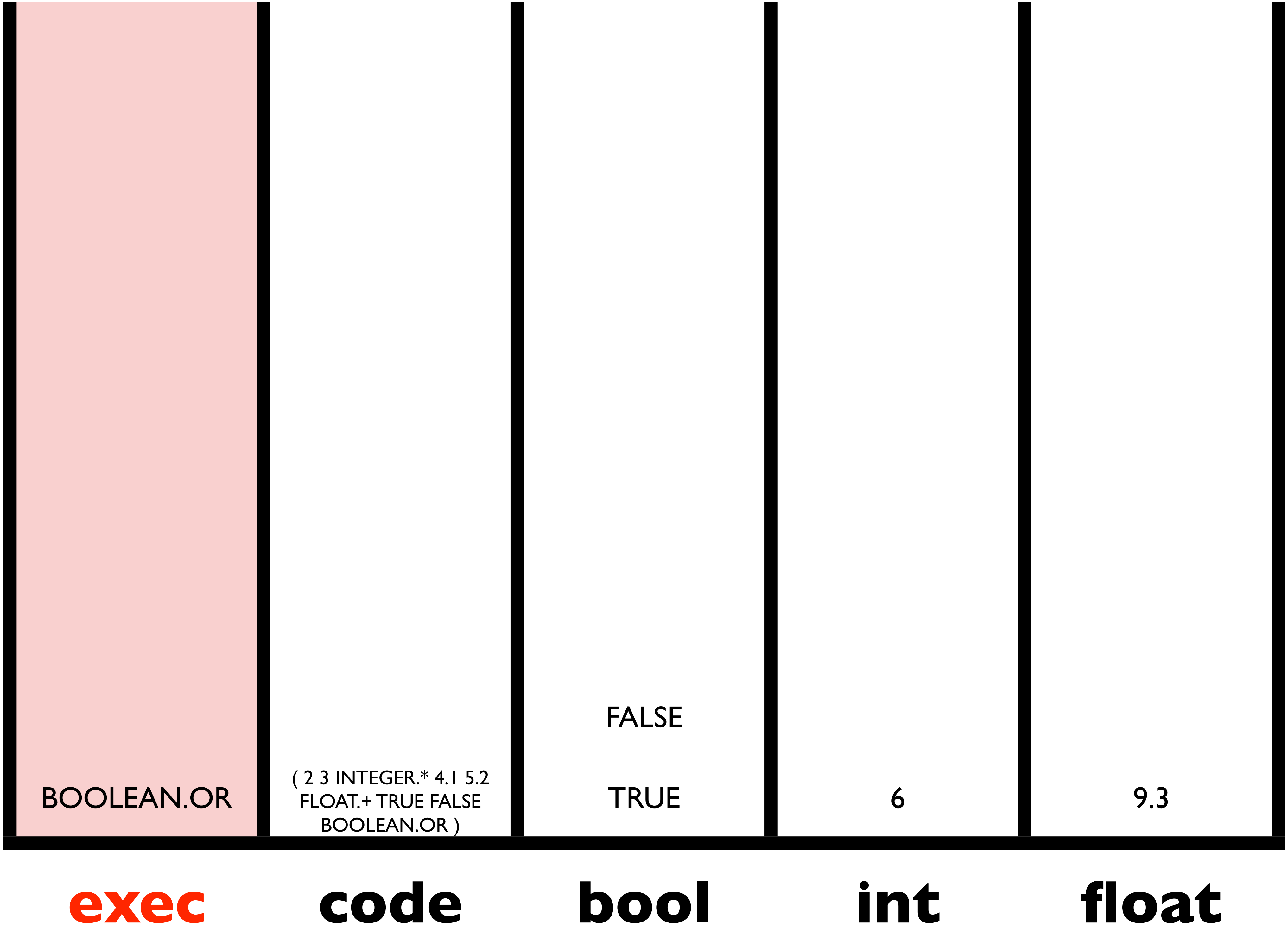

4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
6
exec
code
bool
int
float

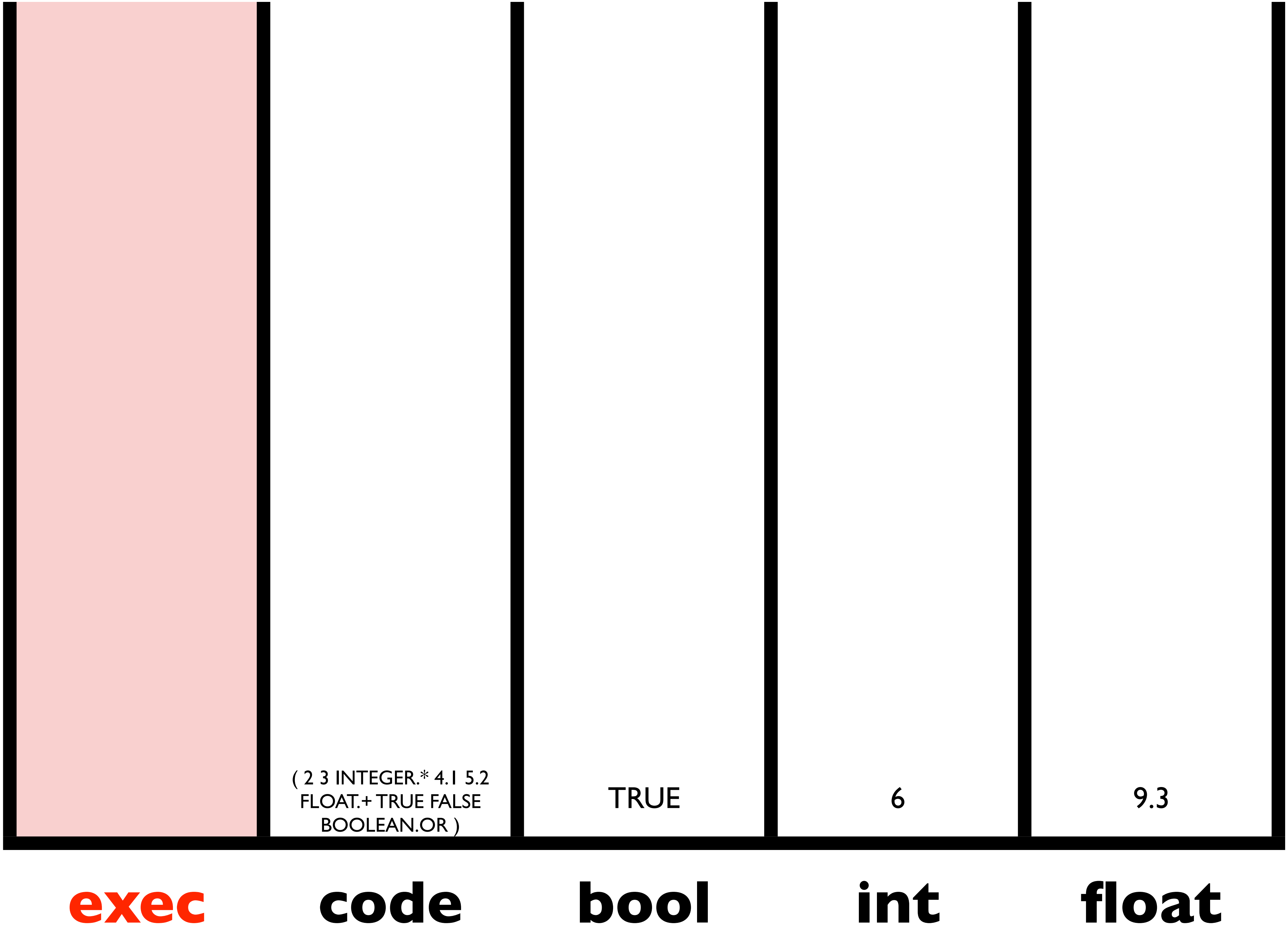

5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
6
4.1
exec
code
bool
int
float

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


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

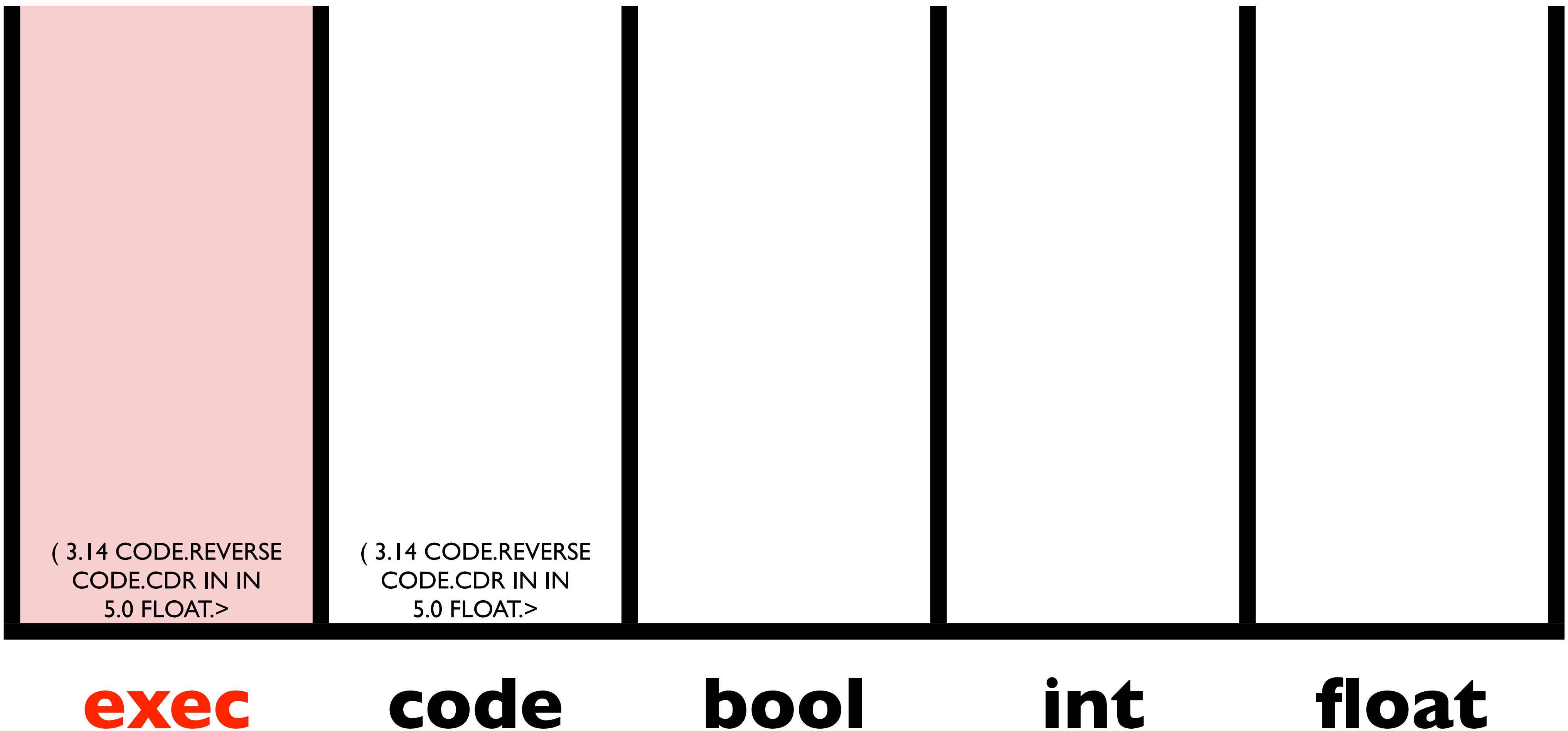
Same Results

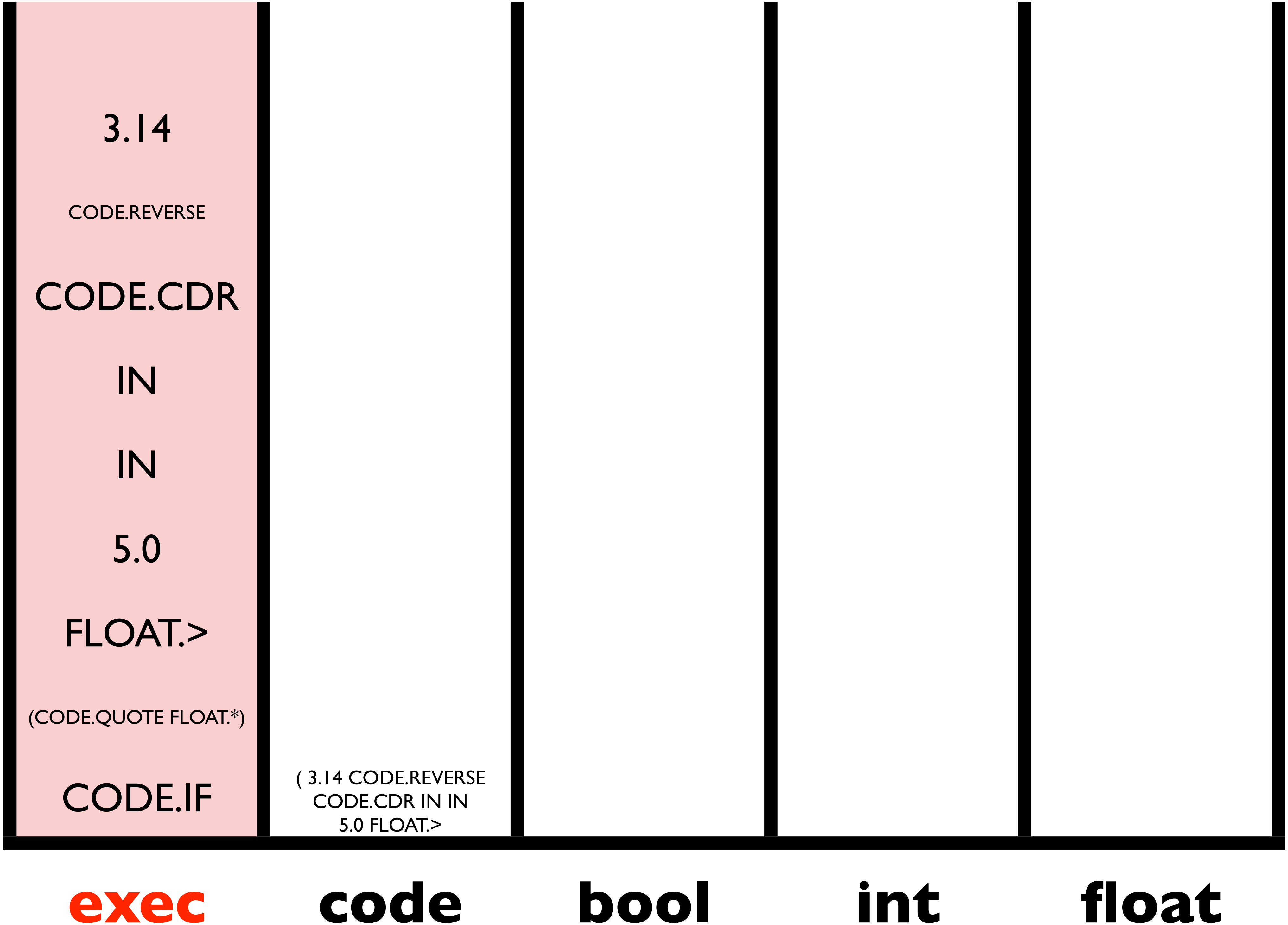
```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+
TRUE FALSE BOOLEAN.OR )
```

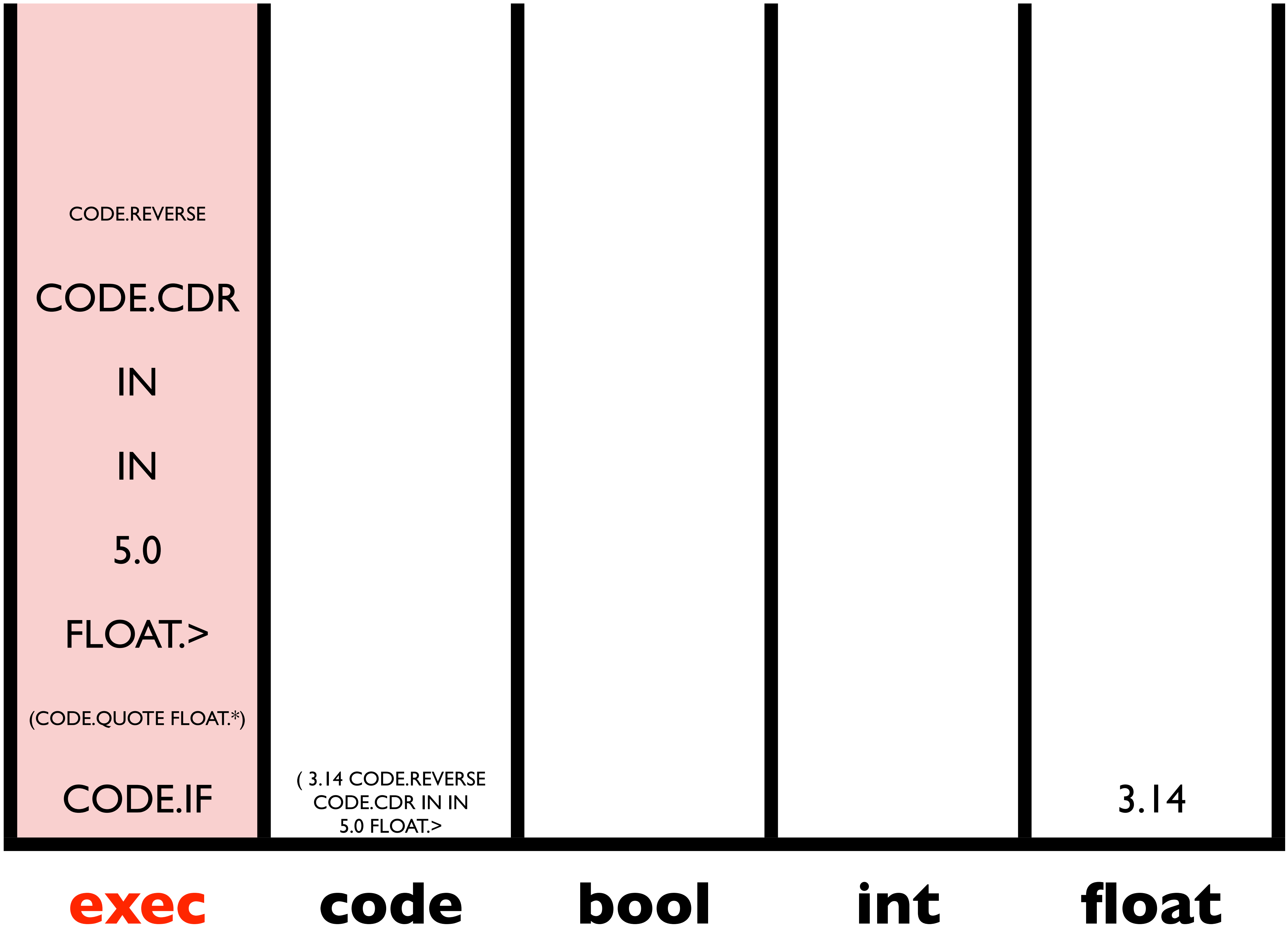
```
( 2 BOOLEAN.AND 4.1 TRUE INTEGER./ FALSE
3 5.2 BOOLEAN.OR INTEGER.* FLOAT.+ )
```

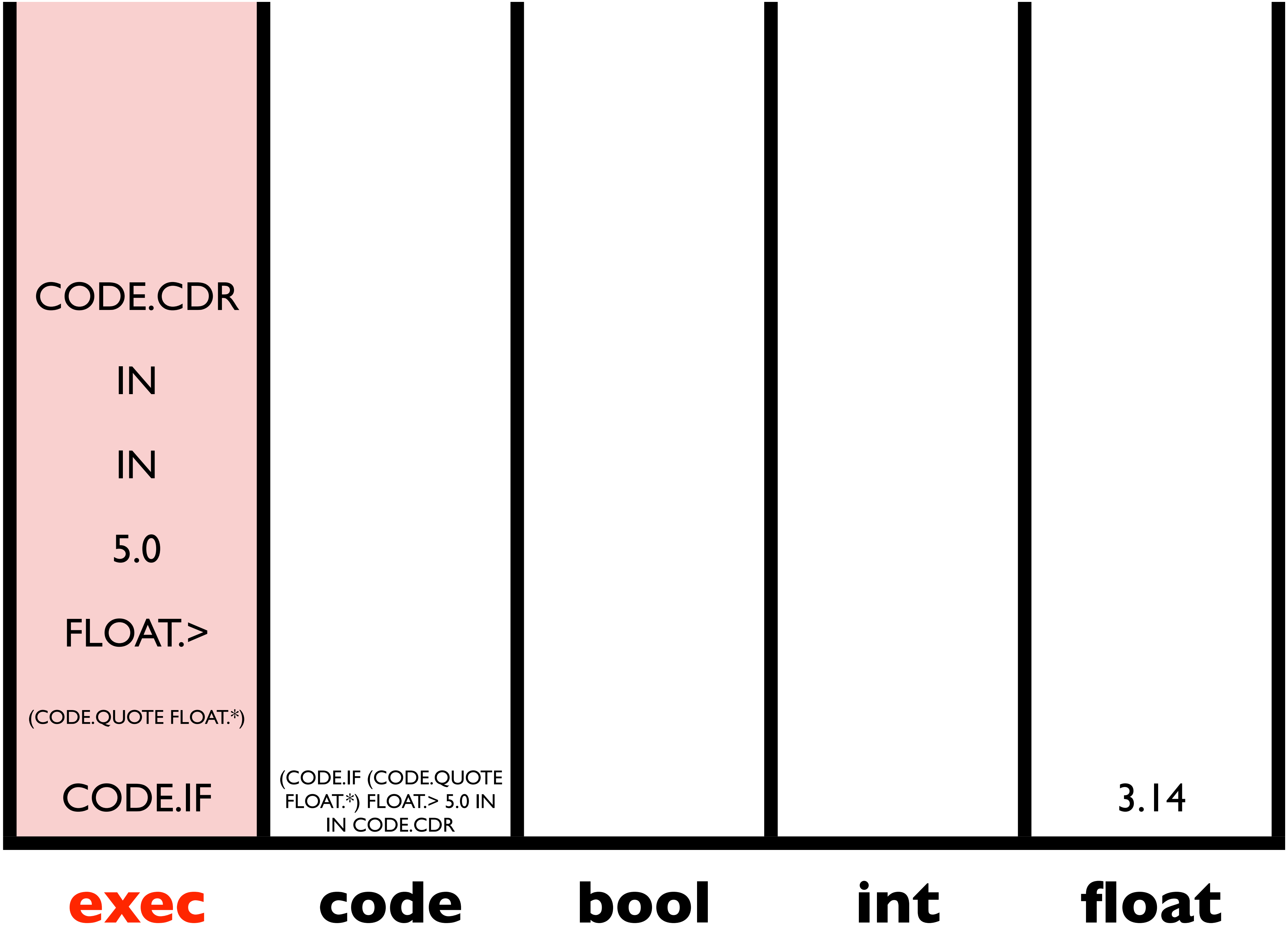
```
( 3.14 CODE.REVERSE CODE.CDR IN IN 5.0
FLOAT.> (CODE.QUOTE FLOAT.*) CODE.IF )
```

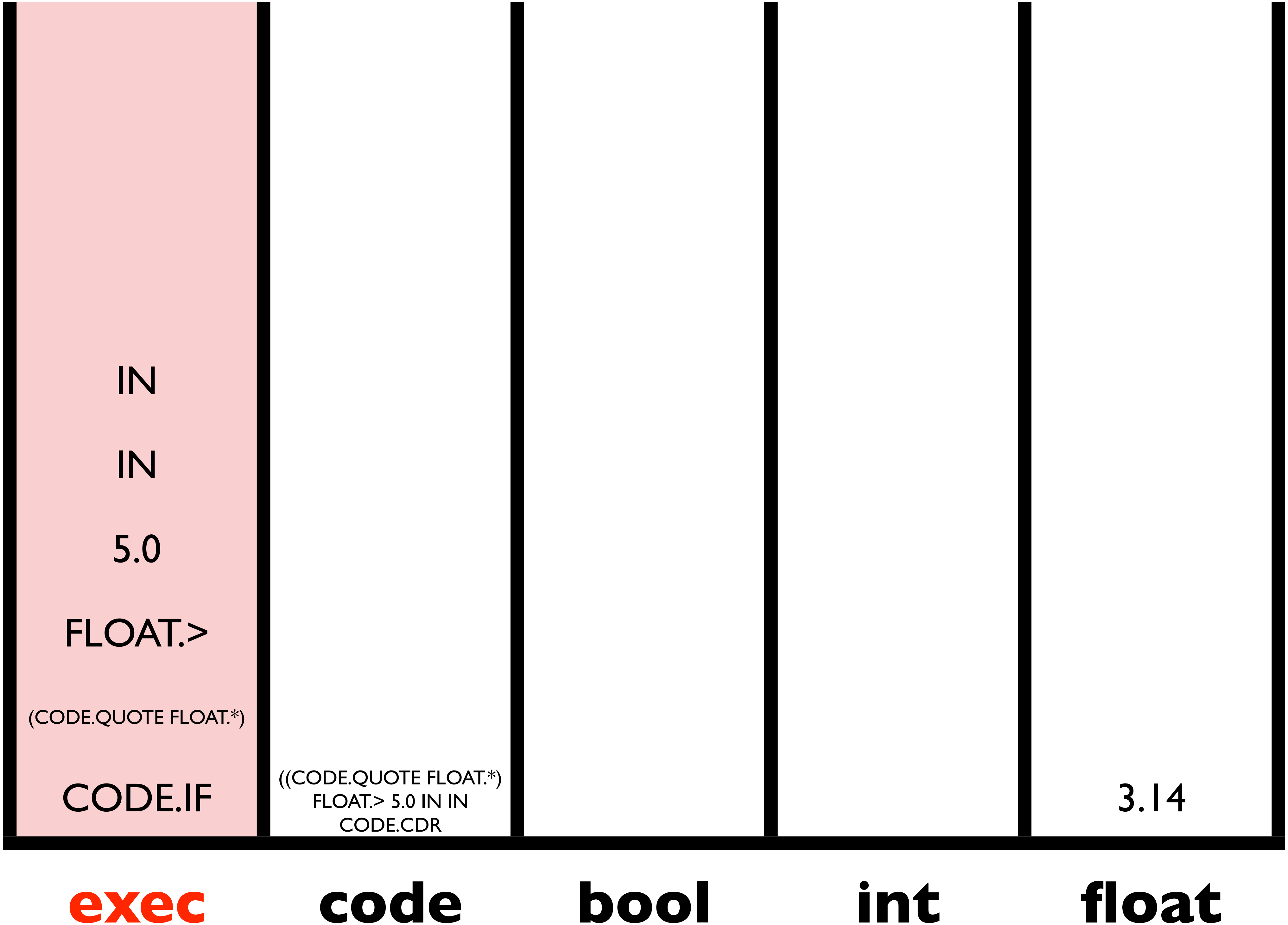
```
IN=4.0
```



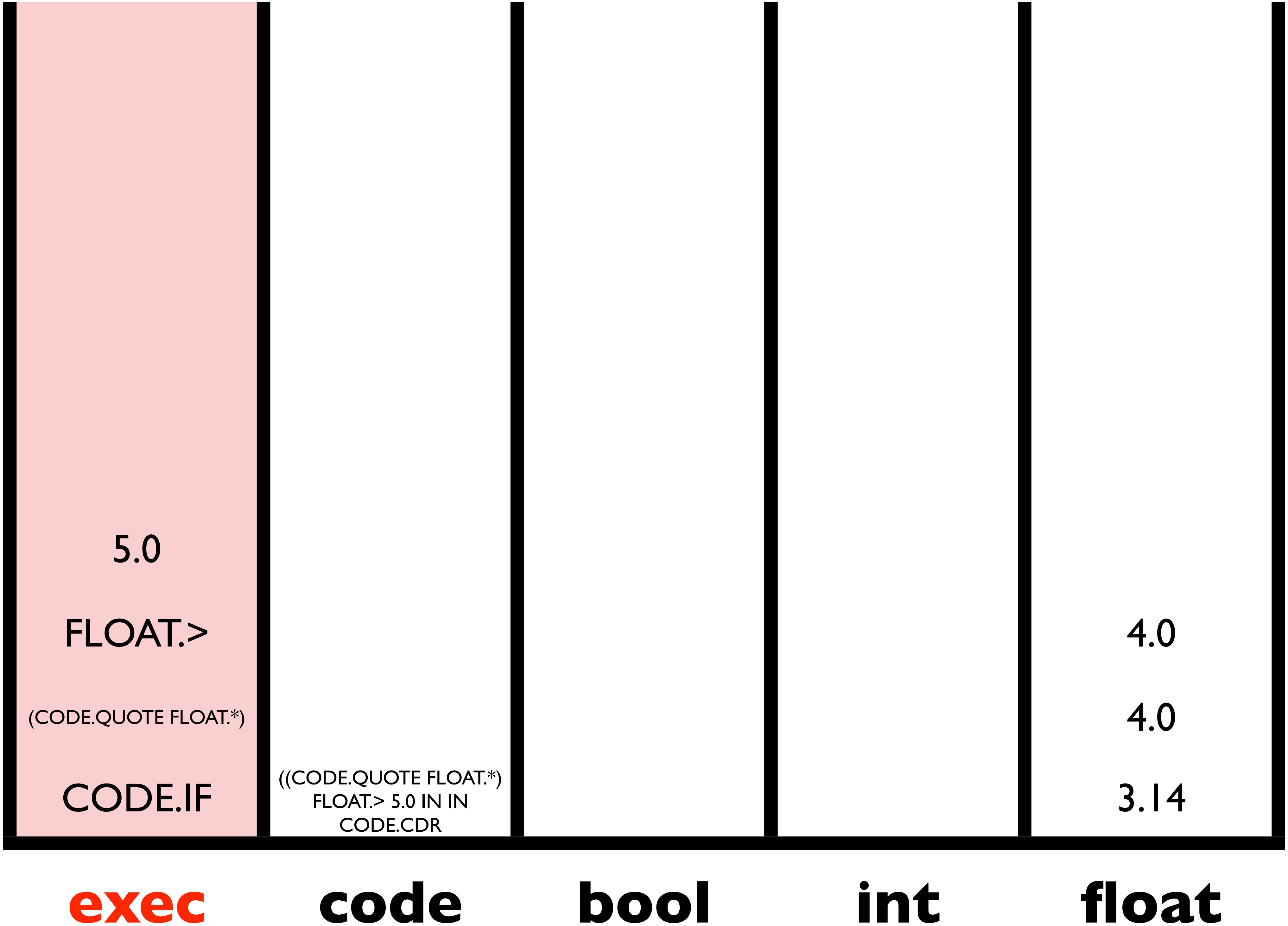

3.14
CODE.REVERSE
CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
(3.14 CODE.REVERSE
CODE.CDR IN IN
5.0 FLOAT.>
exec
code
bool
int
float

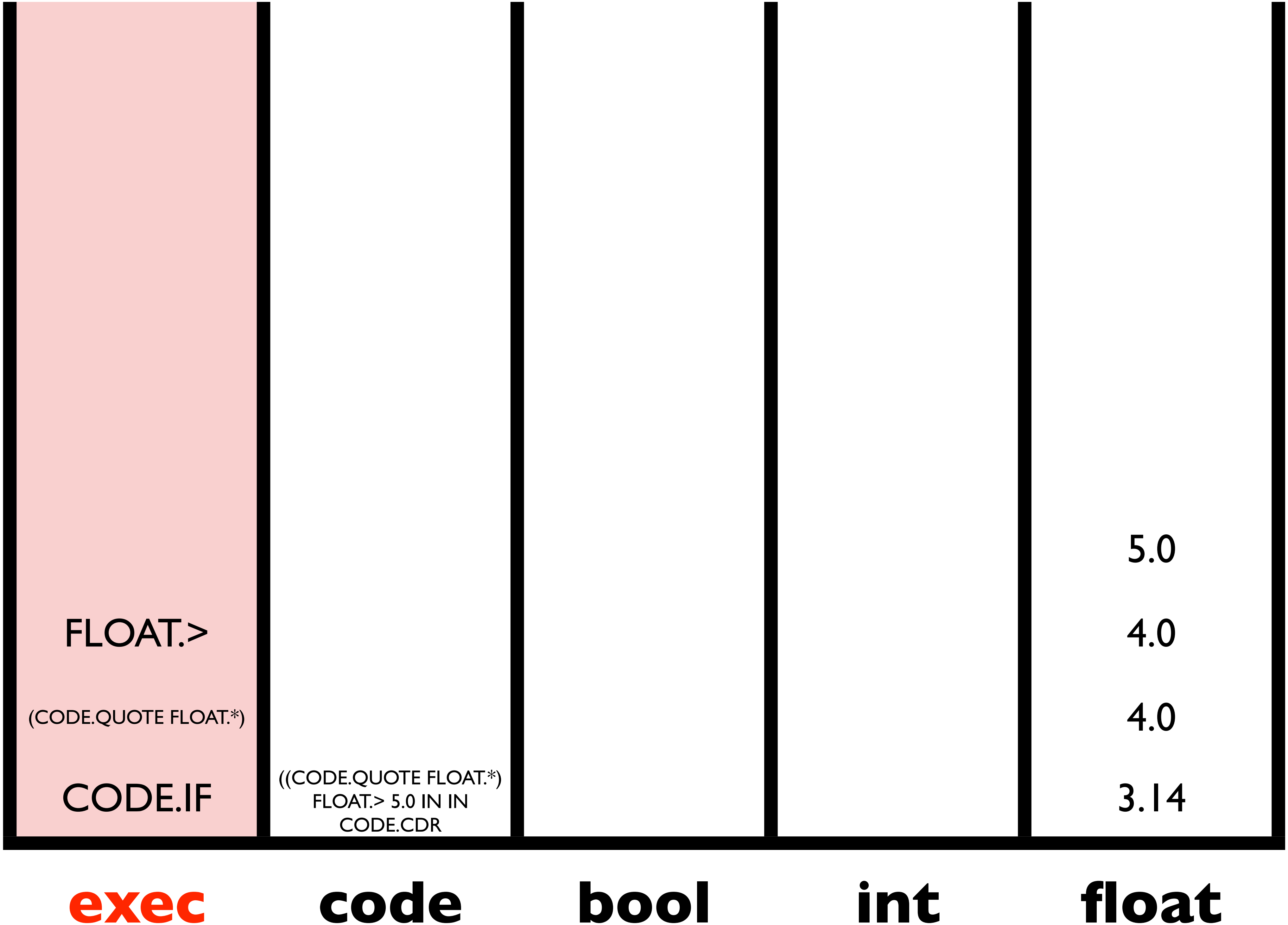

CODE.REVERSE
CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
(3.14 CODE.REVERSE CODE.CDR IN IN 5.0 FLOAT.>
3.14
exec
code
bool
int
float


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

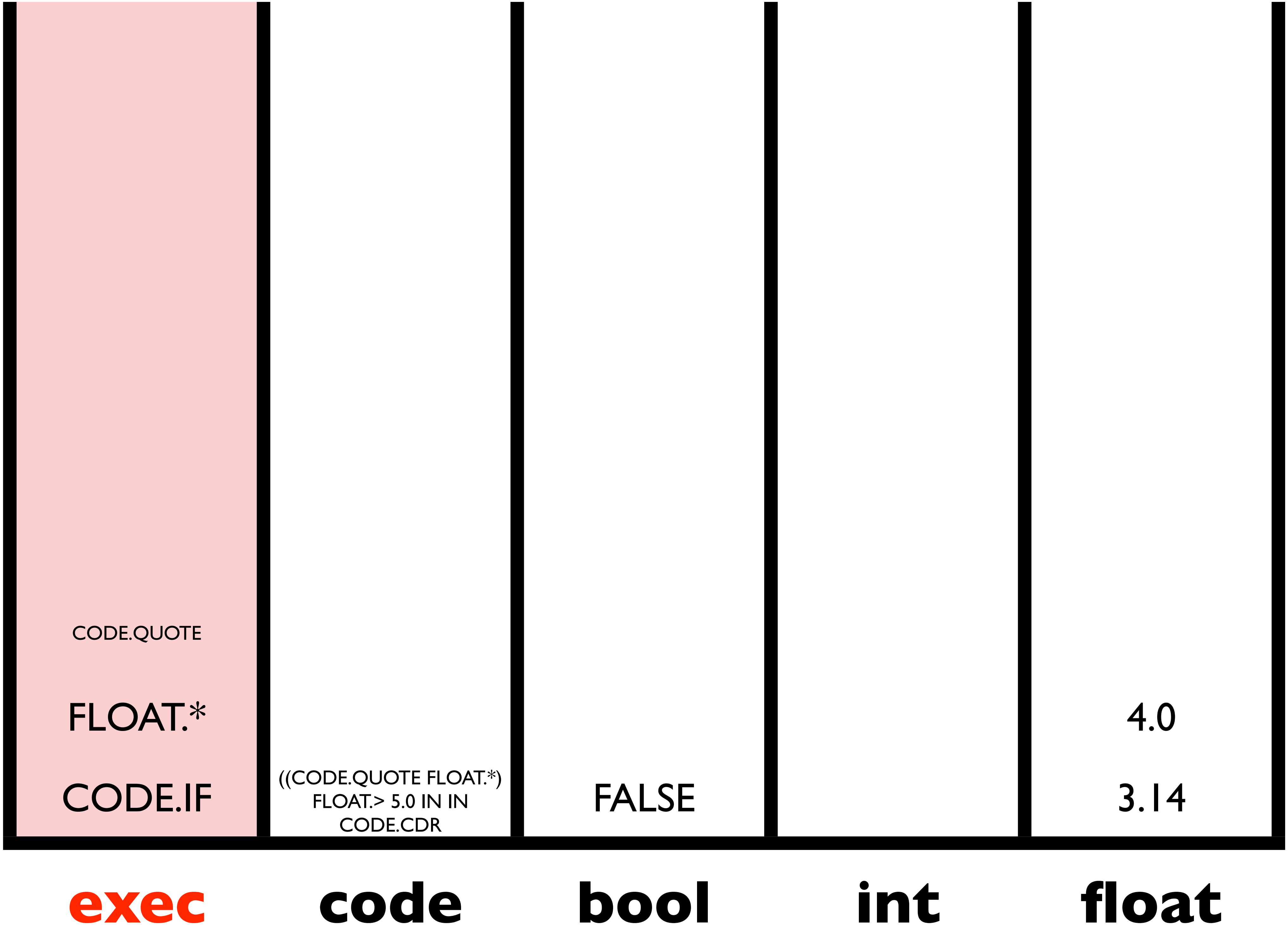

IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
3.14
exec
code
bool
int
float

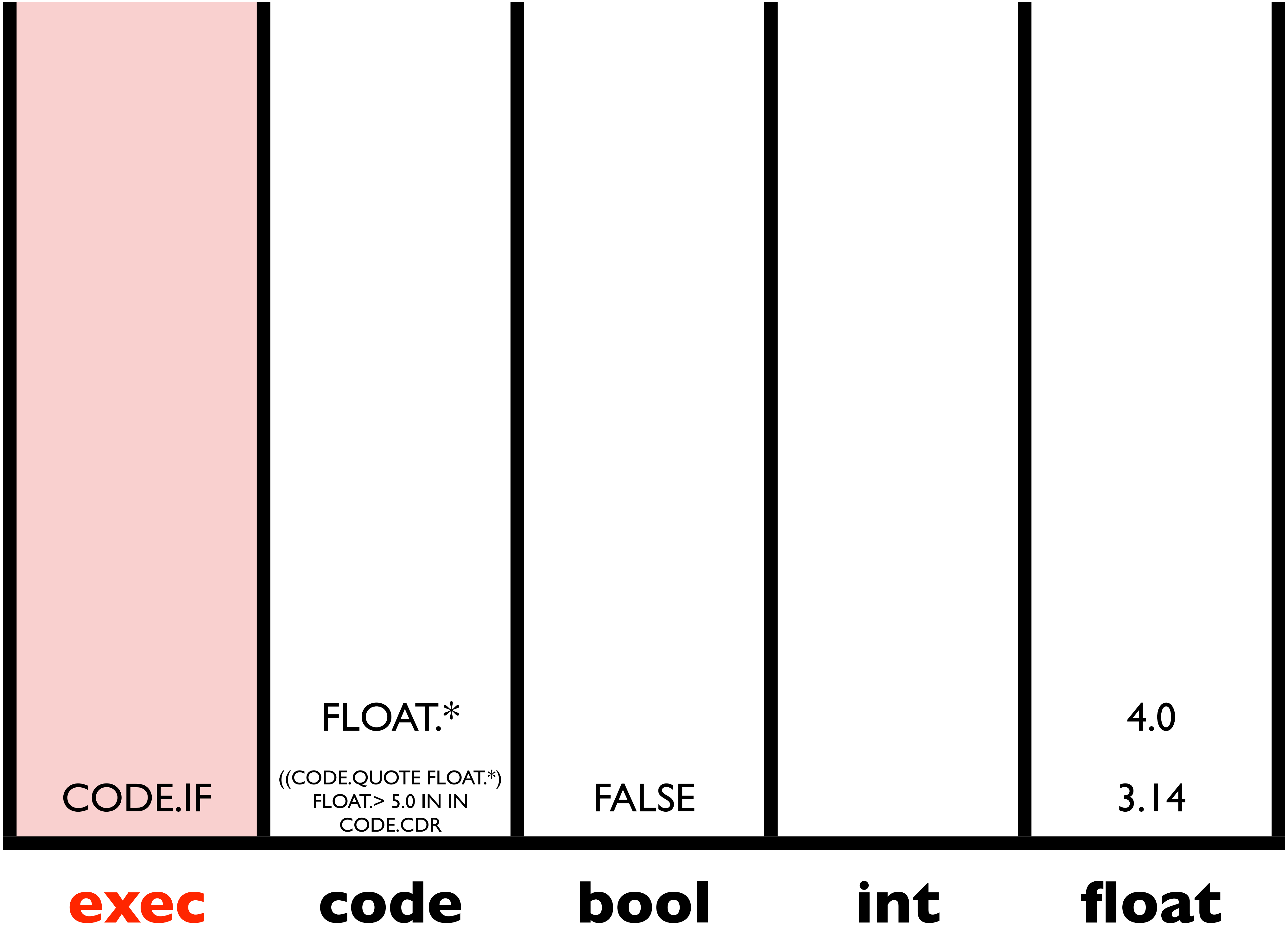
exec	code	bool	int	float
IN				
5.0				
FLOAT.>				
(CODE.QUOTE FLOAT.*)				4.0
CODE.IF	((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR			3.14


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

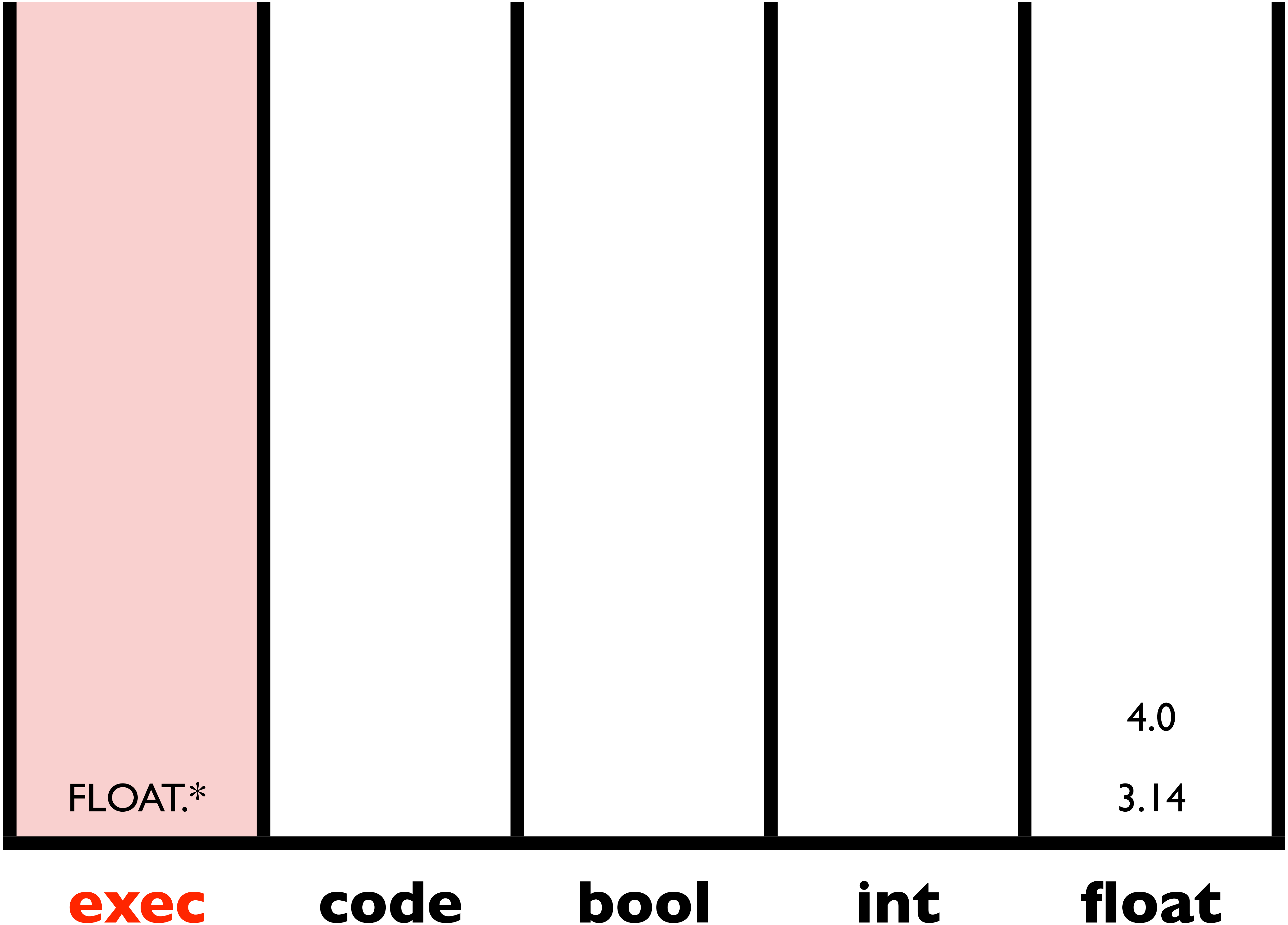

FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*)
FLOAT.> 5.0 IN IN
CODE.CDR
5.0
4.0
4.0
3.14
exec
code
bool
int
float

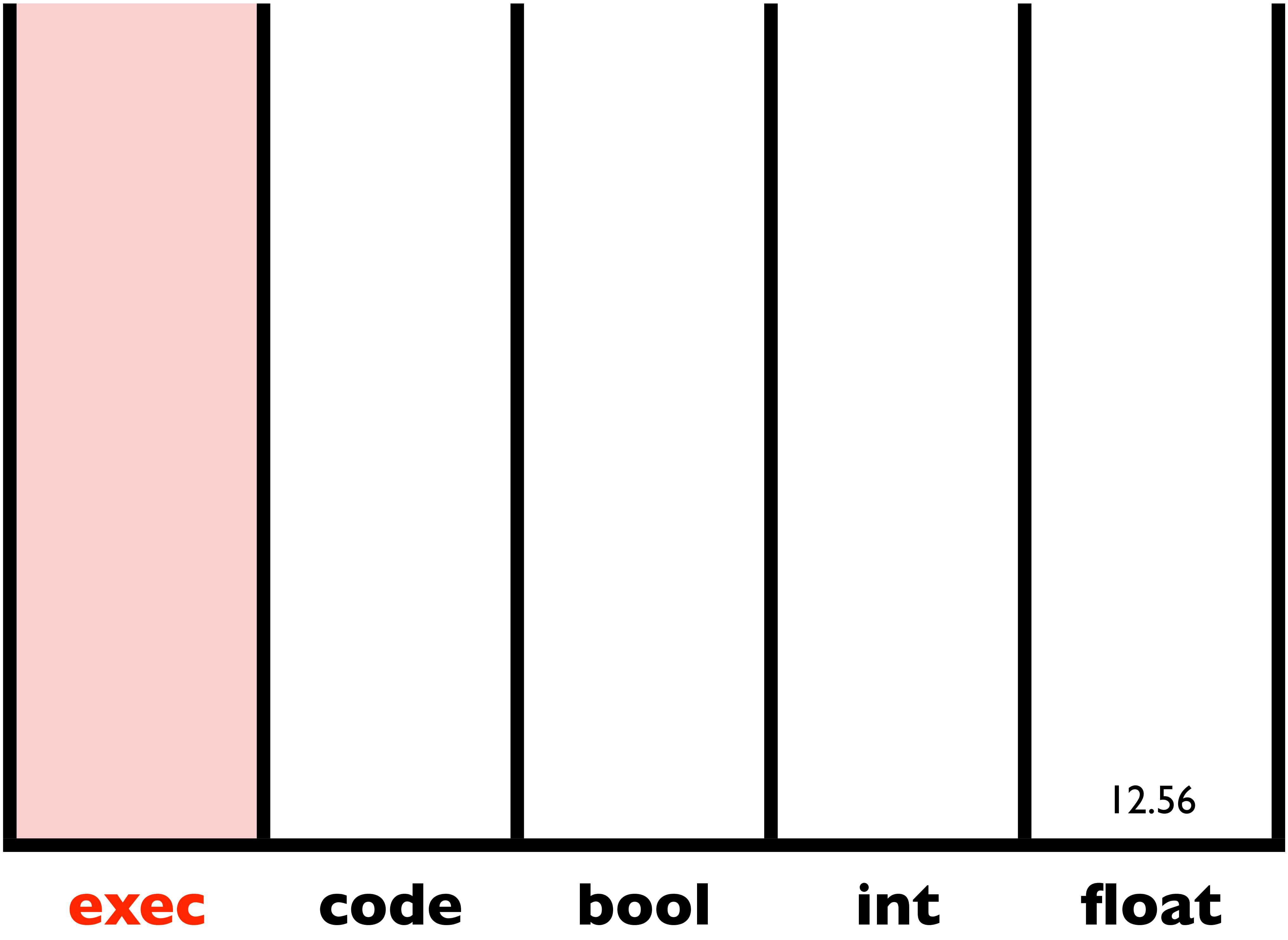
exec	code	bool	int	float
(CODE.QUOTE FLOAT.*)				4.0
CODE.IF	((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR	FALSE		3.14


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



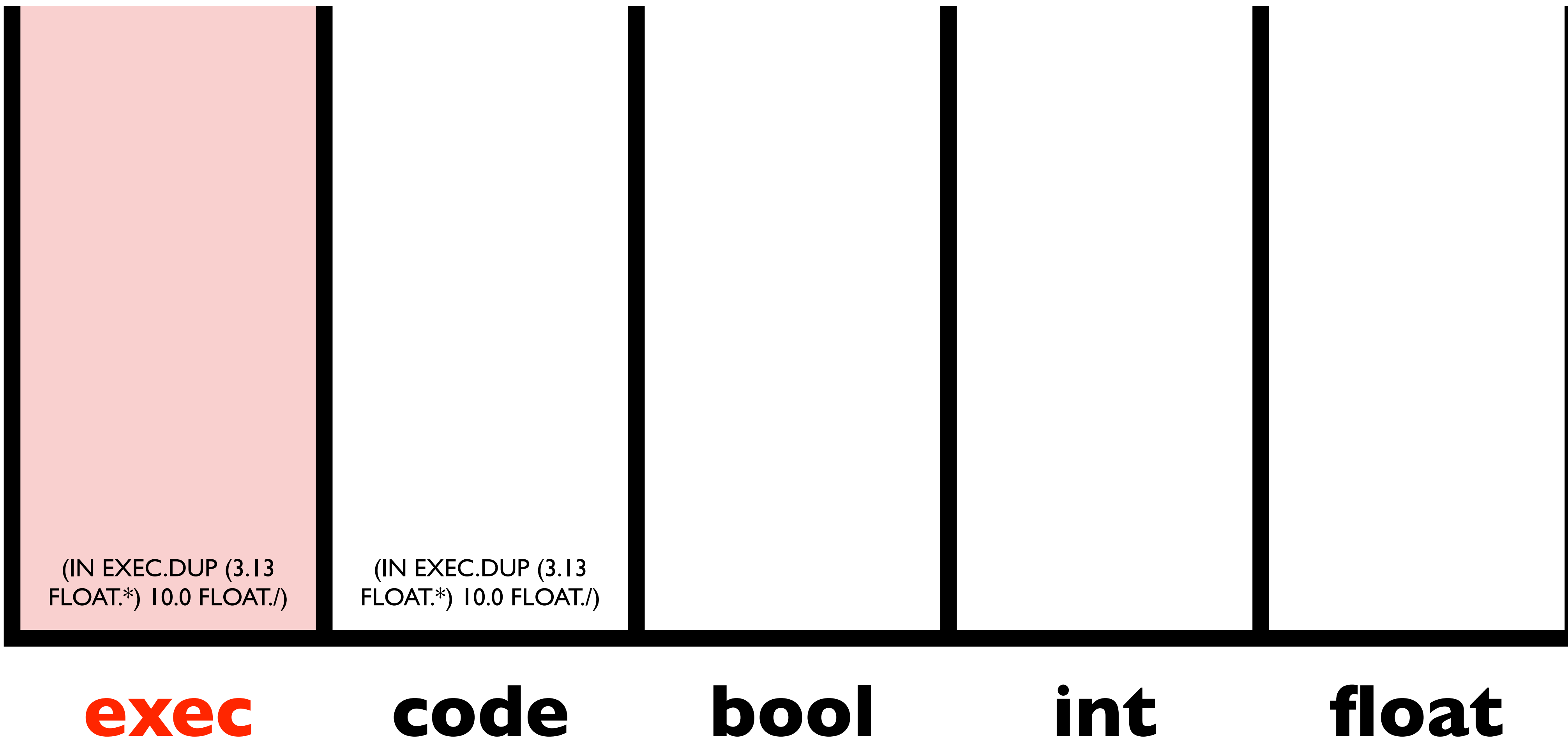
CODE.IF
FLOAT.*
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
FALSE
4.0
3.14
exec
code
bool
int
float

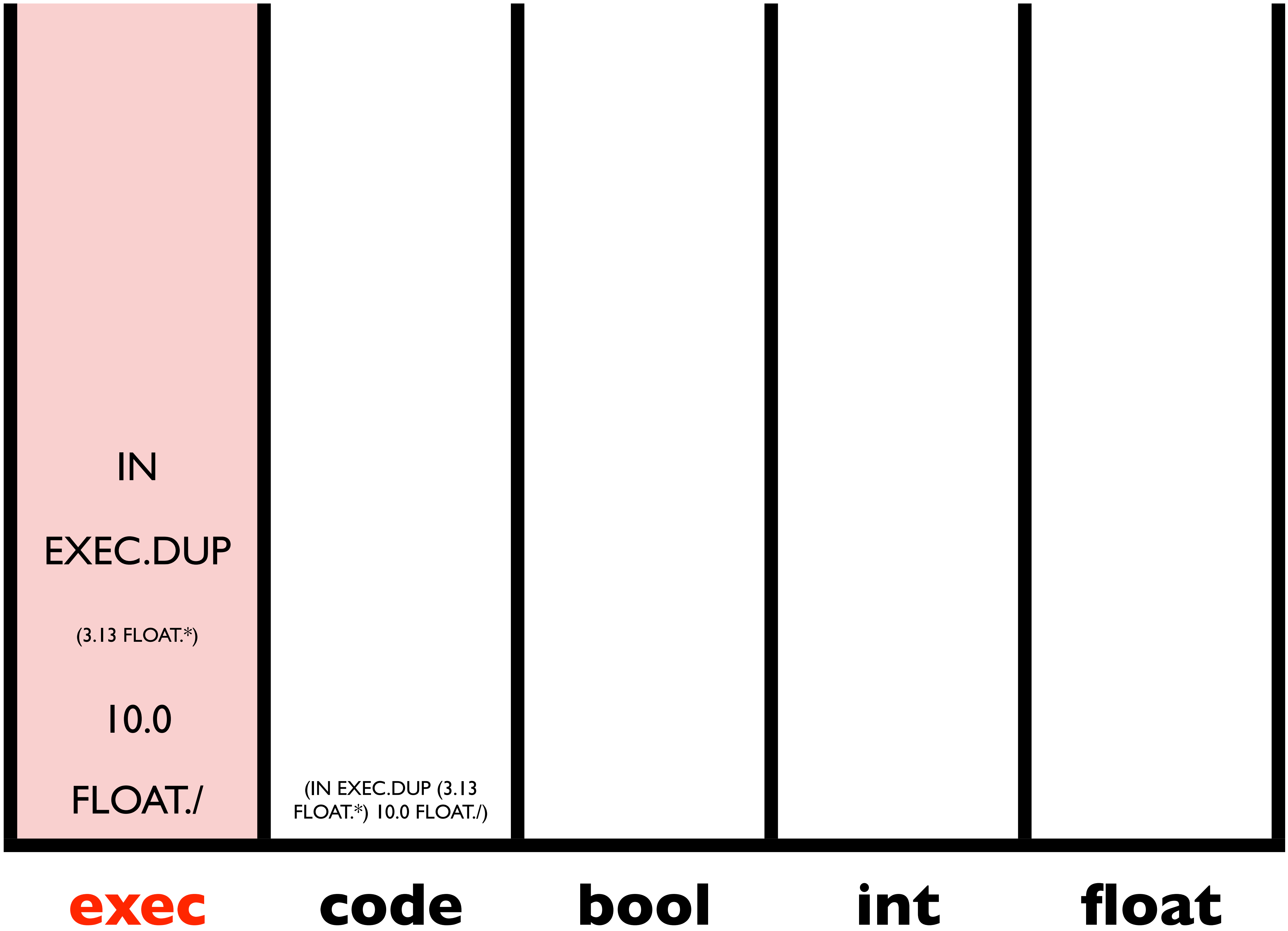

FLOAT.*
4.0
3.14
exec
code
bool
int
float

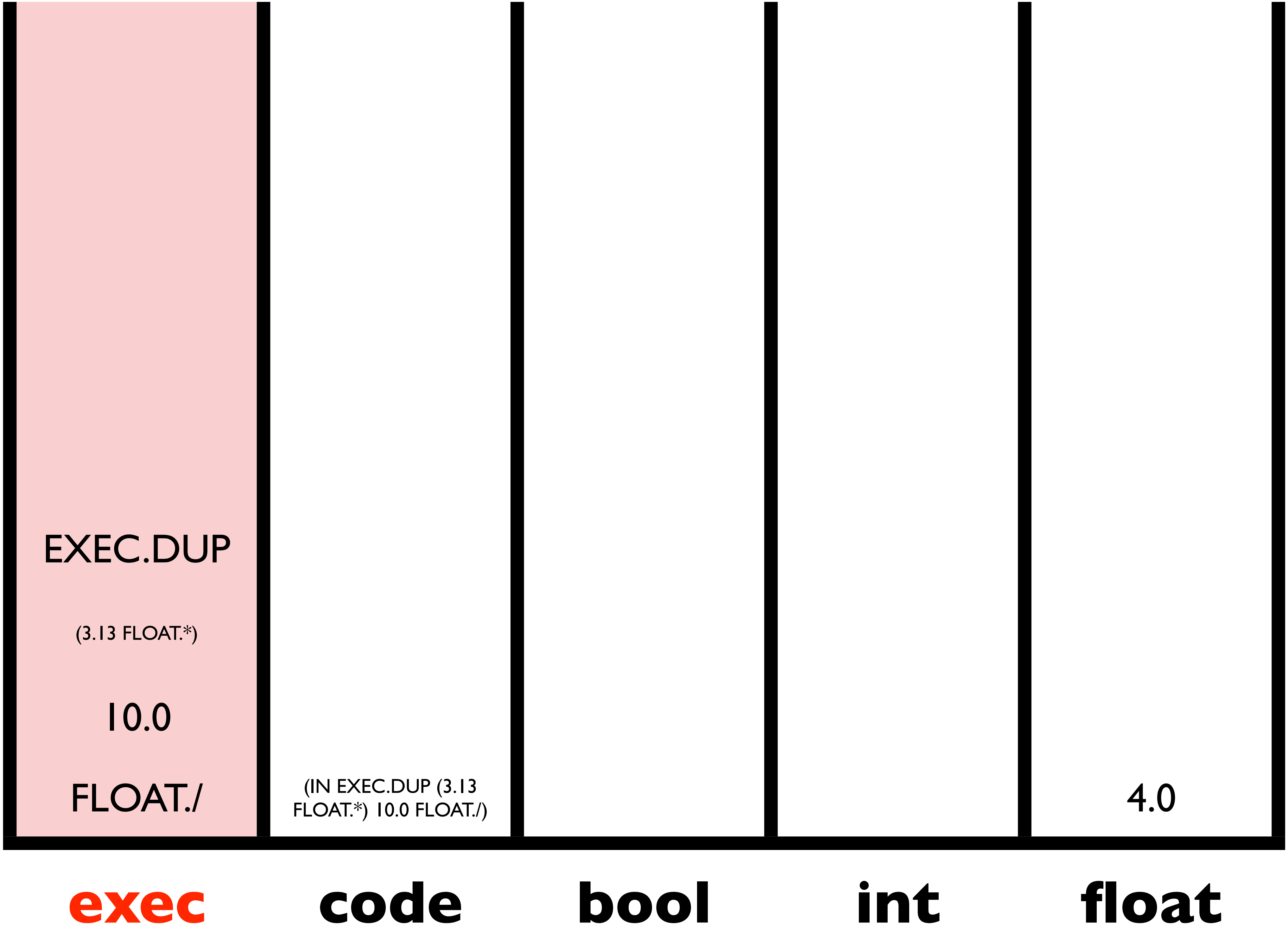

12.56
exec
code
bool
int
float

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

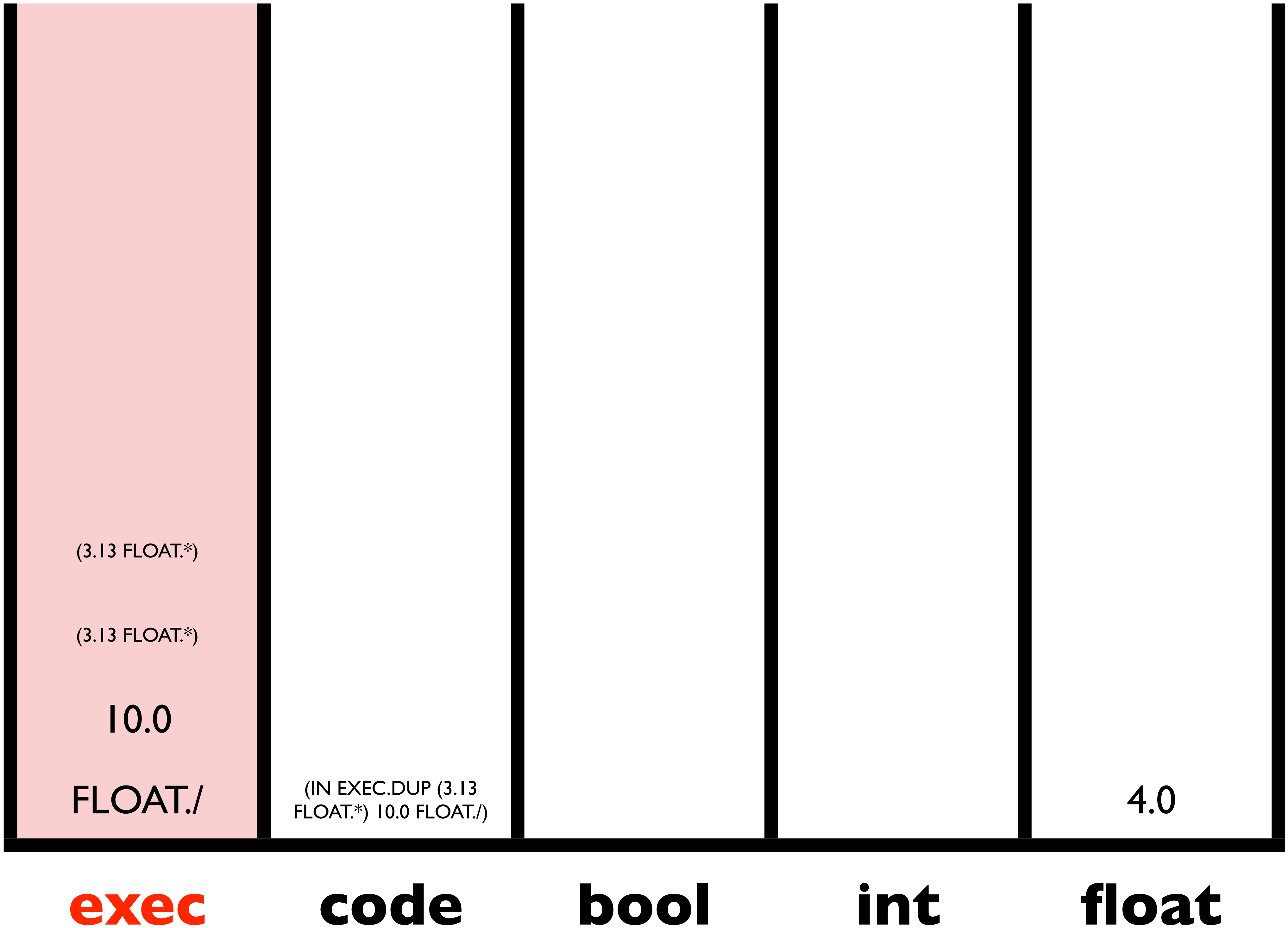
```
IN=4.0
```



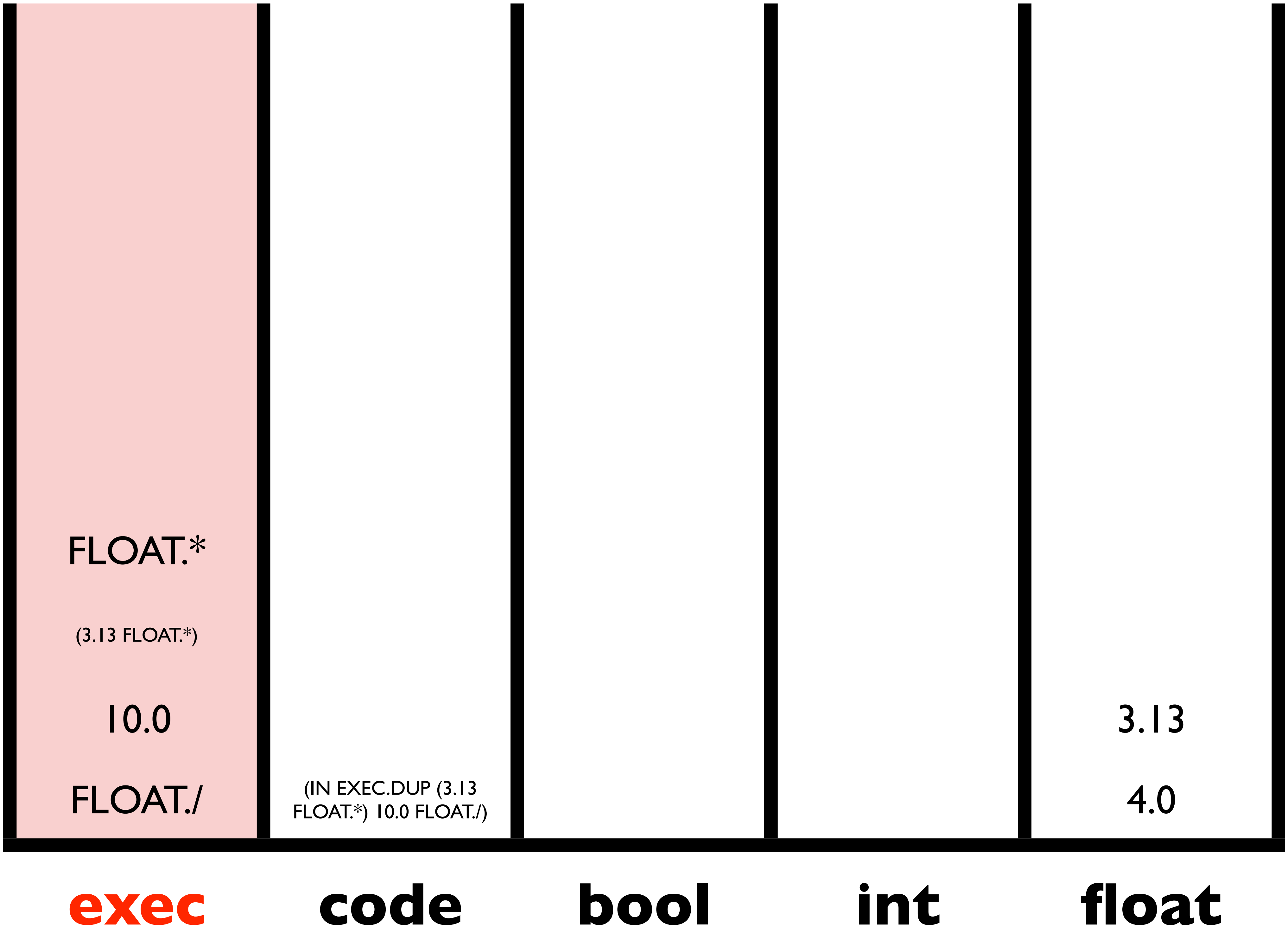

IN
EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
exec
code
bool
int
float

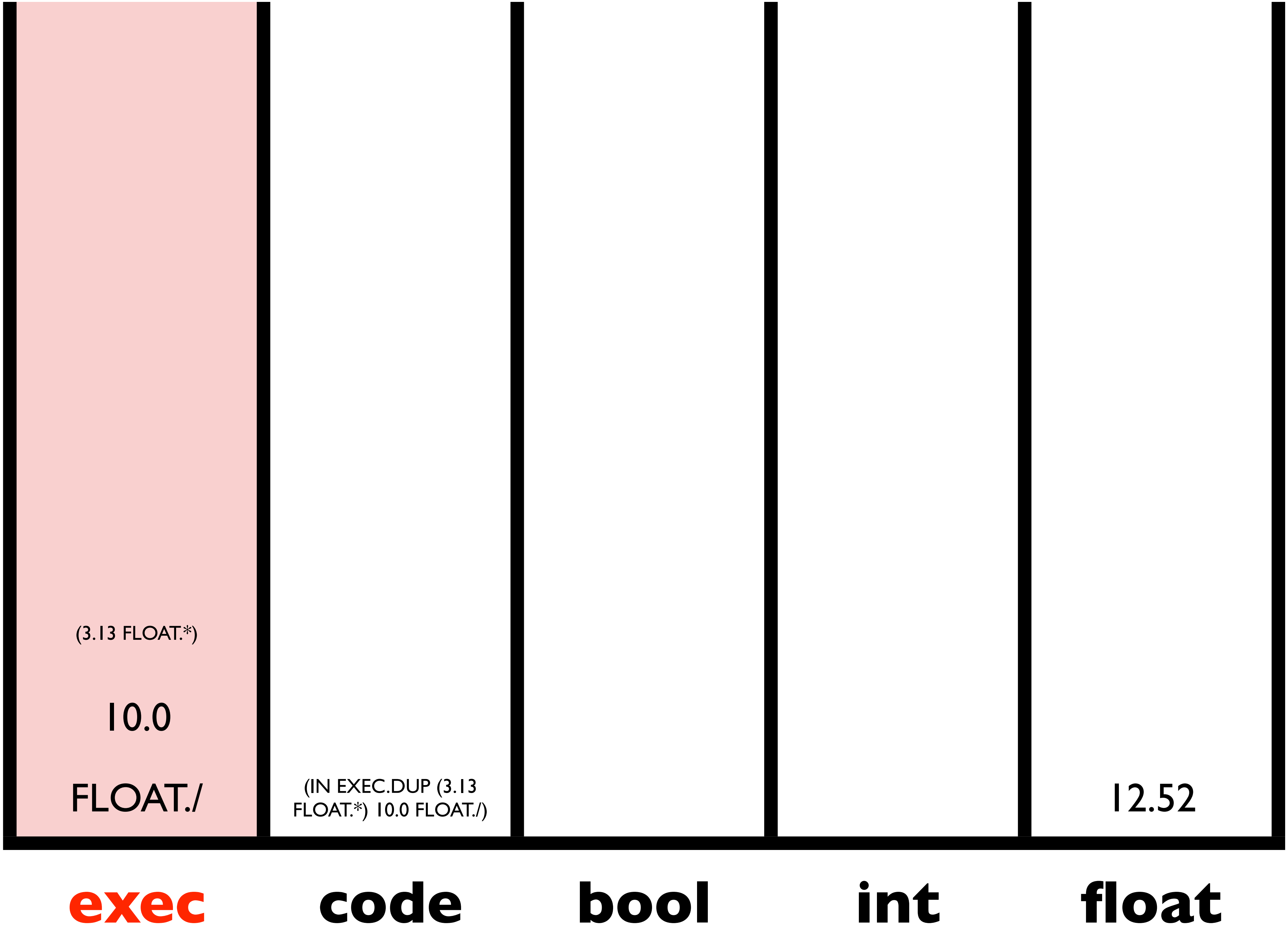


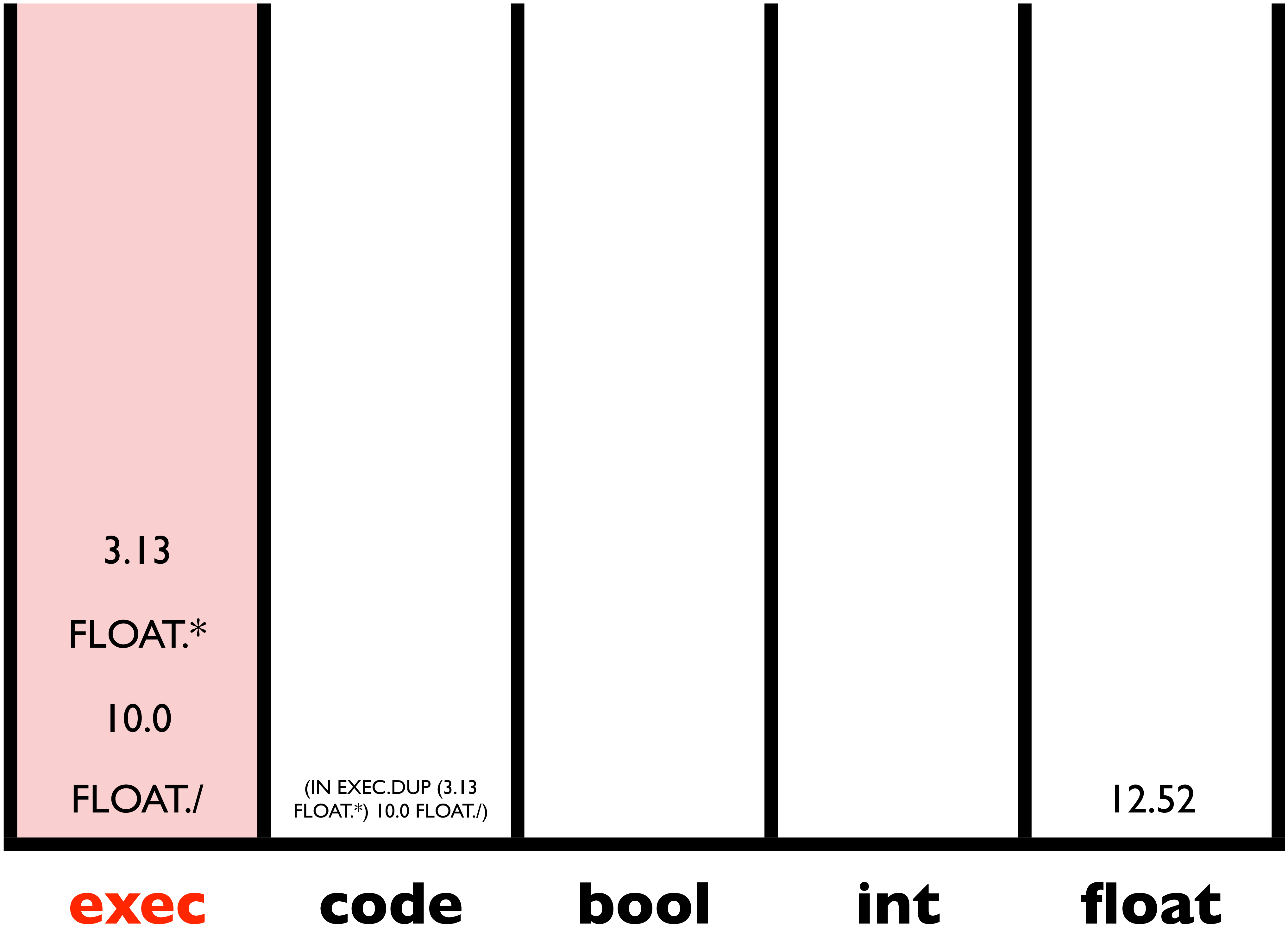
EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

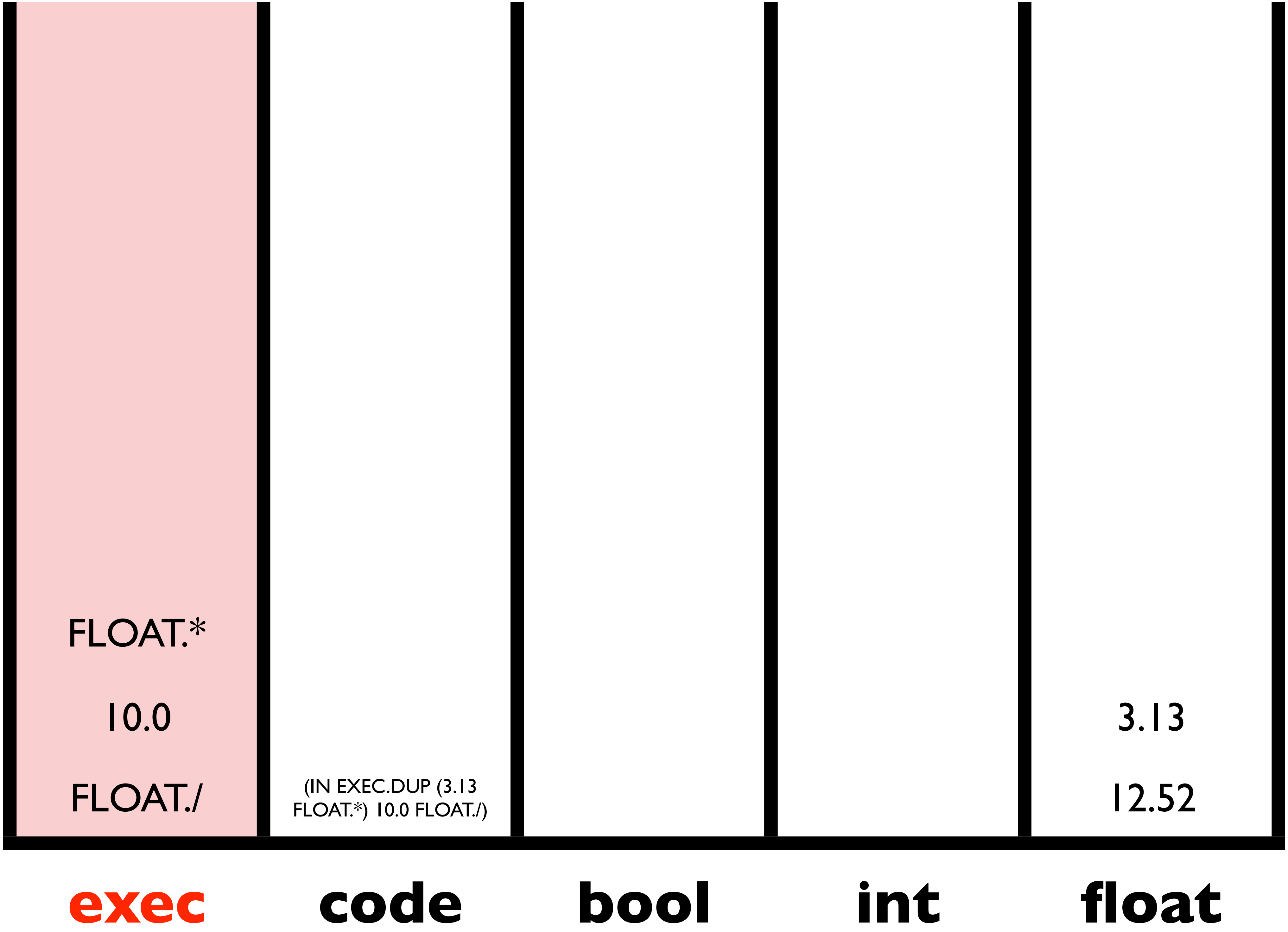

(3.13 FLOAT.*)
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

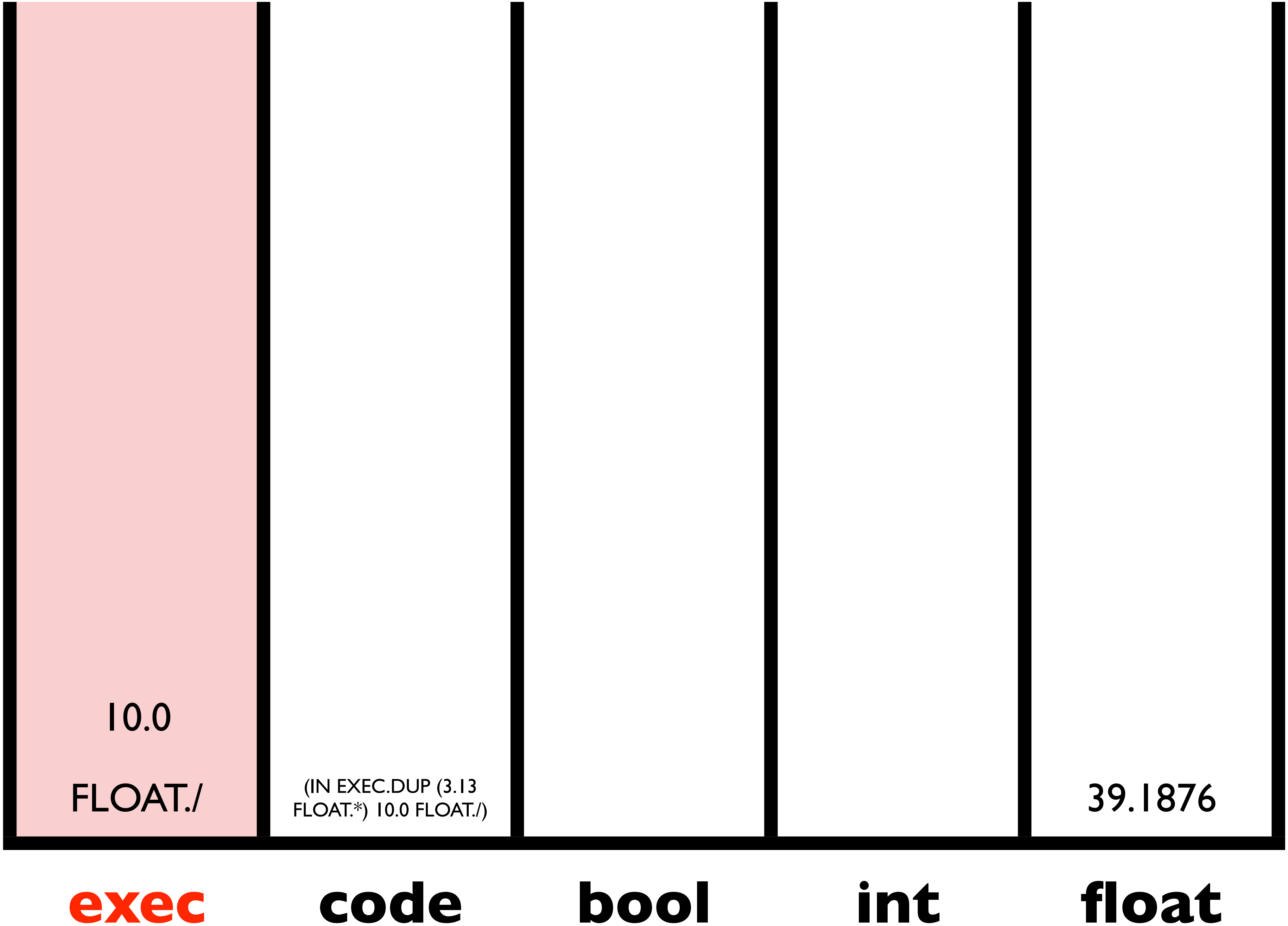
exec	code	bool	int	float
3.13				
FLOAT.*				
(3.13 FLOAT.*)				
10.0				
FLOAT./	(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)			4.0

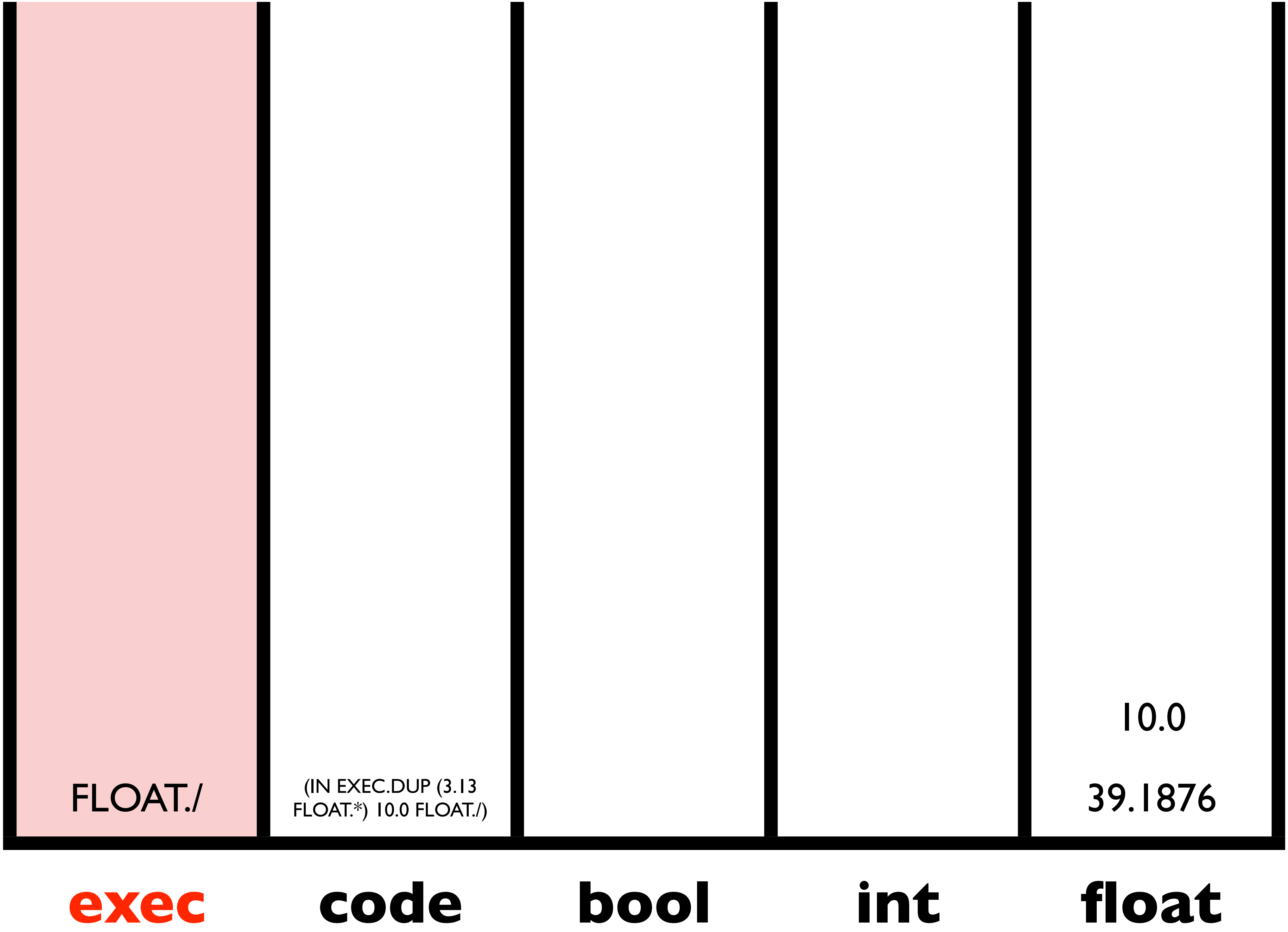

FLOAT.*
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
3.13
4.0
exec
code
bool
int
float

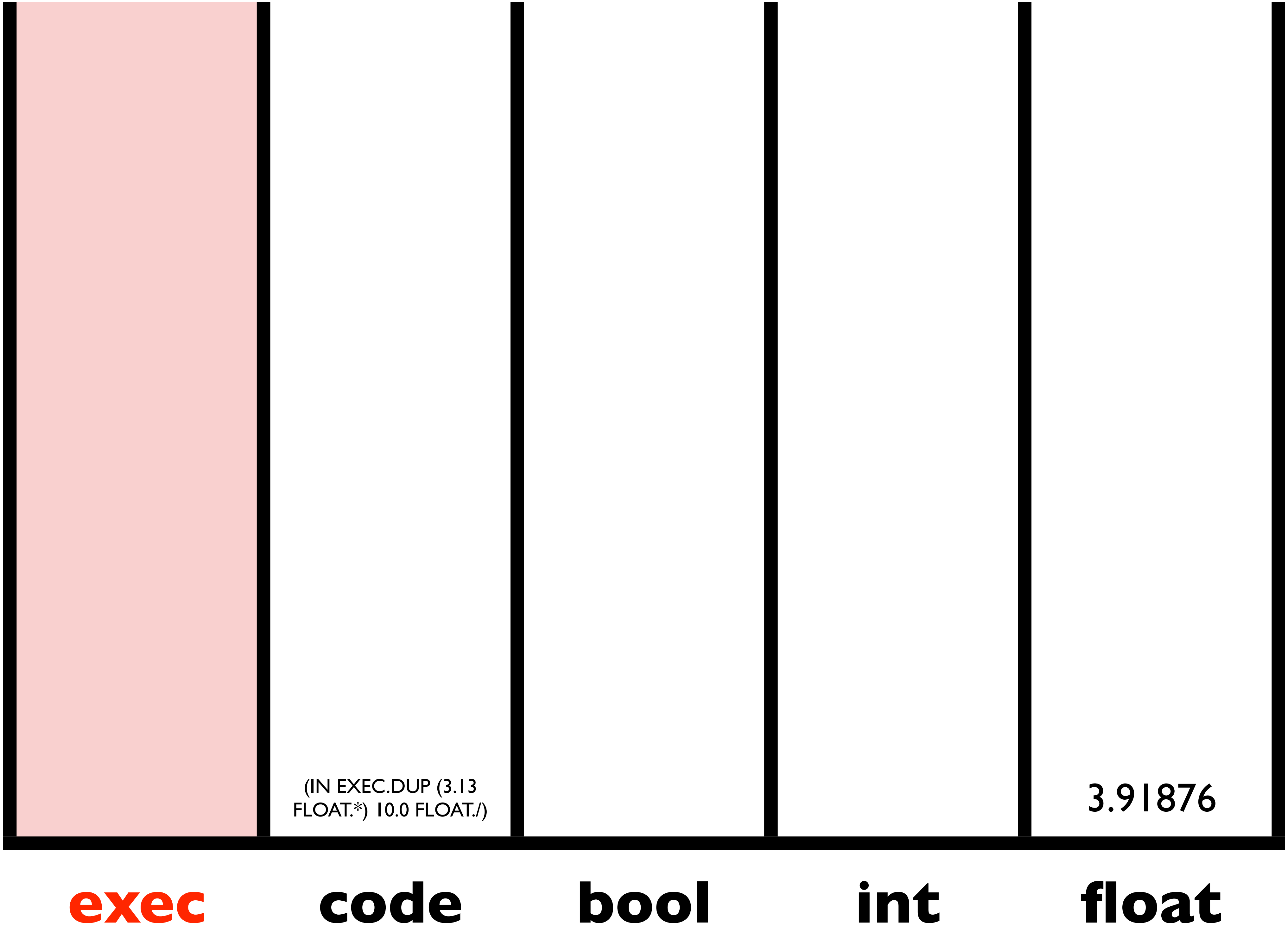

(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
12.52
exec
code
bool
int
float


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

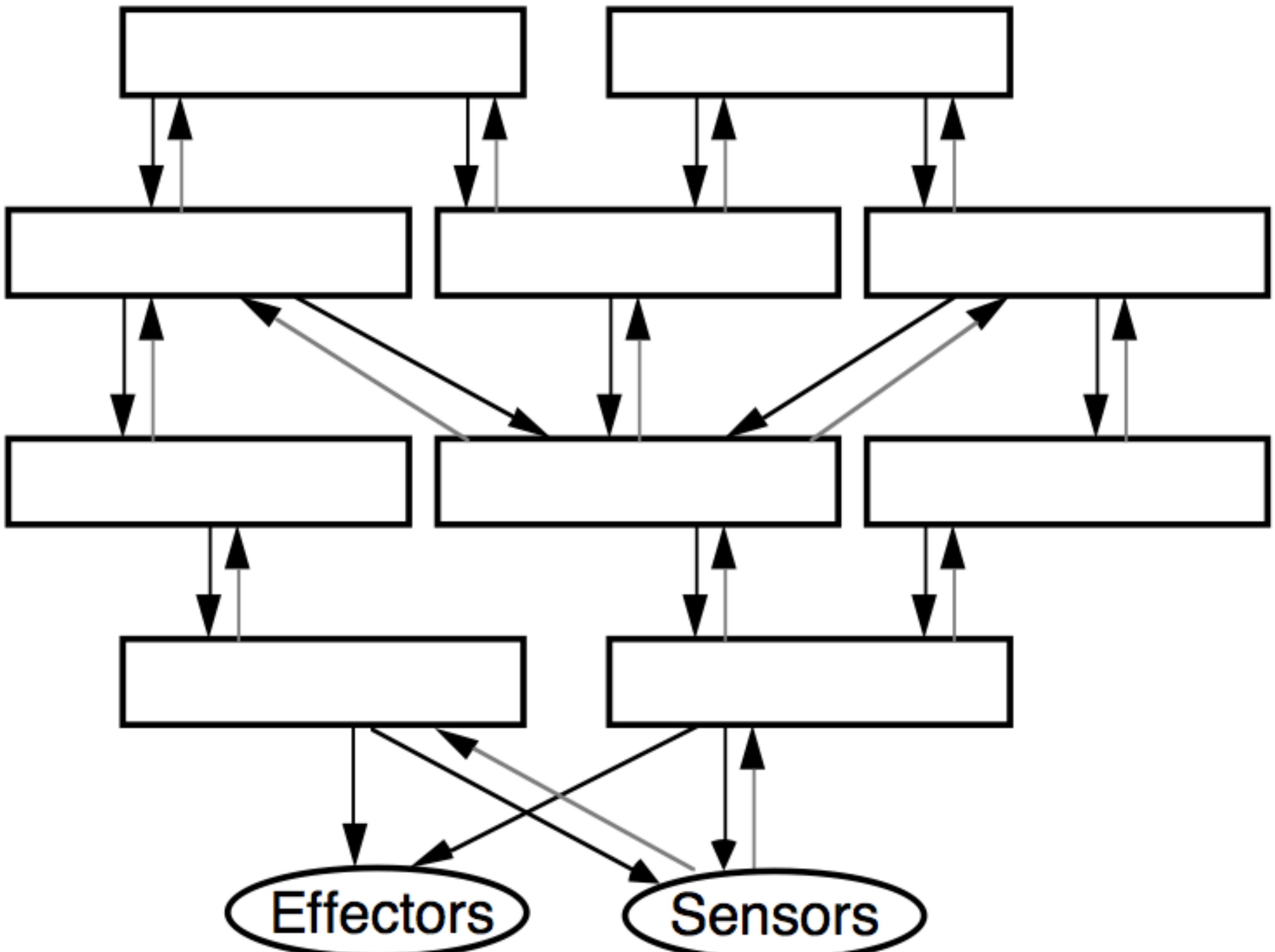

FLOAT.*
10.0
FLOAT./
(IN EXEC.DUP (3.13
FLOAT.*) 10.0 FLOAT./)
3.13
12.52
exec
code
bool
int
float


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

Modularity is Everywhere



Modules in GP

- Automatically-defined functions (Koza), macros (Spector)
- Architecture-altering operations (Koza)
- Module acquisition/encapsulation systems (Kinnear, Roberts, many others)
- Modules in GE (Swafford et al., 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
  ```

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

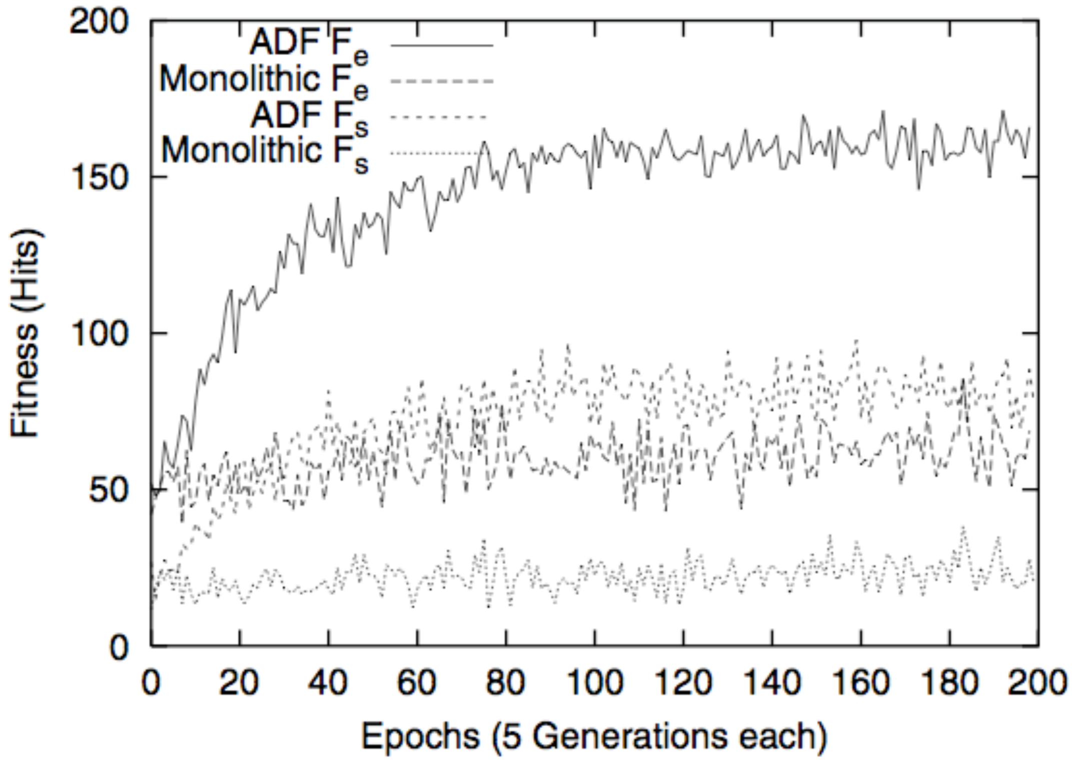
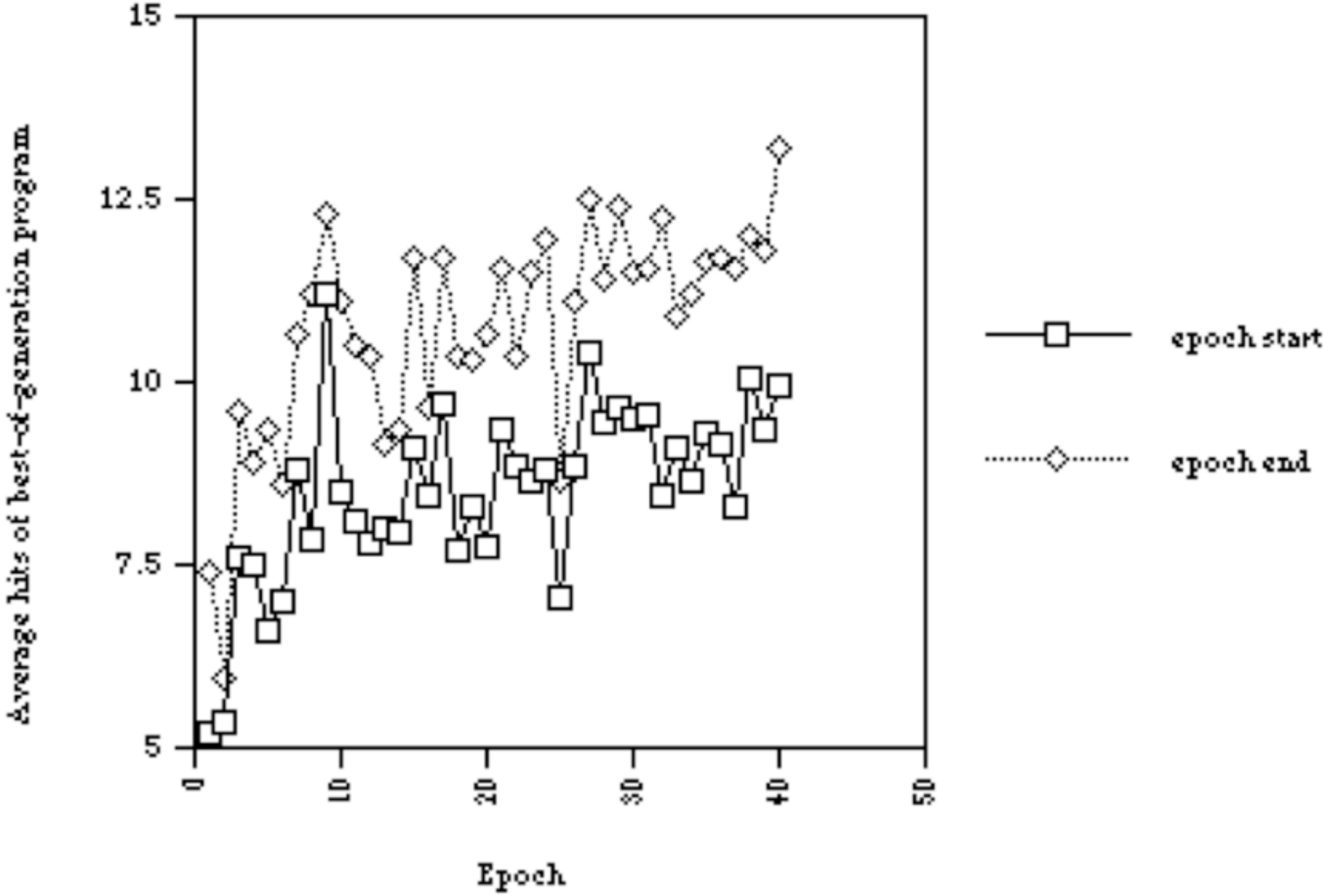


Figure 2: Average fitness values at the start (F_s) and end (F_e) of each epoch when regressing to $y = A\sin(Ax)$. A is selected at the start of each epoch uniformly from the range $[0,6)$.

Code-as-data Modularity in Push



Tags

- 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

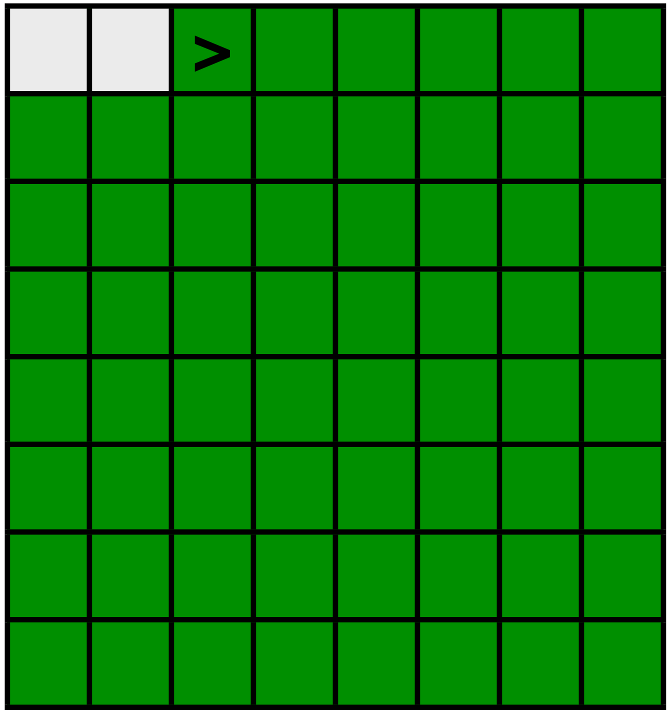
- Add mechanisms for tagging code
- Add mechanisms for retrieving/branching to code with closest matching tag
- As long as any code has been tagged, all branches go somewhere
- Number of tagged modules can grow incrementally over evolutionary time

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

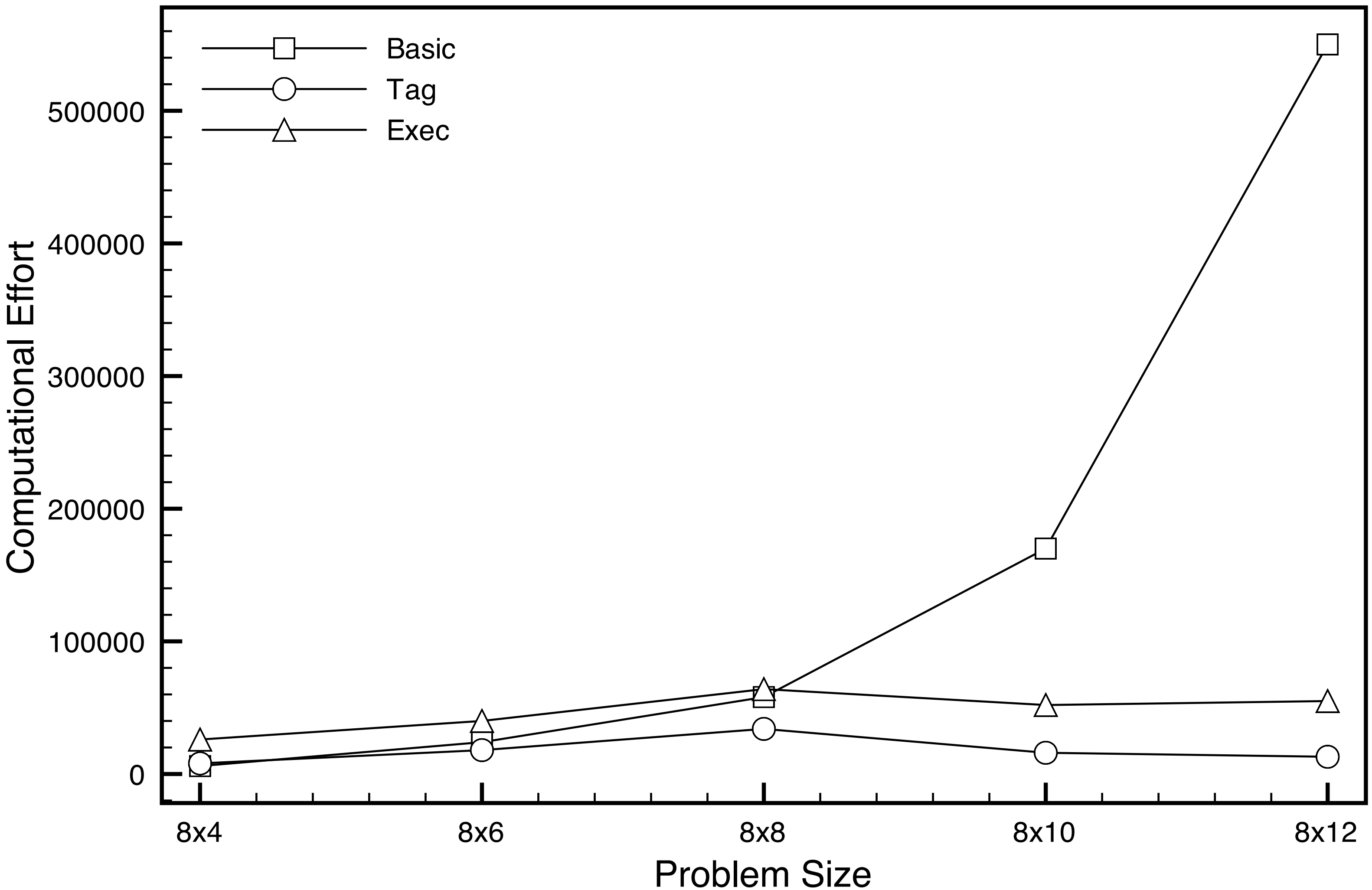


Lawnmower Instructions

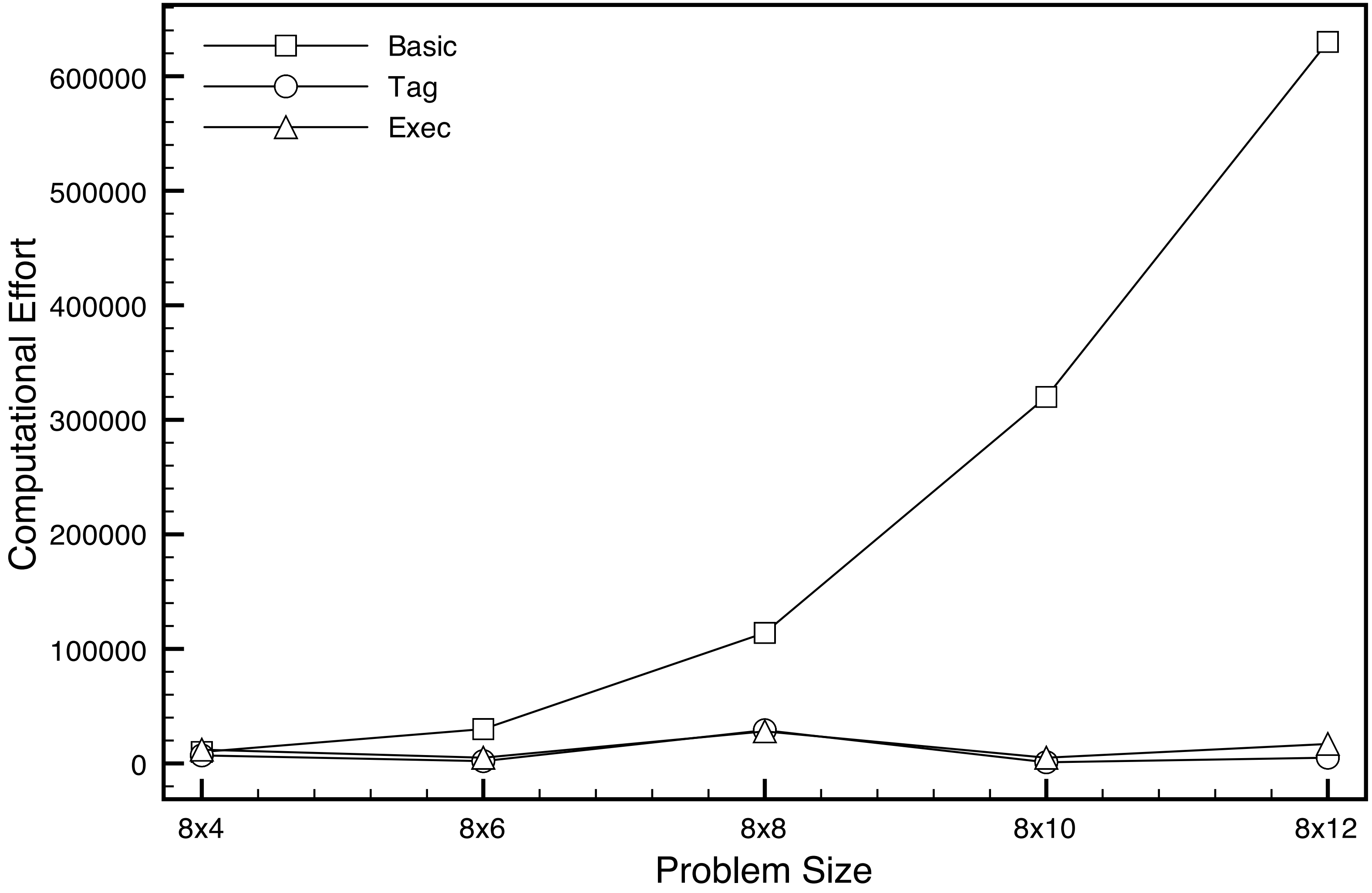
Condition	Instructions
Basic	left, mow, v8a, frog, $\mathcal{R}_{v8}$
Tag	left, mow, v8a, frog, $\mathcal{R}_{v8}$, tag.exec.[1000], tagged.[1000]
Exec	left, mow, v8a, frog, $\mathcal{R}_{v8}$, exec.dup, exec.pop, exec.rot, exec.swap, exec.k, exec.s, exec.y

Lawnmower Effort*

* with frog=noop bug



Lawnmower Effort

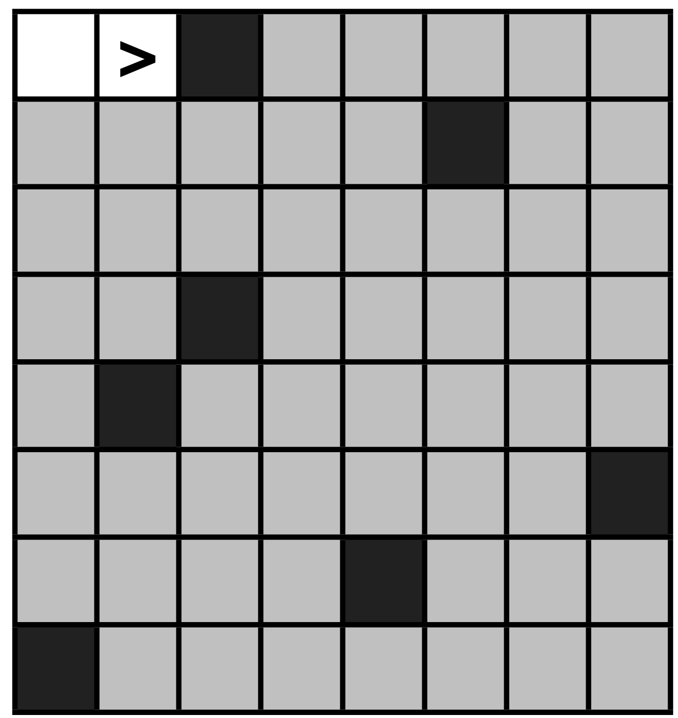


Lawnmower Effort

	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

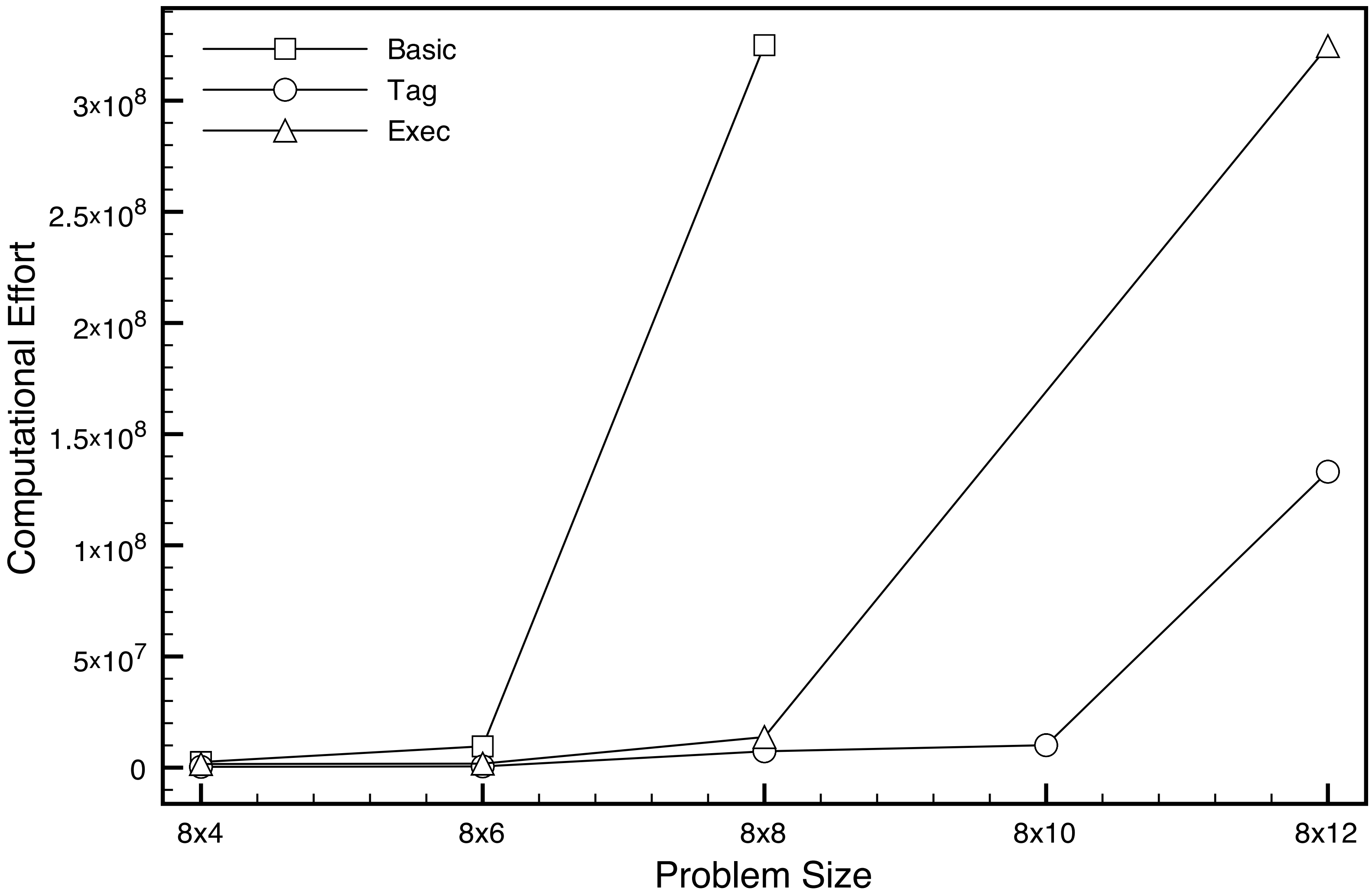


DSOAR Instructions

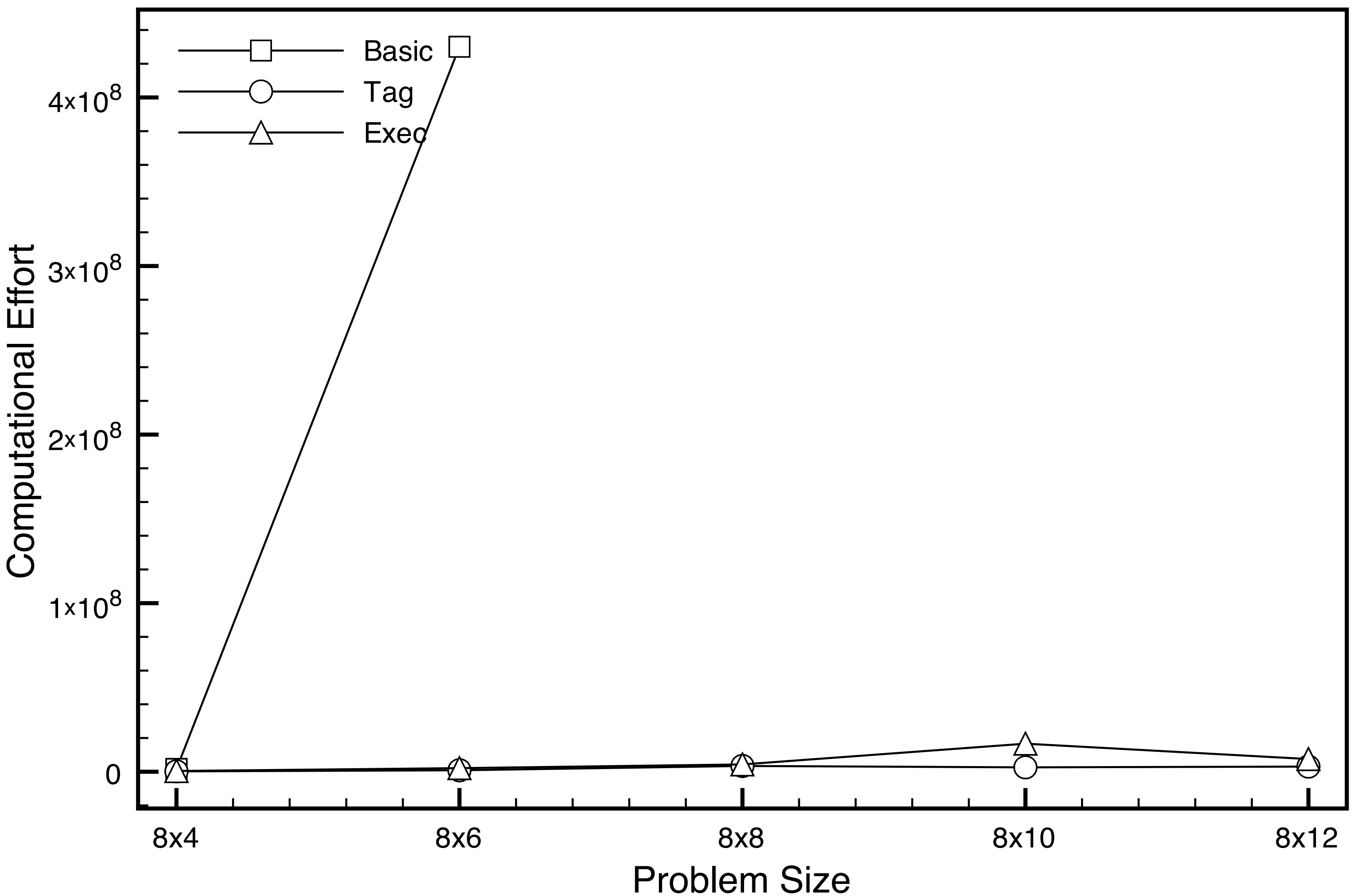
Condition	Instructions
Basic	if-dirty, if-obstacle, left, mop, v8a, frog, $\mathcal{R}_{v8}$
Tag	if-dirty, if-obstacle, left, mop, v8a, frog, $\mathcal{R}_{v8}$, tag.exec.[1000], tagged.[1000]
Exec	if-dirty, if-obstacle, left, mop, v8a, frog, $\mathcal{R}_{v8}$, exec.dup, exec.pop, exec.rot, exec.swap,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

DSOAR Effort*

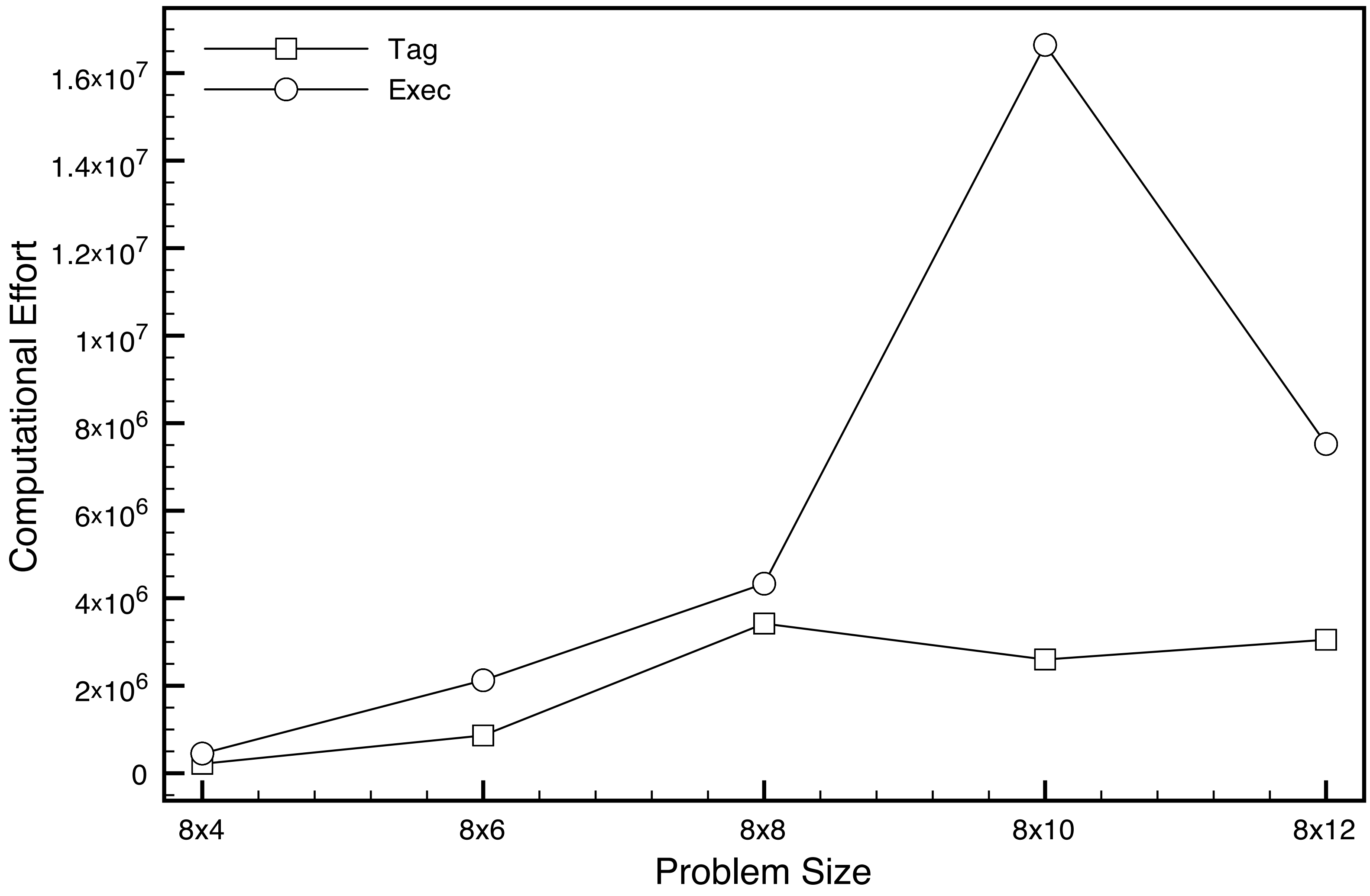
* with frog=noop bug



DSOAR Effort



DSOAR Effort



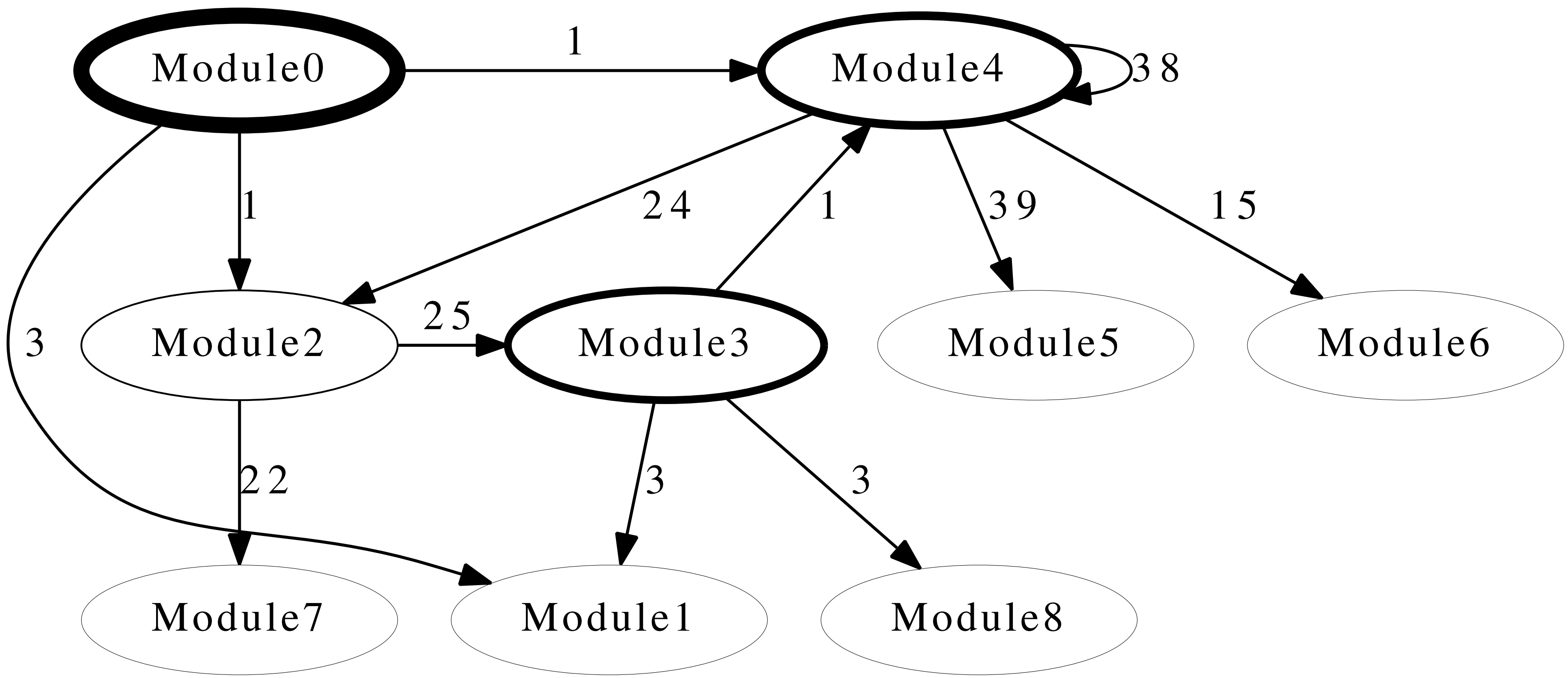
DSOAR Effort

	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

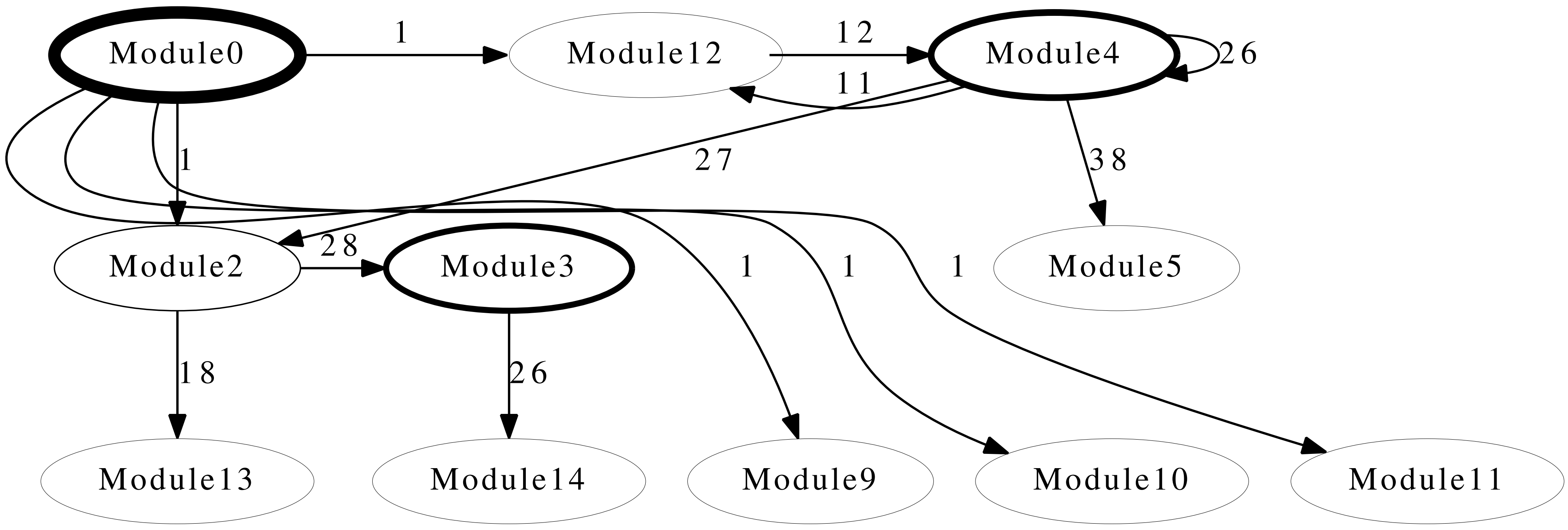
More data, source code, etc, at:

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