What's in an evolved name?

The evolution of modularity *via* tag-based reference

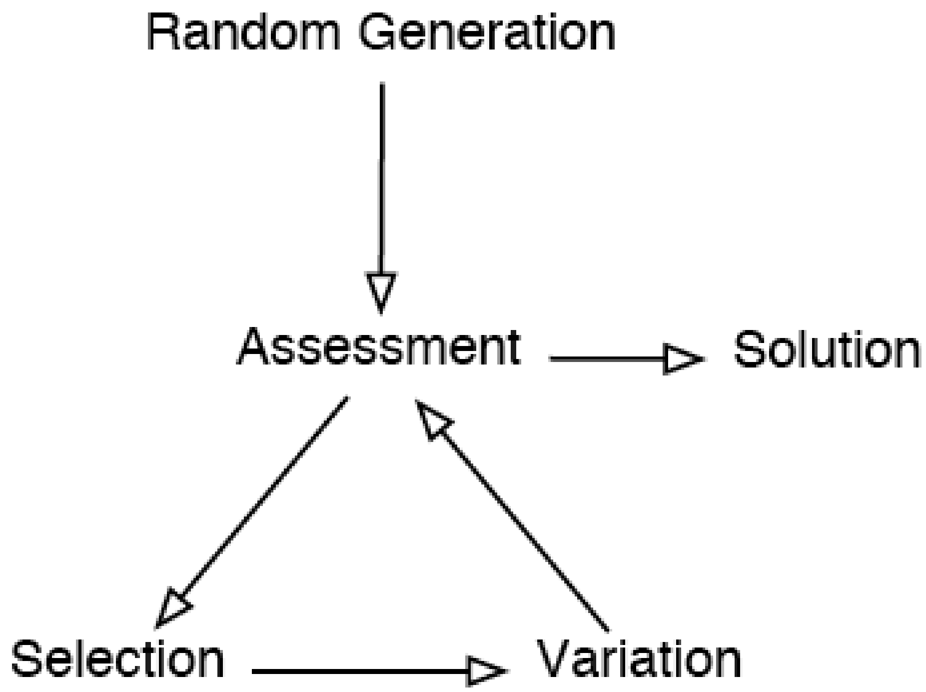
Lee Spector, Kyle Harrington, Brian Martin & Thomas Helmuth

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

Outline

- GP with expressive languages: Push
- Modularity in GP
- Tags
- Tag-based modularity in GP

Evolutionary Computation



Genetic Programming

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

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, ...).

Expressive Languages

- Multiple data types
- User-defined procedures & functions
- User-defined macros & control structures
- User-defined representations
- Dynamic definition & redefinition

Expressive Languages

- Multiple data types
- User-defined procedures & functions
- User-defined macros & control structures
- User-defined representations
- Dynamic definition & redefinition
- Push provides all of the above and more, all without any mechanisms beyond the stack-based execution architecture

Types

- Most useful programs manipulate multiple data types.
- Single type or multiple type closures.
- Strongly typed genetic programming: constraints on code generation and genetic operators (Montana).
- Polymorphism (Yu and Clack).
- Stack-based GP with typed stacks (Spector).

Modules

- Automatically-defined functions (Koza)
- Automatically-defined macros (Spector)
- Architecture-altering operations (Koza)
- Module acquisition/encapsulation systems (Kinnear, Roberts, many others)
- Push approach: instructions that can build/execute modules with no changes to the system's representations or algorithms

We will return to this later!

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*)

Sample Push Instructions

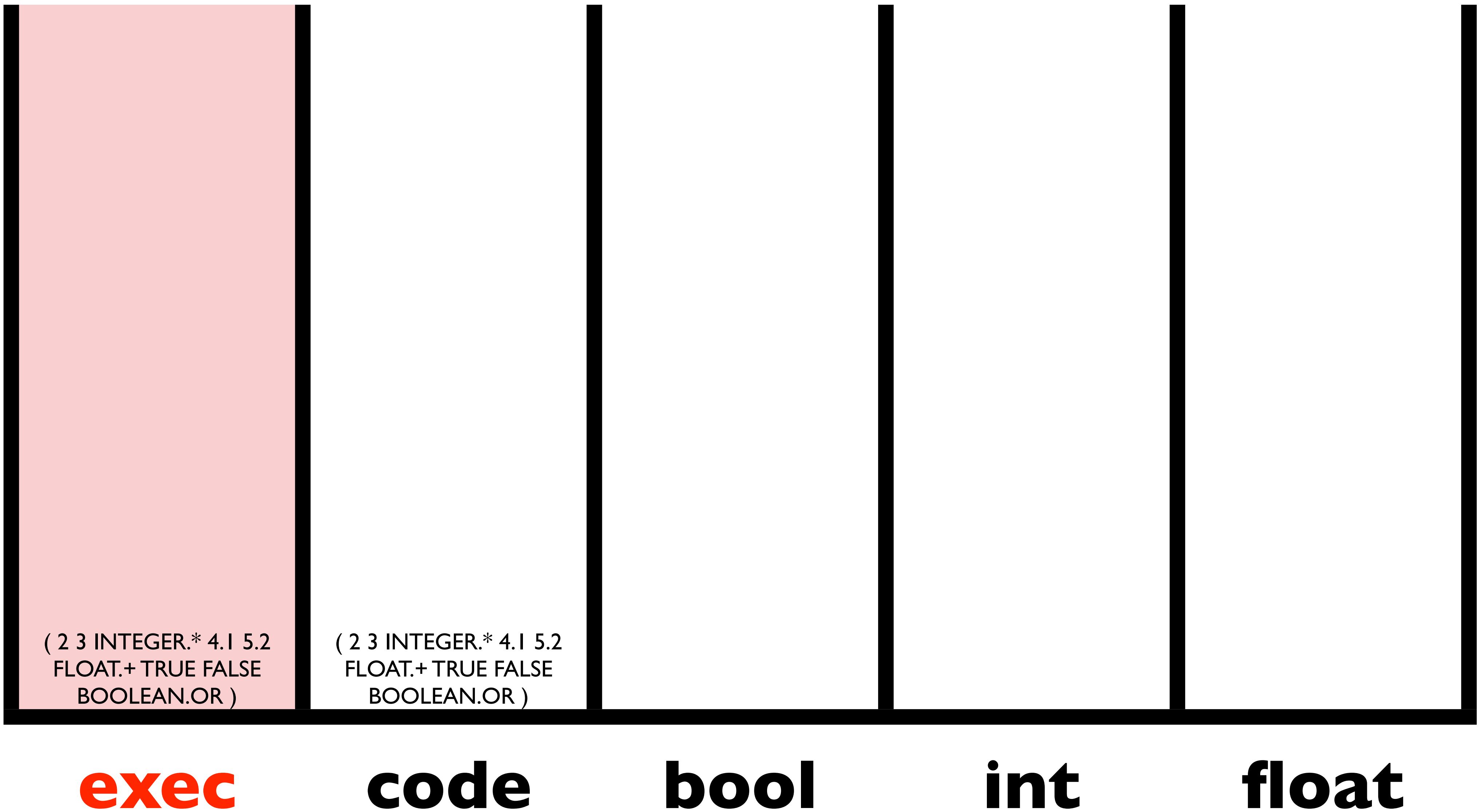
Stack manipulation instructions (all types)	POP, SWAP, YANK, DUP, STACKDEPTH, SHOVE, FLUSH, =
Math (INTEGER and FLOAT)	+, −, /, *, >, <, MIN, MAX
Logic (BOOLEAN)	AND, OR, NOT, FROMINTEGER
Code manipulation (CODE)	QUOTE, CAR, CDR, CONS, INSERT, LENGTH, LIST, MEMBER, NTH, EXTRACT
Control manipulation (CODE and EXEC)	DO*, DO*COUNT, DO*RANGE, DO*TIMES, IF

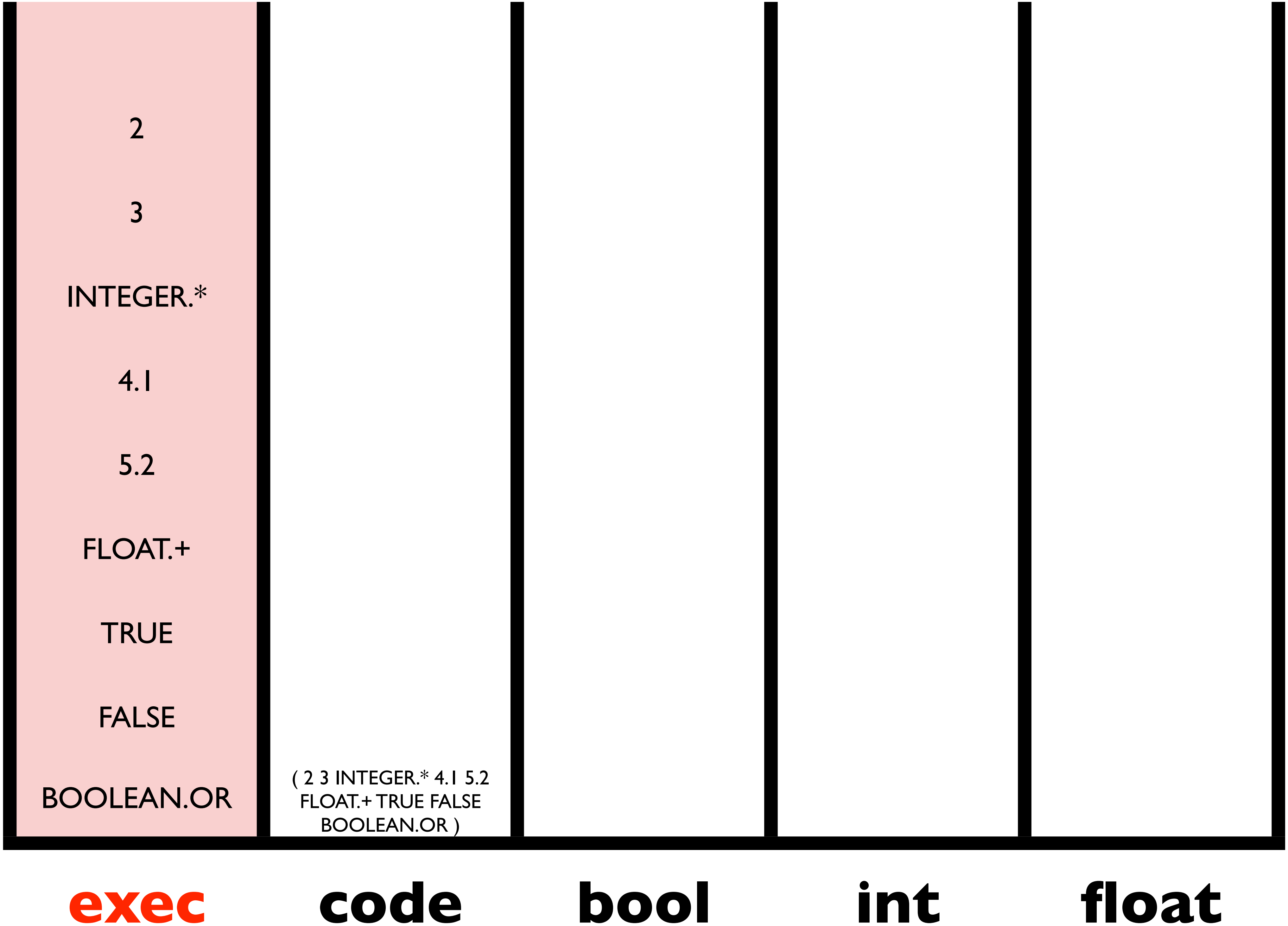
Push(3) Semantics

- To execute program P:

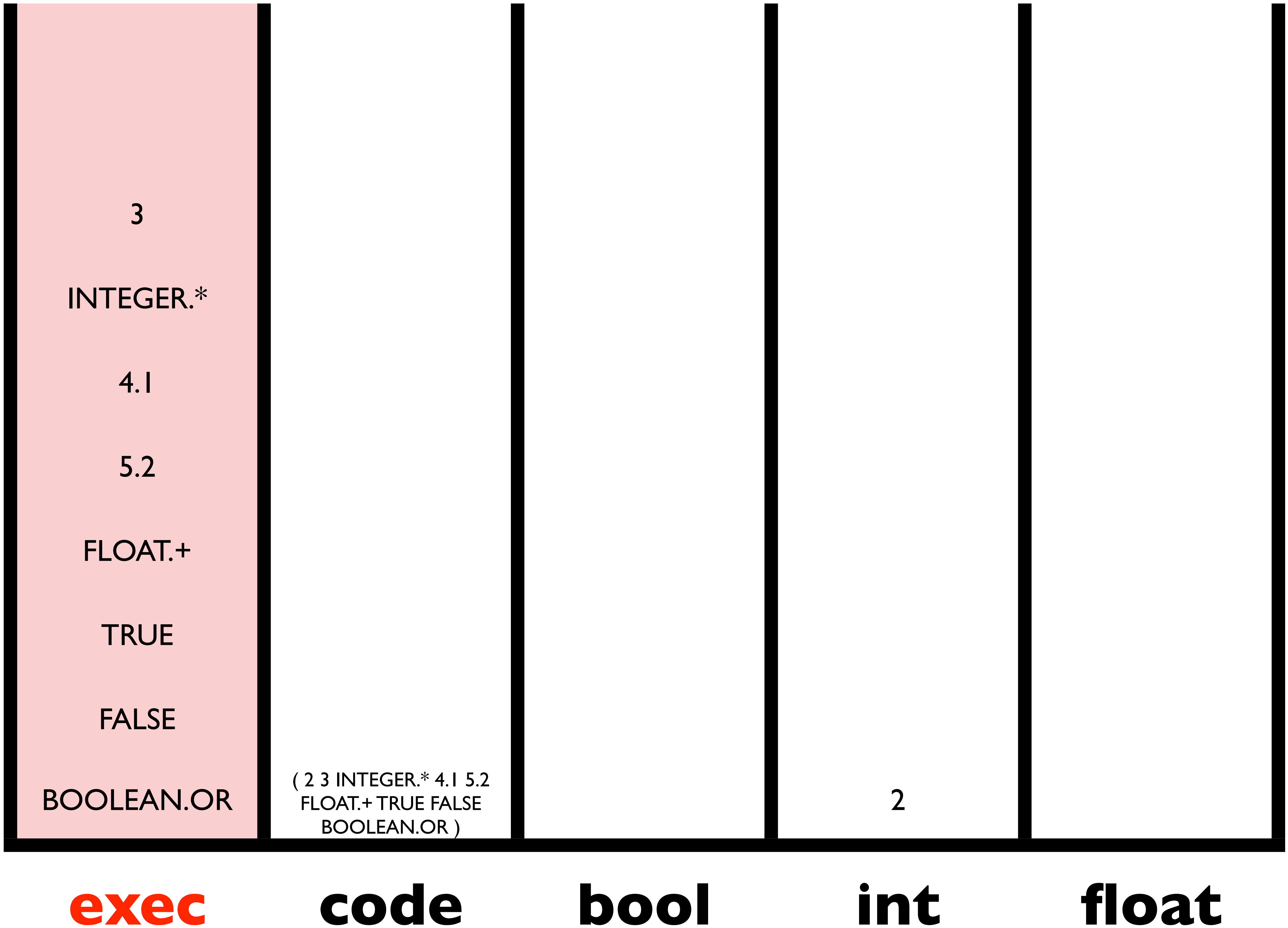
  1. Push P onto the `EXEC` stack.
  2. While the `EXEC` stack is not empty, pop and process the top element of the `EXEC` stack, E:
     (a) If E is an instruction: execute E (accessing whatever stacks are required).
     (b) If E is a literal: push E onto the appropriate stack.
     (c) If E is a list: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

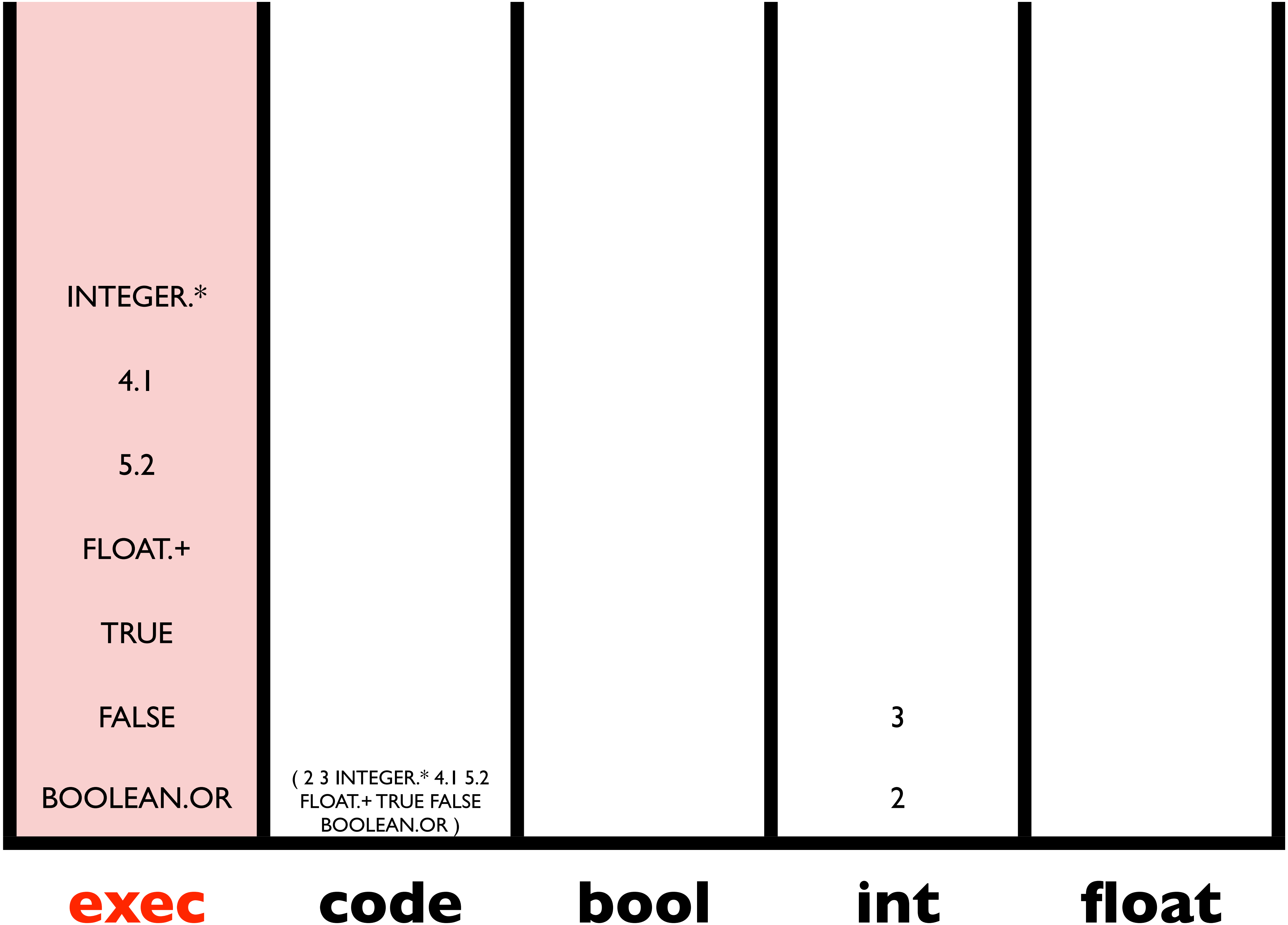
```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+
TRUE FALSE BOOLEAN.OR )
```

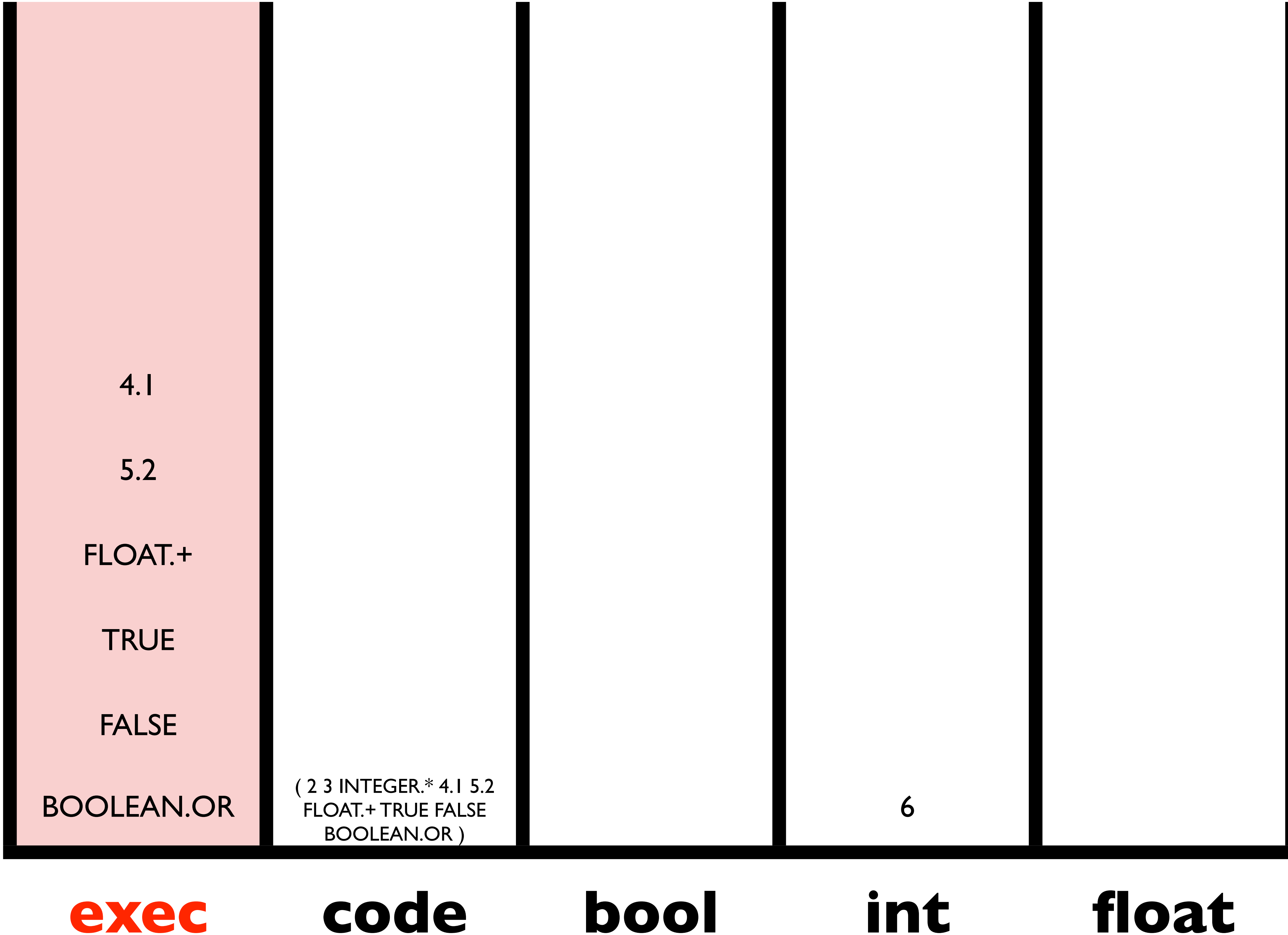




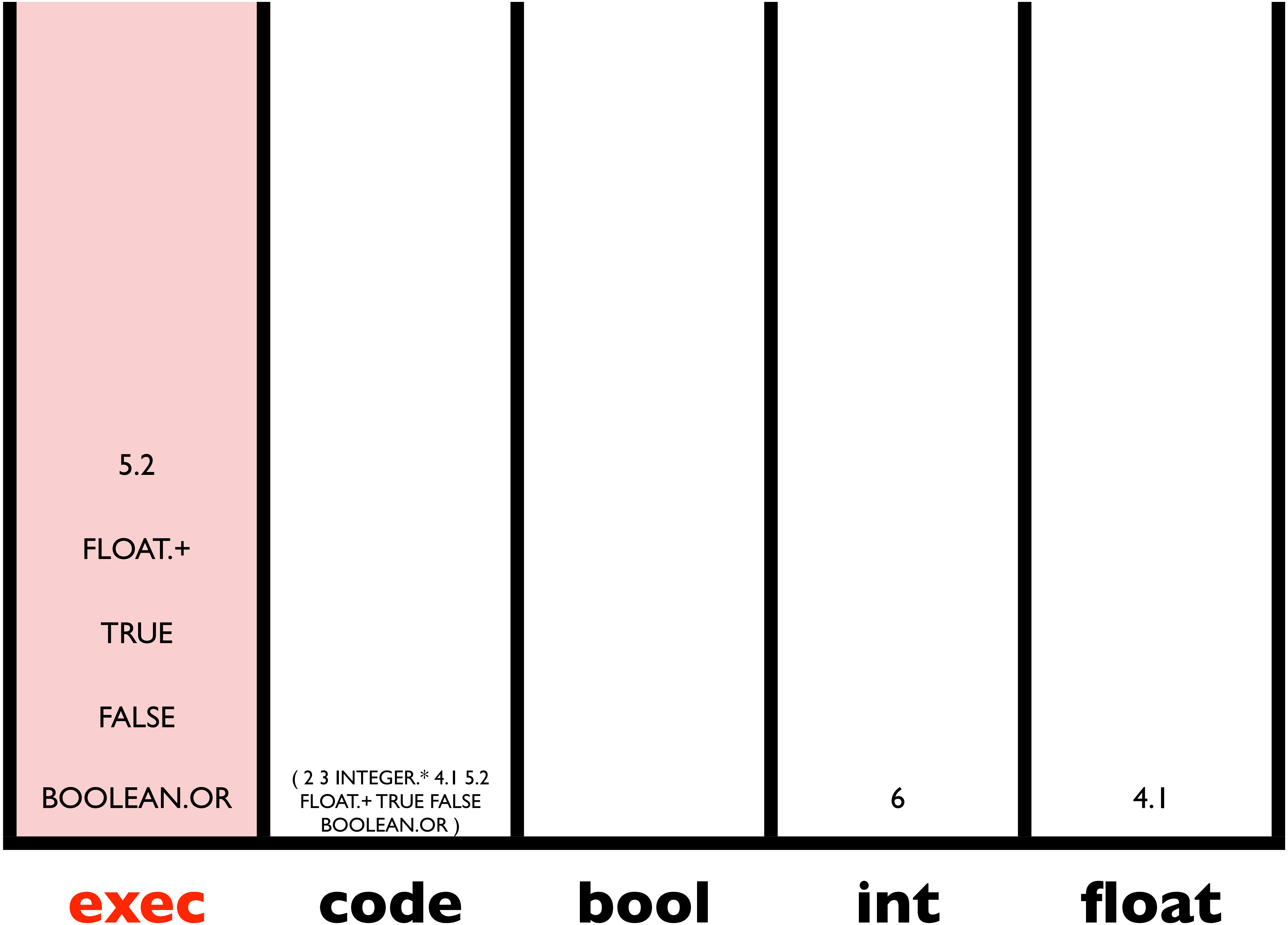
2
3
INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
exec
code
bool
int
float


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

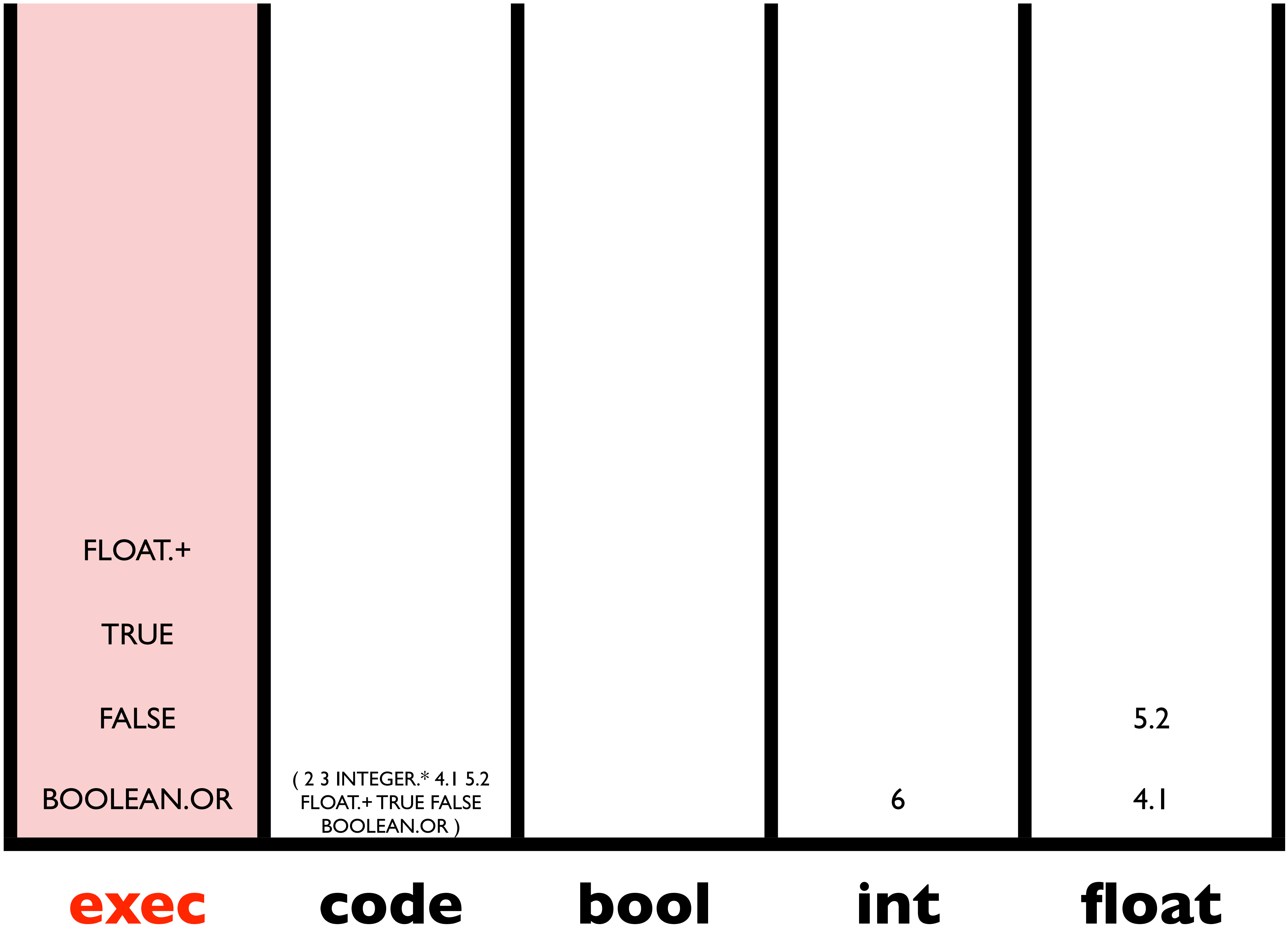

INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
3
2
exec
code
bool
int
float

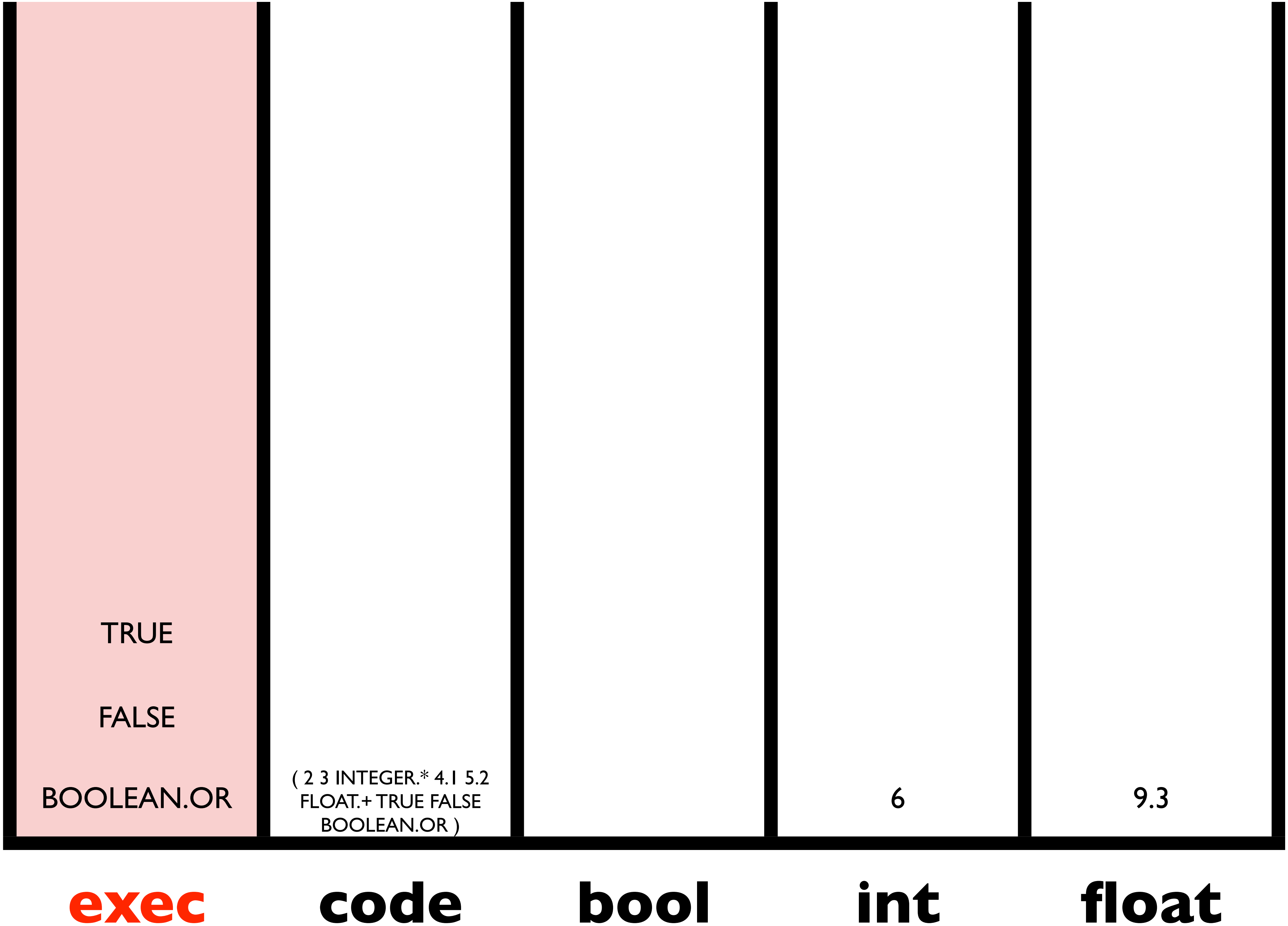


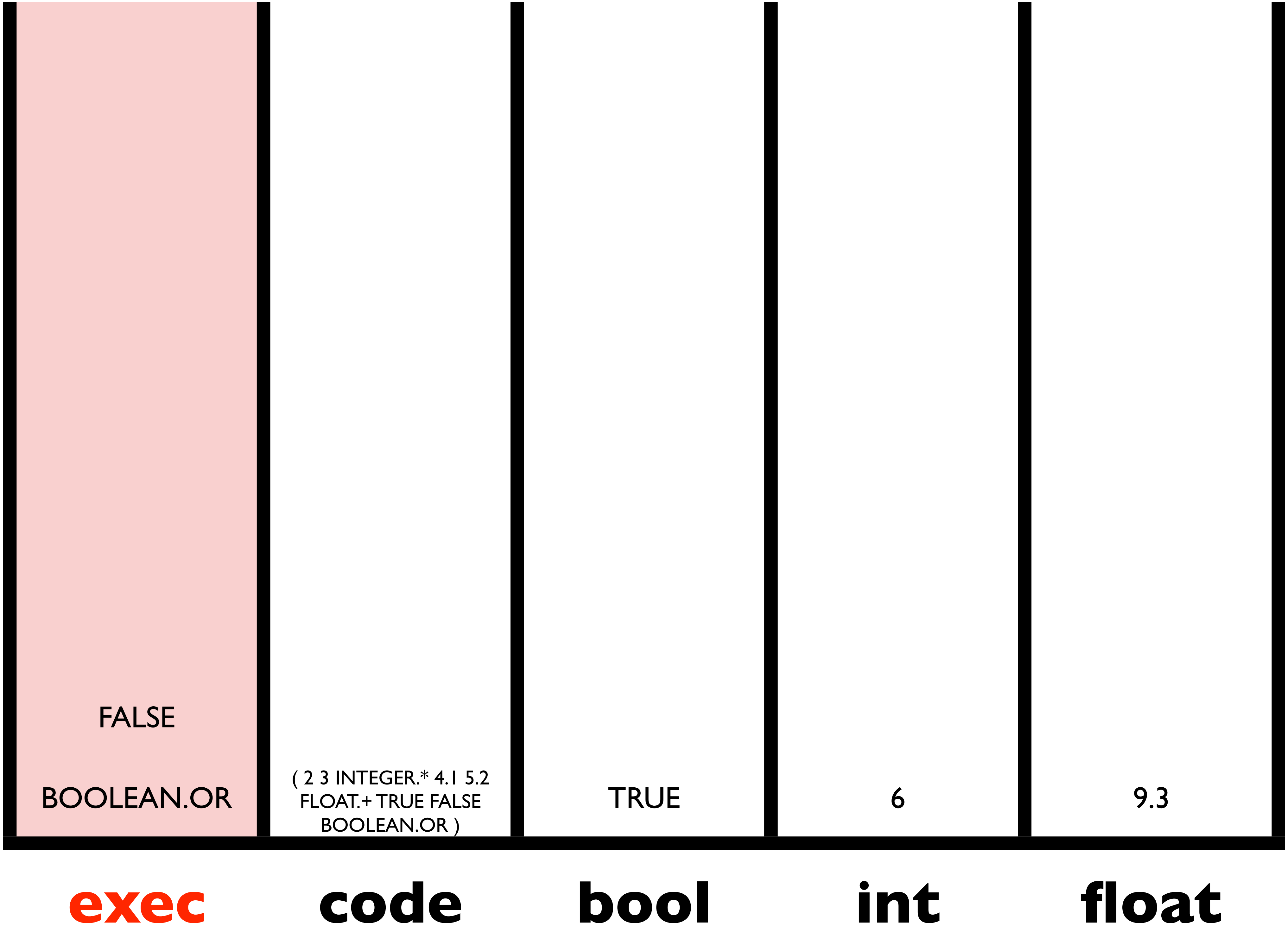
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
6
exec
code
bool
int
float

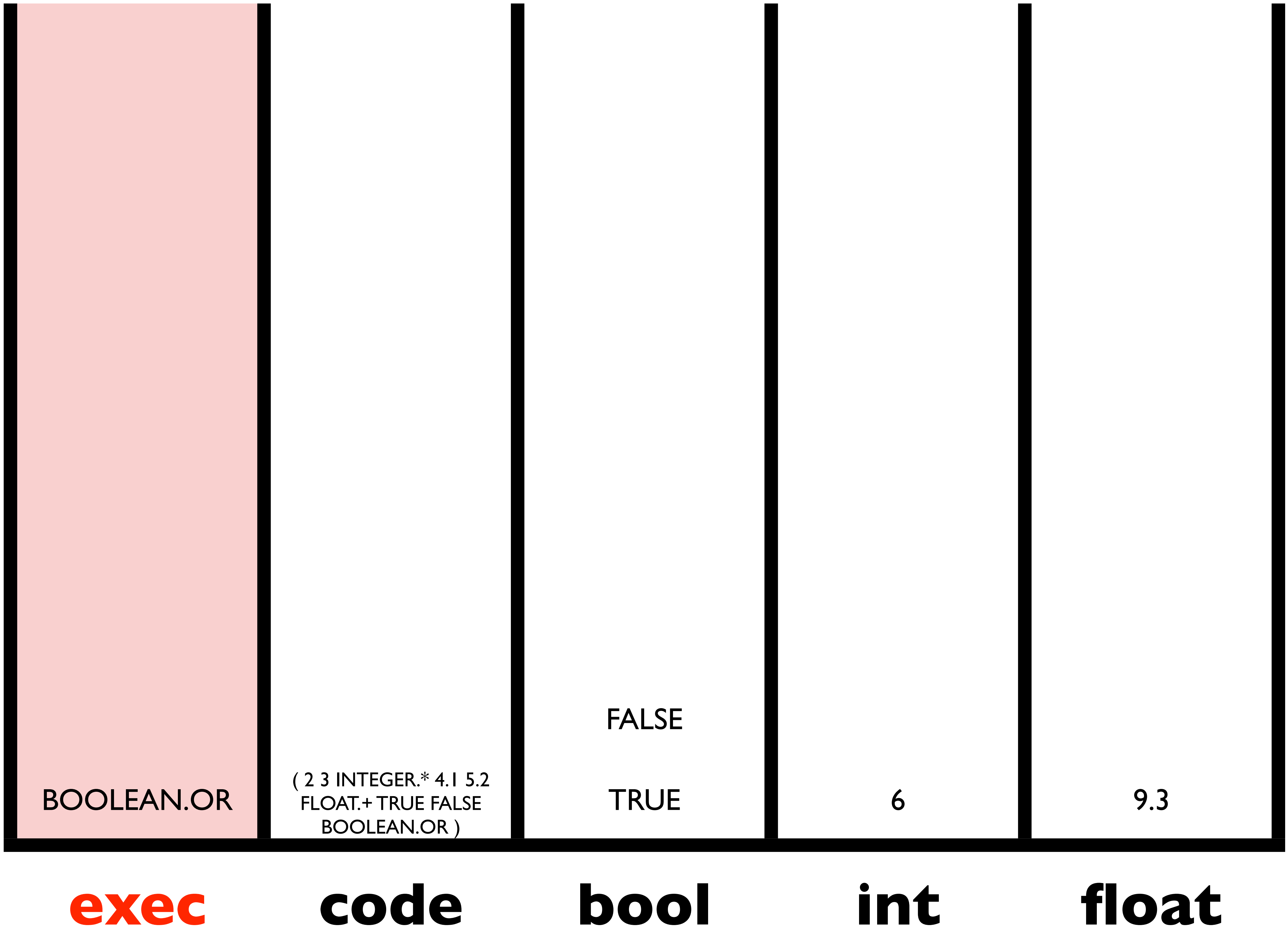


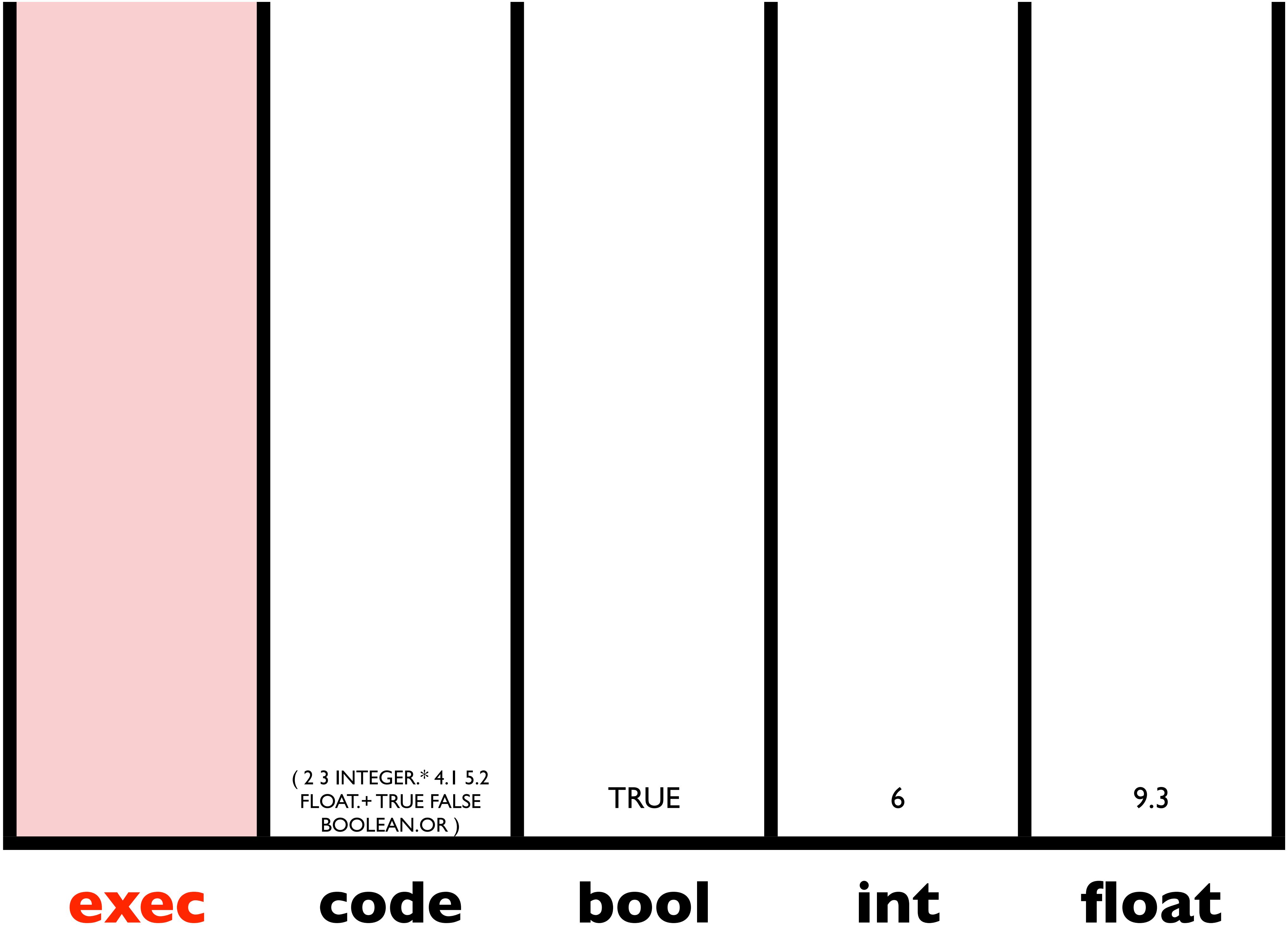
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
6
4.1
exec
code
bool
int
float


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

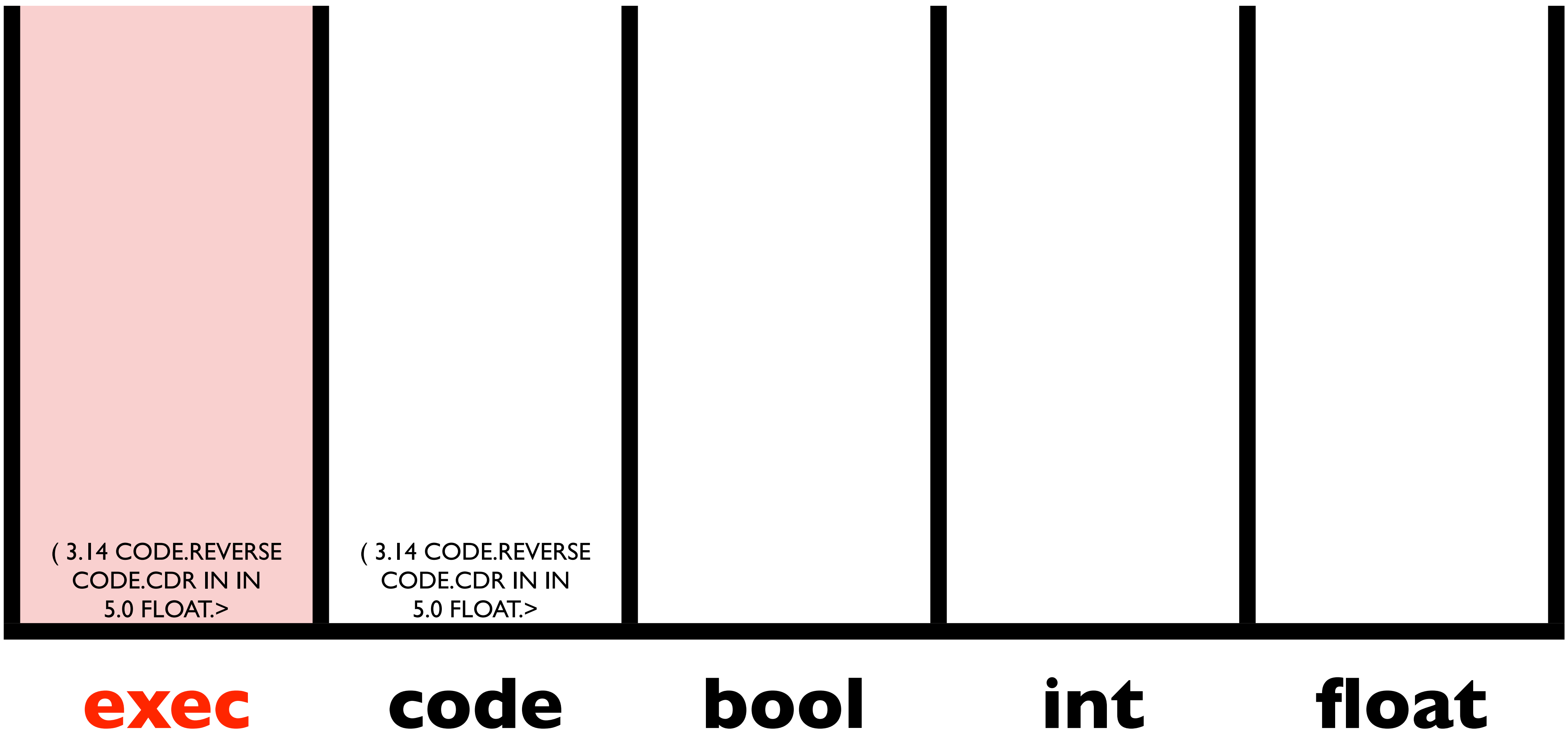
Same Results

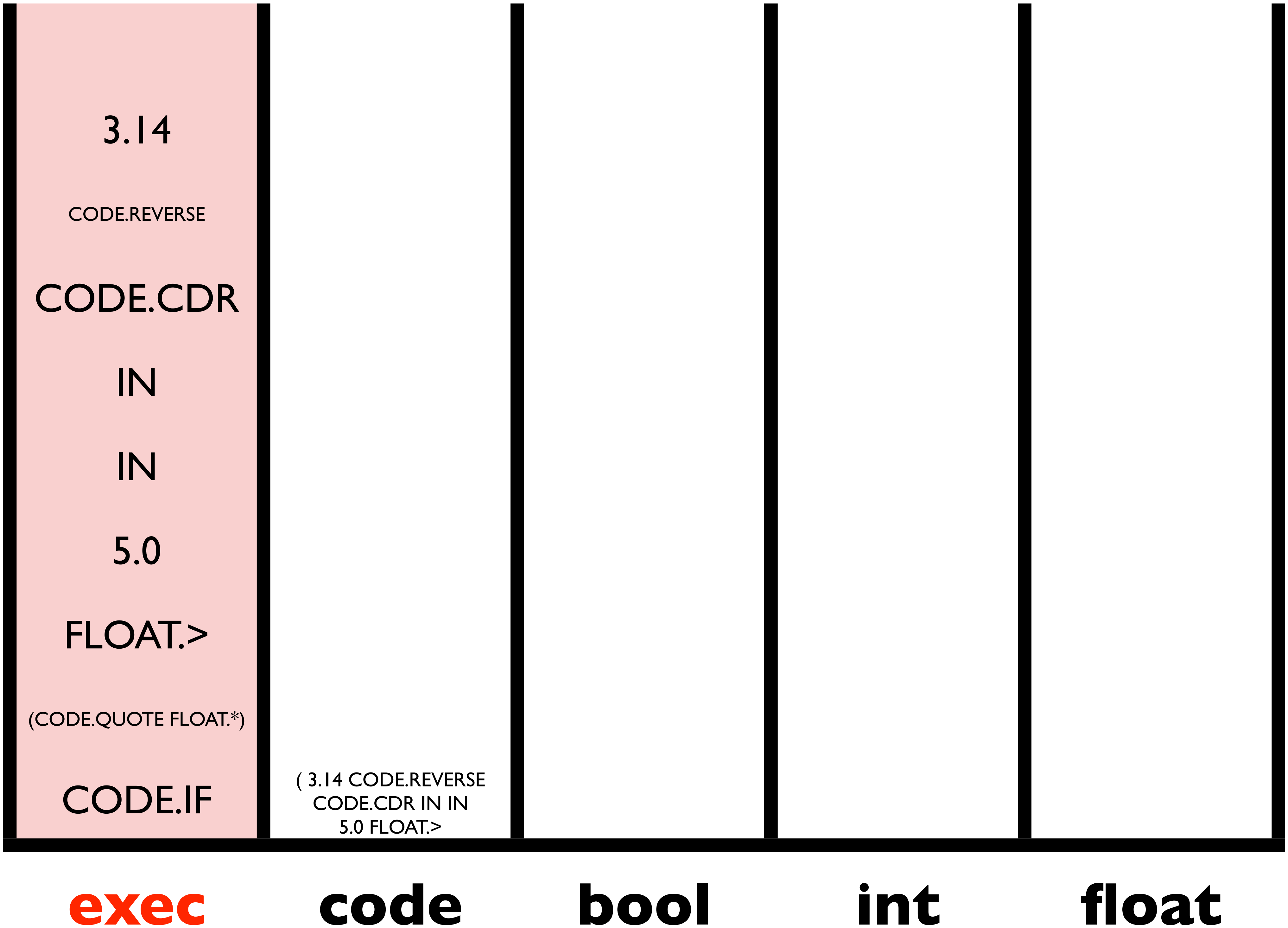
```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+
  TRUE FALSE BOOLEAN.OR )
```

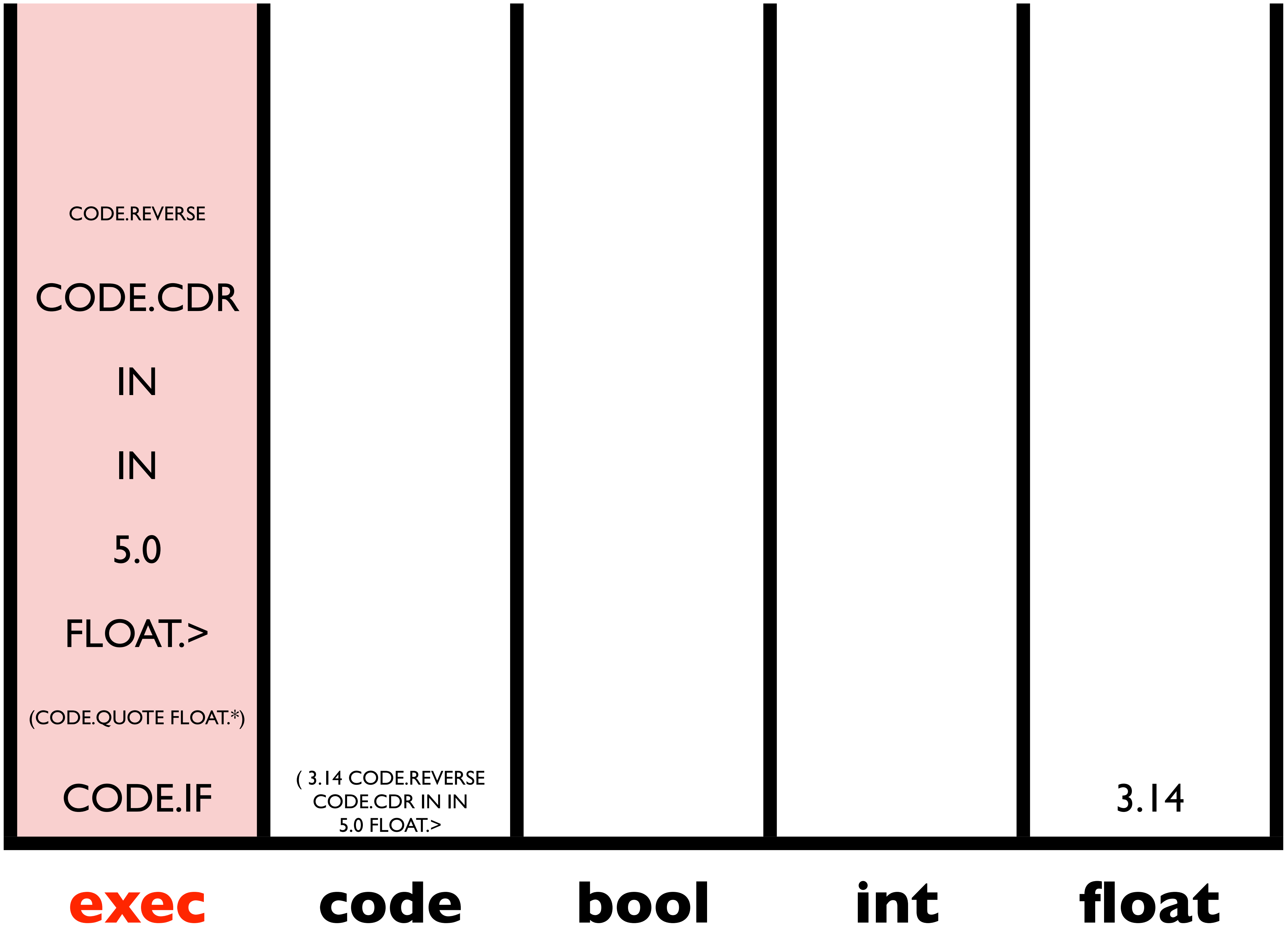
```
( 2 BOOLEAN.AND 4.1 TRUE INTEGER./ FALSE
  3 5.2 BOOLEAN.OR INTEGER.* FLOAT.+ )
```

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

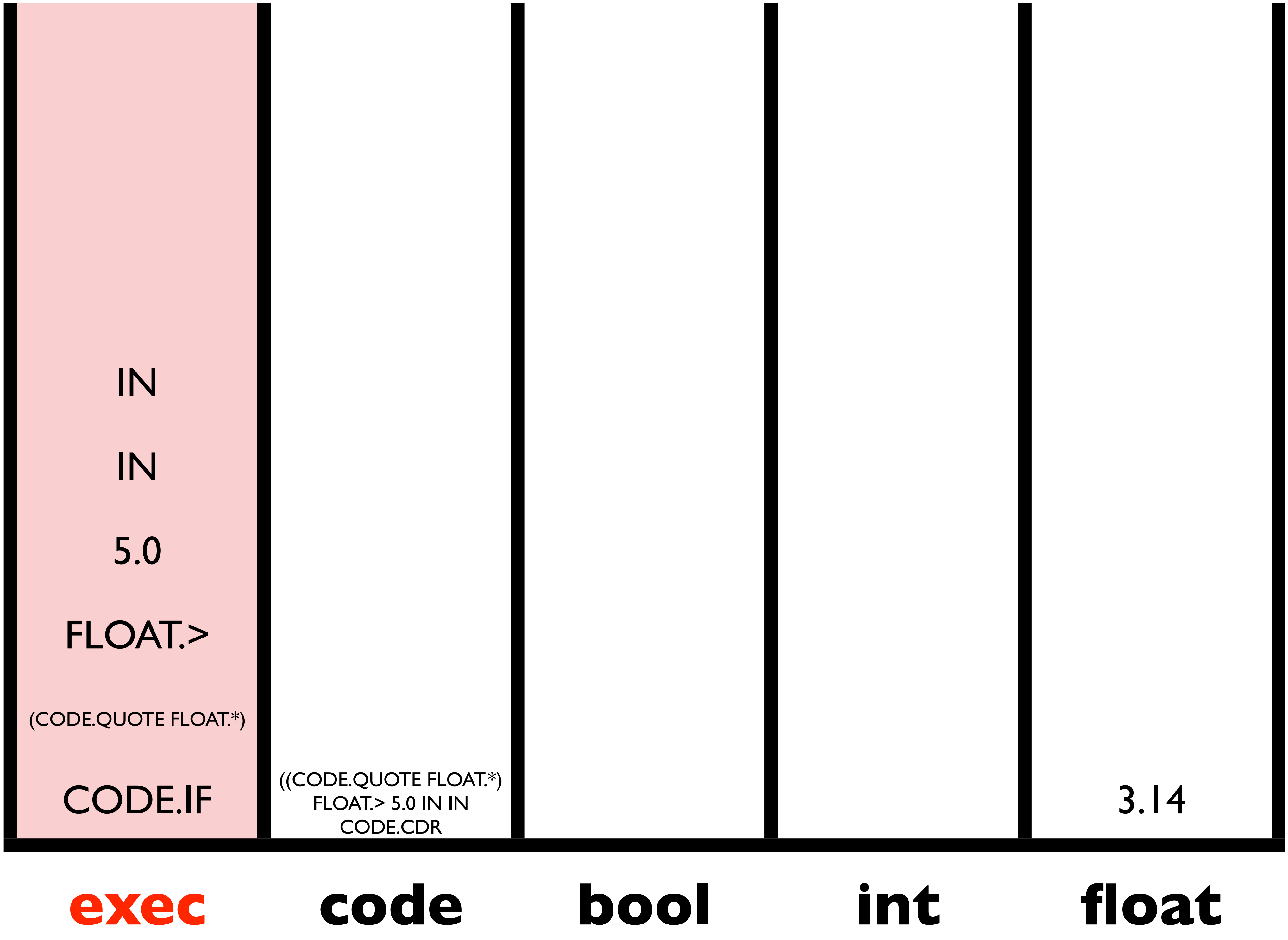
IN=4.0



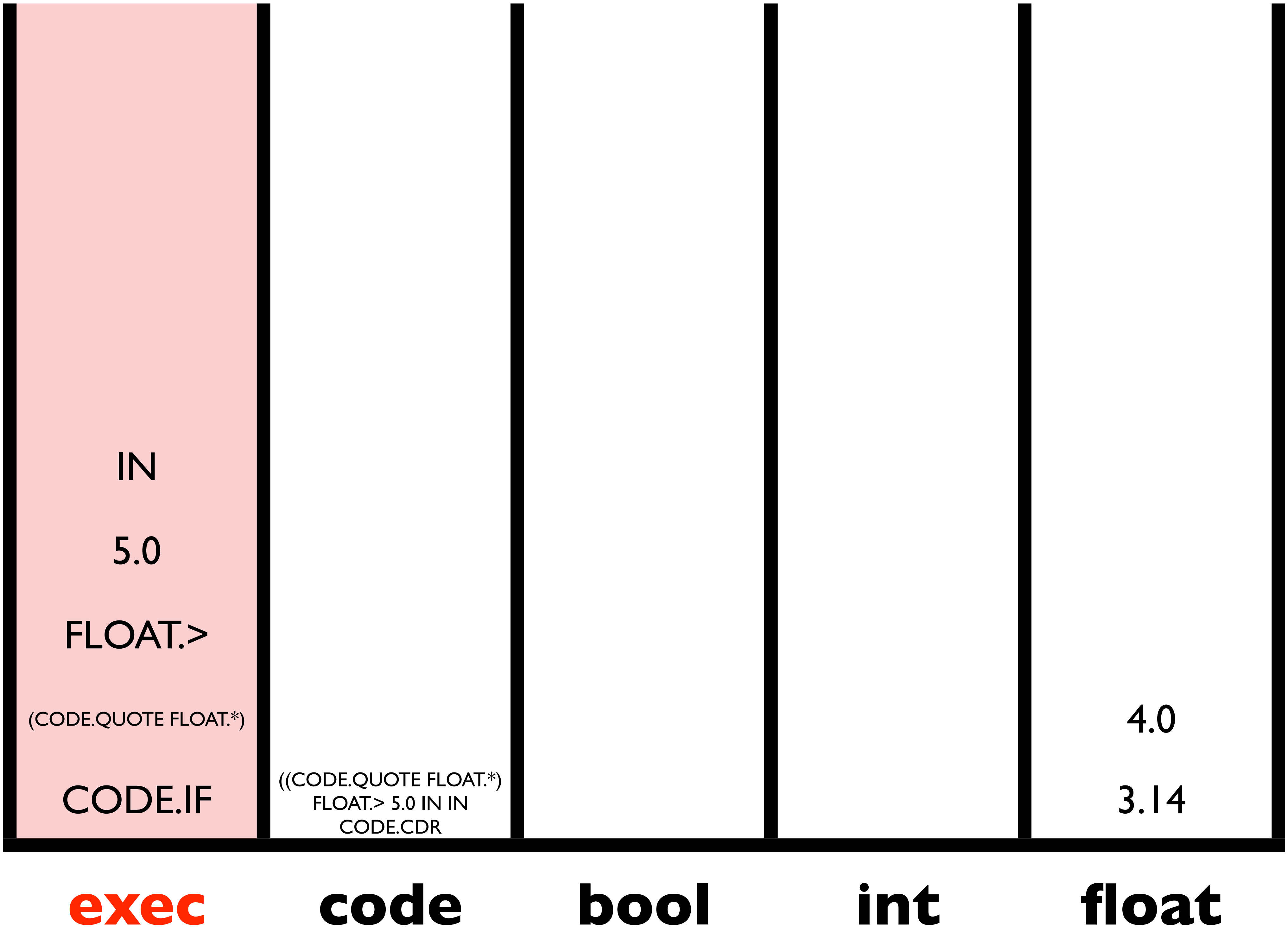

3.14
CODE.REVERSE
CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
(3.14 CODE.REVERSE
CODE.CDR IN IN
5.0 FLOAT.>
exec
code
bool
int
float

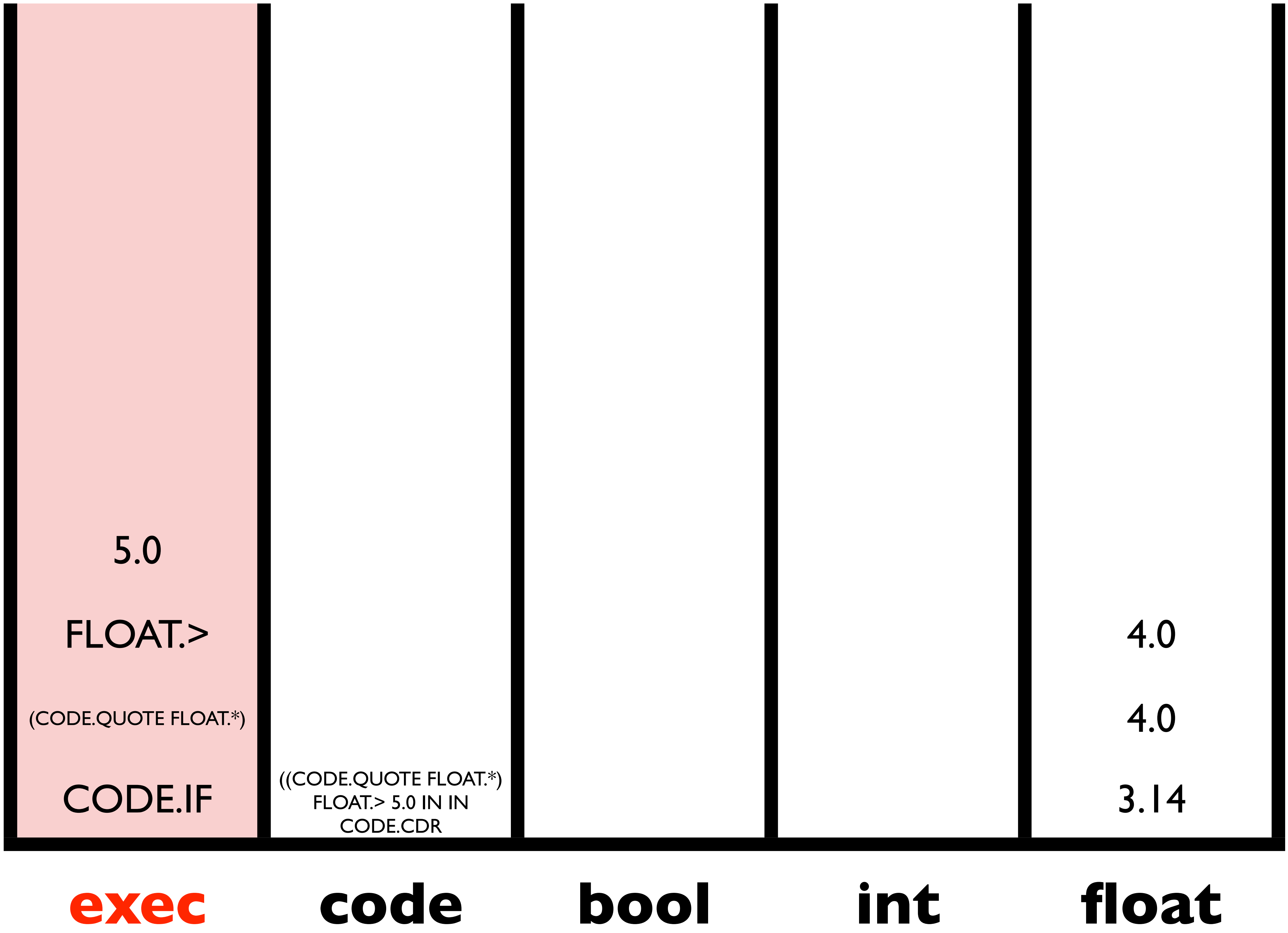

CODE.REVERSE
CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
(3.14 CODE.REVERSE
CODE.CDR IN IN
5.0 FLOAT.>
3.14
exec
code
bool
int
float

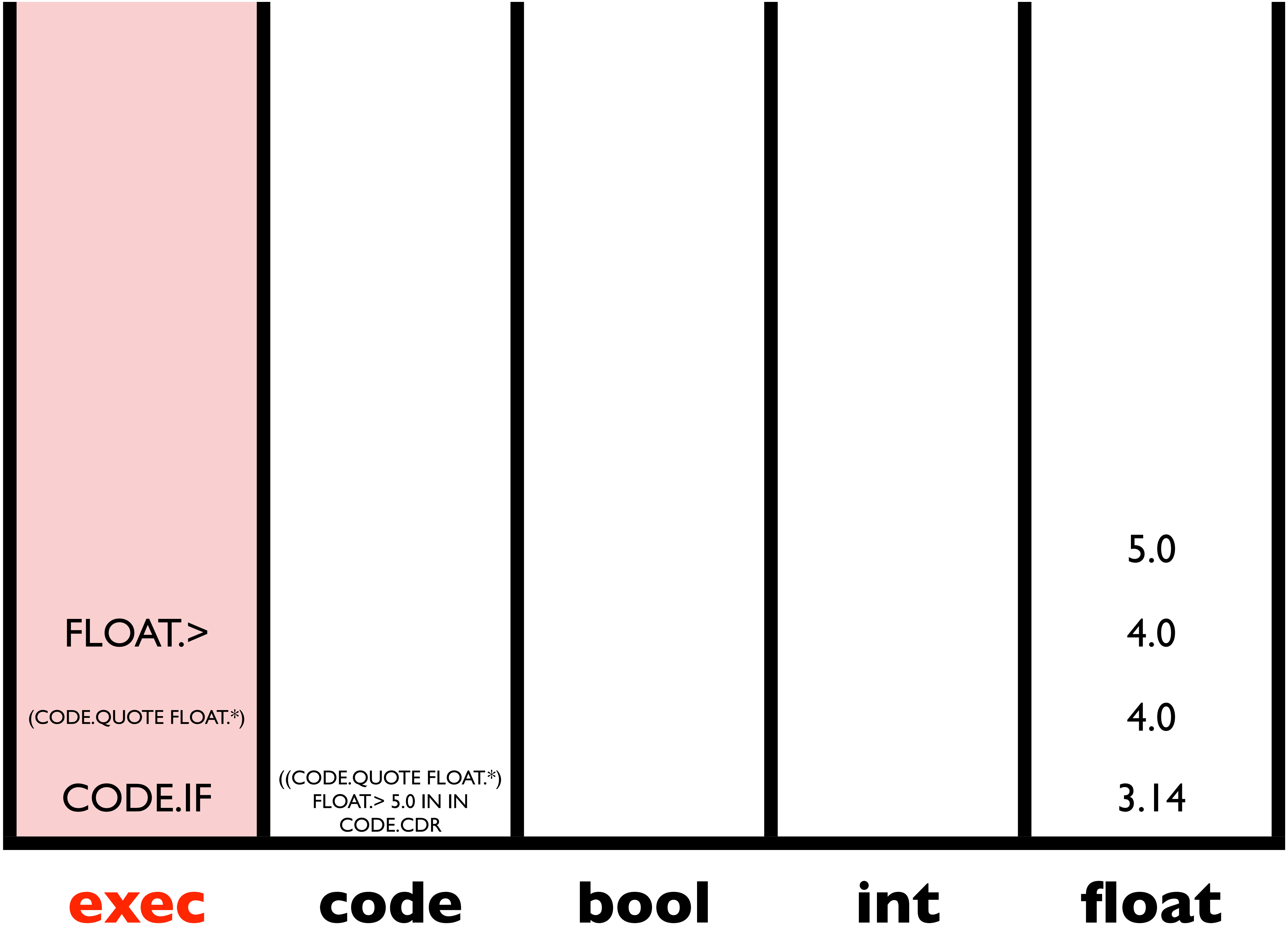
exec	code	bool	int	float
CODE.CDR				
IN				
IN				
5.0				
FLOAT.>				
(CODE.QUOTE FLOAT.*)				
CODE.IF	(CODE.IF (CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR			3.14

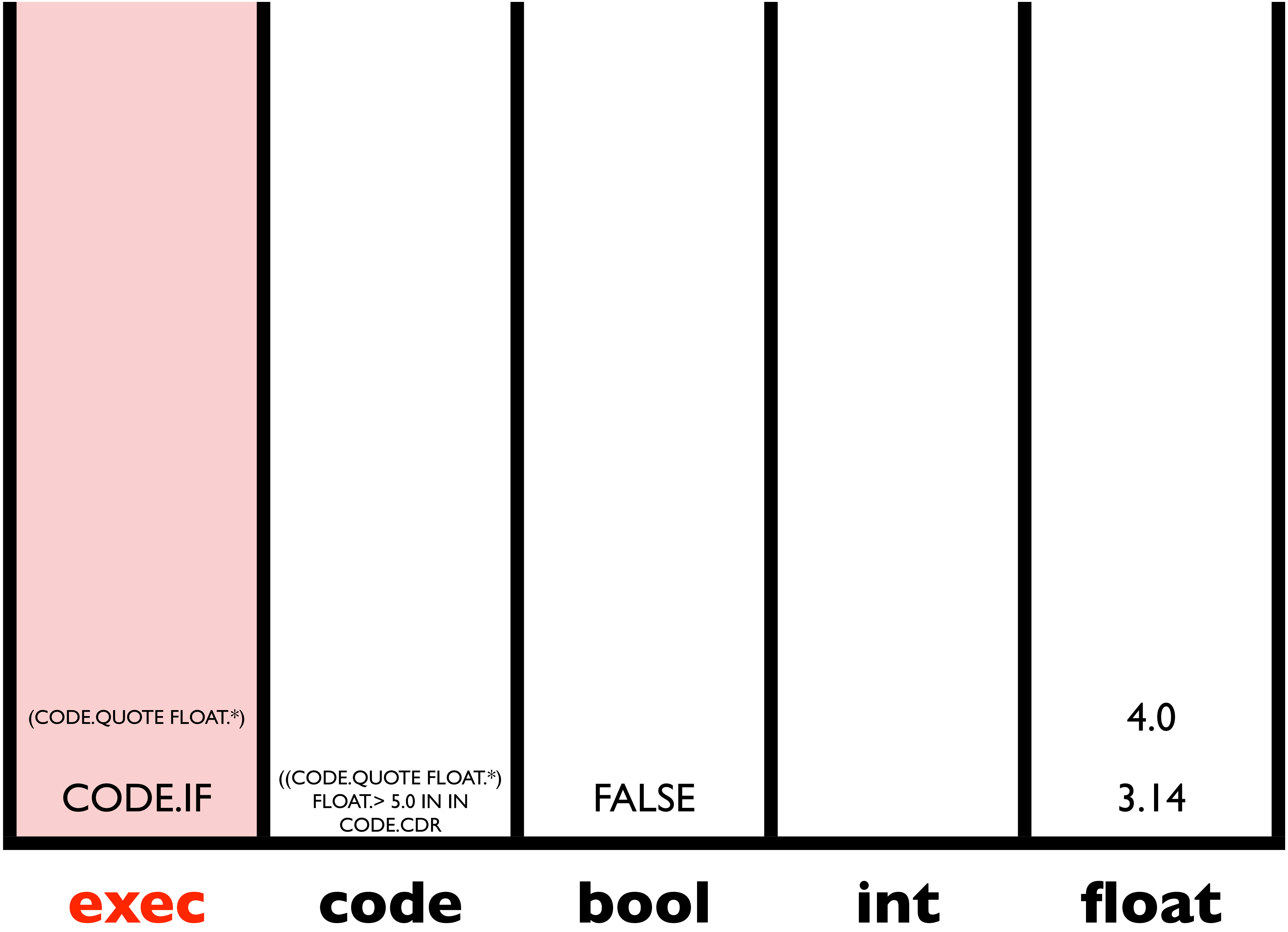


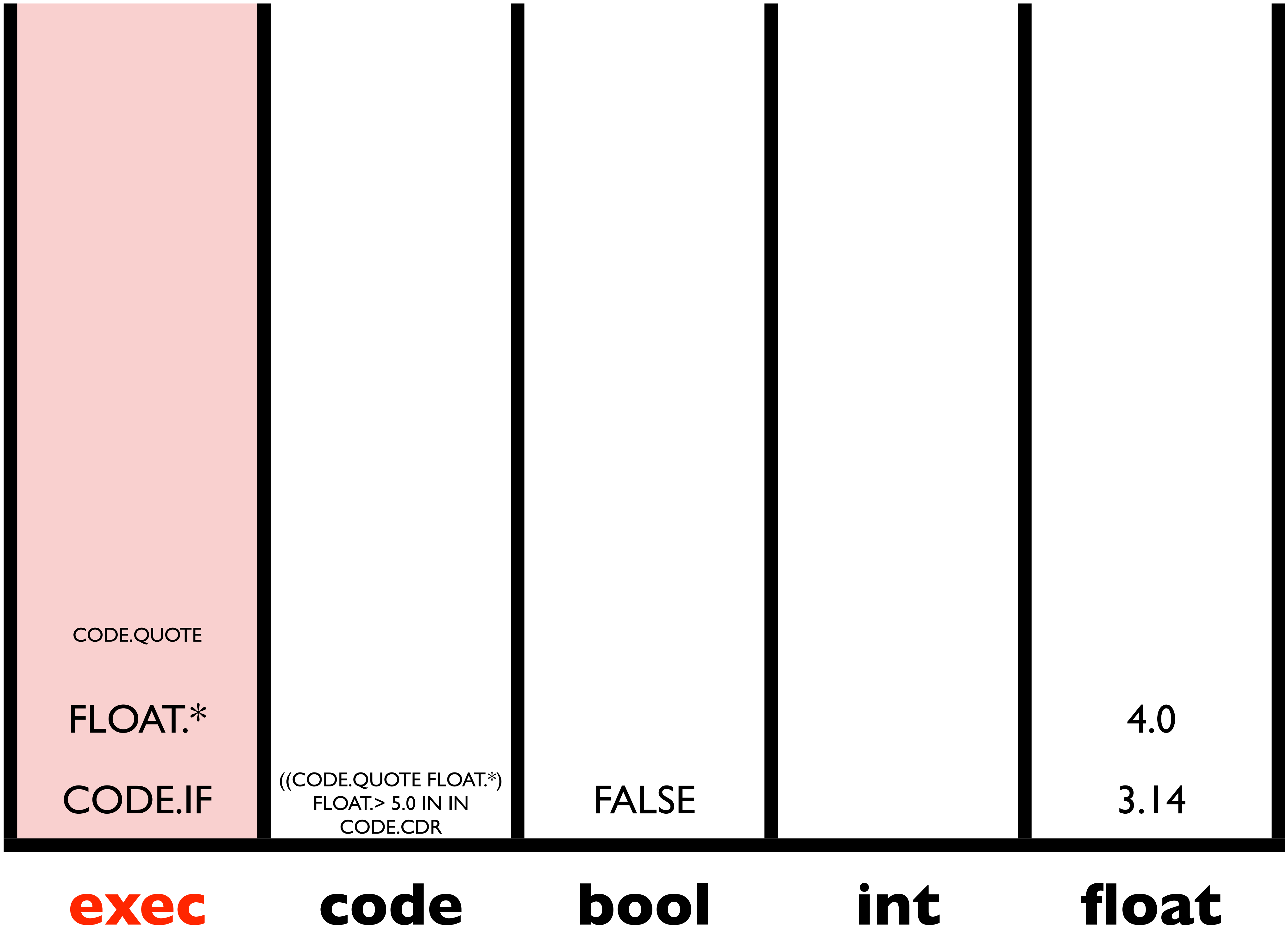
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
3.14
exec
code
bool
int
float

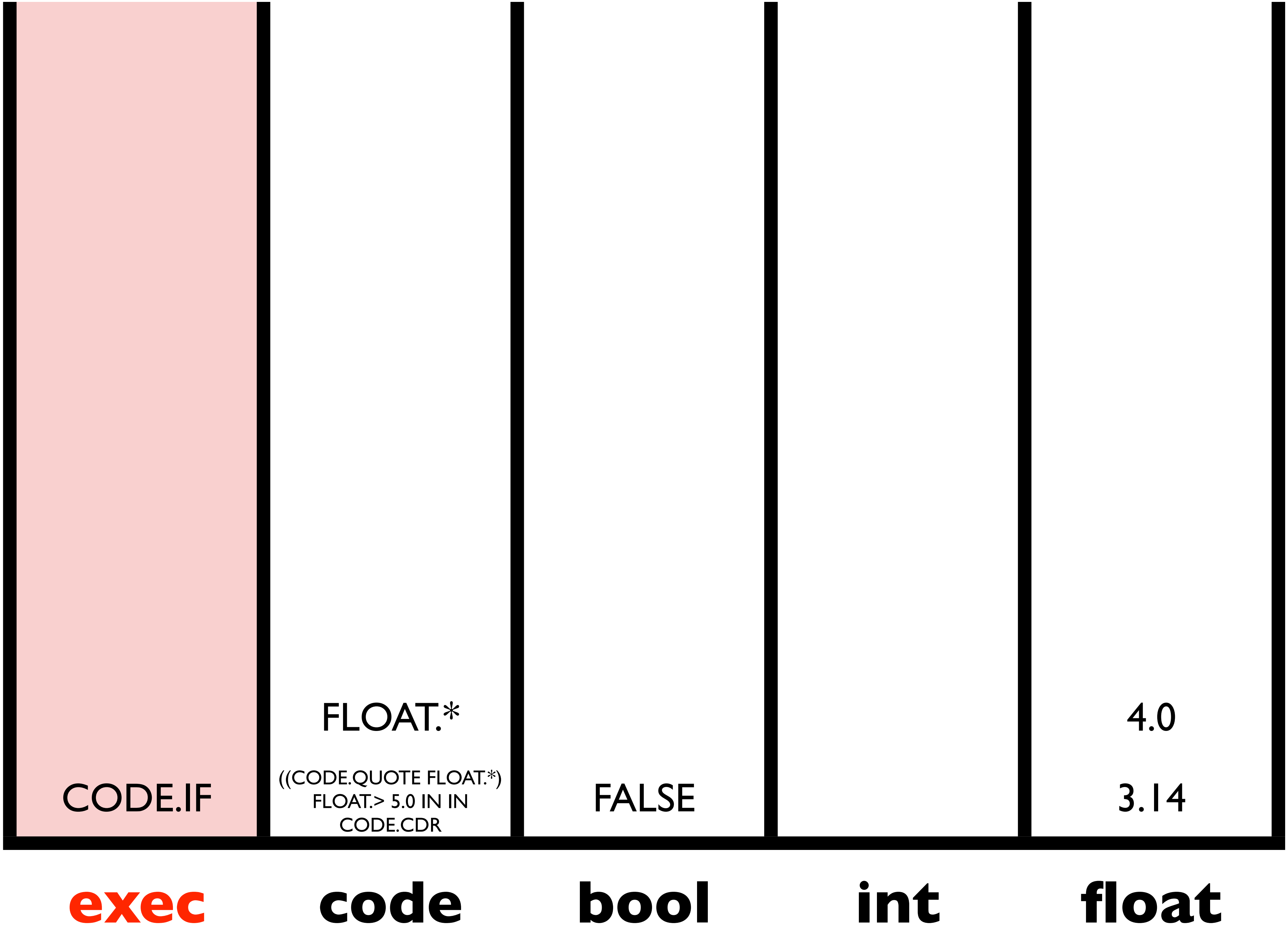

IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
4.0
3.14
exec
code
bool
int
float

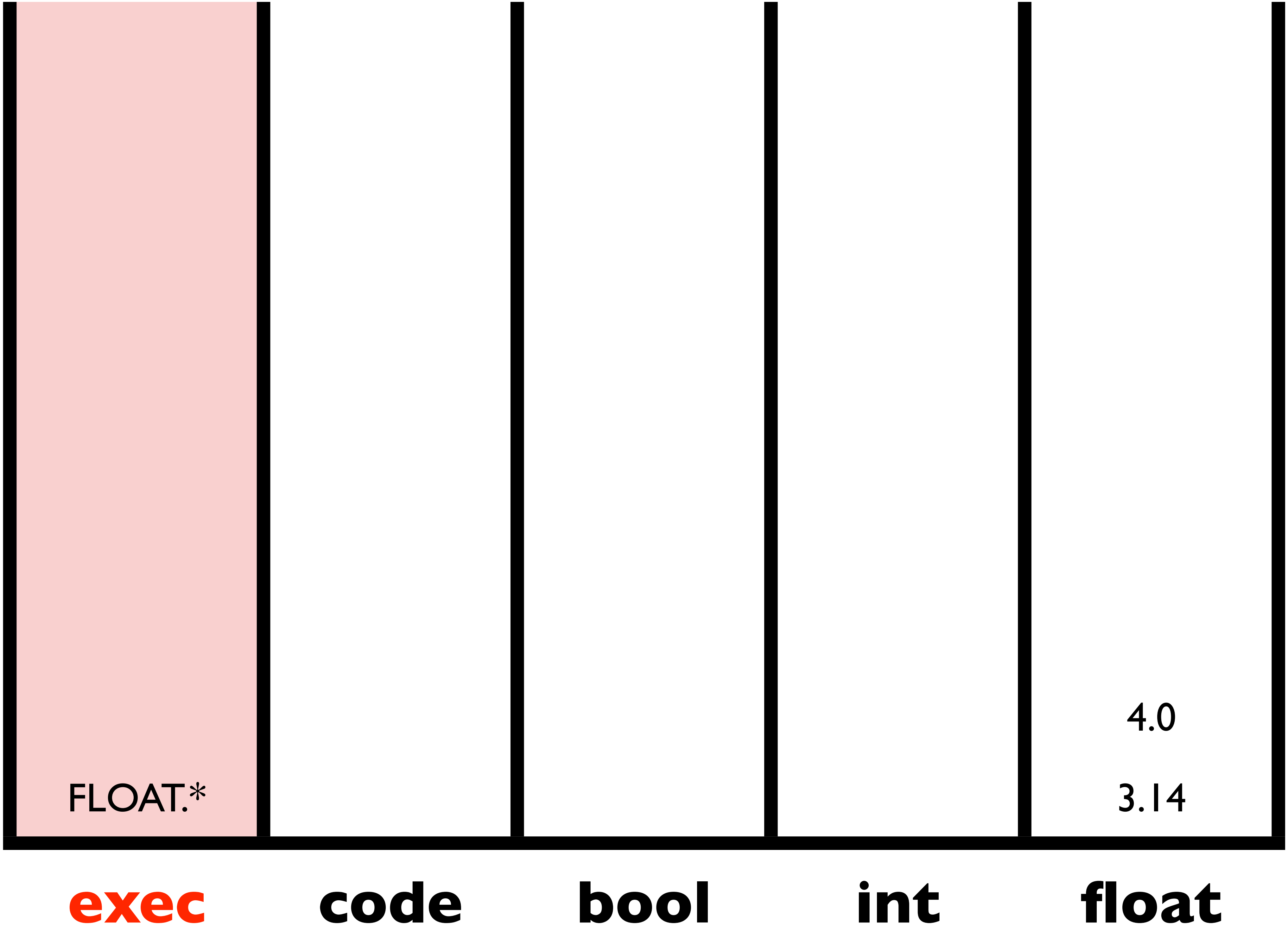

5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
4.0
4.0
3.14
exec
code
bool
int
float


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

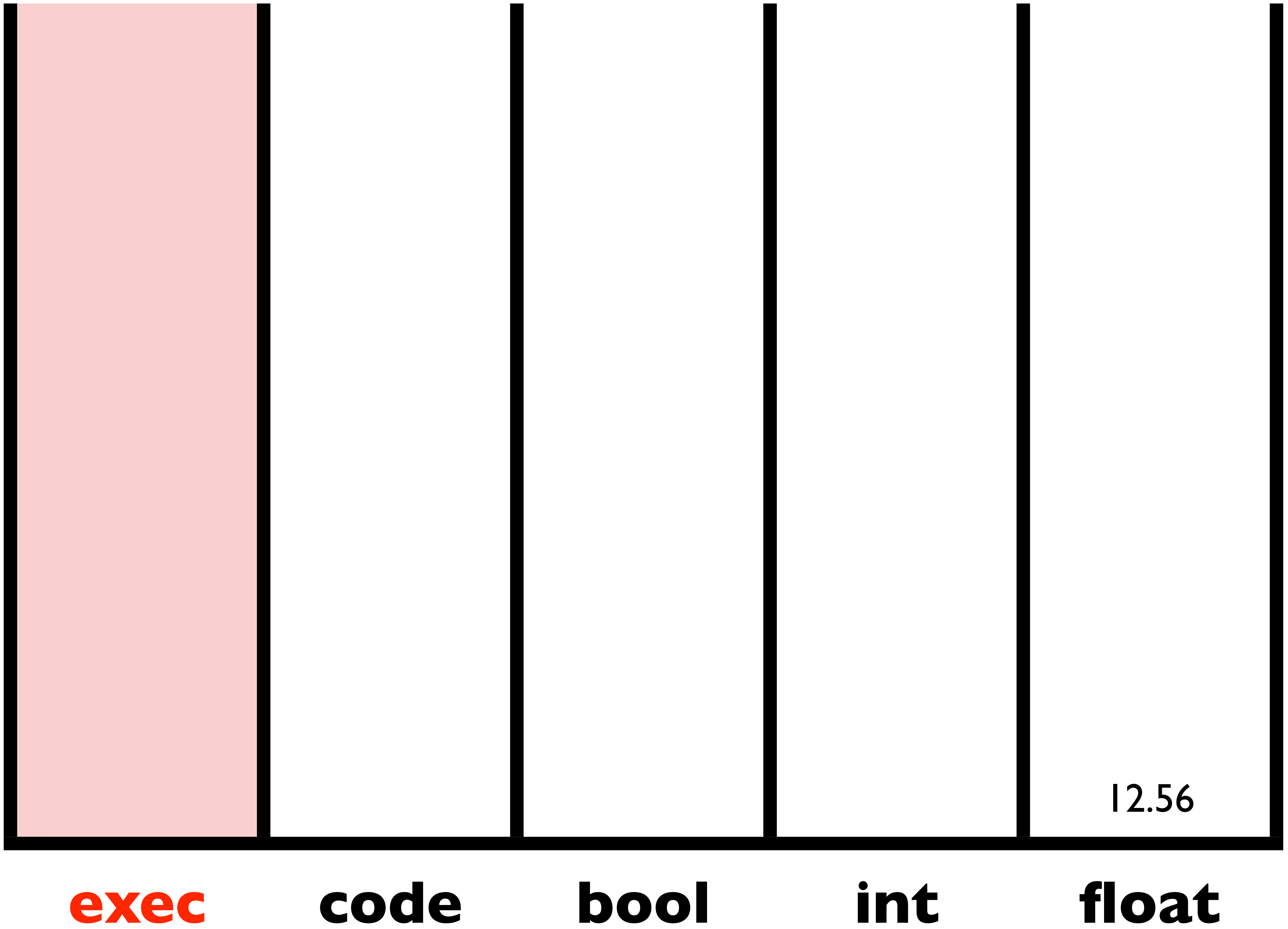

(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
FALSE
4.0
3.14
exec
code
bool
int
float


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

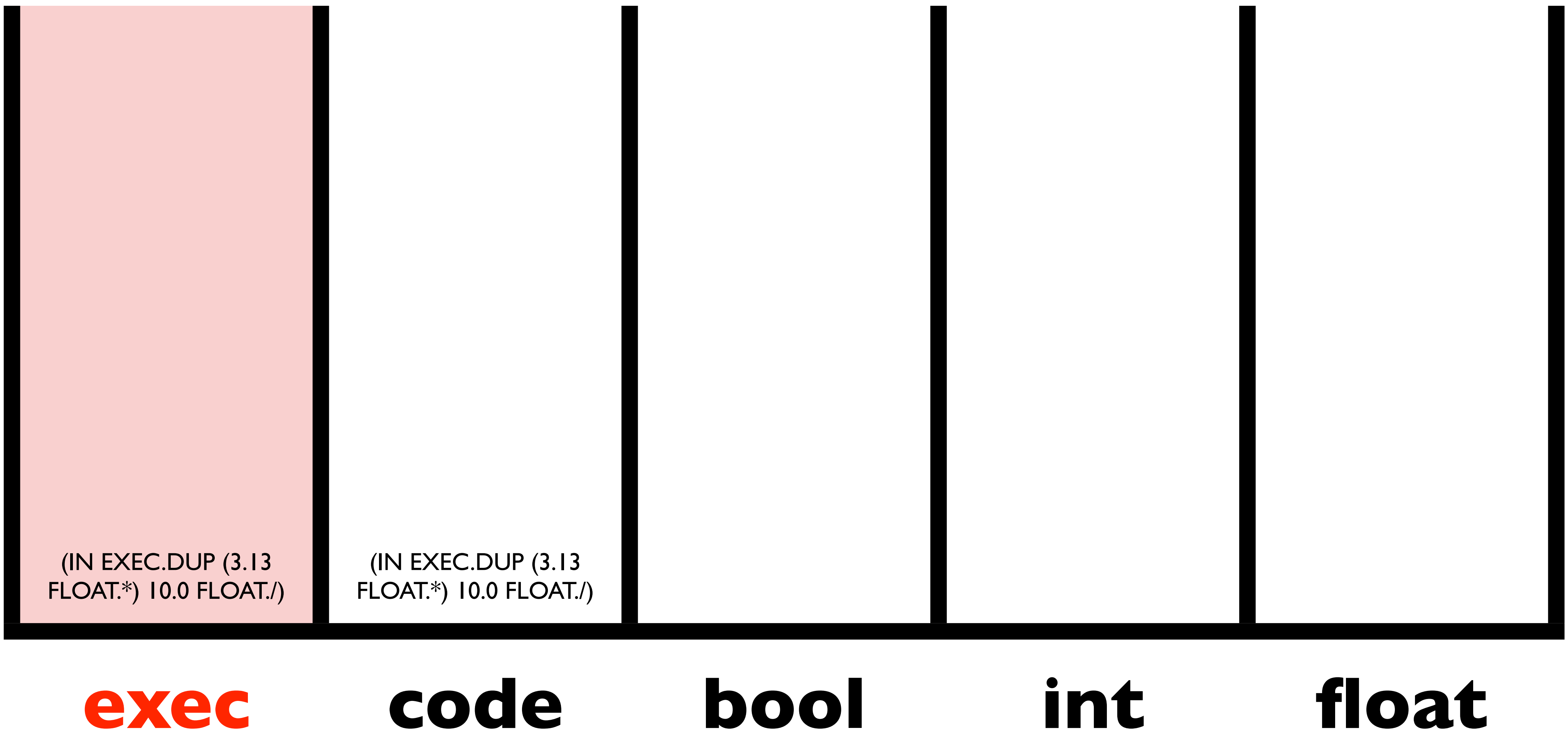


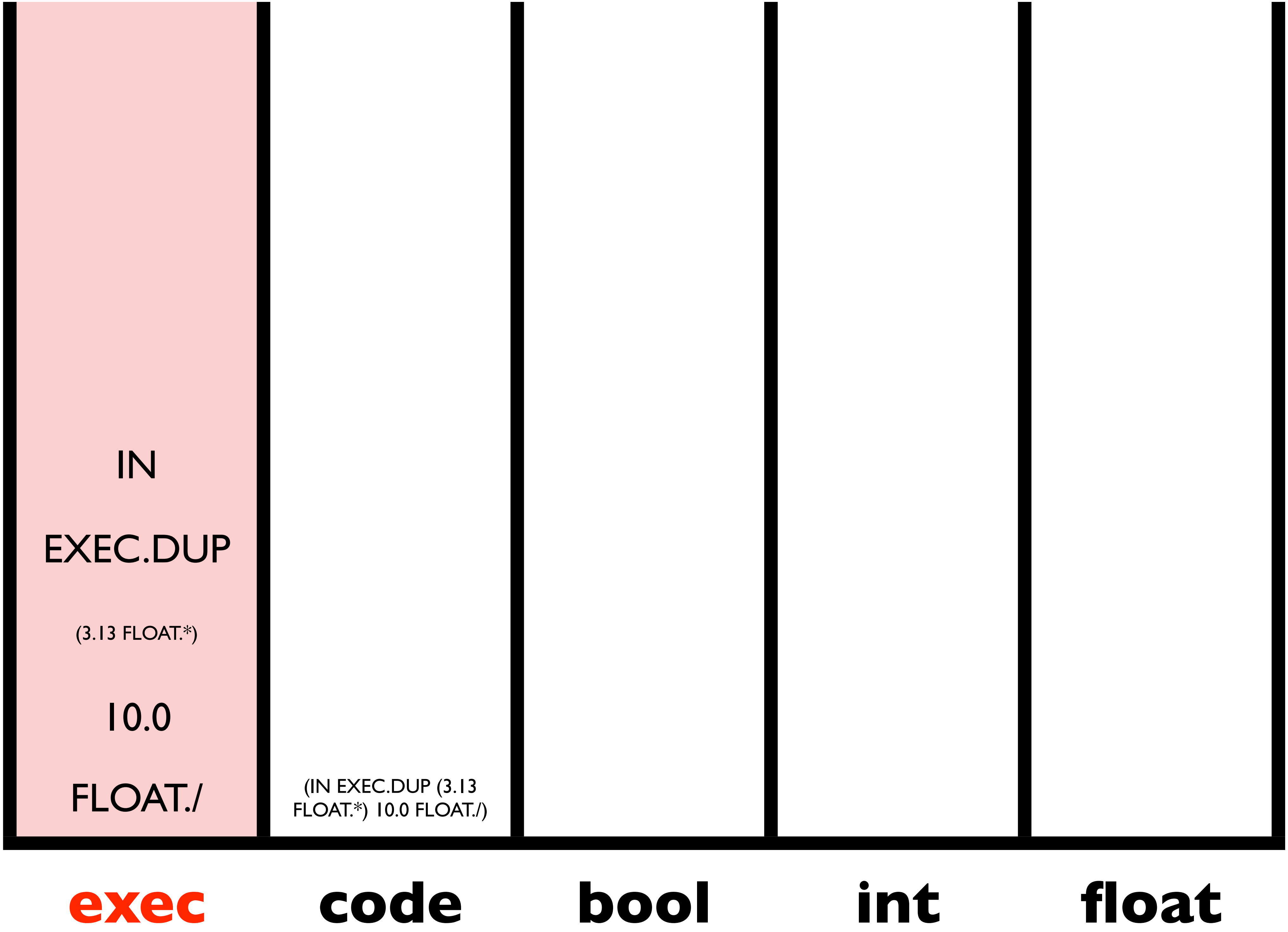
FLOAT.*
4.0
3.14
exec
code
bool
int
float

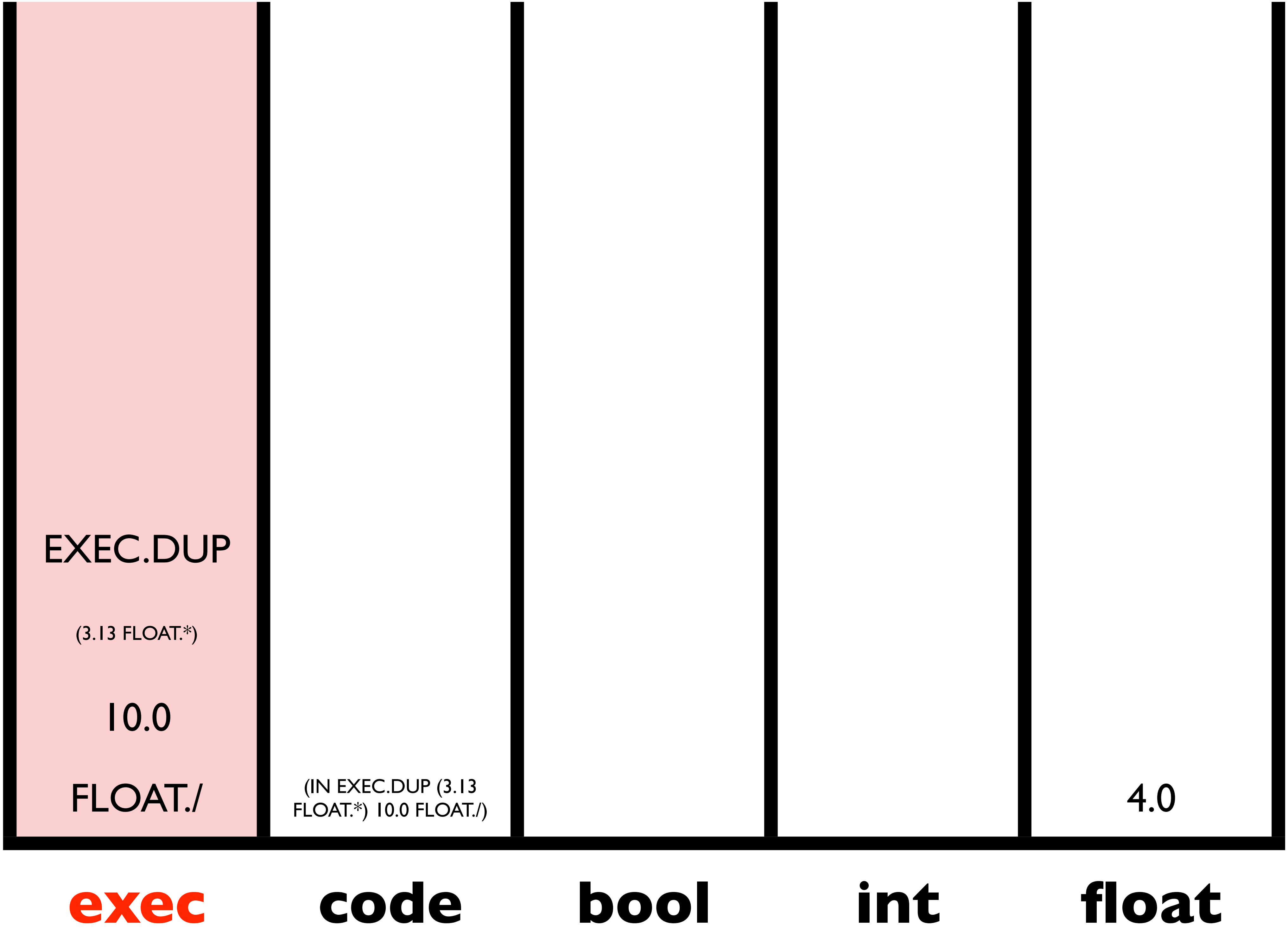

12.56
exec
code
bool
int
float

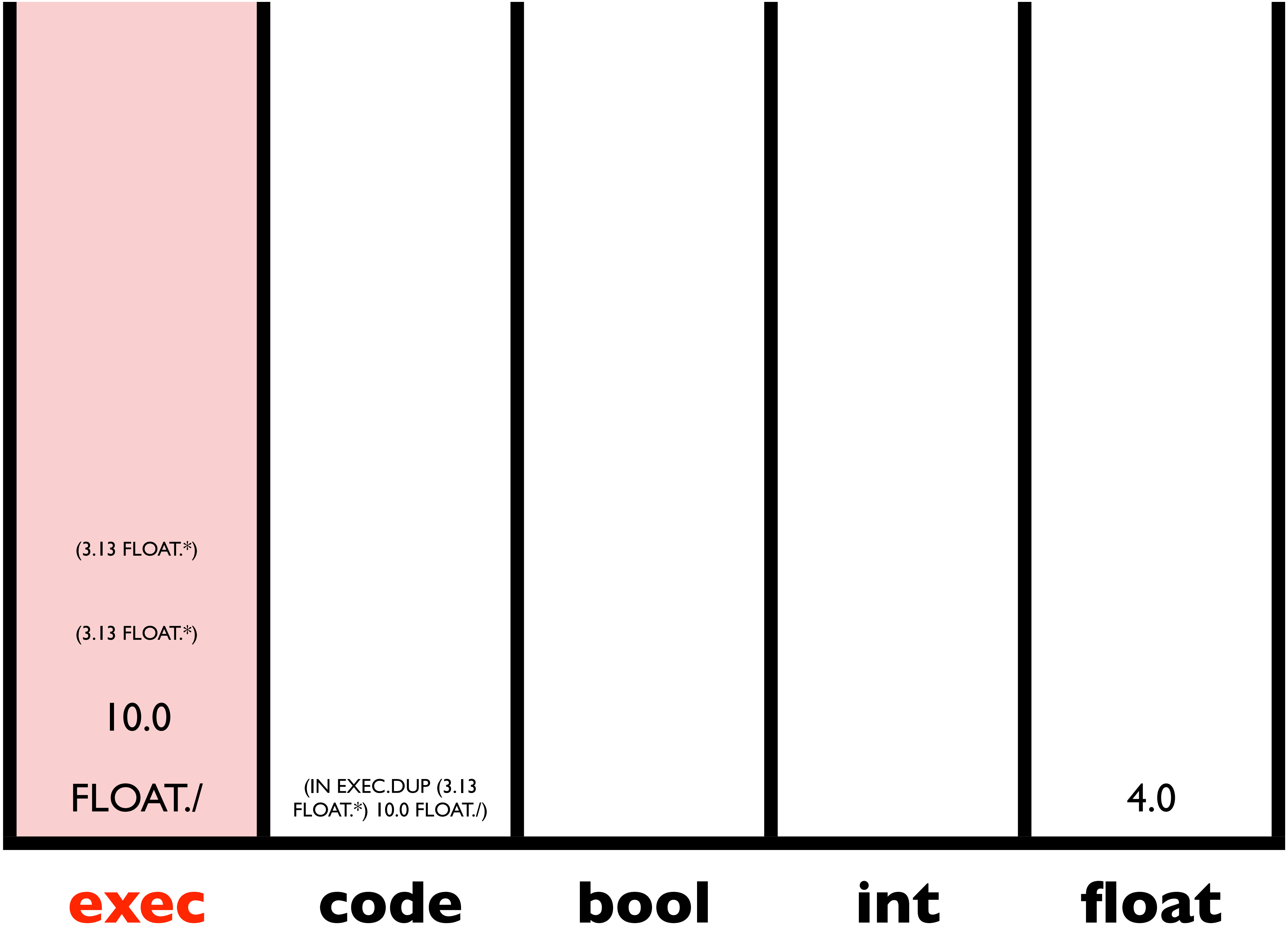
(IN EXEC.DUP (3.13 FLOAT.*)
10.0 FLOAT./)

IN=4.0

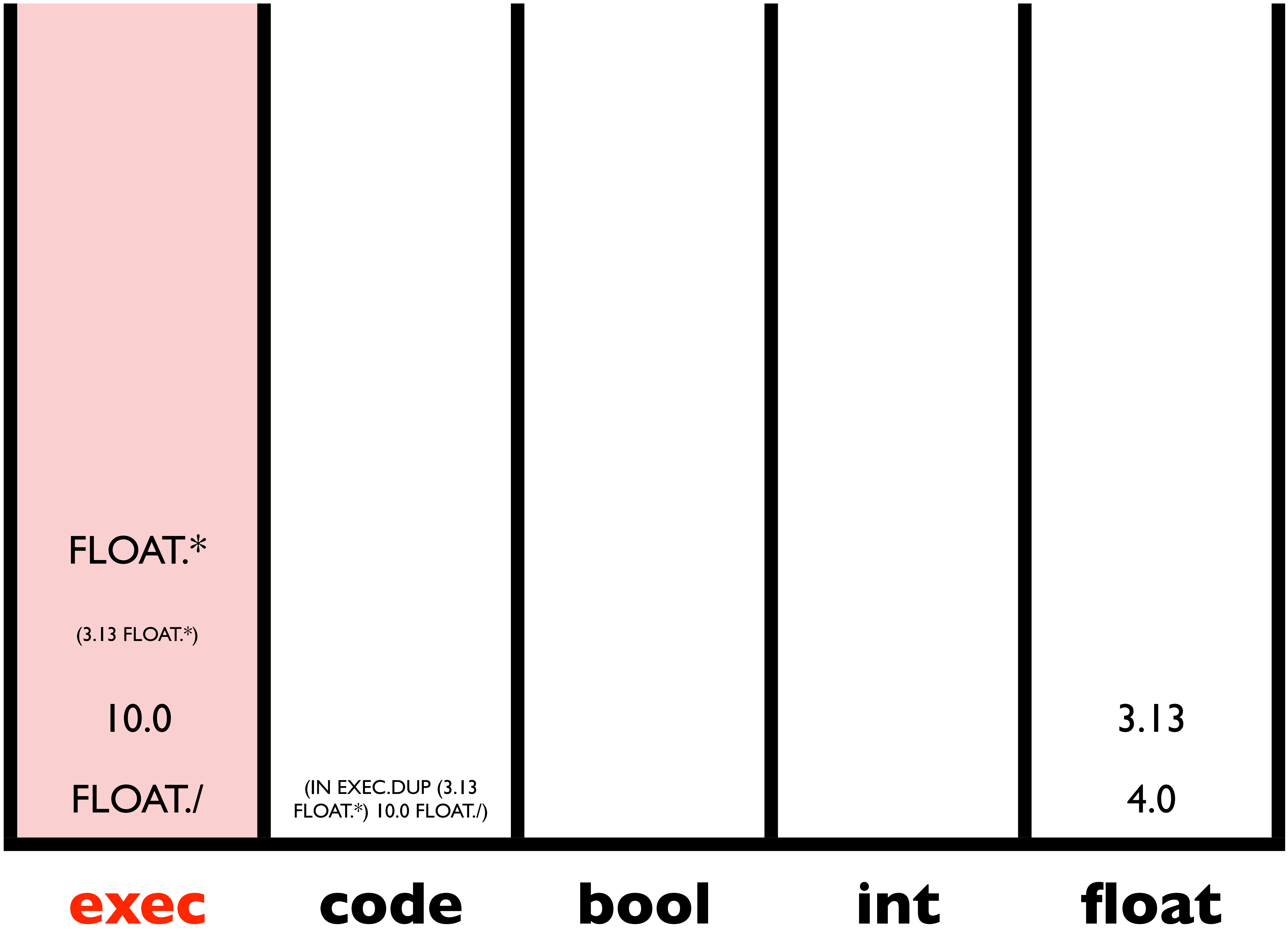


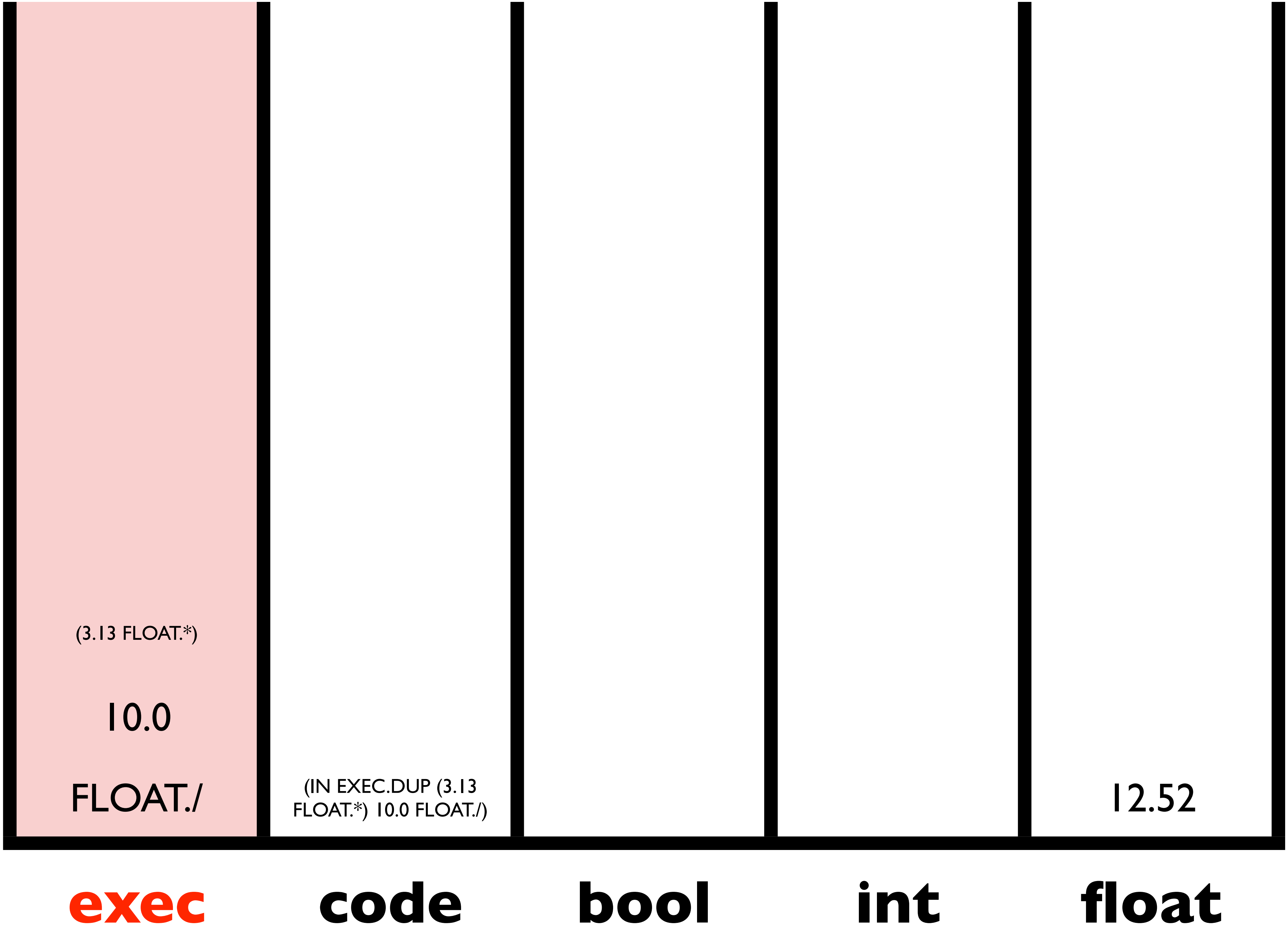

IN
EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
exec
code
bool
int
float

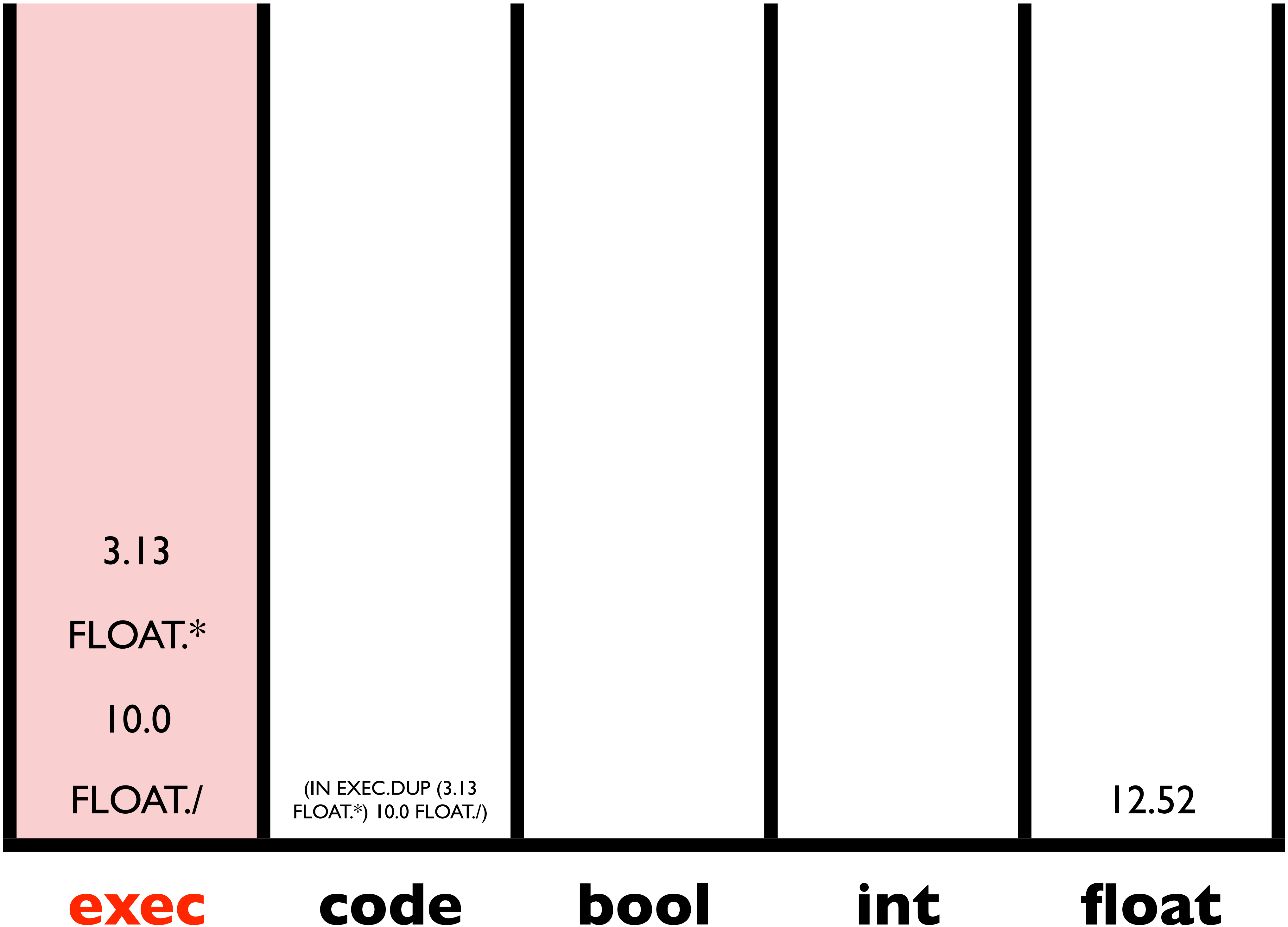

EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

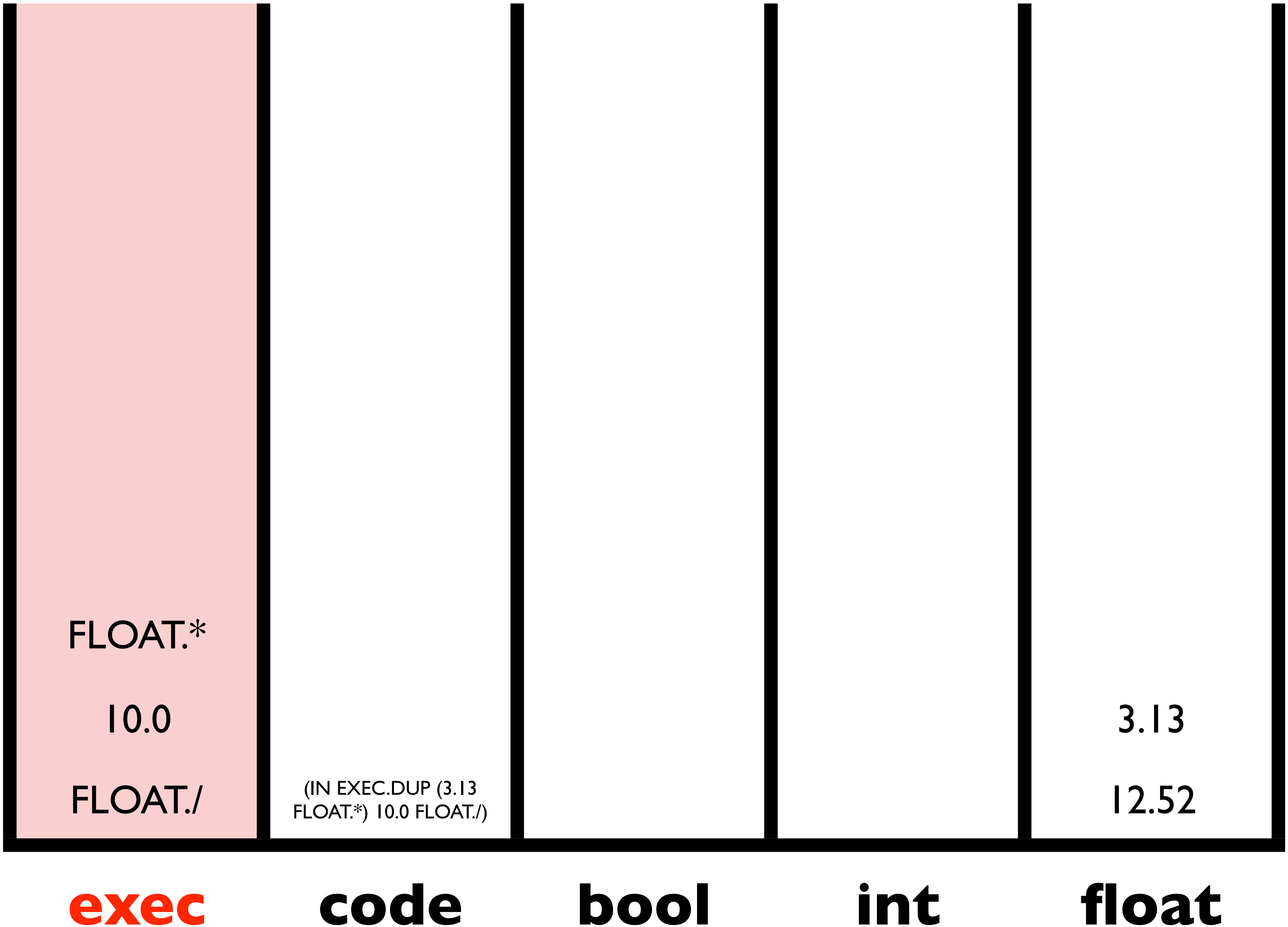

(3.13 FLOAT.*)
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

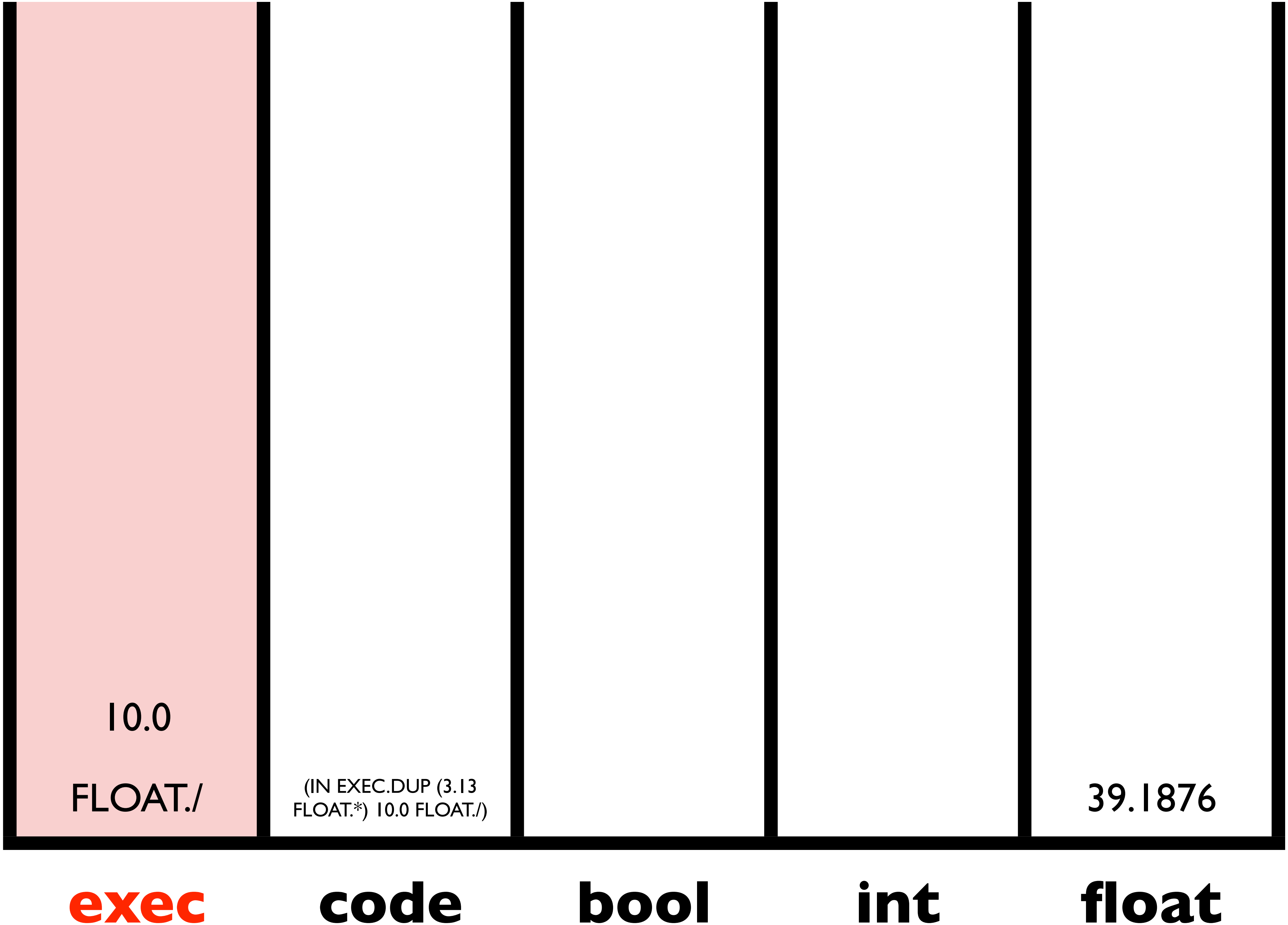
exec	code	bool	int	float
3.13				
FLOAT.*				
(3.13 FLOAT.*)				
10.0				
FLOAT./	(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)			4.0

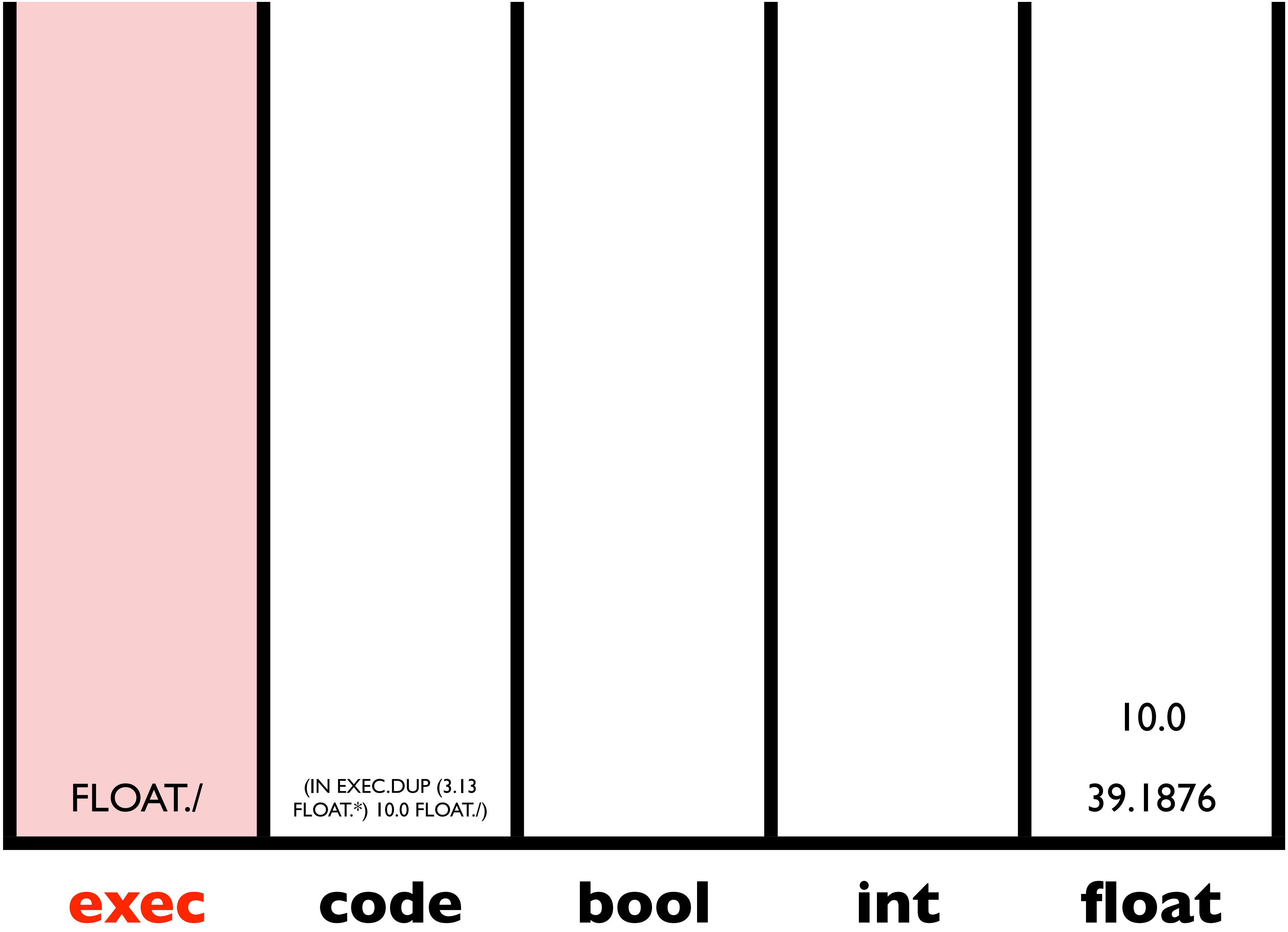

FLOAT.*
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
3.13
4.0
exec
code
bool
int
float


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

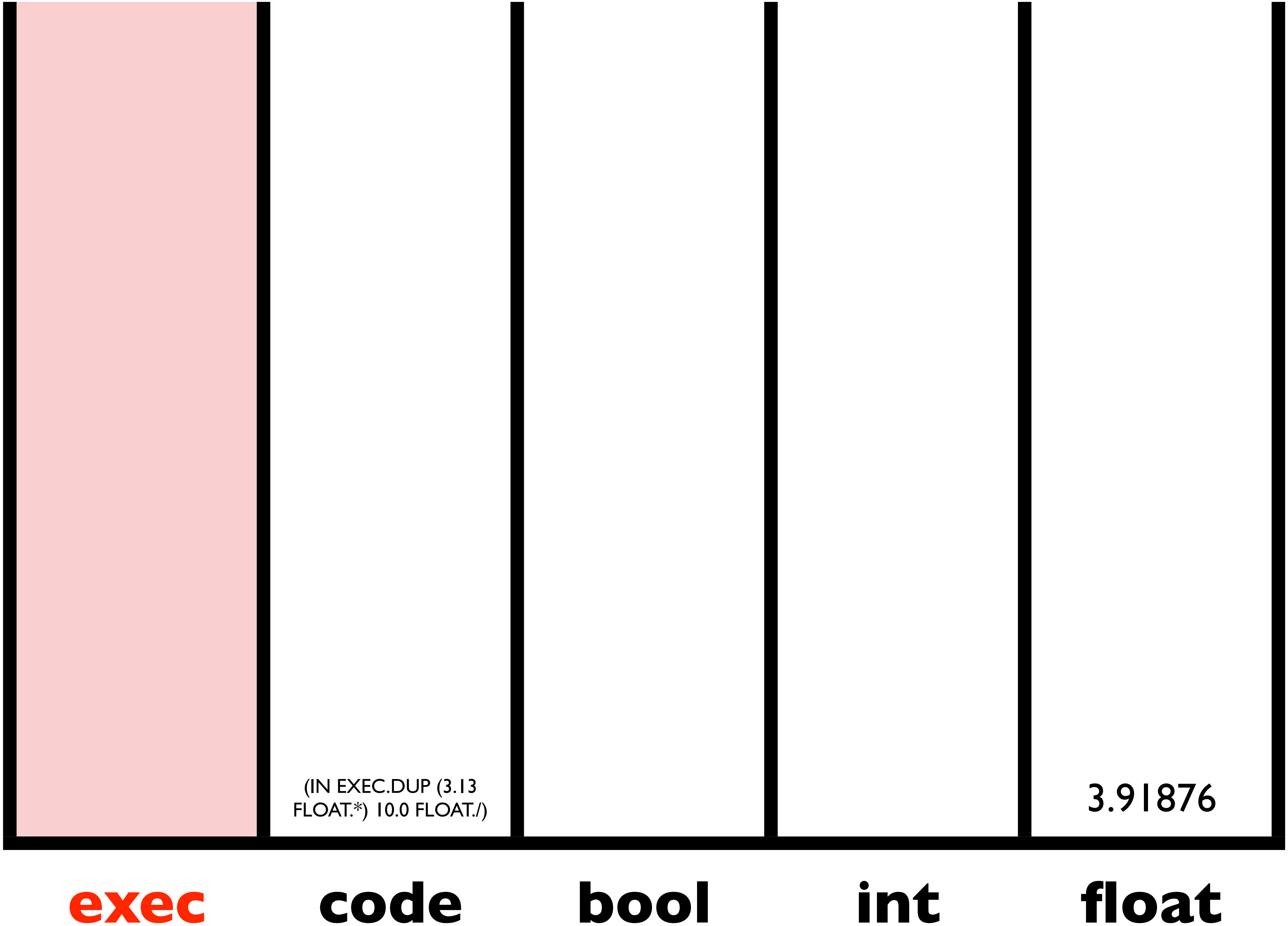

3.13
FLOAT.*
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
12.52
exec
code
bool
int
float


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

Iterators

`CODE.DO*TIMES`, `CODE.DO*COUNT`, `CODE.DO*RANGE`

`EXEC.DO*TIMES`, `EXEC.DO*COUNT`, `EXEC.DO*RANGE`

Additional forms of iteration are supported through code manipulation (e.g. via `CODE.DUP CODE.APPEND CODE.DO`)

Combinators

- Standard *K*, *S*, and *Y* combinators:
  - `EXEC.K` removes the second item from the `EXEC` stack.
  - `EXEC.S` pops three items (call them `A`, `B`, and `C`) and then pushes `(B C)`, `C`, and then `A`.
  - `EXEC.Y` inserts `(EXEC.Y` *`T`*`)` under the top item (*T*).
- A *Y*-based "while" loop:

```
( EXEC.Y
  ( <BODY/CONDITION> EXEC.IF
  ( ) EXEC.POP ) )
```

Named Subroutines

```
( TIMES2 EXEC.DEFINE ( 2 INTEGER.* ) )
```

We will return to this later!

Auto-simplification

Loop:

Make it randomly simpler

If it's as good or better: keep it

Otherwise: revert

Demonstration Results

- Symbolic regression
- Artificial ant
- Boolean problems (e.g. parity, multiplexer)
- List operations (e.g. reversing, sorting)
- ... others

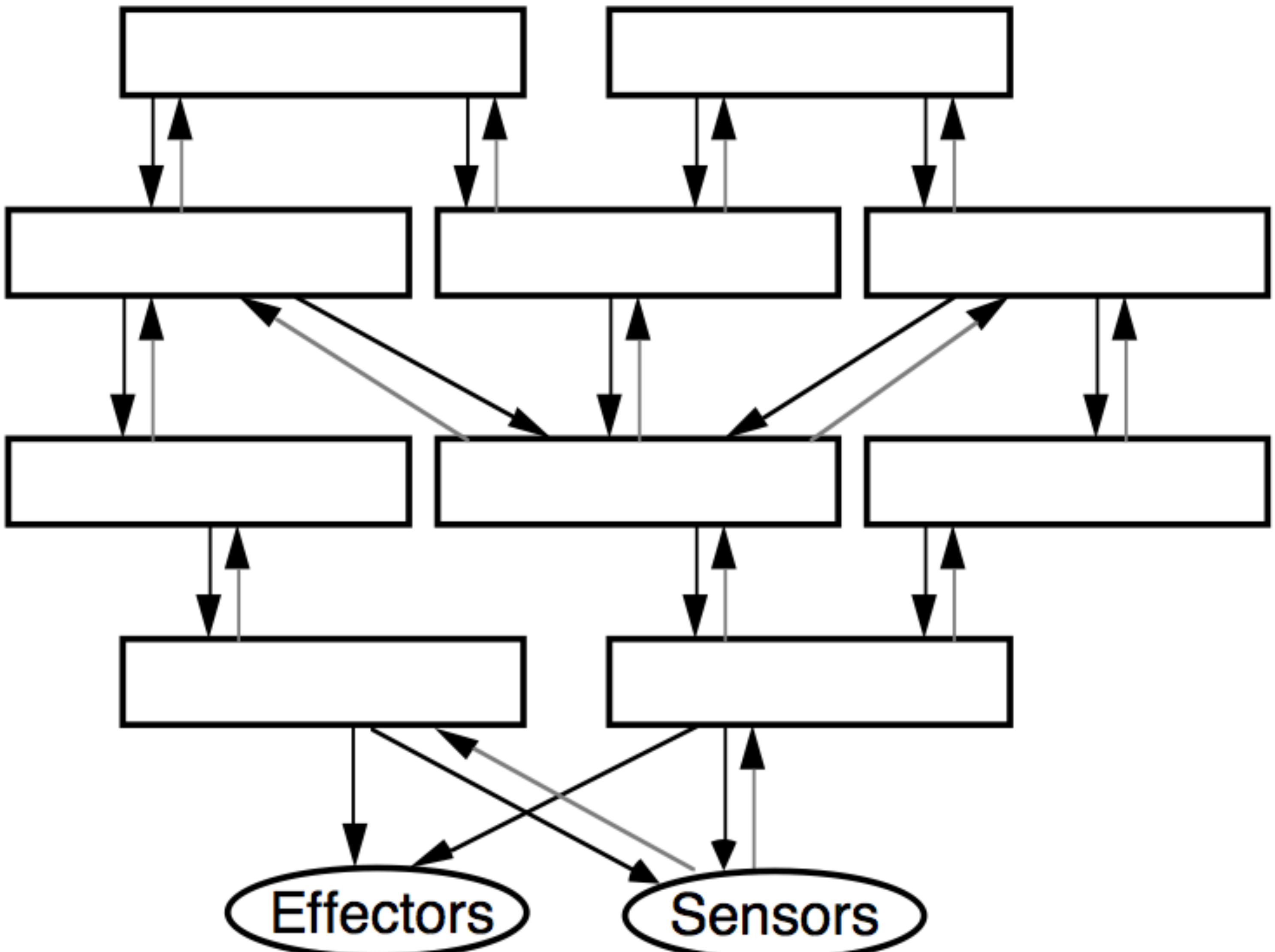
The Odd Problem

- Integer input
- Boolean output
- Was the input odd?
- `((code.nth) code.atom)`

Autoconstructive Evolution

- Individuals make their own children.
- Agents thereby control their own mutation rates, sexuality, and reproductive timing.
- The machinery of reproduction and diversification (i.e., the machinery of evolution) evolves.
- Radical self-adaptation.

Modularity is Everywhere



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 Push1

- Code stack manipulation:
  `(3 code quote (1 integer +) dup do code do)`
- Named modules (complex and ***never used in evolved results!***)

Modularity
Ackley and Van Belle

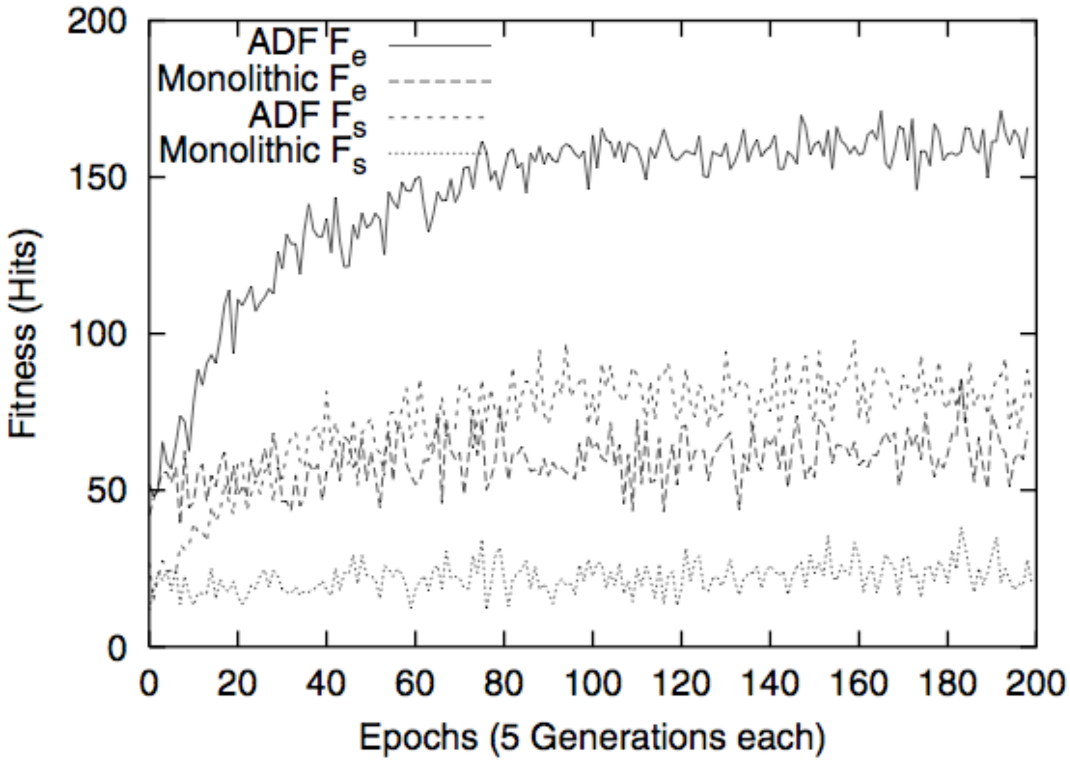
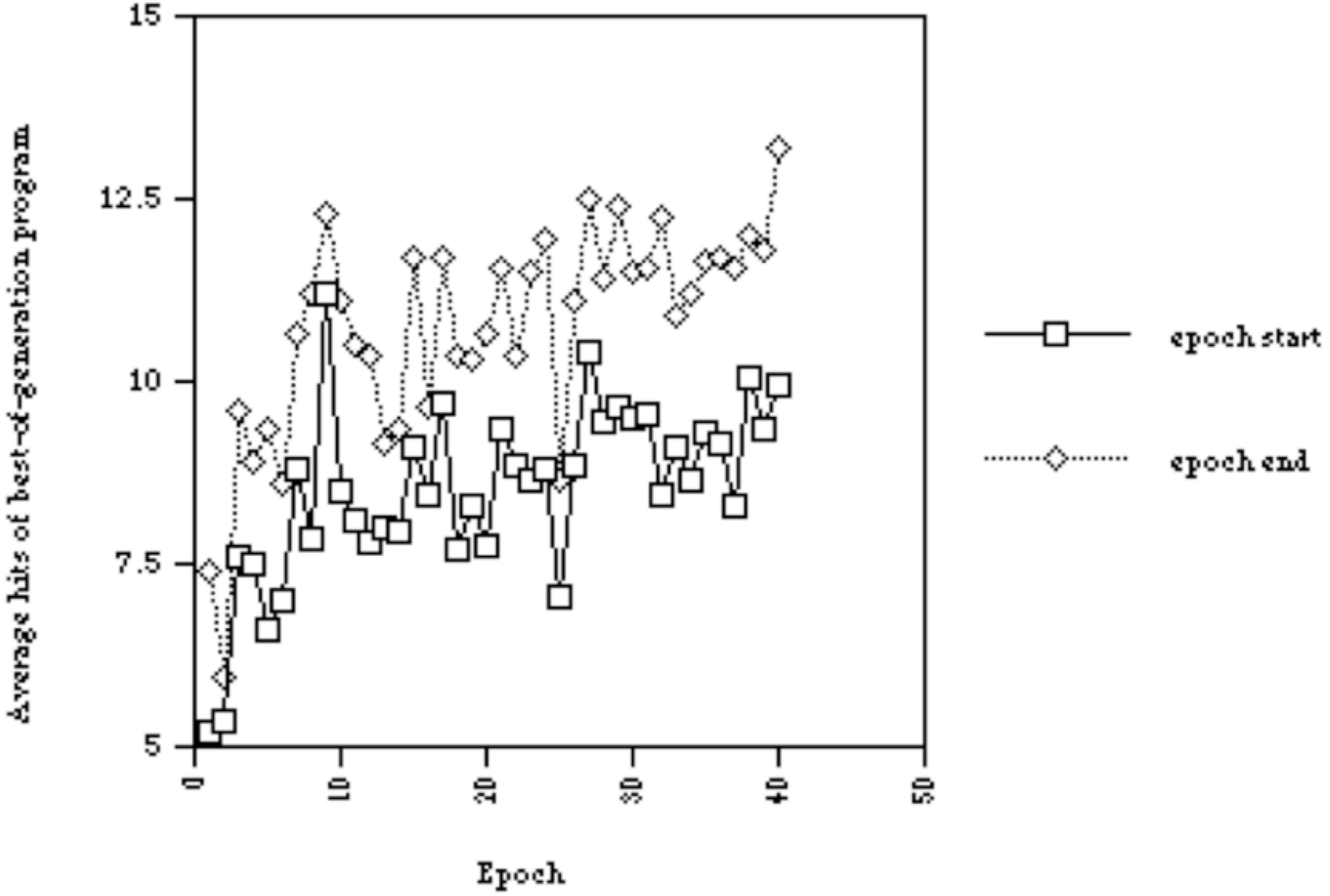


Figure 2: Average fitness values at the start (F_s) and end (F_e) of each epoch when regressing to $y = A\sin(Ax)$. A is selected at the start of each epoch uniformly from the range $[0, 6)$.

Modularity via Push1



Modules in Push3

- Execution stack manipulation:

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

  Can be more complex, and has produced nice results, but tricky in complex contexts

- Named modules:

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

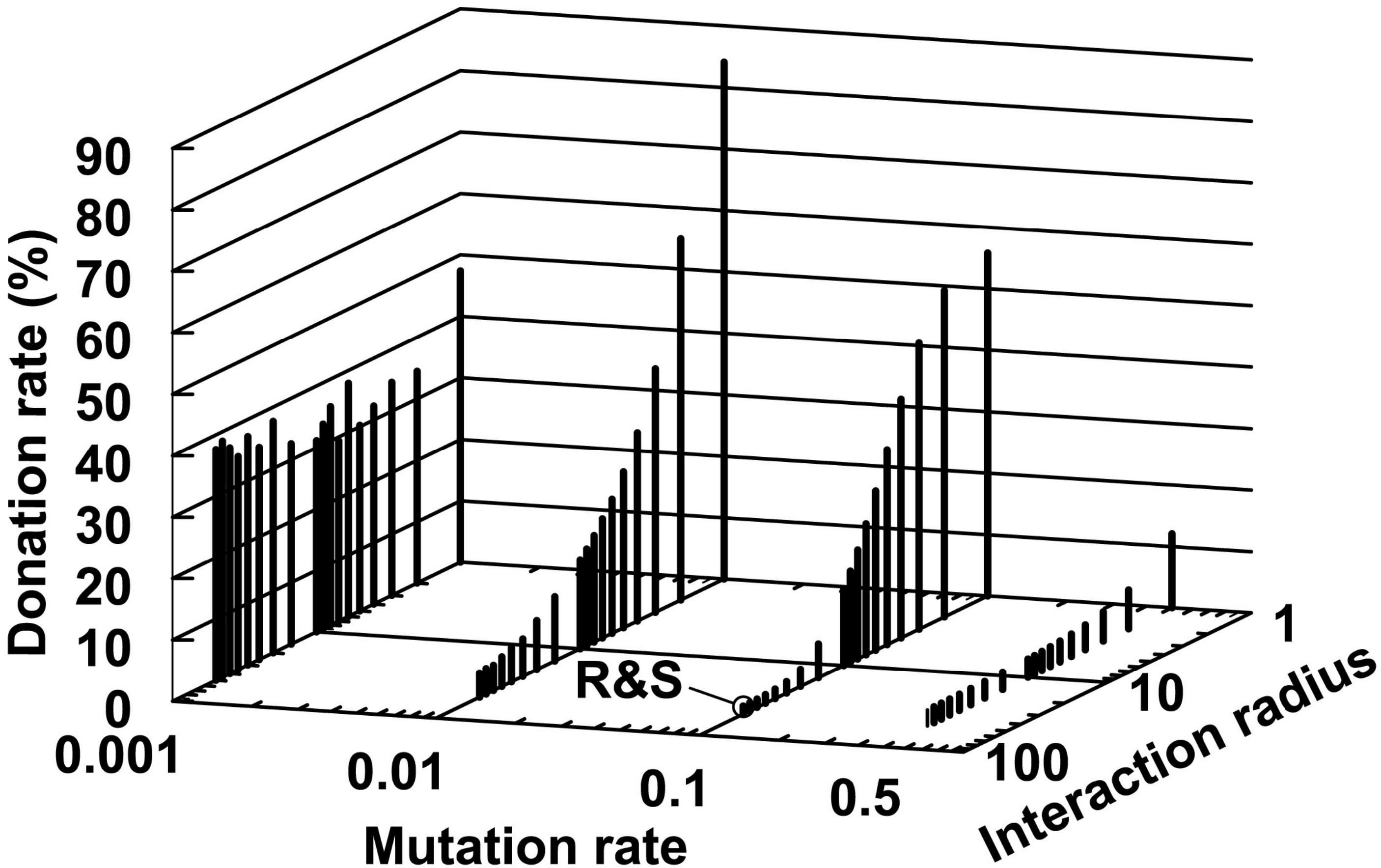
  Simpler than in Push1; general but coordinating definitions/references is tricky ***and this also never arises in evolution!***

- How can we do better?

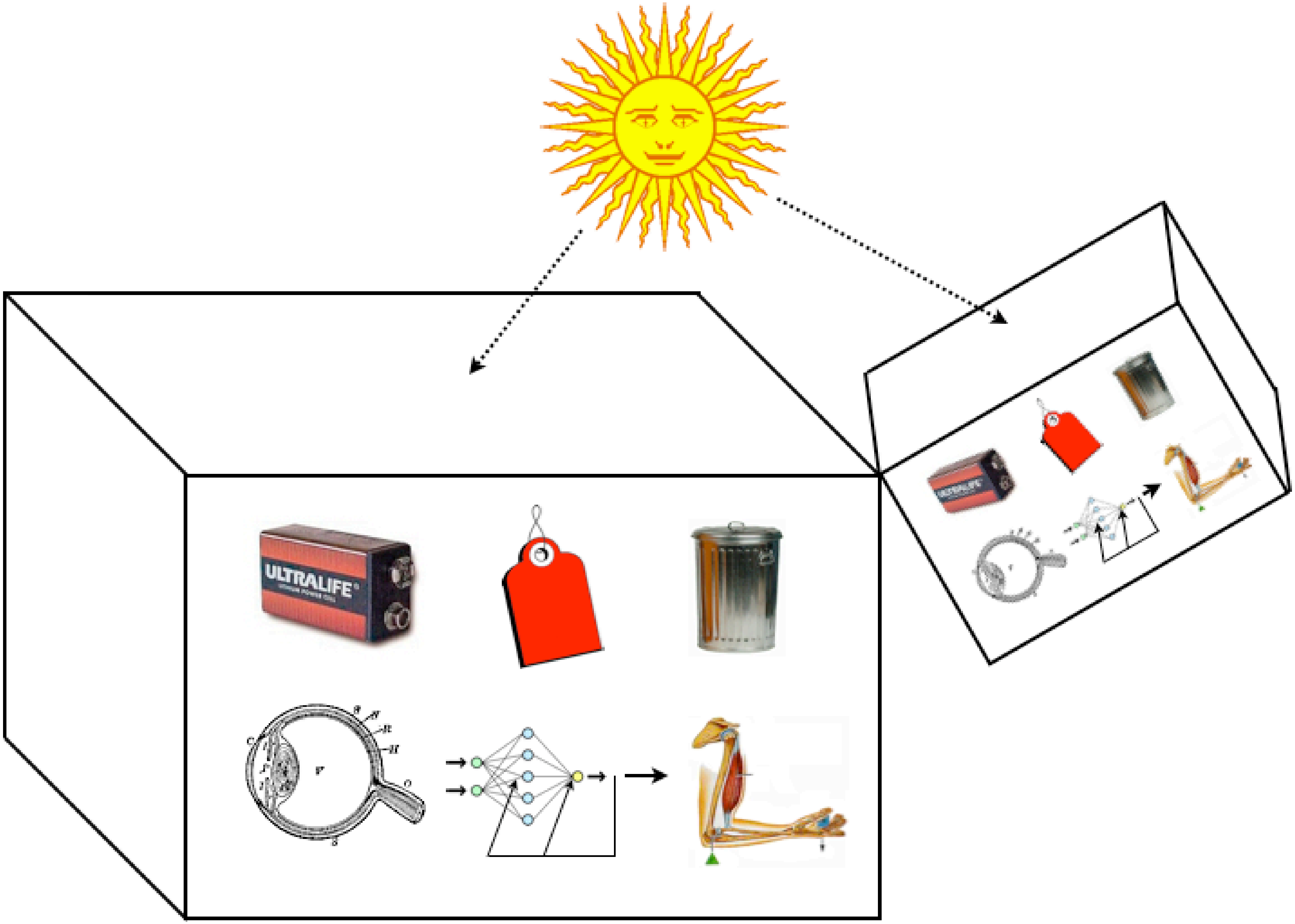
Tag-Mediated Altruism

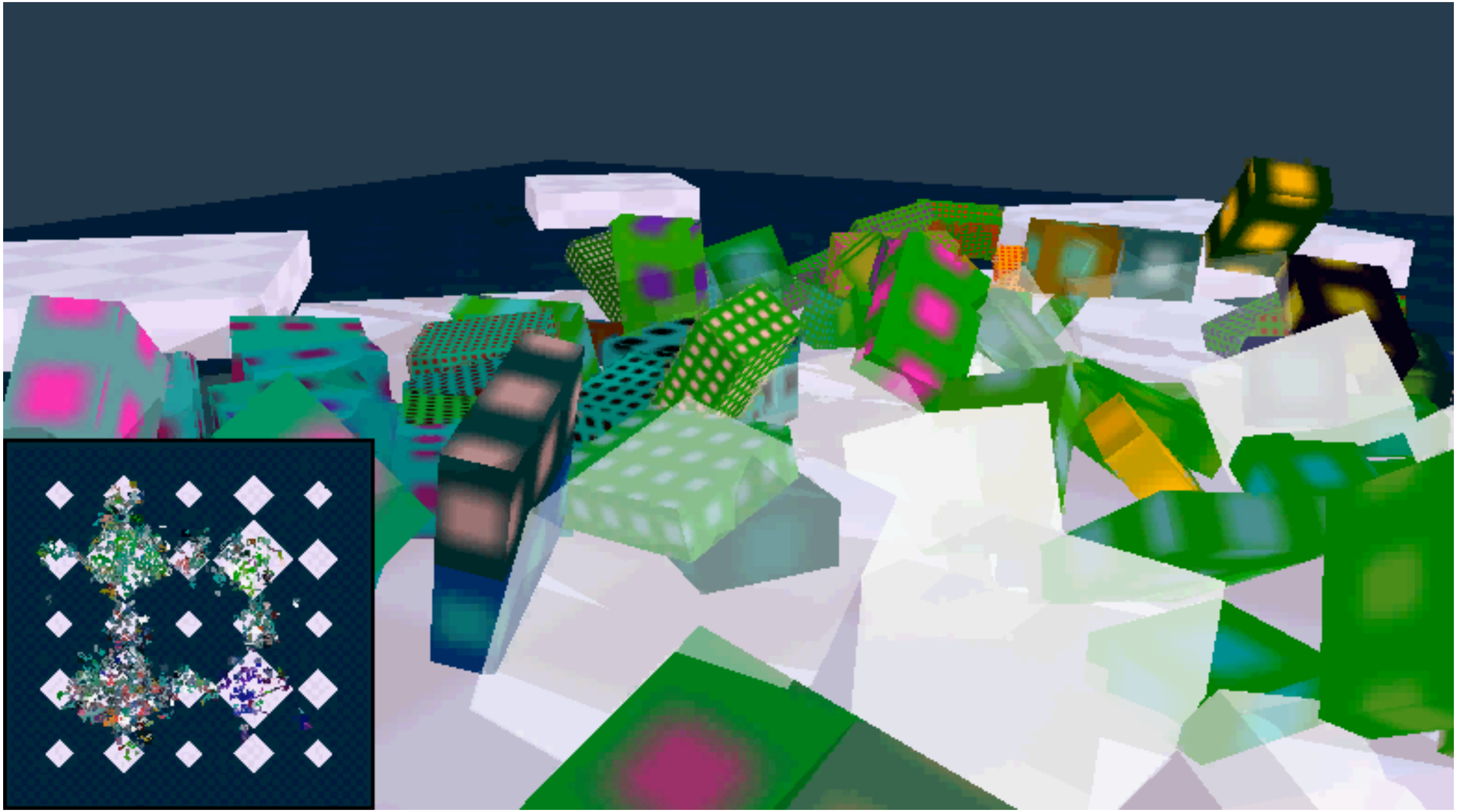
- Tags = arbitrary identifiers (Holland, 1995)
- Riolo *et al*. (*Nature*, 2001) showed that altruism based only on tag similarity can evolve in simple simulations.
- Roberts & Sherratt (*Nature*, 2002) claimed that Riolo *et al.*'s result held only when agents with identical tags were *required* to donate to one another.

Genetic Stability and Territorial Structure



Spector, L., and Klein, J. Genetic stability and territorial structure facilitate the evolution of tag-mediated altruism. In *Artificial Life*.


ULTRALIFE

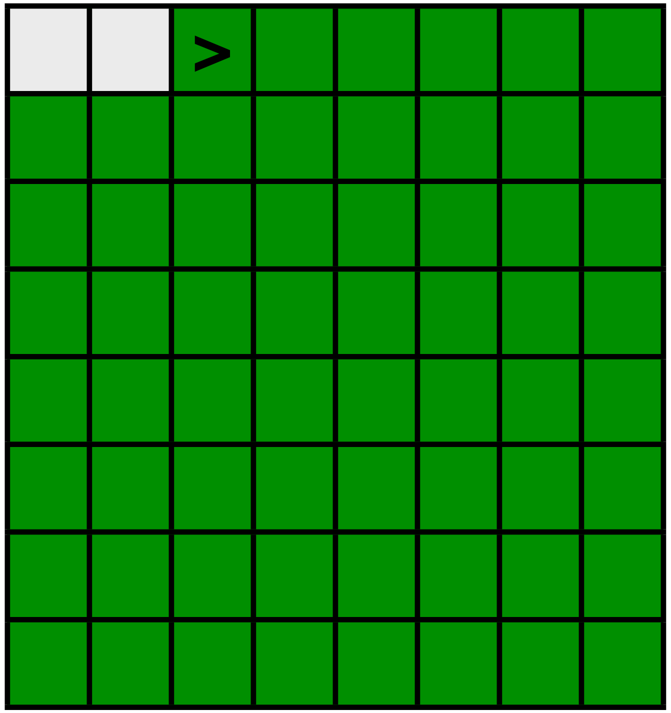


Tags in Push

- 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 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

Rocks-Cluster Physical View for Wed, 02 Mar 2011 07:28:18 -0500

Get Fresh Data

Full View

Grid > Rocks-Cluster > --Choose a Node

Rocks-Cluster cluster - Physical View | Columns 5

Verbosity level (Lower is more compact): 3 ○ 2 ◉ 1 ○

Total CPUs: **158**
Total Memory: **229.5 GB**

Total Disk: **3798.2 GB**
Most Full Disk: **compute-1-11.local (77.2% Used)**

Rack 0

- fly.local — 0.11 — cpu: 1.56G (8) — mem: 15.68G

Rack 1

- compute-1-16.local — 4.00 — cpu: 2.93G (4) — mem: 7.77G
- compute-1-15.local — 3.97 — cpu: 2.93G (4) — mem: 7.77G
- compute-1-14.local — 3.99 — cpu: 2.93G (4) — mem: 7.77G

Rack 2

- compute-2-10.local — 2.00 — cpu: 2.93G (2) — mem: 3.86G
- compute-2-9.local — 2.00 — cpu: 2.93G (2) — mem: 3.86G
- compute-2-8.local — 2.00 — cpu: 2.93G (2) — mem: 3.86G
- compute- — 2.12

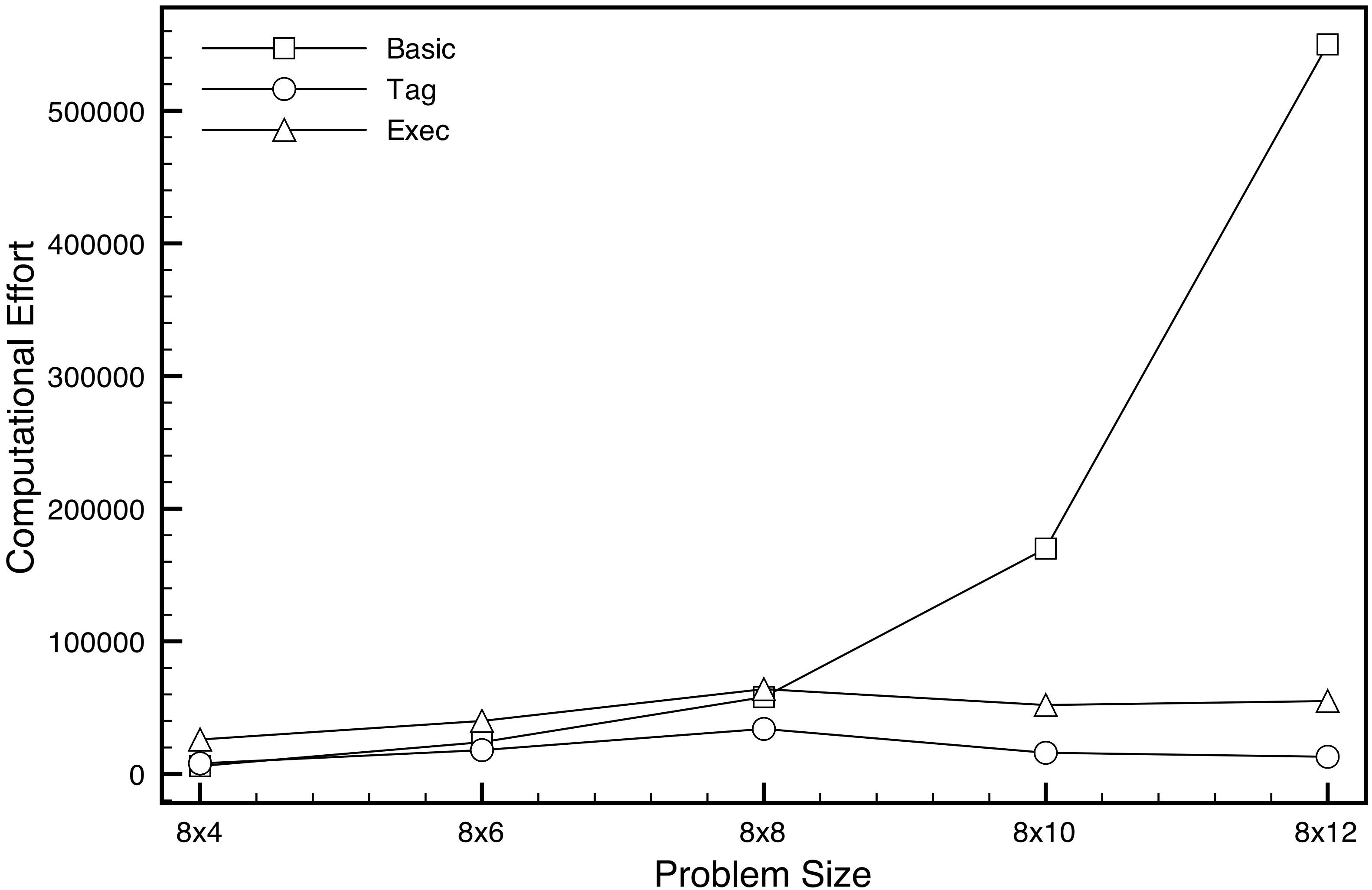
Rack 3

- compute-3-3.local — 2.00 — cpu: 1.95G (2) — mem: 3.87G

Rack 4

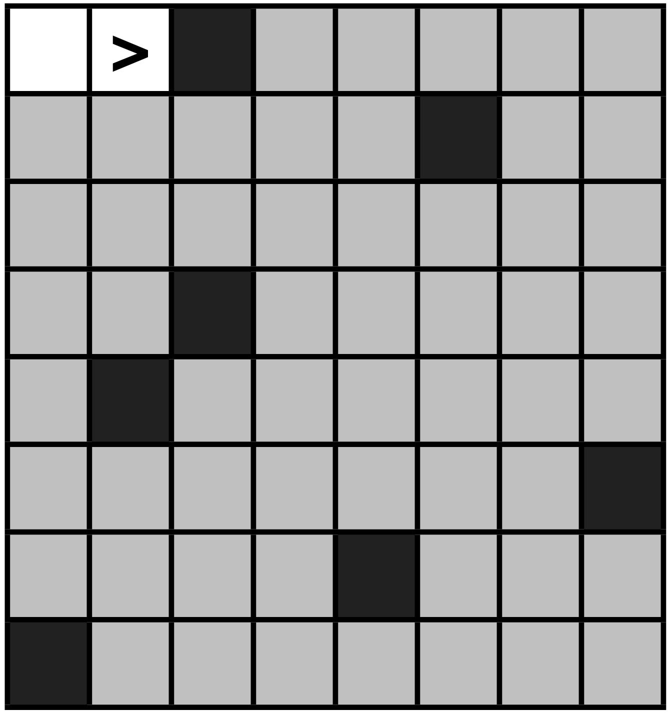
- compute-4-2.local — 49.12 — cpu: 1.86G (48) — mem: 31.42G
- compute-4-1.local — 16.04 — cpu: 2.21G (16) — mem: 23.53G

Lawnmower Effort



Dirt-Sensing, Obstacle-Avoiding Robot Problem

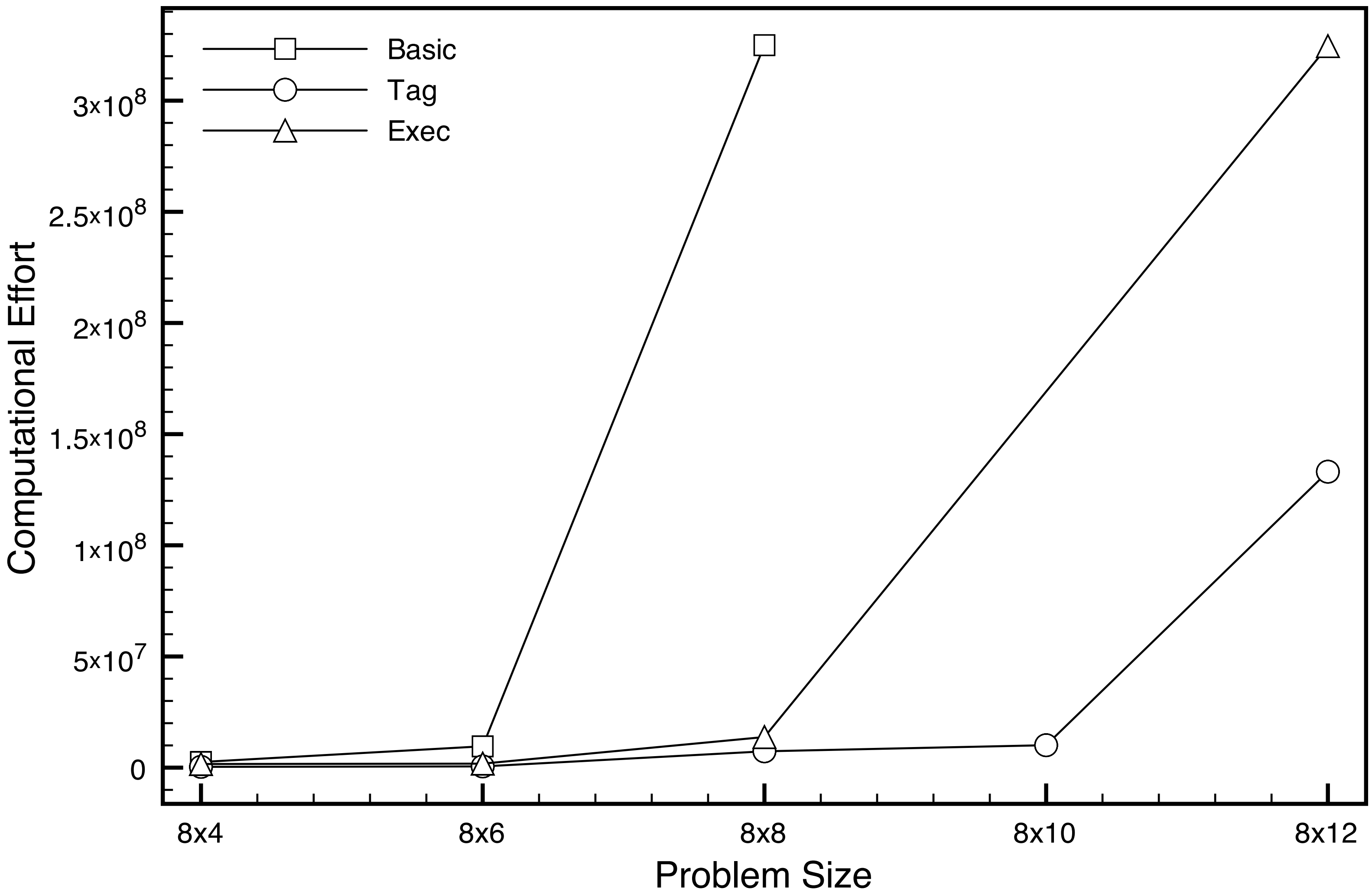
Like the lawnmower problem but harder and less uniform



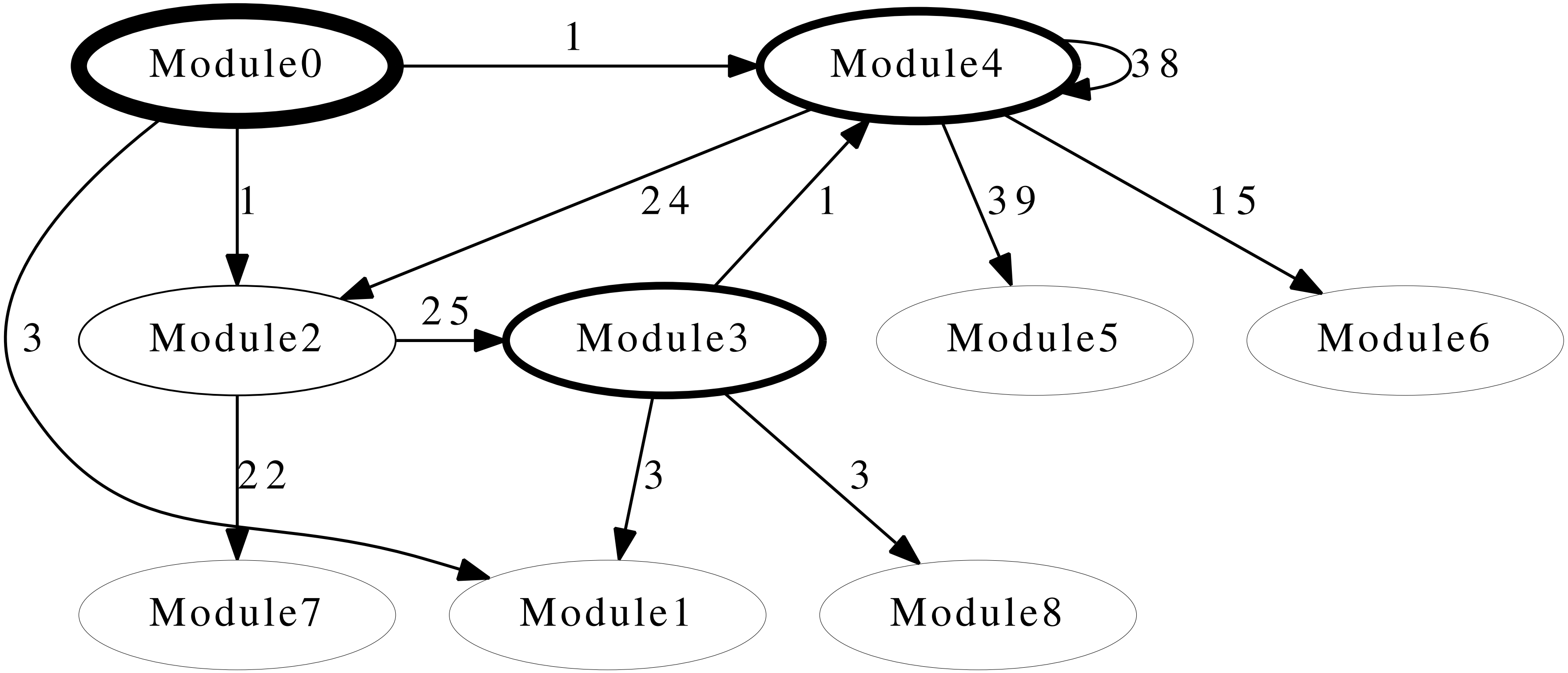
DSOAR Instructions

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

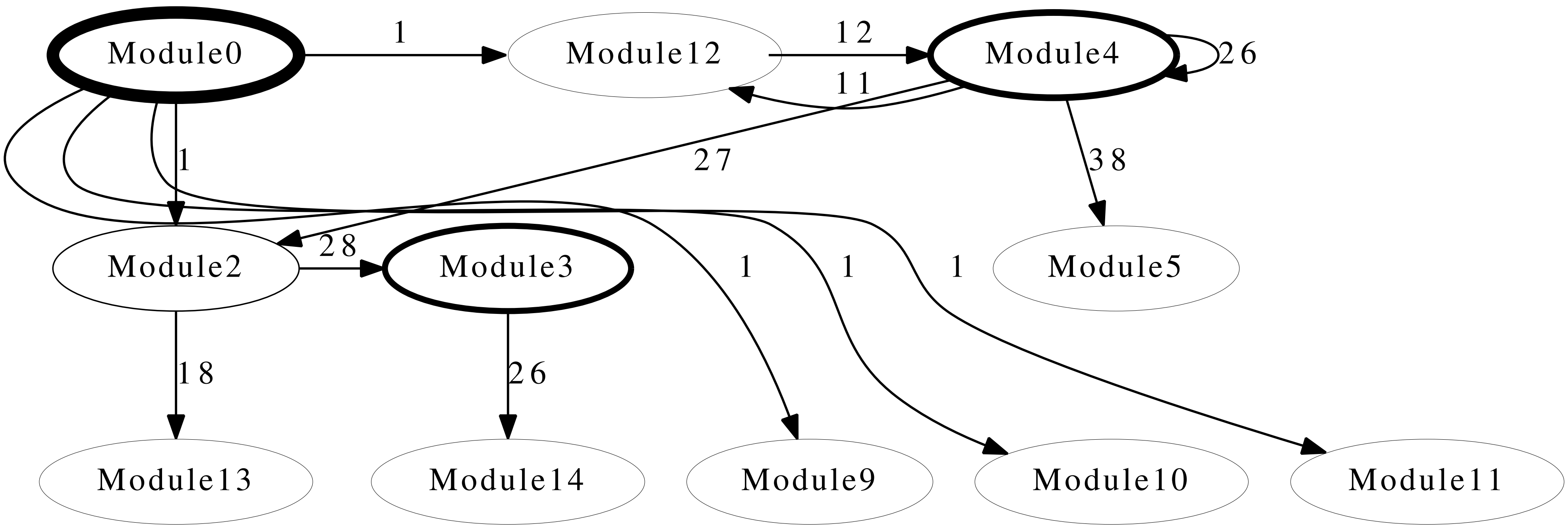
DSOAR Effort



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

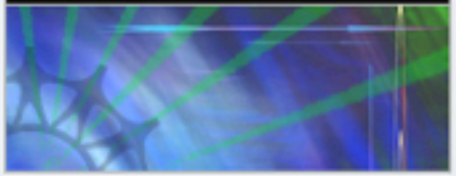
- Execution stack manipulation supports the evolution of modular programs in many situations
- Tag-based modules are more effective in complex, non-uniform problem environments
- Tag-based modules may help to evolve complex software and solutions to unsolved problems in the future

National Science Foundation
WHERE DISCOVERIES BEGIN

SEARCH
NSF Web Site

HOME | FUNDING | AWARDS | DISCOVERIES | NEWS | PUBLICATIONS | STATISTICS | ABOUT | FastLane

Awards



Search Awards
Recent Awards
Presidential and Honorary Awards
About Awards

How to Manage Your Award

Grant Policy Manual
Grant General Conditions
Cooperative Agreement Conditions
Special Conditions
Federal Demonstration Partnership
Policy Office Website

Award Abstract #1017817

RI: Small: RUI: Evolution of Robustly Intelligent Computational Systems

NSF Org:	IIS Division of Information & Intelligent Systems
Initial Amendment Date:	August 19, 2010
Latest Amendment Date:	August 19, 2010
Award Number:	1017817
Award Instrument:	Standard Grant
Program Manager:	Sven G. Koenig IIS Division of Information & Intelligent Systems CSE Directorate for Computer & Information Science & Engineering
Start Date:	September 1, 2010
Expires:	August 31, 2013 (Estimated)
Awarded Amount to Date:	$423288
Investigator(s):	Lee Spector lspector@hampshire.edu (Principal Investigator)
Sponsor:	Hampshire College