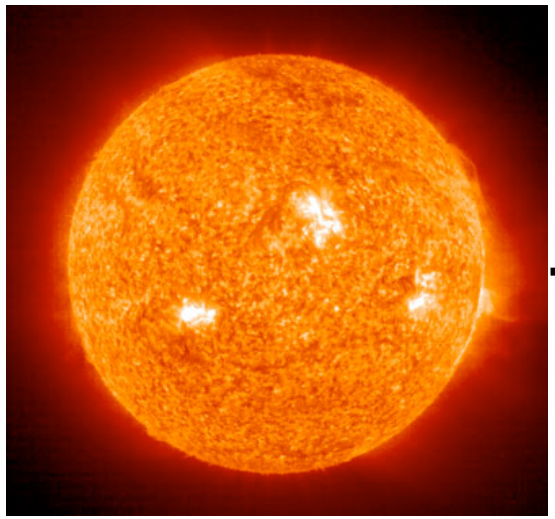
Towards practical autoconstructive evolution:

self-evolution of problem-solving genetic programming systems

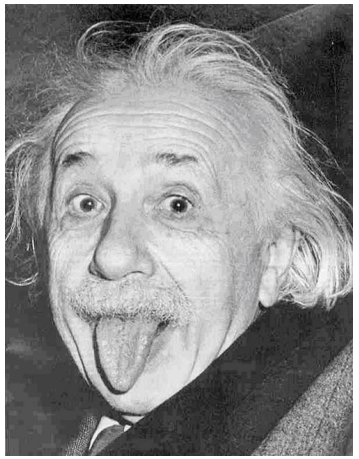
Lee Spector
Cognitive Science
Hampshire College
[+ adjunct at UMass Amherst]

Outline

- ***Ontogenetic programming***: evolved program self-modification
- ***Push***: evolved arbitrary, structured program self-modification
- ***Autoconstructive evolution***: production of offspring by arbitrary, structured program self-modification
- ***Pushpop*** and ***AutoPush***
- Results and prospects



?



Motivation

- We have **very little clue** about the best way to generate offspring in standard GP.
- We have **no clue whatsoever** about the best way to generate offspring in GP with the rich program representations that will become increasingly important.
- Natural reproductive methods evolved.
- Natural reproductive methods co-evolved with the organisms that use them, in the environments in which they use them.

Ontogenetic Programming
(1996)

- Phylogeny and Ontogeny
- Ontogenetic HiGP
- Examples:
  - Binary Sequence Prediction
  - Wumpus World

Phylogeny and Ontogeny

- *Phylogeny* = the developmental progression of a population through evolutionary time.
- *Ontogeny* = the developmental progression of an individual throughout its lifespan.
- GP uses biologically inspired phylogenetic mechanisms.
- Through the addition of ontogenetic mechanisms, GP can produce adaptive programs that solve more difficult problems.

Ontogeny and Morphology

- *Morphology* = the developmental progression of an individual from genotype to phenotype. ("growth phase")
- Morphological components in GP include Gruau's encoding → network transforms, Zomorodian's tree → PDA transforms, and Spector's ADM expansions. See [Angeline 1995] for formal definitions and a survey.

Ontogeny and Morphology

- *Ontogeny* = the developmental progression of an individual *throughout its lifespan*. **Note** that this development may be guided by the runtime environment.
- Morphology $\subset$ Ontogeny.

Ontogenetic Mechanisms

- Runtime memory mechanisms:
- Indexed memory [Teller 1994]
- Memory terminals [Iba et al. 1995]
- Runtime "morphology" implemented via program self-modification operators. We call this strategy ontogenetic programming.

Program Representations

- **Lisp-style symbolic expressions** (Koza, ...).
- Purely functional/lambda expressions (Walsh, Yu, ...).
- Linear sequences of machine/byte code (Nordin et al., ...).
- **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, ...).

HiGP

Virtual Stack Machine Example

```
push-x noop push-y * push-x
push-z noop - + noop noop
```

The noops have no effect and the remainder is equivalent to the Lisp expression:

```
(+ (* x y) (- x z))
```

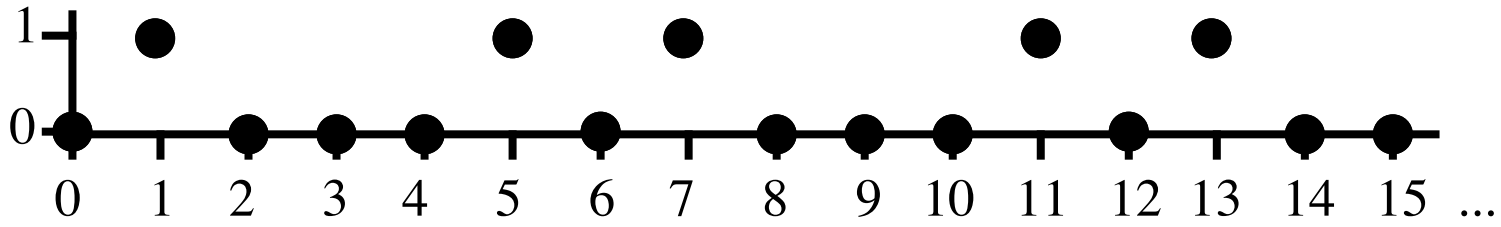
and to the C expression:

```
(x * y) + (x - z)
```

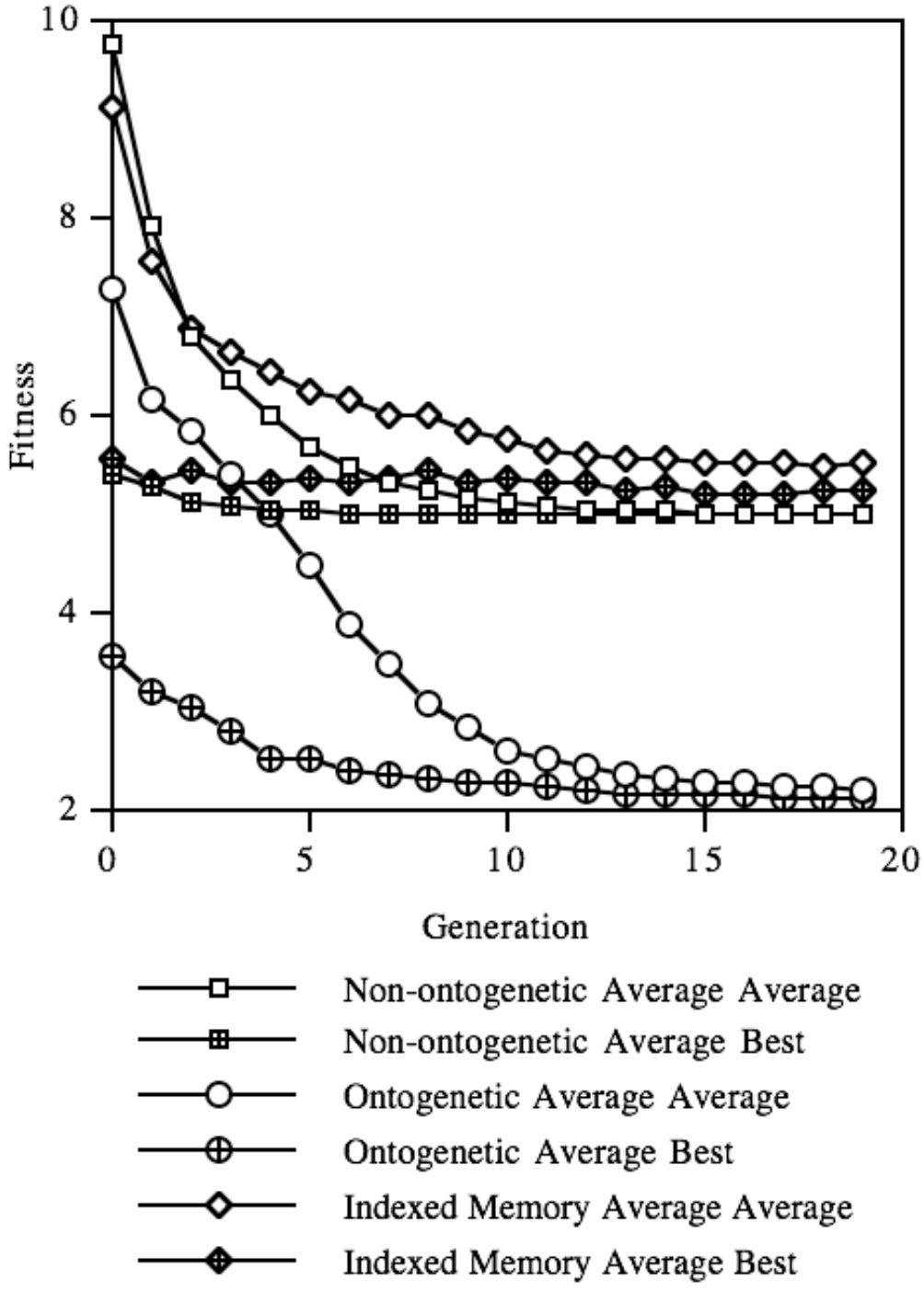
Ontogenetic HiGP

- **segment-copy** copies a part of the linear program over another part of the program. The function takes 3 arguments from the stack: the start position of the segment to copy, the length of the segment, and the position to which it should be copied.
- **shift-left** rotates the program to the left. The call takes one argument from the stack: the distance by which the program is to be rotated.
- **shift-right** rotates the program to the right. The call takes one argument from the stack: the distance by which the program is to be rotated.

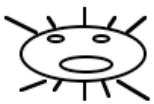
Binary Sequence Prediction

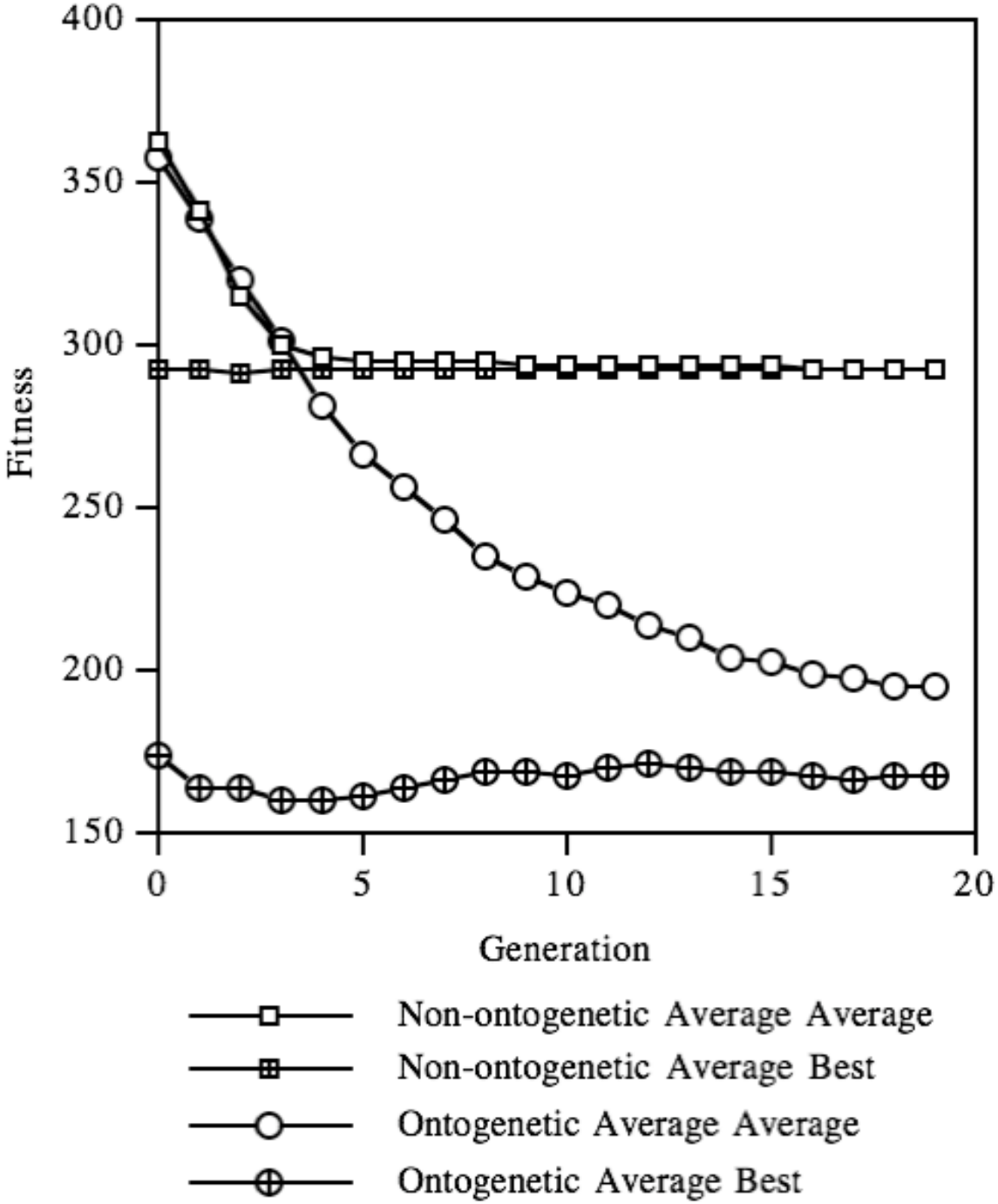


- As in symbolic regression, attempt to evolve a function of x that produces the corresponding y.
- Run programs on a sequence of x values (0–17 here), always in the same order, for each fitness test "lifetime."


10
8
6
4
2
Fitness
0
5
10
15
20
Generation
Non-ontogenetic Average Average
Non-ontogenetic Average Best
Ontogenetic Average Average
Ontogenetic Average Best
Indexed Memory Average Average
Indexed Memory Average Best

Wumpus World

Breeze	O Pit	Breeze		Breeze	O Pit
O Pit	Breeze			Breeze	O Pit
Breeze		Breeze			Breeze
	Breeze	O Pit	Breeze Stench		$ Gold
		Breeze Stench	 Wumpus	Stench	Breeze
 Agent			Stench	Breeze	O Pit


400
350
300
250
200
150
Fitness
0
5
10
15
20
Generation
Non-ontogenetic Average Average
Non-ontogenetic Average Best
Ontogenetic Average Average
Ontogenetic Average Best

Ontogenetic Programming with S-Expressions

- subtree-copy (from-index, to-index)
  - between rather than during executions
  - global indices not meaningful after crossover
  - explosive ontogenetic growth
- structured-subtree copy (from-index, to-index, rpb)
- dynamic ADFs and ADMs
  - versions of defun, funcall etc. in function set
  - store functions/macros in indexed memory
  - runtime self-modification via module redefinition

Expressive Languages

- Strongly typed genetic programming
- Automatically defined functions
- Automatically defined macros
- Architecture-altering operations
- Development and self-modification

Expressive Languages

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

Push

- A programming language designed for programs that evolve
- Simplifies evolution of programs that may use:
  - multiple data types
  - subroutines (any architecture)
  - recursion and iteration
  - evolved control structures
  - evolved evolutionary mechanisms

Push

- Stack-based postfix language with one stack per type
- Turing complete
- 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

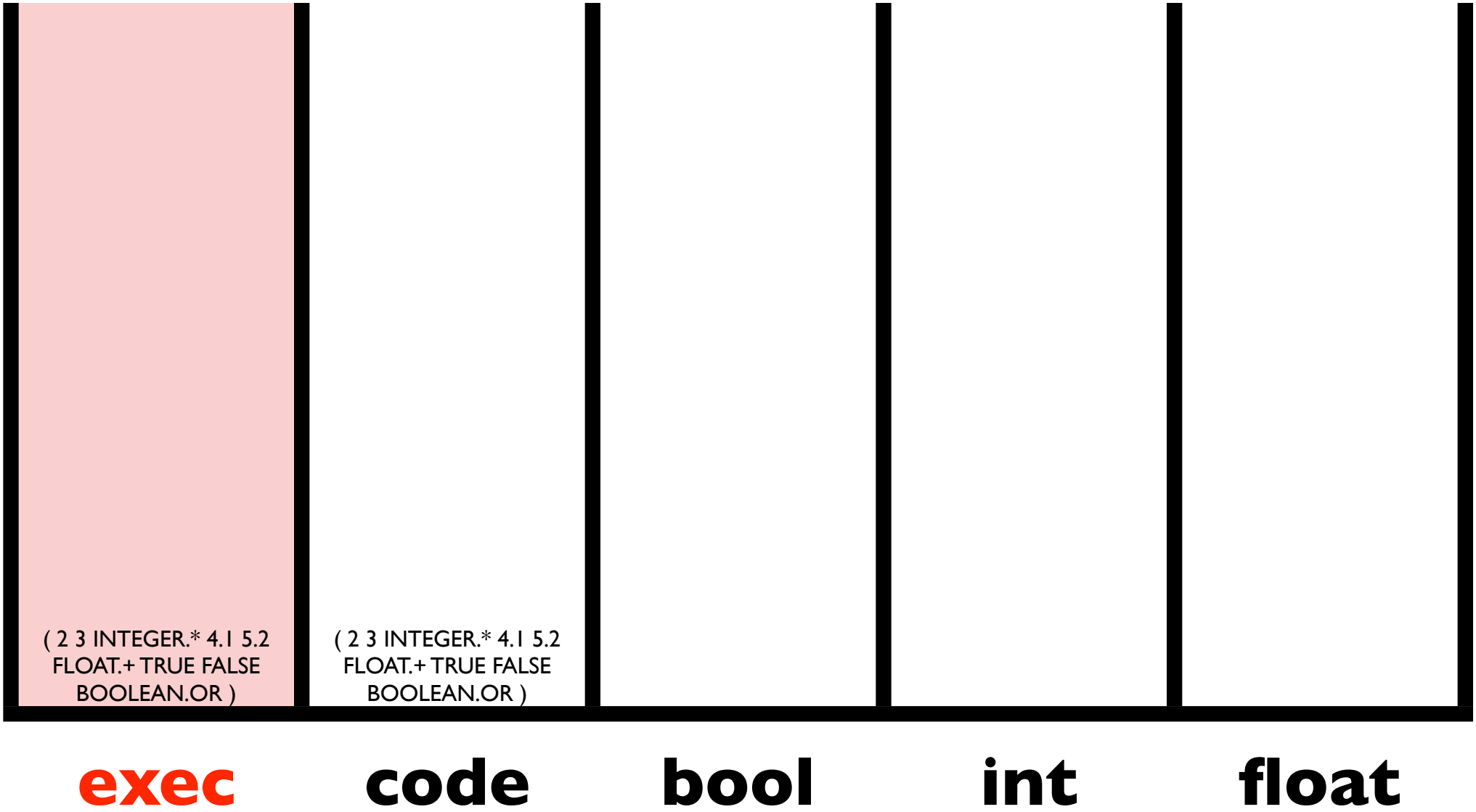
Stack manipulation instructions (all types)	POP, SWAP, YANK, DUP, STACKDEPTH, SHOVE, FLUSH, =
Math (INTEGER and FLOAT)	+, −, /, *, >, <, MIN, MAX
Logic (BOOLEAN)	AND, OR, NOT, FROMINTEGER
Code manipulation (CODE)	QUOTE, CAR, CDR, CONS, INSERT, LENGTH, LIST, MEMBER, NTH, EXTRACT
Control manipulation (CODE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

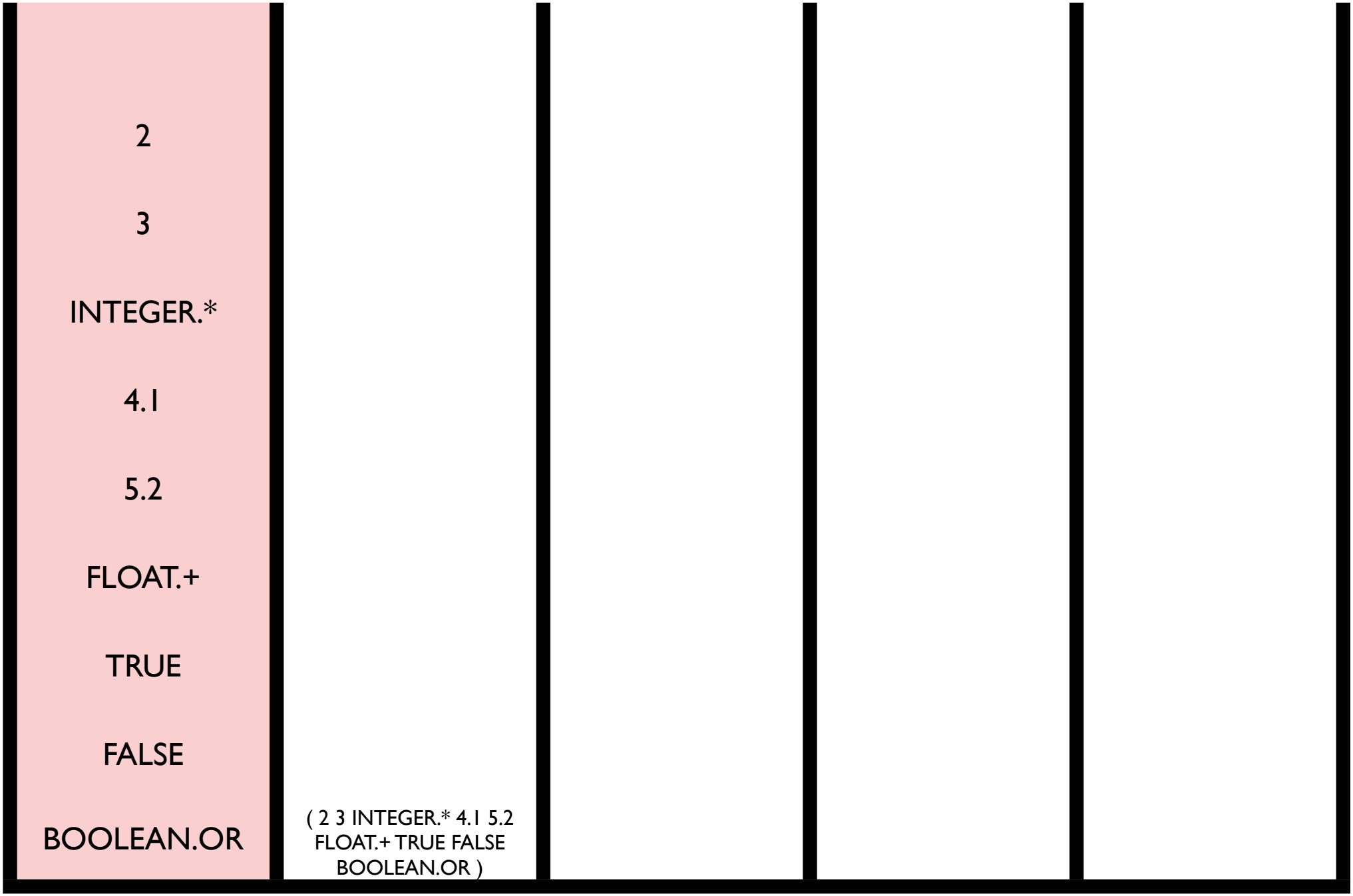
Push(3) Semantics

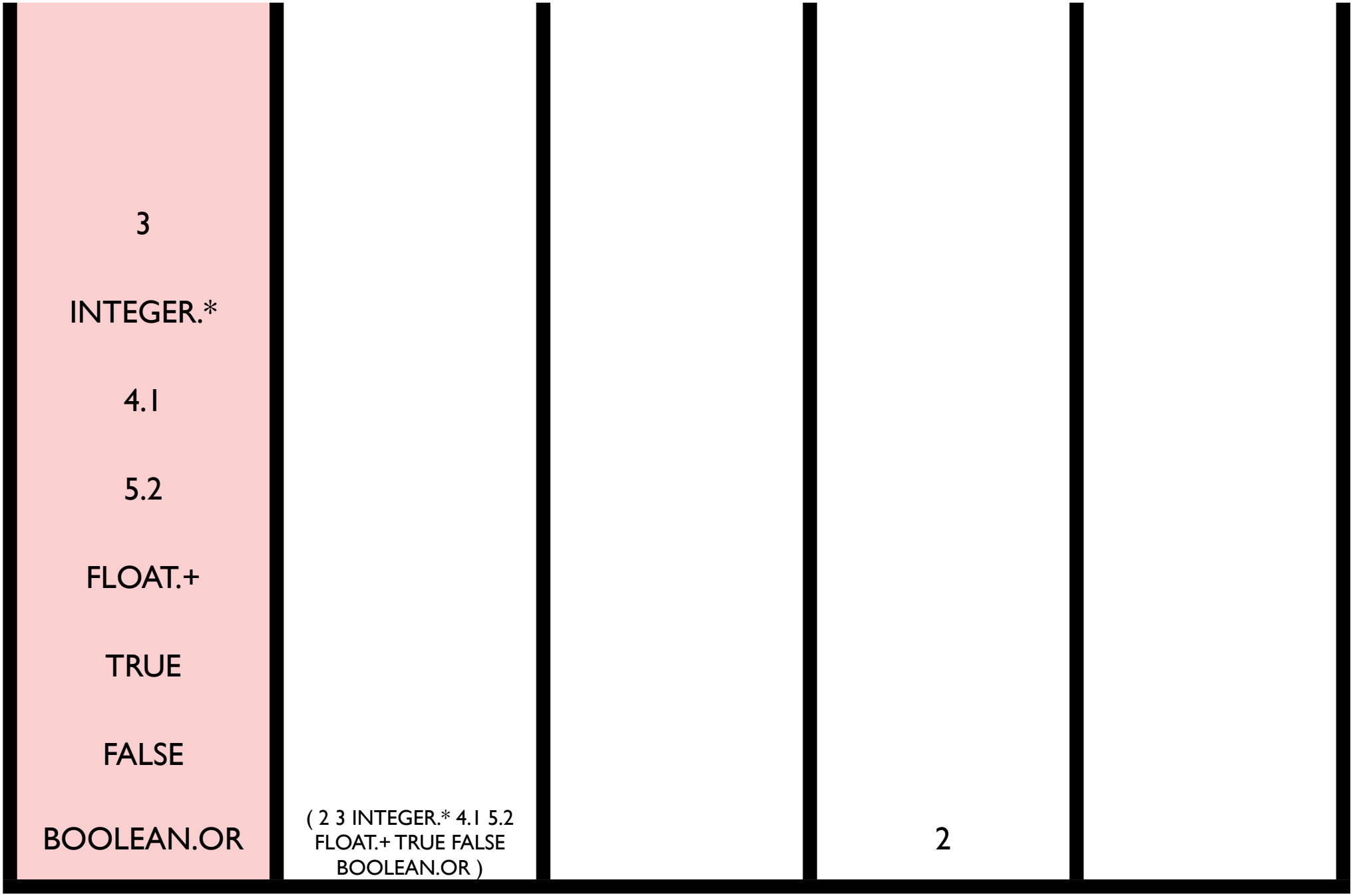
- To execute program P:

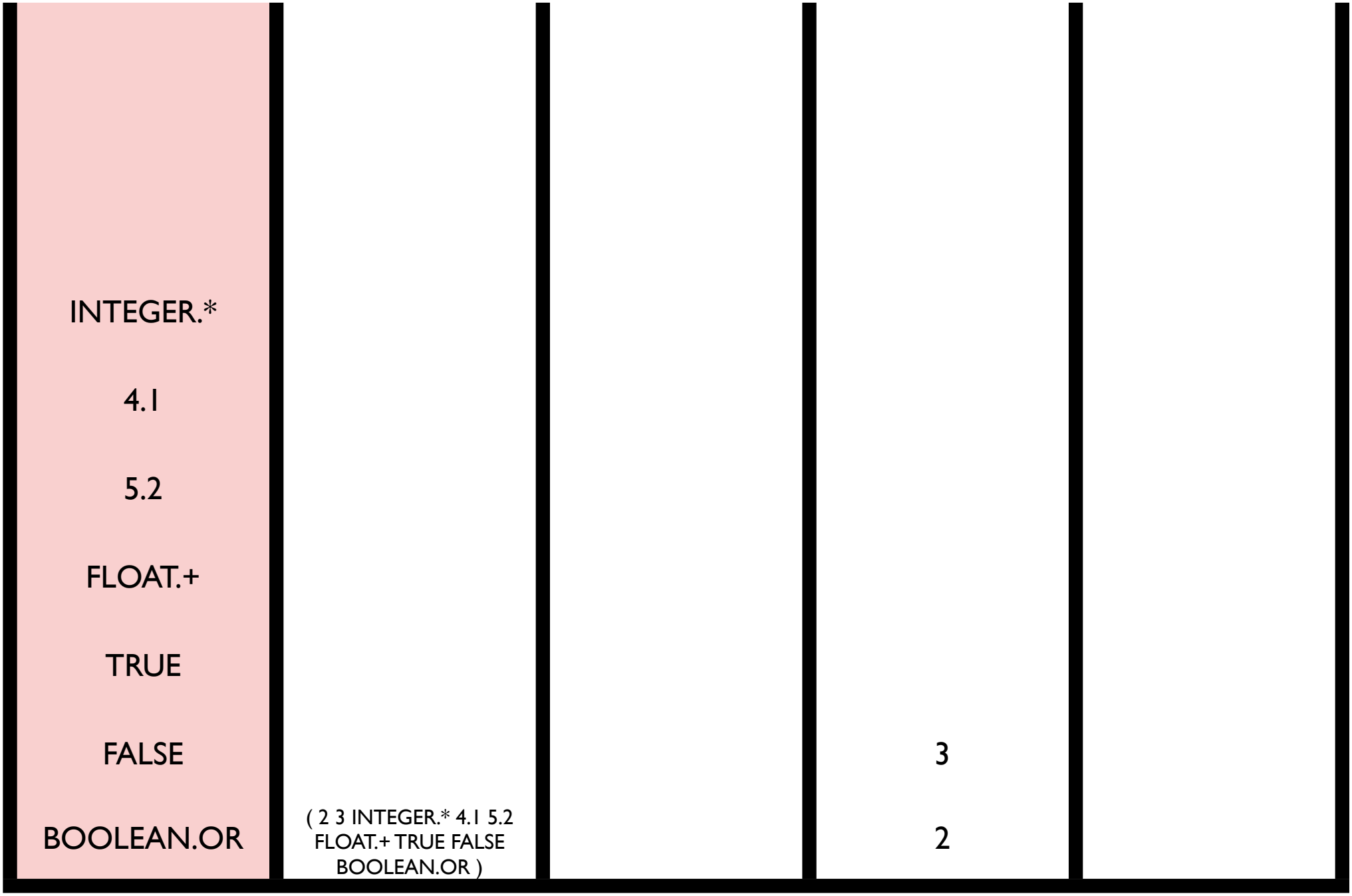
  1. Push P onto the `EXEC` stack.
  2. While the `EXEC` stack is not empty, pop and process the top element of the `EXEC` stack, E:
     (a) If E is an instruction: execute E (accessing whatever stacks are required).
     (b) If E is a literal: push E onto the appropriate stack.
     (c) If E is a list: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

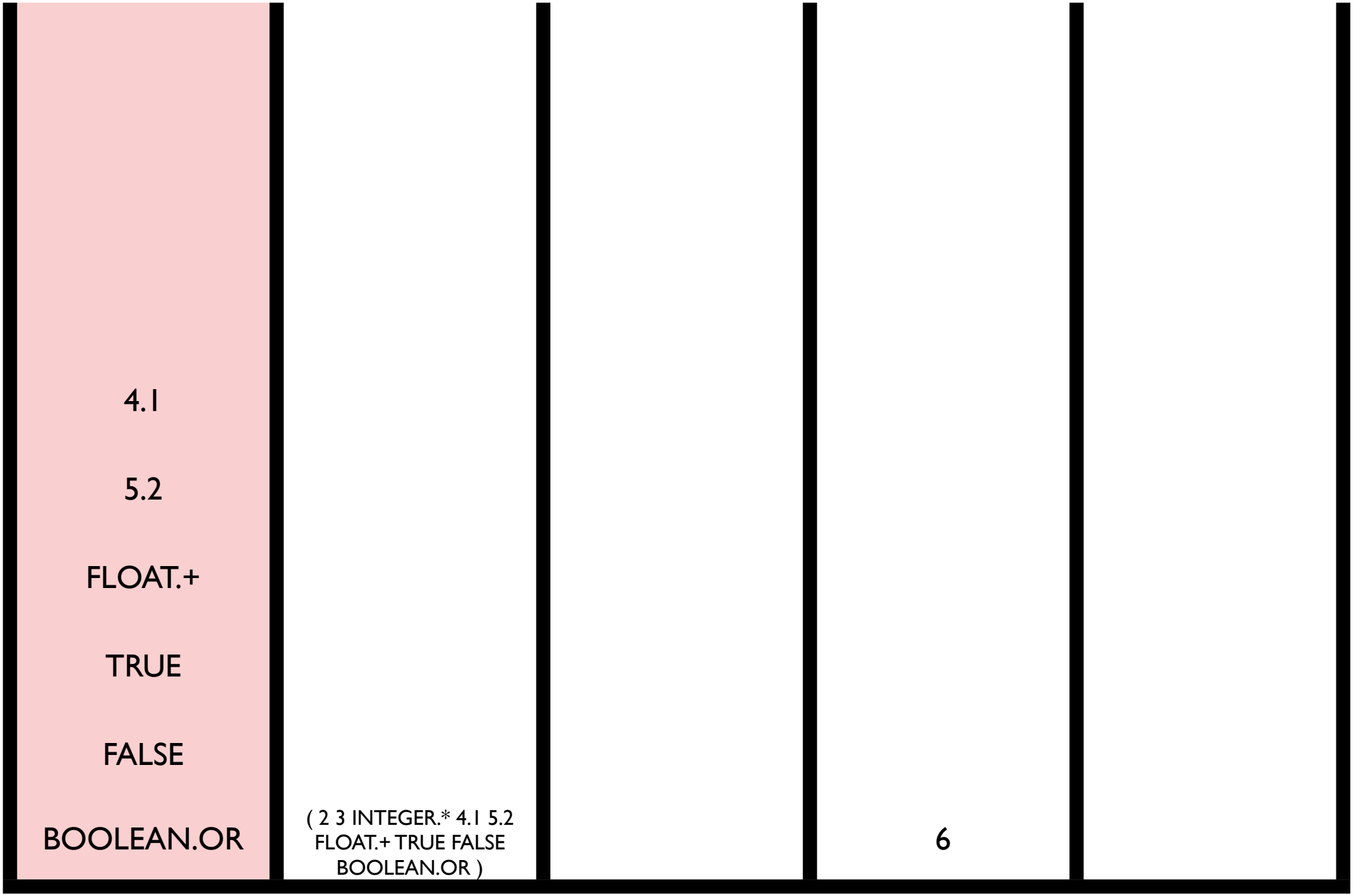
```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+
TRUE FALSE BOOLEAN.OR )
```

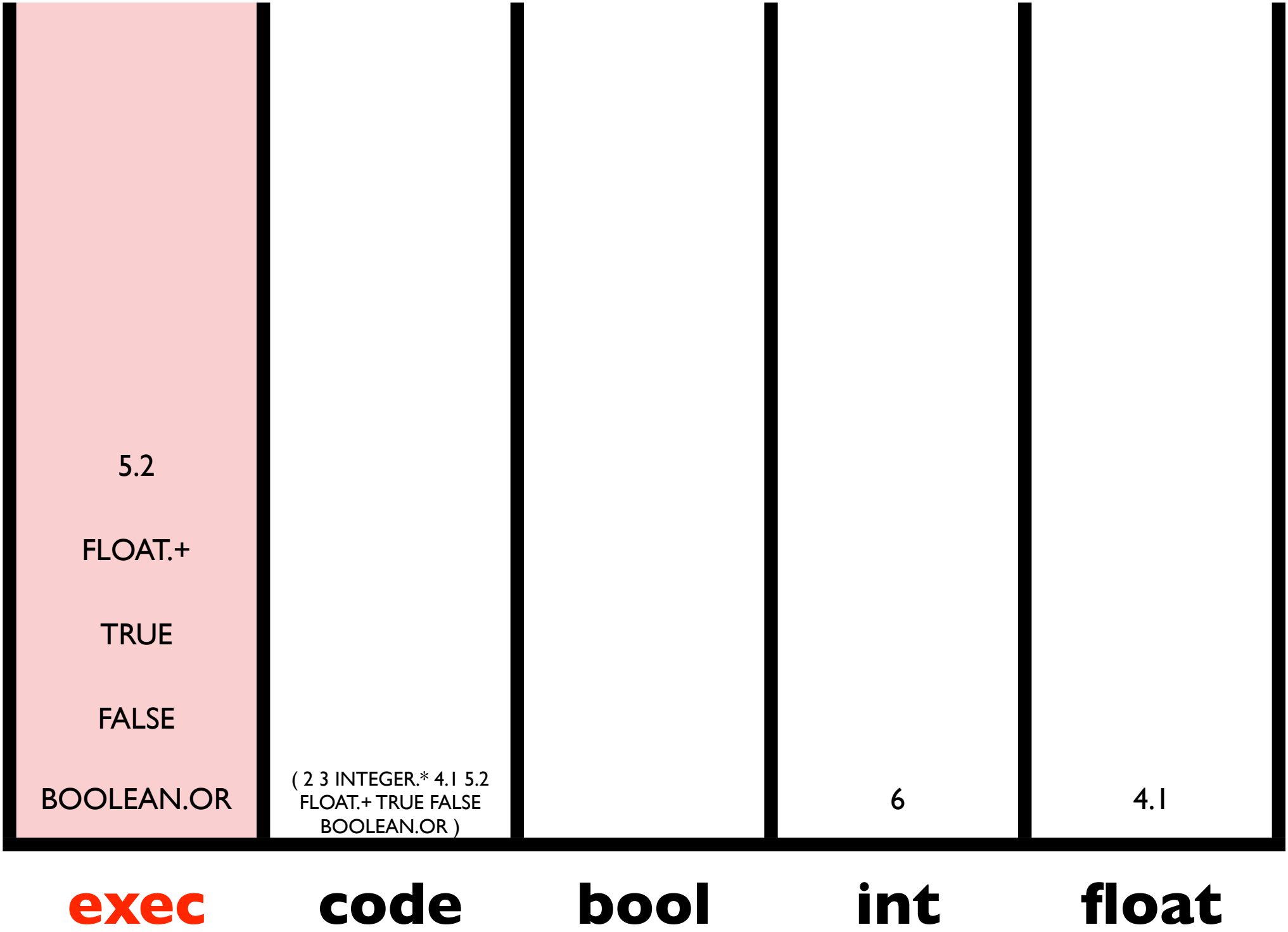


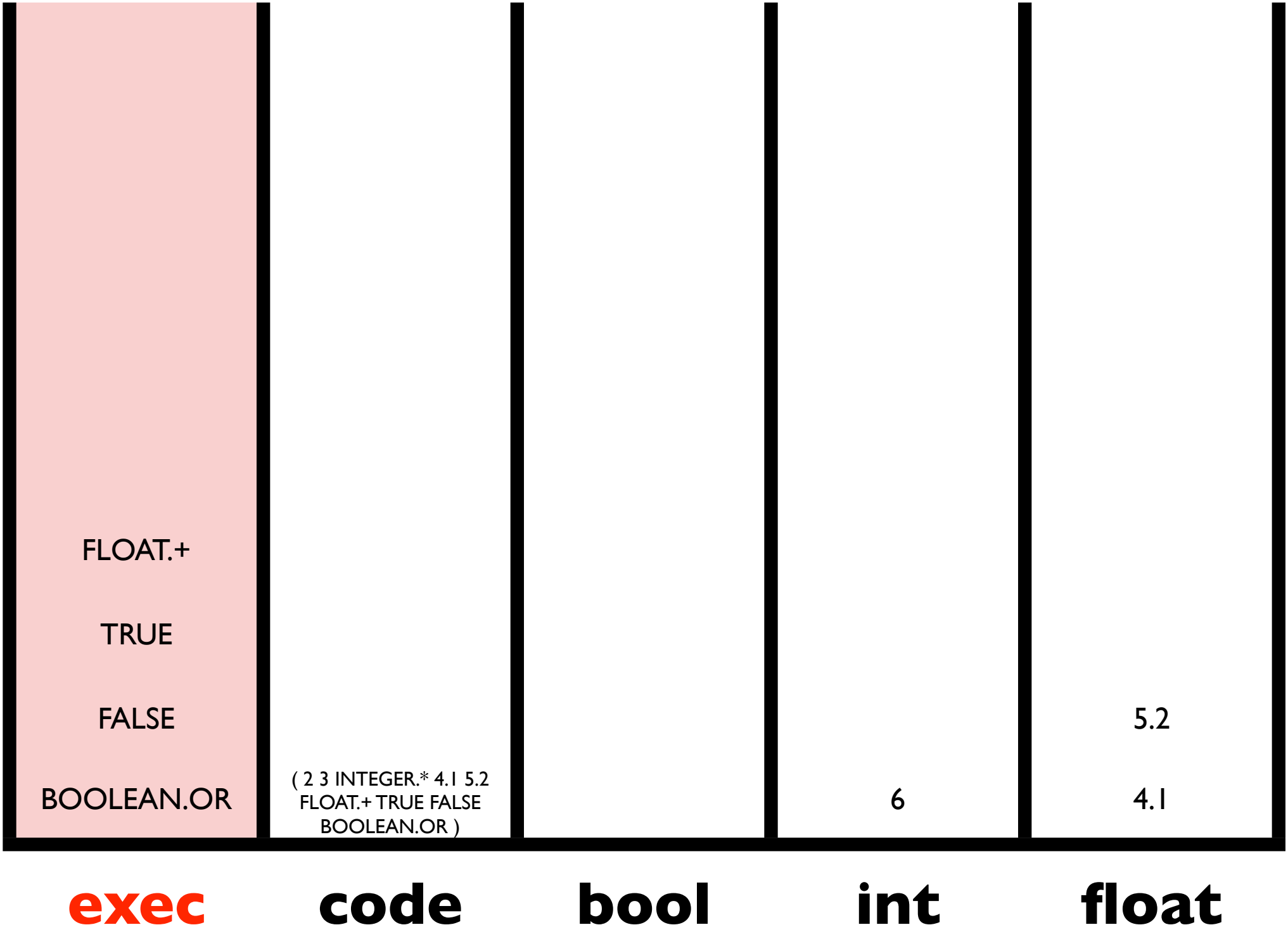

2
3
INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
exec
code
bool
int
float

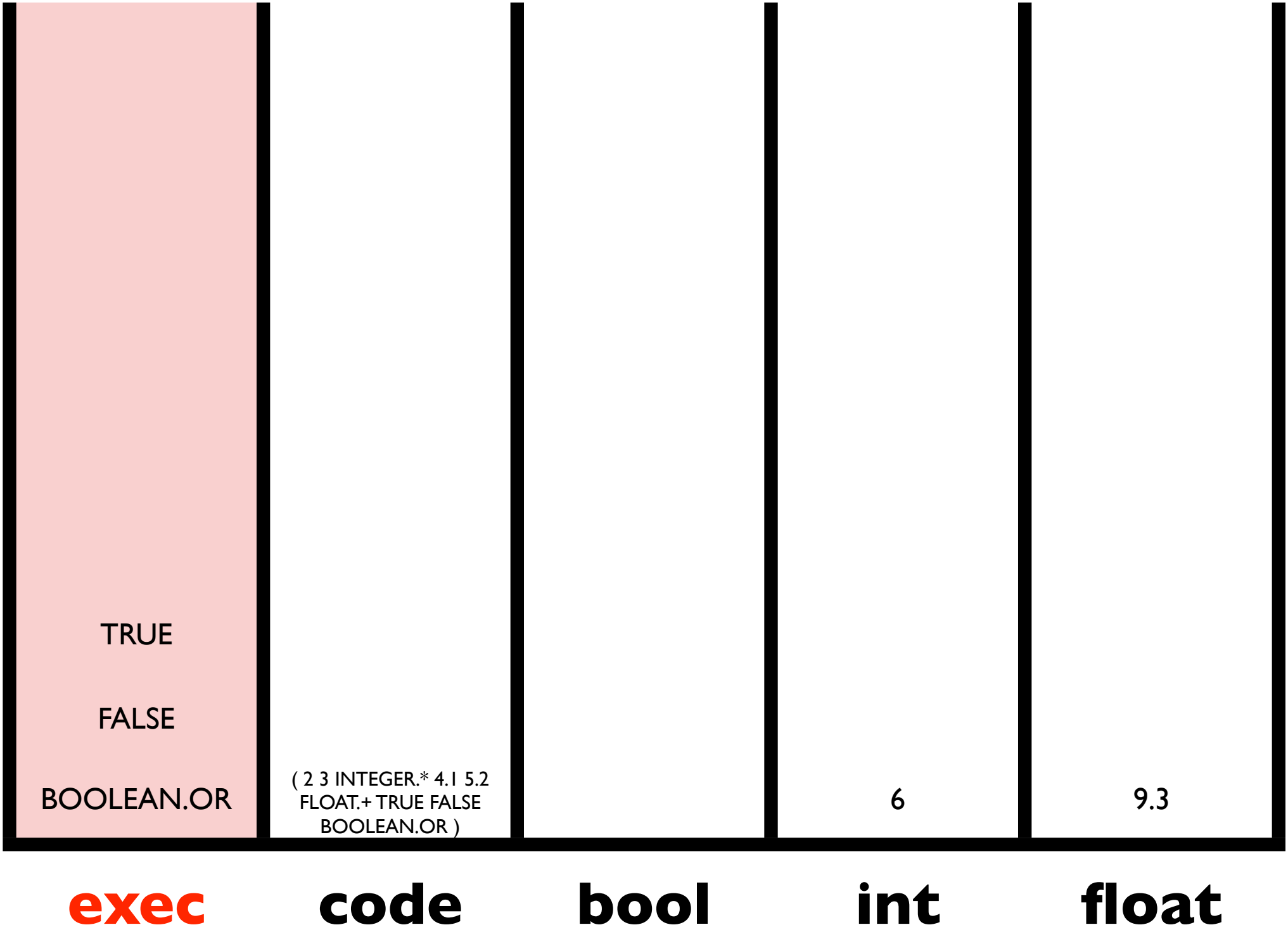

3
INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
2
exec
code
bool
int
float

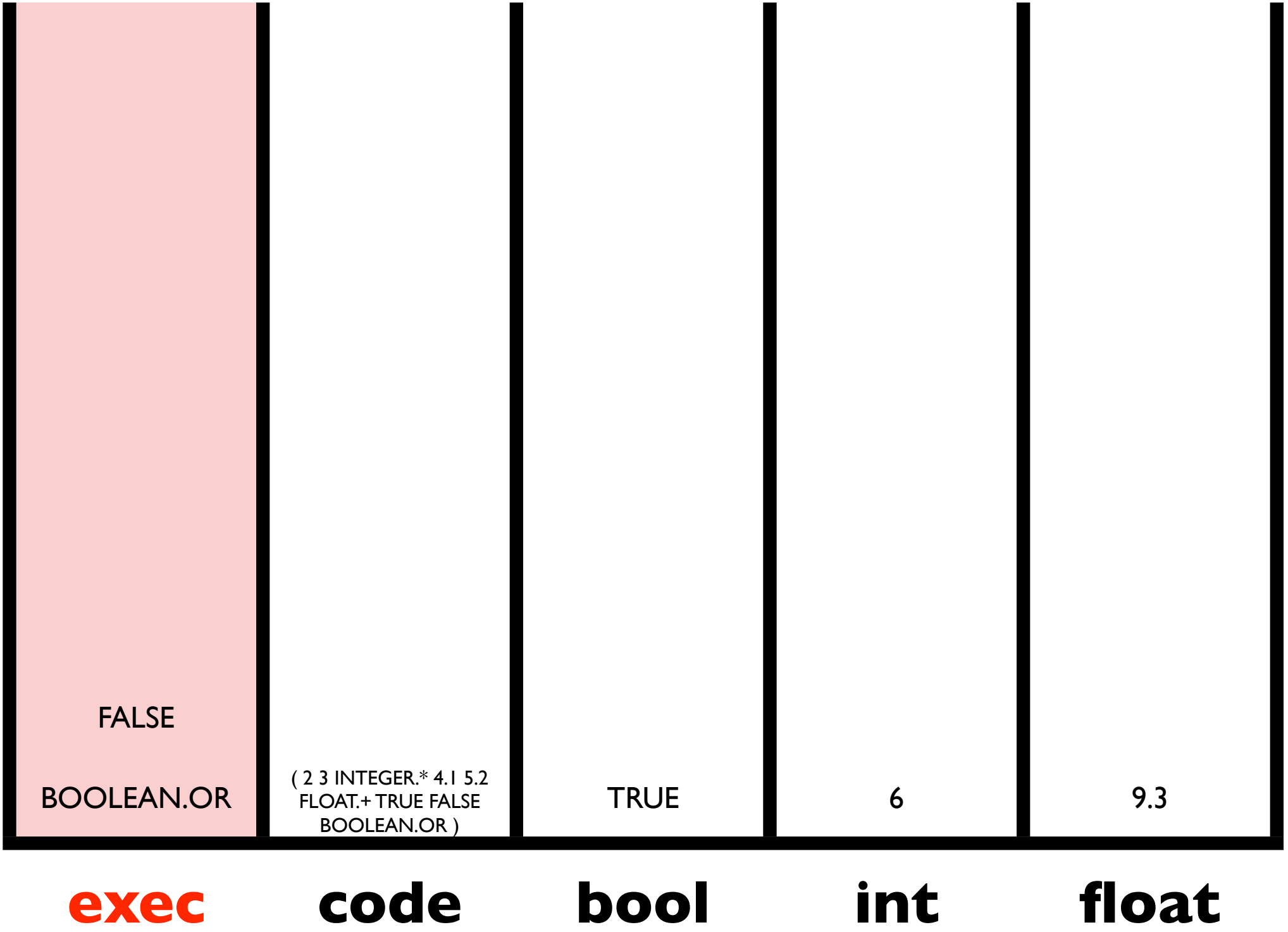

INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
3
2
exec
code
bool
int
float

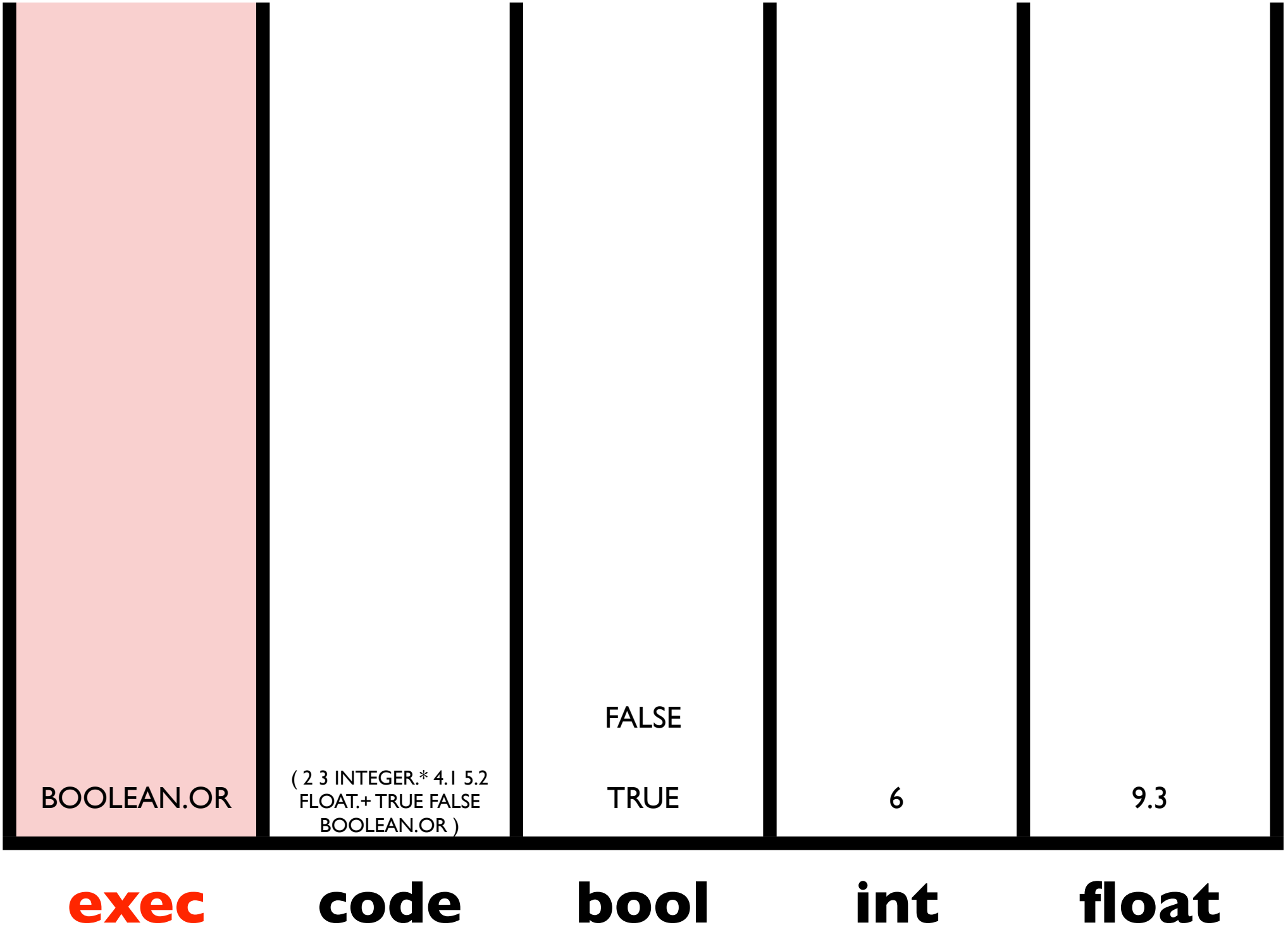

4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
6
exec
code
bool
int
float

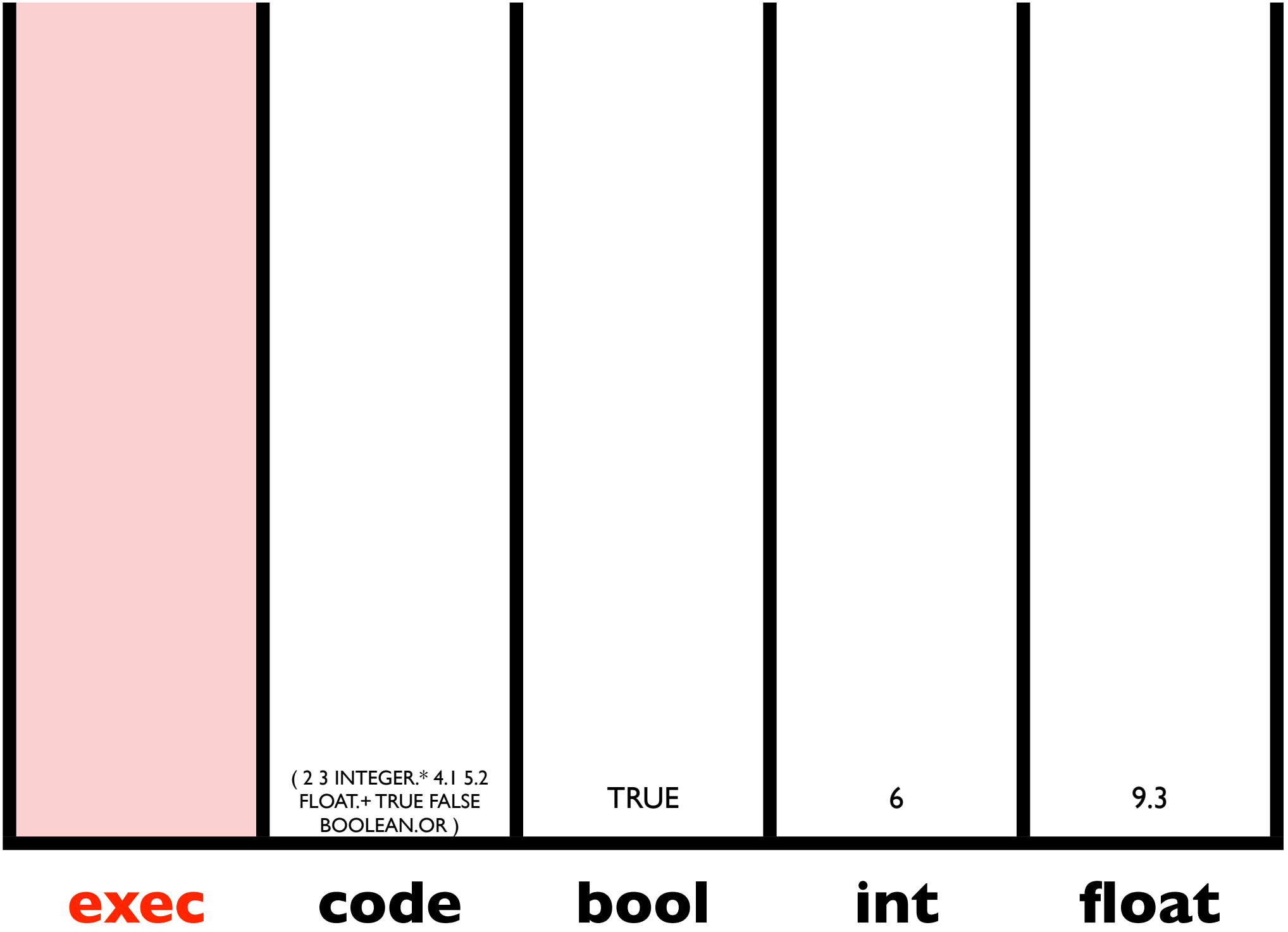

5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
6
4.1
exec
code
bool
int
float


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

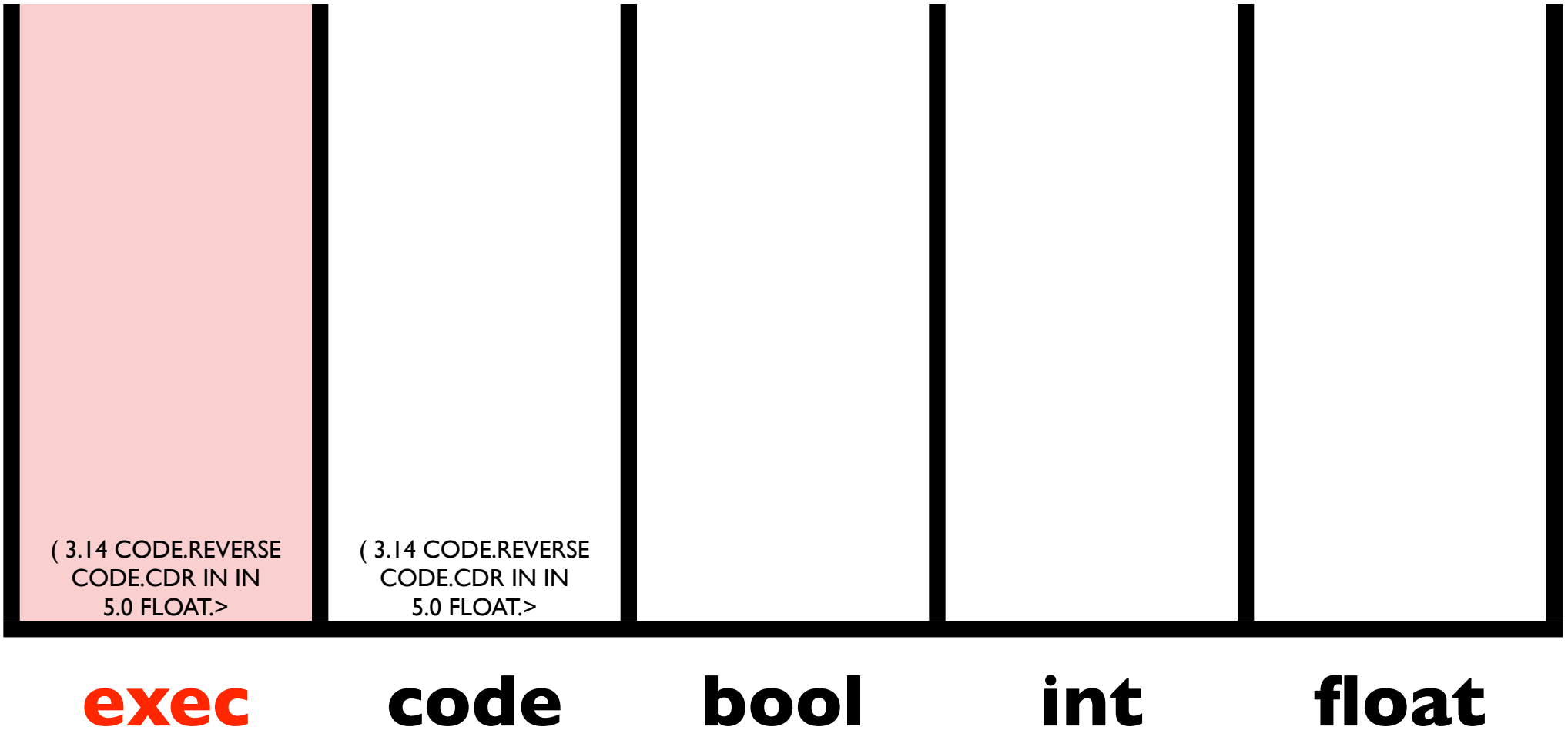
Same Results

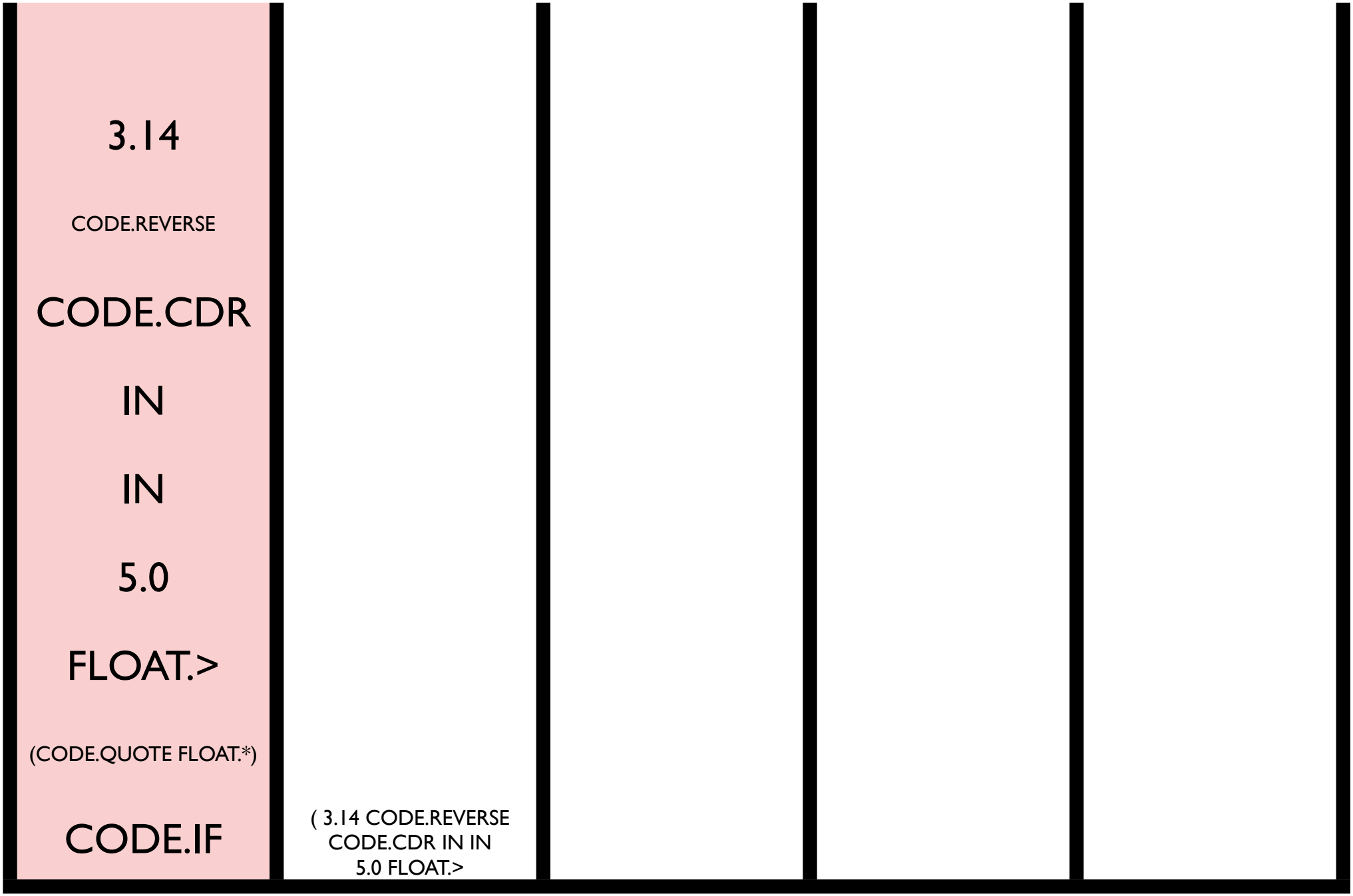
```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+
    TRUE FALSE BOOLEAN.OR )
```

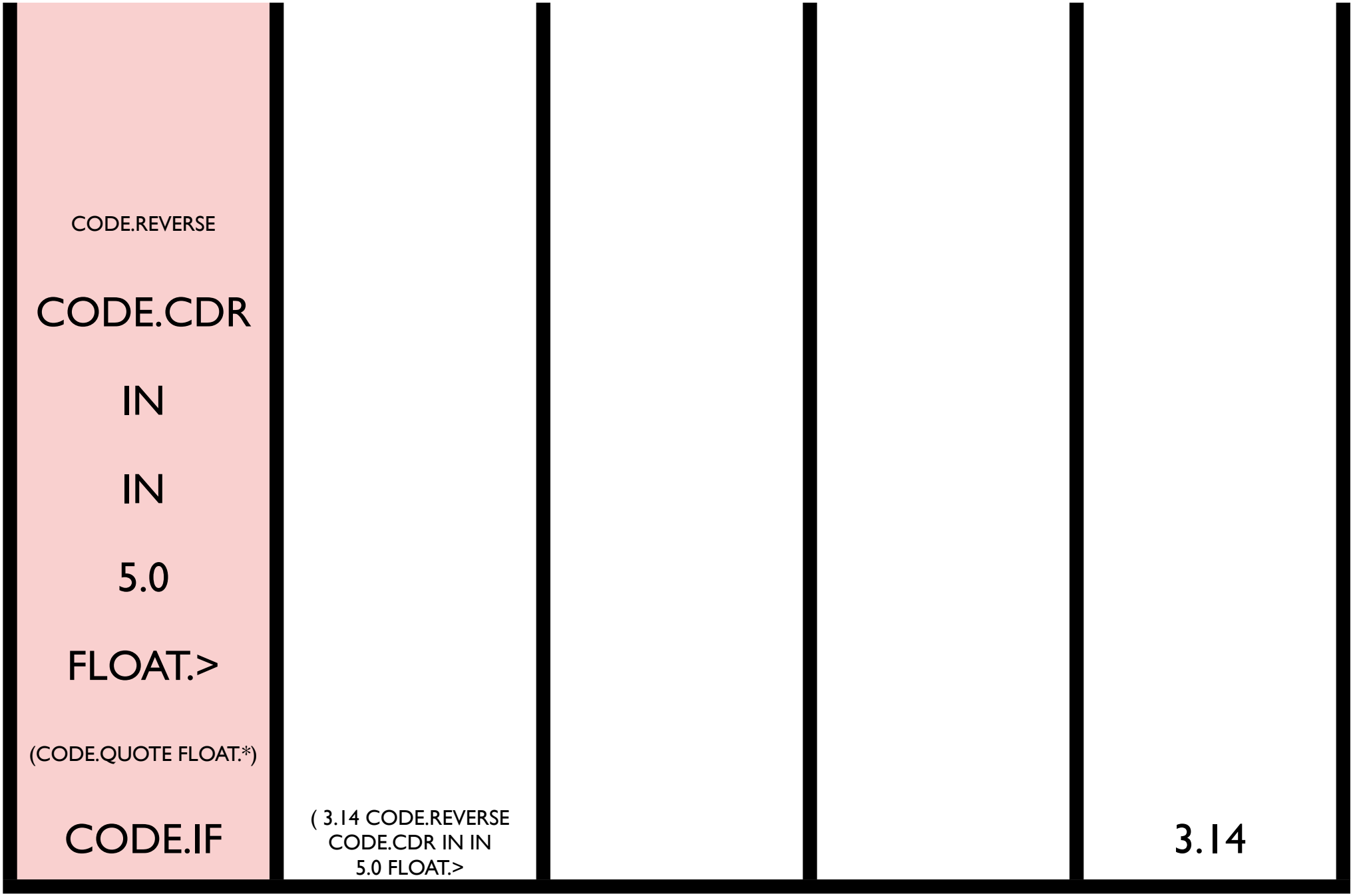
```
( 2 BOOLEAN.AND 4.1 TRUE INTEGER./ FALSE
  3 5.2 BOOLEAN.OR INTEGER.* FLOAT.+ )
```

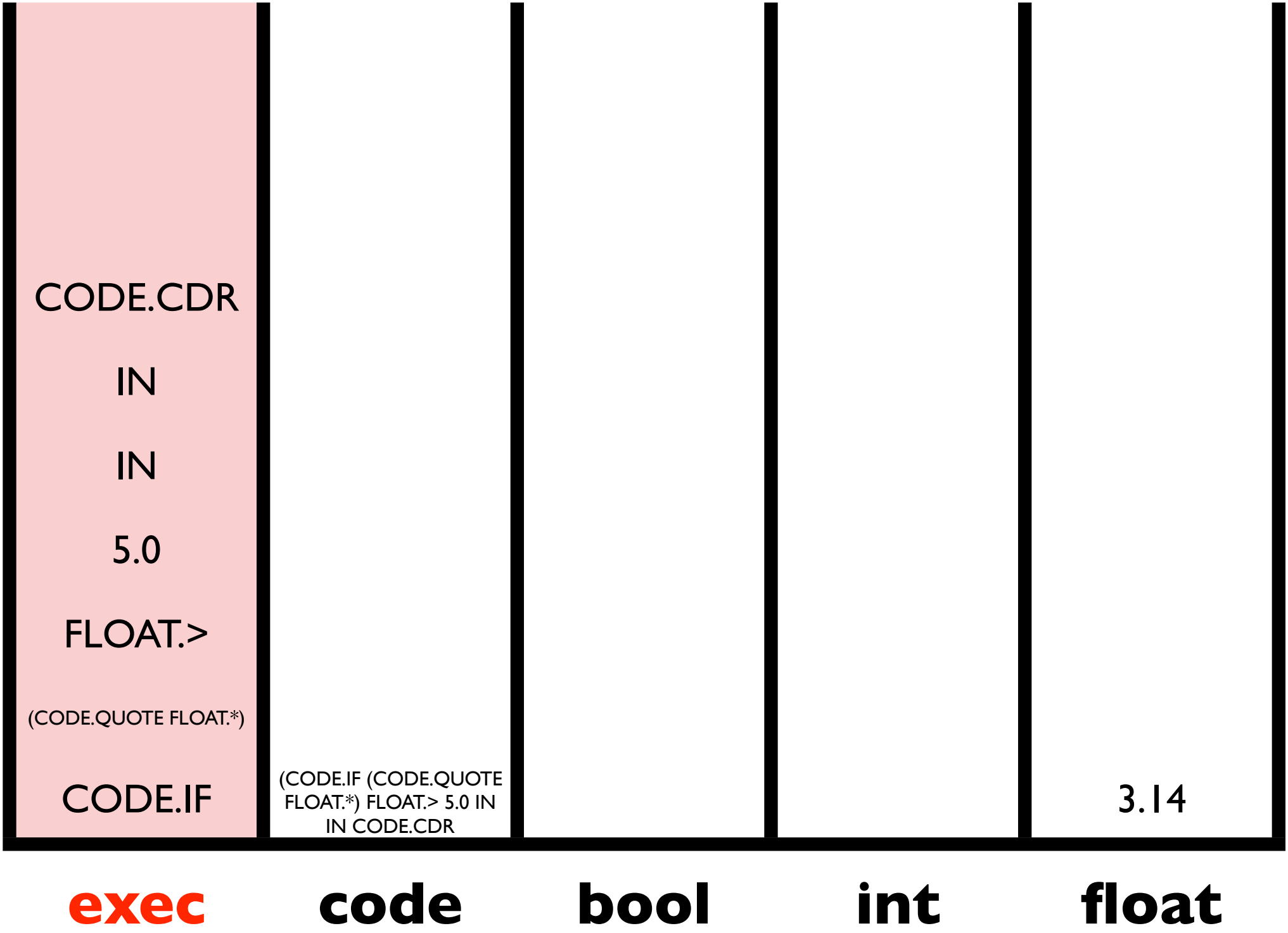
```
( 3.14 CODE.REVERSE CODE.CDR IN IN 5.0
FLOAT.> (CODE.QUOTE FLOAT.*) CODE.IF )
```

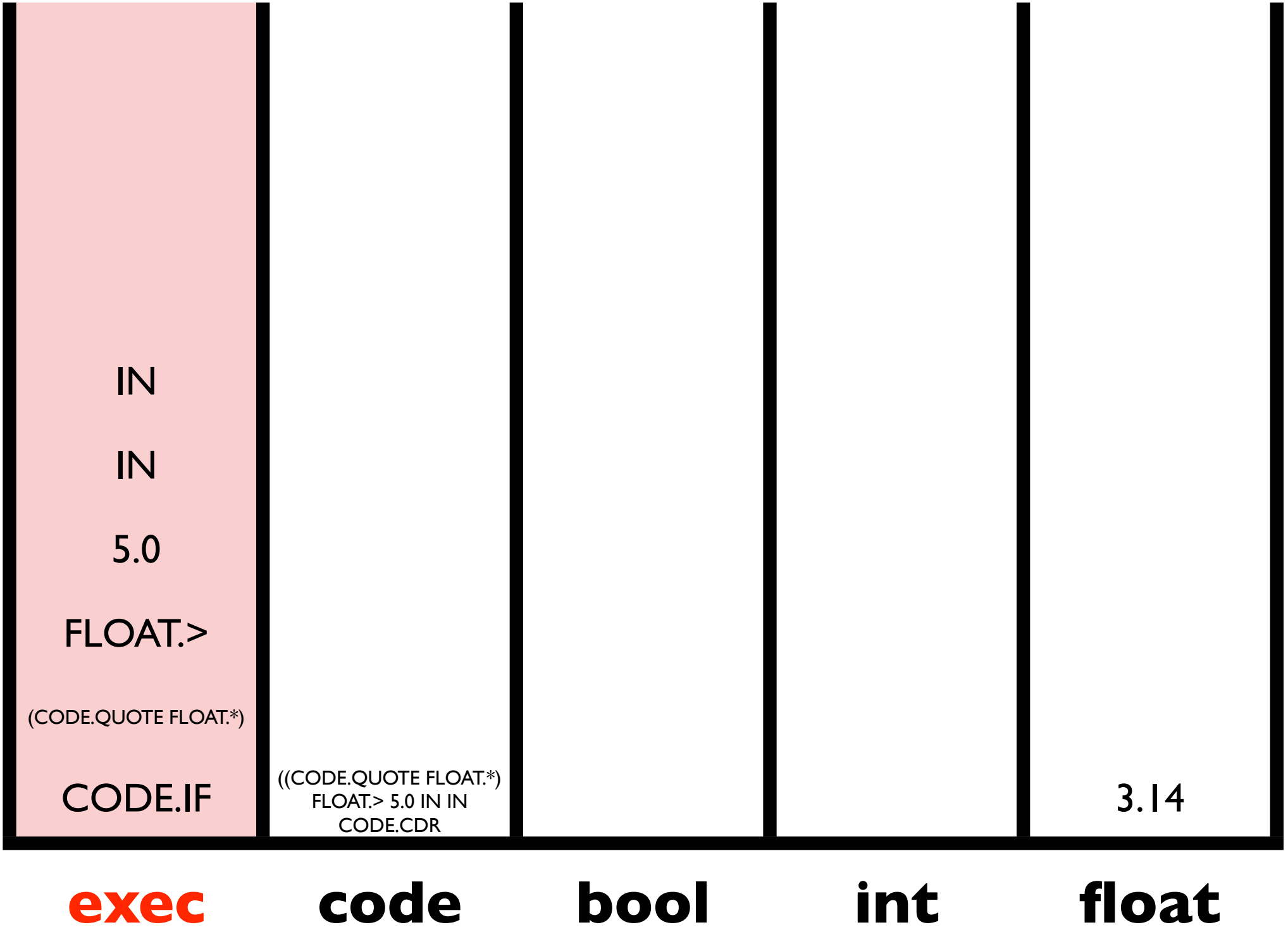
IN=4.0

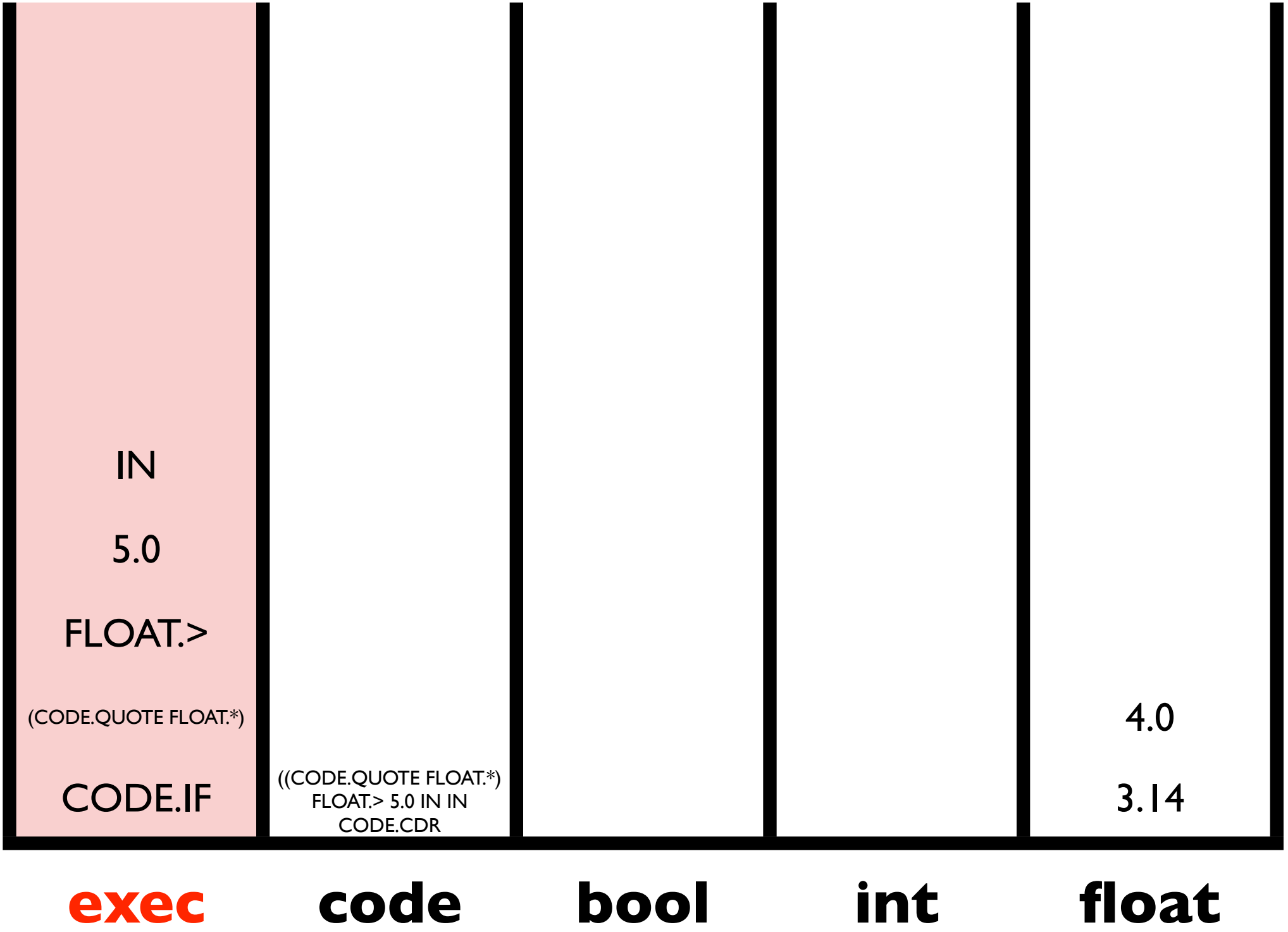


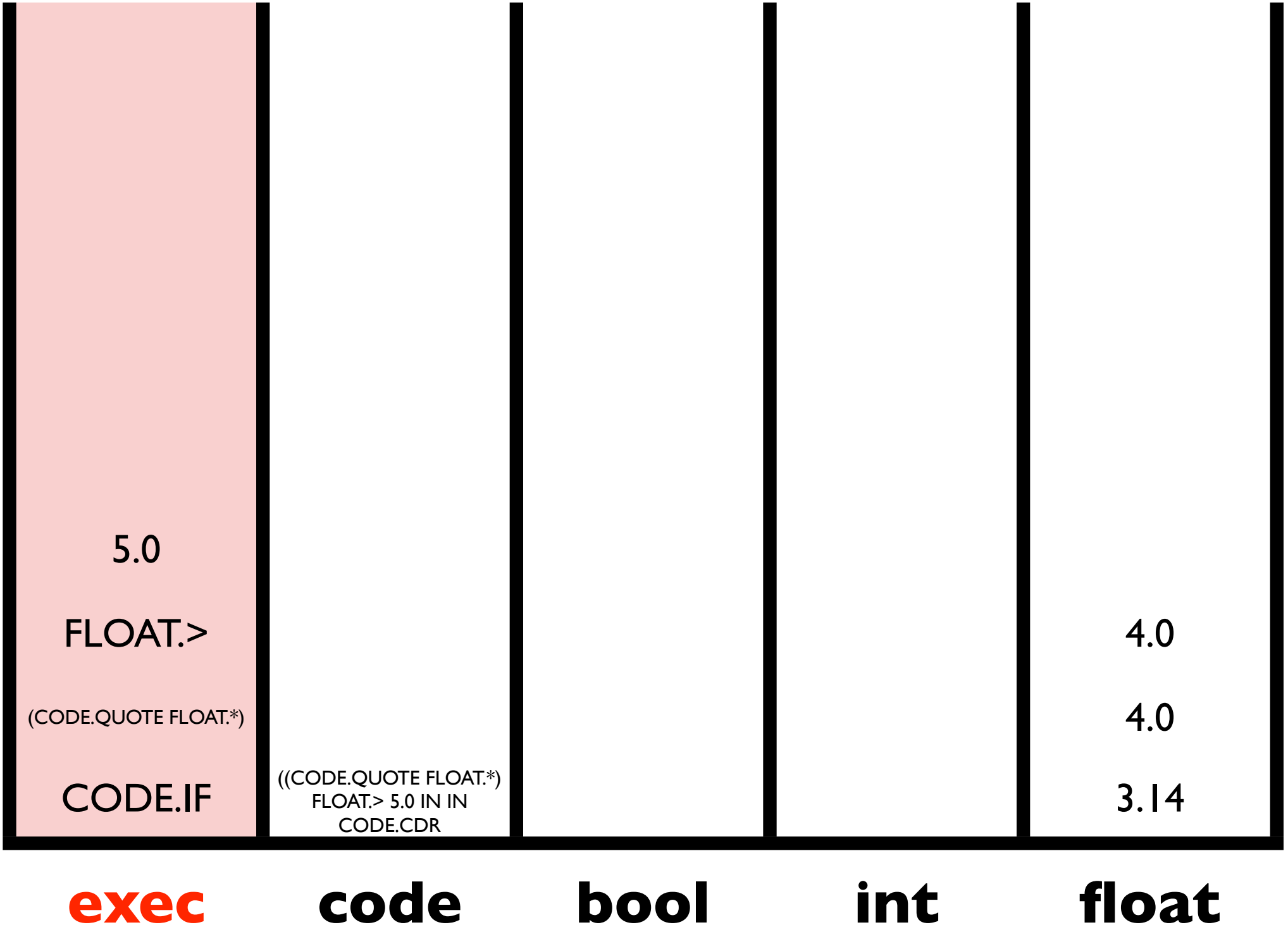

3.14
CODE.REVERSE
CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
(3.14 CODE.REVERSE
CODE.CDR IN IN
5.0 FLOAT.>
exec
code
bool
int
float

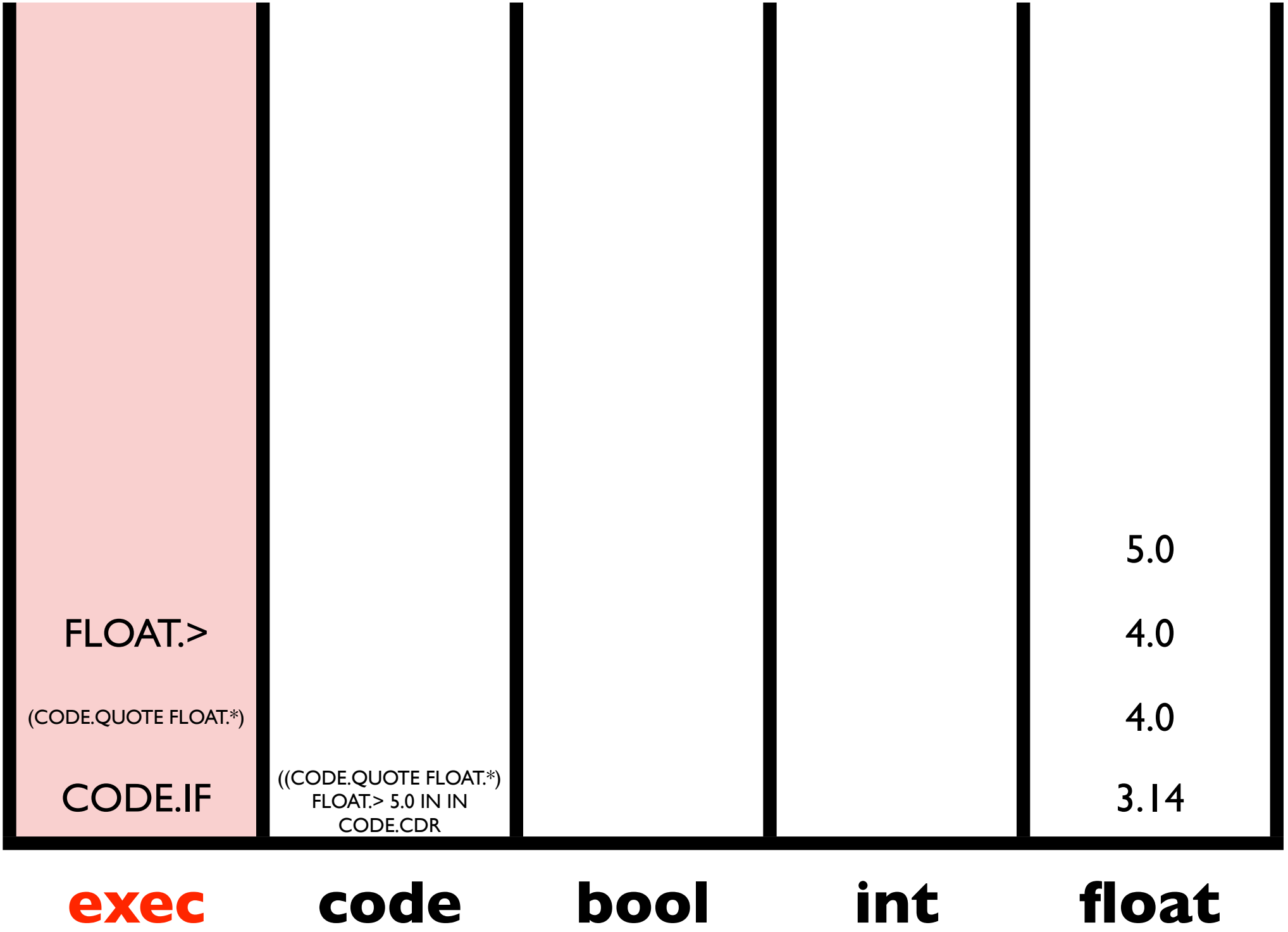

CODE.REVERSE
CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
(3.14 CODE.REVERSE CODE.CDR IN IN 5.0 FLOAT.>
3.14
exec
code
bool
int
float

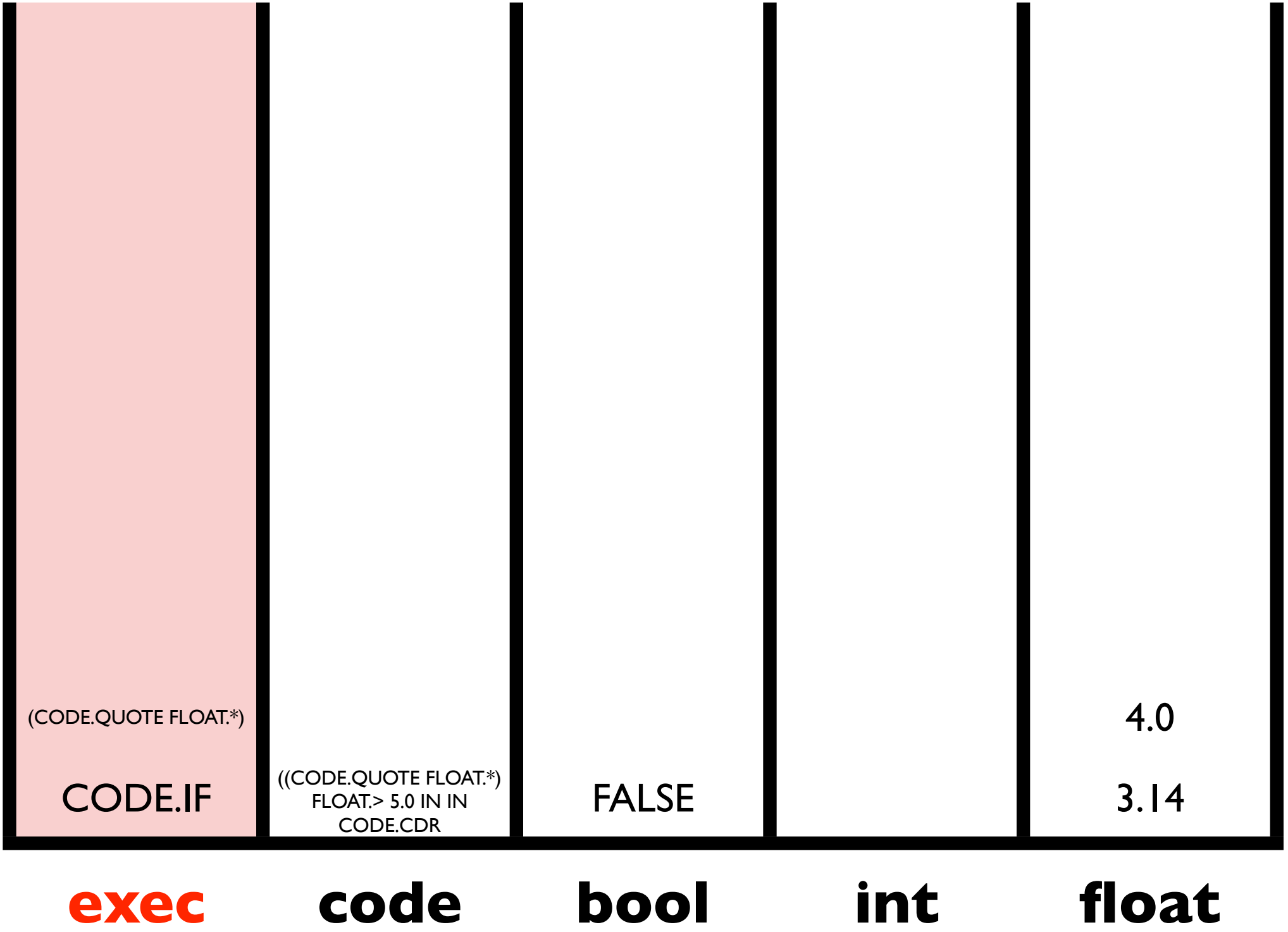

CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
(CODE.IF (CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
3.14
exec
code
bool
int
float

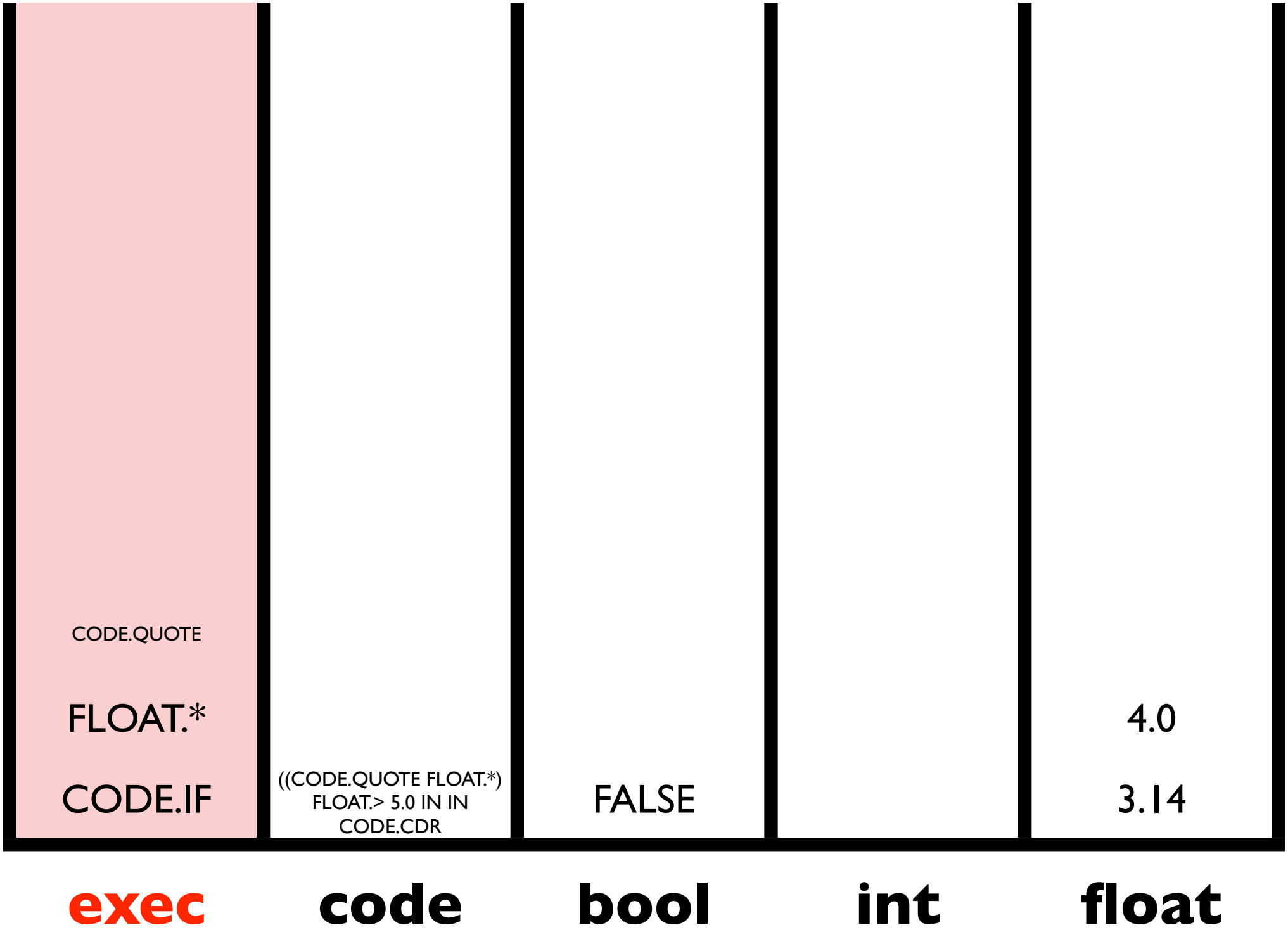

IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
3.14
exec
code
bool
int
float

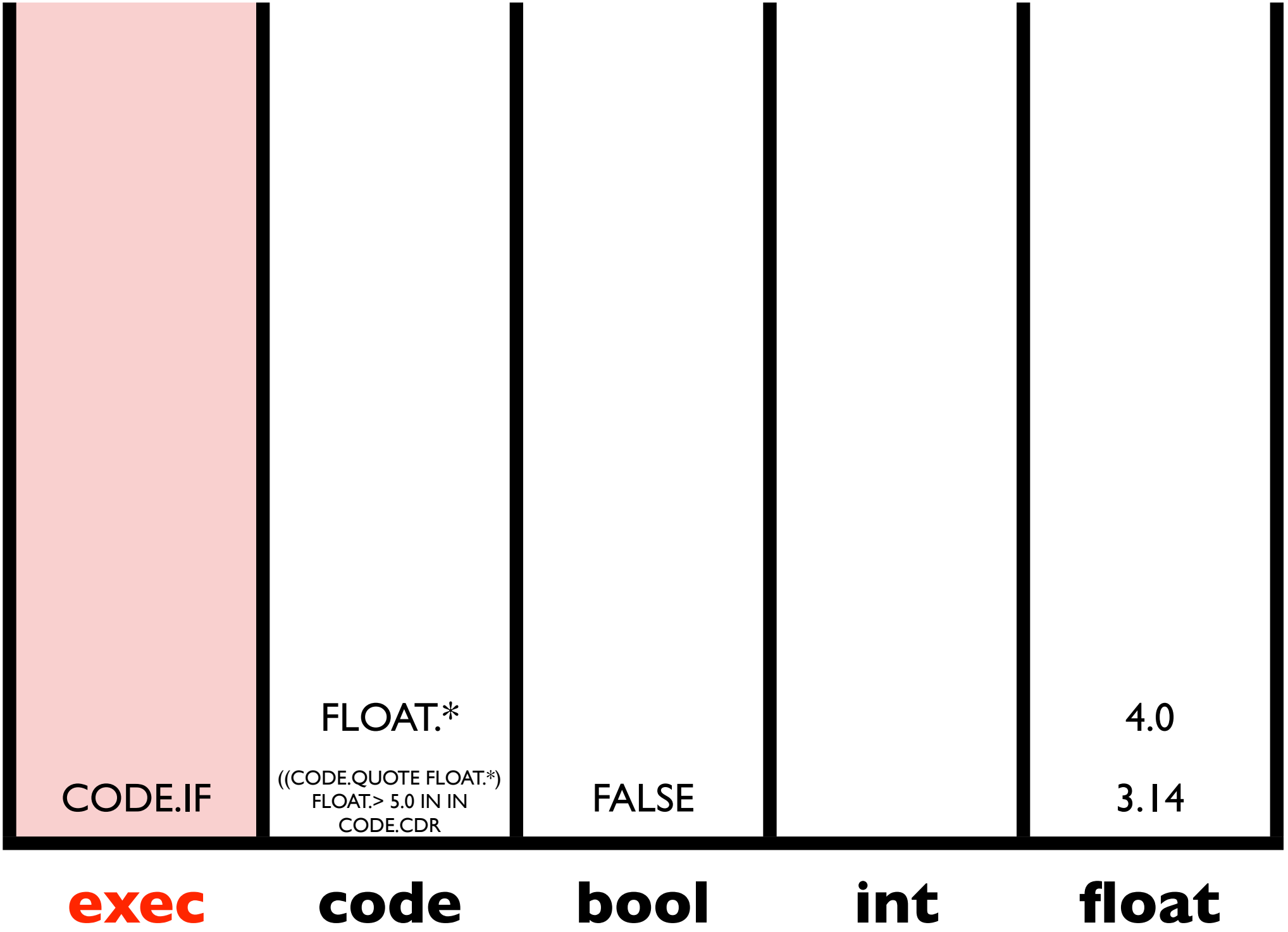

IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
4.0
3.14
exec
code
bool
int
float

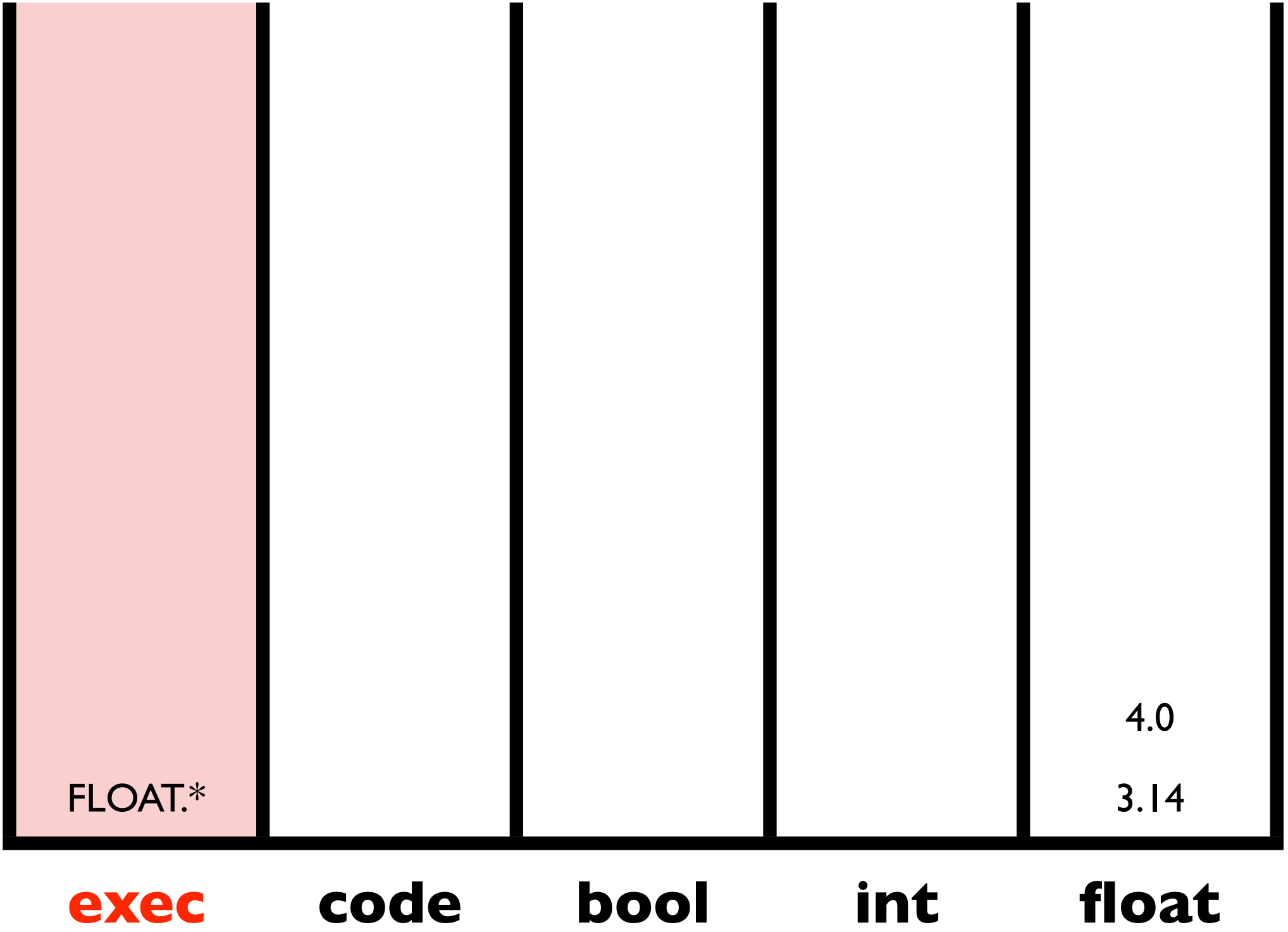

5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
4.0
4.0
3.14
exec
code
bool
int
float

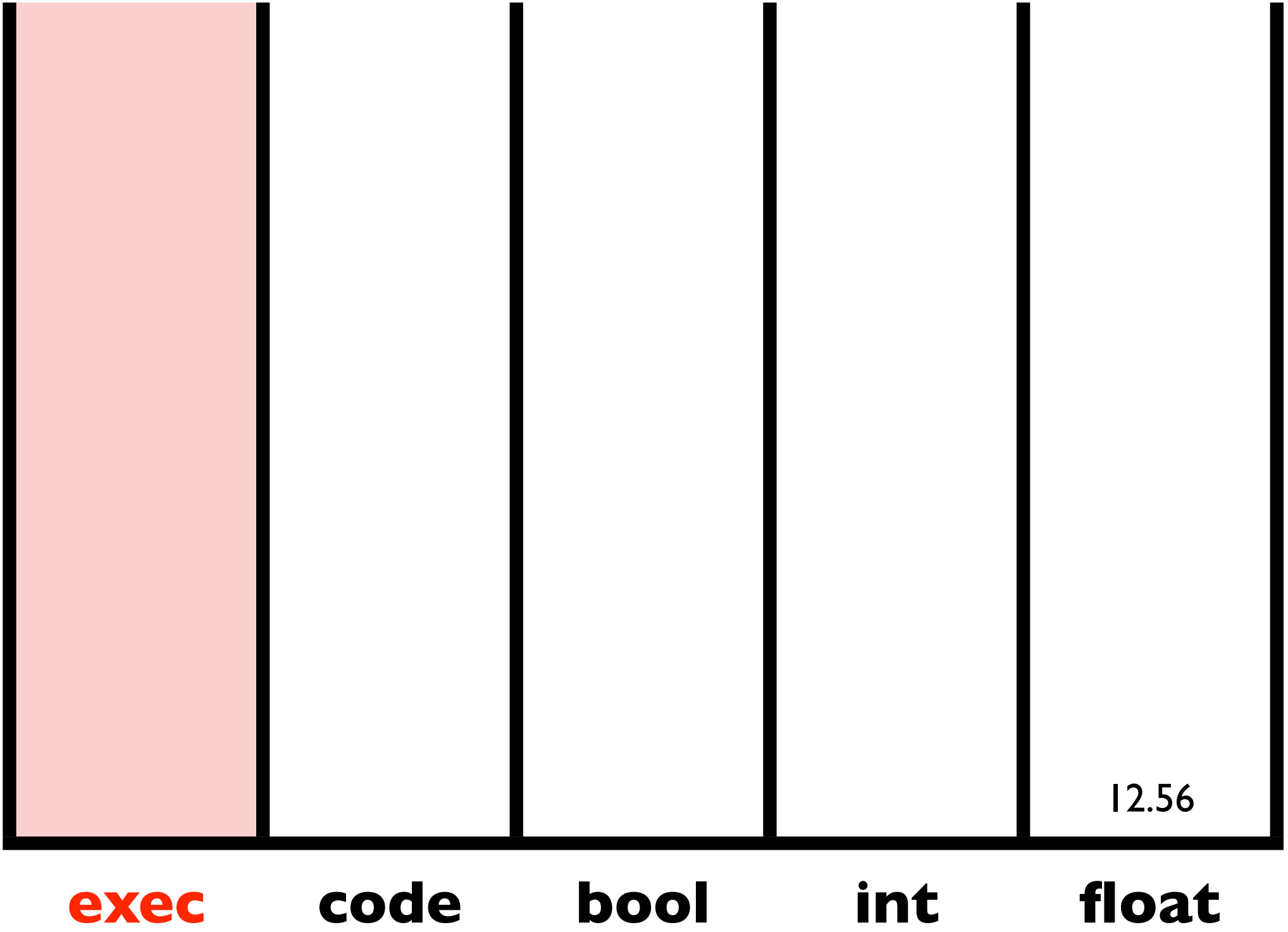

FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
5.0
4.0
4.0
3.14
exec
code
bool
int
float


(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

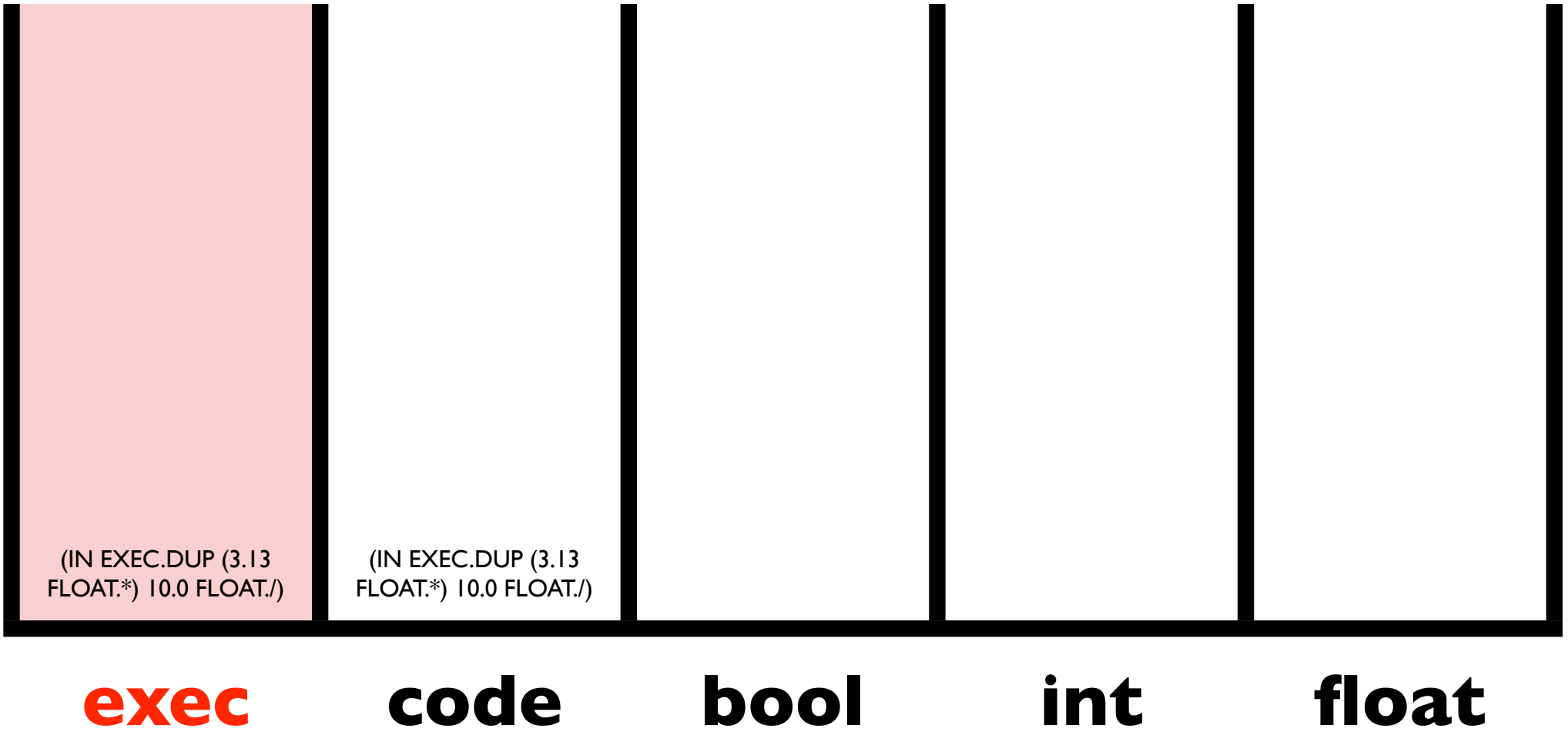

CODE.QUOTE
FLOAT.*
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
FALSE
4.0
3.14
exec
code
bool
int
float

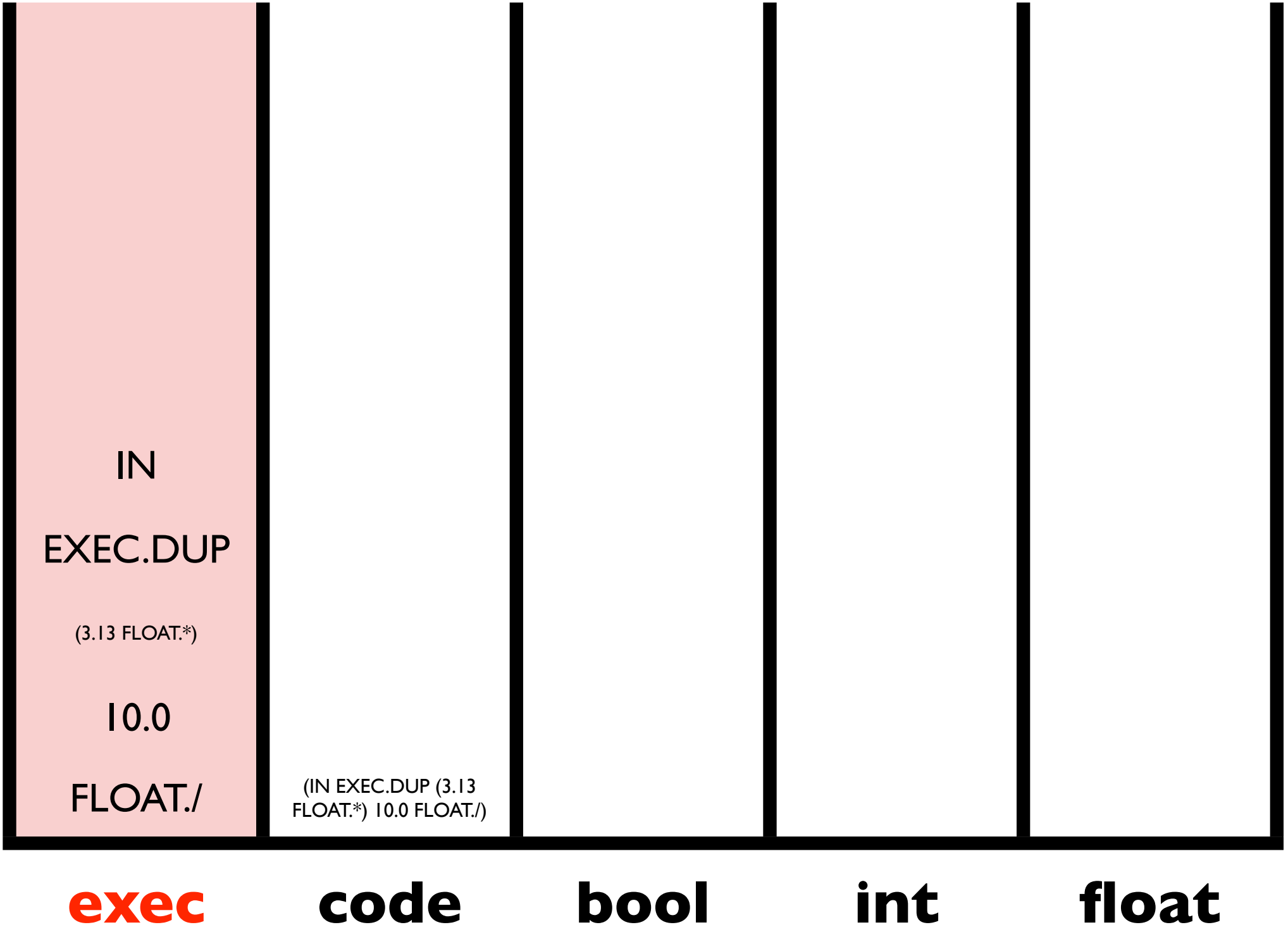

CODE.IF
FLOAT.*
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
FALSE
4.0
3.14
exec
code
bool
int
float

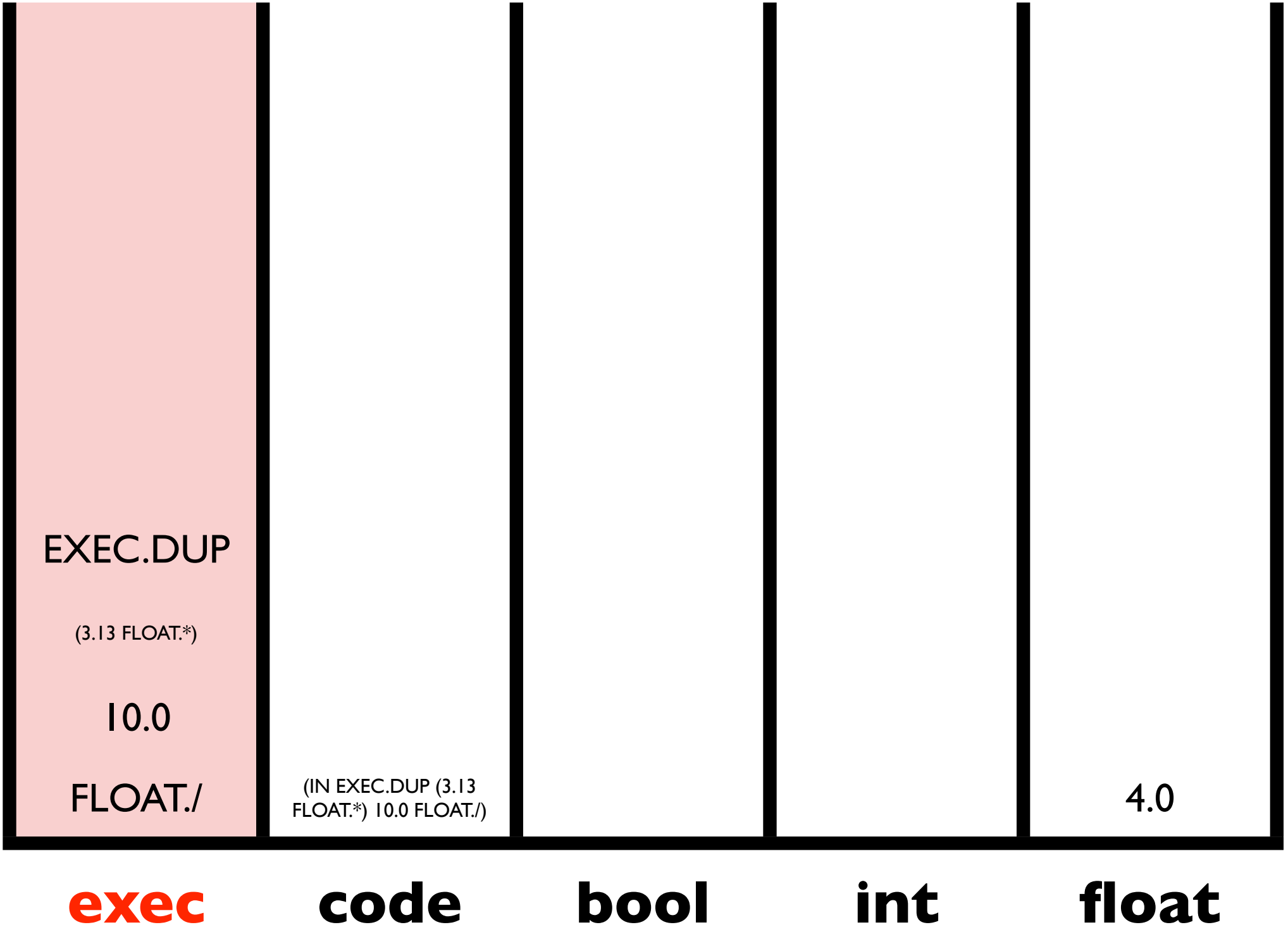

FLOAT.*
4.0
3.14
exec
code
bool
int
float

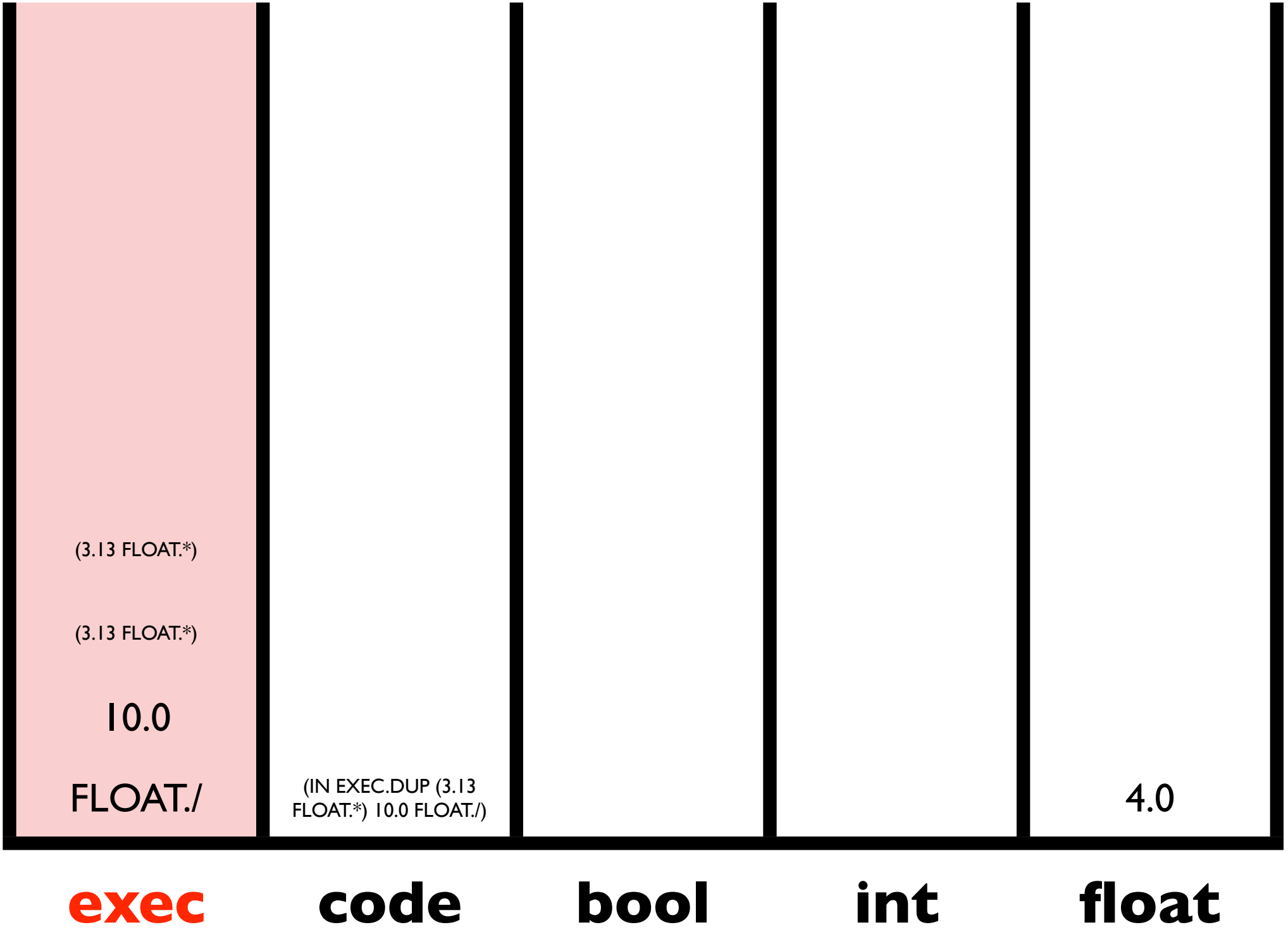

12.56
exec
code
bool
int
float

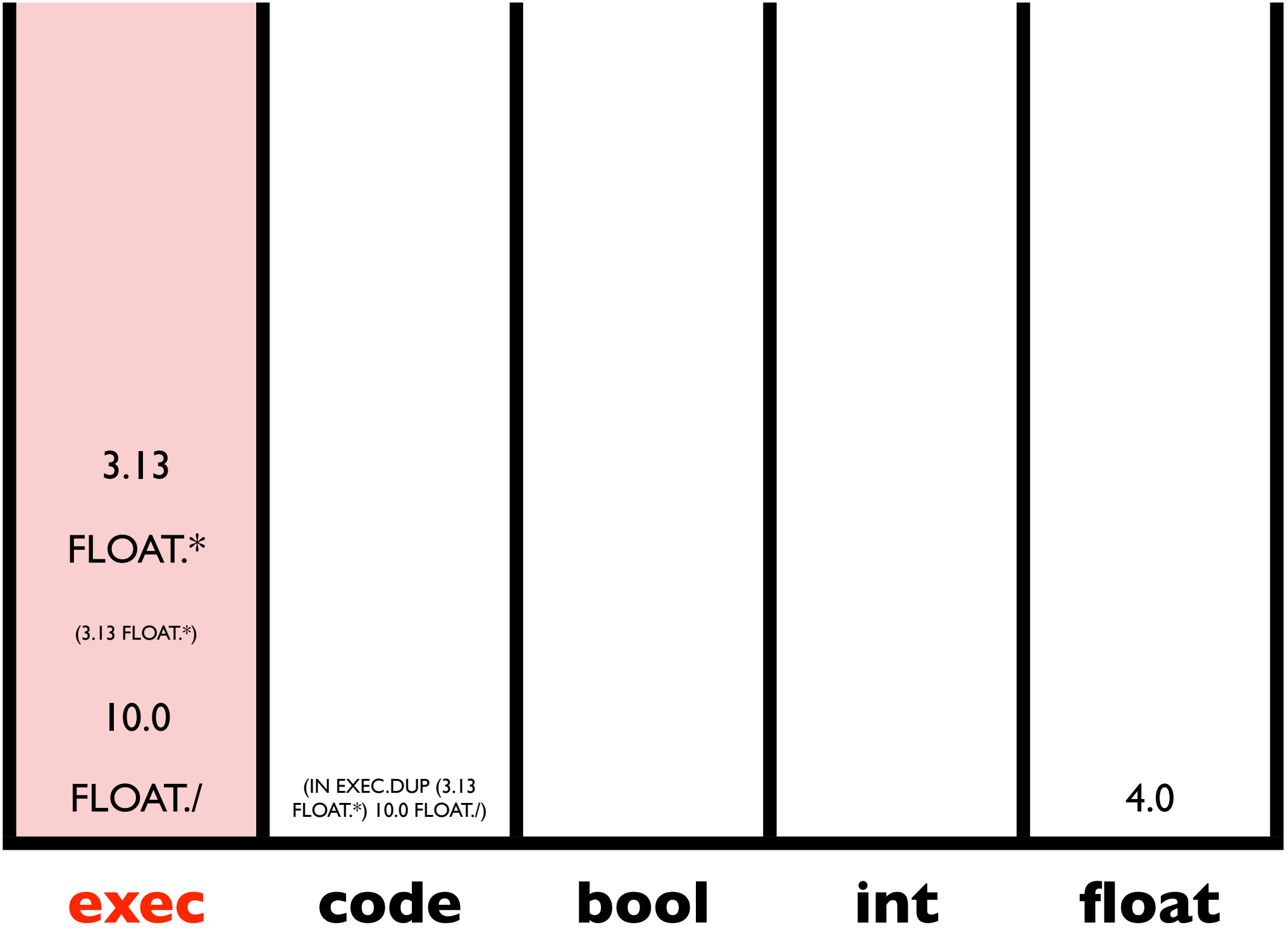
```
(IN EXEC.DUP (3.13 FLOAT.*)
10.0 FLOAT./)

IN=4.0
```

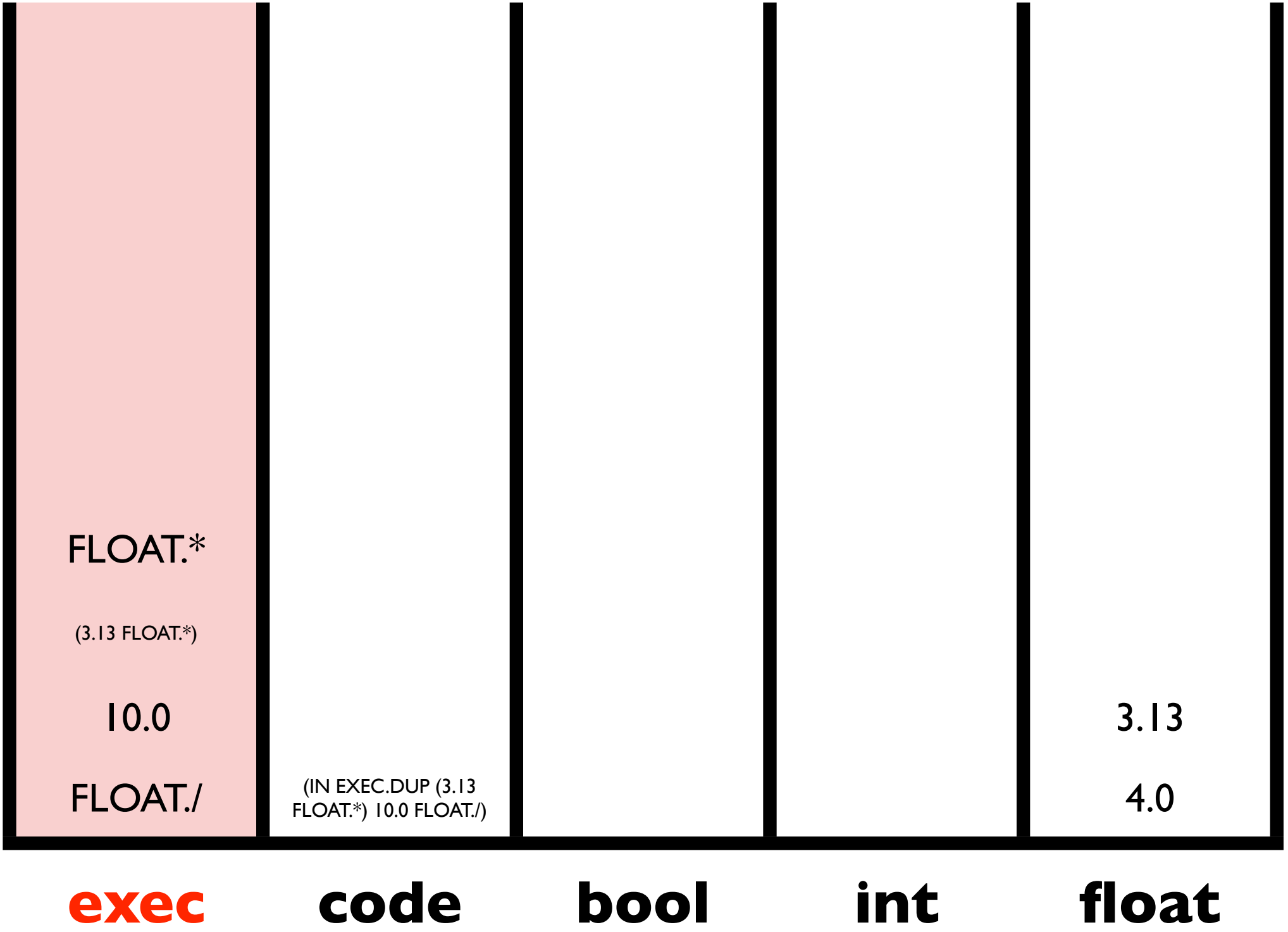


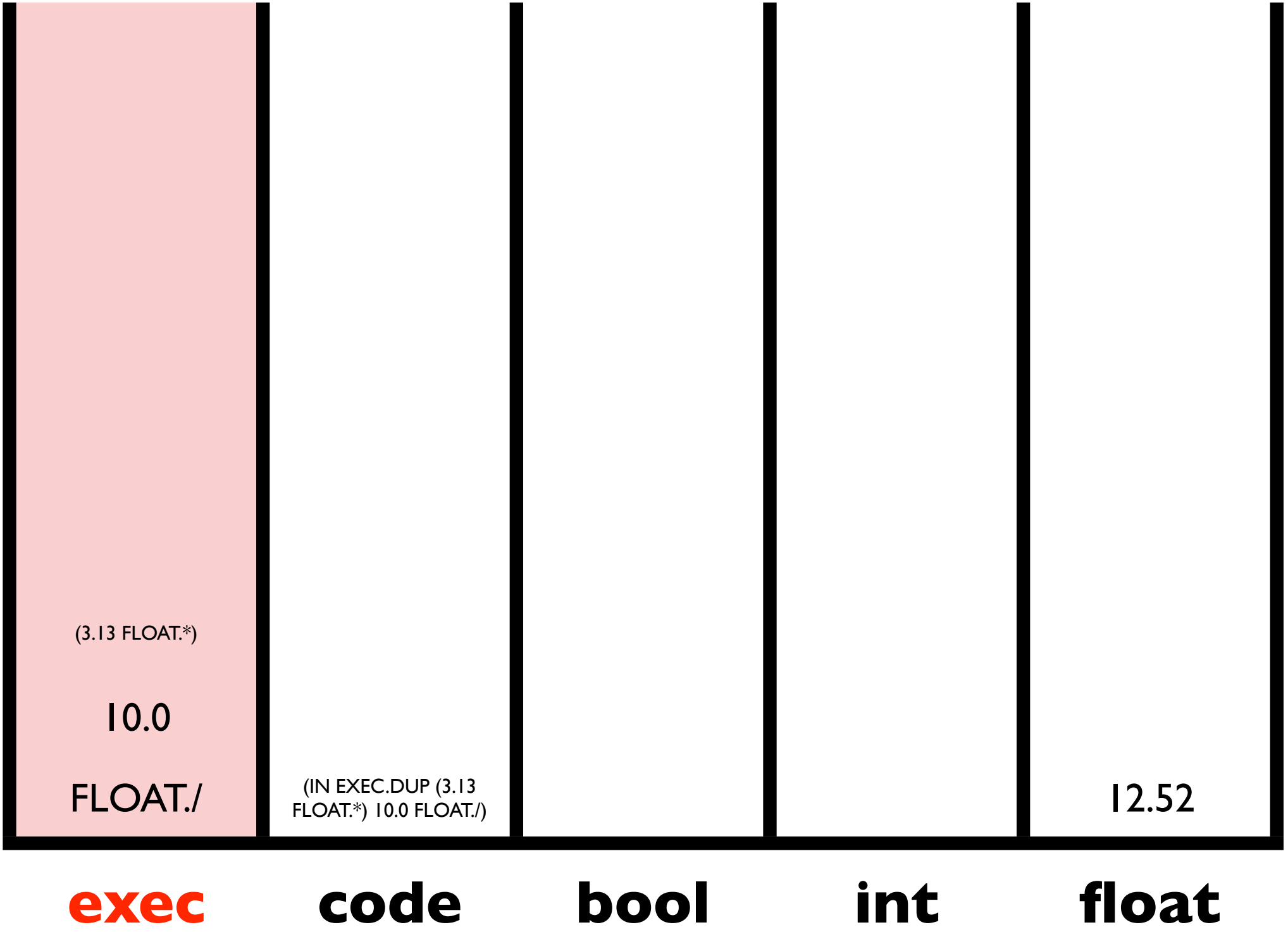

IN
EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
exec
code
bool
int
float

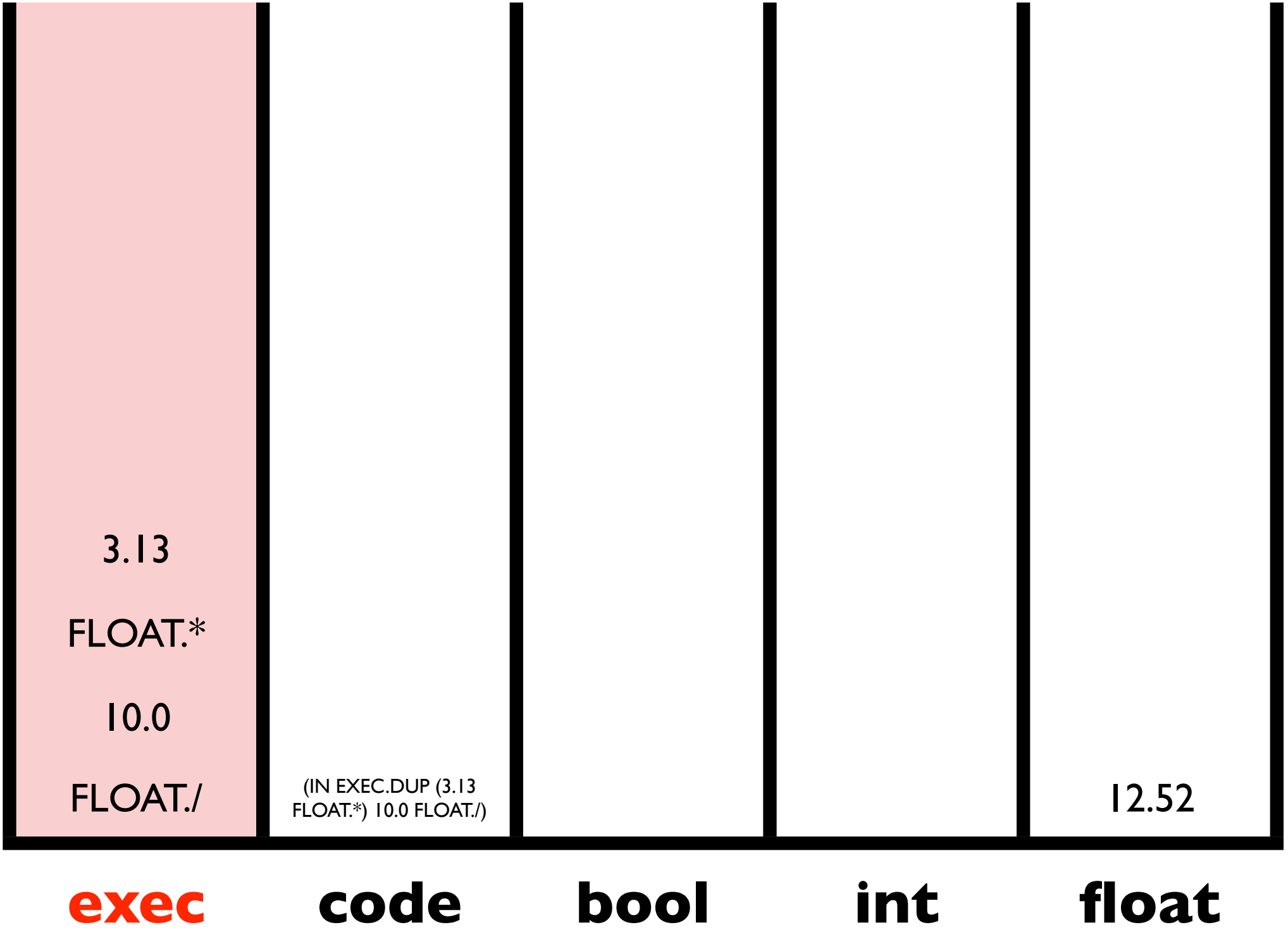

EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

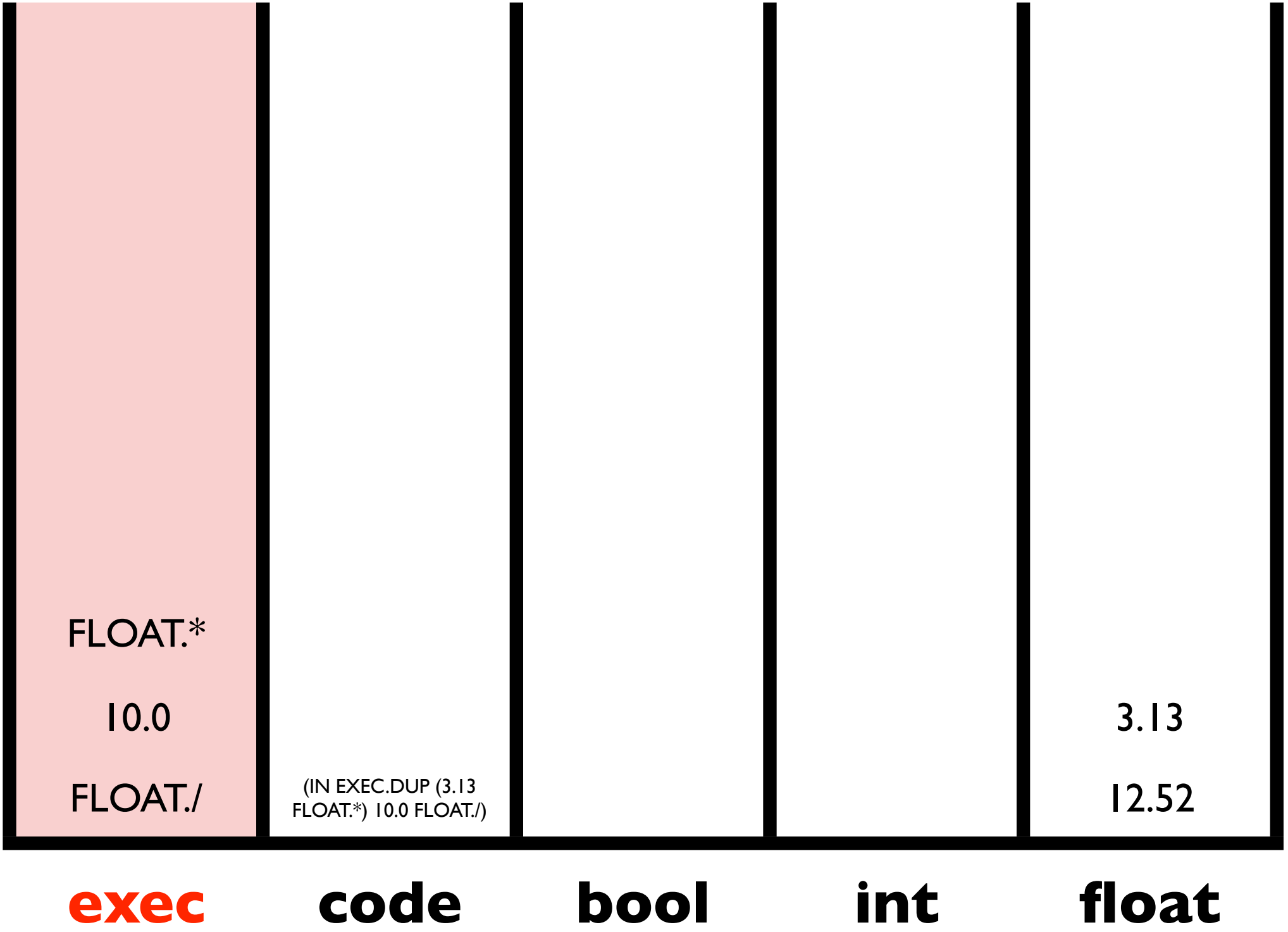

(3.13 FLOAT.*)
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

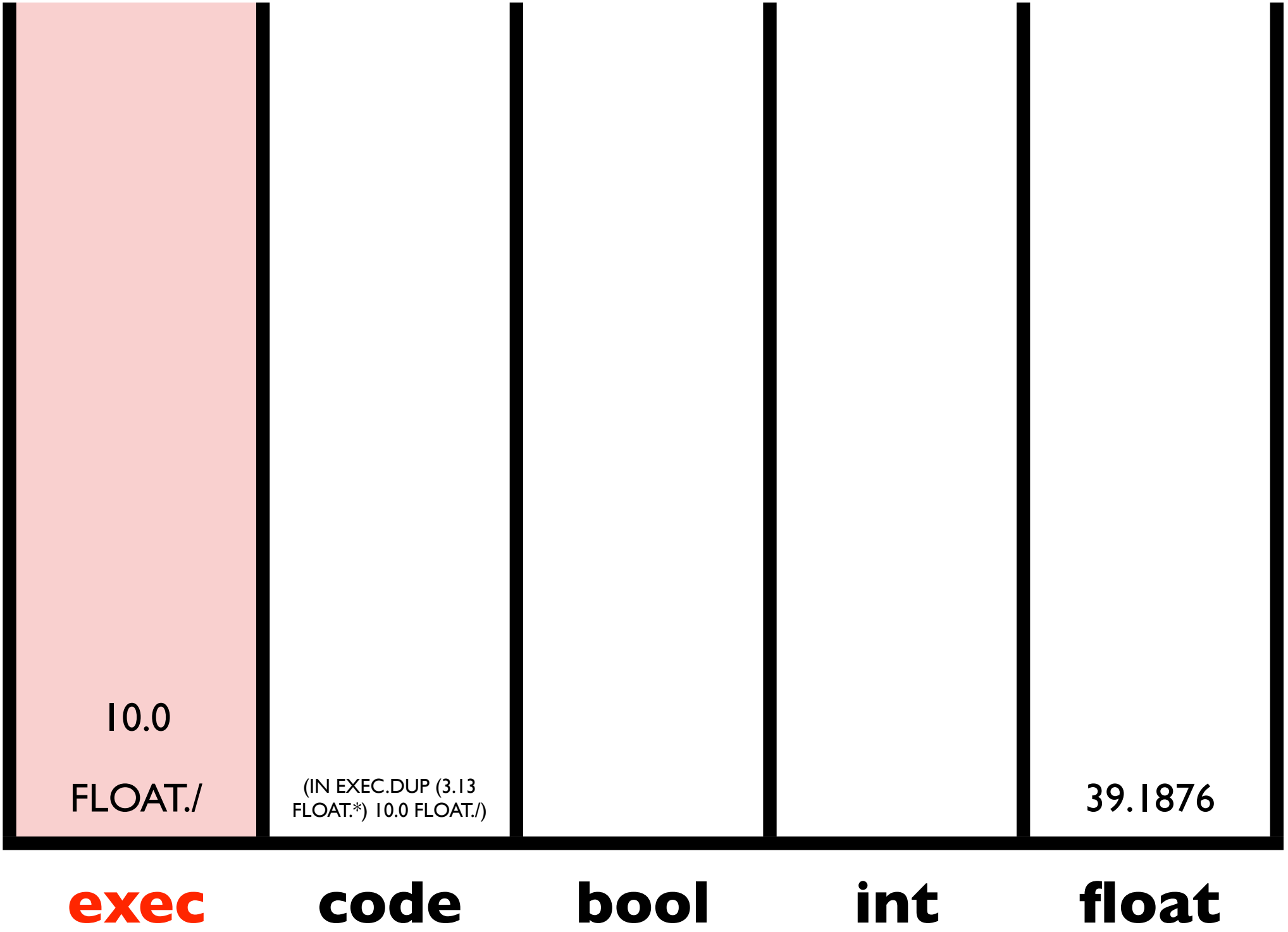

3.13
FLOAT.*
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

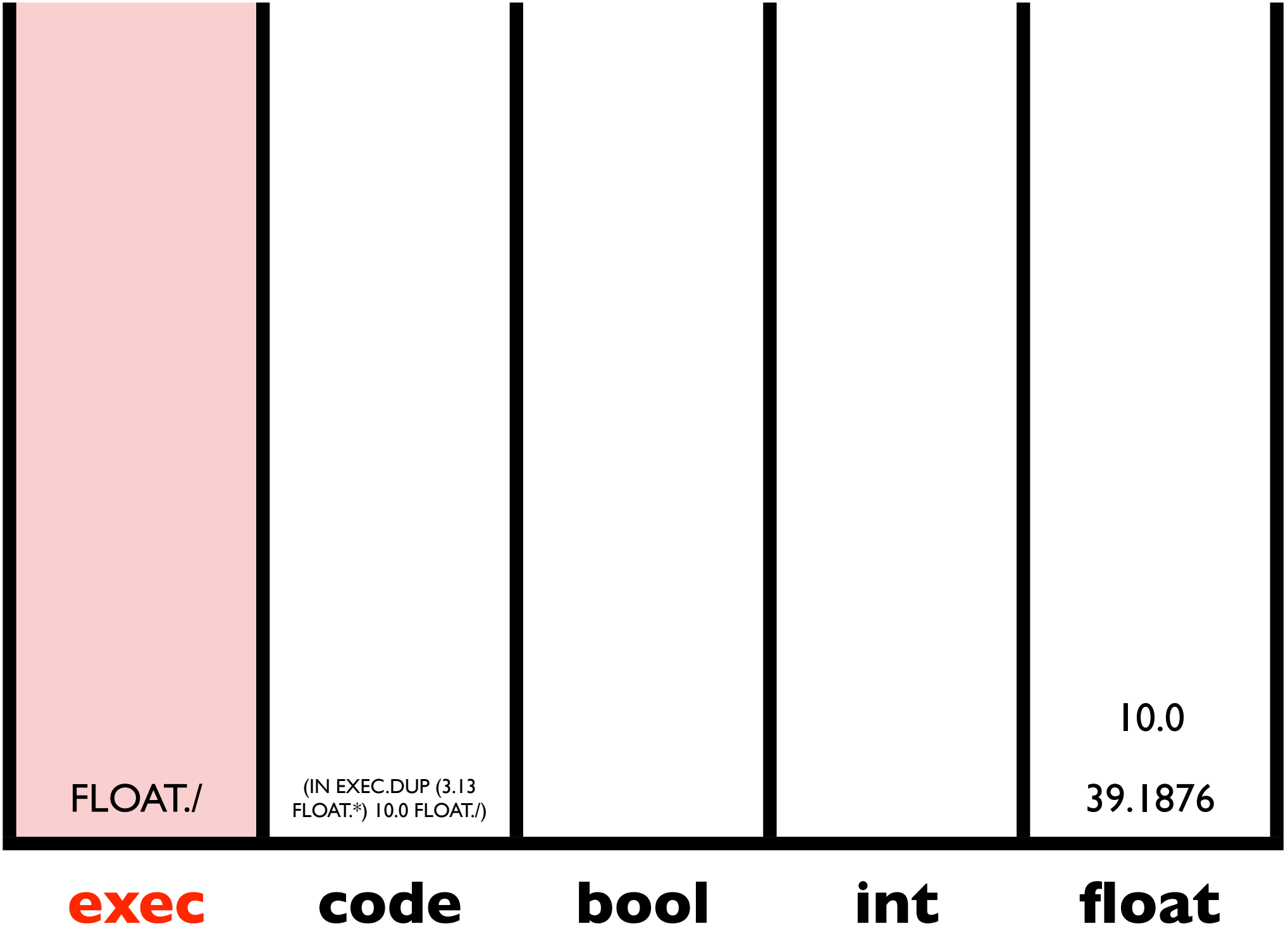

FLOAT.*
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
3.13
4.0
exec
code
bool
int
float

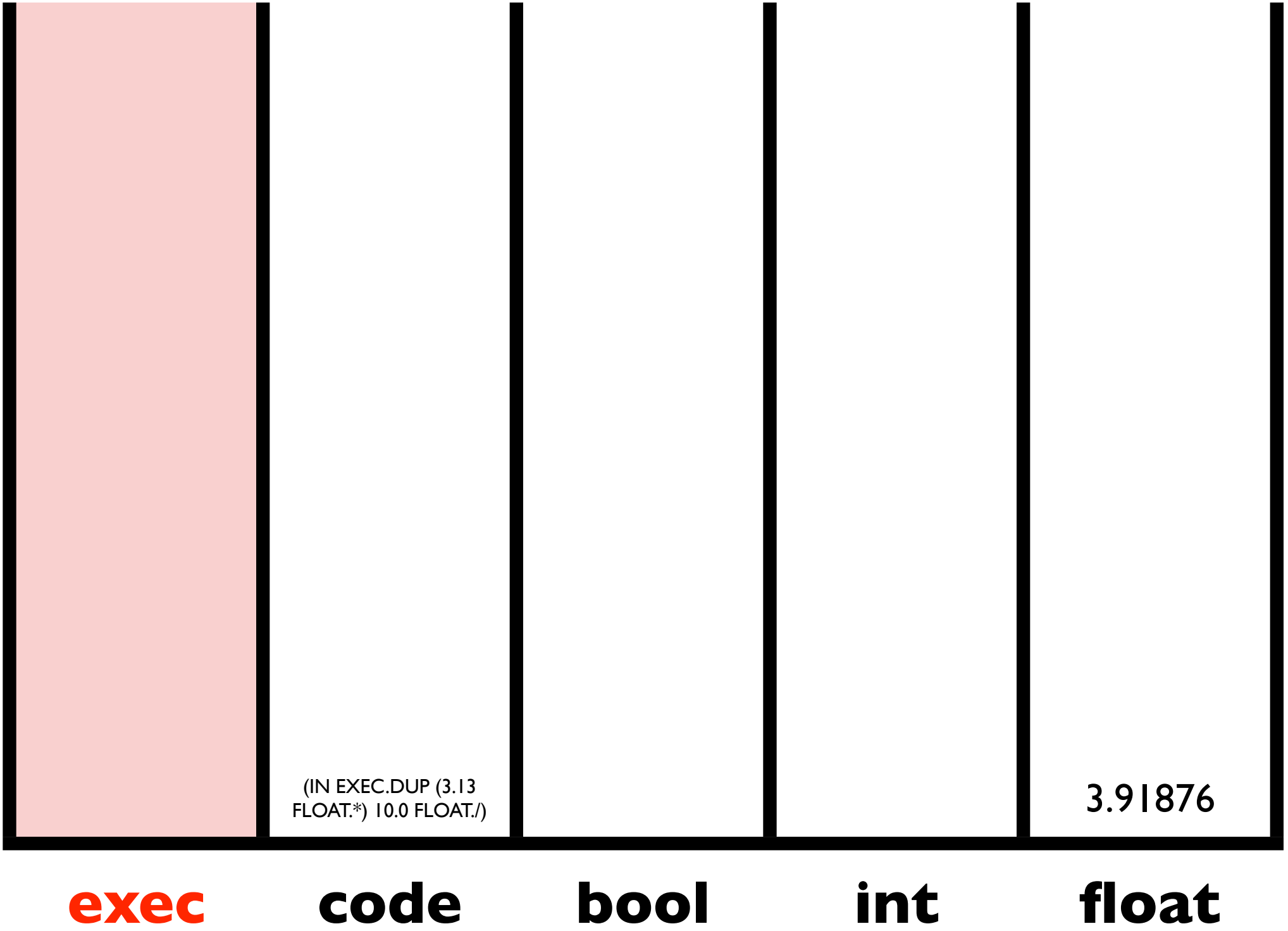

(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
12.52
exec
code
bool
int
float


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

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 ) )
```

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`)

Named Subroutines

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

Auto-simplification

Loop:

Make it randomly simpler

If it's as good or better: keep it

Otherwise: revert

Problems Solved by PushGP in the GECCO-2005 Paper on Push3

- Reversing a list
- Factorial (many algorithms)
- Fibonacci (many algorithms)
- Parity (any size input)
- Exponentiation
- Sorting

Modularity
Ackley and Van Belle

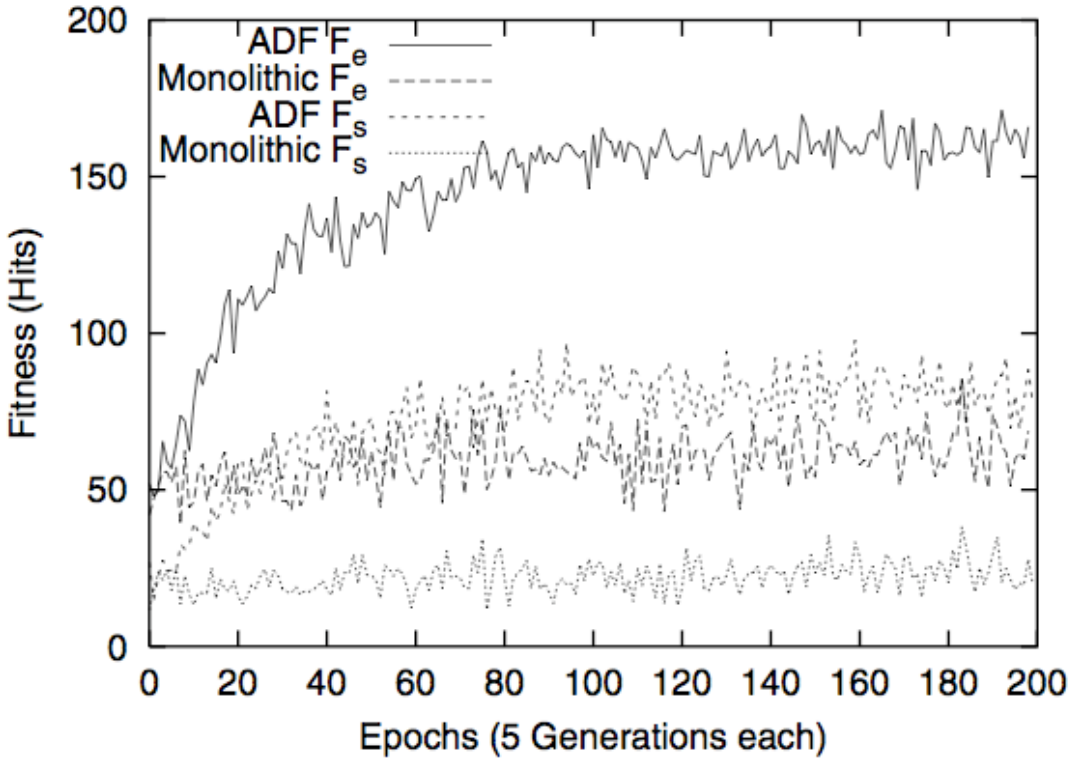
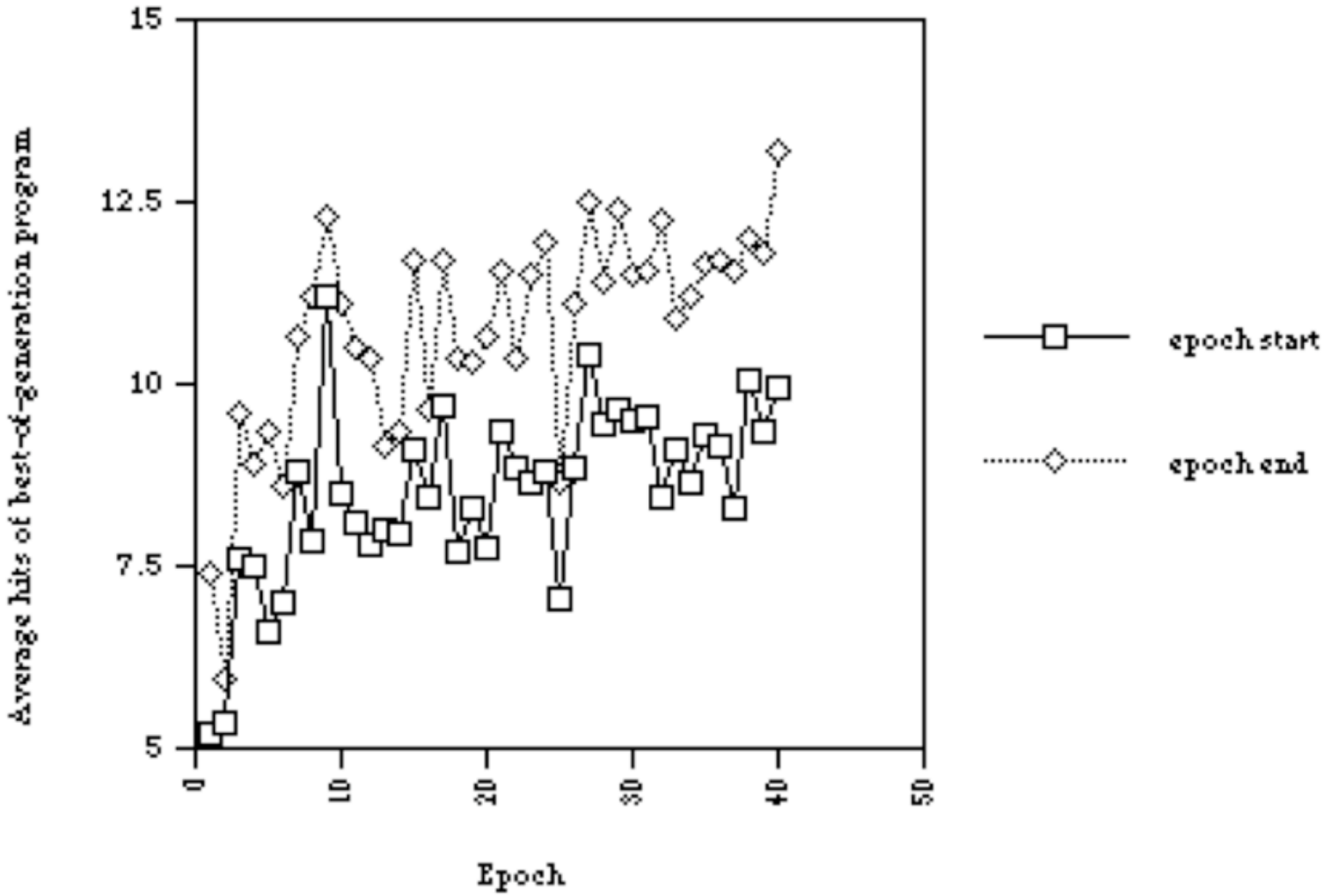
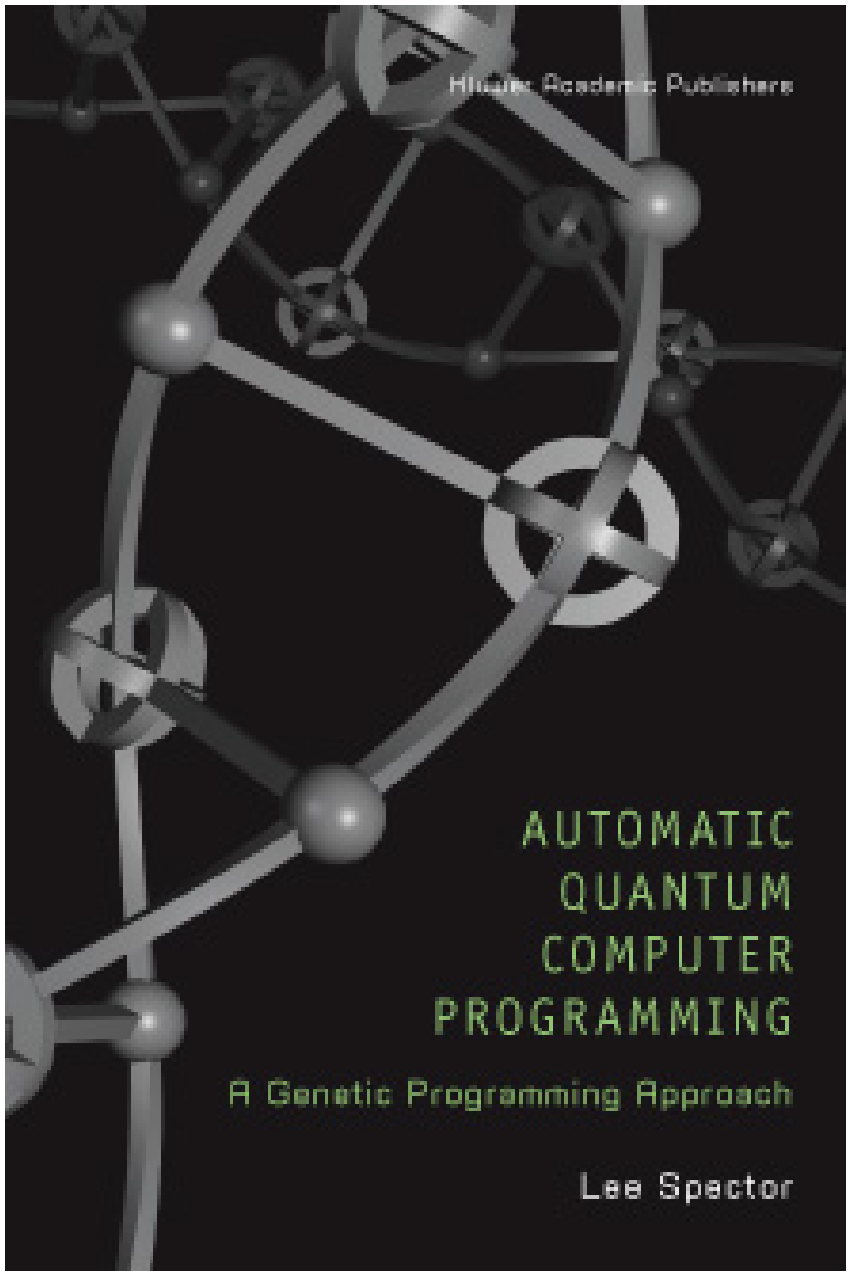


Figure 2: Average fitness values at the start (F_s) and end (F_e) of each epoch when regressing to $y = A\sin(Ax)$. A is selected at the start of each epoch uniformly from the range $[0,6)$.

Modularity via Push





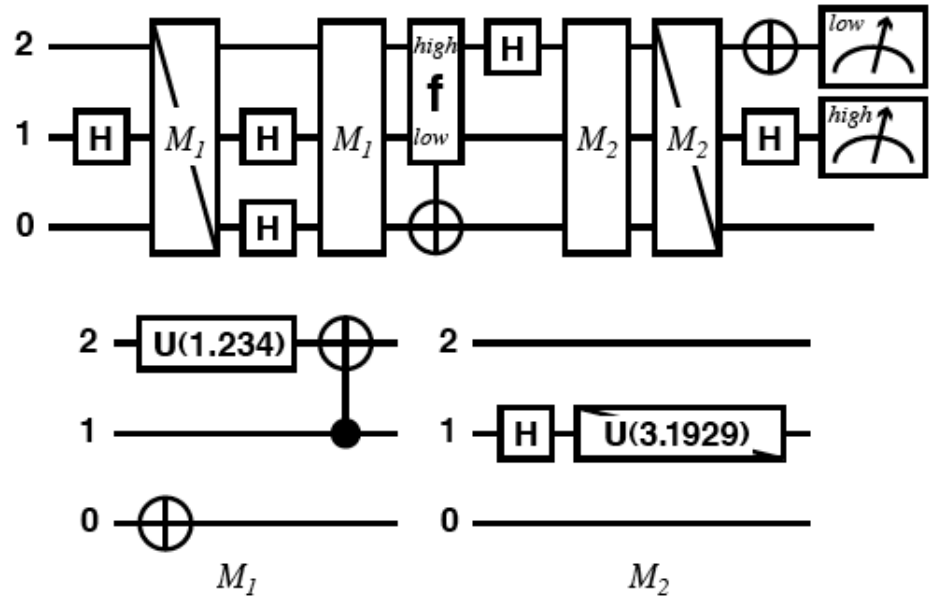


Figure 8.7. A gate array diagram for an evolved version of Grover's database search algorithm for a 4-item database. The full gate array is shown at the top, with M_1 and M_2 standing for the smaller gate arrays shown at the bottom. A diagonal line through a gate symbol indicates that the matrix for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

Humies 2004
GOLD MEDAL

Autoconstructive Evolution

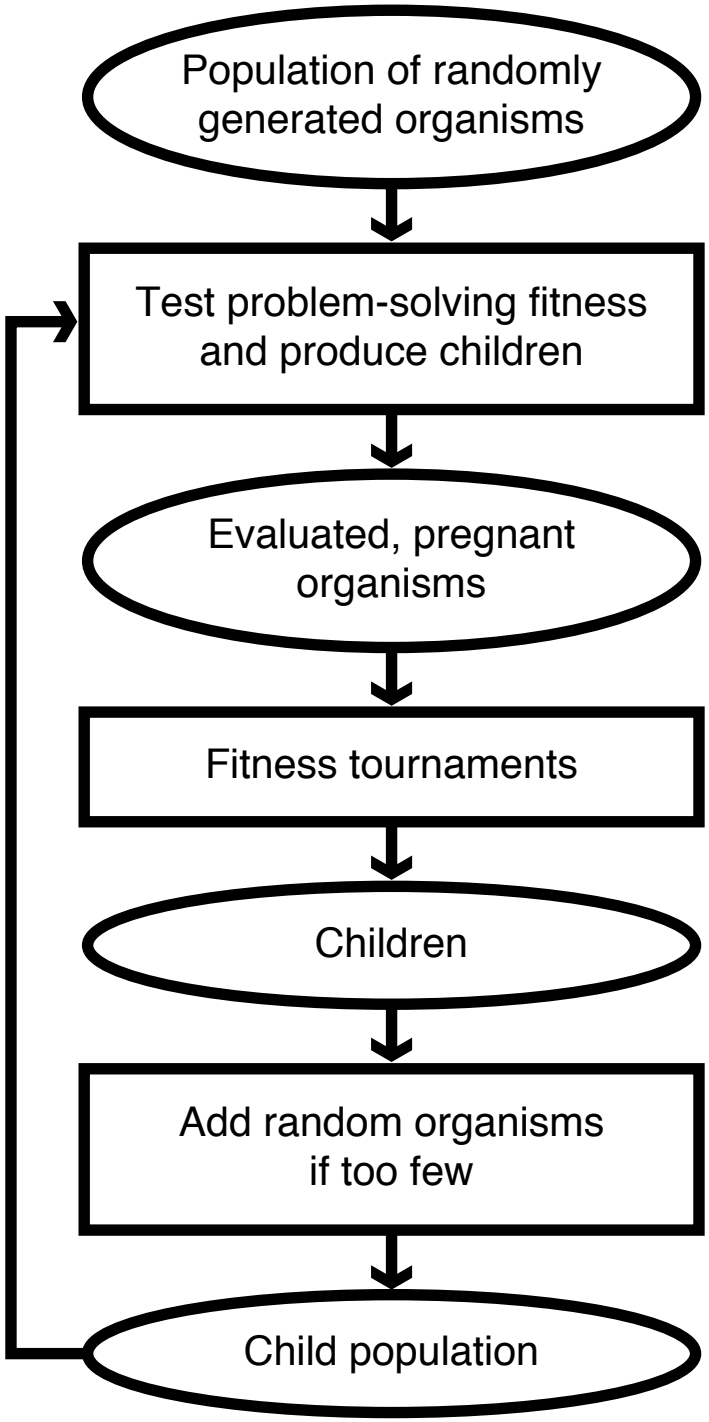
- Individuals make their own children.
- Agents thereby control their own mutation rates, sexuality, and reproductive timing.
- The machinery of reproduction and diversification (i.e., the machinery of evolution) evolves.
- Radical self-adaptation.

Related Work

- MetaGP: but (1) programs and reproductive strategies dissociated and (2) generally restricted reproductive strategies.
- ALife systems such as Tierra, Avida, SeMar: but (1) hand-crafted ancestors, (2) reliance on cosmic ray mutation, and (3) weak problem solving.
- Evolved self-reproduction: but generally exact reproduction, non-improving (exception: Koza, but very limited tools for problem solving *and* for construction of offspring).

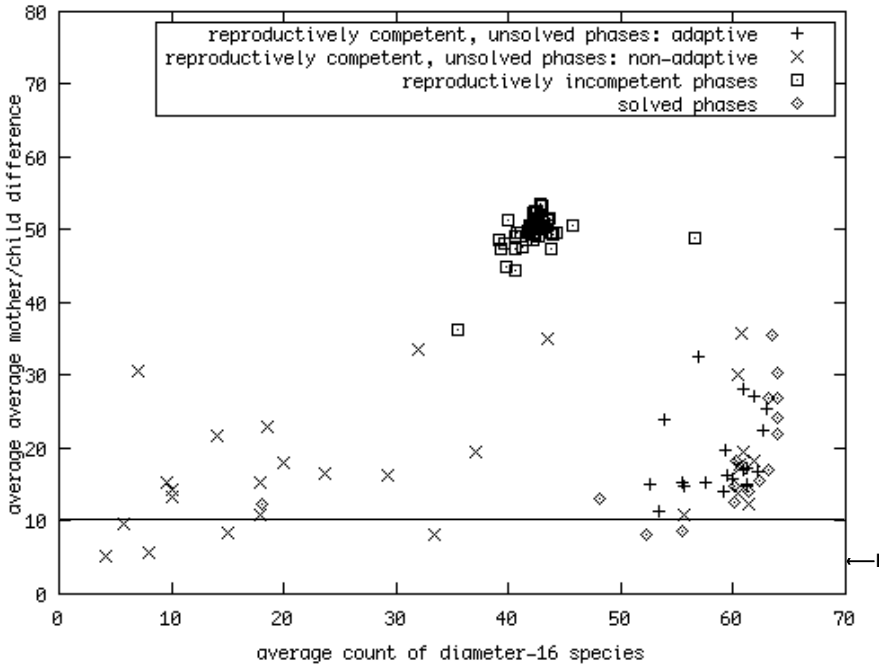
Pushpop

- A soup of evolving Push programs.
- Reproductive procedures emerge ex nihilo:
  - No hand-designed "ancestor."
  - Children constructed by any computable process.
  - No externally applied mutation procedure or rate.
  - Exact clones are prohibited, but near-clones are permitted.
- Selection for problem-solving performance.

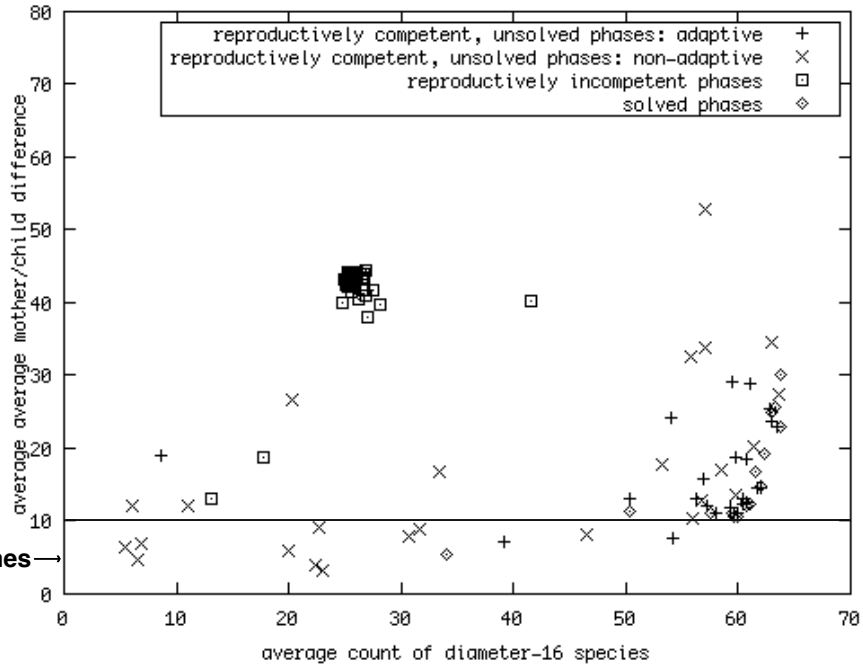

Population of randomly generated organisms
Test problem-solving fitness and produce children
Evaluated, pregnant organisms
Fitness tournaments
Children
Add random organisms if too few
Child population

Species vs. Mother/Child Differences

Note distribution of "+" points: adaptive populations have many species and mother/daughter differences in a relatively high, narrow range (above near-clone levels).



←near-clones→



Runs including sexual instructions

Runs without sexual instructions

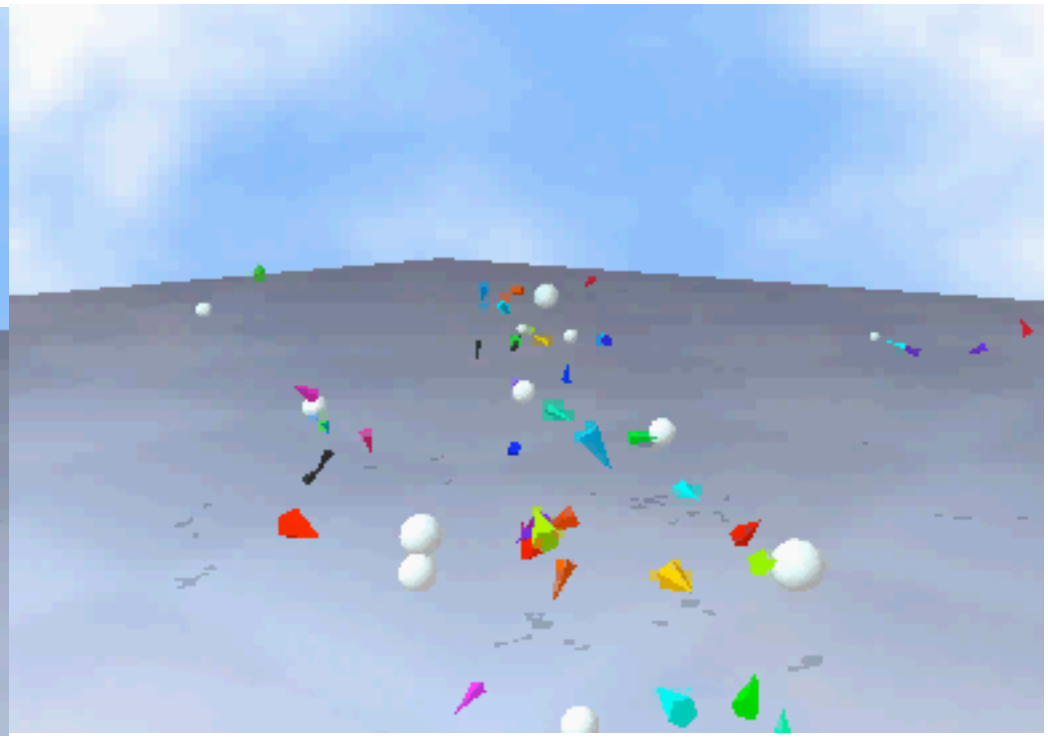
Pushpop Results

- In adaptive populations:
  - Species are more numerous.
  - Diversification processes are more reliable.
- Selection can promote diversity.
- Provides a possible explanation for the evolution of diversifying reproductive systems.
- Weak problem-solving power.
- Difficult to analyze results.

SwarmEvolve 2.0

- Behavior (including reproduction) controlled by evolved Push programs.
- Color, color-based agent discrimination controlled by agents.
- Energy conservation.
- Facilities for communication, energy sharing.
- Ample user feedback (e.g. diversity metrics, agent energy determines size).

SwarmEvolve 2.0



AutoPush

- Goals:
  - Superior problem-solving performance.
  - Tractable analysis.
- Push3.
- Clojure (incidental, but fun!)

- Asexual (for now).
- Children produced on demand (not during fitness testing).
- Constraints on selection and birth.

Definitions

- **Improvement**: Recency-weighted average of vector of improvements (1), declines (-1), and repeats (0).
- **Discrepancy**: Sum, over all unique expressions in two programs, of the difference between the numbers of occurrences of the expression in the two programs.

Constraints on Selection

- Prefer reproductively competent parents.
- Prefer parents with non-stagnant lineages (changed performance in the most recent half of the lineage, after some threshold lineage length).
- Prefer parents with good problem-solving performance.
- (Possibly) Prefer parents from lineages with better-improving problem-solving performance

Constraints on Birth

- Prevent birth from lineages with insufficient improvement.
- Prevent birth from lineages with constant discrepancies.
- Prevent birth from parents with fitness penalties, e.g. for non-termination.
- Prevent birth of children of illegal sizes.
- Prevent birth of children identical to ancestors or potential siblings.

Preliminary Results

- Simple symbolic regression successes
  - $y=x^3-2x^2-x$
  - $y=x^6-2x^4+x^3-2$
- Prime-generating polynomials
- Instructive lineage traces

Ancestor of Success

(for $y=x^3-2x^2-x$)

```
((code_if (code_noop) boolean_fromfloat (2)
integer_fromfloat) (code_rand integer_rot)
exec_swap code_append integer_mult)
```

Produces children of the form:

```
(RANDOM-INSTRUCTION (code_if (code_noop)
boolean_fromfloat (2) integer_fromfloat)
(code_rand integer_rot) exec_swap
code_append integer_mult)
```

Six Generations Later

A descendent of the form:

```
(SUB-EXPRESSION-1 SUB-EXPRESSION-2)
```

Produces children of the form:

```
((RANDOM-INSTRUCTION-1 (SUB-EXPRESSION-1))
(RANDOM-INSTRUCTION-2 (SUB-EXPRESSION-2)))
```

One Generation Later

A solution, which incidentally inherits the same reproductive strategy:

```
((integer_stackdepth (boolean_and
code_map)) (integer_sub (integer_stackdepth
(integer_sub (in (code_wrap (code_if
(code_noop) boolean_fromfloat (2)
integer_fromfloat) (code_rand integer_rot)
exec_swap code_append integer_mult))))))
```

Conclusions

- Autoconstructive evolution can solve problems.
- It can be refined for broader applicability and more tractable analysis.
- **Bold (unsupported!) prediction:** *The most powerful, practical genetic programming systems of the future will be autoconstructive.*