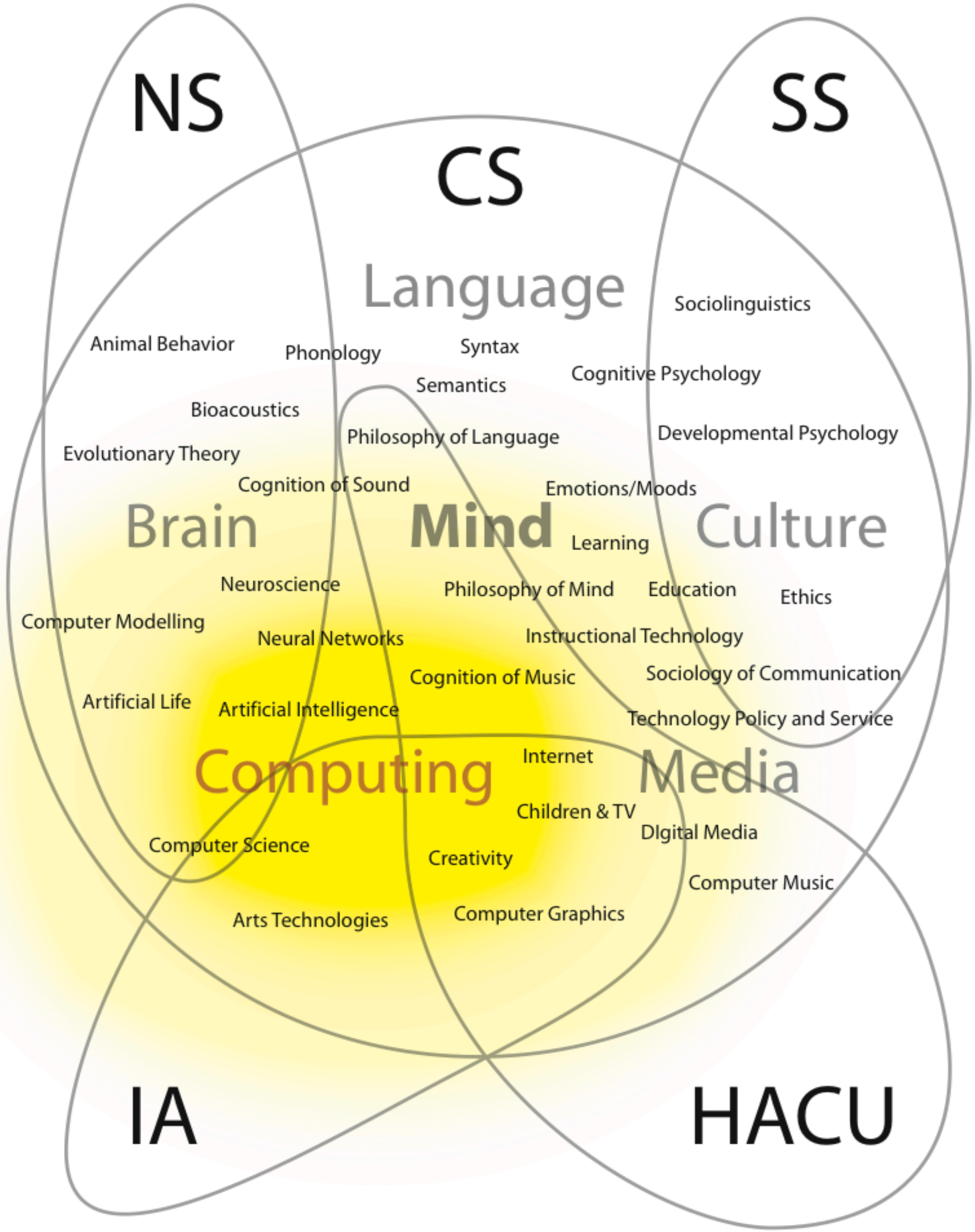
# Evolving the Future of Mathematics

Lee Spector
Cognitive Science
Hampshire College
hampshire.edu/lspector

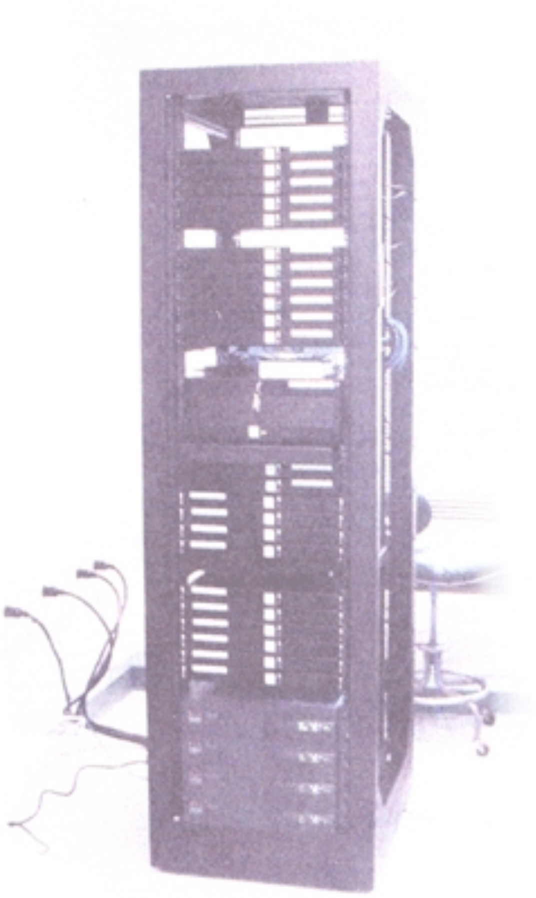
# Outline

- Genetic programming
- Human competitive genetic programming
- An application to finite algebras
- How to think about possible future applications
- Autoconstructive evolution

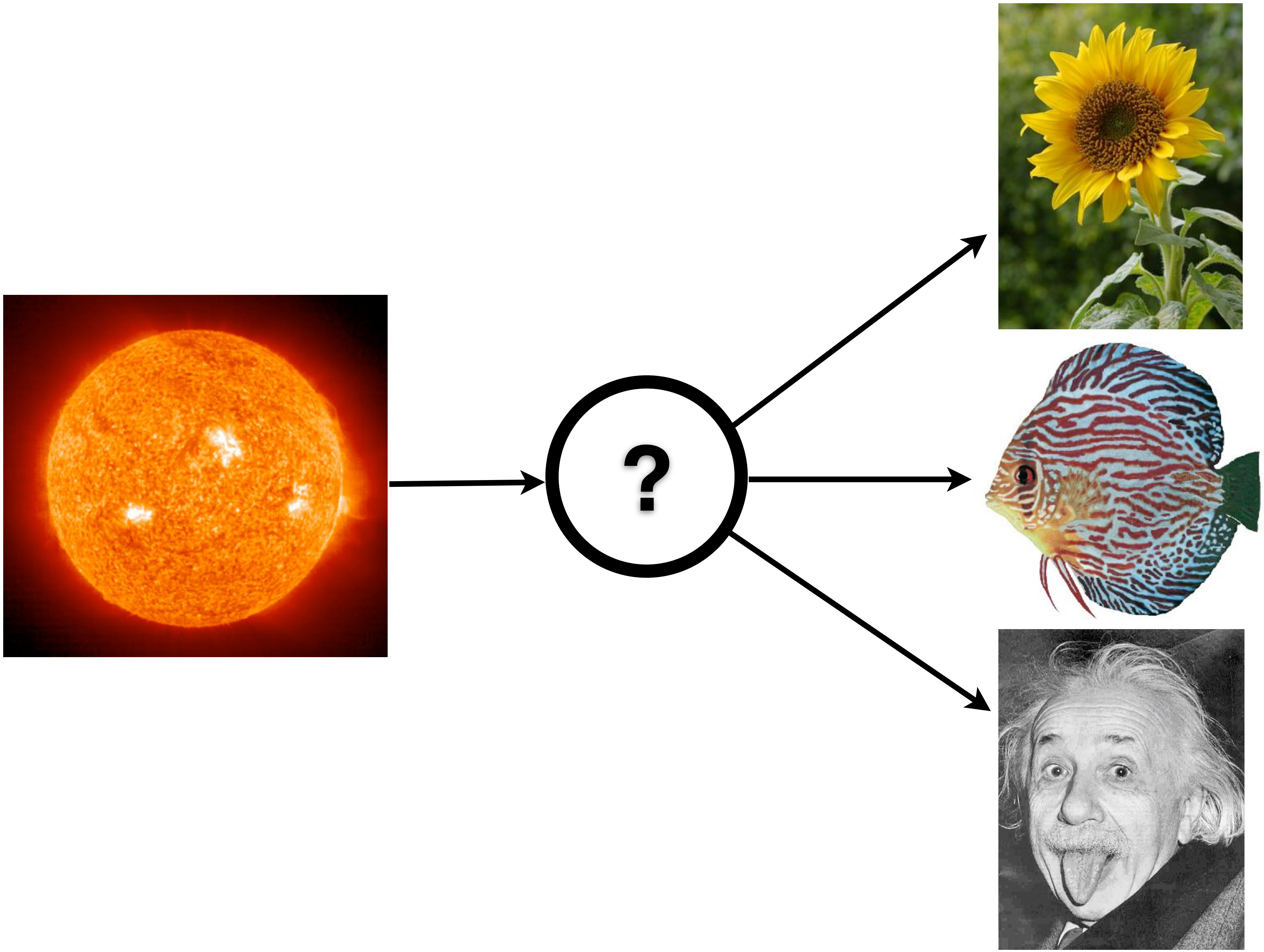


NS
SS
CS
Language
Sociolinguistics
Animal Behavior
Phonology
Syntax
Cognitive Psychology
Semantics
Bioacoustics
Philosophy of Language
Developmental Psychology
Evolutionary Theory
Cognition of Sound
Emotions/Moods
Brain
Mind
Learning
Culture
Neuroscience
Philosophy of Mind
Education
Ethics
Computer Modelling
Neural Networks
Instructional Technology
Cognition of Music
Sociology of Communication
Artificial Life
Artificial Intelligence
Technology Policy and Service
Computing
Internet
Media
Children & TV
DIgital Media
Computer Science
Creativity
Computer Music
Arts Technologies
Computer Graphics
IA
HACU

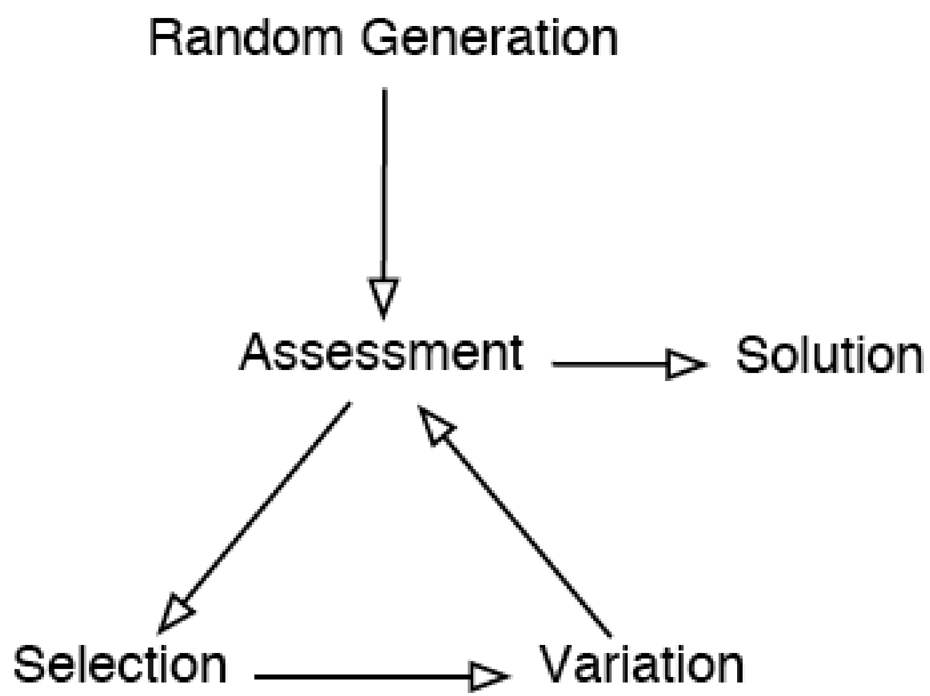
# Cluster Computing Facility

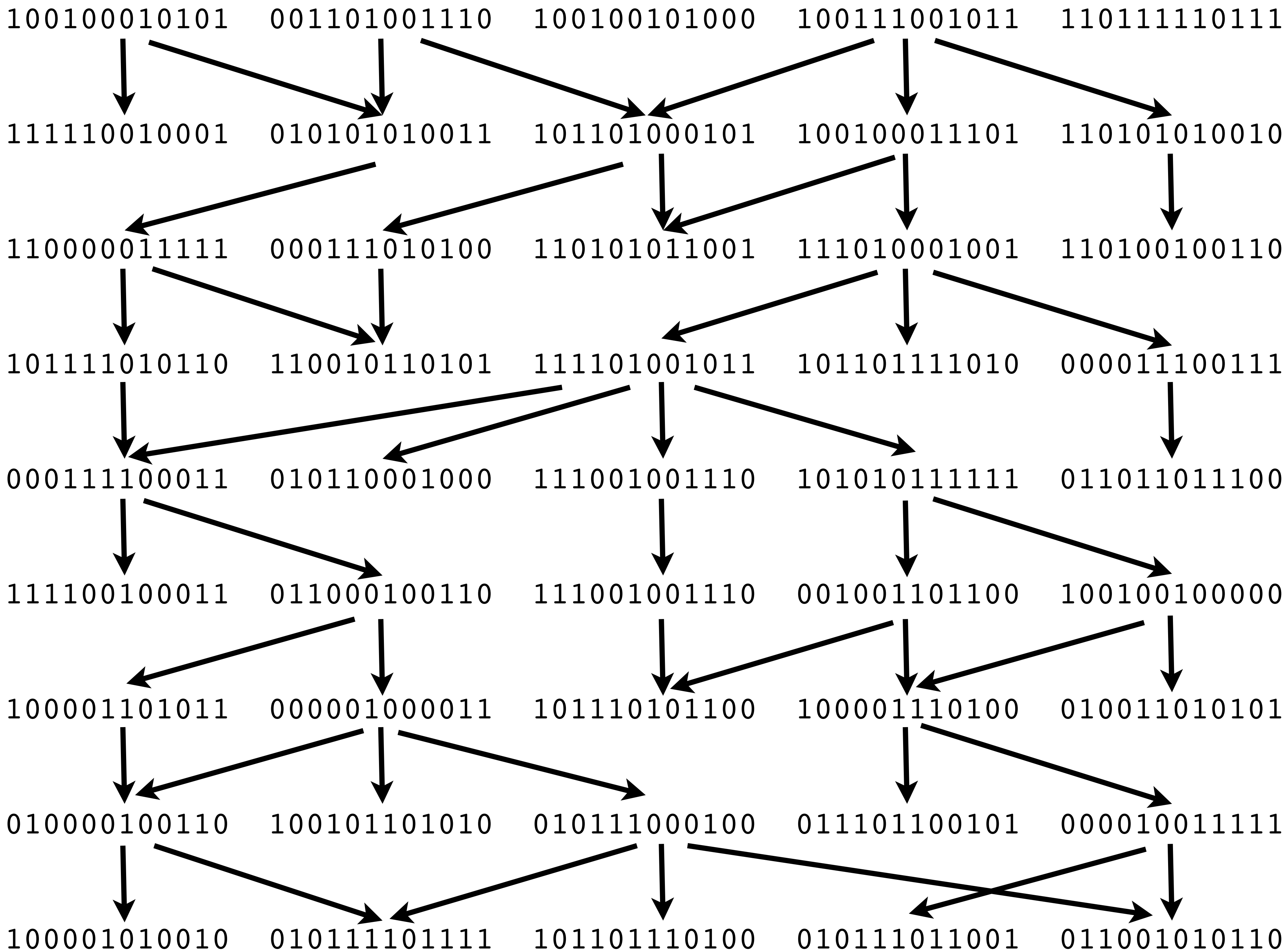


- A mixed-architecture 100+ core high-performance computer cluster
- Available to both faculty and students for course and project work
- Hosts a wide range of research and application software
- Runs Linux and the open source ROCKS clustering software package
- Uses include research and development projects in distributed computing, physical and ecological simulation, quantum computing, evolutionary algorithms, and applications of high-performance computing to the arts



# Evolutionary Algorithms




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

# 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.

# Genetic Programming

- Evolutionary algorithm in which the candidate solutions are executable computer programs.
- Candidate solutions are assessed, at least in part, 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., ...).
- 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*.

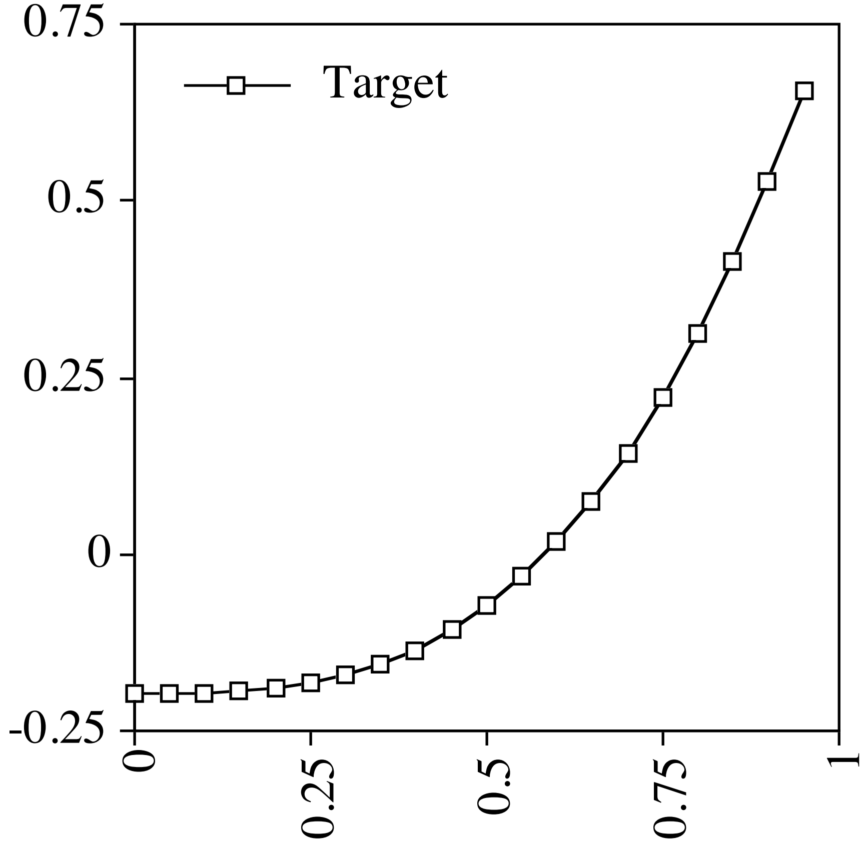
Primordial ooze: +, -, *, %, *x*, 0.1

Fitness = error (smaller is better)

# GP Parameters

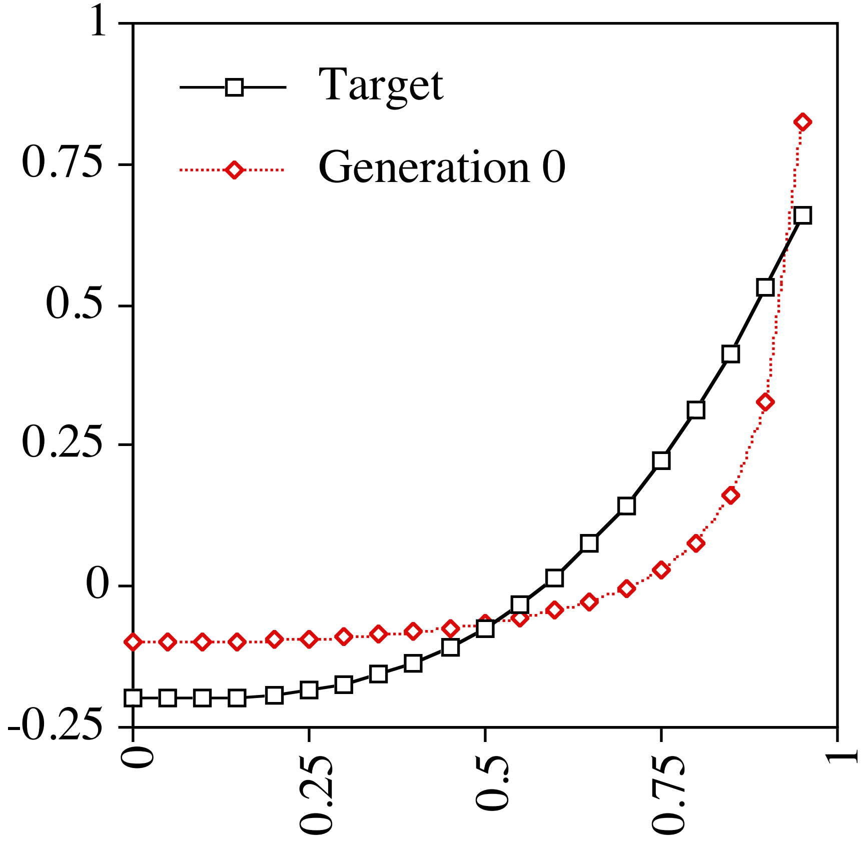
Maximum number of Generations: 51
Size of Population: 1000
Maximum depth of new individuals: 6
Maximum depth of new subtrees for mutants: 4
Maximum depth of individuals after crossover: 17
Fitness-proportionate reproduction fraction: 0.1
Crossover at any point fraction: 0.3
Crossover at function points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

# Evolving $y = x^3 - 0.2$



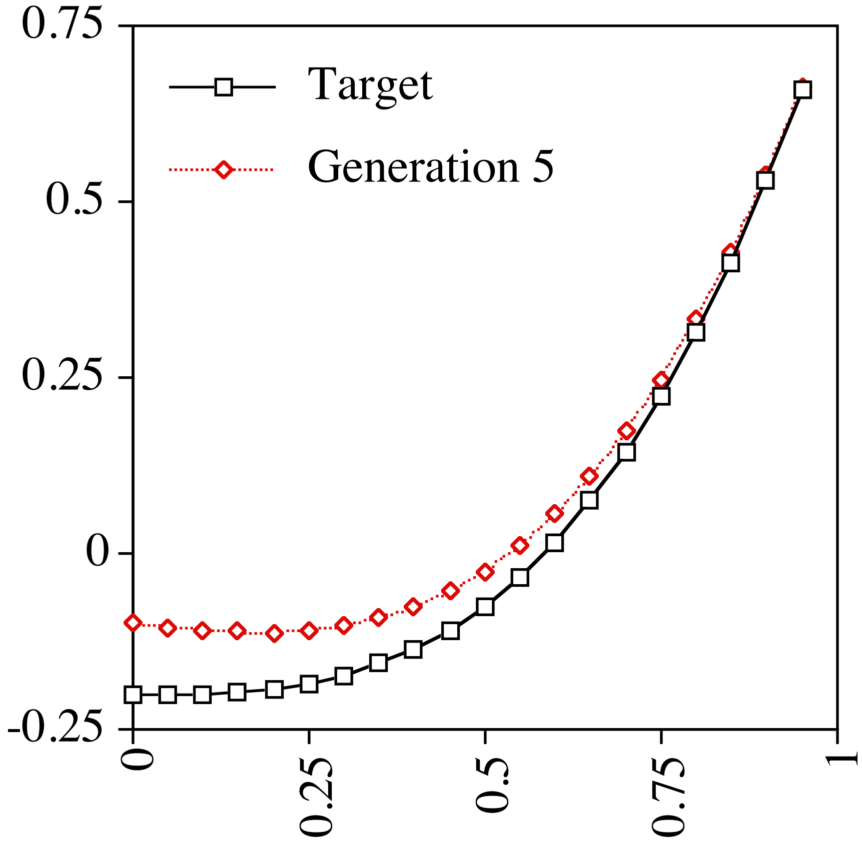
# Best Program, Gen 0

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



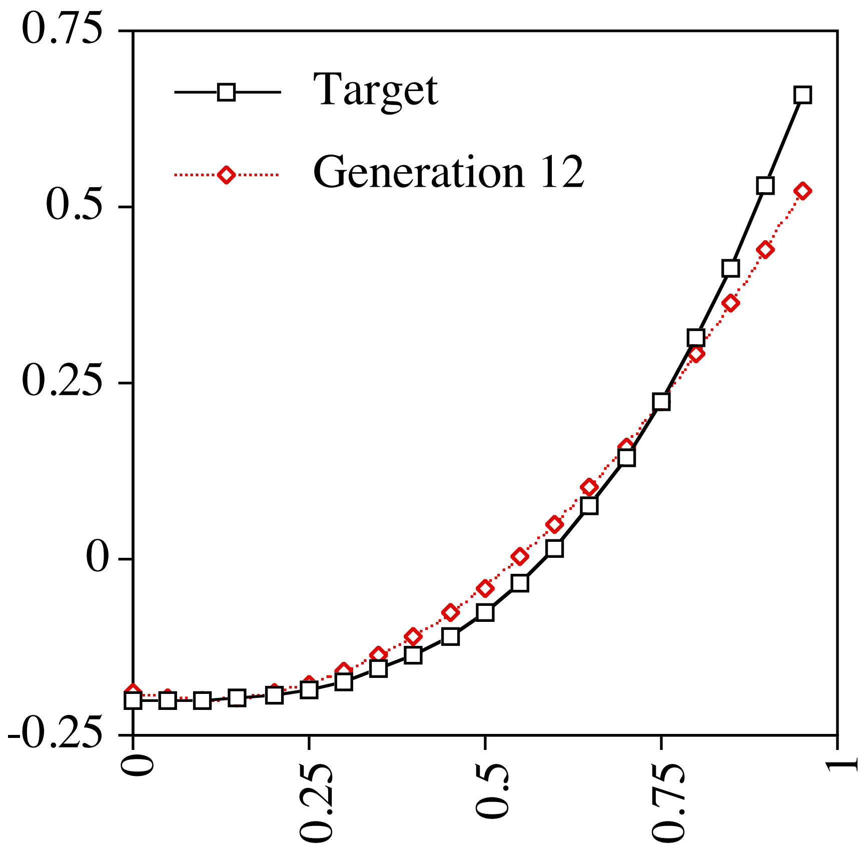
# Best Program, Gen 5

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



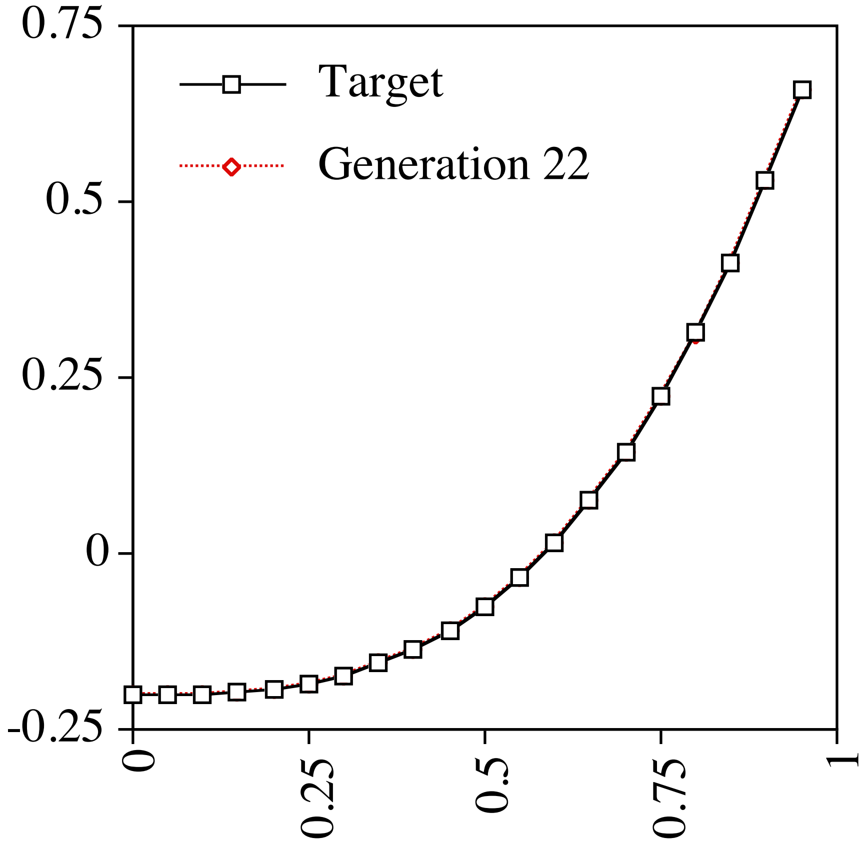
# Best Program, Gen 12

```
(+ (- (- 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)
```



AIEDAM
Artificial Intelligence for Engineering Design, Analysis and Manufacturing

VOLUME 22 SUMMER 2008 NUMBER 3


## SPECIAL ISSUE

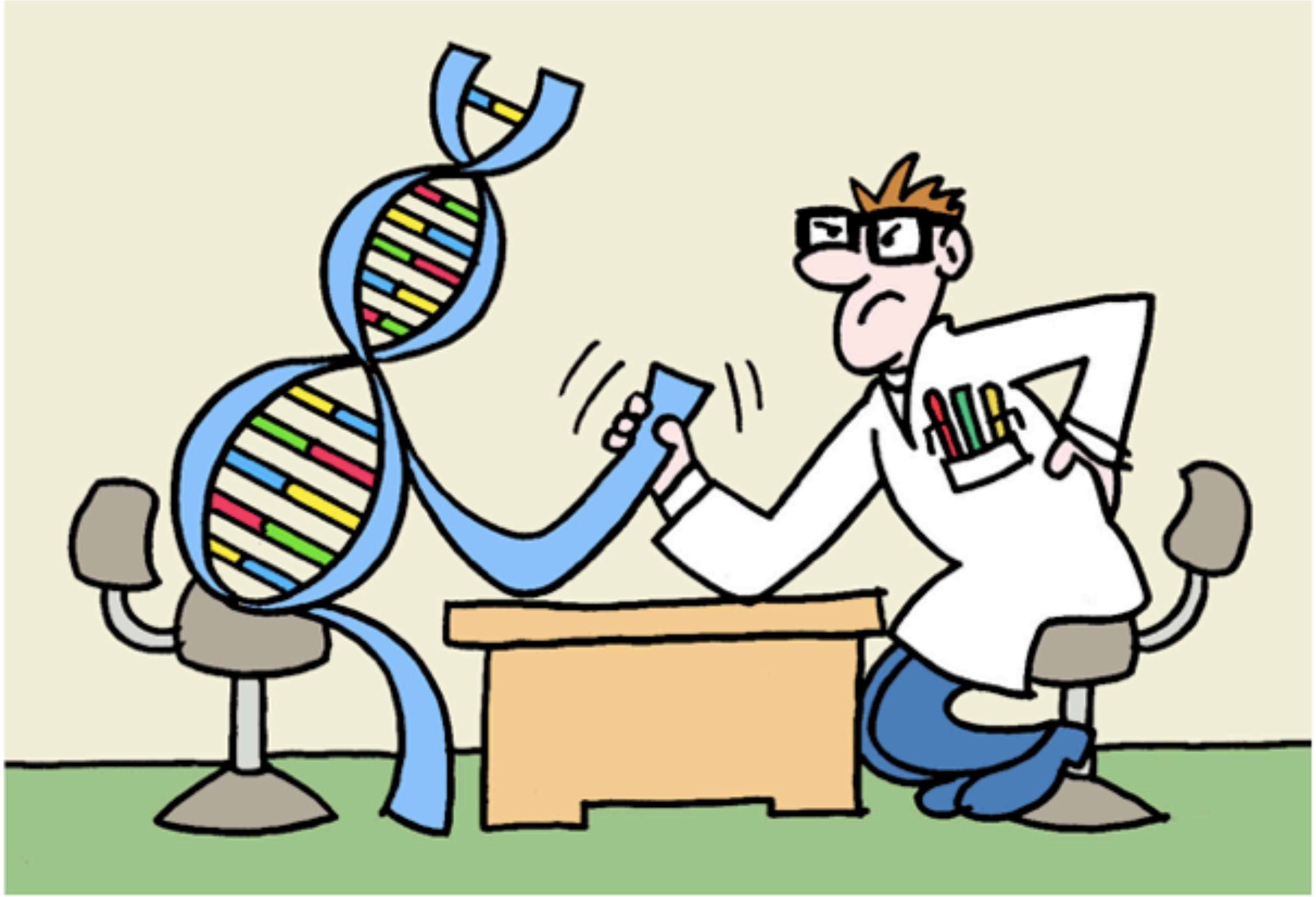
# *Genetic Programming for Human-Competitive Designs*

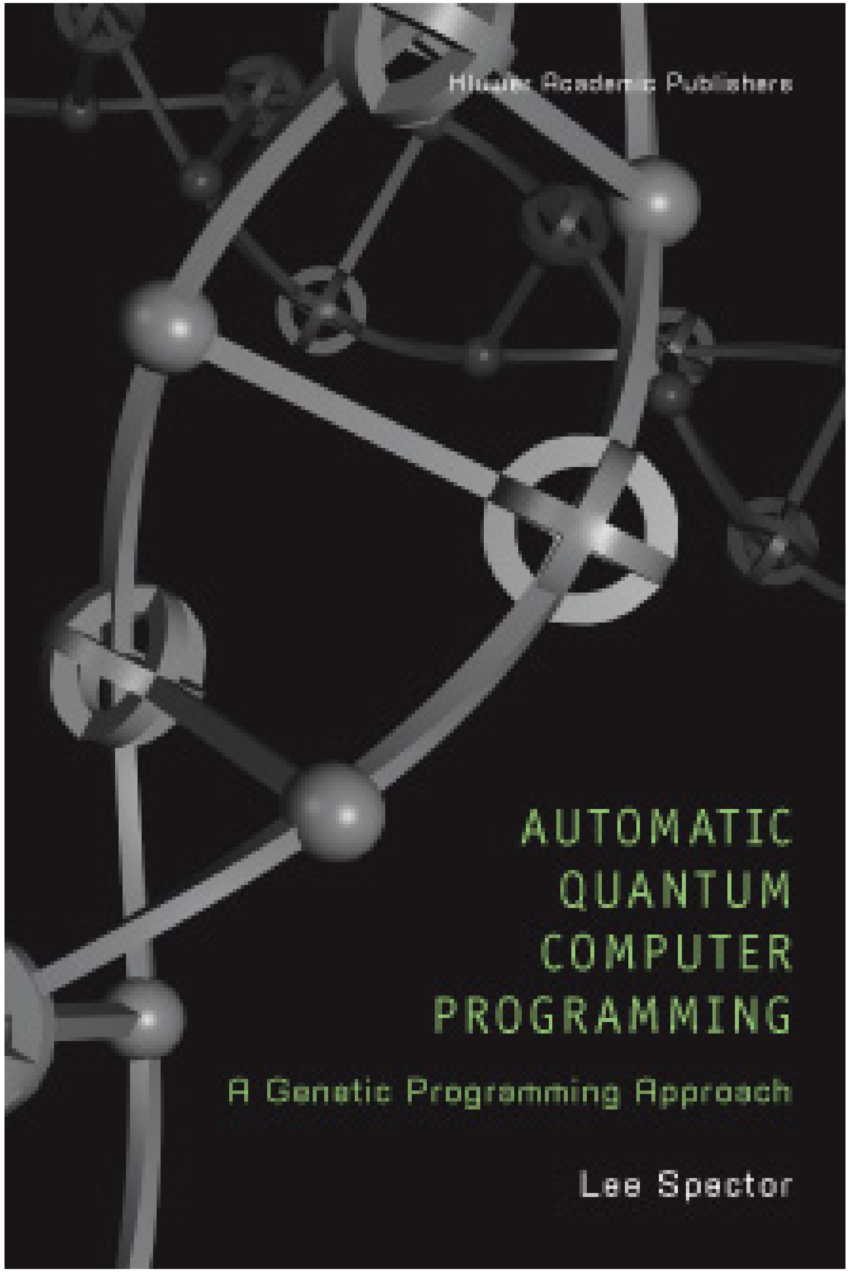
## Guest Editor

LEE SPECTOR

http://www.human-competitive.org/

# THE 5th ANNUAL (2008) "HUMIES" AWARDS FOR HUMAN-COMPETITIVE RESULTS PRODUCED BY GENETIC AND EVOLUTIONARY COMPUTATION HELD AT THE GENETIC AND EVOLUTIONARY COMPUTATION CONFERENCE





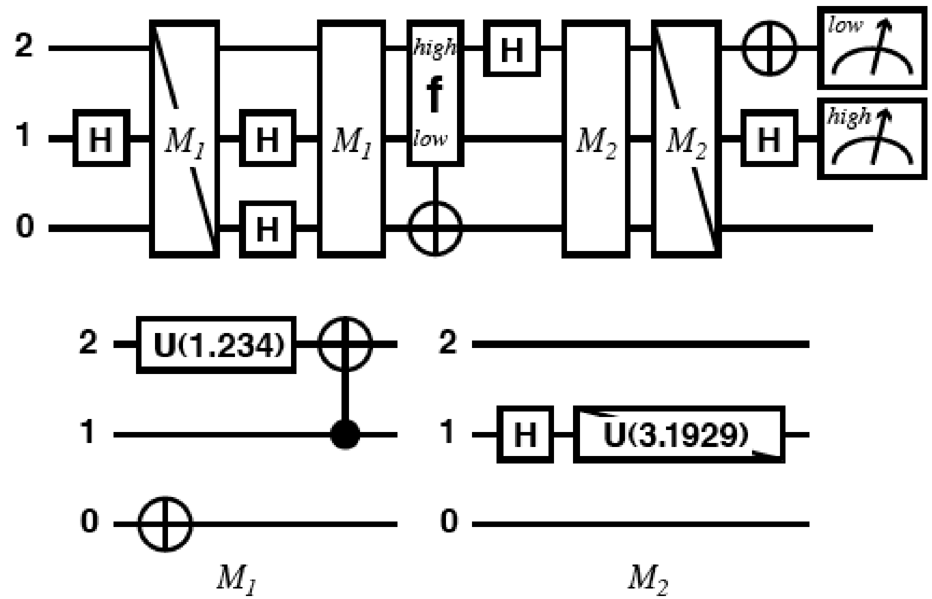



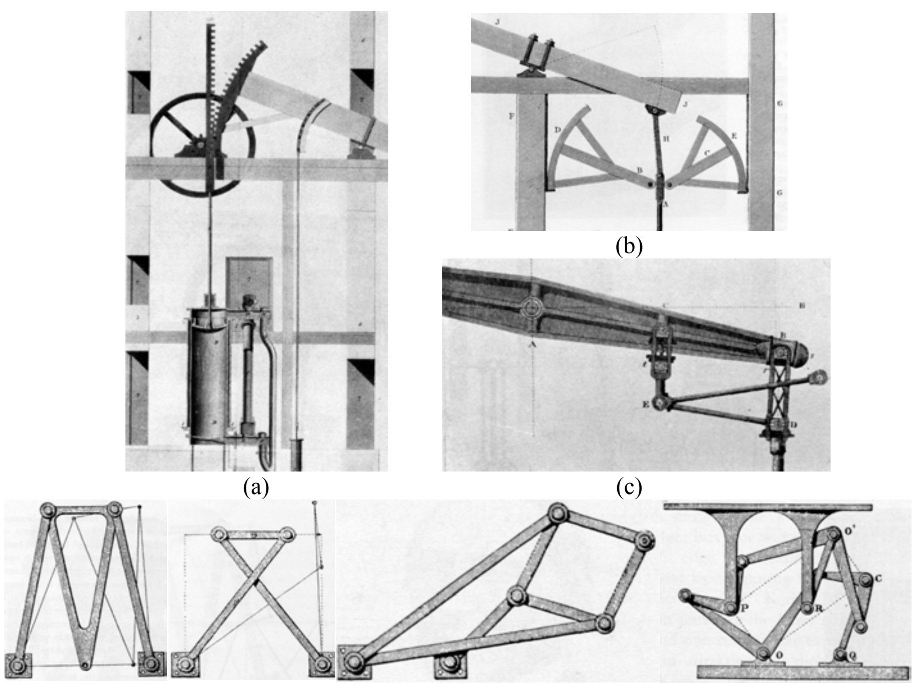

*Figure 8.7.* A gate array diagram for an evolved version of Grover's database search algorithm for a 4-item database. The full gate array is shown at the top, with $M_1$ and $M_2$ standing for the smaller gate arrays shown at the bottom. A diagonal line through a gate symbol indicates that the matrix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

# Humies 2004 GOLD MEDAL

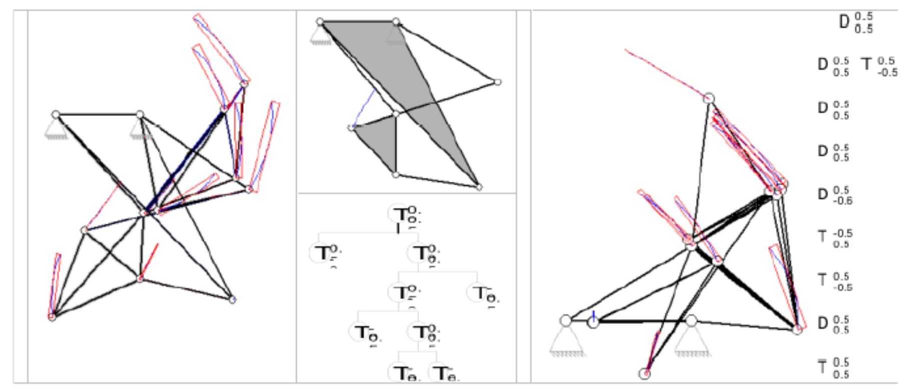
# Line-Drawing Mechanism

Without reference to an existing straight line.

Human-competitive results; challenged world's greatest inventors for a century (spanning 18th and 19th).



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



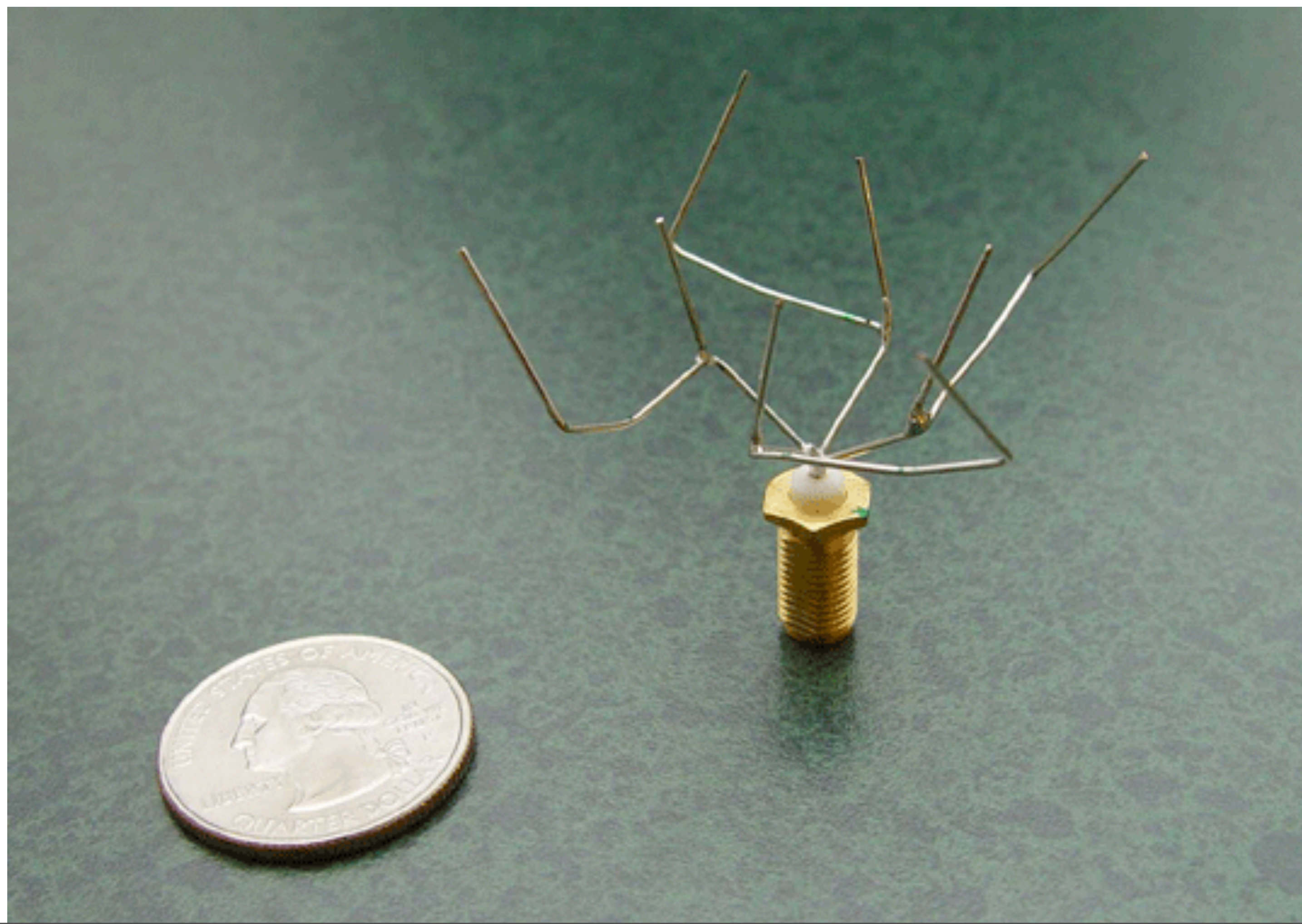
(a) (b)

**Fig. 10. Two Evolved mechanisms** and their tree representations (a) Linearity 1:12819; The simplified equivalent shown top right, and (b) Linearity 1:4979.

Lipson, H. 2004. How to Draw a Straight Line Using a GP: Benchmarking Evolutionary Design Against 19th Century Kinematic Synthesis. GECCO-2004.

# Evolved Antenna

- Human-competitive result.
- For NASA Space Technology 5 Mission.
- Lohn, Hornby, and Linden.



# Everybody's Favorite Finite Algebra

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

| $\wedge$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 0 |
| 1 | 0 | 1 |

| $\vee$ | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 1 |

| | $\neg$ |
|---|---|
| 0 | 1 |
| 1 | 0 |

*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
- Under active investigation for decades, with major advances (cited fully in the paper) in 1939, 1954, 1970, 1975, 1979, 1991, 2008

# 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

# Specific Algebras

| $\mathbf{A}_1$ $*$ | 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 |

# Methods

- Traditional genetic programming with ECJ
- Stack-based genetic programming with PushGP
- Alternative random code generators
- Asynchronous islands
- Trivial geography
- Parsimony-based selection
- Alpha-inverted selection pressure
- HAH = Historically Assessed Hardness

# Results

- Discriminators for $A_1, A_2, A_3, A_4, A_5$
- Mal'cev and majority terms for $B_1$
- Parameter tables and result terms in paper
- Example discriminator term for $A_1$:

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

# Assessing Significance

Relative to prior methods:

- Uninformed search:
  - Exhaustive: analytical (expected value) and empirical search time comparisons
  - Random: analytical (expected value) and empirical search time comparisons
- Primality method: empirical term size comparisons

# Expected Value Analysis

Since $\mathrm{Exp}(X)$ is the weighted sum of the values of $X$,

$$\mathrm{Exp}(X) = \sum_{j=1}^{\infty} j p_j = \sum_{k=1}^{\infty} \sum_{j=k}^{\infty} p_j = \sum_{k=1}^{\infty} P_k \approx \sum_{k=1}^{\infty} \left(\tfrac{n-1}{n}\right)^{k-1}$$

$$= \frac{1}{1 - \frac{n-1}{n}} = n.$$

We recapitulate this conclusion as follows.

*The expected value* $\mathrm{Exp}(X)$ *of the number* $X$ *of trials required to find a term representing the function* $f$ *is approximately the size* $n = |A|^{|B|}$ *of the search space* $A^B$ *of all functions from* $B$ *to* $A$.

- Verified via empirical results with random search and exhaustive search

# 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 $A_1$)

# 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

# Potential Impact

These results are in an foundational area of pure mathematics with:

- A long history
- Many outstanding problems of theoretical significance and quantifiable difficulty
- Applications across the sciences

# The case for the prize

- Using GP, we have improved significantly on extensive past efforts of both humans and machines to solve problems related to finite algebras
- This is an important and previously unexplored application area for GP, with many open problems and quantitative measures of success

# 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!

# Other applications in mathematics?

- Define representation
- Define fitness measure (need not be perfect)
- Use/define mutation/crossover algorithms that have sufficient likelihood of producing improvements

# Towards practical autoconstructive evolution:

## self-evolution of problem-solving genetic programming systems

Lee Spector
Cognitive Science
Hampshire College

To appear in Riolo, Rick L., McConaghy, Trent, and Vladislavleva, Ekaterina, editors, *Genetic Programming Theory and Practice VIII*. Springer. 2010.

# 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.

# 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* )

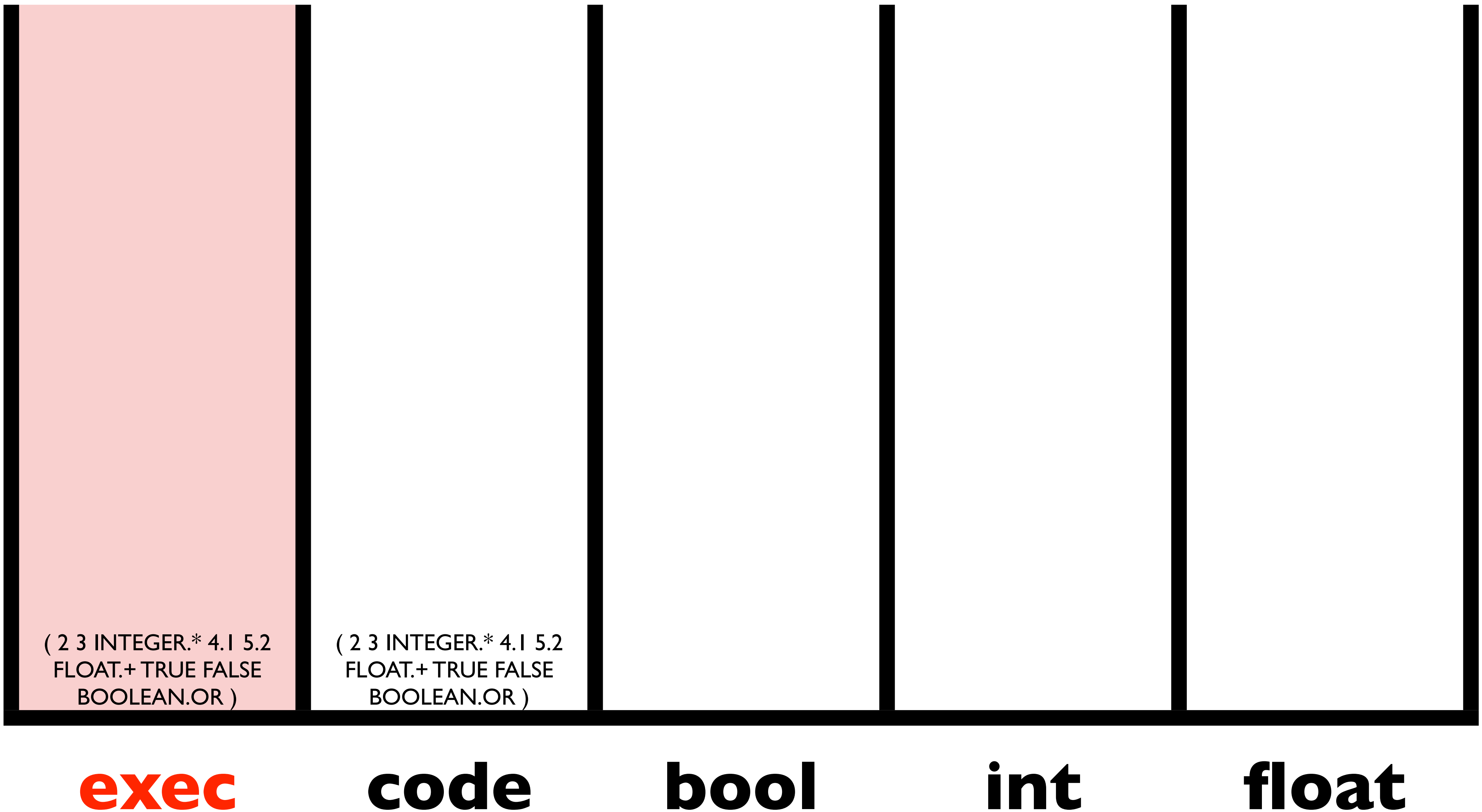
# Sample Push Instructions

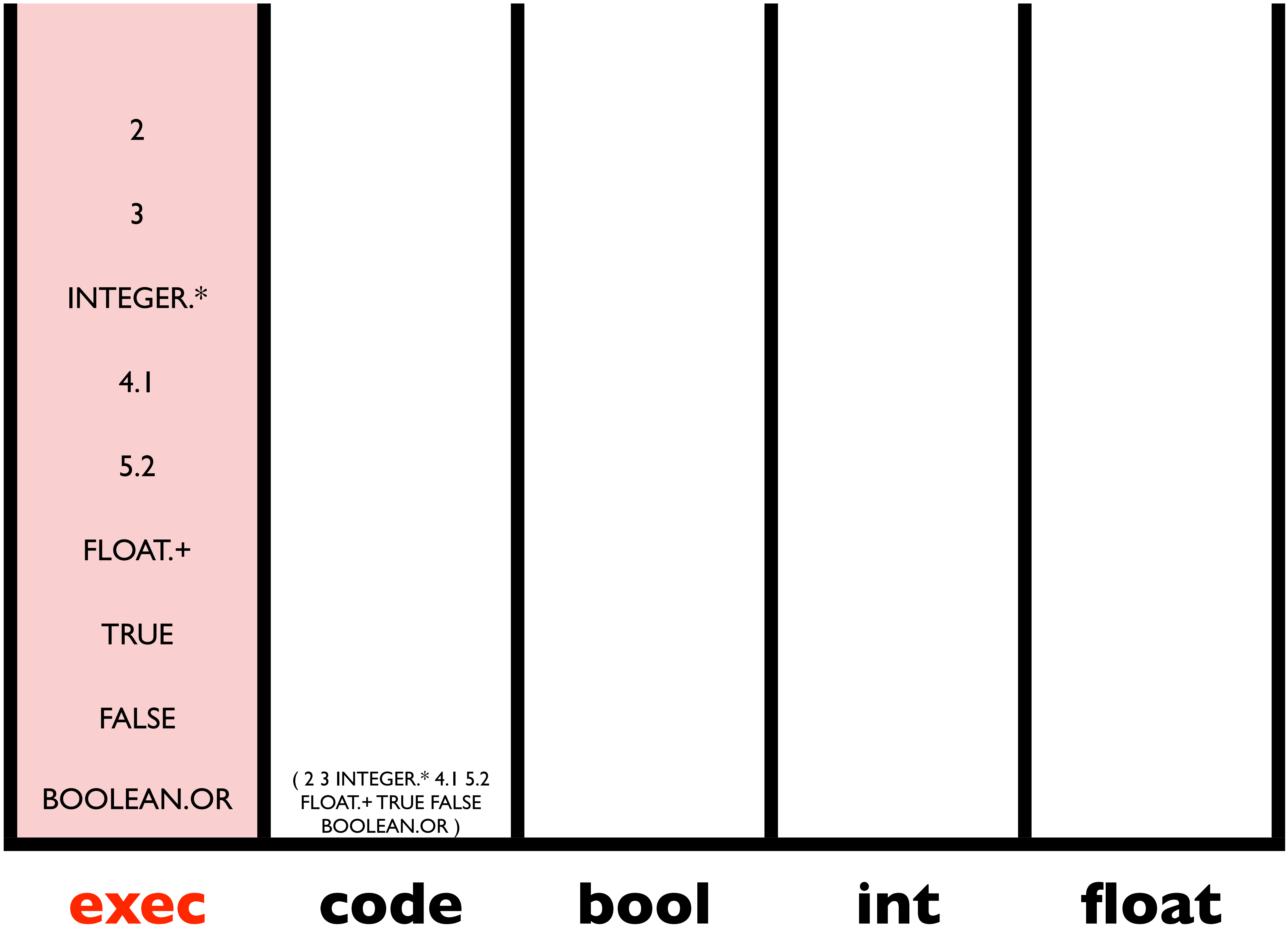
| | |
|---|---|
| Stack manipulation instructions (all types) | POP, SWAP, YANK, DUP, STACKDEPTH, SHOVE, FLUSH, $=$ |
| Math (INTEGER and FLOAT) | $+$, $-$, $/$, $*$, $>$, $<$, MIN, MAX |
| Logic (BOOLEAN) | AND, OR, NOT, FROMINTEGER |
| Code manipulation (CODE) | QUOTE, CAR, CDR, CONS, INSERT, LENGTH,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

# Push(3) Semantics

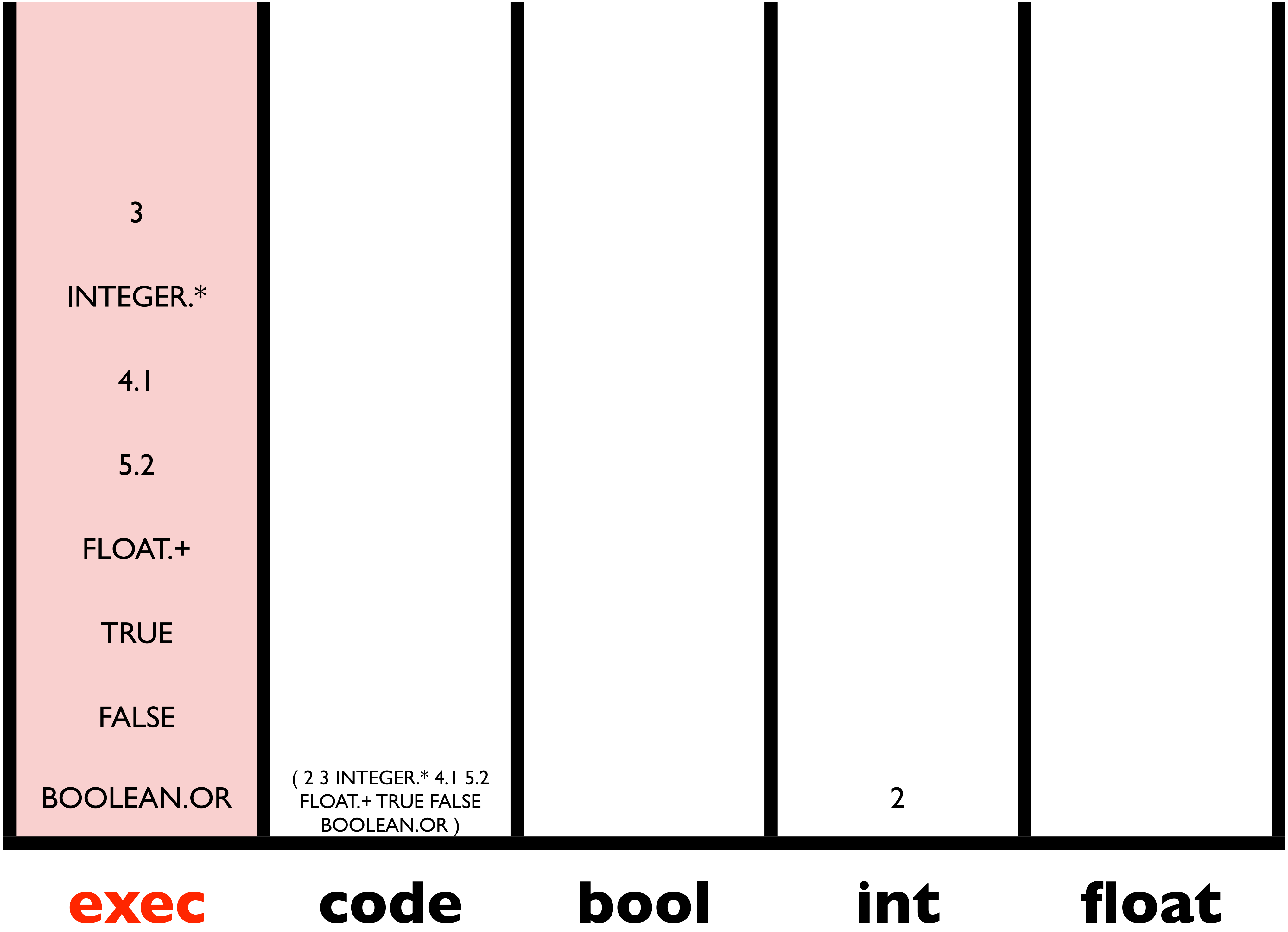
- To execute program $P$:
  1. Push $P$ onto the `EXEC` stack.
  2. While the `EXEC` stack is not empty, pop and process the top element of the `EXEC` stack, $E$:
     (a) If $E$ is an instruction: execute $E$ (accessing whatever stacks are required).
     (b) If $E$ is a literal: push $E$ onto the appropriate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

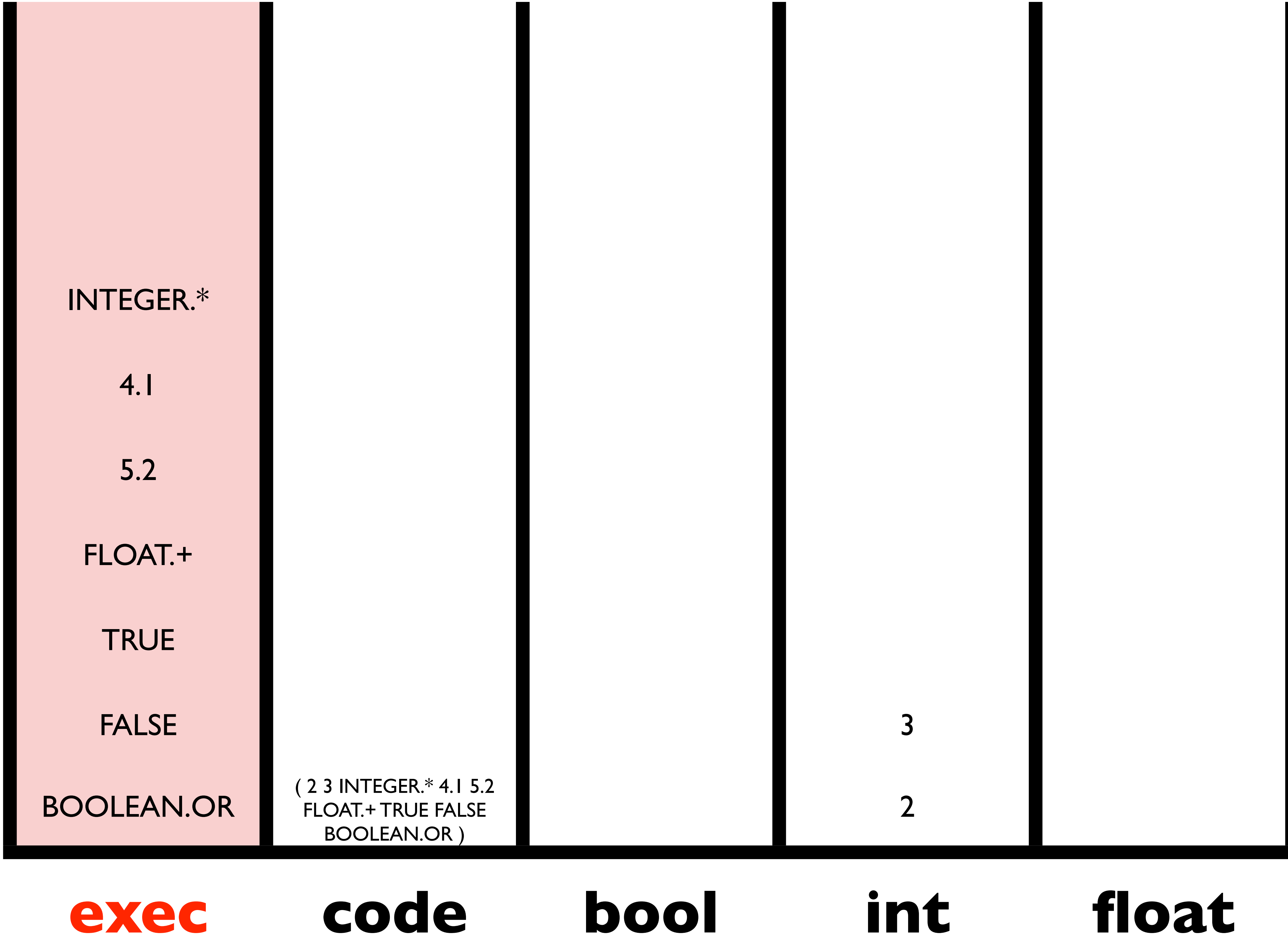
```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+
TRUE FALSE BOOLEAN.OR )
```



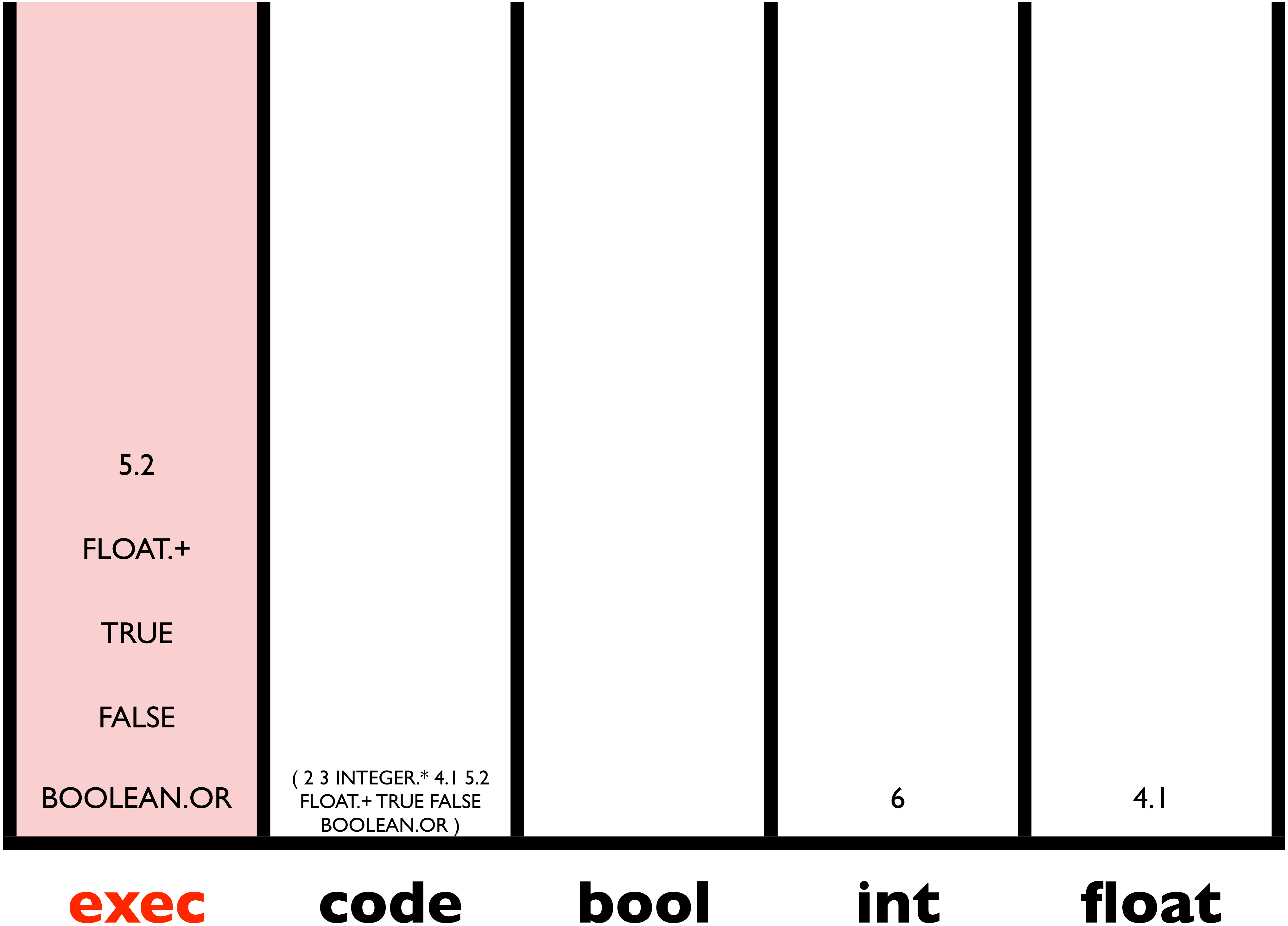


2
3
INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )
exec
code
bool
int
float


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

| 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 | |

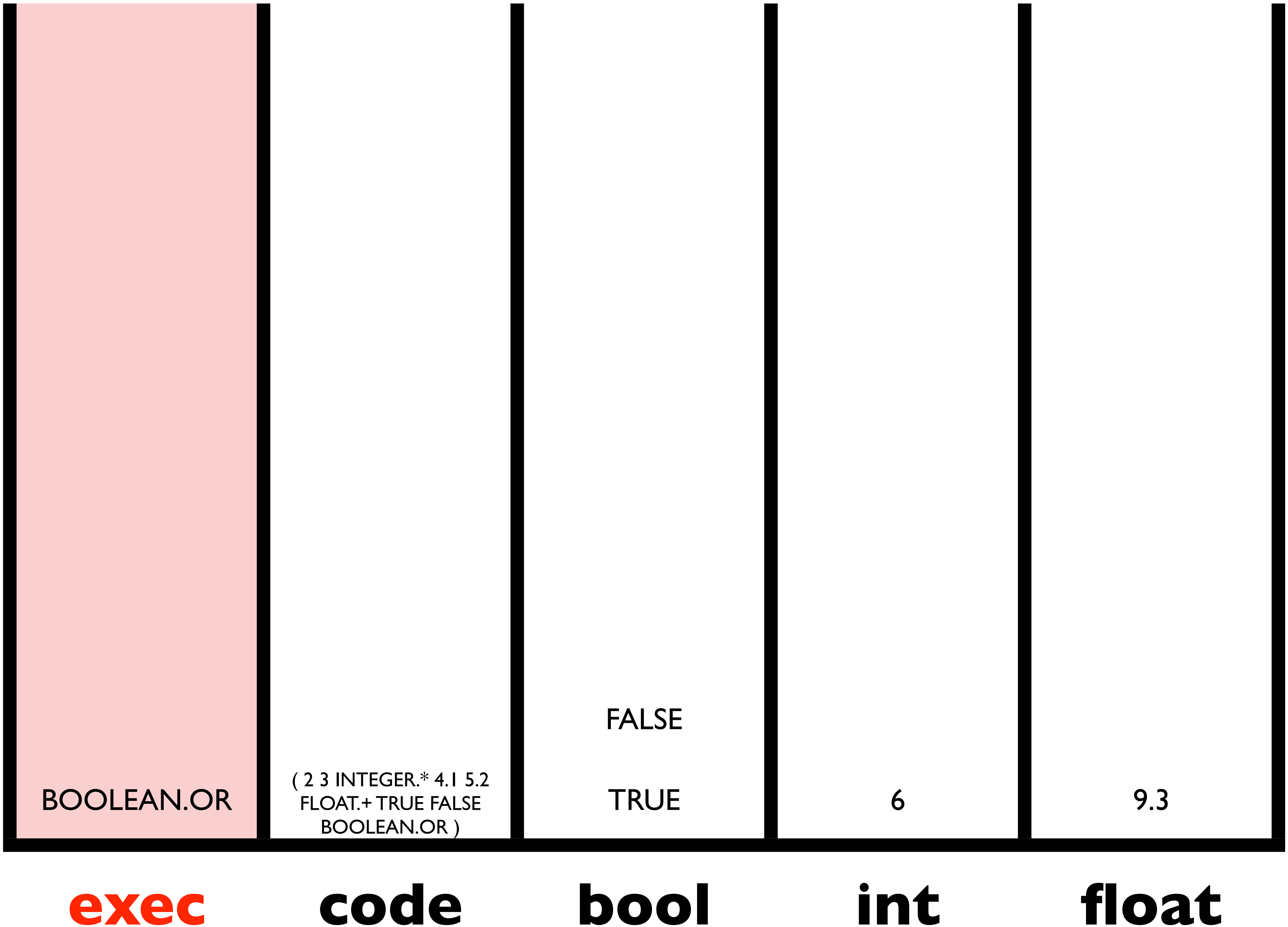


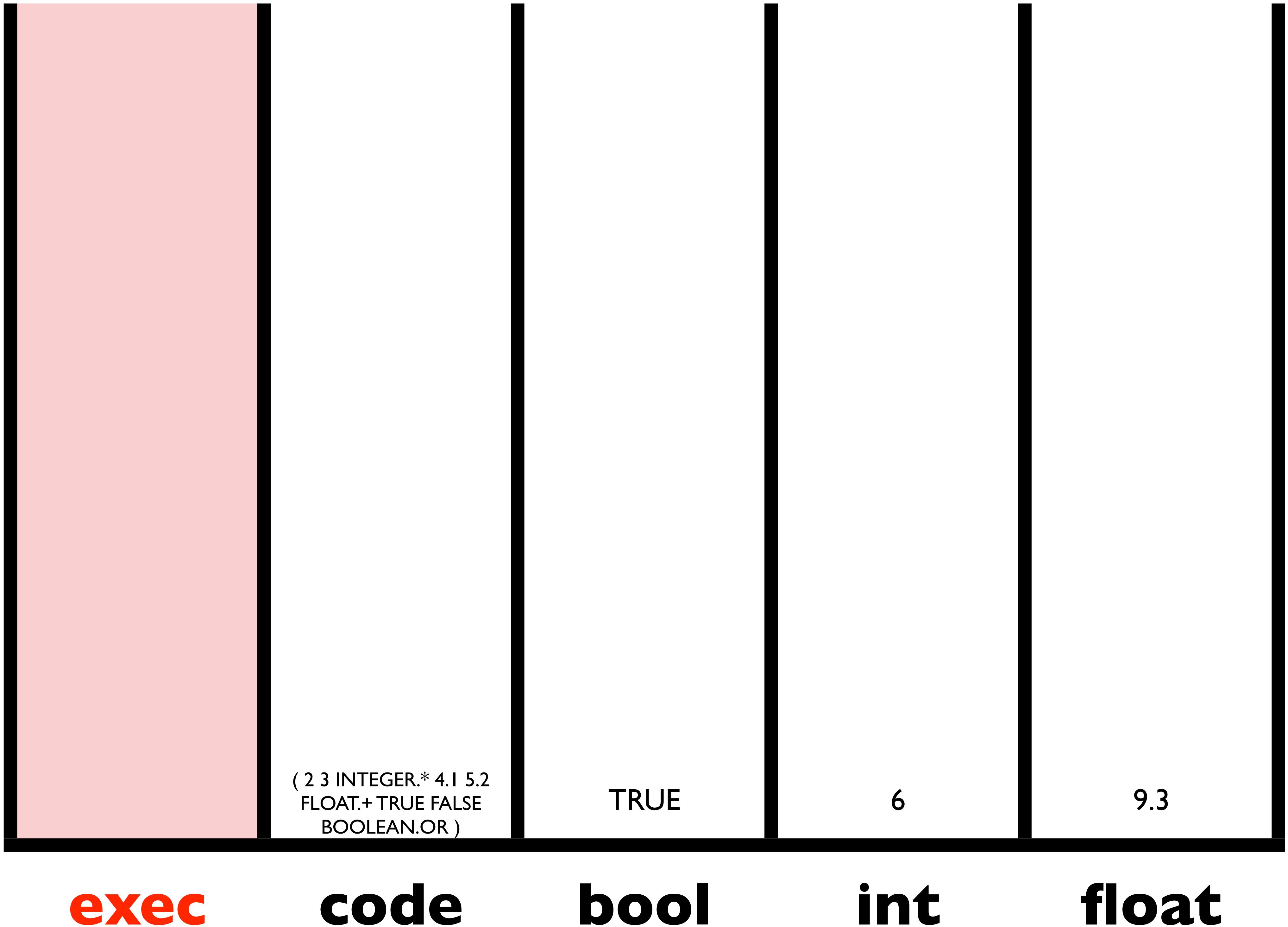
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR )
6
4.1
exec
code
bool
int
float

| 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 |

| 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 |


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

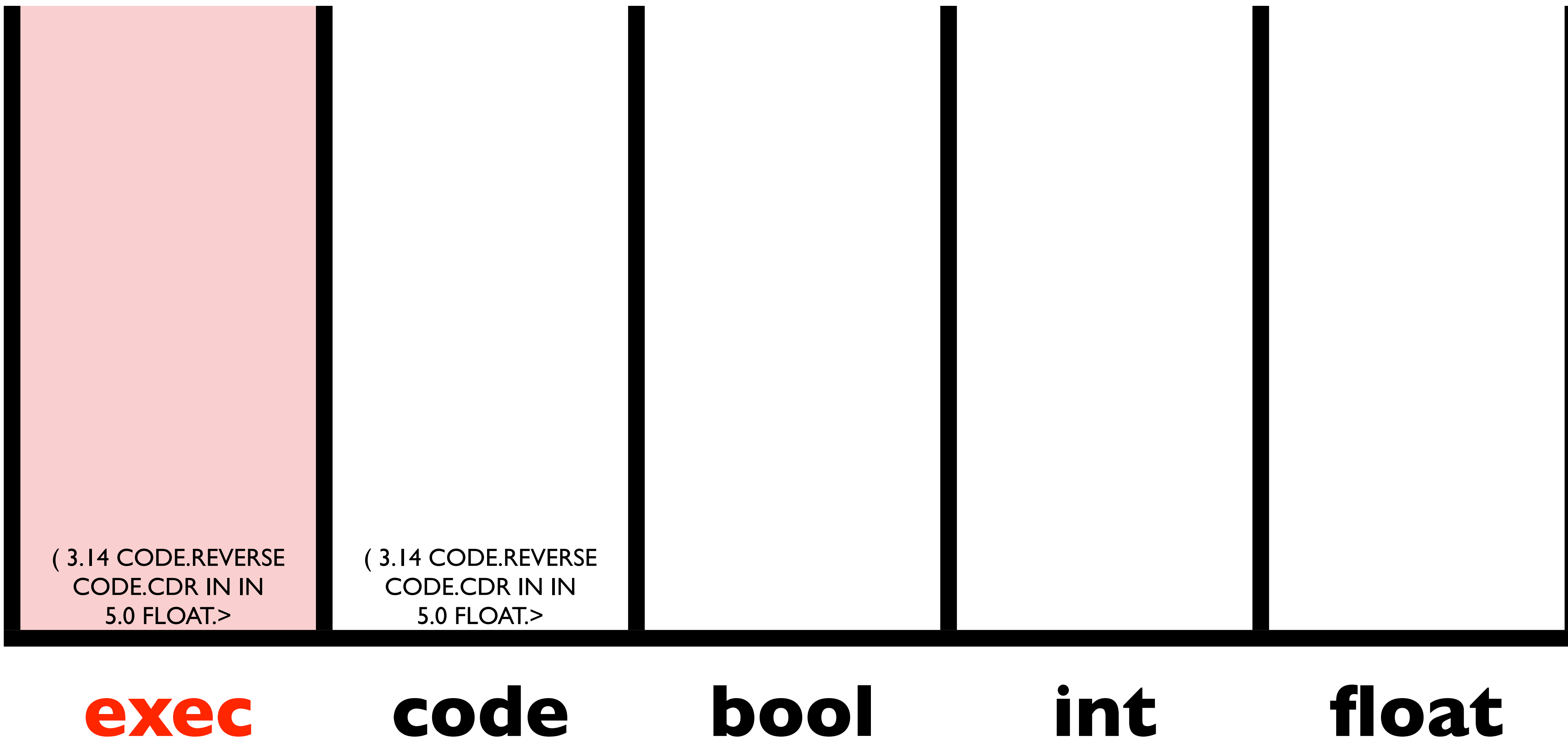
# Same Results

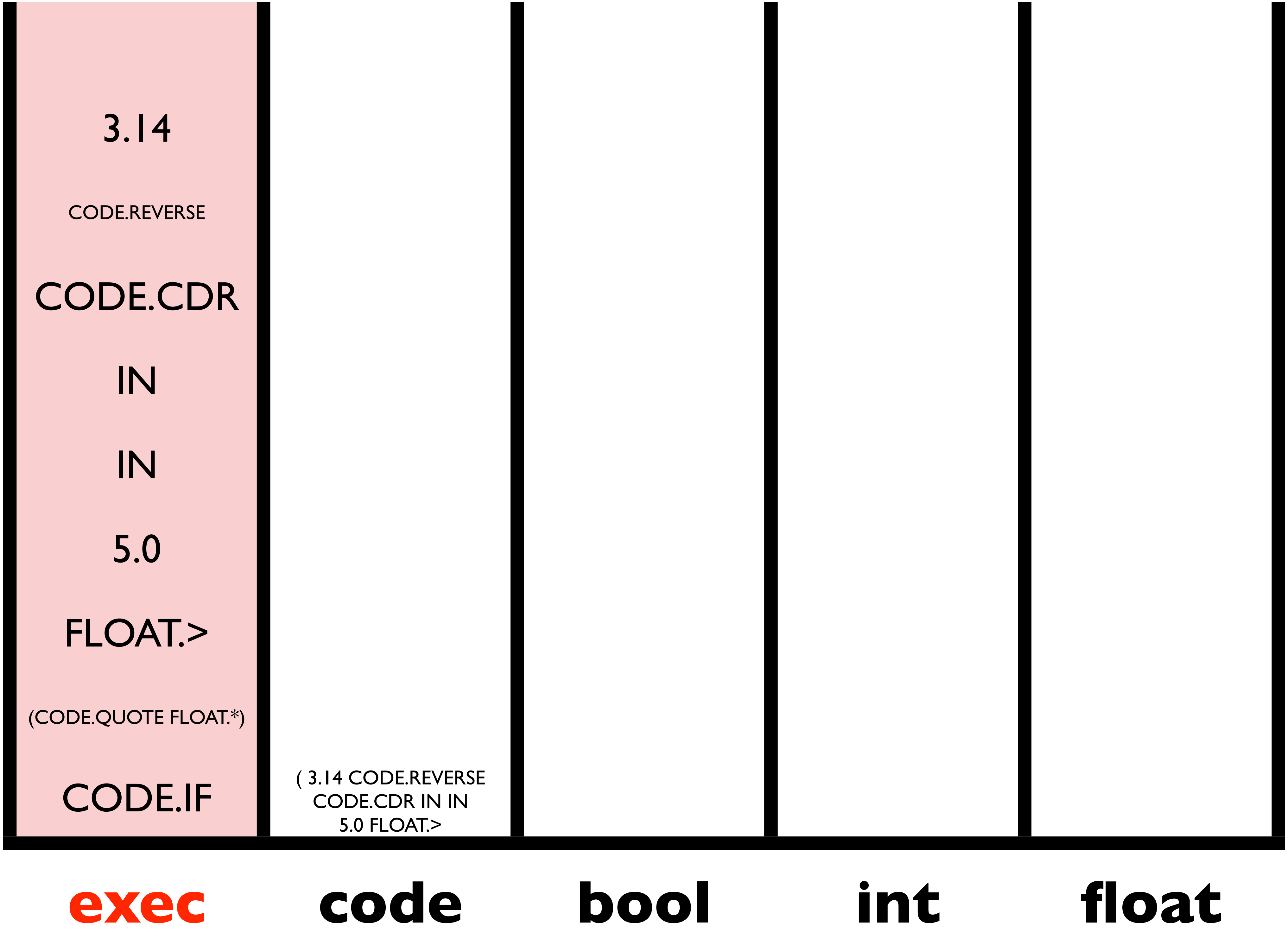
```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+
    TRUE FALSE BOOLEAN.OR )
```

```
( 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 )

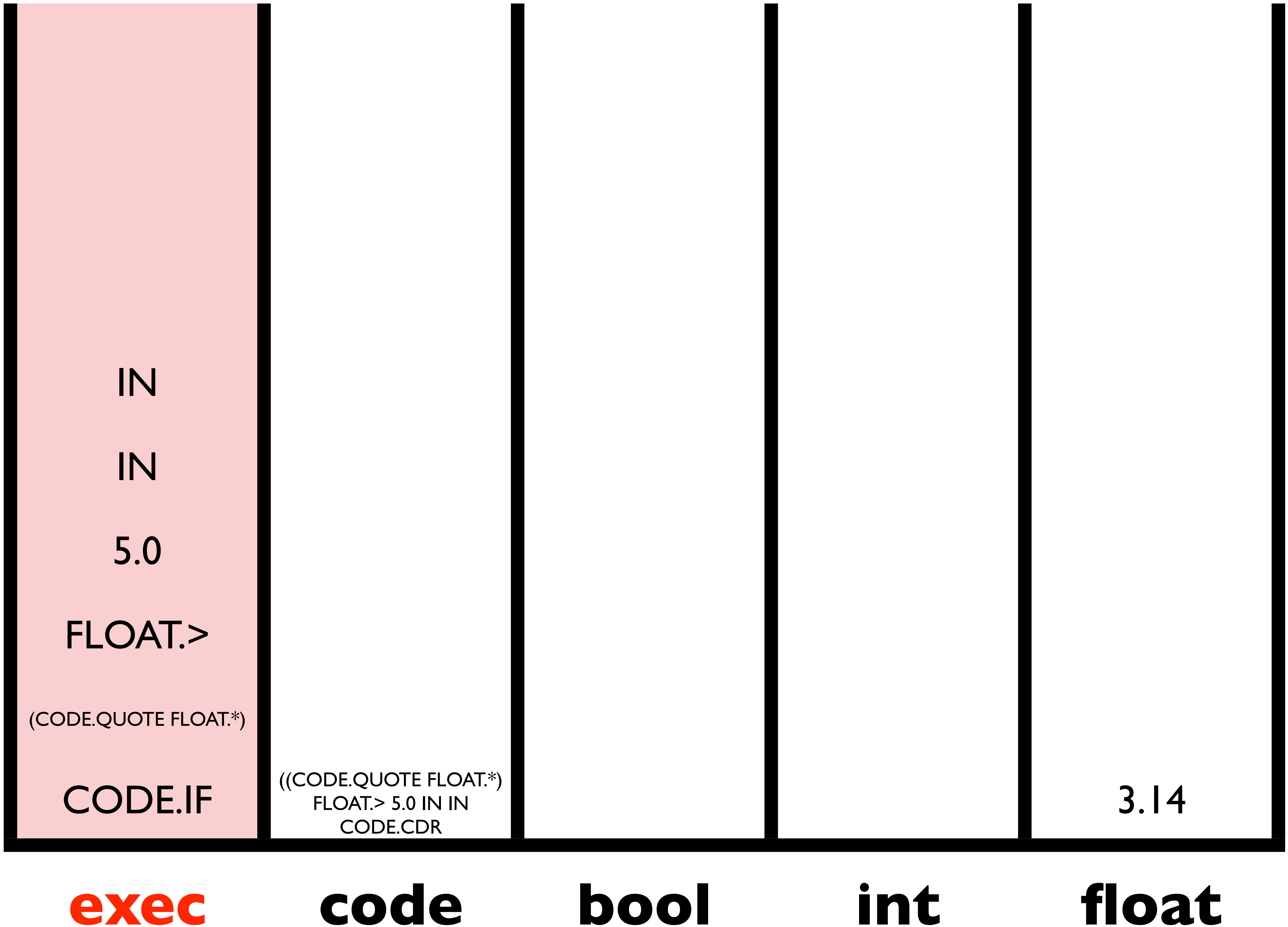
IN=4.0




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

| 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 |

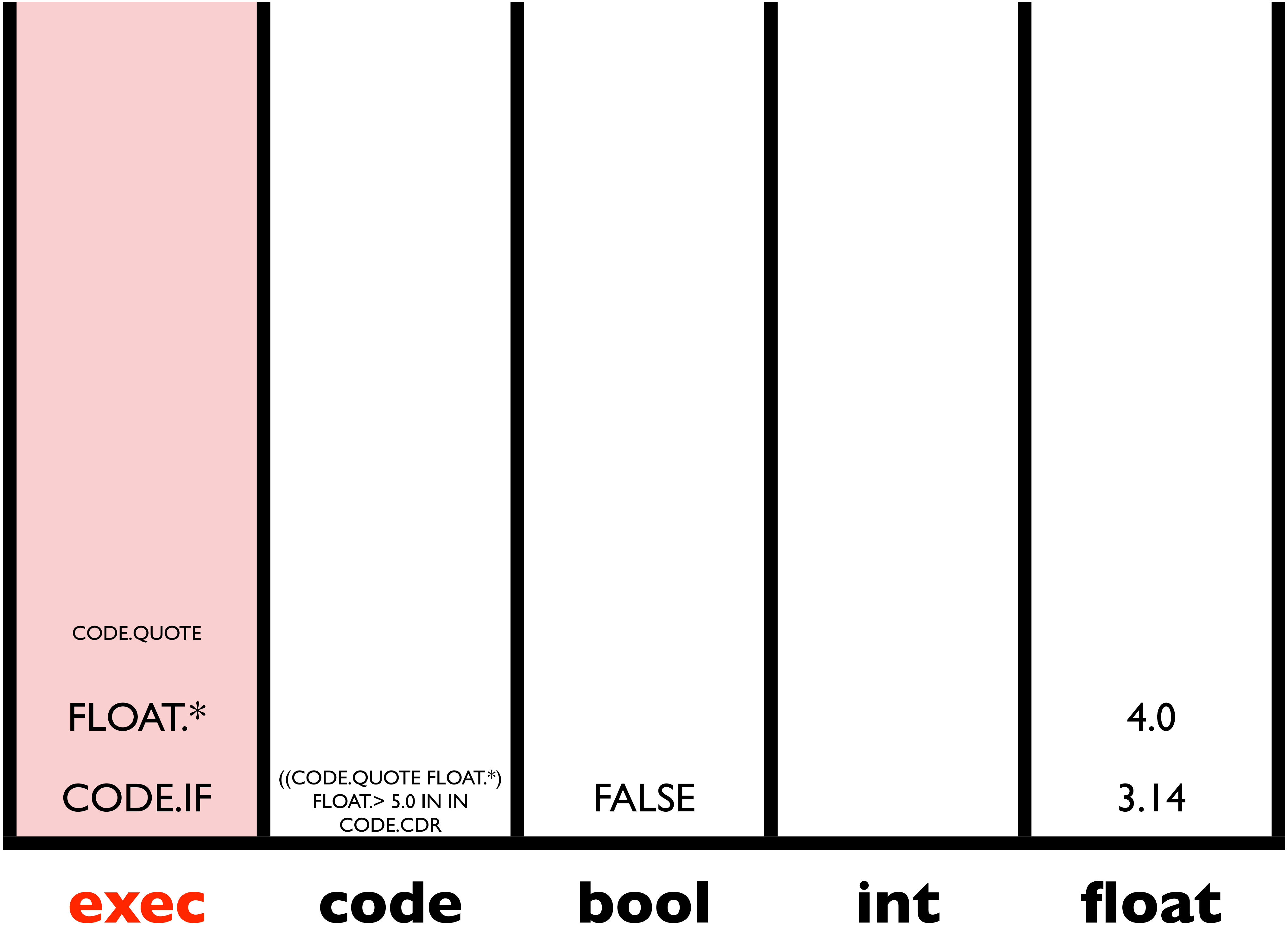

IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
3.14
exec
code
bool
int
float

| 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 |

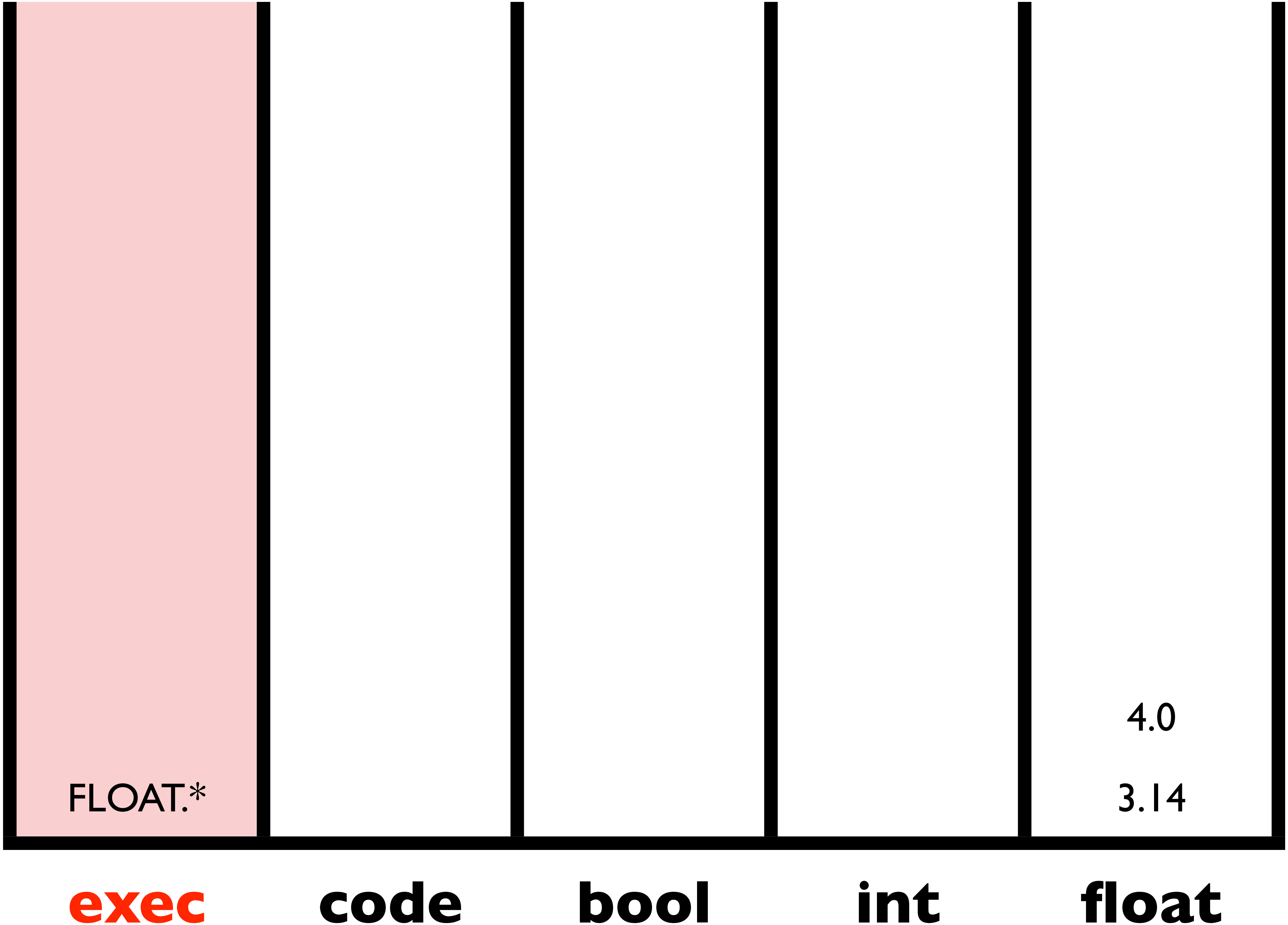
| exec | code | bool | int | float |
|---|---|---|---|---|
| 5.0 | | | | |
| FLOAT.> | | | | 4.0 |
| (CODE.QUOTE FLOAT.*) | | | | 4.0 |
| CODE.IF | ((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR | | | 3.14 |

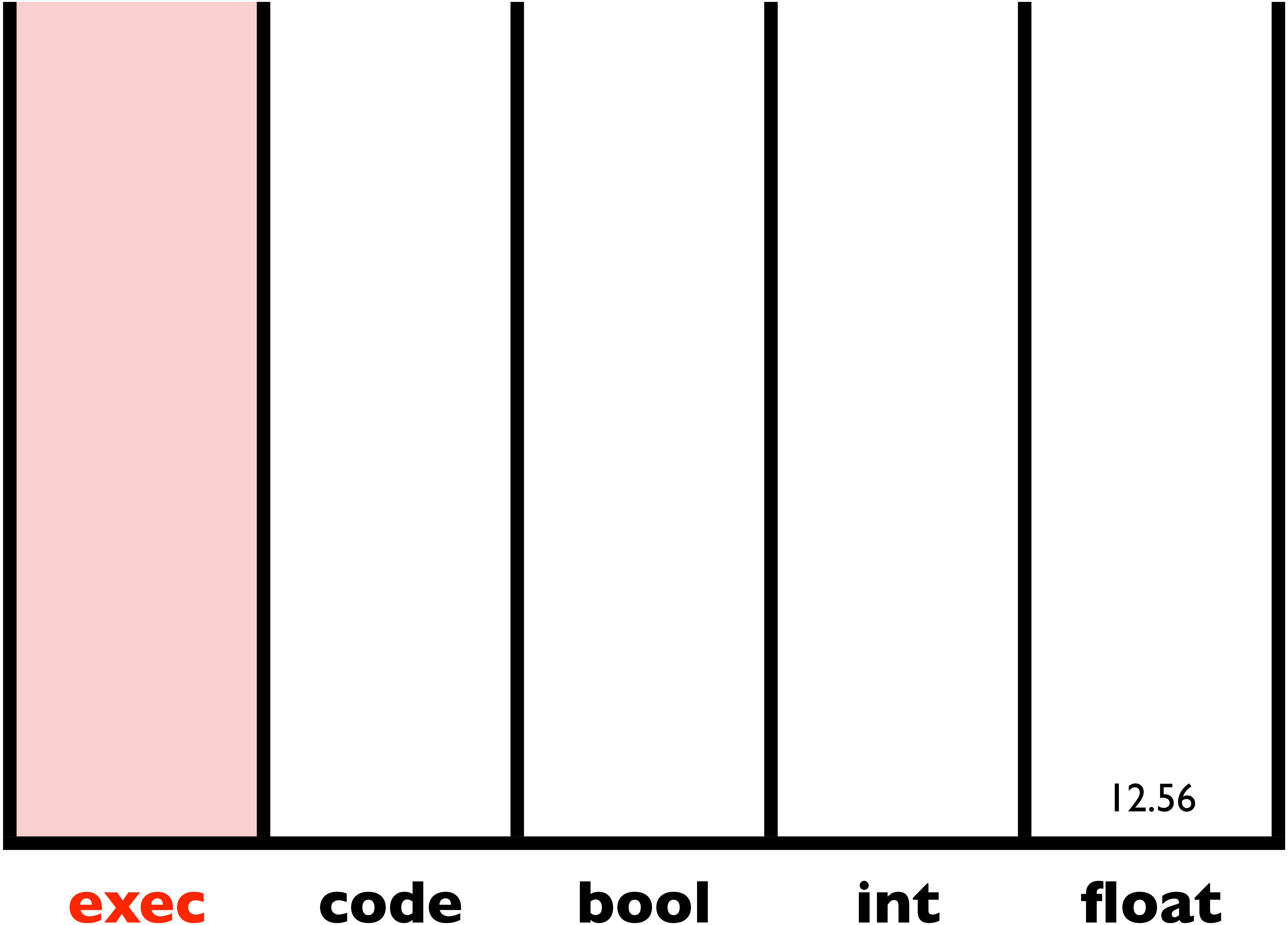
| exec | code | bool | int | float |
|---|---|---|---|---|
| | | | | 5.0 |
| FLOAT.> | | | | 4.0 |
| (CODE.QUOTE FLOAT.*) | | | | 4.0 |
| CODE.IF | ((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR | | | 3.14 |

| 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

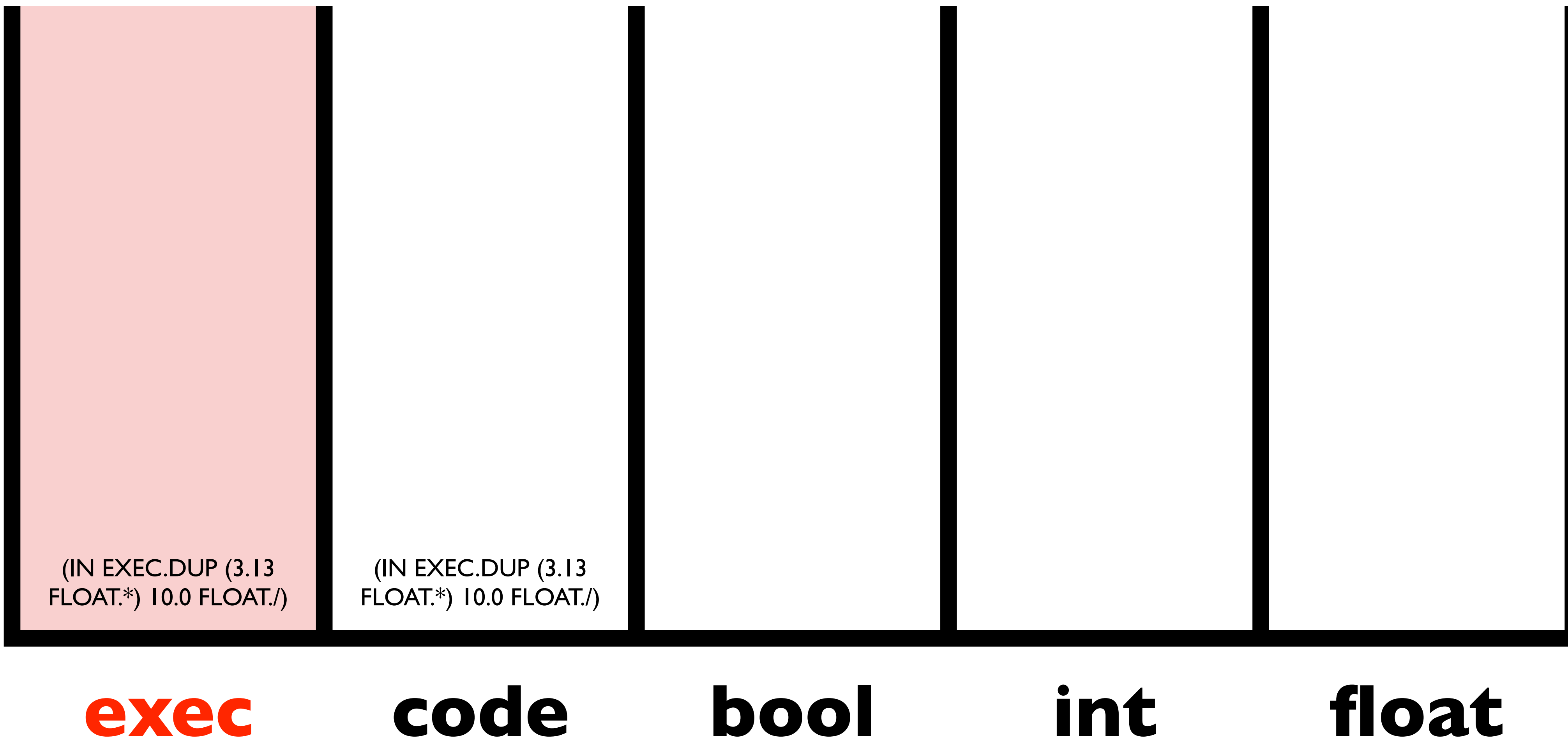
| exec | code | bool | int | float |
| --- | --- | --- | --- | --- |
| | FLOAT.* | | | 4.0 |
| CODE.IF | ((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR | FALSE | | 3.14 |


FLOAT.*
4.0
3.14
exec
code
bool
int
float

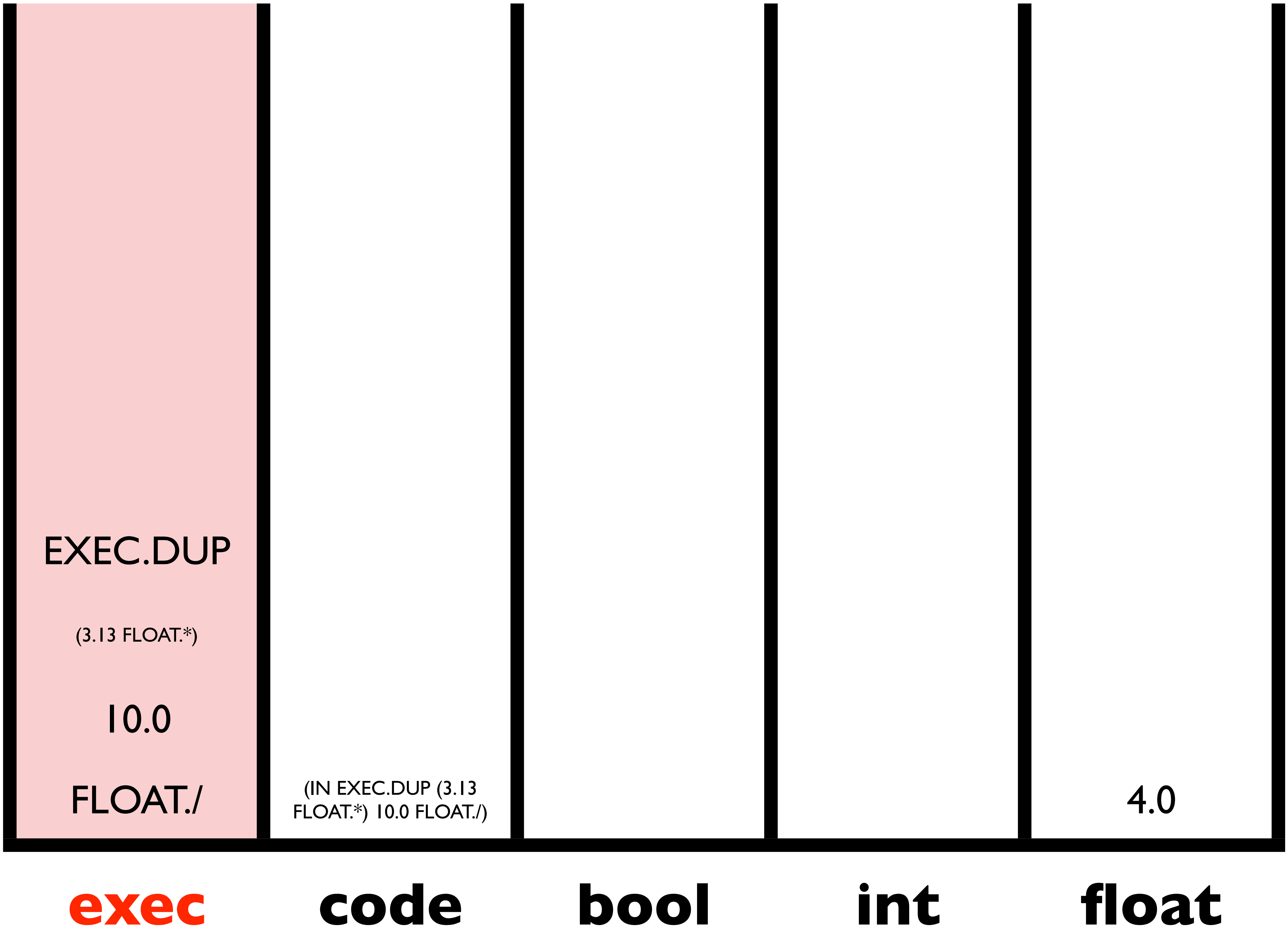

12.56
exec
code
bool
int
float

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

```
IN=4.0
```

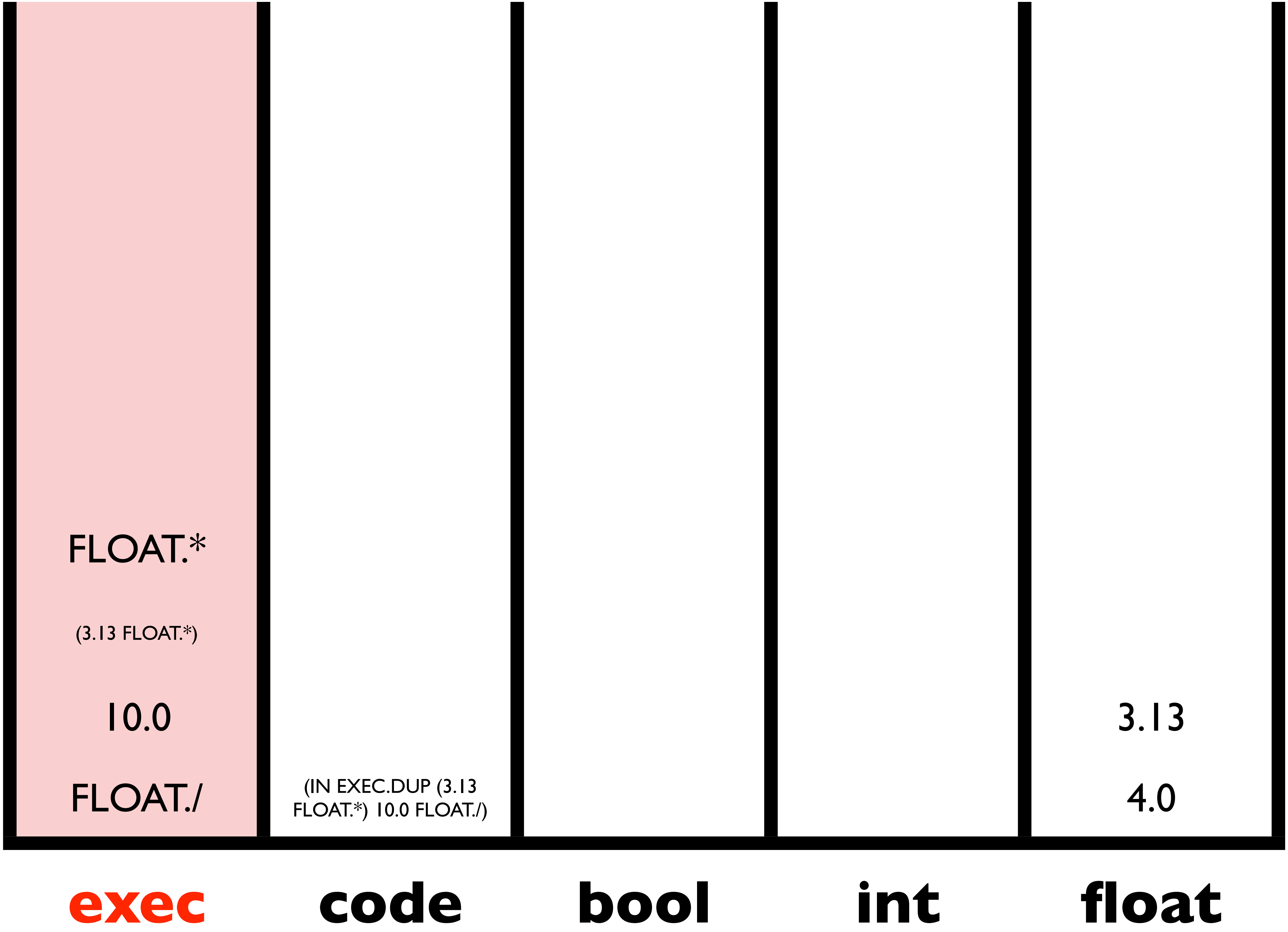


| exec | code | bool | int | float |
|---|---|---|---|---|
| IN | | | | |
| EXEC.DUP | | | | |
| (3.13 FLOAT.*) | | | | |
| 10.0 | | | | |
| FLOAT./ | (IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./) | | | |

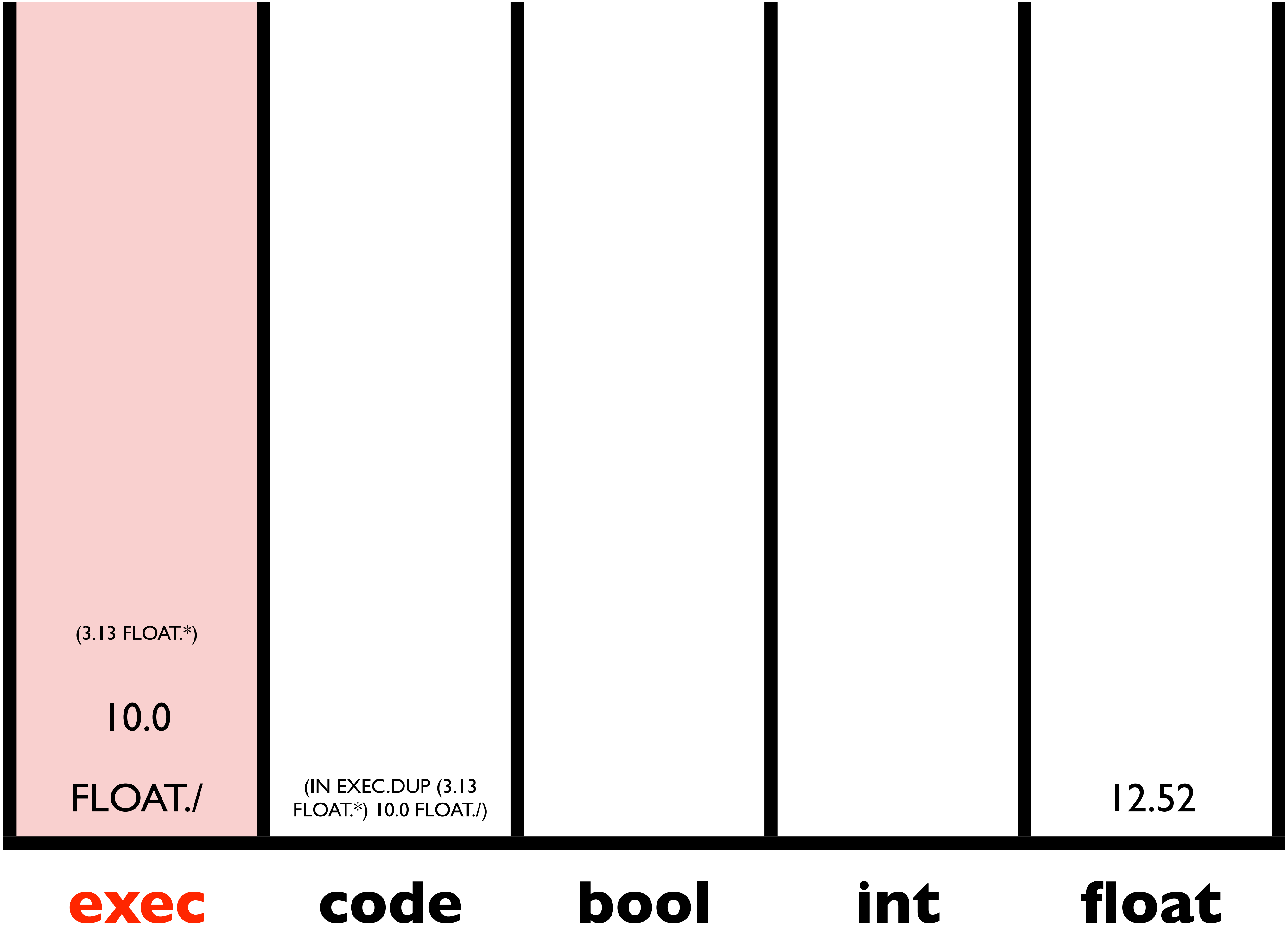

EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

| 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 |

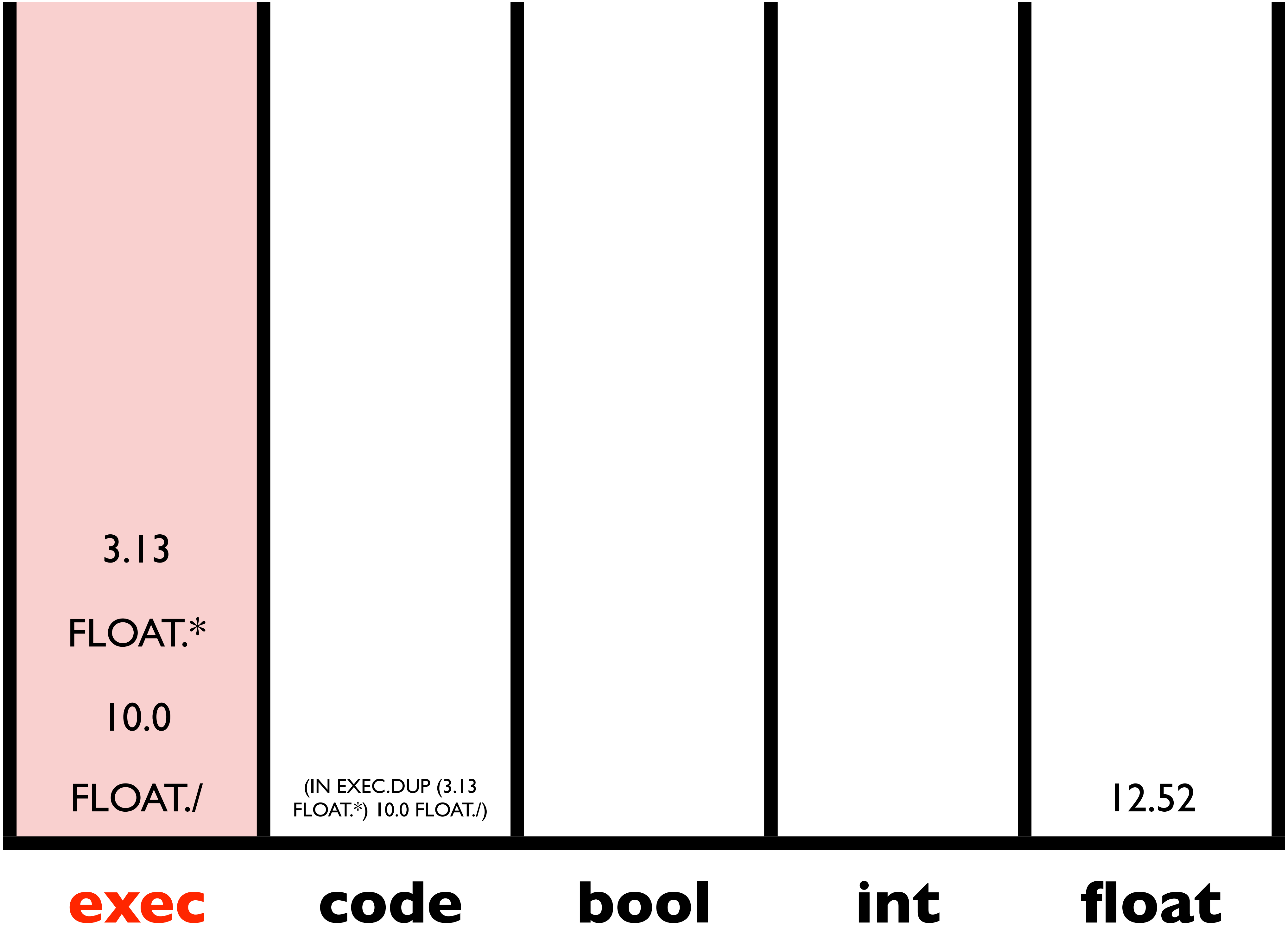
| exec | code | bool | int | float |
|---|---|---|---|---|
| 3.13 | | | | |
| FLOAT.* | | | | |
| (3.13 FLOAT.*) | | | | |
| 10.0 | | | | |
| FLOAT./ | (IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./) | | | 4.0 |

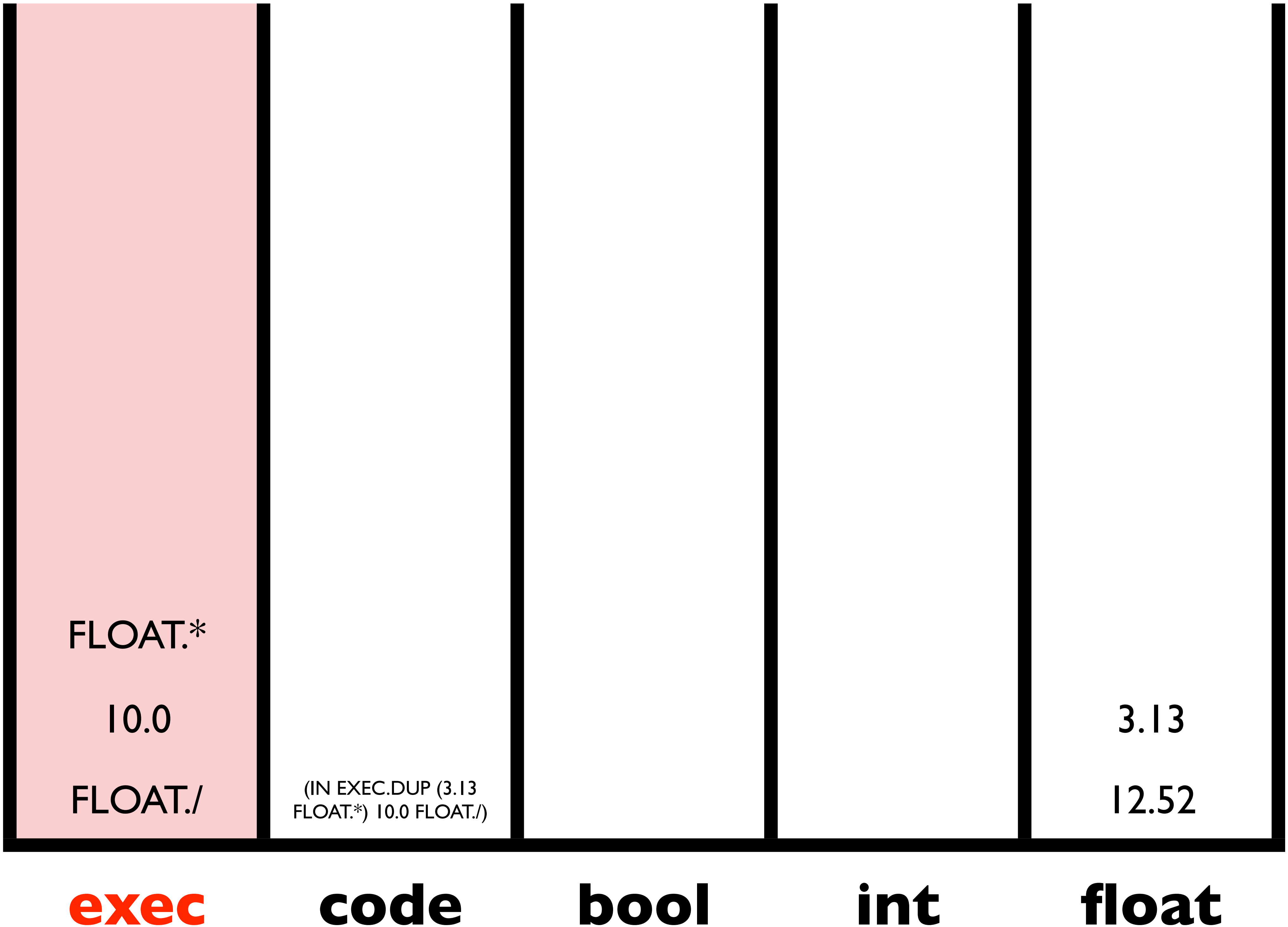


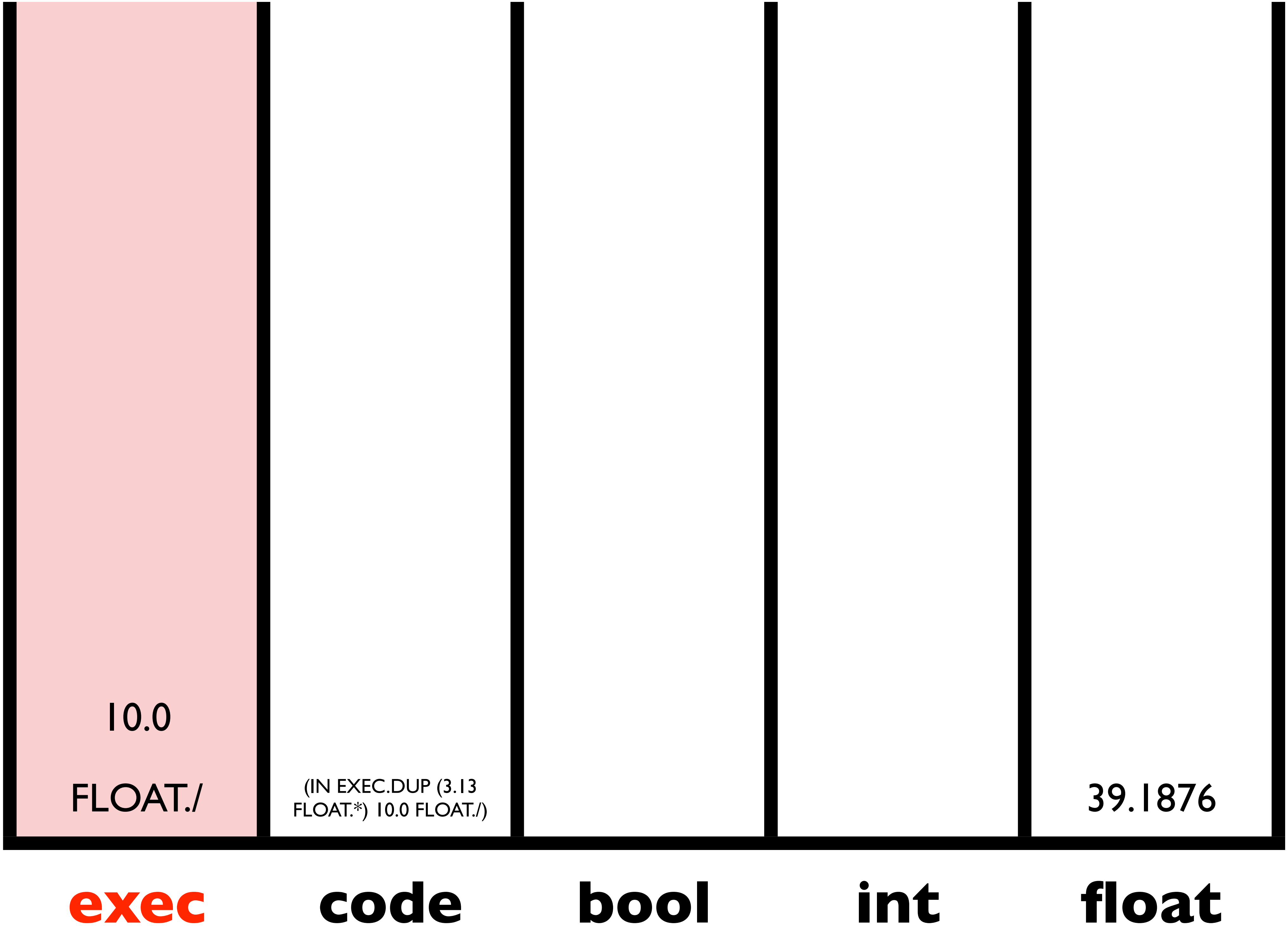
FLOAT.*
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
3.13
4.0
exec
code
bool
int
float

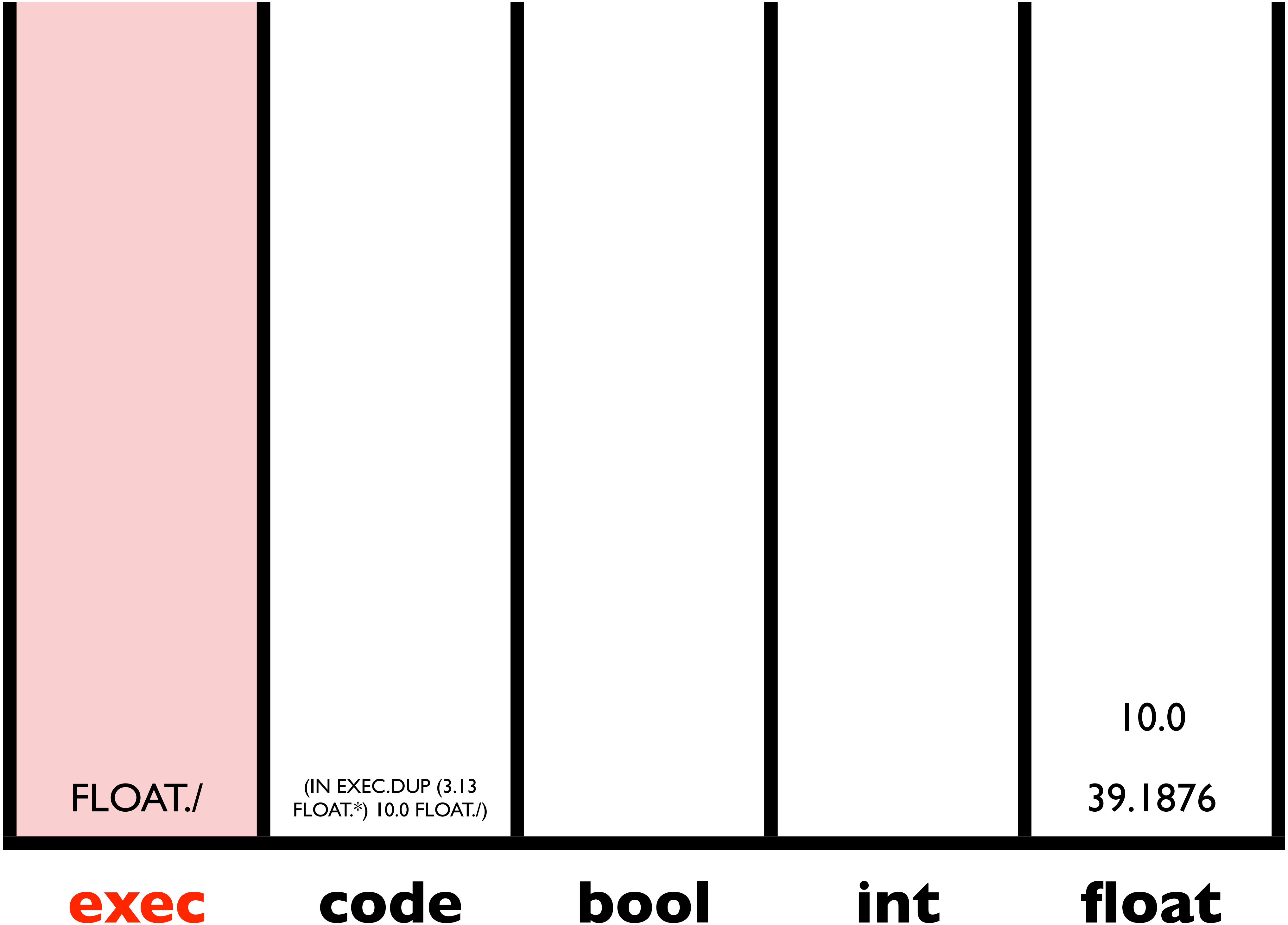


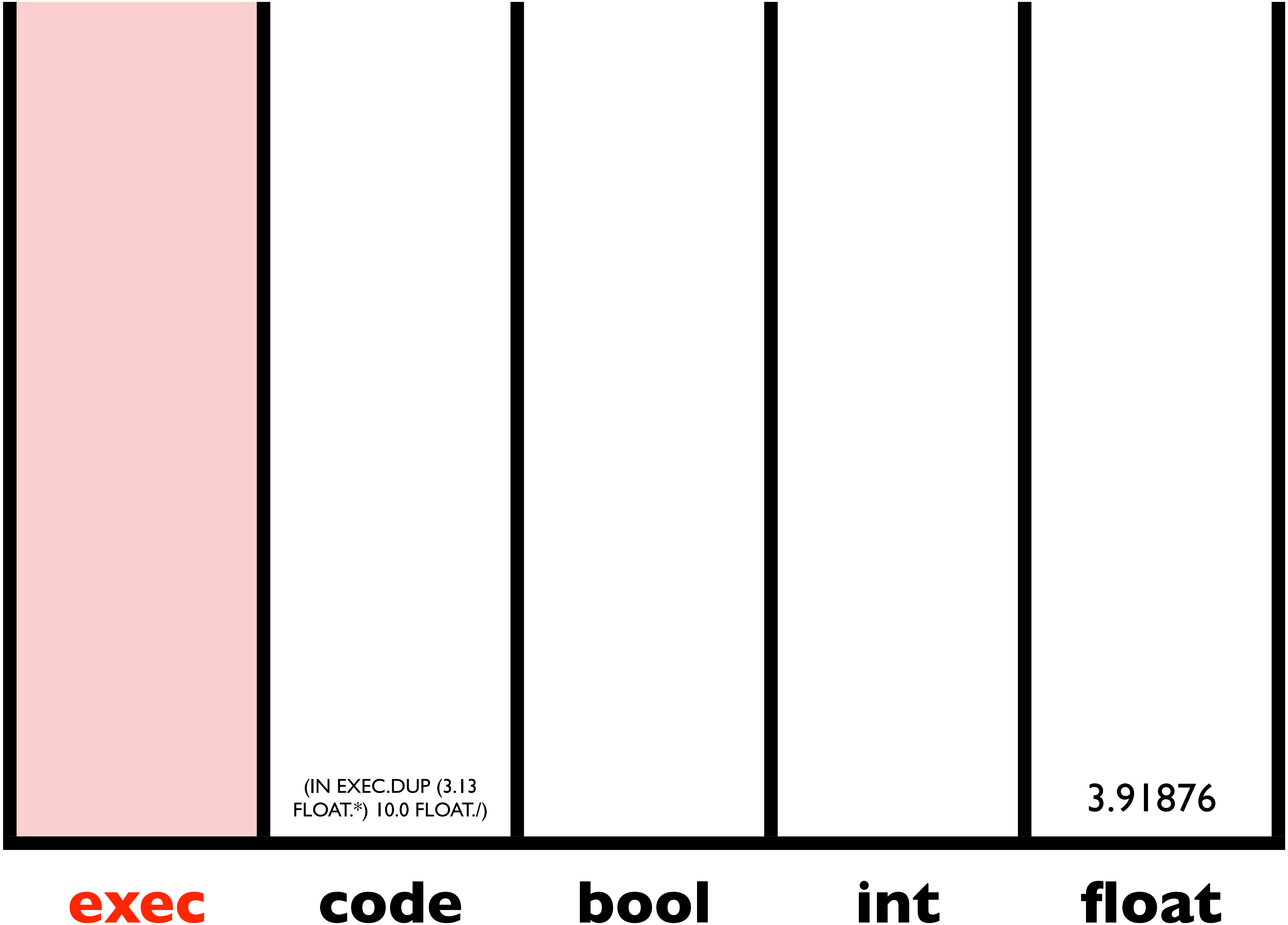
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
12.52
exec
code
bool
int
float


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

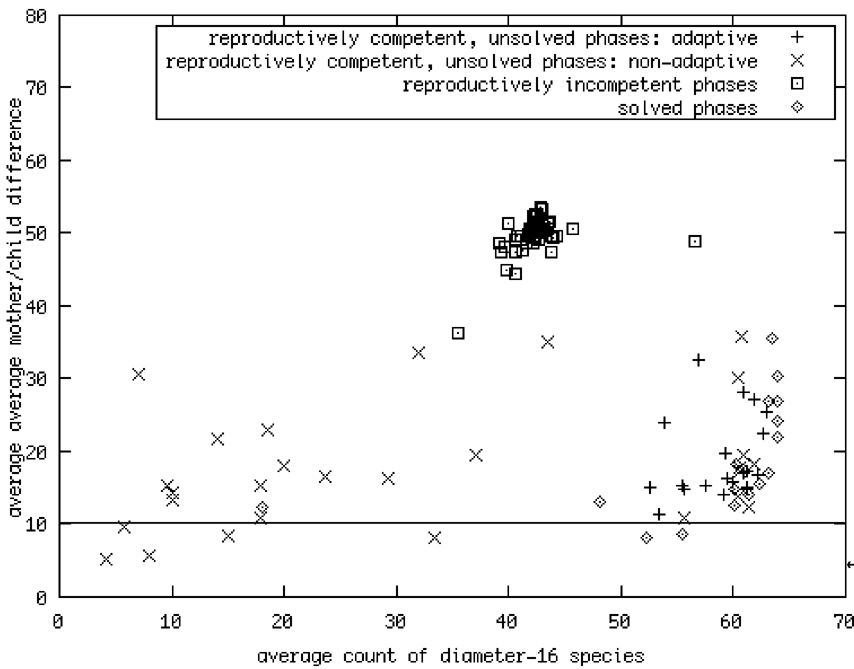
- Reversing a list
- Factorial (many algorithms)
- Fibonacci (many algorithms)
- Parity (any size input)
- Exponentiation
- Sorting

# Pushpop

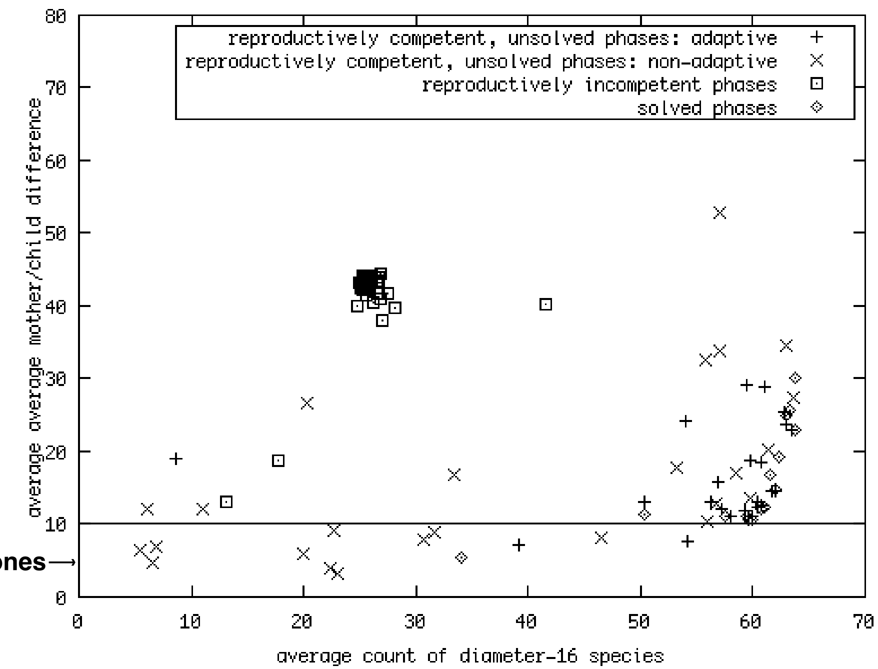
- A soup of evolving Push programs.
- Reproductive procedures emerge ex nihilo:
  - No hand-designed "ancestor."
  - Children constructed by any computable process.
  - No externally applied mutation procedure or rate.
  - Exact clones are prohibited, but near-clones are permitted.
- Selection for problem-solving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

# # 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



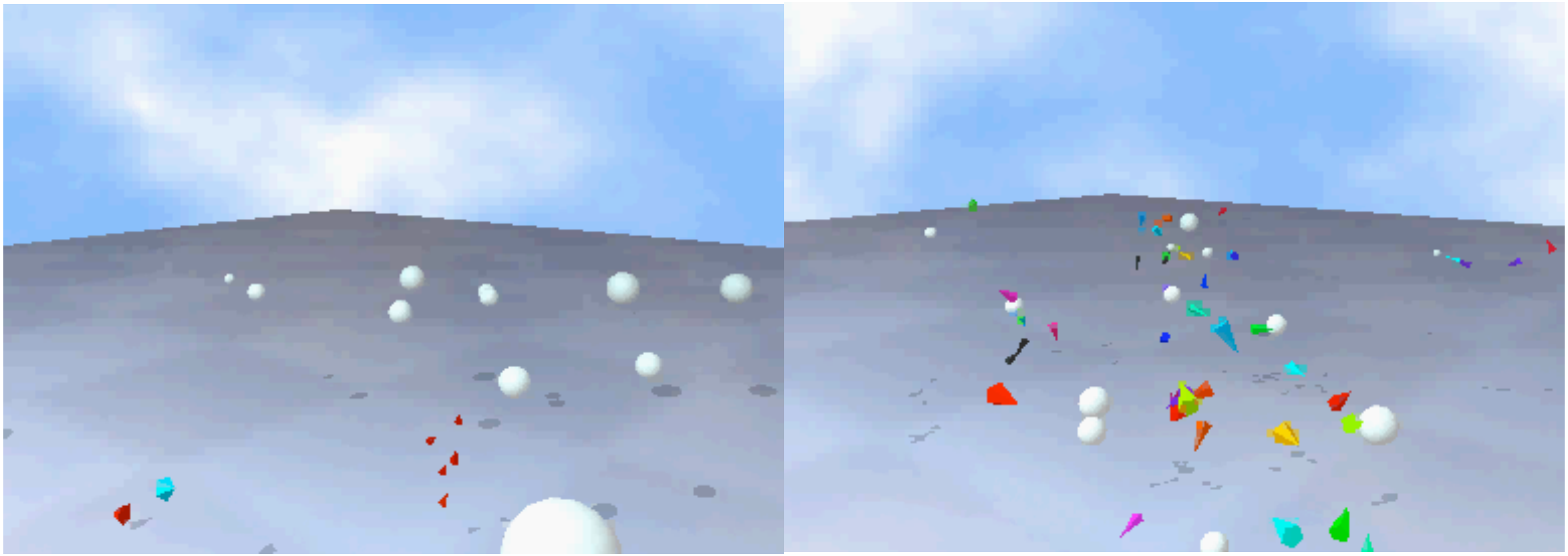

**Runs including sexual instructions**

**Runs without sexual instructions**

# 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.

# 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))))))
```

# Conclusion

Genetic programming systems have an important role to play in the future of mathematics.