

# Expressive Languages for Evolved Programs and Autoconstructive Evolution

Lee Spector

Cognitive Science

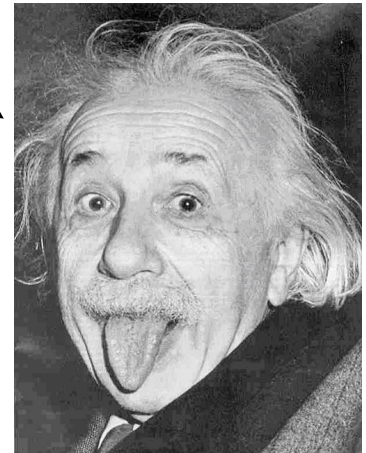
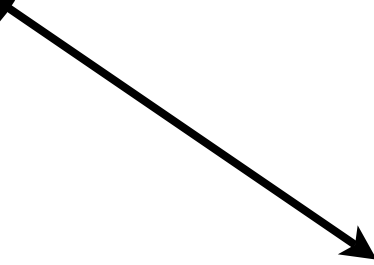
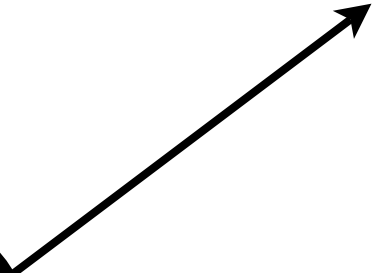
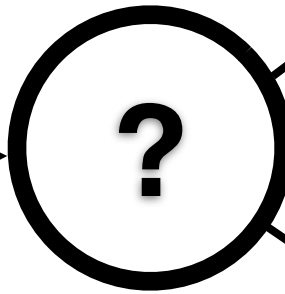
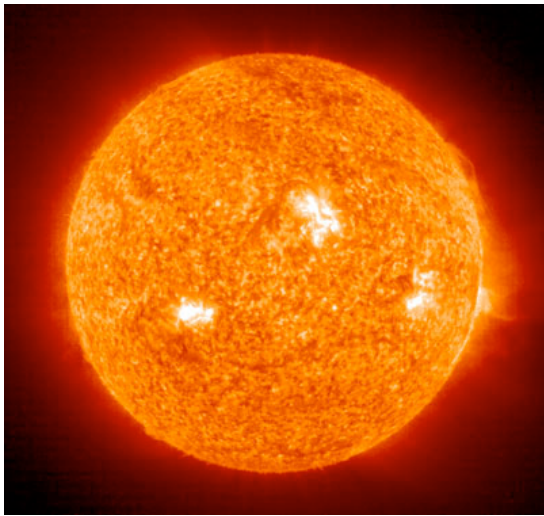
Hampshire College

(& Computer Science, U. Mass Amherst)

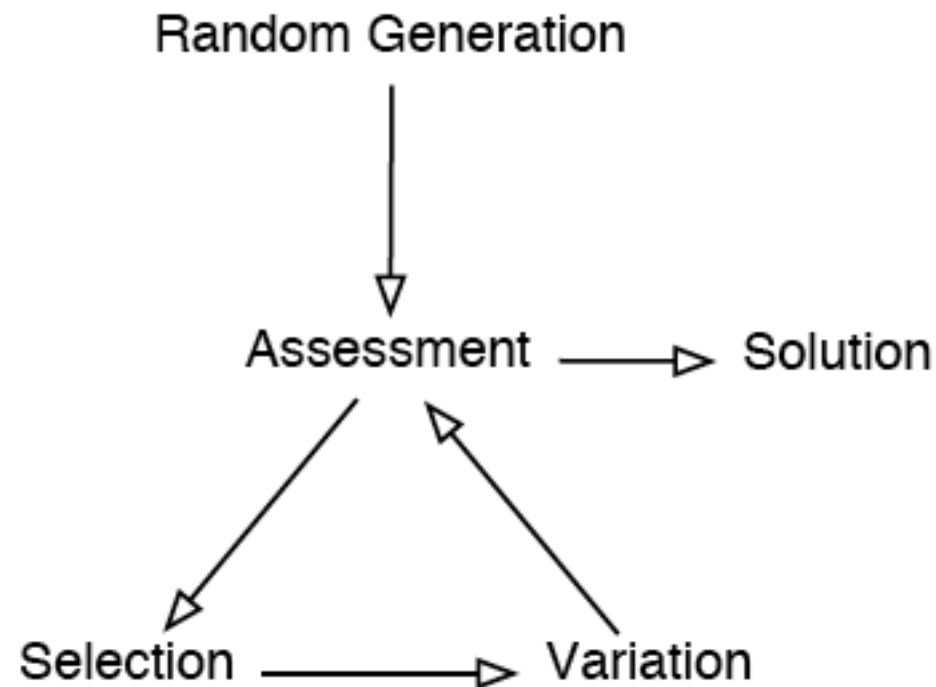
<http://hampshire.edu/lspector>

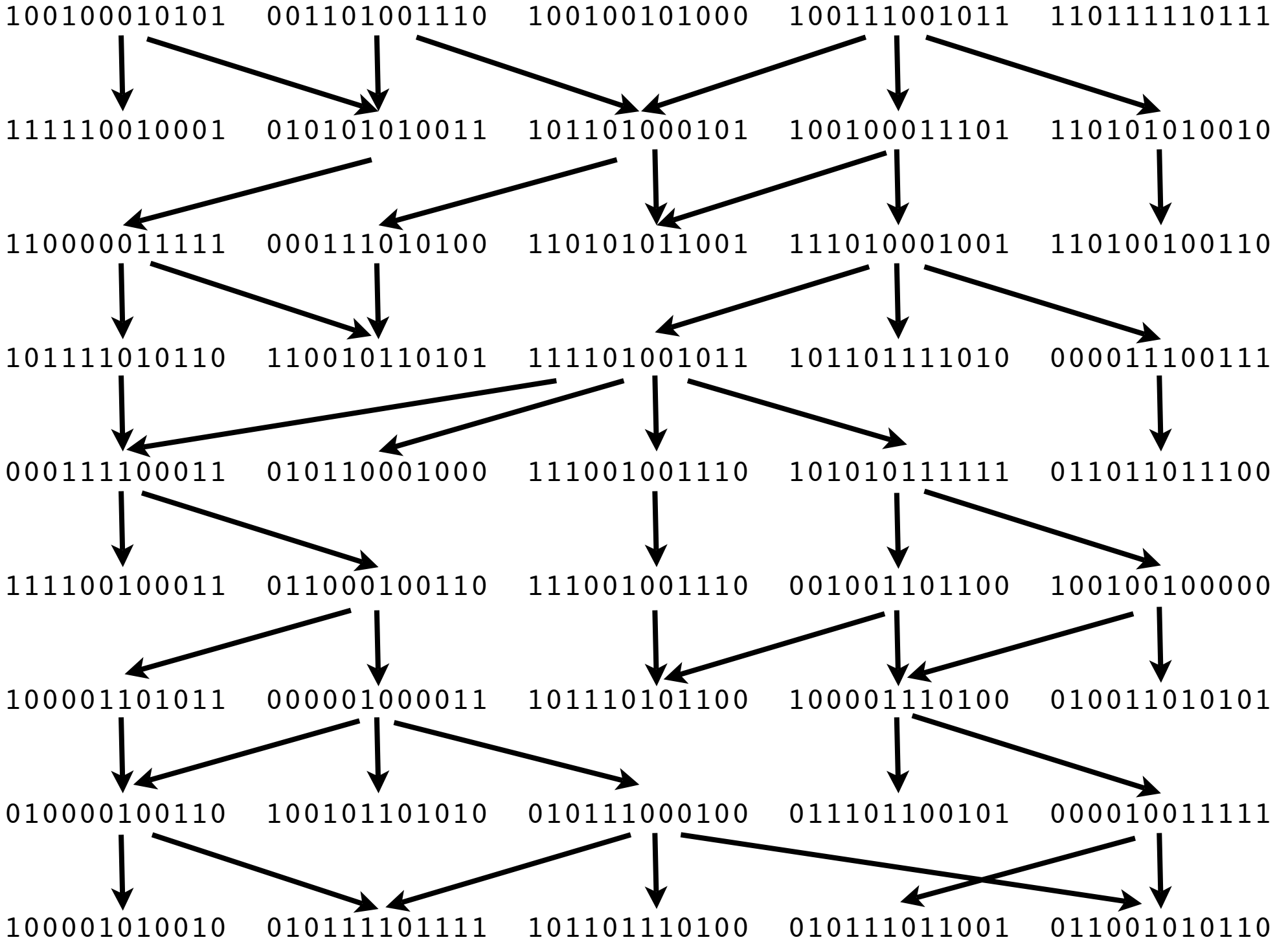
# Outline

- Evolutionary computation comes of age
- Representations
  - Genetic programming
  - Expressive languages
  - Push
- Autoconstructive Evolution



# Evolutionary Computation

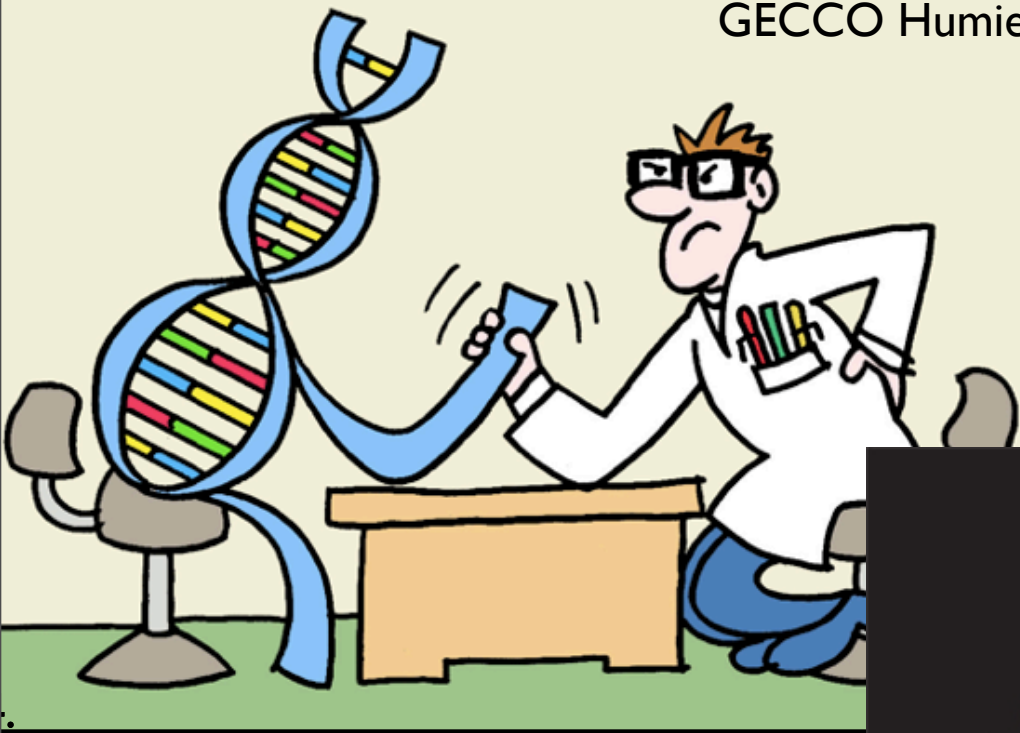




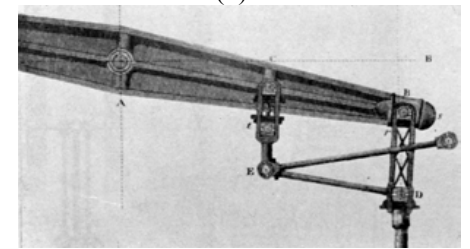
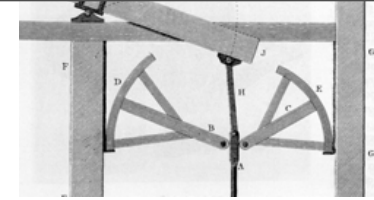
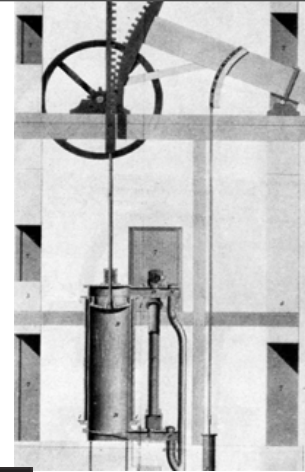
# Traditional Genetic Algorithms

- Interesting dynamics
- Rarely solve interesting hard problems

GECCO Humies

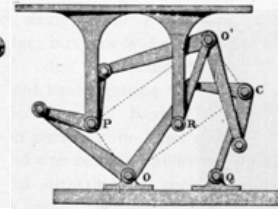
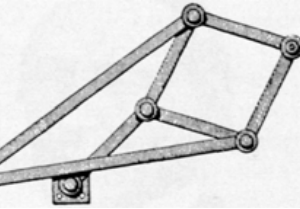
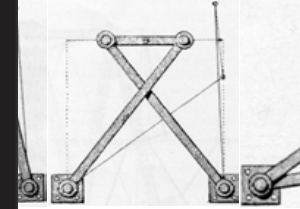


Lipson



(a)

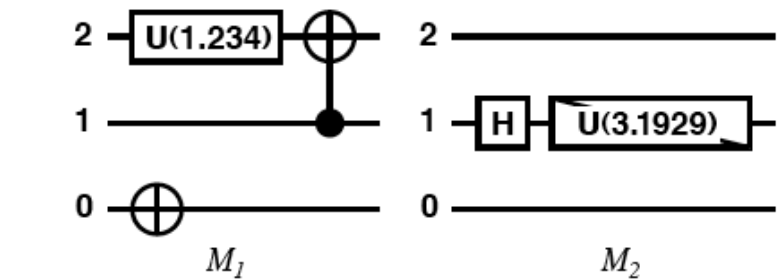
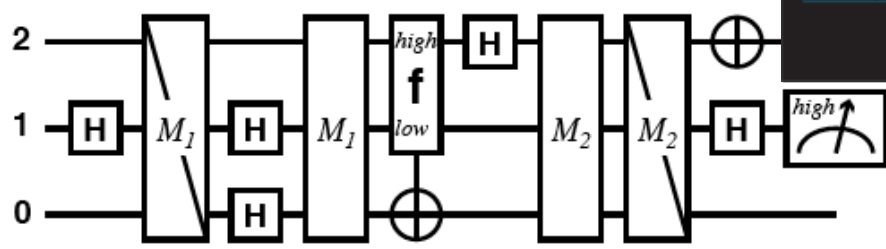
(c)



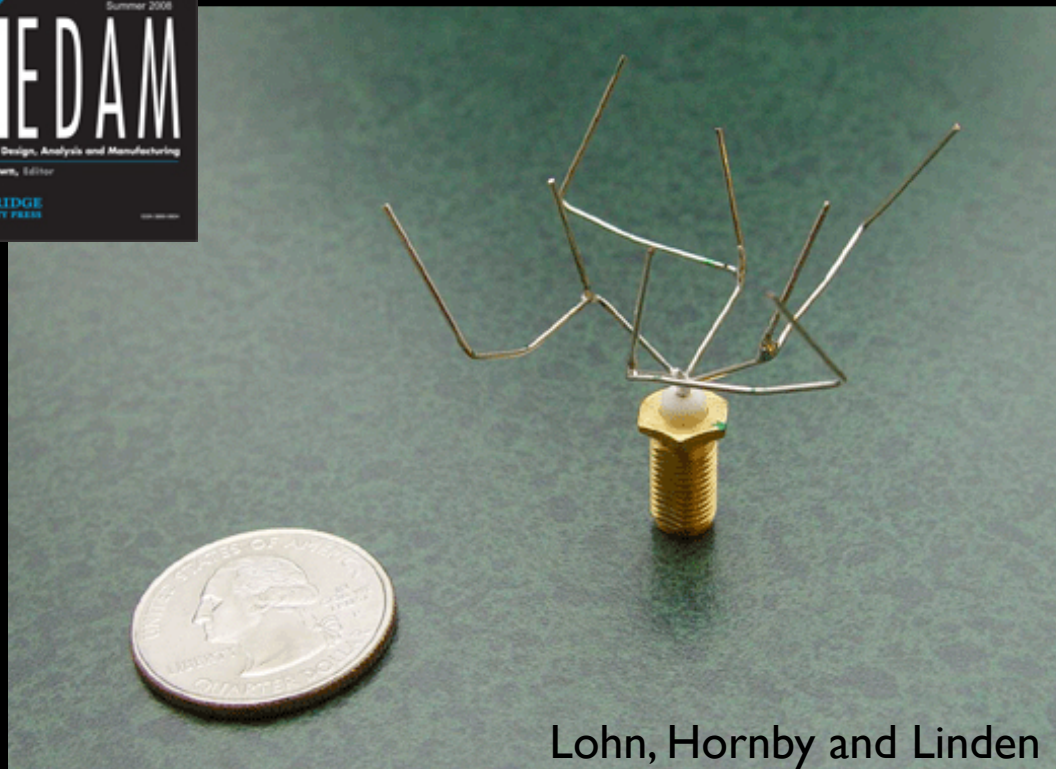
(e)

(f)

(g)

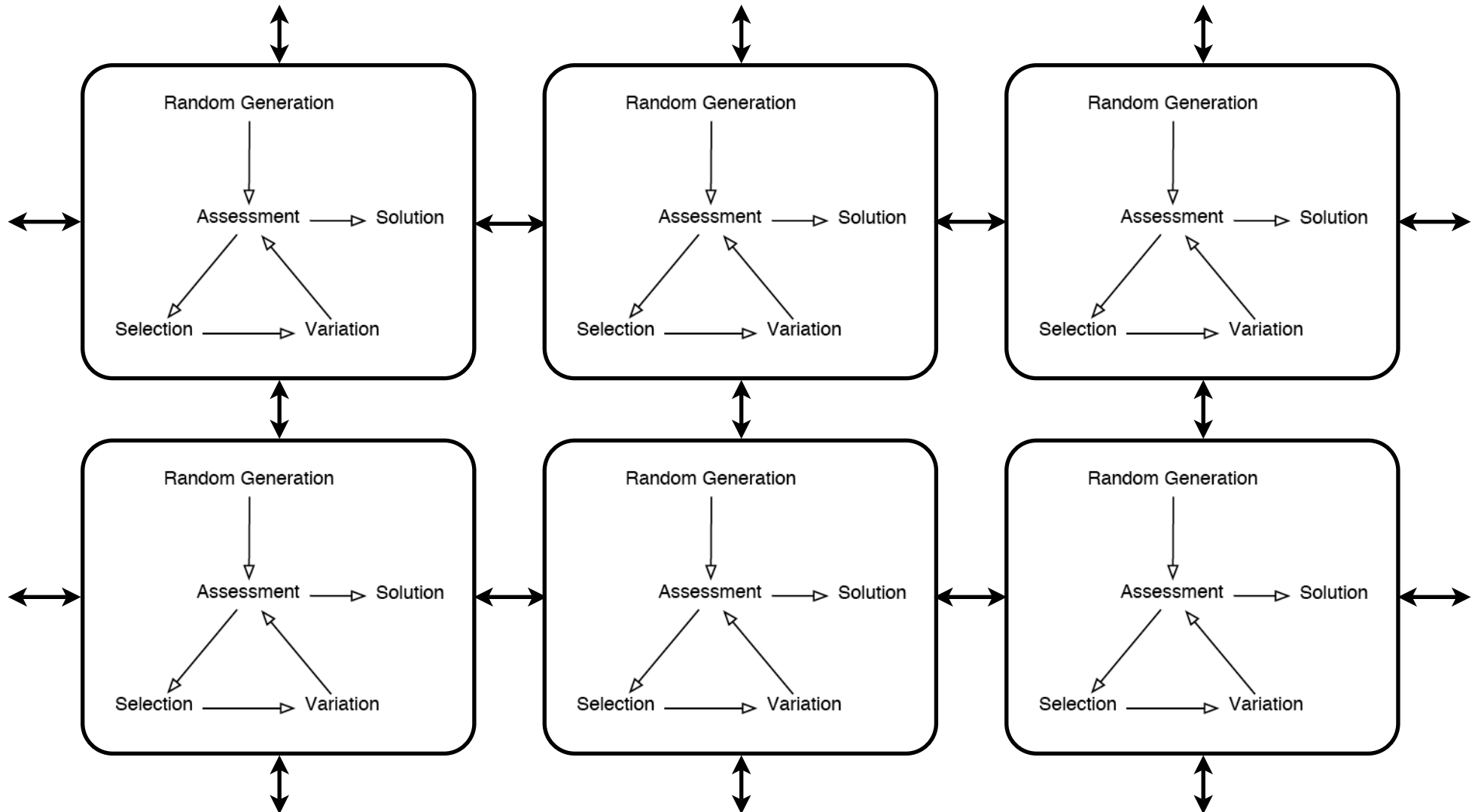


Spector



Lohn, Hornby and Linden

# Islands and Migration





# 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, ...).

# 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

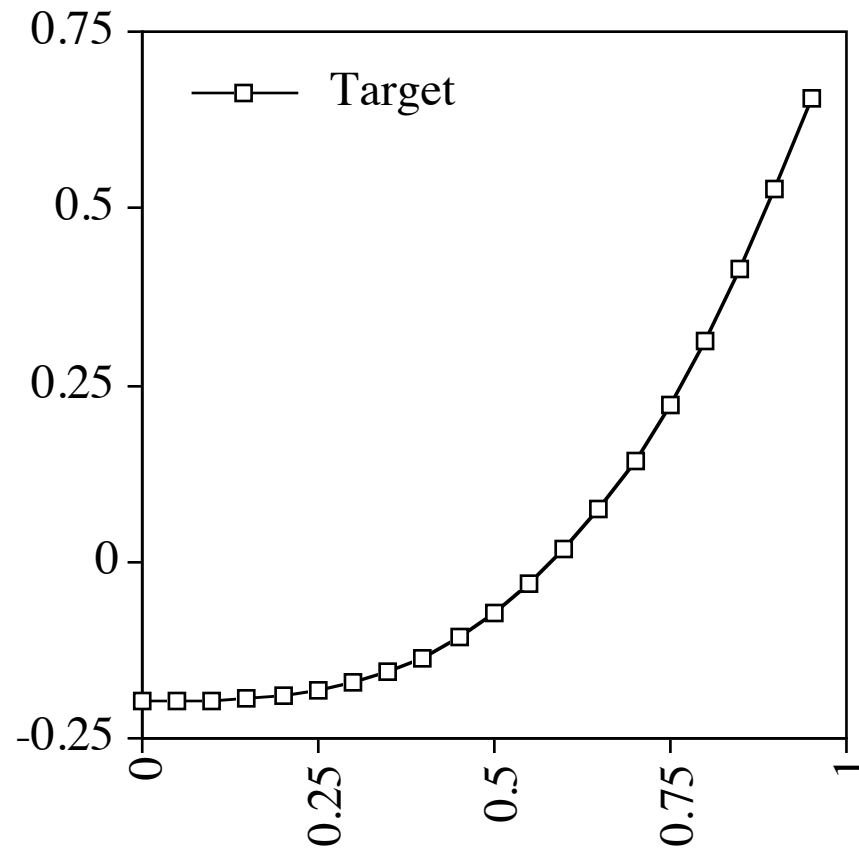
Crossover at function points fraction: 0.5

Selection method: FITNESS-PROPORTIONATE

Generation method: RAMPED-HALF-AND-HALF

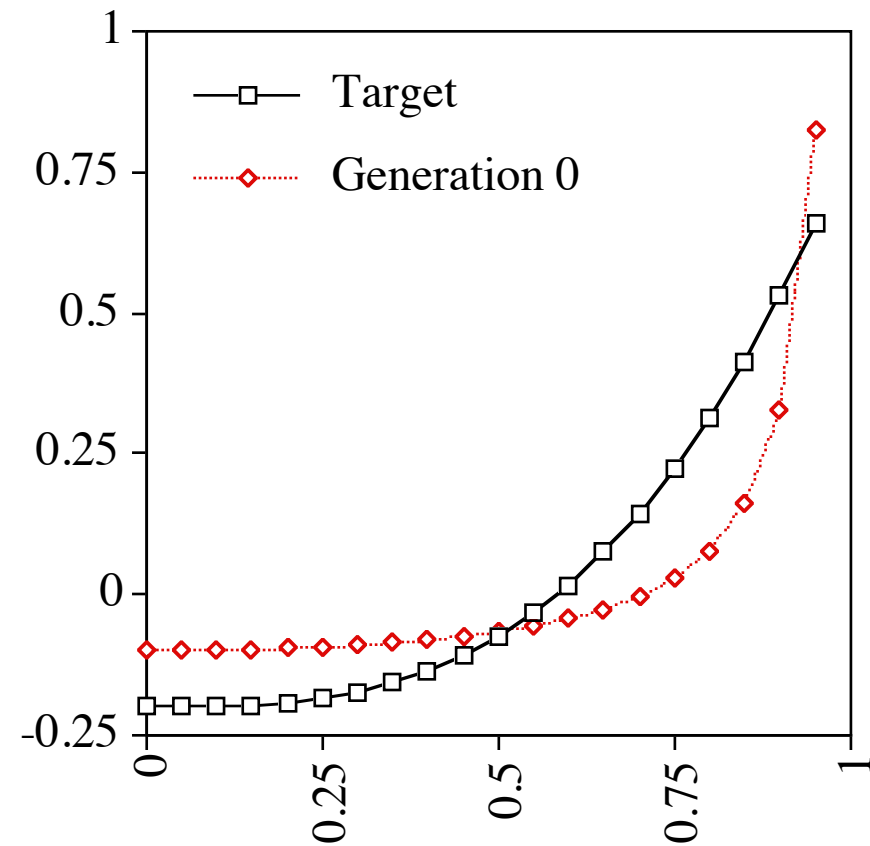
Randomizer seed: 1.2

# Evolving $y = x^3 - 0.2$



# Best Program, Gen 0

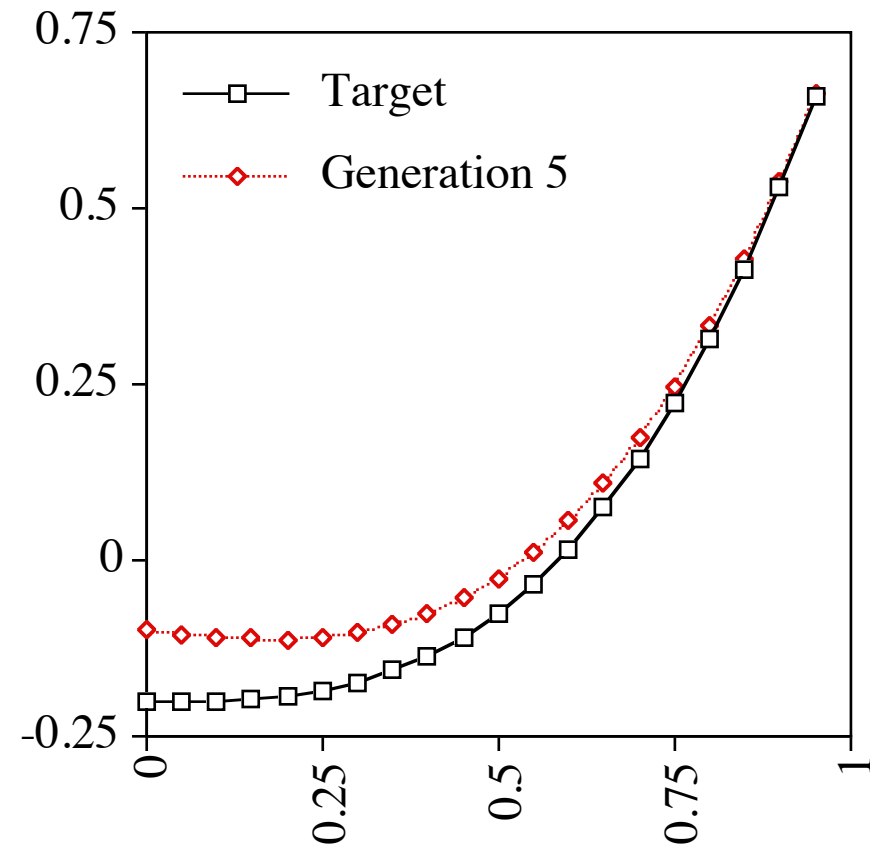
```
(- (% (* 0.1
      (* X X) )
  (- (% 0.1 0.1)
      (* X X) ) )
0.1)
```





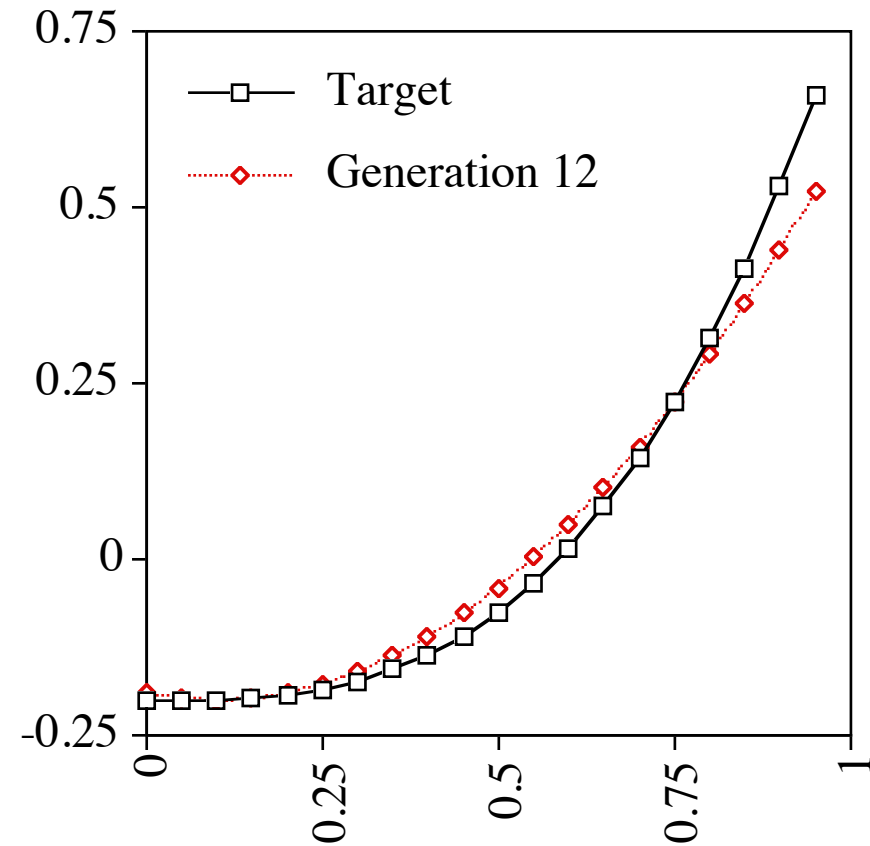
# Best Program, Gen 5

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



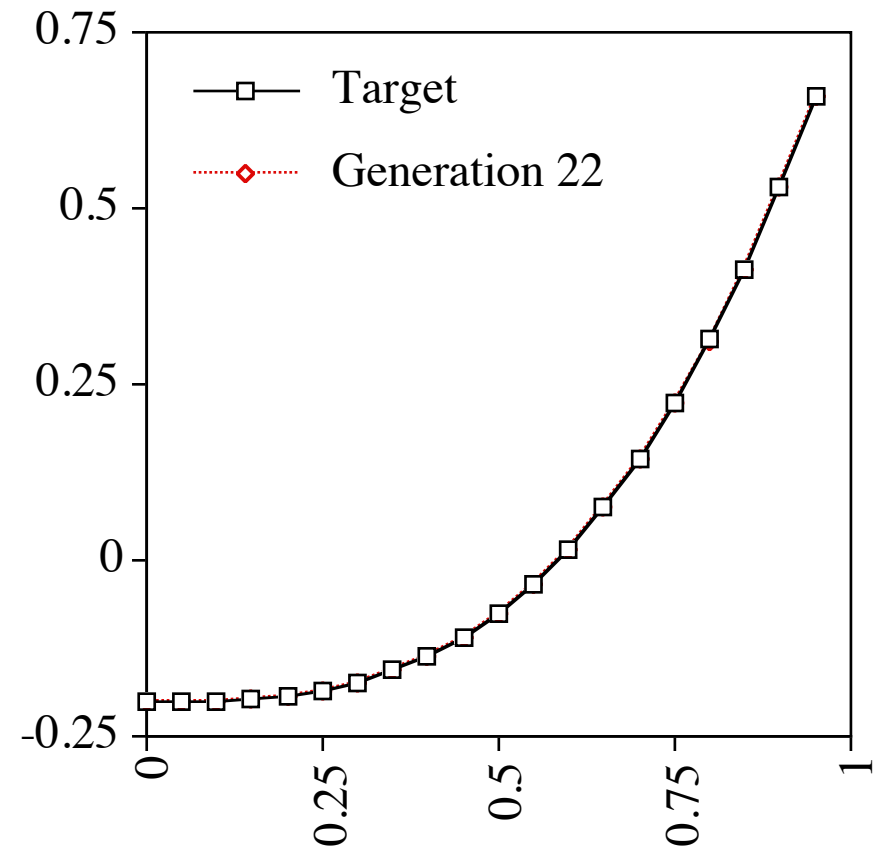
# Best Program, Gen 12

```
(+ (- (- 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

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

# 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

$\begin{array}{c ccc} \mathbf{A}_1 * & 0 & 1 & 2 \\ \hline 0 & 2 & 1 & 2 \\ 1 & 1 & 0 & 0 \\ 2 & 0 & 0 & 1 \end{array}$	$\begin{array}{c ccc} \mathbf{A}_2 * & 0 & 1 & 2 \\ \hline 0 & 2 & 0 & 2 \\ 1 & 1 & 0 & 2 \\ 2 & 1 & 2 & 1 \end{array}$
$\begin{array}{c ccc} \mathbf{A}_3 * & 0 & 1 & 2 \\ \hline 0 & 1 & 0 & 1 \\ 1 & 1 & 2 & 0 \\ 2 & 0 & 0 & 0 \end{array}$	$\begin{array}{c ccc} \mathbf{A}_4 * & 0 & 1 & 2 \\ \hline 0 & 1 & 0 & 1 \\ 1 & 0 & 2 & 0 \\ 2 & 0 & 1 & 0 \end{array}$
$\begin{array}{c ccc} \mathbf{A}_5 * & 0 & 1 & 2 \\ \hline 0 & 1 & 0 & 2 \\ 1 & 1 & 2 & 0 \\ 2 & 0 & 1 & 0 \end{array}$	$\begin{array}{c cccc} \mathbf{B}_1 * & 0 & 1 & 2 & 3 \\ \hline 0 & 1 & 3 & 1 & 0 \\ 1 & 3 & 2 & 0 & 1 \\ 2 & 0 & 1 & 3 & 1 \\ 3 & 1 & 0 & 2 & 0 \end{array}$



# 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  $B_1$ :

$$\begin{aligned} & (((((((((x*(y*x))*x)*z)*(z*x))*((x*(z*(x*(z*y)))))*z))*z) \\ & *z)*(z*((((x*((z*z)*x)*(z*x))))*x)*y)*(((y*(z*(z*y))))* \\ & (((y*y)*x)*z))*((x*((z*z)*x)*(z*(x*(z*y)))))) \end{aligned}$$

# Significance, Time

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

# Significance, Time

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

# 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  $A_1$ )

# 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

# 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



# 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 (Kinneer, Roberts, many others).
- Push approach: instructions that can build/execute modules with no changes to the system's representations or algorithms.

# 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
- Types include: integer, float, Boolean, name, **code**, **exec**, vector, matrix, quantum gate, [add more as needed]
- Missing argument? NOOP
- Trivial syntax:  
program  $\rightarrow$  instruction | literal | ( program\* )

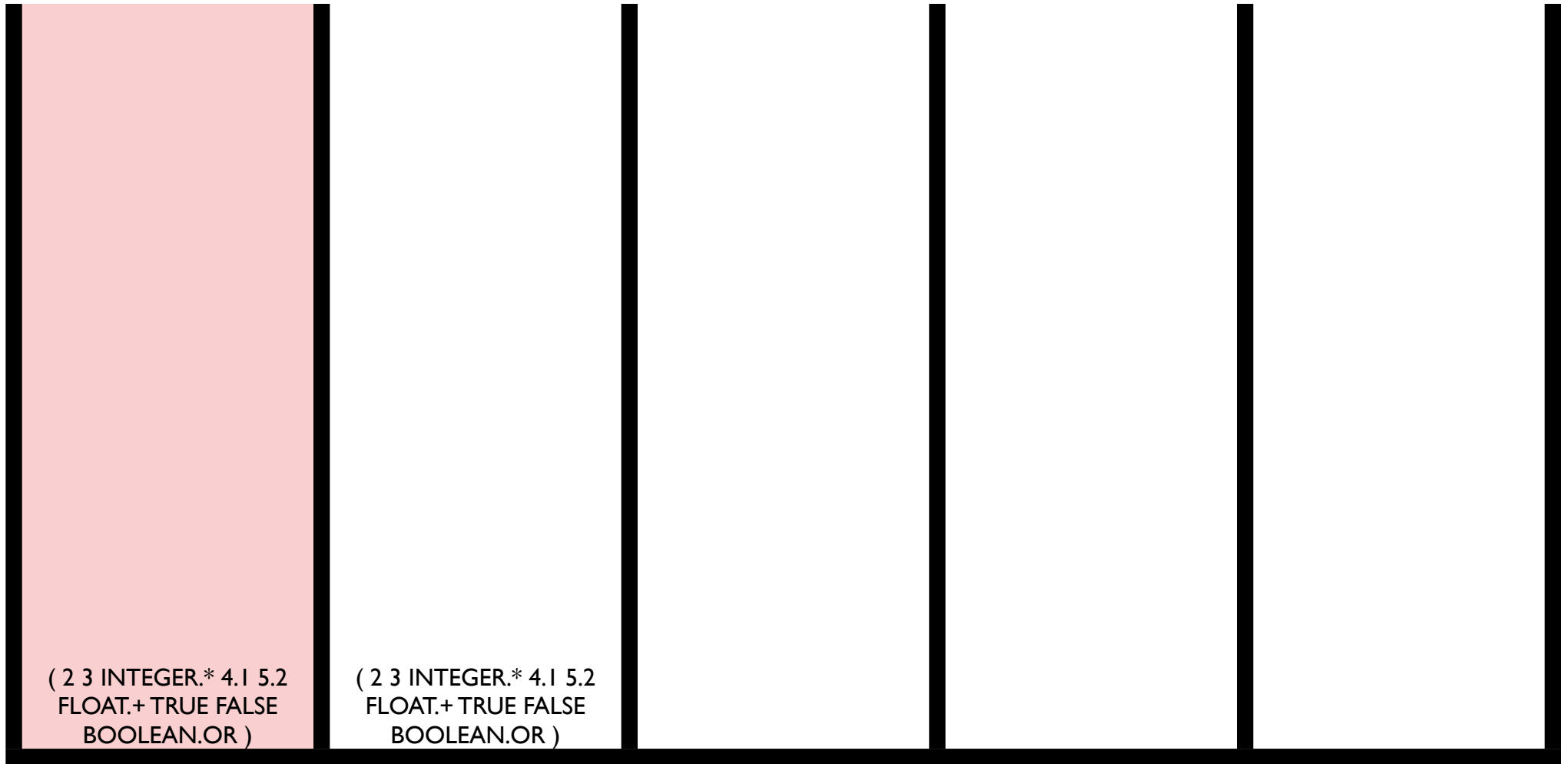
# Sample Push Instructions

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 )
```



**exec**

**code**

**bool**

**int**

**float**

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**

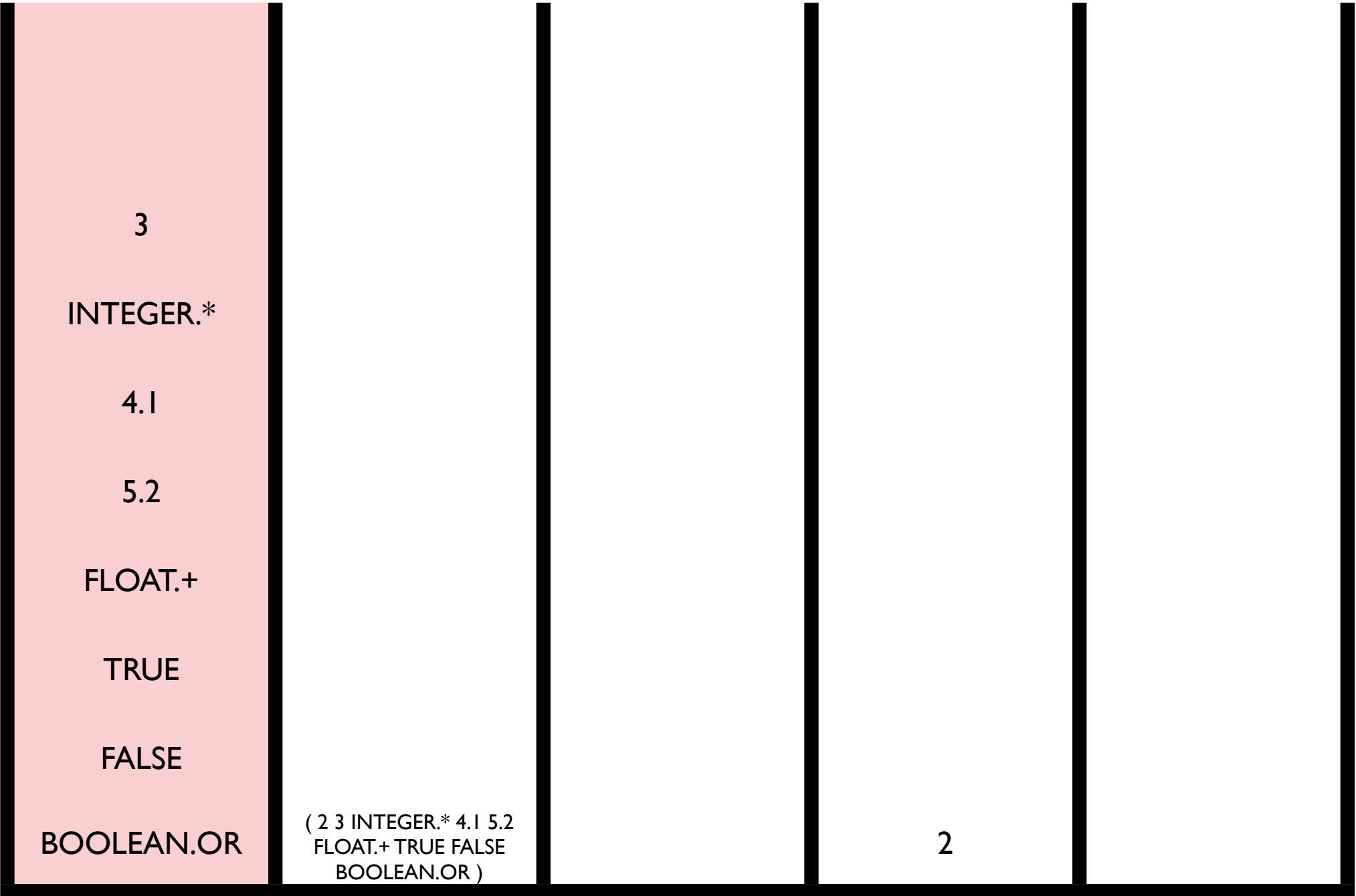
**code**

**bool**

**int**

**float**





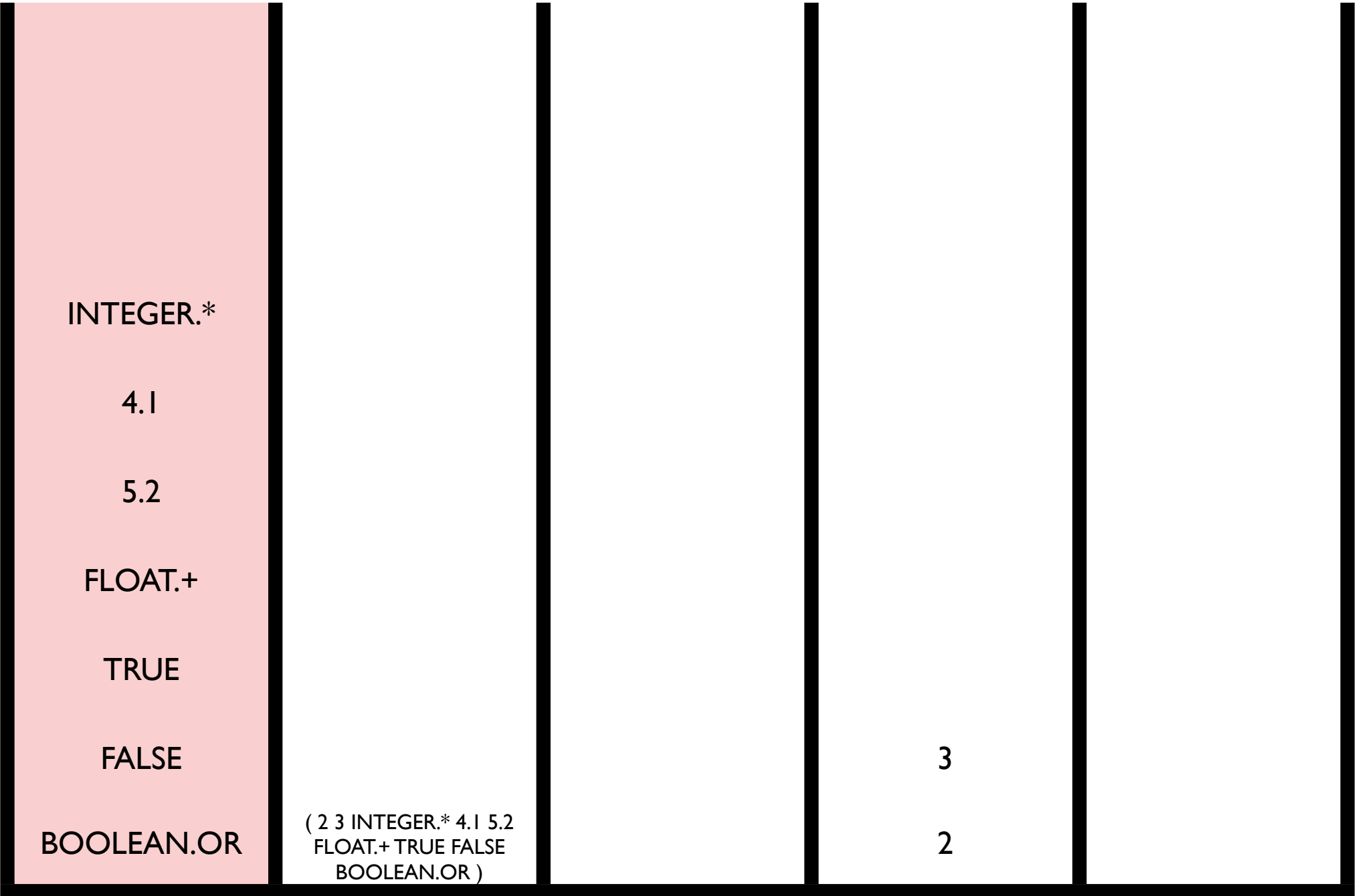
**exec**

**code**

**bool**

**int**

**float**



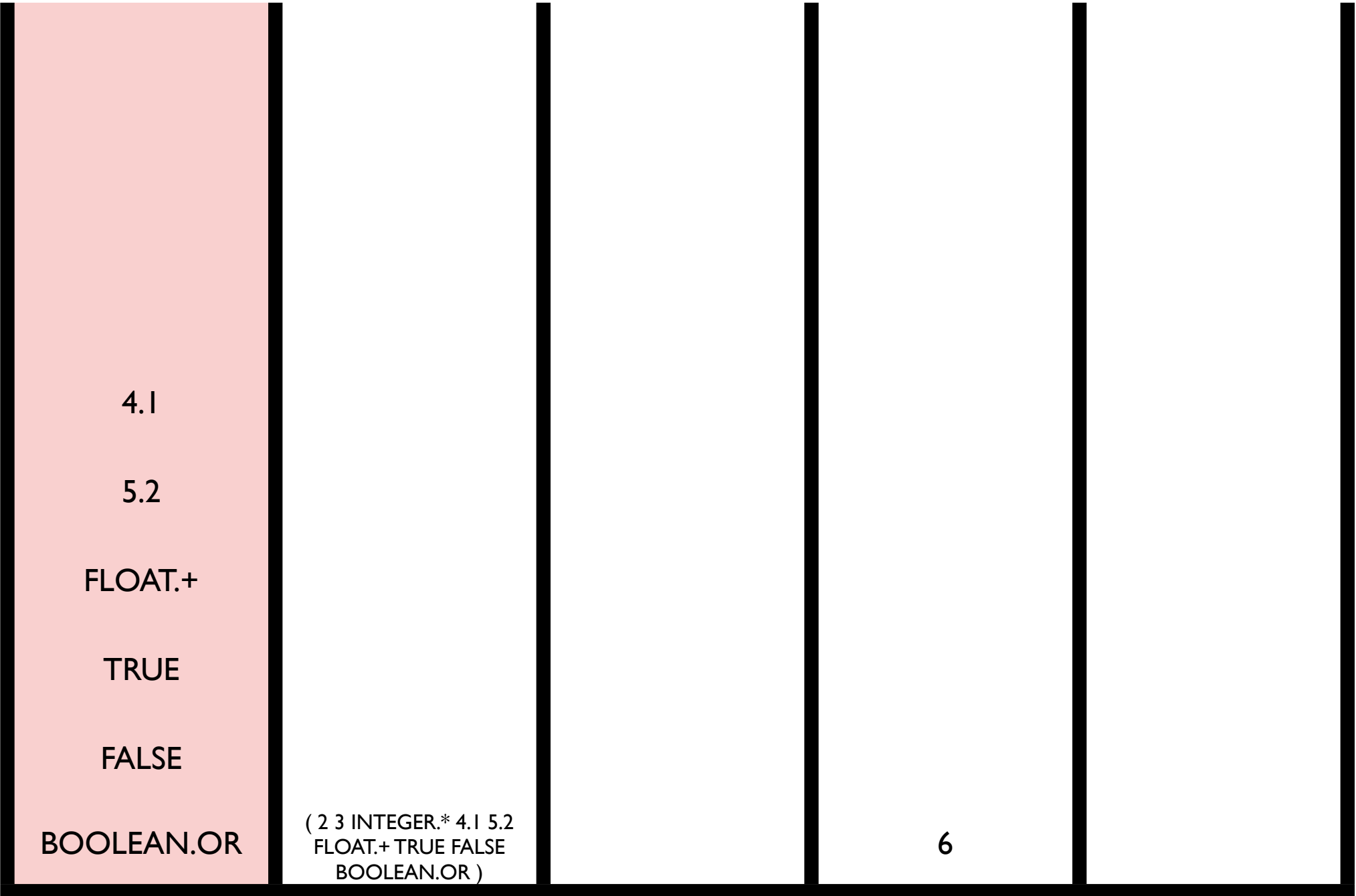
**exec**

**code**

**bool**

**int**

**float**



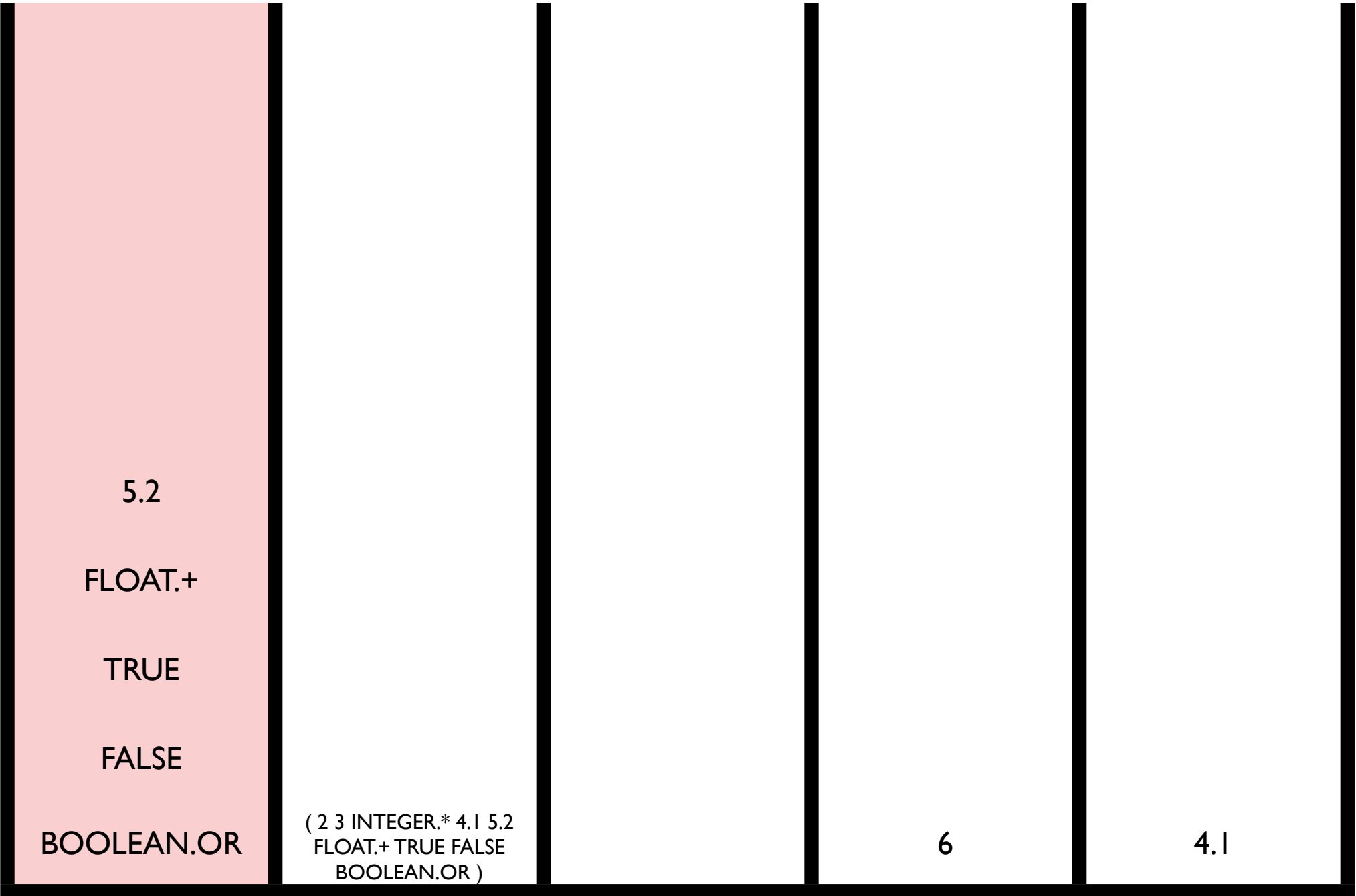
**exec**

**code**

**bool**

**int**

**float**



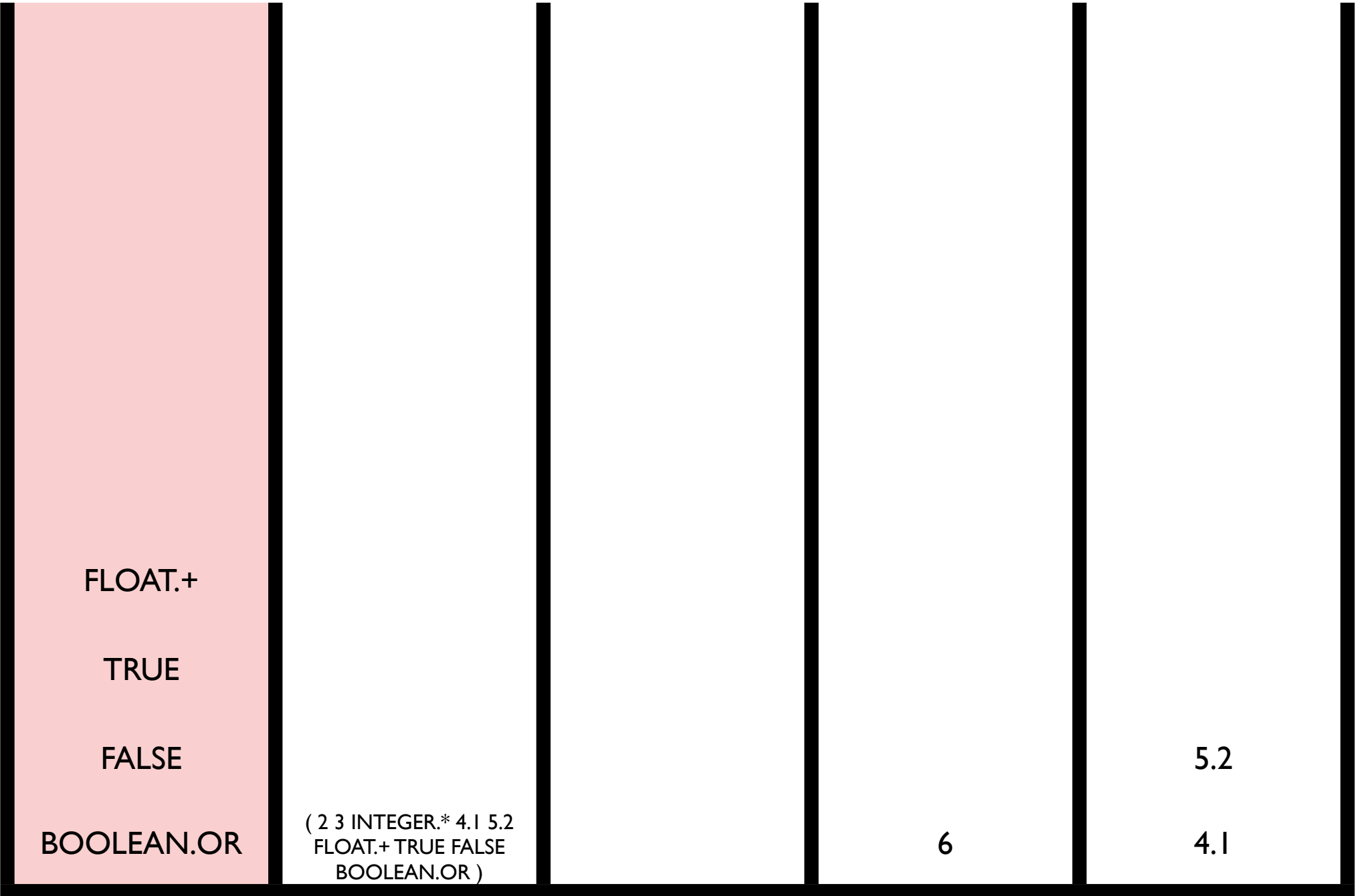
**exec**

**code**

**bool**

**int**

**float**



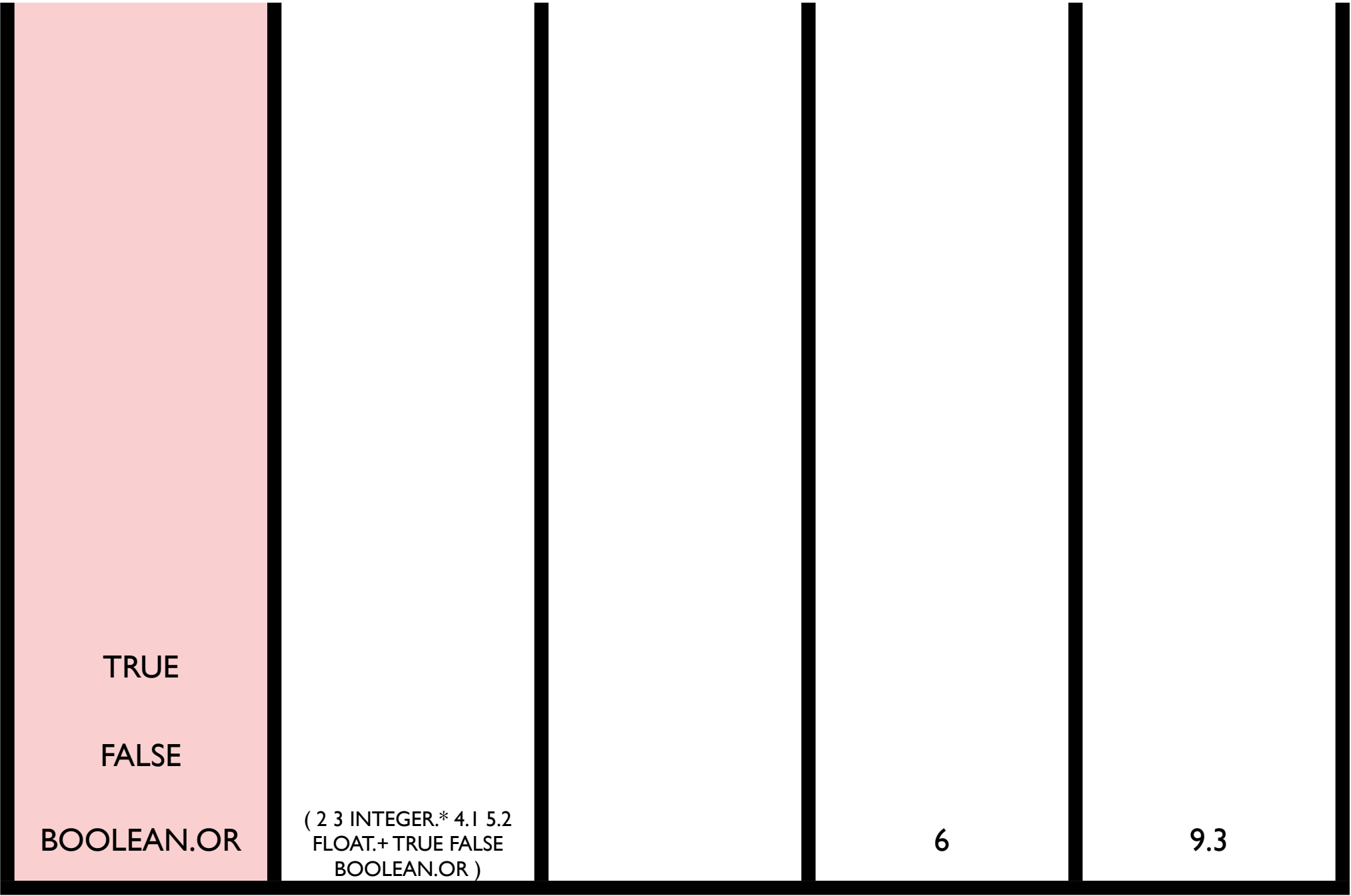
**exec**

**code**

**bool**

**int**

**float**



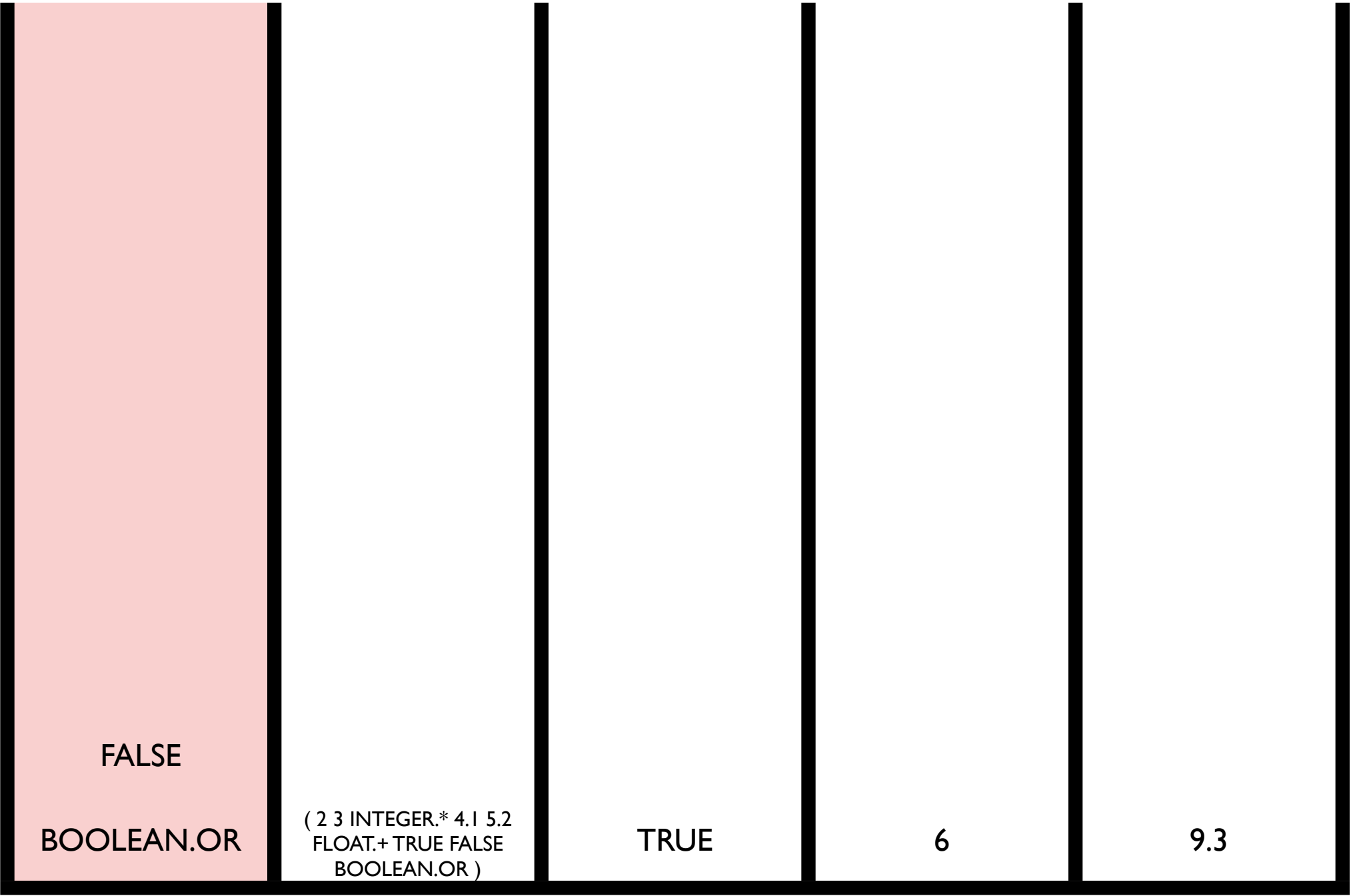
**exec**

**code**

**bool**

**int**

**float**



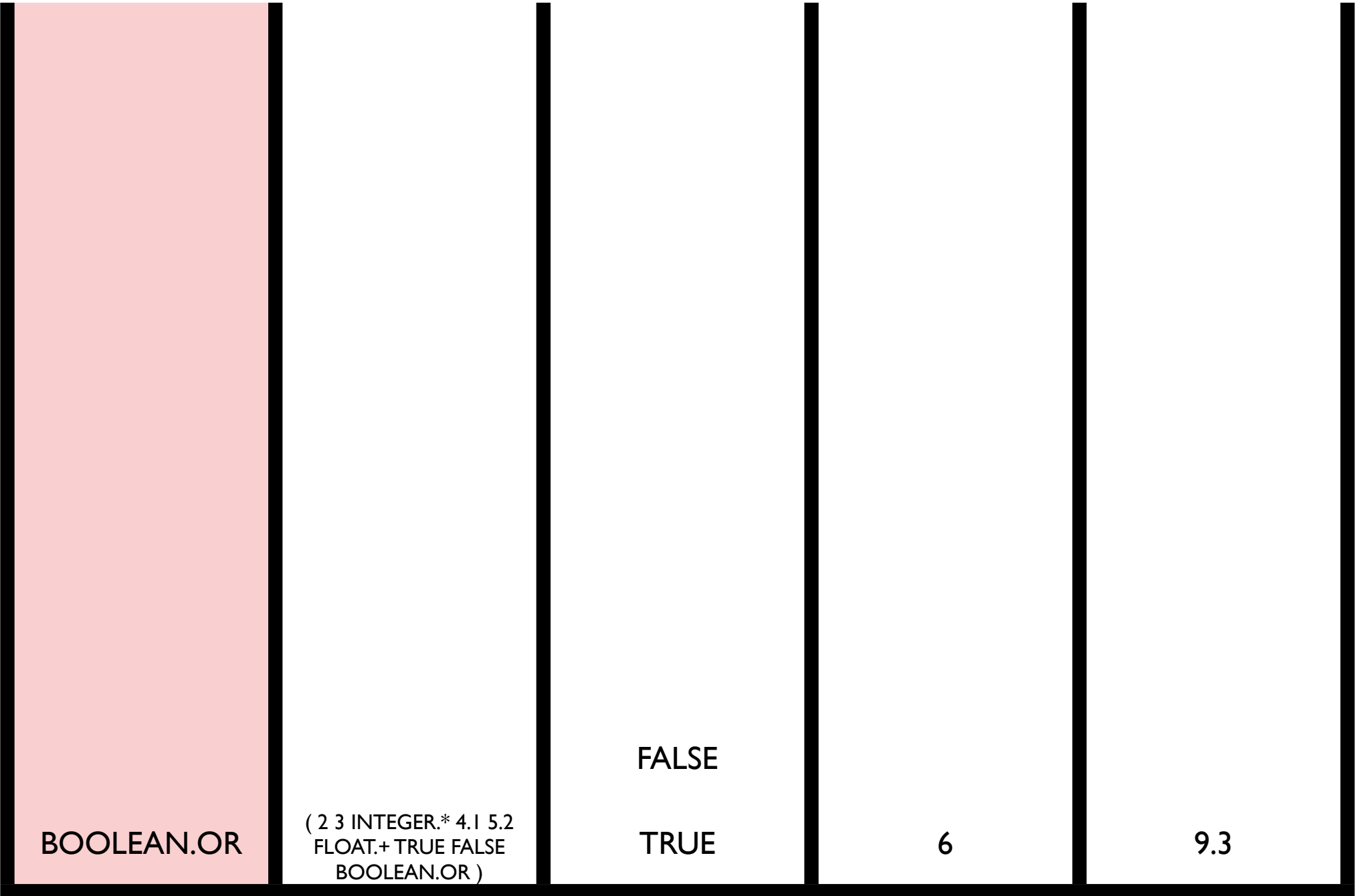
**exec**

**code**

**bool**

**int**

**float**



**exec**

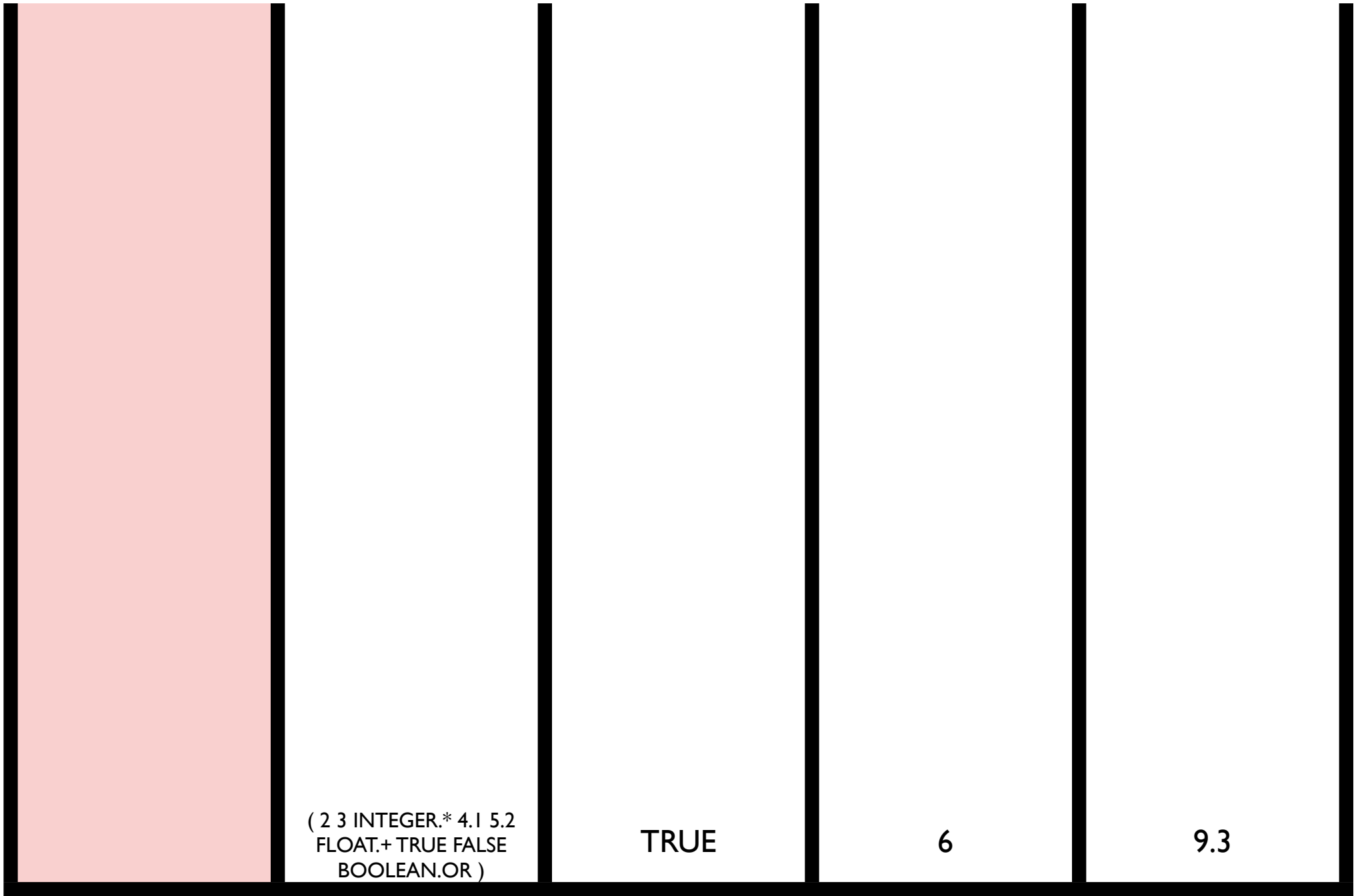
**code**

**bool**

**int**

**float**





**exec**

**code**

**bool**

**int**

**float**

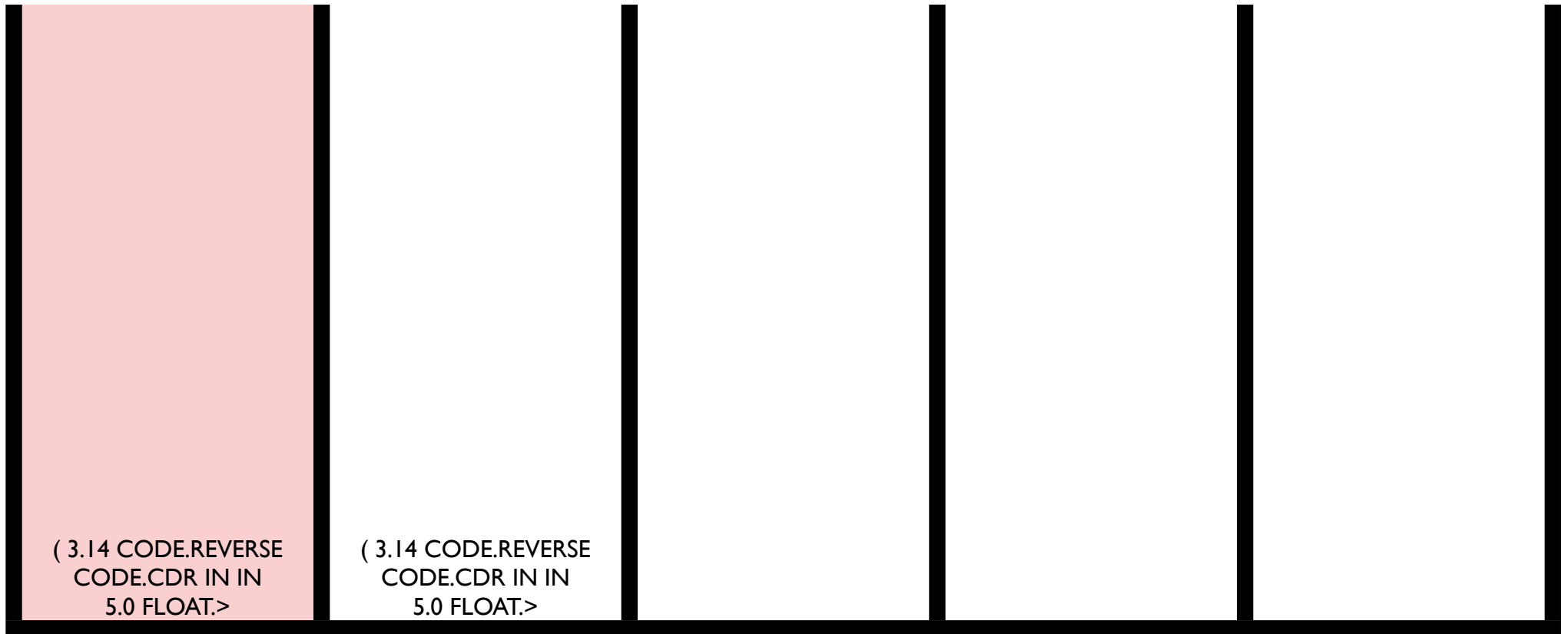
# Same Results

```
( 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



**exec**

**code**

**bool**

**int**

**float**

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.>

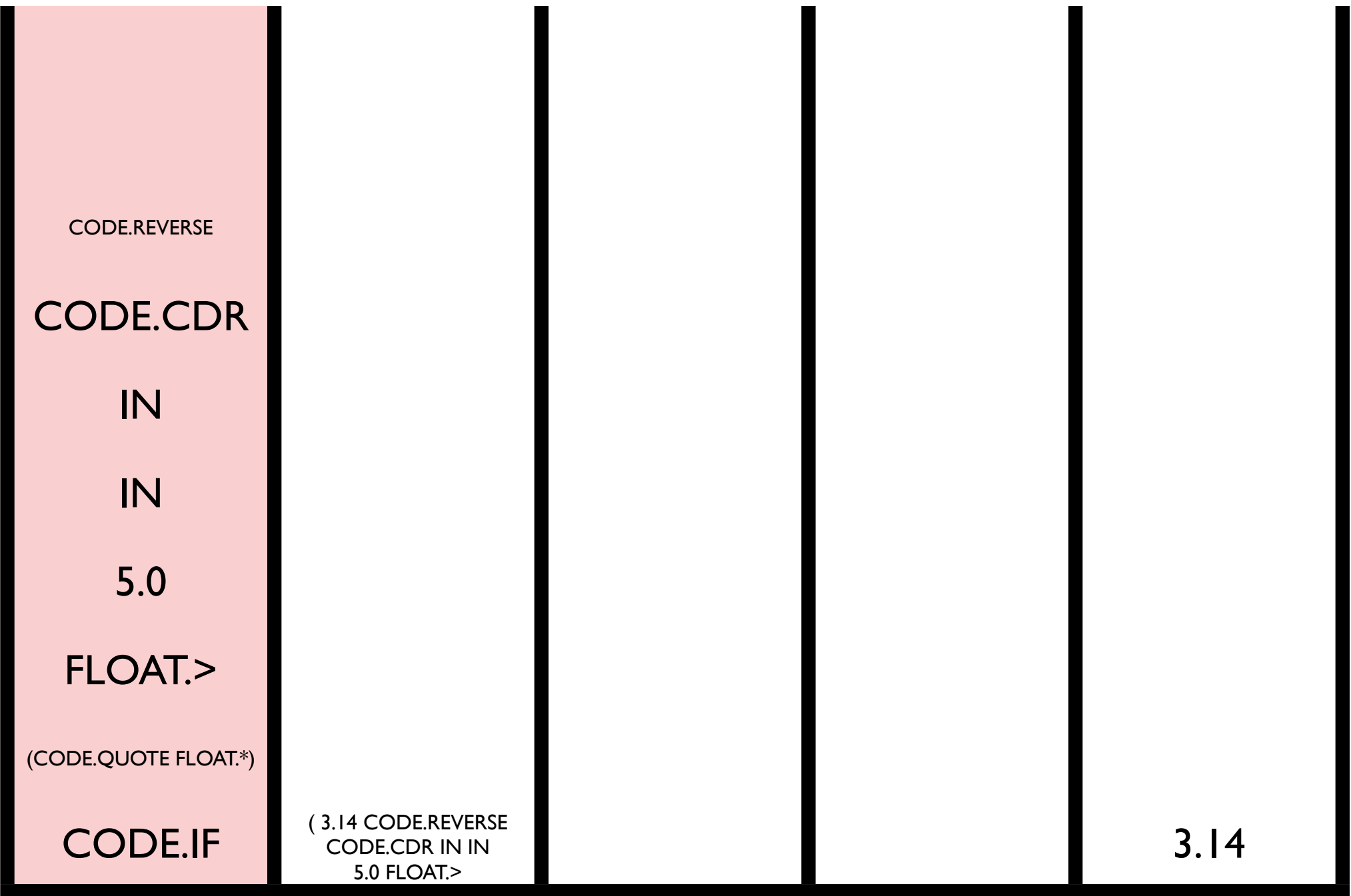
**exec**

**code**

**bool**

**int**

**float**



**exec**

**code**

**bool**

**int**

**float**

CODE.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

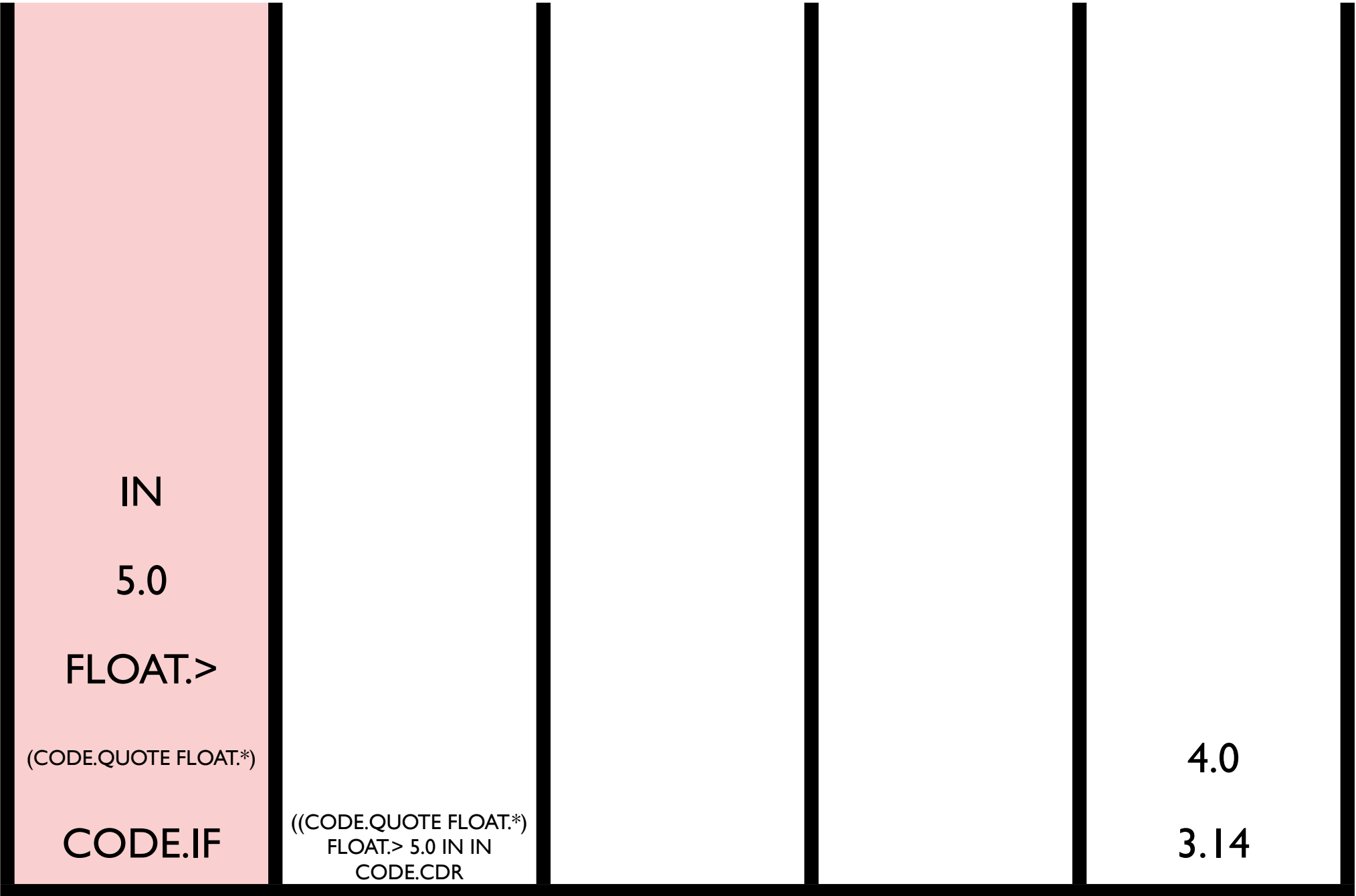
**exec**

**code**

**bool**

**int**

**float**



**exec**

**code**

**bool**

**int**

**float**



5.0

FLOAT.>

(CODE.QUOTE FLOAT.\*)

CODE.IF

((CODE.QUOTE FLOAT.\*)  
FLOAT.> 5.0 IN IN  
CODE.CDR

4.0

4.0

3.14

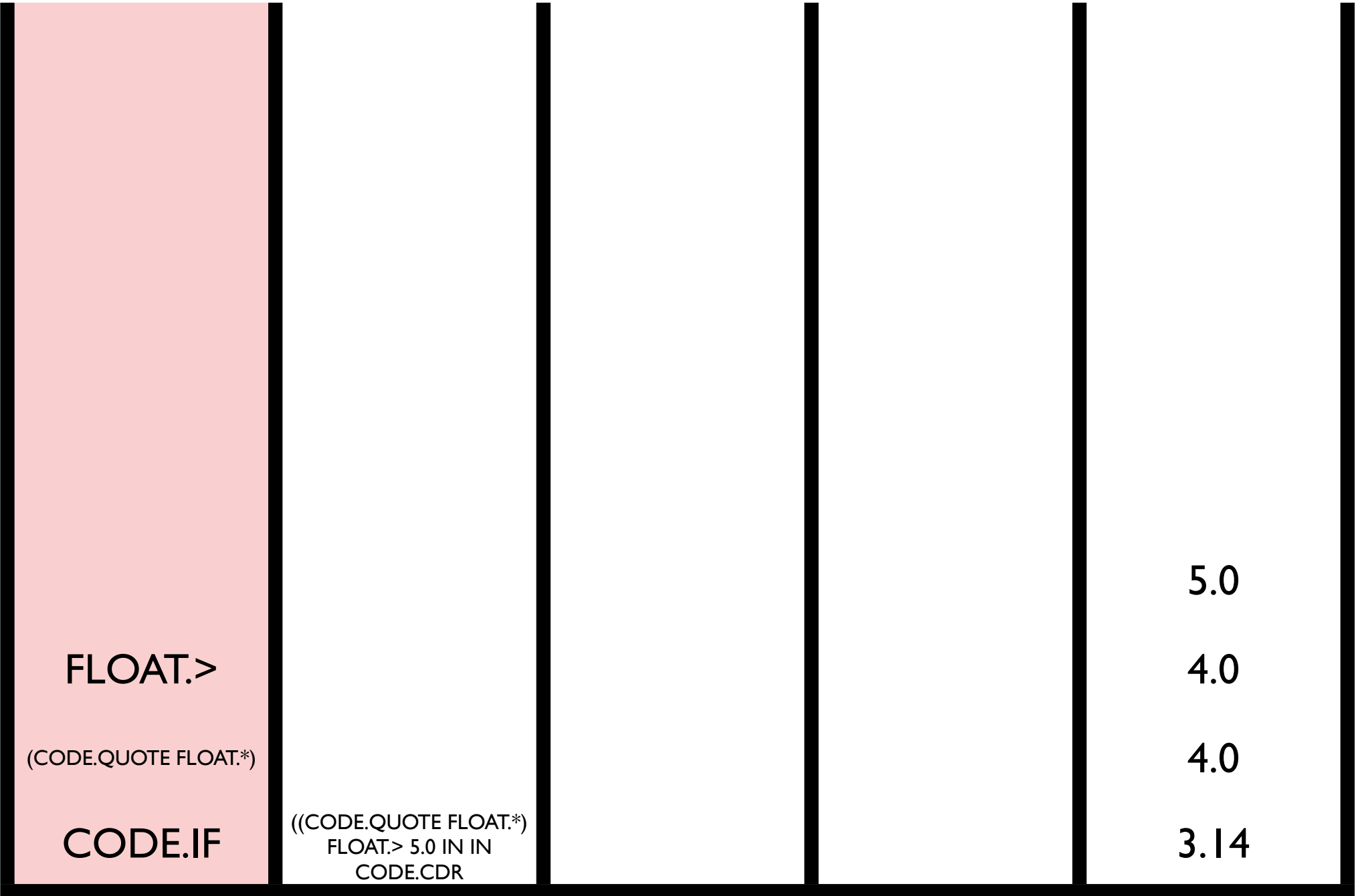
**exec**

**code**

**bool**

**int**

**float**



**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

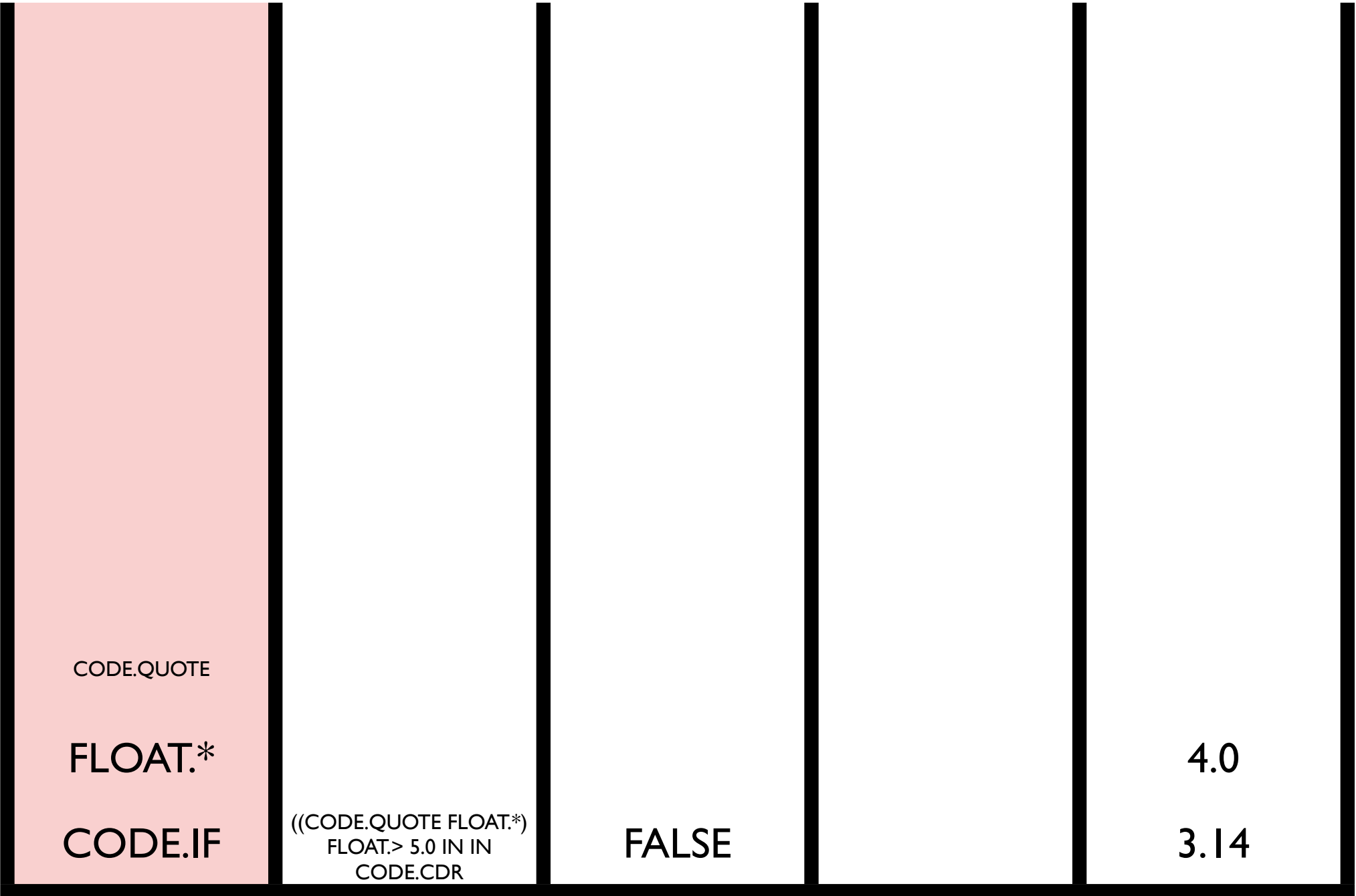
**exec**

**code**

**bool**

**int**

**float**



**exec**

**code**

**bool**

**int**

**float**

CODE.IF

FLOAT.\*  
((CODE.QUOTE FLOAT.\*)  
FLOAT.> 5.0 IN IN  
CODE.CDR

FALSE

4.0  
3.14

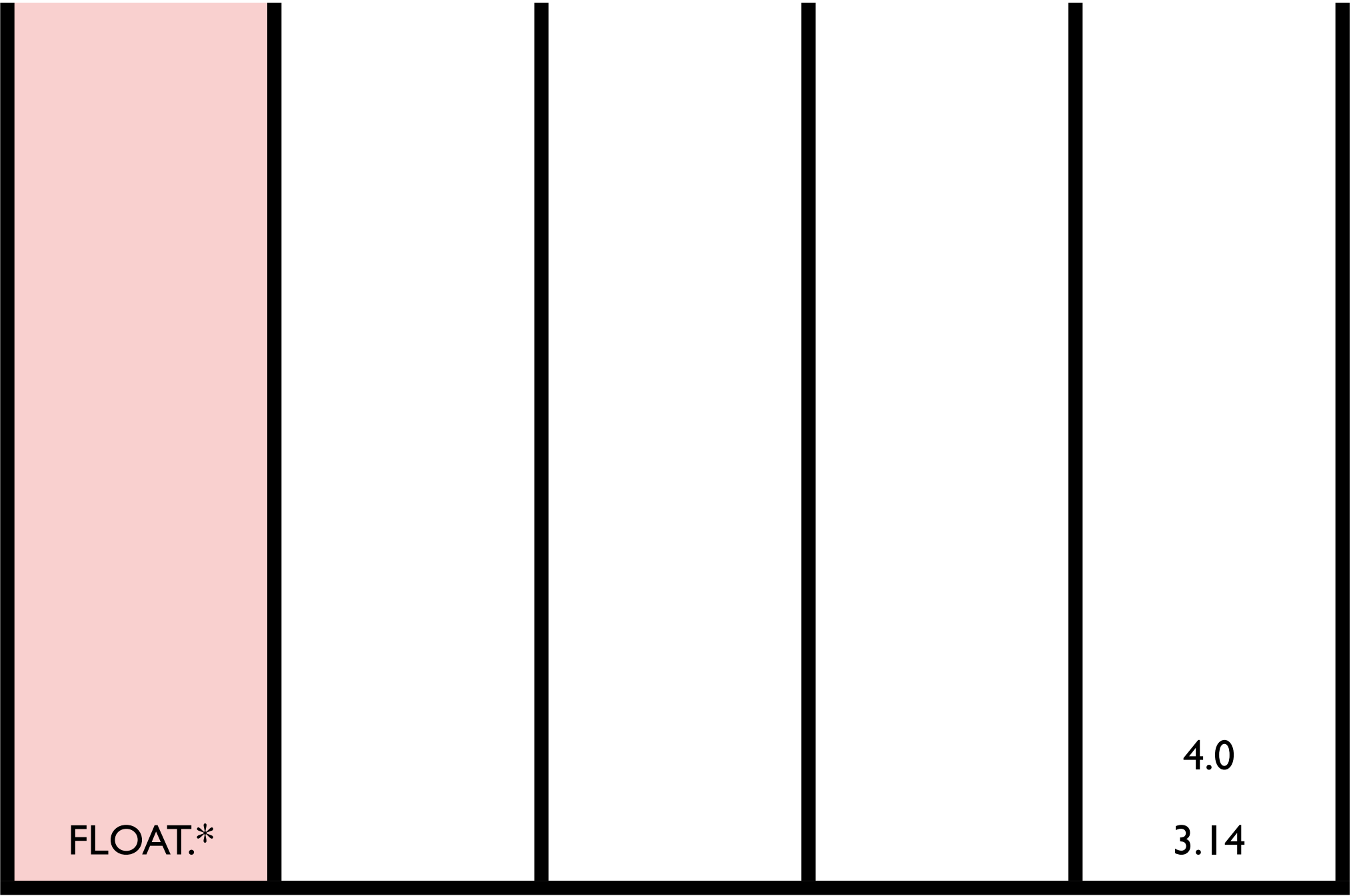
**exec**

**code**

**bool**

**int**

**float**



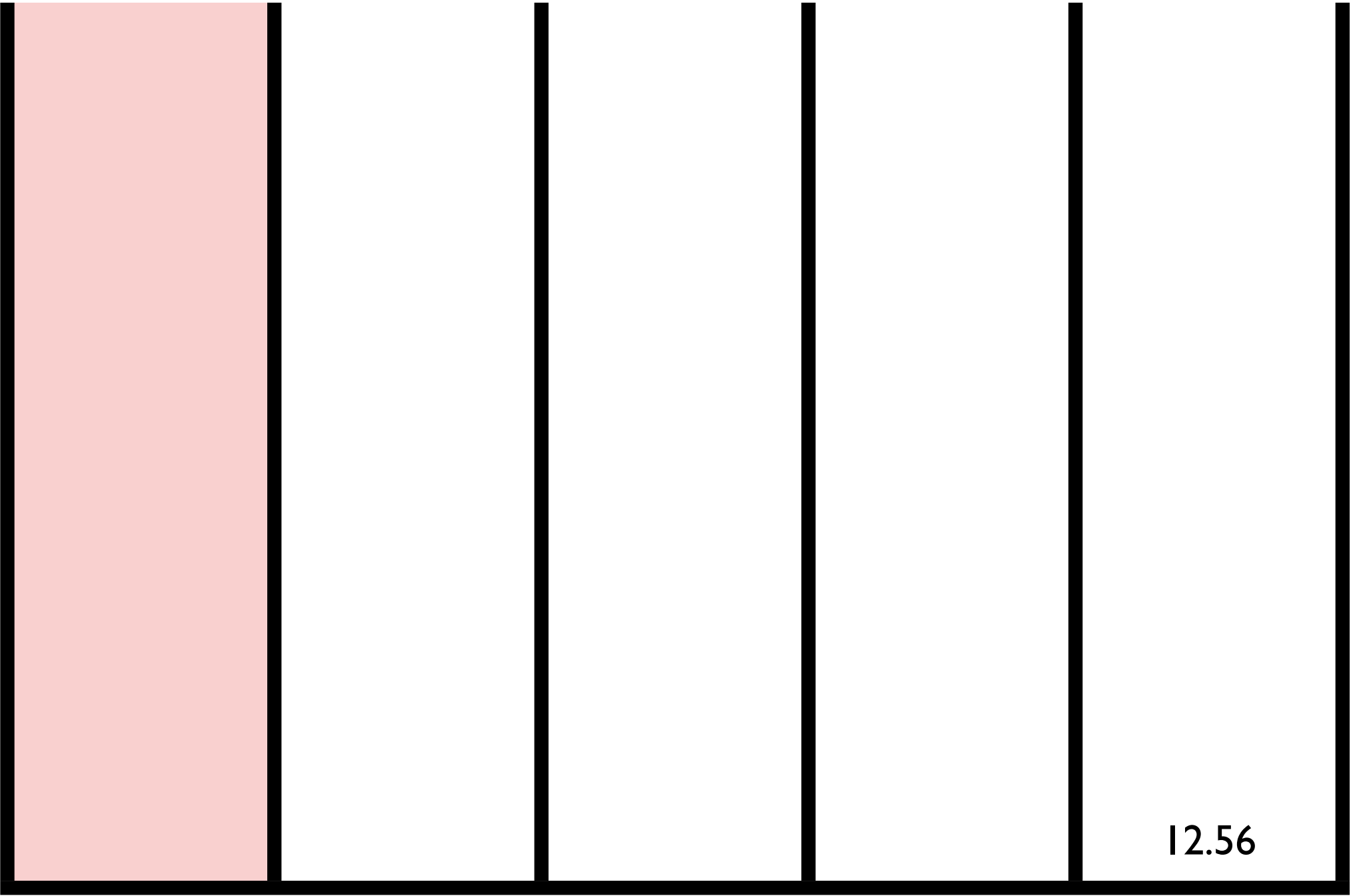
**exec**

**code**

**bool**

**int**

**float**



**exec**

**code**

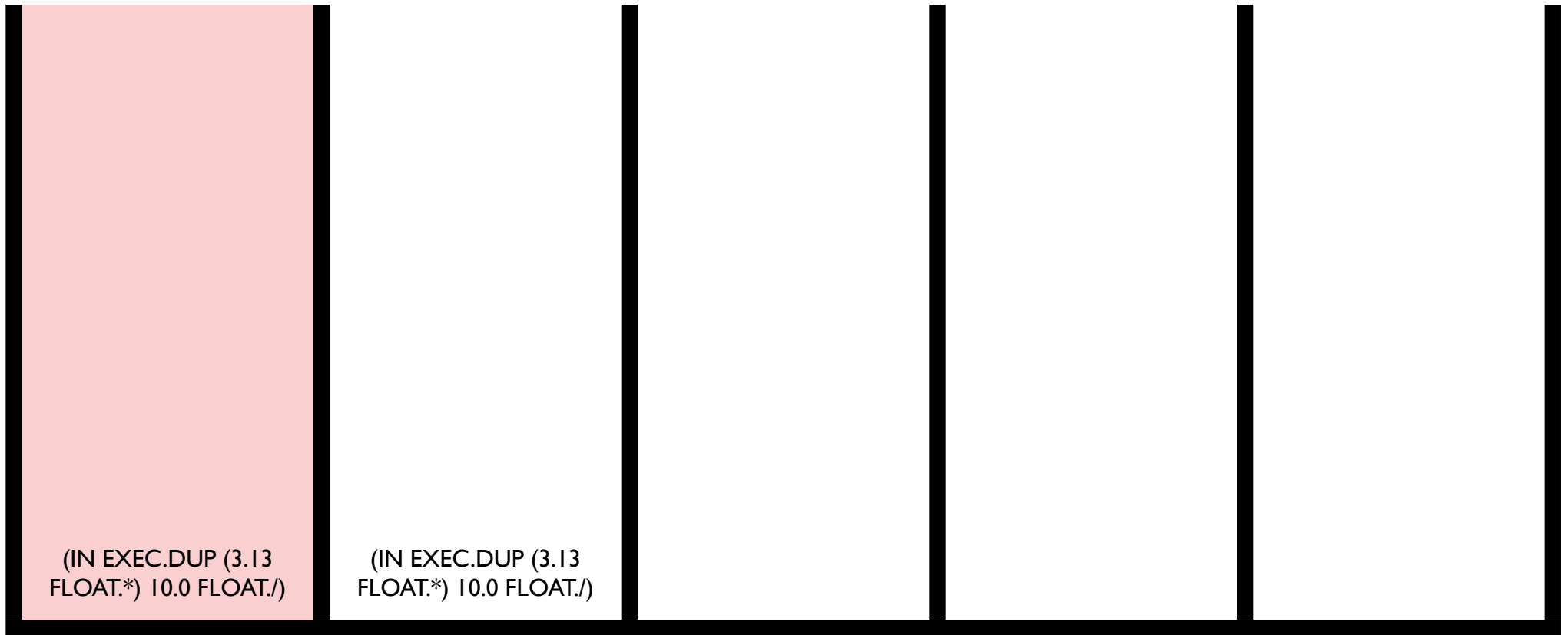
**bool**

**int**

**float**

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

IN=4.0



**exec**

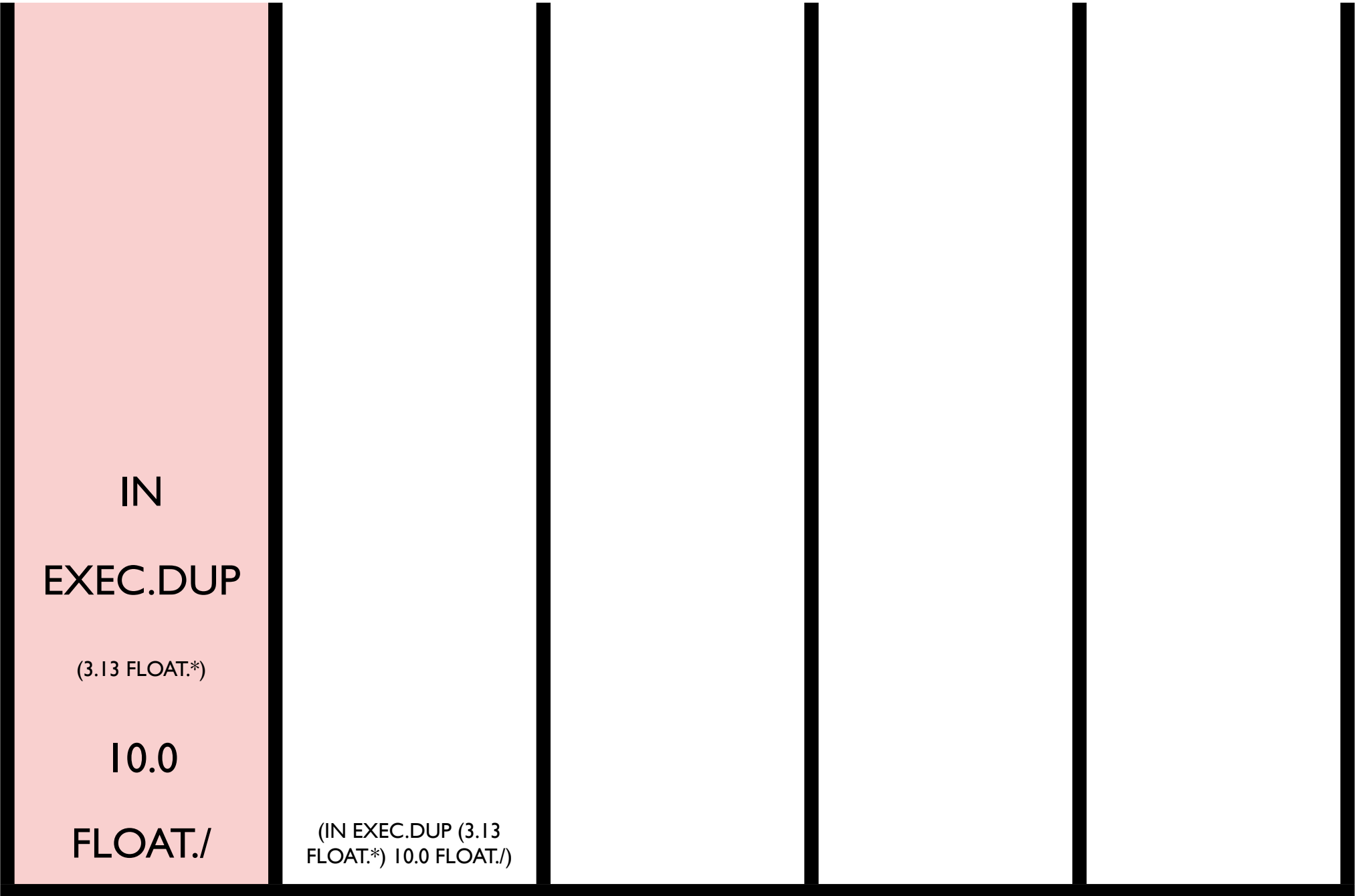
**code**

**bool**

**int**

**float**





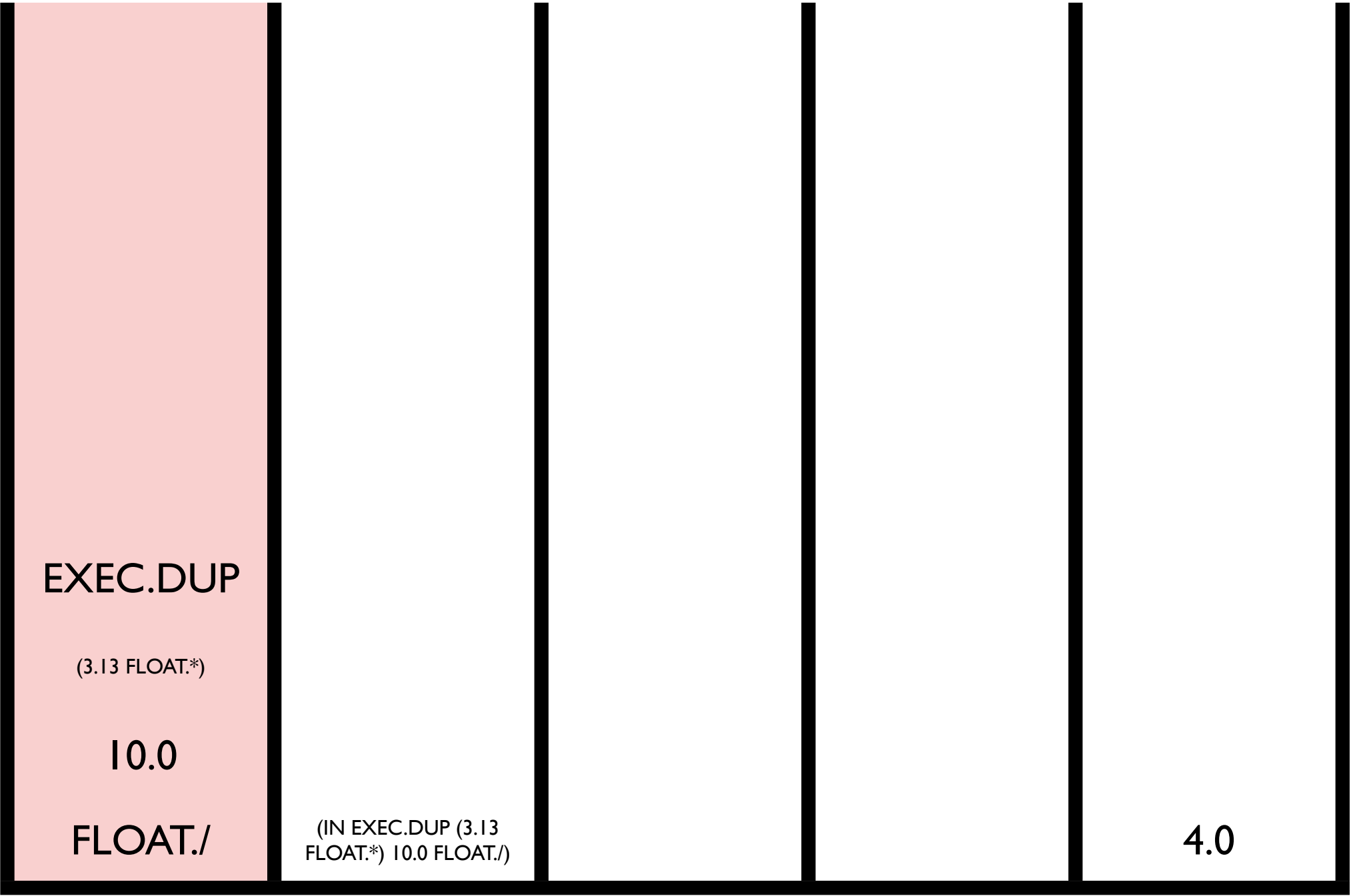
**exec**

**code**

**bool**

**int**

**float**



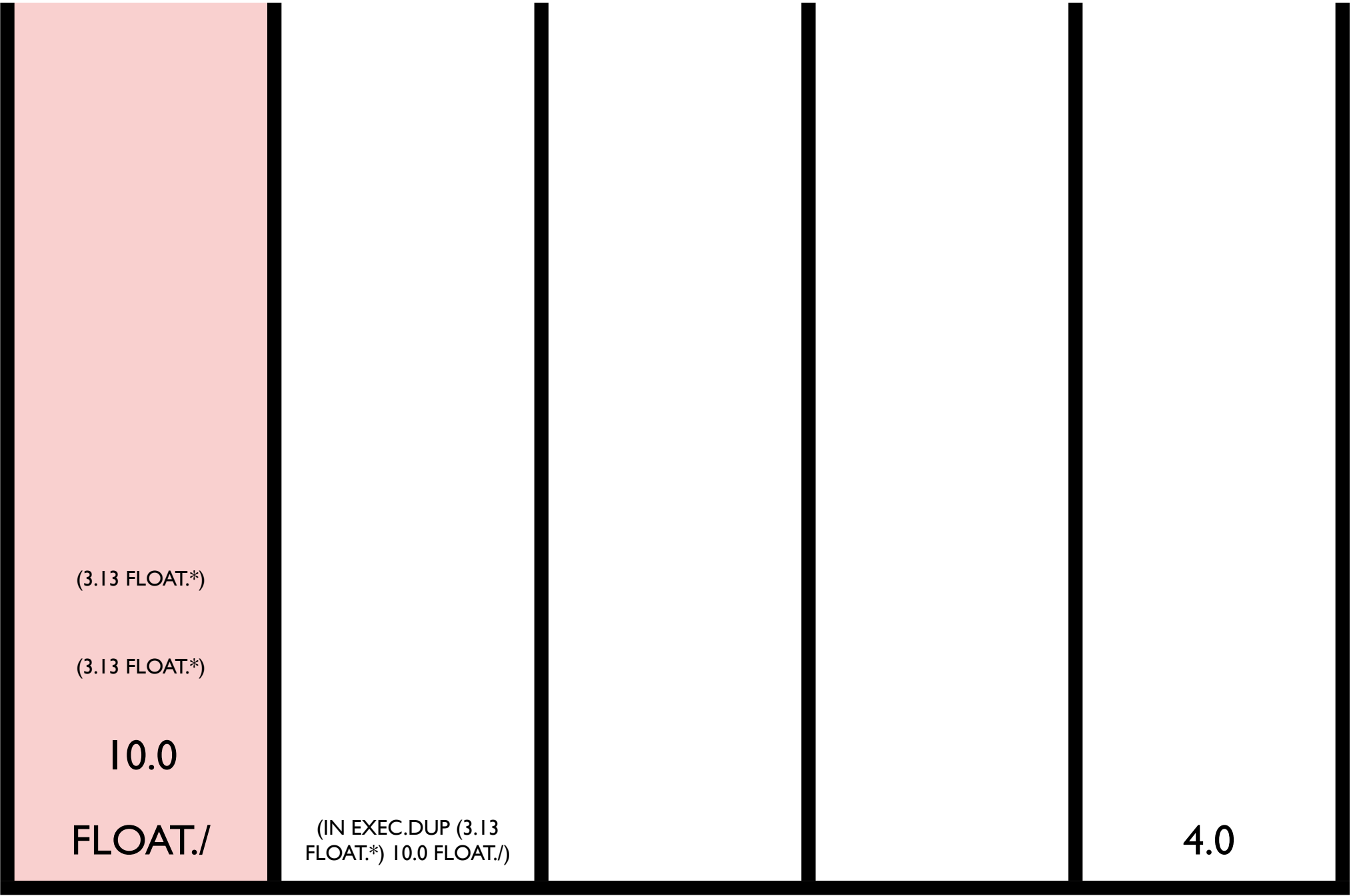
**exec**

**code**

**bool**

**int**

**float**



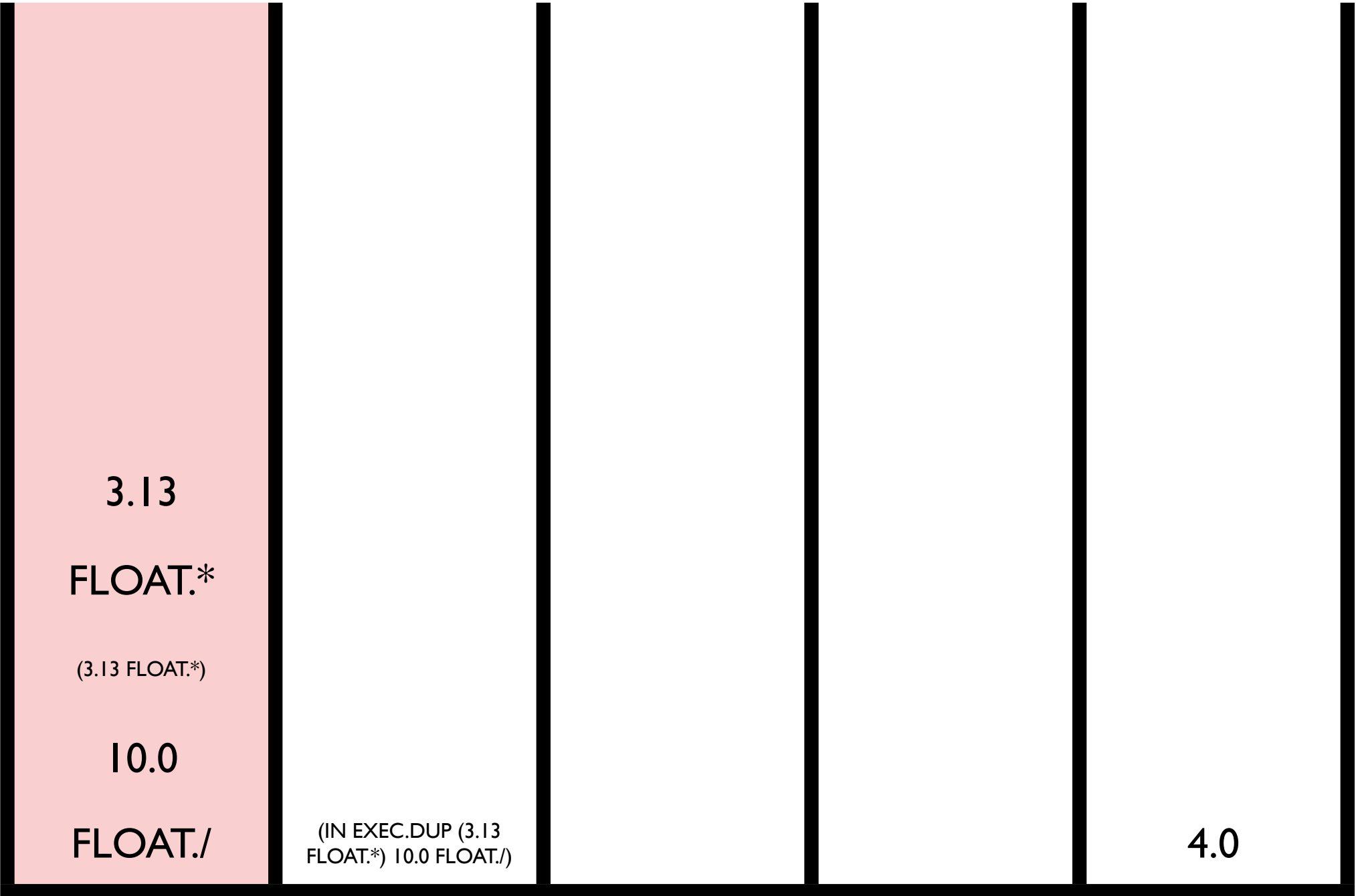
**exec**

**code**

**bool**

**int**

**float**



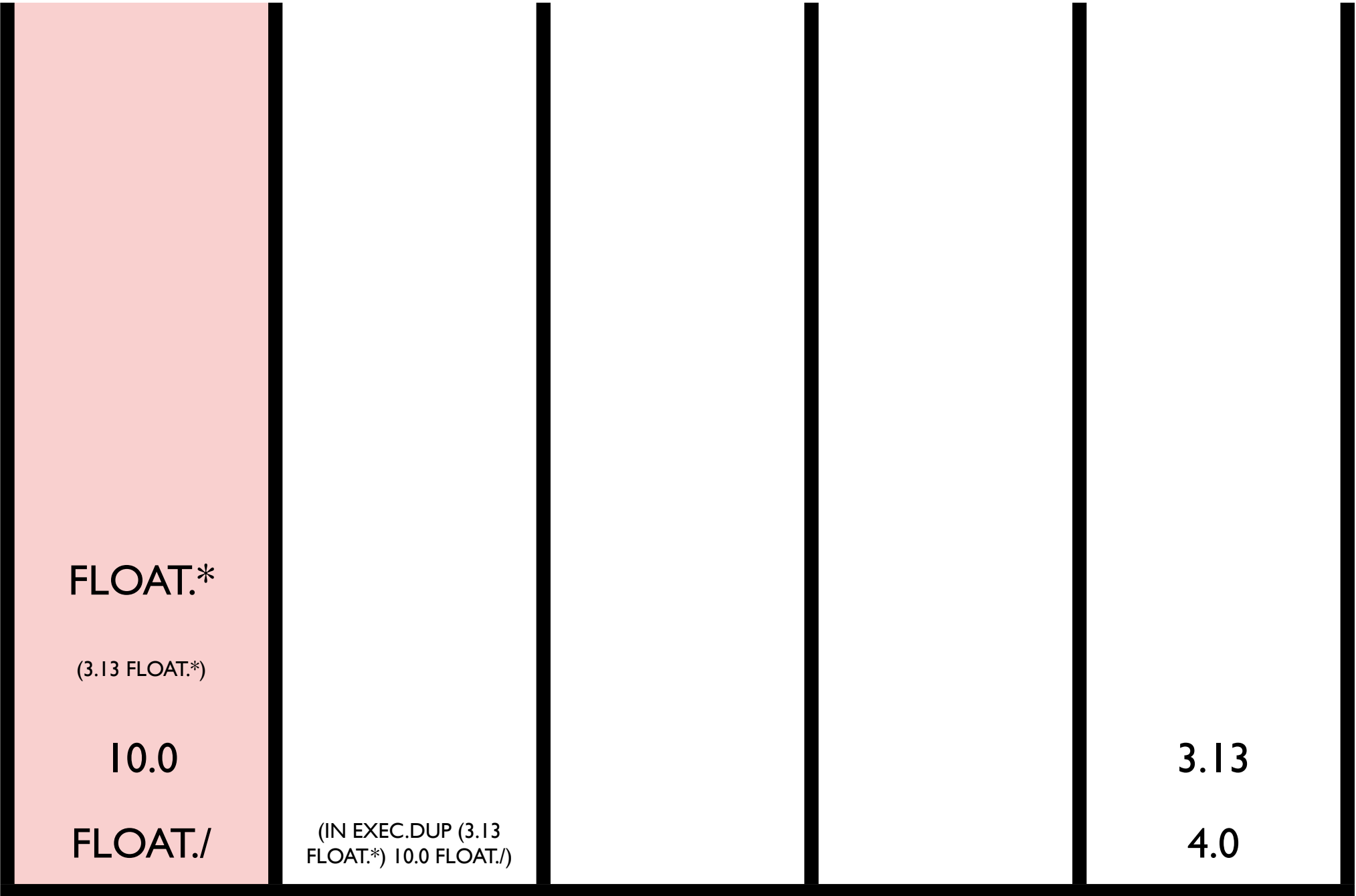
**exec**

**code**

**bool**

**int**

**float**



FLOAT.\*

(3.13 FLOAT.\*)

10.0

FLOAT./

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

3.13

4.0

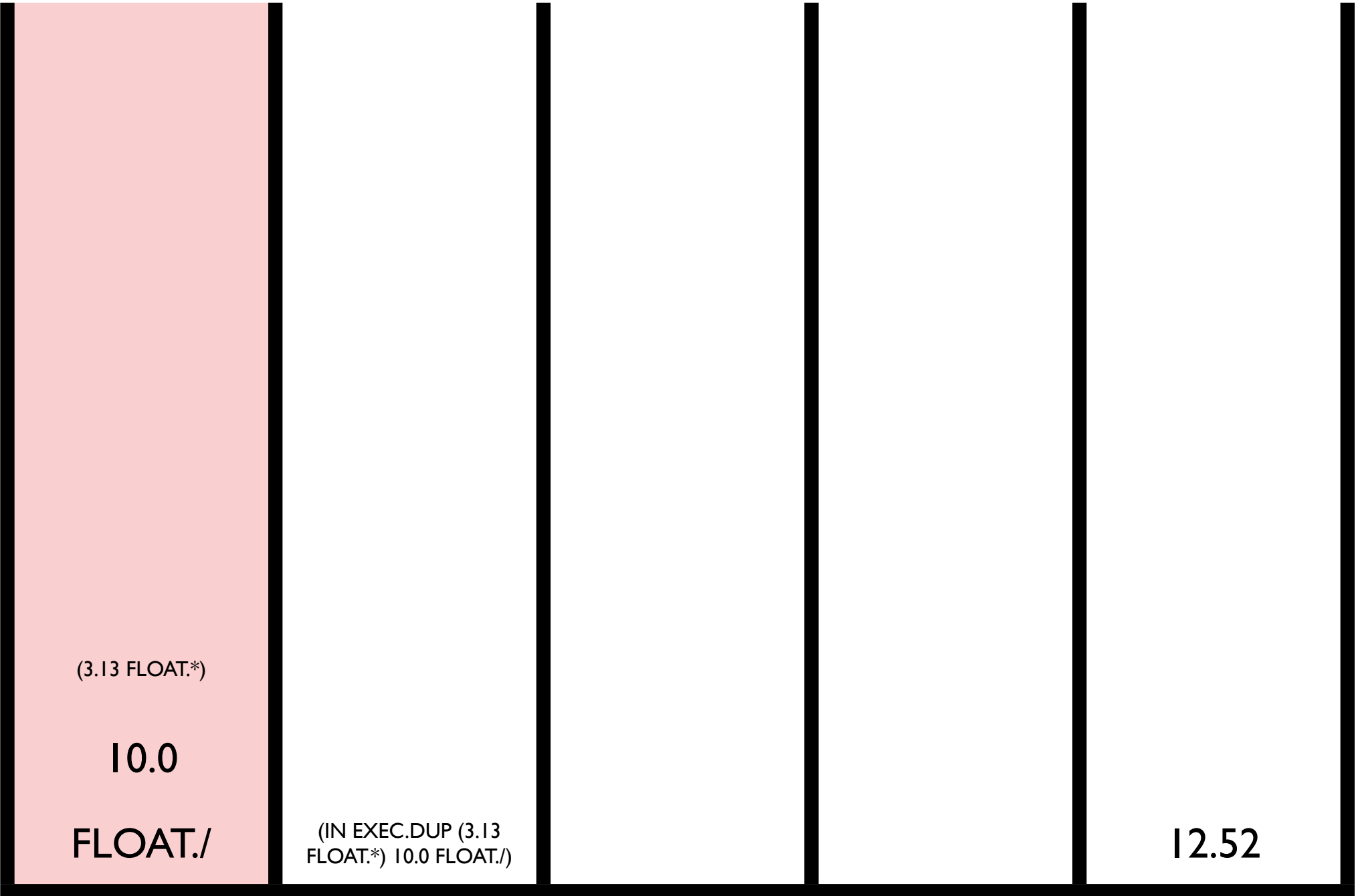
**exec**

**code**

**bool**

**int**

**float**



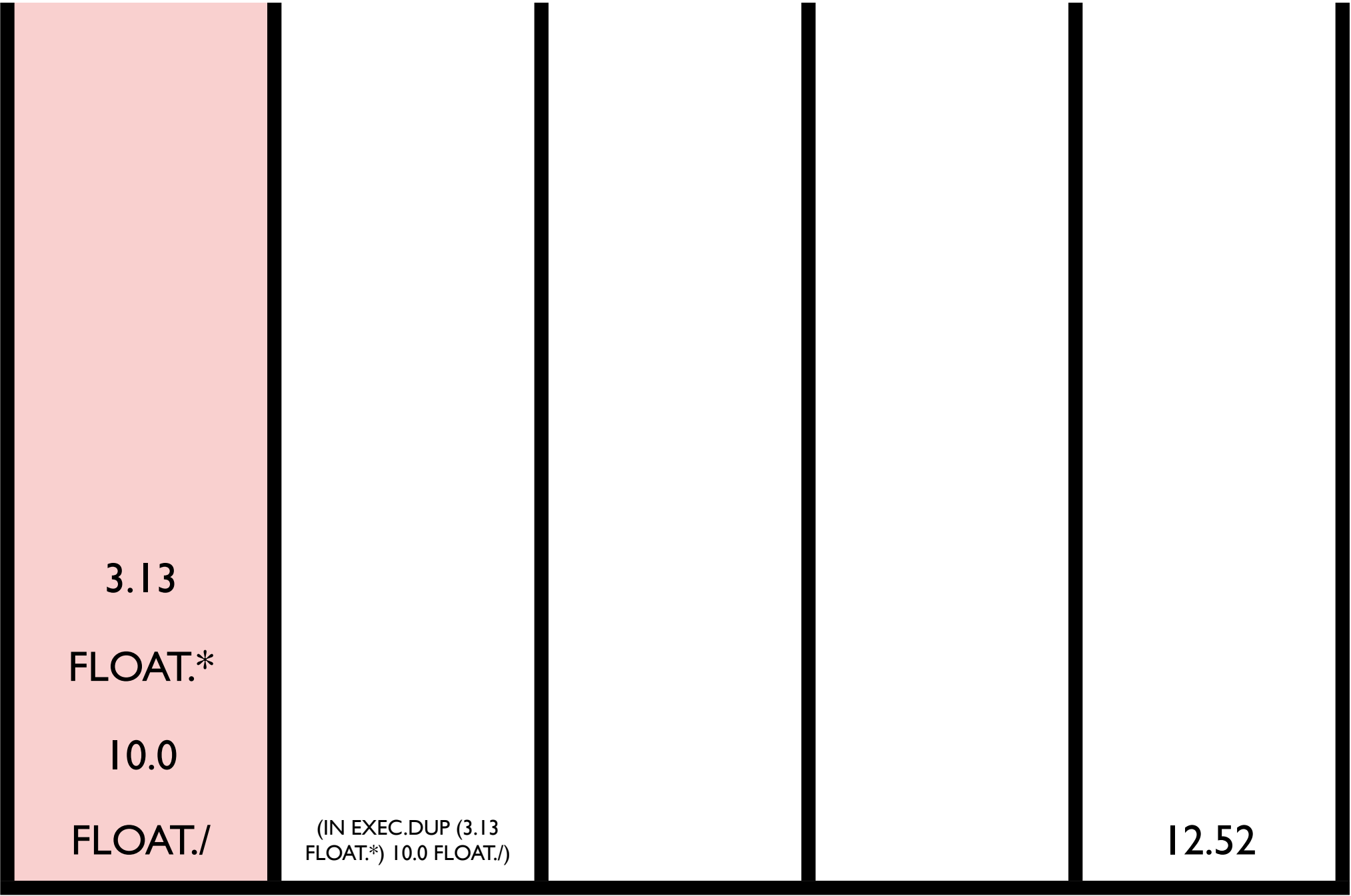
**exec**

**code**

**bool**

**int**

**float**



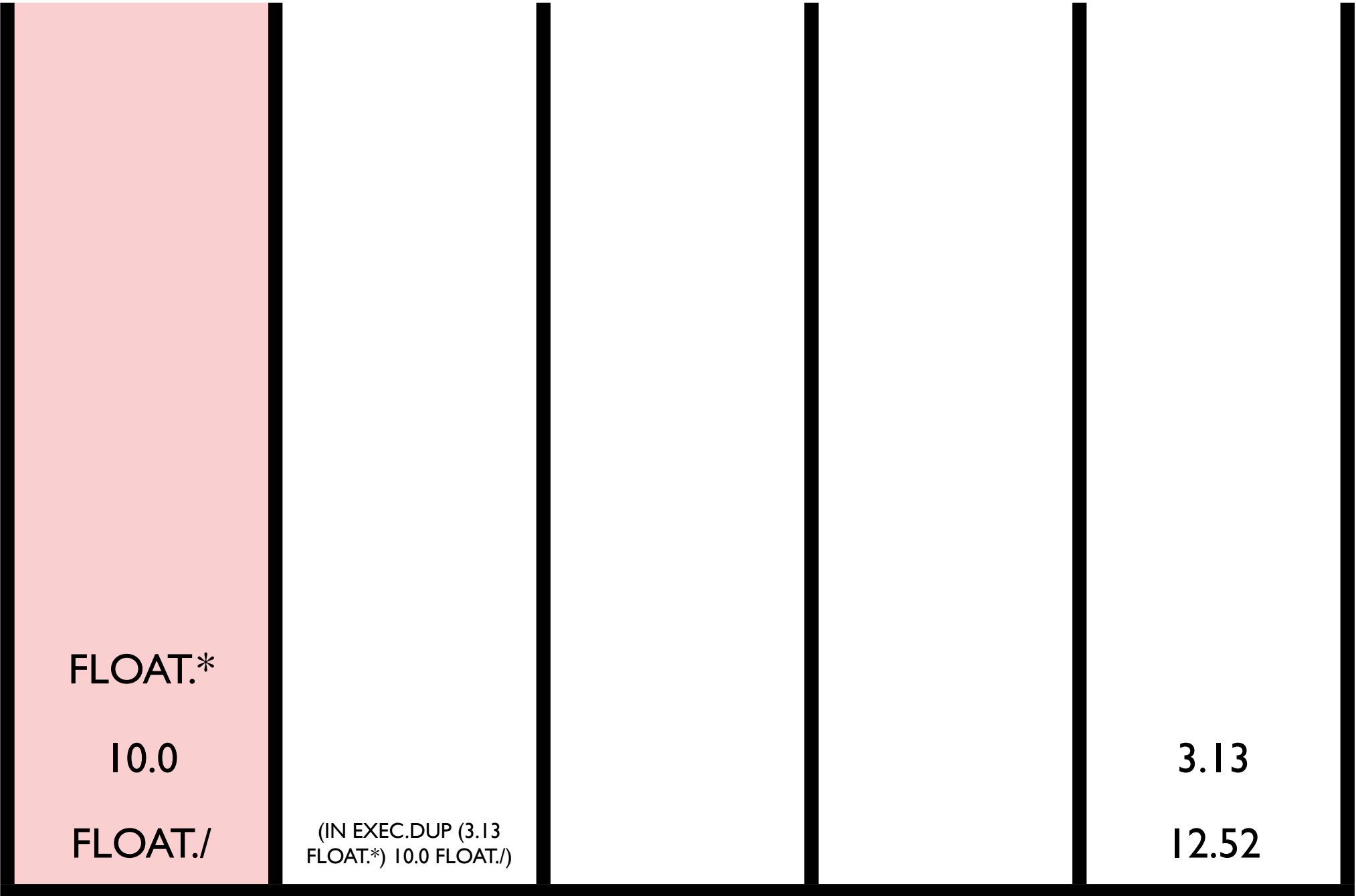
**exec**

**code**

**bool**

**int**

**float**



FLOAT.\*

10.0

FLOAT./

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

3.13

12.52

**exec**

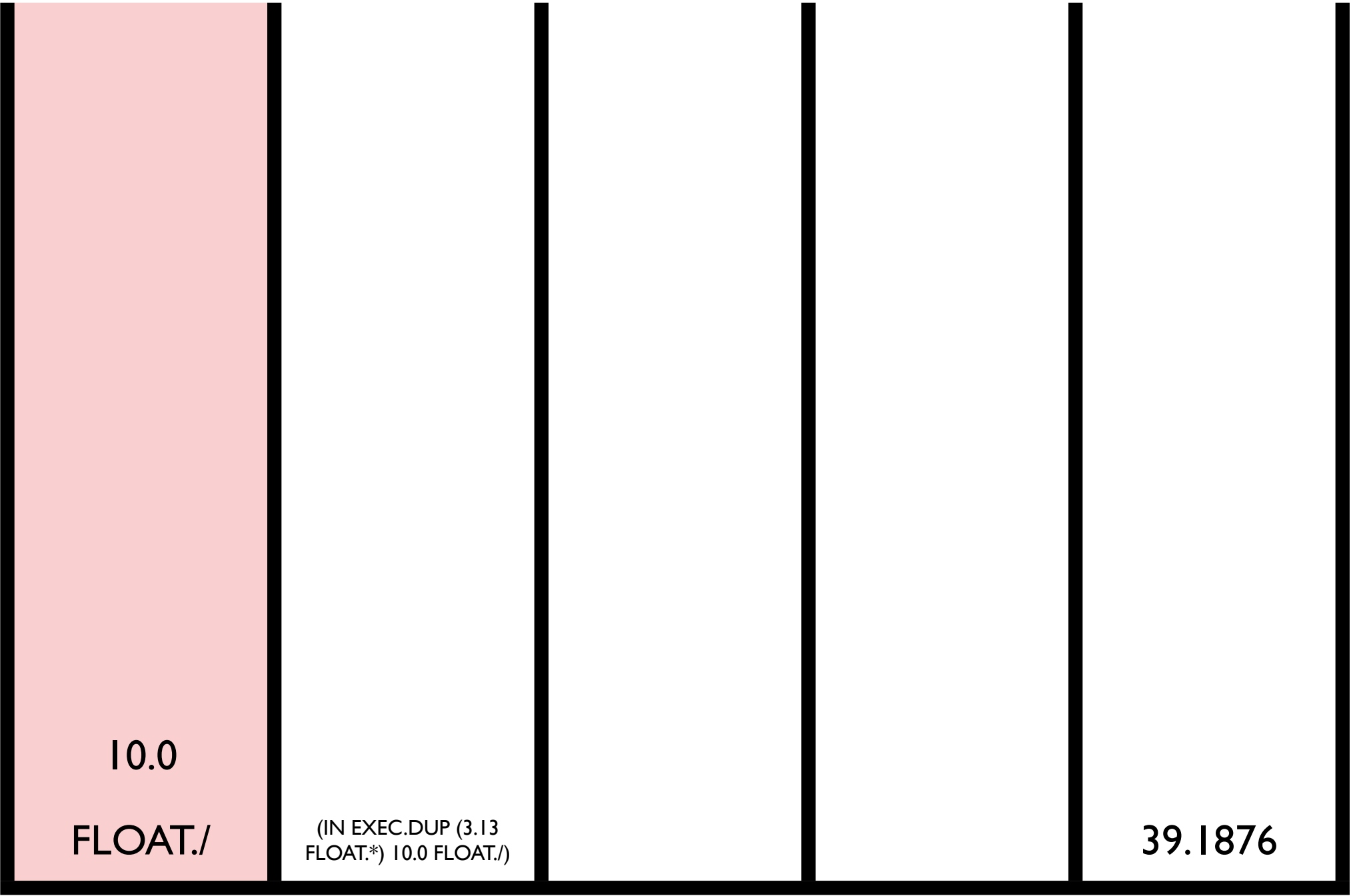
**code**

**bool**

**int**

**float**





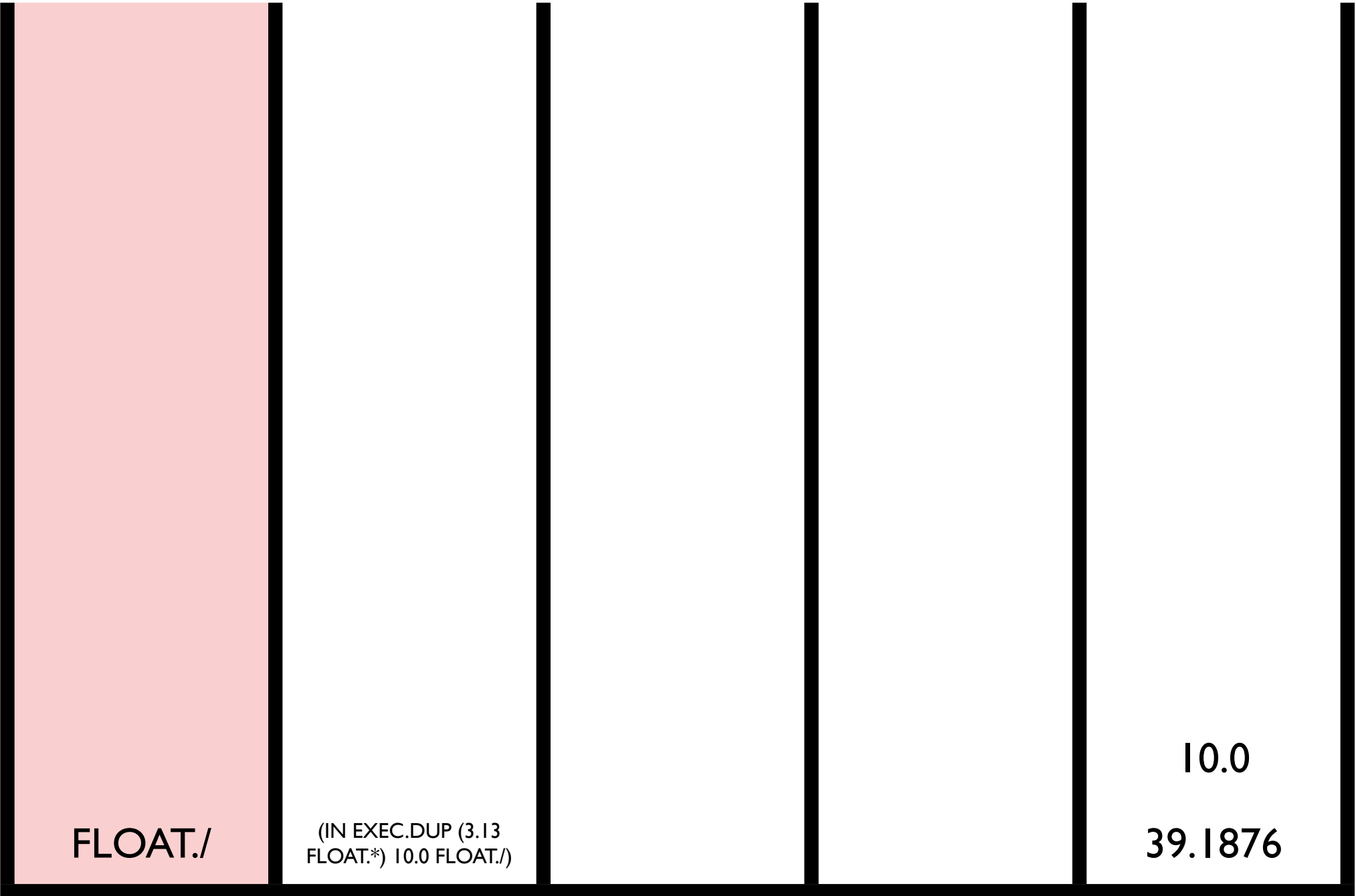
**exec**

**code**

**bool**

**int**

**float**



FLOAT./

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

10.0  
39.1876

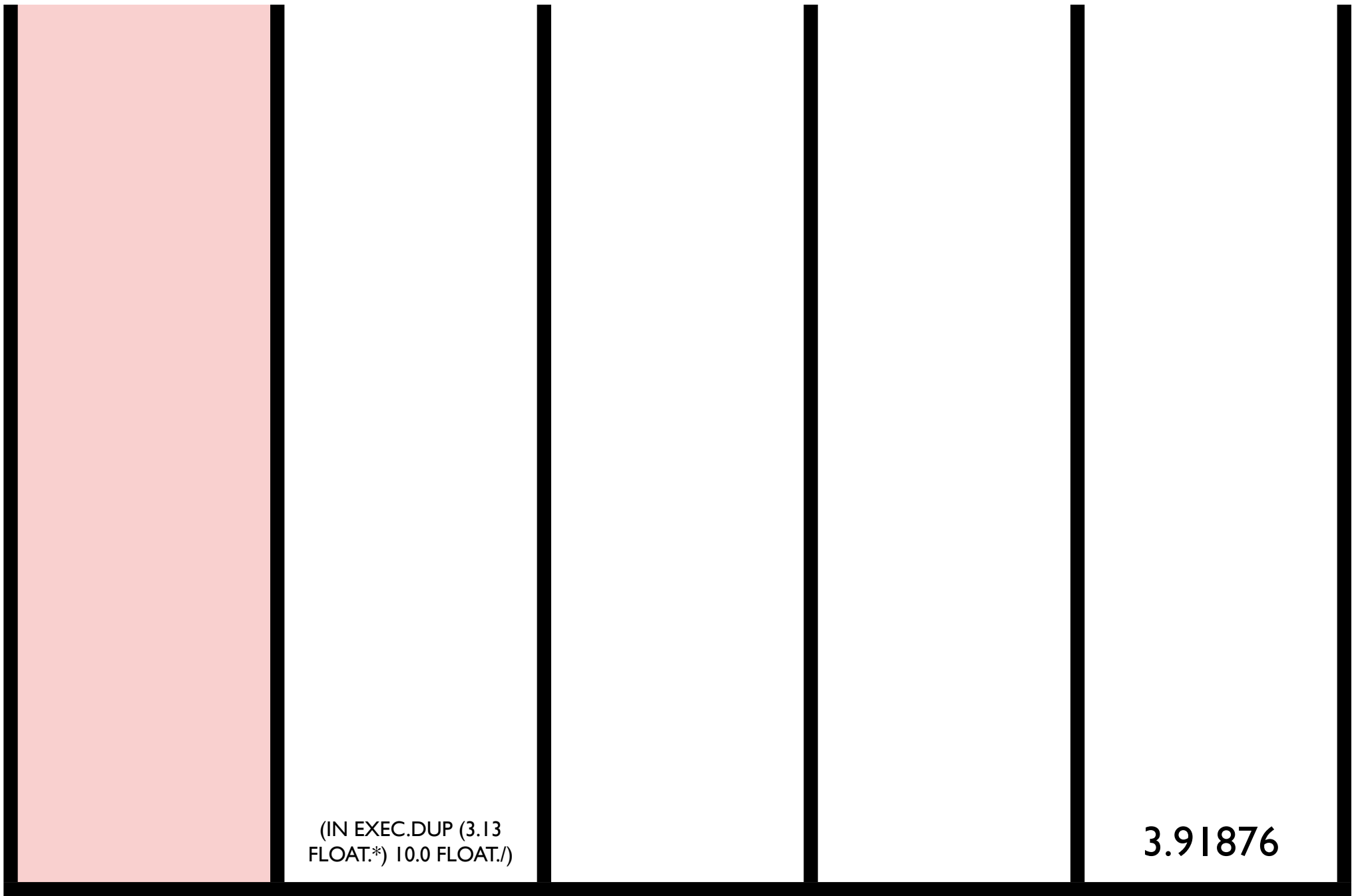
**exec**

**code**

**bool**

**int**

**float**



**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.* ) )
```

# Modularity

## Ackley and Van Belle

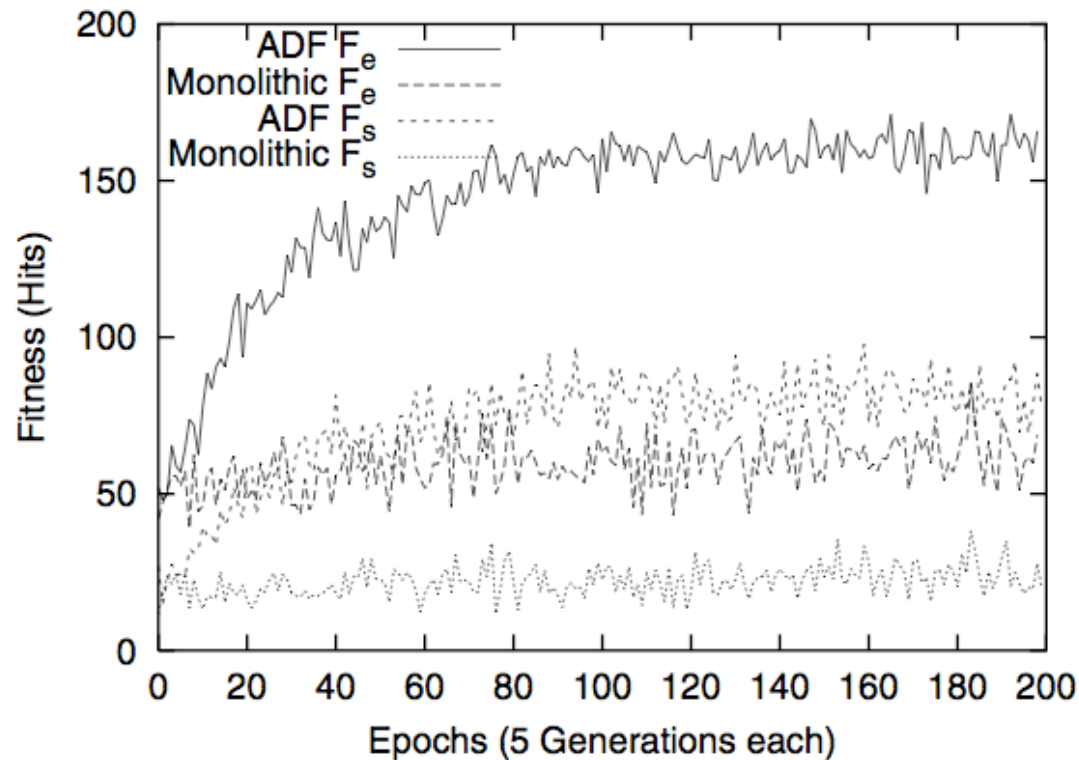
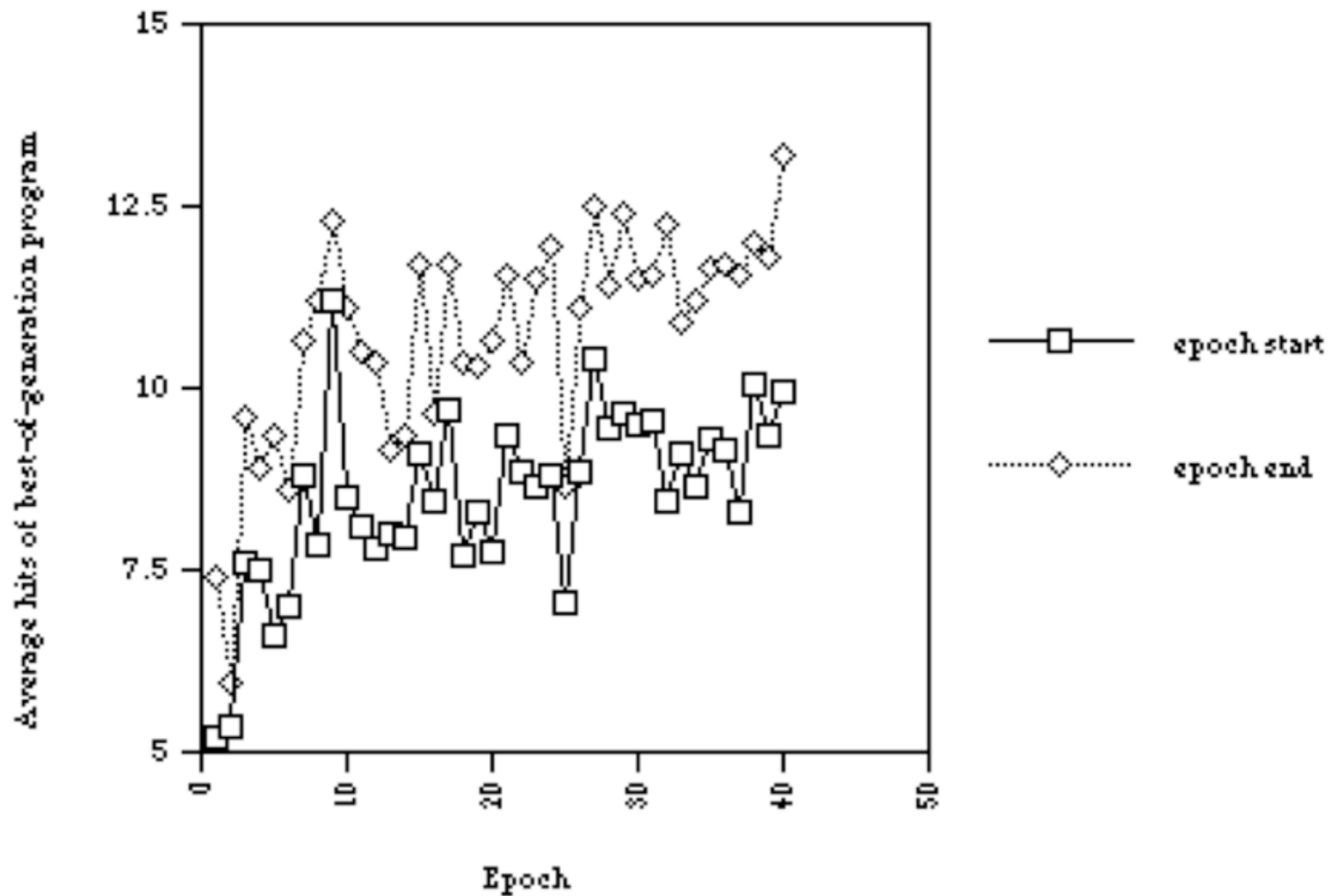


Figure 2: Average fitness values at the start ( $F_s$ ) and end ( $F_e$ ) of each epoch when regressing to  $y = A \sin(Ax)$ .  $A$  is selected at the start of each epoch uniformly from the range  $[0, 6)$ .

# Modularity via Push





# The Odd Problem

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



# Auto-simplification

Loop:

Make it randomly simpler

If it's as good or better: keep it

Otherwise: revert

# Evolved List Reverse (2)

- The evolved general solution simplifies to:  

```
(CODE.DO* INTEGER.STACKDEPTH EXEC.DO*TIMES  
CODE.FROMINTEGER CODE.STACKDEPTH  
EXEC.DO*TIMES CODE.CONST)
```
- This works by executing the input list, then moving all of the integers individually to the CODE stack, then building the reversed list.

# Evolved Factorial

Two simplified evolved general solutions:

```
( 1 EXEC.DO*RANGE INTEGER.* )
```

*Runs a loop that just multiplies all of the loop counter values.*

```
( INTEGER.* INTEGER.STACKDEPTH CODE.DO*RANGE  
INTEGER.MAX )
```

*Recursively executes the whole program, which is on the CODE stack; INTEGER.STACKDEPTH produces the 1 for the loop index lower bound, and INTEGER.MAX pulls each product out from under each INTEGER.STACKDEPTH; only the first CODE.DO\*RANGE is executed in a context with code on the CODE stack.*

# Evolved Fibonacci

Two simplified evolved general solutions:

```
(EXEC.DO*TIMES (CODE.LENGTH EXEC.S)  
INTEGER.STACKDEPTH CODE.YANKDUP)
```

*Builds an expression with Fibonacci(input) instances of  
INTEGER.STACKDEPTH on the EXEC stack, then executes them all.*

```
(EXEC.DO*COUNT EXEC.S CODE.QUOTE NAME.=  
CODE.DO*COUNT CODE.YANKDUP CODE.DO*COUNT  
CODE.CON CONS CODE.STACKDEPTH)
```

*Builds an expression with Fibonacci(input) instances of NAME.= on  
the CODE stack, then executes CODE.STACKDEPTH.*

# Evolved Even Parity

- Input is list of Boolean values on the CODE stack.
- Goal is a *general* solution that solves the problem for any number of inputs.

# Evolved Even Parity (2)

Two simplified evolved general solutions:

```
(CODE.DO* EXEC.Y BOOLEAN.=)
```

*Terminates only when execution limit is reached; works only for even number of inputs.*

```
(( ((CODE.POP CODE.DO BOOLEAN.STACKDEPTH)  
(EXEC.DO*TIMES) (BOOLEAN.= BOOLEAN.NOT))) )
```

*100% correct, general, terminating; see paper for explanation.*



# Evolved Expt(2,*n*)

- Normally an easy problem, but here we attempted to evolve solutions without iteration instructions.
- The following evolved solution uses novel evolved control structures (but does not generalize beyond the training cases,  $n=1-8$ ):

```
((INTEGER.DUP EXEC.YANKDUP EXEC.FLUSH 2
CODE.LENGTH) 8 (2 8 INTEGER.* INTEGER.DUP)
(EXEC.YANK 8 INTEGER.* ((CODE.IF (EXEC.ROT))
BOOLEAN.DEFINE EXEC.YANK)))
```

# Evolved Sort

- Input/output in an external data structure accessed with `INTEGER.LIST-SWAP`, `INTEGER.LIST-LENGTH`, `INTEGER.LIST-GET`, `INTEGER.LIST-COMPARE`.
- Simplified evolved general solution that makes  $n*(n-1)$  comparisons:

```
( INTEGER.LIST-LENGTH INTEGER.SHOVE  
  INTEGER.STACKDEPTH CODE.DO*RANGE  
  INTEGER.YANKDUP INTEGER.DUP EXEC.DO*COUNT  
  INTEGER.LIST-COMPARE INTEGER.LIST-SWAP )
```

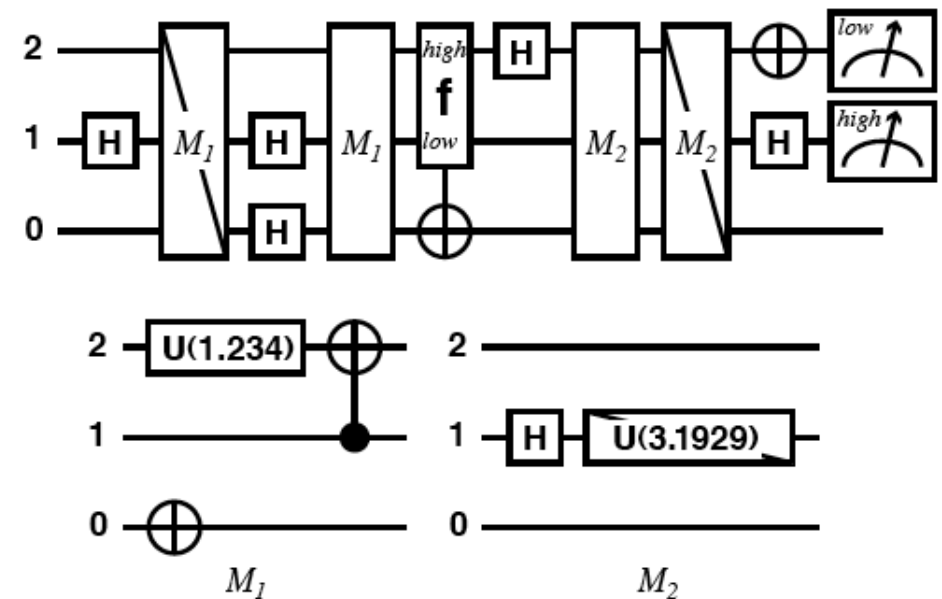
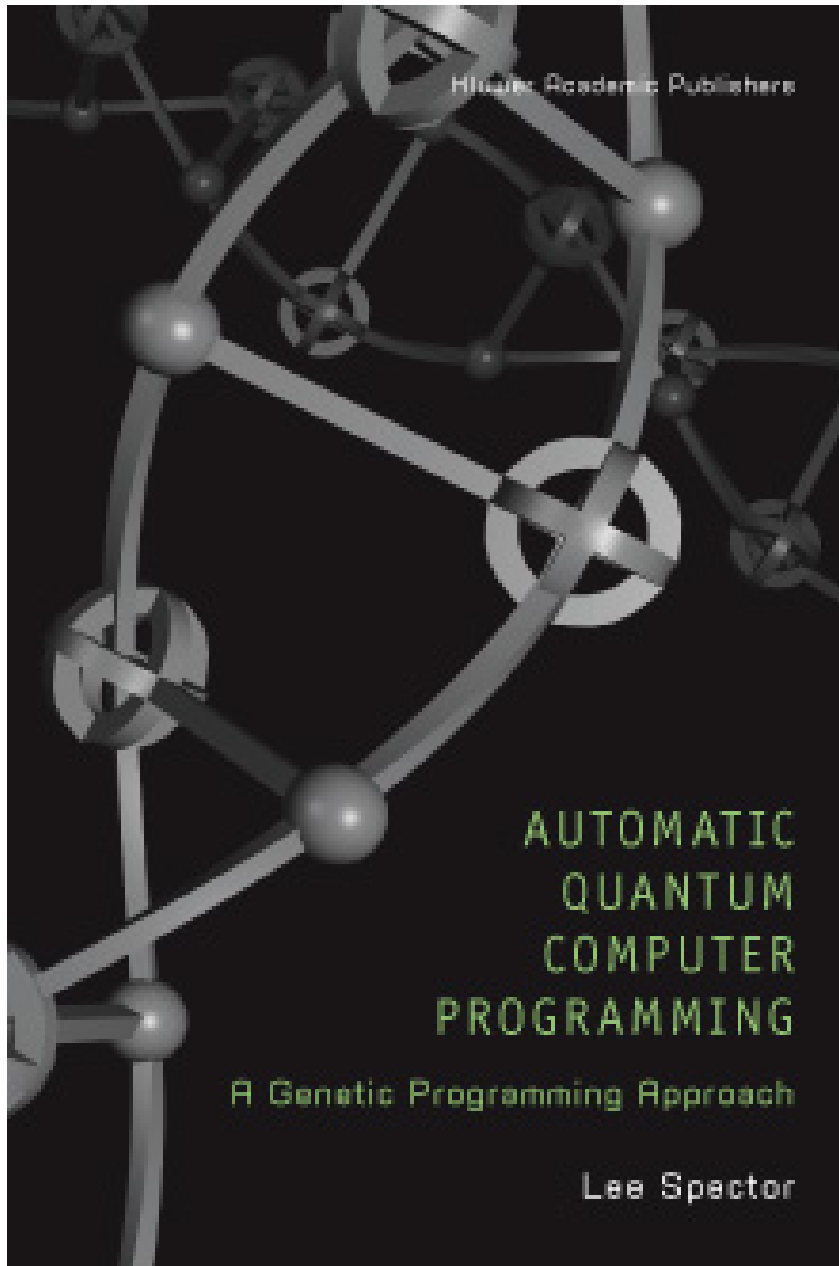


Figure 8.7. A gate array diagram for an evolved version of Grover's database search algorithm for a 4-item database. The full gate array is shown at the top, with  $M_1$  and  $M_2$  standing for the smaller gate arrays shown at the bottom. A diagonal line through a gate symbol indicates that the matrix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

# Motivation

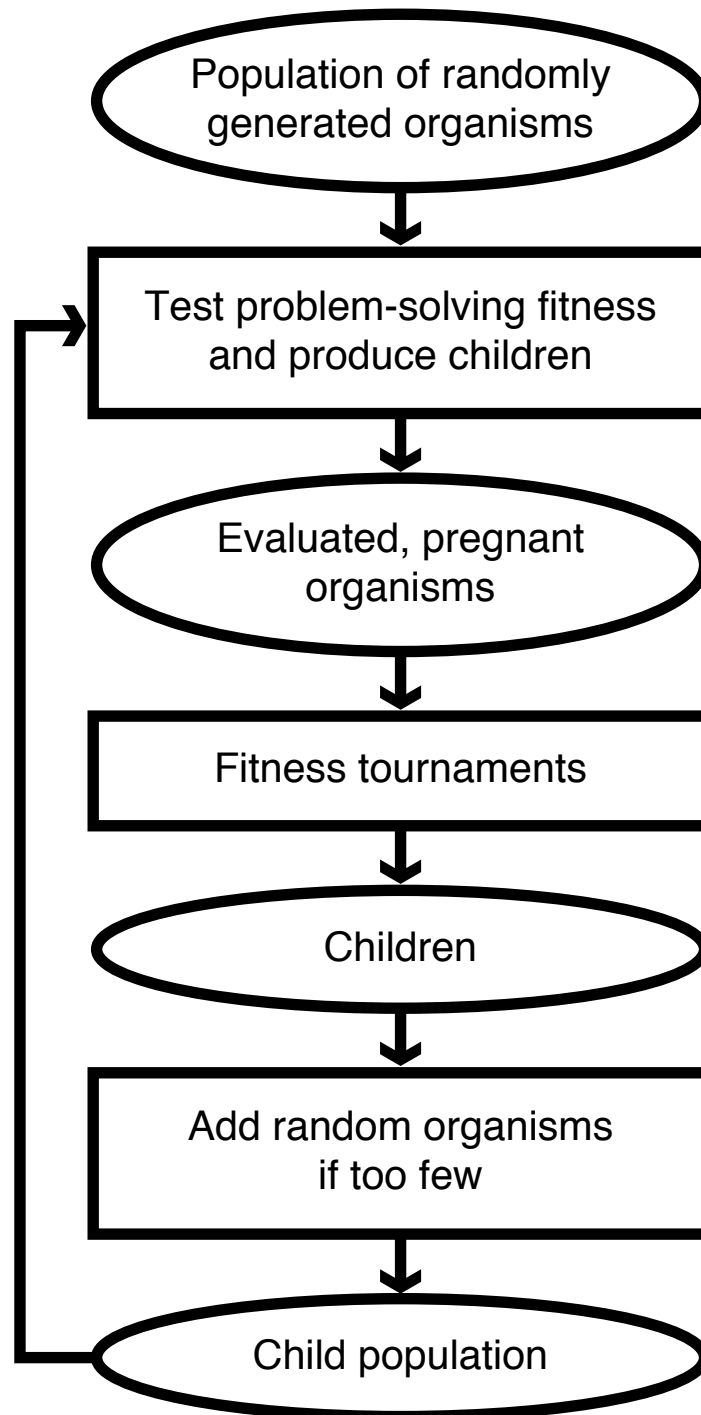
- We have **very little clue** about the best way to generate offspring in standard GP.
- We have **no clue whatsoever** about the best way to generate offspring in GP with the rich program representations that will become increasingly important.
- Natural reproductive methods evolved.
- Natural reproductive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

# 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

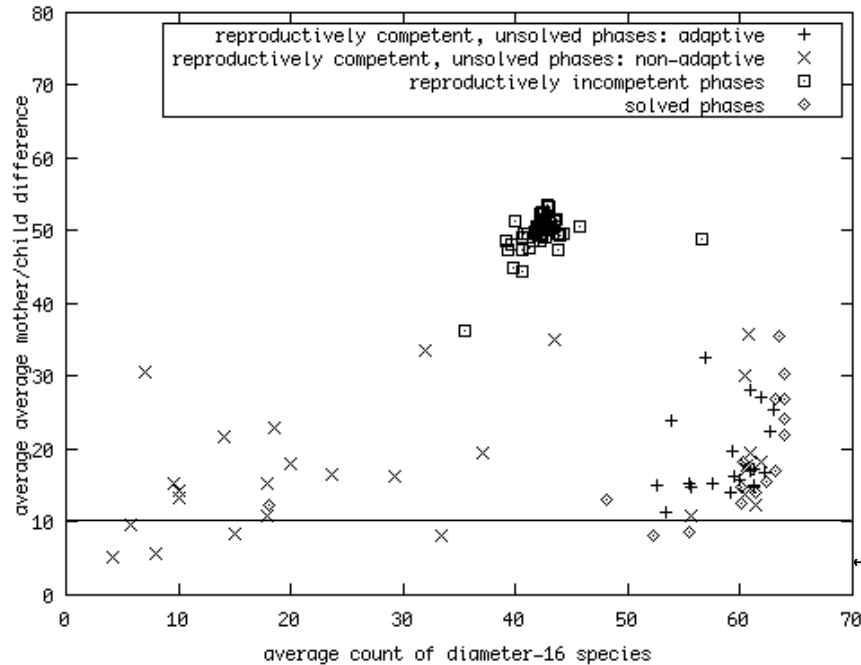
- A soup of evolving Push programs.
- Reproductive procedures emerge ex nihilo:
  - No hand-designed “ancestor.”
  - Children constructed by any computable process.
  - No externally applied mutation procedure or rate.
  - Exact clones are prohibited, but near-clones are permitted.
- Selection for problem-solving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



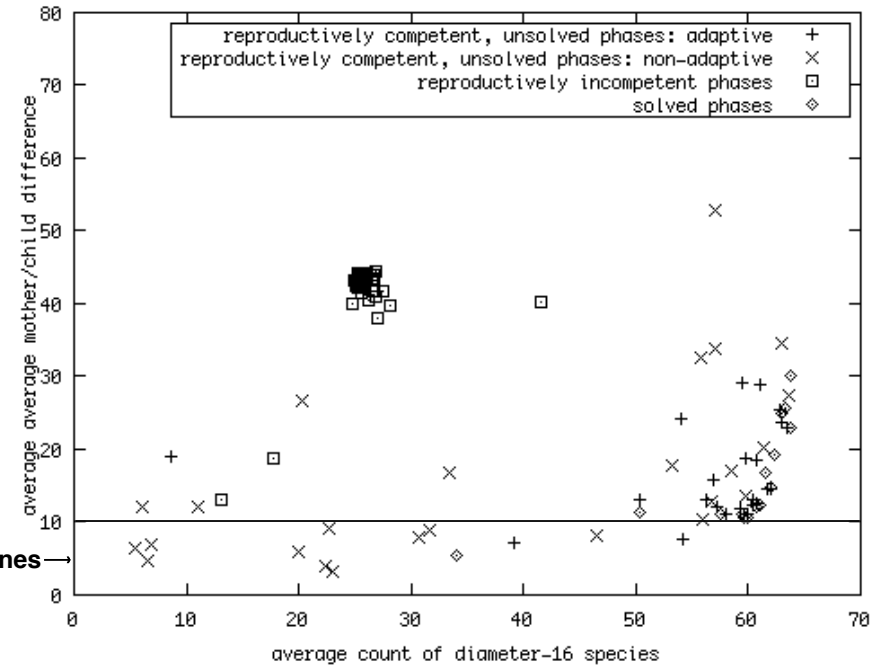


# # Species vs. Mother/Child Differences

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



**Runs including  
sexual instructions**



**Runs without  
sexual instructions**

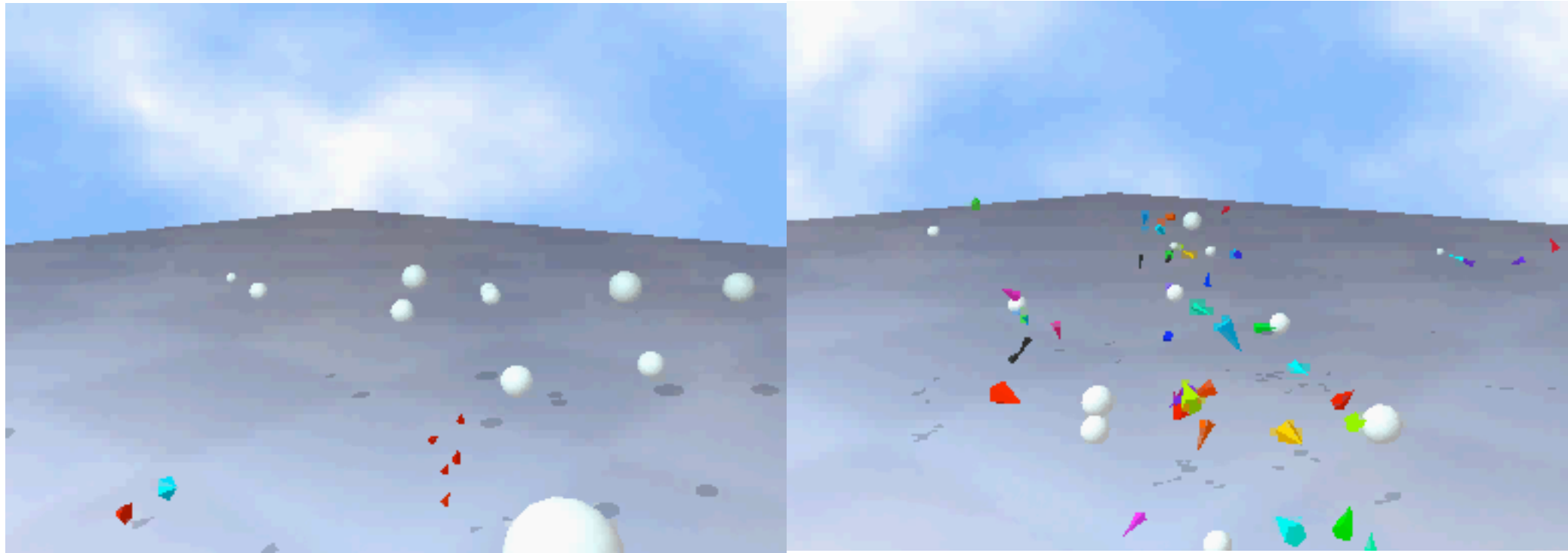
# Pushpop Results

- 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:

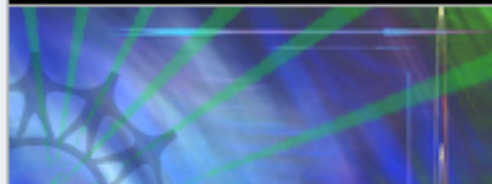
```
((integer_stackdepth (boolean_and  
code_map)) (integer_sub (integer_stackdepth  
(integer_sub (in (code_wrap (code_if  
(code_noop) boolean_fromfloat (2)  
integer_fromfloat) (code_rand integer_rot)  
exec_swap code_append integer_mult))))))
```

# Recent Enhancements

- Decimation (r-selection vs. k-selection)
- Reference via tags (Holland).



## 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

<b>NSF Org:</b>	<a href="#">IIS</a> <a href="#">Division of Information &amp; Intelligent Systems</a>
<b>Initial Amendment Date:</b>	August 19, 2010
<b>Latest Amendment Date:</b>	August 19, 2010
<b>Award Number:</b>	1017817
<b>Award Instrument:</b>	Standard Grant
<b>Program Manager:</b>	Sven G. Koenig IIS Division of Information & Intelligent Systems CSE Directorate for Computer & Information Science & Engineering
<b>Start Date:</b>	September 1, 2010
<b>Expires:</b>	August 31, 2013 (Estimated)
<b>Awarded Amount to Date:</b>	\$423288

This project is extending the science of automatic programming, using concepts derived from evolutionary biology and software engineering, to permit the evolution of general and robust computational systems with multiple interacting functionalities and interfaces. The project uses the PI's **Push** programming language as the target language for evolved programs. Push programs are syntactically unconstrained, which facilitates evolution, but they can make use of arbitrary control and data structures; this supports the evolution of complex, modular programs.

This project will add **new features to the Push language** and develop new methods that **allow requirements specifications and tests**, of the type employed in software engineering practice, **to be transformed into fitness functions** that drive evolution. The cumulative effect of these extensions will be to support the evolution of significantly more general and robust computational systems.

The effectiveness of the technologies developed in this project will be demonstrated in **two application areas**: the automatic programming of **small but complete productivity software applications** and the automatic programming of robustly intelligent software **agents for complex, time-varying economic games**. The project is contributing to long-standing goals in computer science of building robustly intelligent systems and automatic synthesis of useful computer programs.

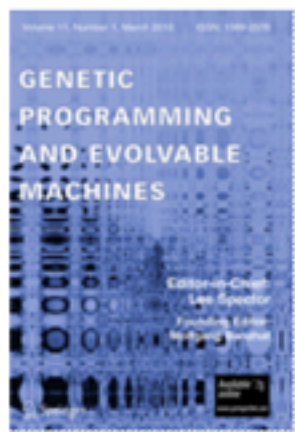
# Conclusions

- Rich representations, such as those provided by the Push programming language, can allow genetic programming to solve a wide range of difficult problems.
- **Bold (unsupported!) prediction:** *The most powerful, practical genetic programming systems of the future will be autoconstructive.*



[New User](#)
[LOGIN](#)
[HOME](#) > [MY SPRINGER](#) > [SUBJECTS](#) > [SERVICES](#) > [IMPRINTS & PUBLISHERS](#) > [ABOUT US](#)
 [GO](#)

## » Artificial Intelligence

[Home](#) > [Computer Science](#) > [Artificial Intelligence](#)
[SUBDISCIPLINES](#) > [JOURNALS](#) > [TEXTBOOKS](#) > [SERIES](#) > [BOOKS](#)
   [SHARE](#)


## Genetic Programming and Evolvable Machines

Main editor: L. Spector

ISSN: 1389-2576 (print version)

ISSN: 1573-7632 (electronic version)

Journal no. 10710


[ORDER OR RECOMMEND TO LIBRARIAN](#)
[ABOUT THIS JOURNAL](#) > [EDITORIAL BOARD](#)

...A unique source reporting on methods for artificial evolution of programs and machines...

Methods for artificial evolution of active components are rapidly developing branches of adaptive computation and adaptive engineering. They entail the development, evaluation and application of methods that mirror the process of neo-Darwinian evolution. Genetic Programming and Evolvable Machines reports innovative and significant progress in automatic evolution of software and hardware. It features both theoretical and application papers and covers hardware implementations, artificial life, molecular computing and emergent computation techniques.

In addition to its main topics, the journal covers related topics such as evolutionary algorithms with variable-size genomes, alternate methods of program induction, approaches to engineering

[READ THIS JOURNAL ON SPRINGERLINK](#)
[Online First Articles](#)
[Current Issue](#)
[Free Electronic Sample Copy](#)
[FOR AUTHORS AND EDITORS](#)
[Aims and Scope](#)
[Submit Online](#)
[Instructions for Authors](#)
[CFP: Evolvable Hardware Challenges \(pdf...](#)
[CFP EAs for Data Mining \(pdf, 120 kB\)](#)
[SERVICES FOR THE JOURNAL](#)
[Contacts](#)
[Order back issues](#)
[Bulk Orders](#)
[Article Reprints](#)
[ALERTS FOR THIS JOURNAL](#)