ULTRA!

Uniform Linear Transformation with Repair and Alternation in Genetic Programming

Lee Spector (*, +)
Thomas Helmuth (+)
* Cognitive Science, Hampshire College
+ Computer Science, UMass Amherst

Outline

- Uniformity
- The ULTRA genetic operator
- Push
- Problems
- Results

Uniformity

- Conversation with Evelyn Fox Keller
- Interactions between genome size, expression, functionality, gene competition, etc. in traditional GP; e.g. bloat from protection against crossover
- "But why would you do it that way?"
- Biological genetic variation, while not fully uniform, has uniformity properties that prevent some of the problems we see in GP

Uniform Variation

- All genetic material that a child inherits should be ≈ likely to be mutated
- Parts of both parents should be ≈ likely to appear in children (at least if they are ≈ in size), and to appear in a range of combinations
- Should be applicable to genomes of varying size and structure

Why Uniformity?

- No hiding from mutation
- All parts of parents subject to variation and recombination

Prior Work

- Point mutations or "uniform crossovers" that replace/swap nodes but only in restricted ways; cannot change structure, has depth biases (McKay et al, 1995; Page et al, 1998; Poli and Langdon, 1998; Poli and Page, 2000; Semenkin and Semenkina, 2012)
- Uniform mutation via size-based numbers of tree replacements; depth biases, little demonstrated benefit (McKay et al, 1995; Van Belle and Ackley, 2002)

Why it’s hard in GP

- Genomes are programs that have structure
- “Parts” of programs contain other parts!

Why $\approx$?

- Sequences and positions often matter, and many of the obvious ways to fully satisfy the stated uniformity criteria would not retain them
- Not clear that full uniformity is what one wants when parents have very different sizes

ULTRA

- Achieve uniformity by treating genomes as linear sequences, even if they are hierarchically structured
- Repair after transform to ensure structural validity

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
- Minimal syntax:
  program → instruction | literal | (program*)

Sample Push Instructions

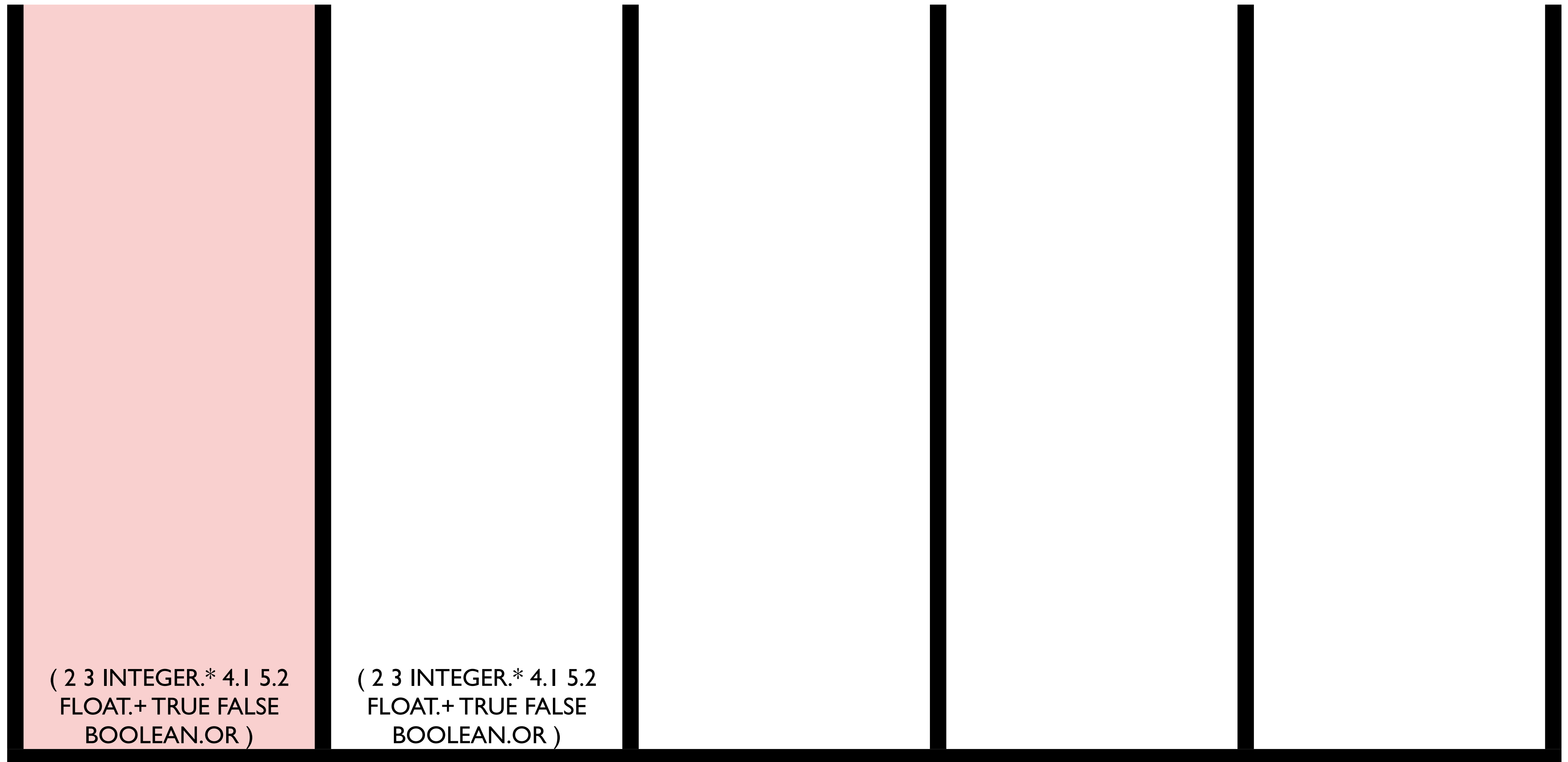
Stack manipulation instructions (all types)	POP, SWAP, YANK, DUP, STACKDEPTH, SHOVE, FLUSH, =
Math (INTEGER and FLOAT)	+, −, /, *, >, <, MIN, MAX
Logic (BOOLEAN)	AND, OR, NOT, FROMINTEGER
Code manipulation (CODE)	QUOTE, CAR, CDR, CONS, INSERT, LENGTH, LIST, MEMBER, NTH, EXTRACT
Control manipulation (CODE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

Push(3) Semantics

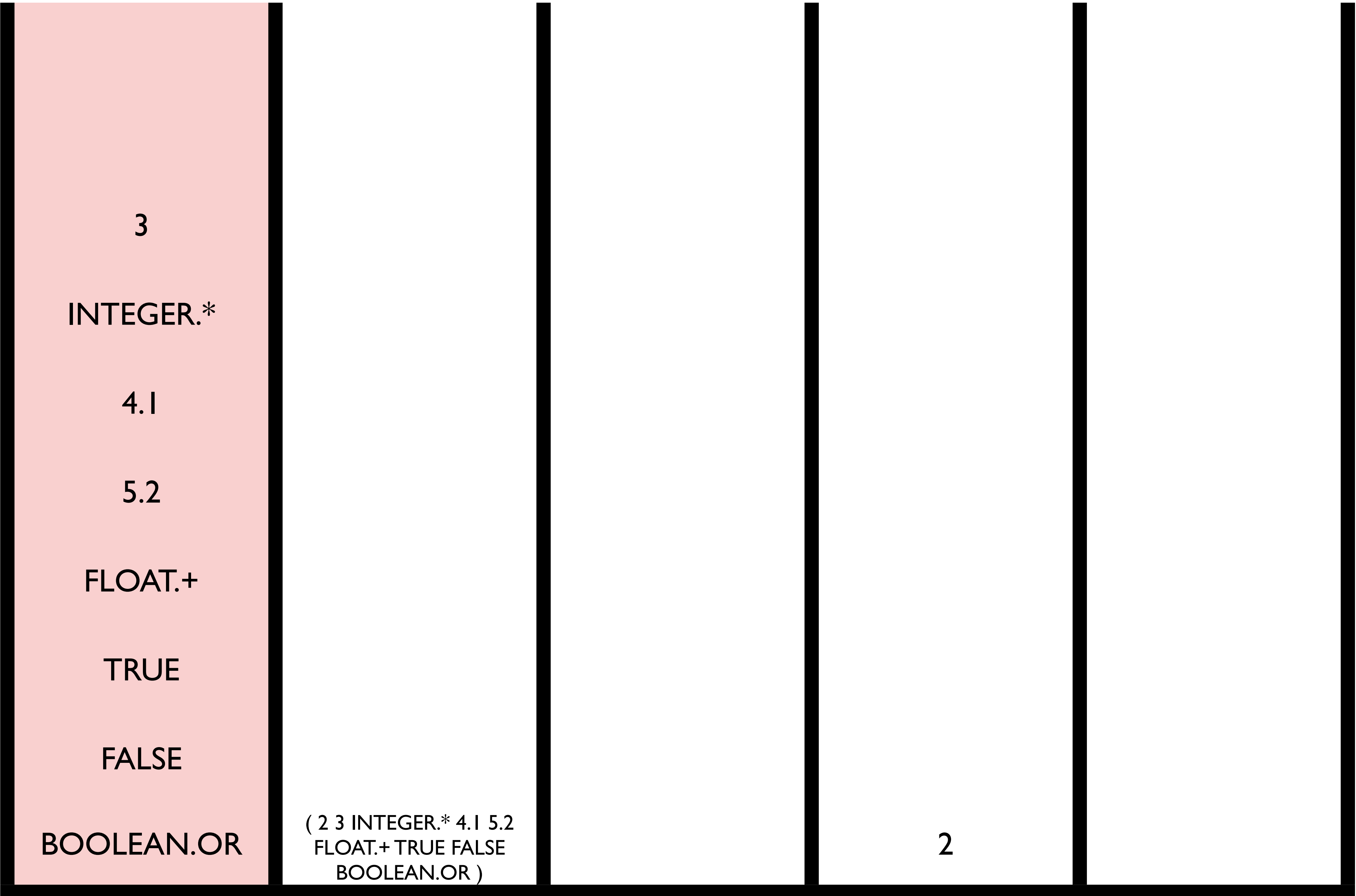
- To execute program P:

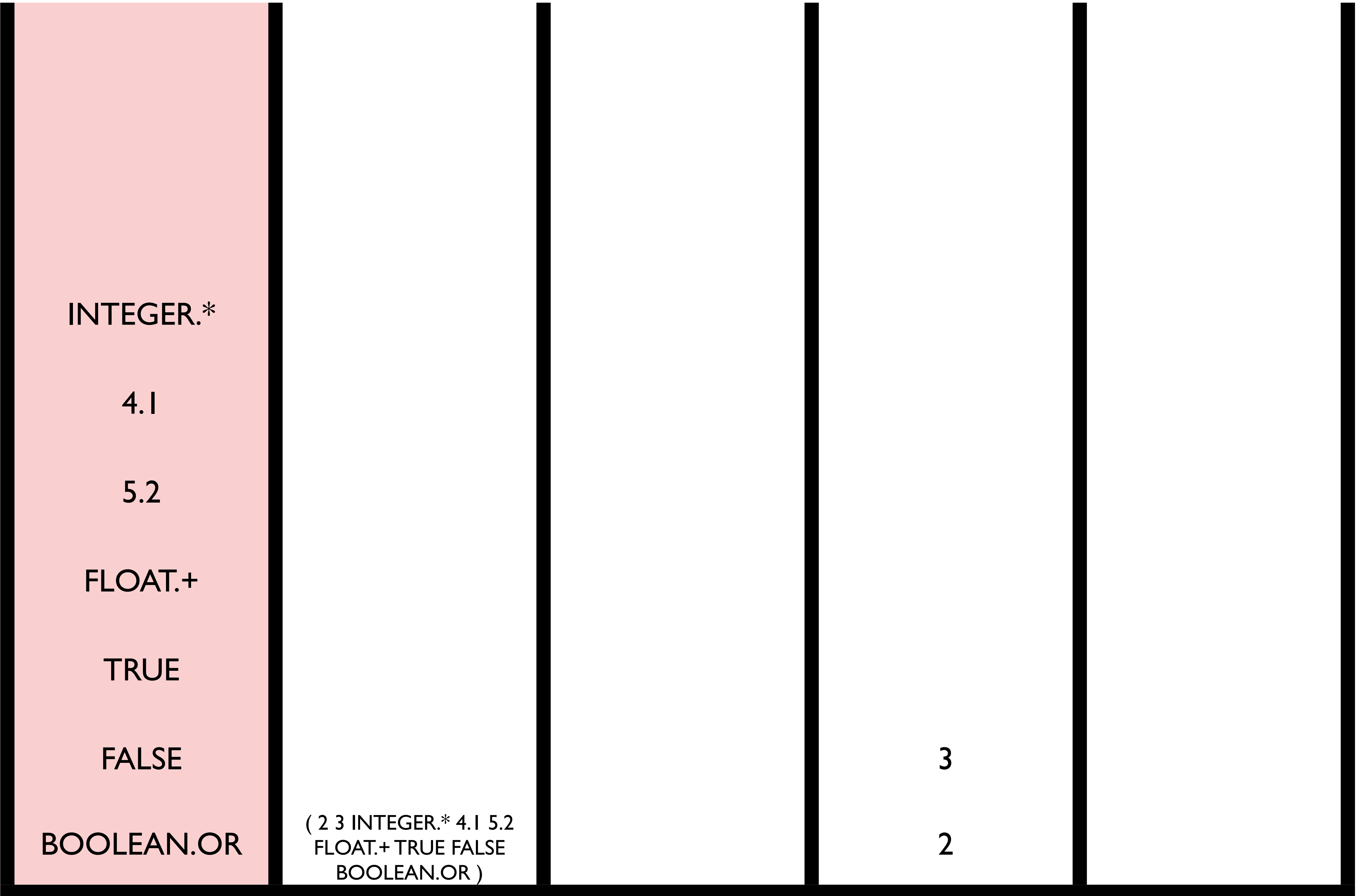
  1. Push P onto the **EXEC** stack.
  2. While the **EXEC** stack is not empty, pop and process the top element of the **EXEC** stack, E:
     (a) If E is an instruction: execute E (accessing whatever stacks are required).
     (b) If E is a literal: push E onto the appropriate stack.
     (c) If E is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

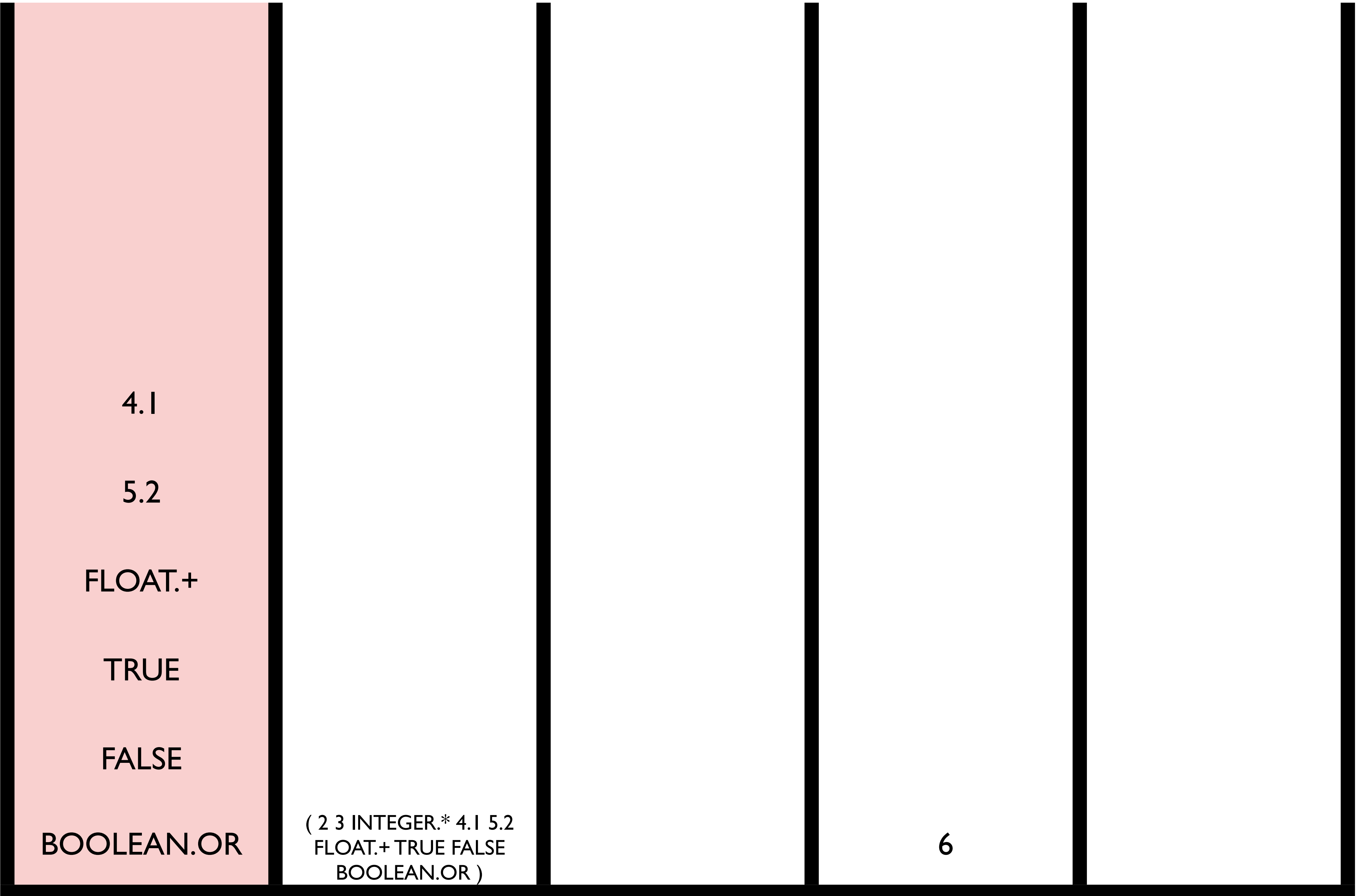
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)



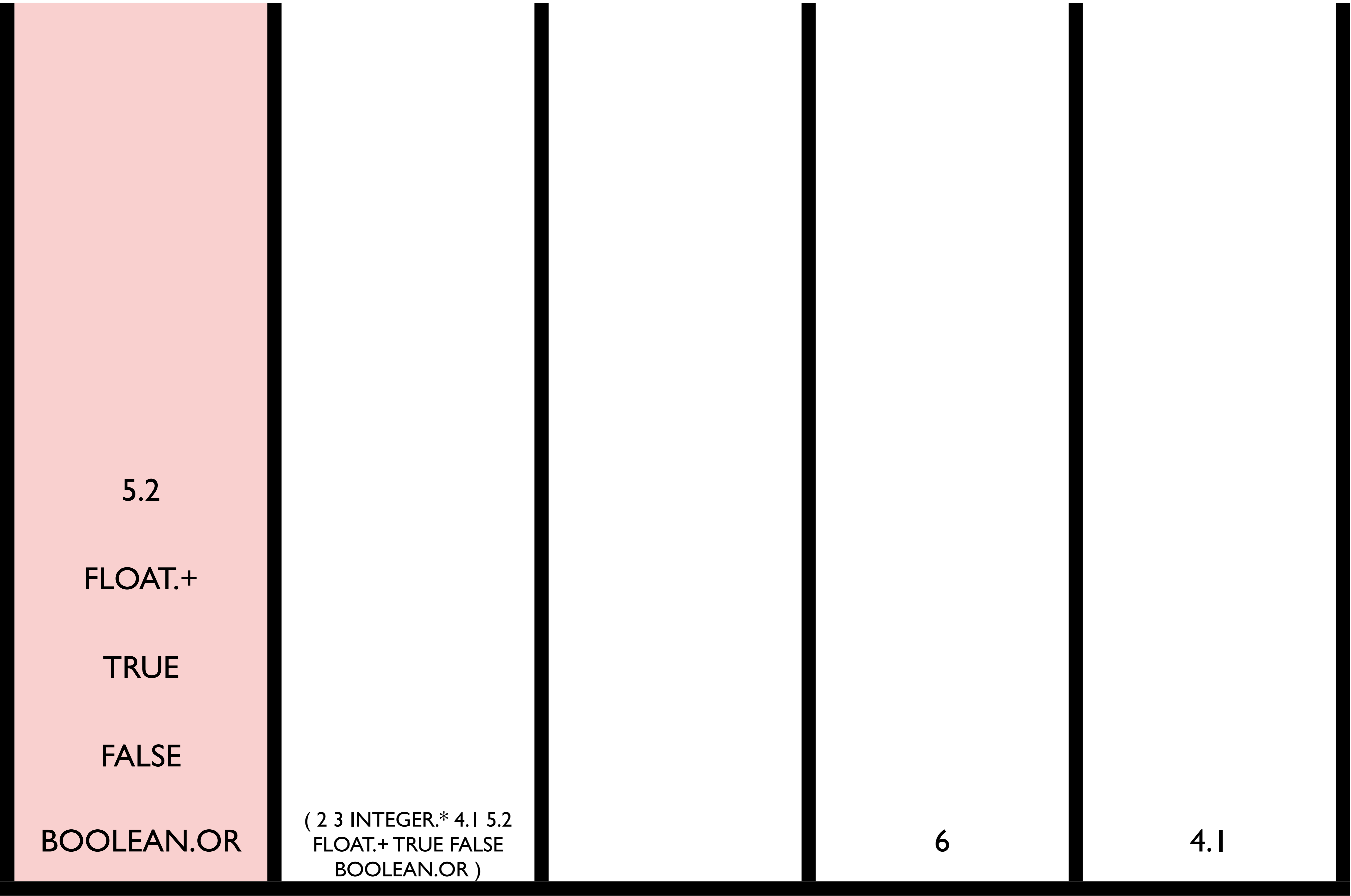
exec	code	bool	int	float
2				
3				
INTEGER.*				
4.1				
5.2				
FLOAT.+				
TRUE				
FALSE				
BOOLEAN.OR	(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)			

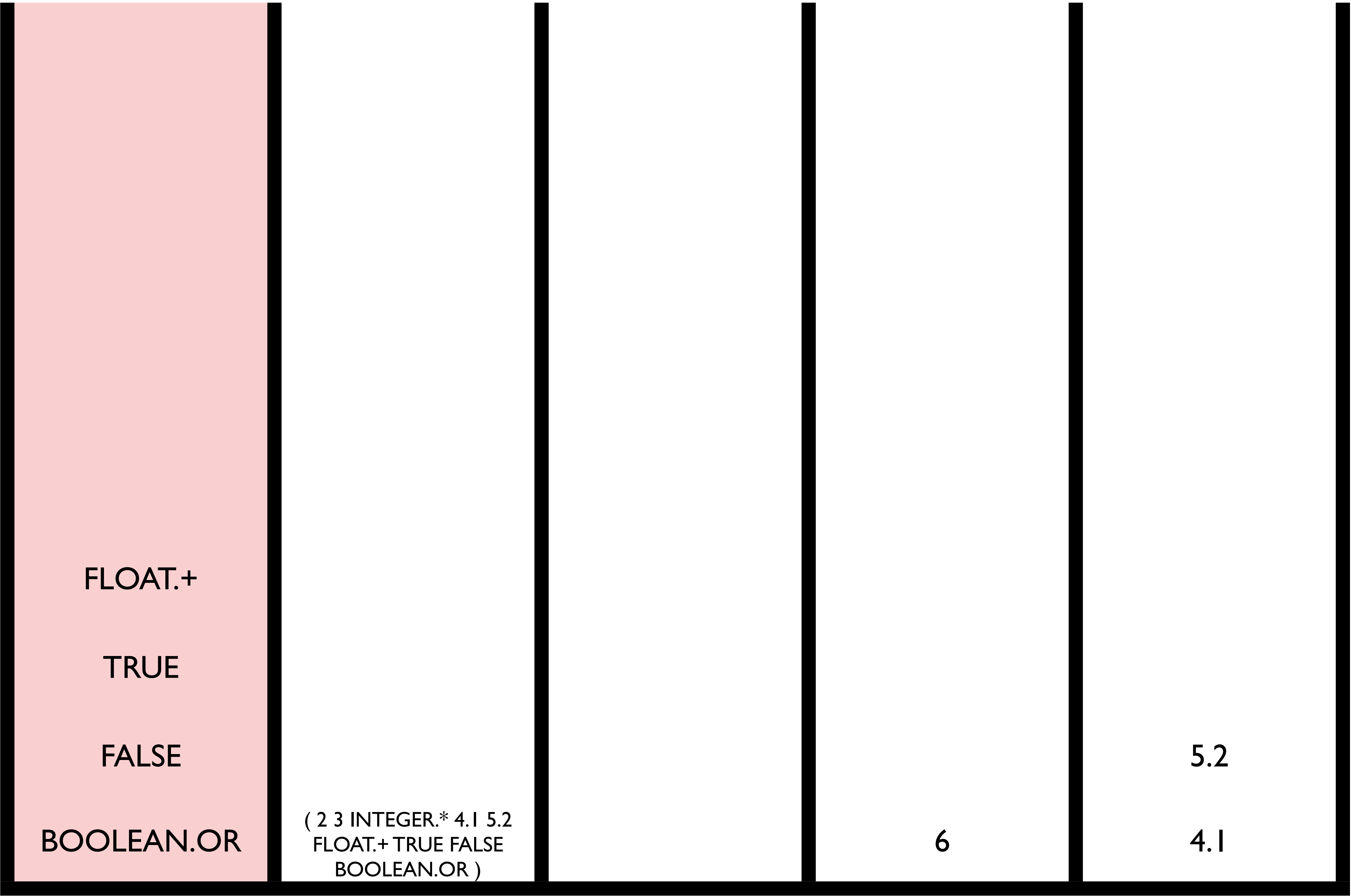

3
INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2
FLOAT.+ TRUE FALSE
BOOLEAN.OR)
2
exec
code
bool
int
float

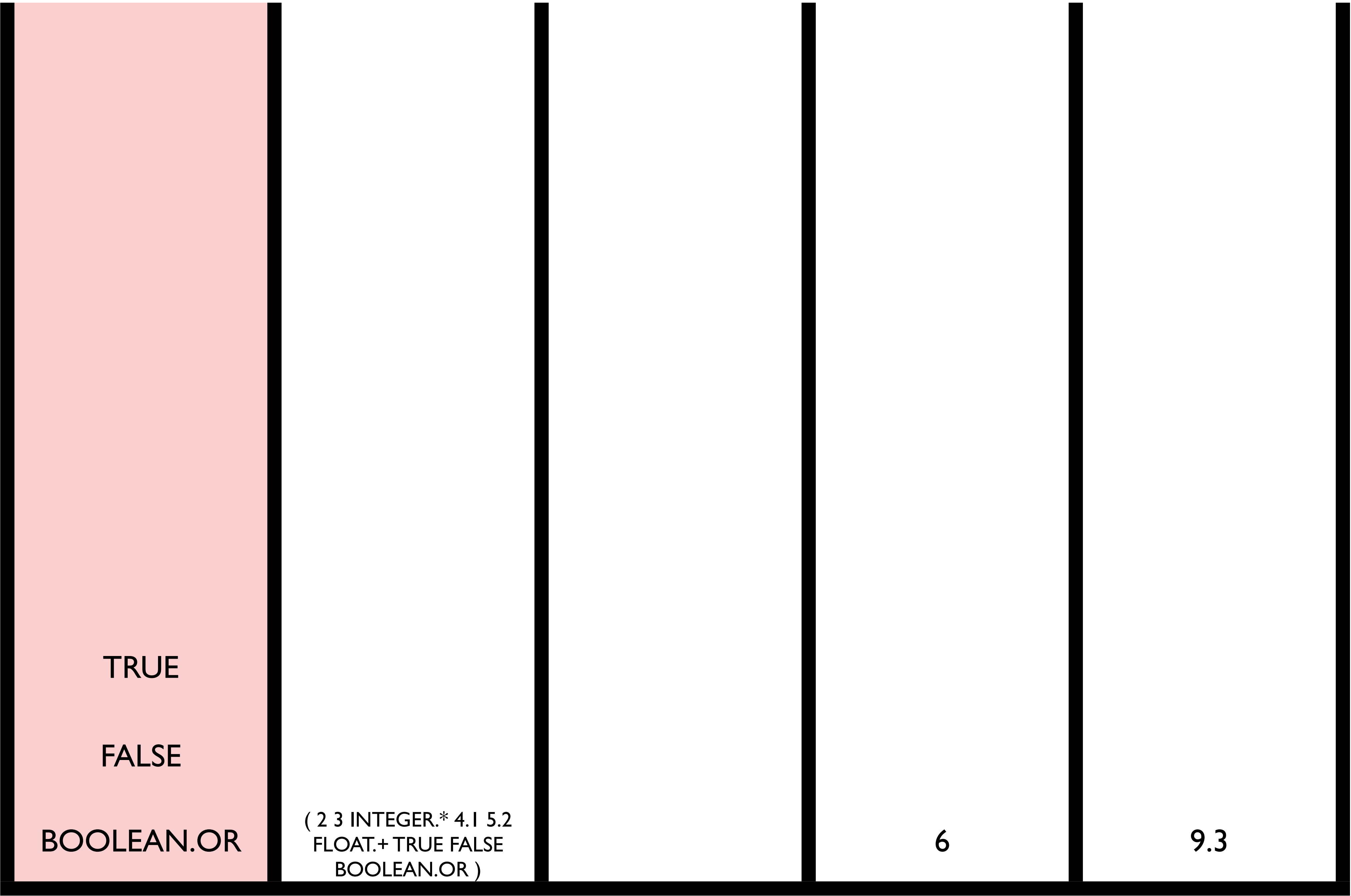

INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
3
2
exec
code
bool
int
float

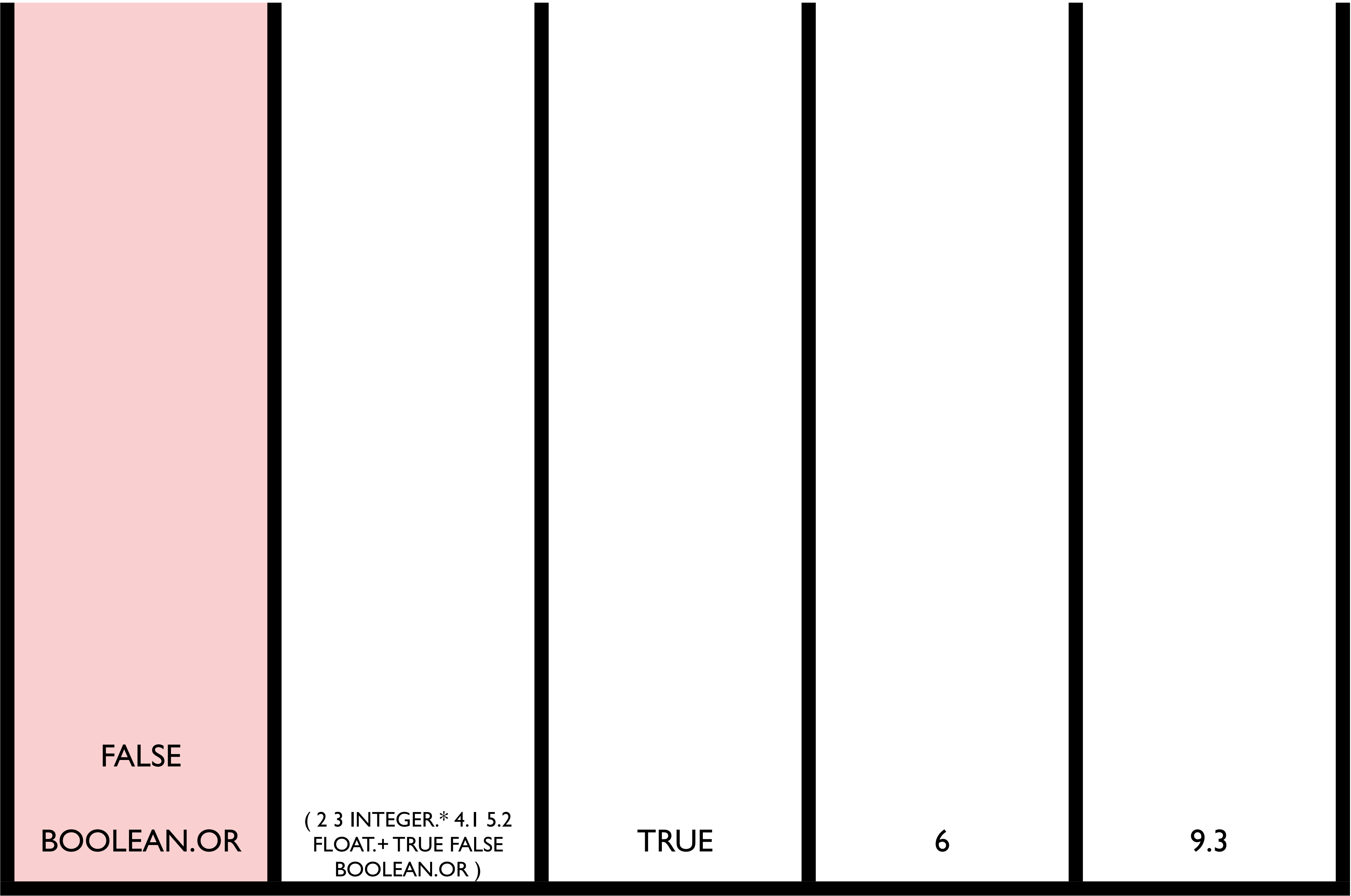


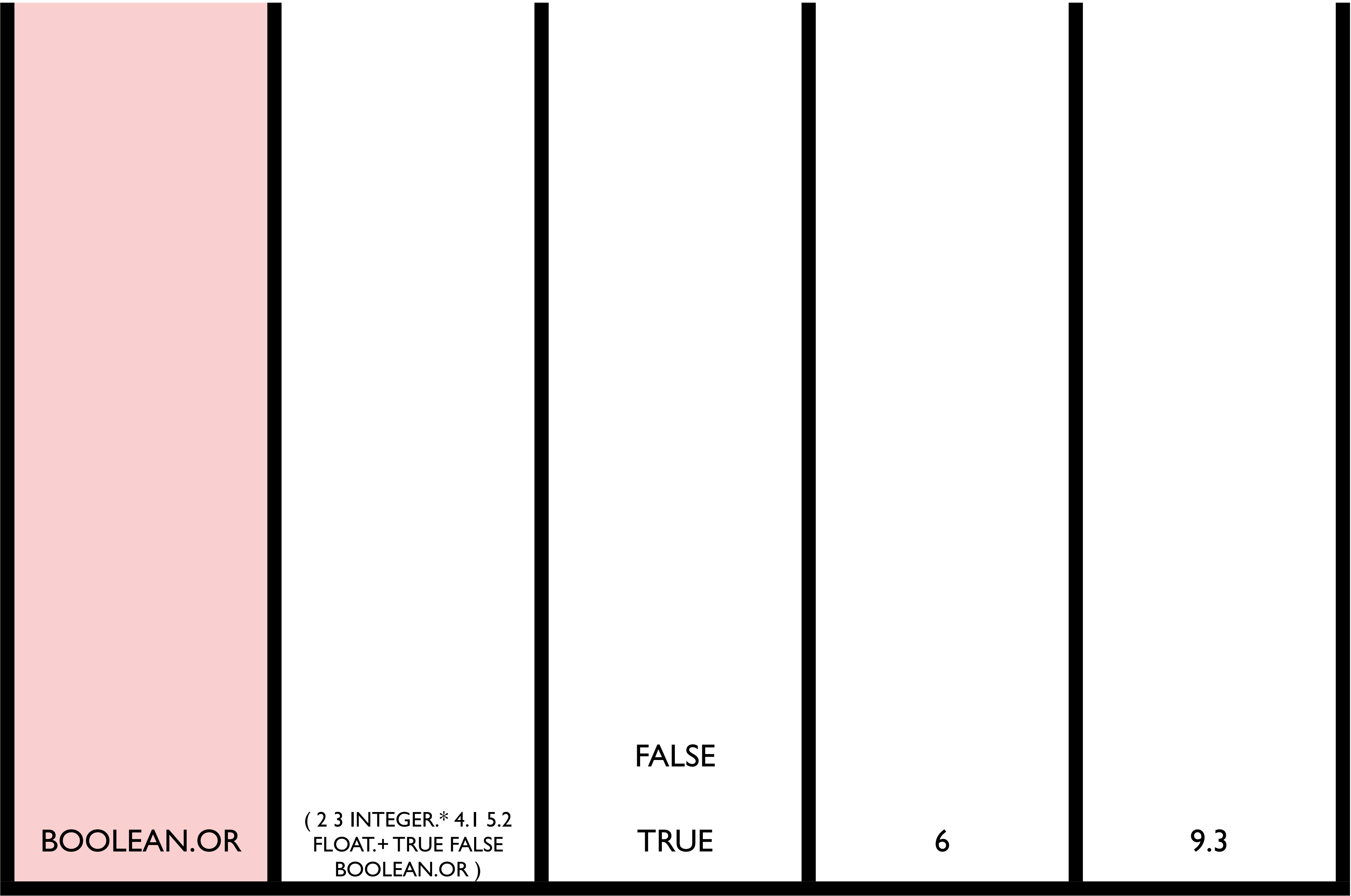
exec code bool int float

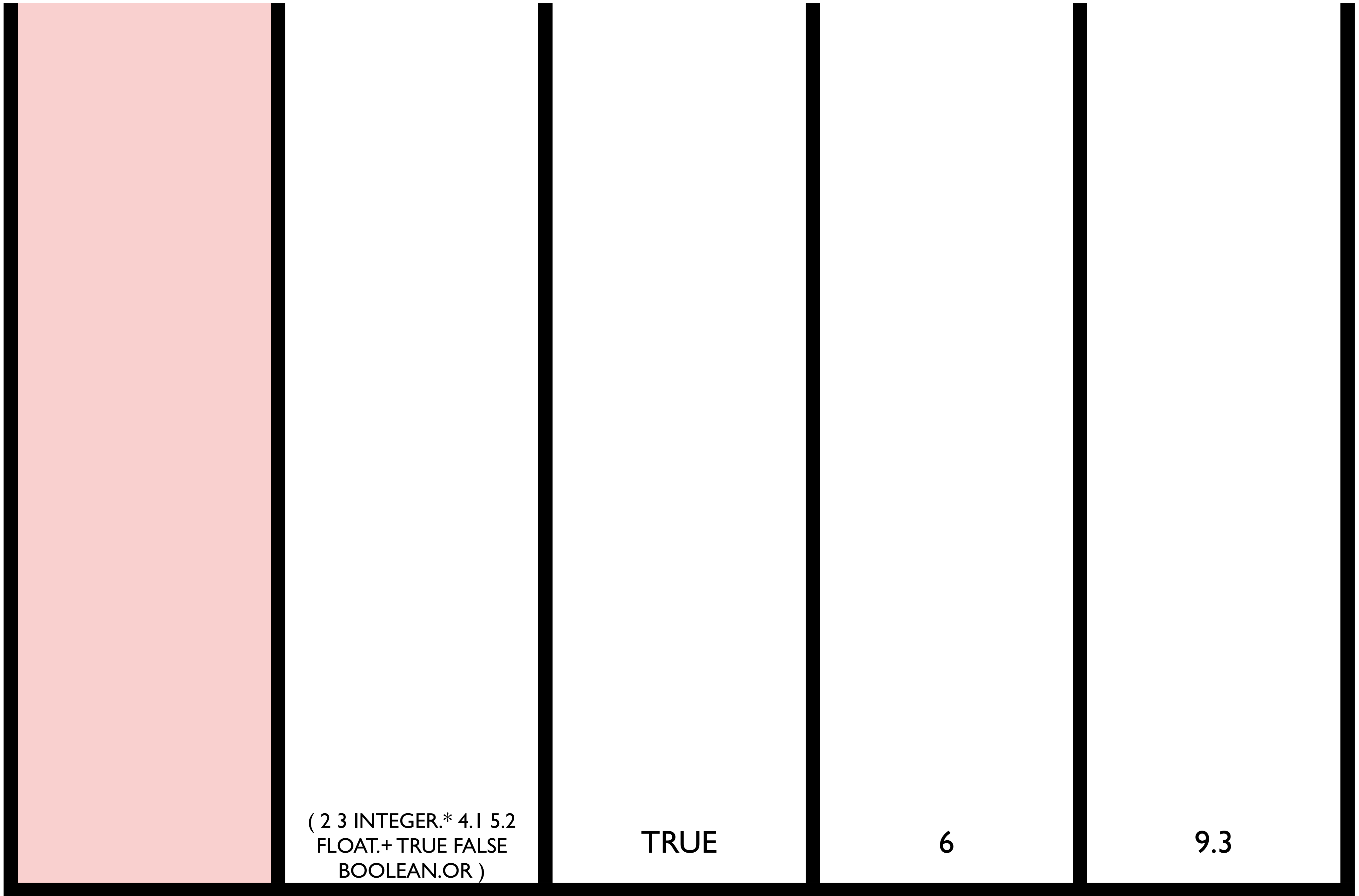

5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2
FLOAT.+ TRUE FALSE
BOOLEAN.OR)
6
4.1
exec
code
bool
int
float


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

Why Push?

- Many reasons!
- But one good reason to use it here is:
    - Push programs can be hierarchically structured, but every push program is structurally valid if its parentheses are balanced
    - This makes the Repair step of ULTRA, which we will see shortly, particularly simple
- But note that ULTRA can also be applied to other GP representations

Why Push (Elsewhere)?

- Highly expressive: data types, data structures, variables, conditionals, loops, recursion, modules, ...
- Elegant: minimal syntax and a simple, stack-based execution architecture
- Evolvable
- Extensible
- Supports several forms of meta-evolution

ULTRA

- Achieve uniformity by treating genomes as linear sequences, even if they are hierarchically structured
- Repair after transform to ensure structural validity

ULTRA Algorithm

- Linearize two parents
- Alternate according to *alternation rate*, subject to *alignment deviation*
- Uniformly mutate, subject to *mutation rate*
- Repair

Parents:

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

(1 (2 (3 4) 5) 6)

Result of alternation:

(a b 2 (3 4 d)) 6)

Result of repair:

(a (b 2 (3 4 d)) 6)

Problems

- Bioavailability
- Pagie-1
- Factorial
- 6-Multiplexer

Parameters

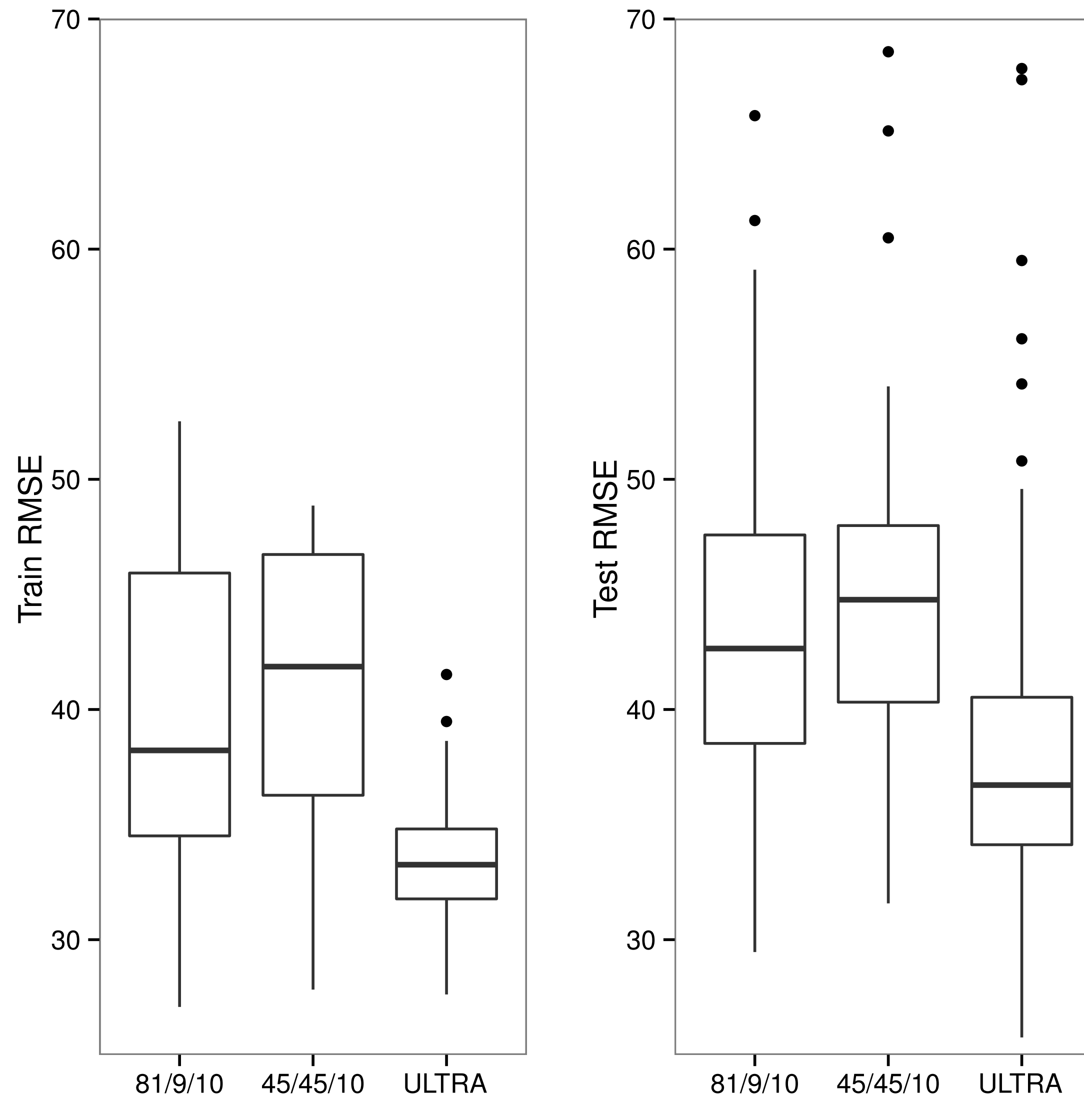
Problem	Bioavailability	Pagie-1	Factorial	MUX6
Runs Per Condition	200	100	100	100
Population Size	500	1000	1000	500
Max Generations	100	1000	500	200
Max Program Size	500	500	500	200
Max Inital Program Size	500	500	100	200
Max Size for Mutation Code	50	50	20	20
Parent Selection Tournament Size	7	7	Lexicase	7

Problem	Bioavailability, Pagie-1, MUX6	Factorial
ULTRA Mutation Rate	0.01	0.05
ULTRA Alternation Rate	0.01	0.05
ULTRA Alignment Deviation	10	10

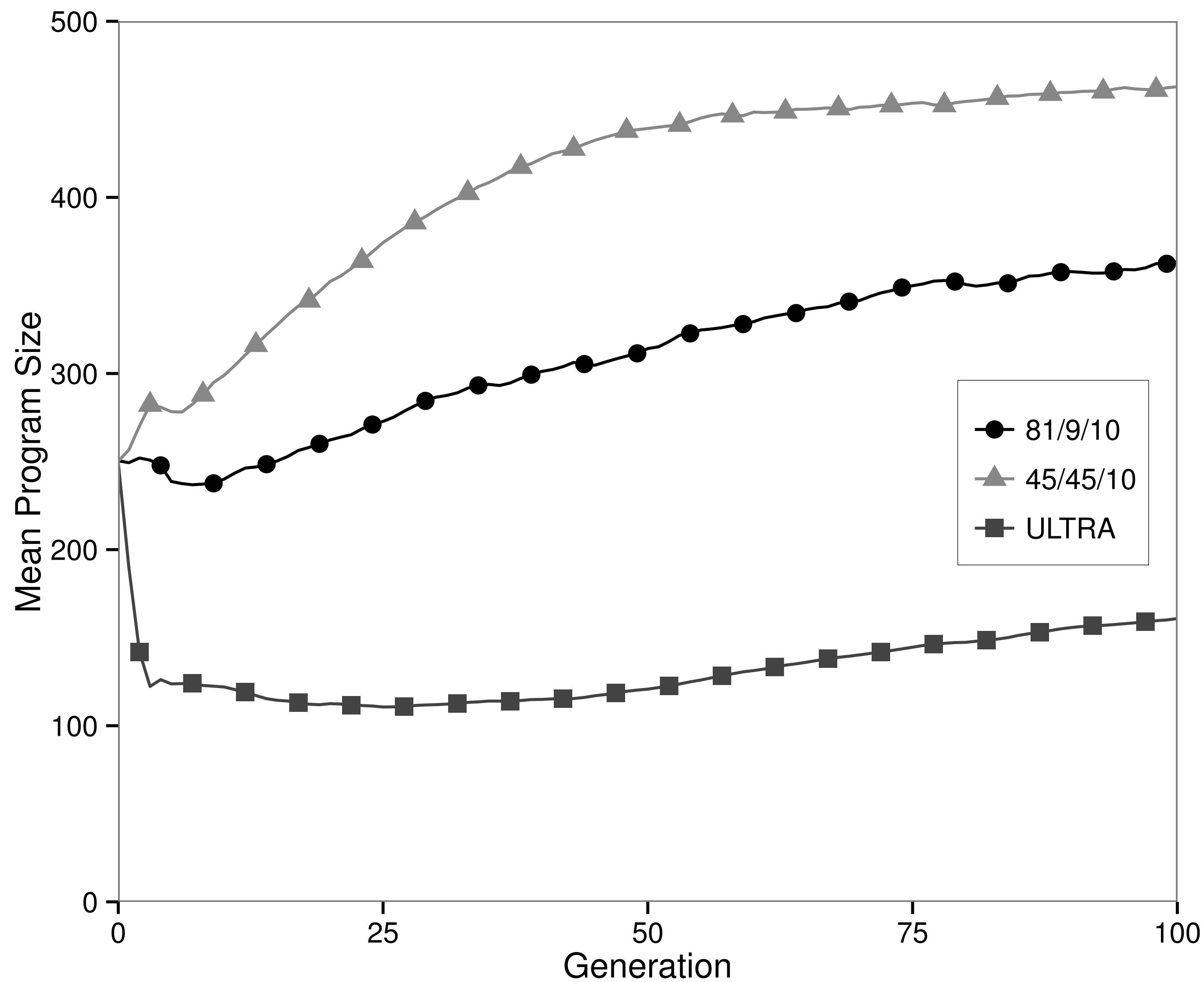
Bioavailability

- Floating-point predictive modeling
- Programs must predict the human oral bioavailability of a set of drug compounds given their molecular structure (Silva and Vanneschi, 2009, 2010)
- 241 floating point inputs, 1 floating point output, `add`, `sub`, `mult`, `div`
- Training: 70% of cases; Testing: 30% of cases
- Dataset is available online

Bioavailability Results



Bioavailability Sizes



Pagie-1

- Symbolic regression (Pagie and Hogeweg, 1997)
- $f(x,y) = \dfrac{1}{(1+x^{-4})} + \dfrac{1}{(1+y^{-4})}$
- Inputs for x and y taken uniformly from the range [−5, 5] in steps of 0.4, resulting in 676 fitness cases.
- `x`, `y`, `add`, `sub`, `mult`, `div`
- Used and recommended for GP benchmarking (Harper, 2012; McDermott et al, 2012; White et al, 2013)

100

Pa

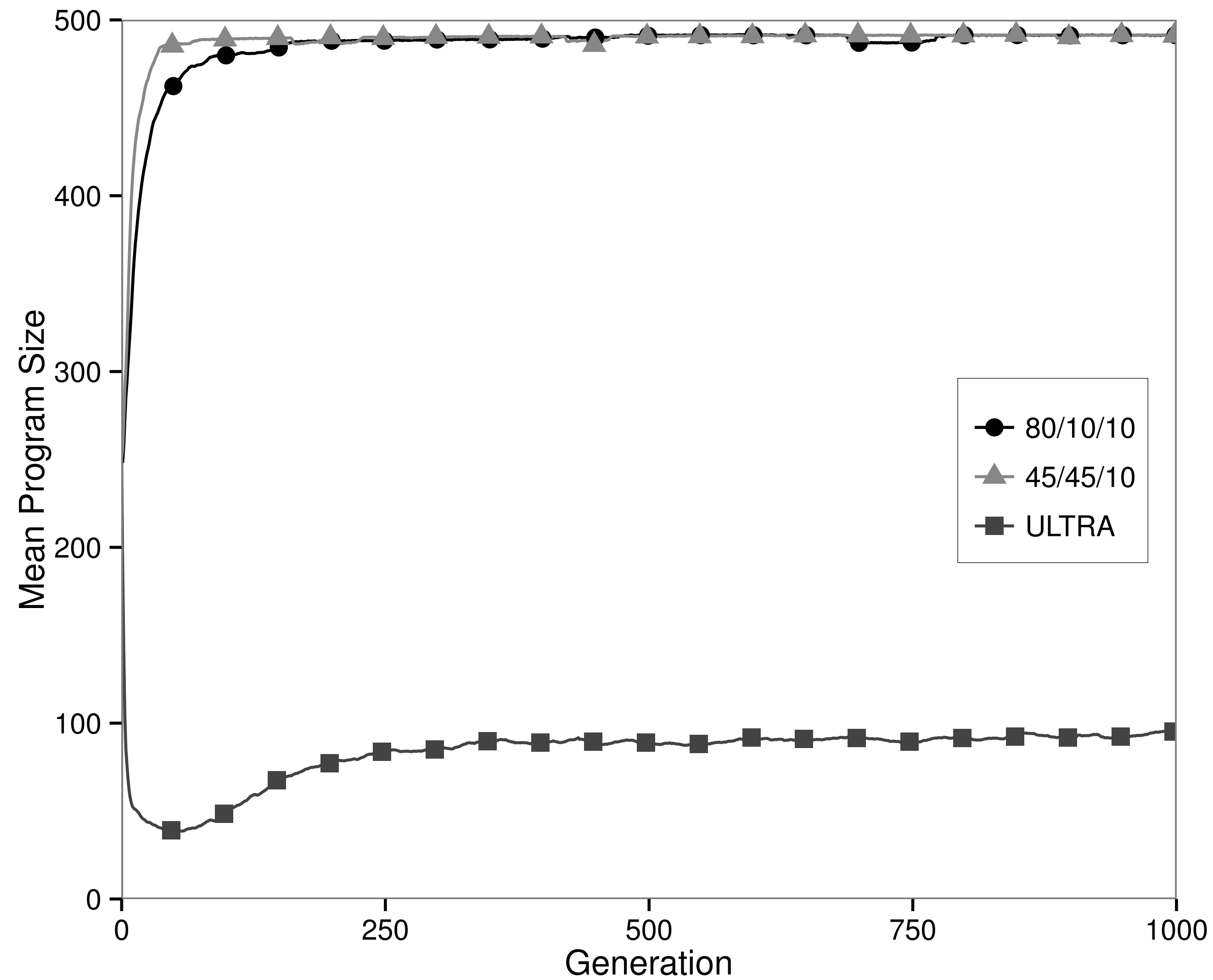
0

0 25 50 75

Generation

Operators	Successes	MBF
Subtree Replacement 80/10/10	0	0.363
Subtree Replacement 45/45/10	0	0.319
ULTRA	15	0.036

Pagie-1 Sizes



Factorial

- Integer symbolic regression
- 5 cases ranging from 1! = 1 to 5! = 120
- Parentheses are semantically relevant
- `in, and, dup, eq, frominteger, not, or, pop, rot, swap, add, div, dup, eq, fromboolean, gt, lt, mod, mult, pop, rot, sub, swap, if, k, noop, pop, rot, s, swap, when, y`
- Lexicase selection instead of tournament selection, because error magnitudes vary significantly across cases

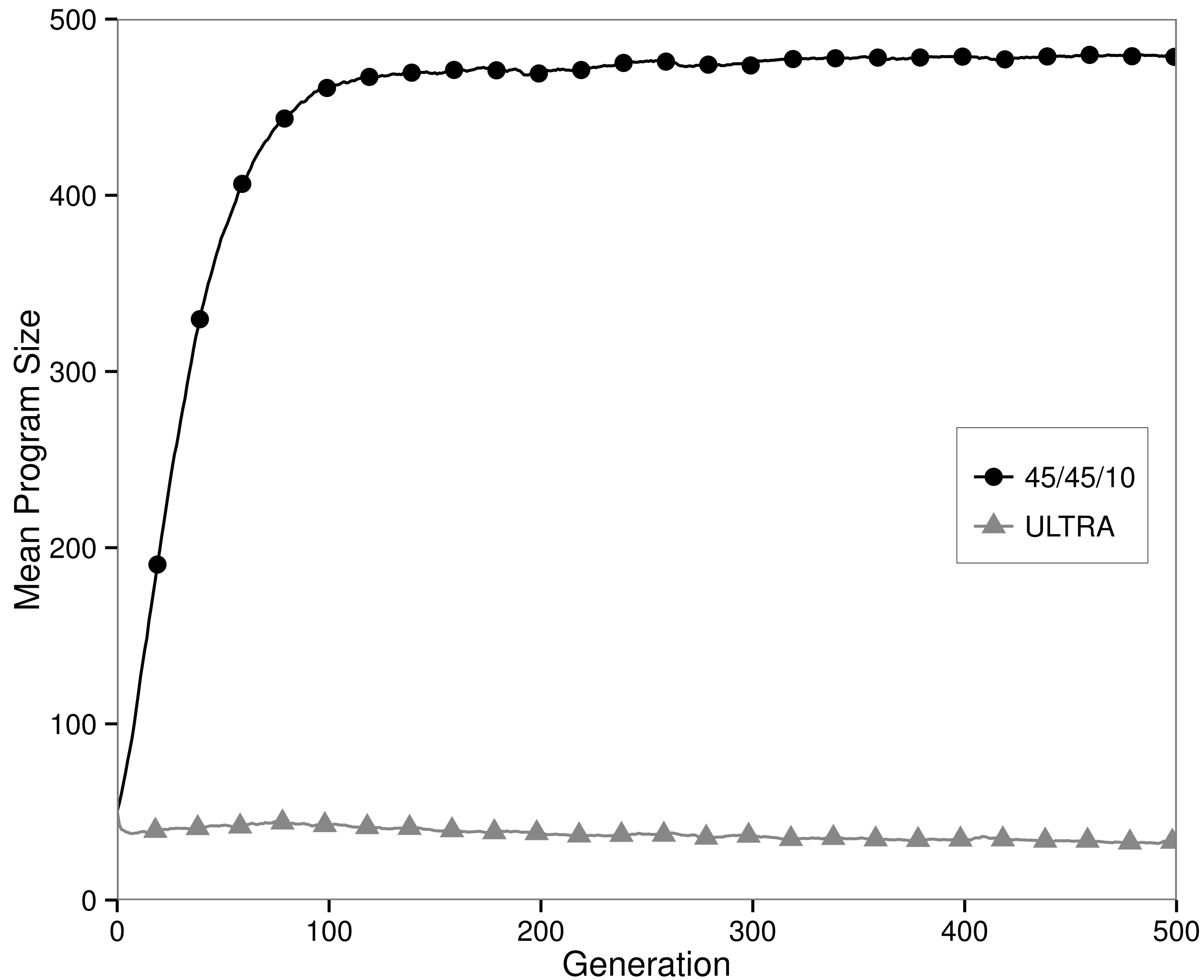
Lexicase Selection

- Each parent is selected by filtering the entire population, one one case at a time (in random order), keeping only the elite at each stage (Spector, 2012)
- Useful for "modal" problems, which require qualitatively different responses to different inputs
- All comparisons are "within case," so may be useful whenever cases are non-comparable

Fac

Operators	Successes	CE	MBF
Subtree Replacement 45/45/10	32	4,325,000	1.17
ULTRA	58	2,760,000	0.41

Factorial Sizes



6-Multiplexer

- Standard Boolean 6-multiplexer problem from (Koza, 1992)
- `a0`, `a1`, `d0`, `d1`, `d2`, `d3`, `if`, `and`, `or`, `not`

6-Multiplexer Results

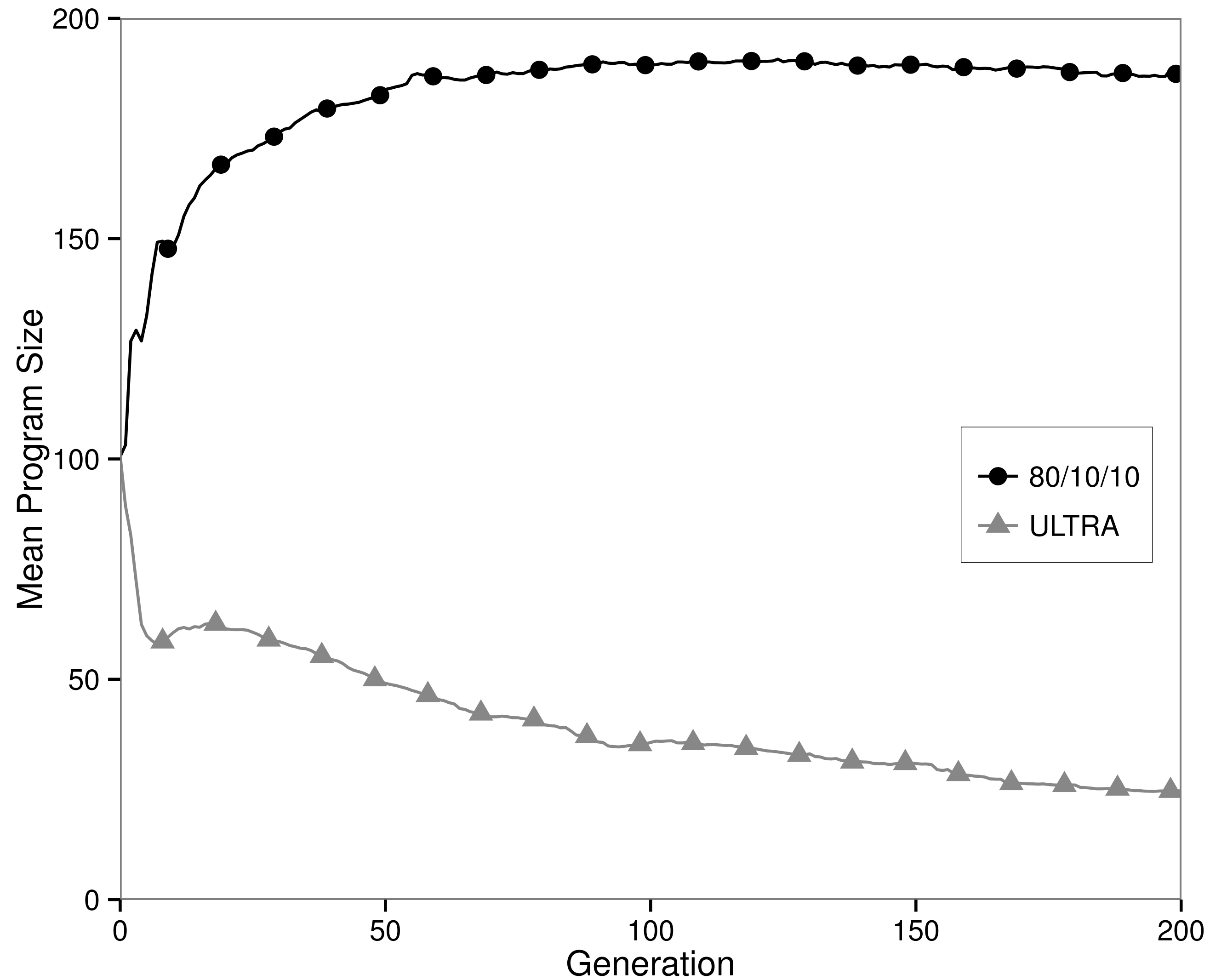
Operators	Successes	CE	MBF
Subtree Replacement 80/10/10	85	135,000	0.009
ULTRA	66	356,000	0.036

200

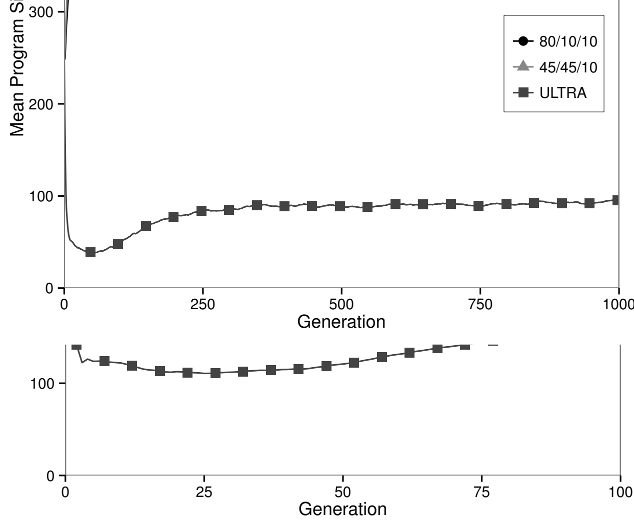
150

m Size

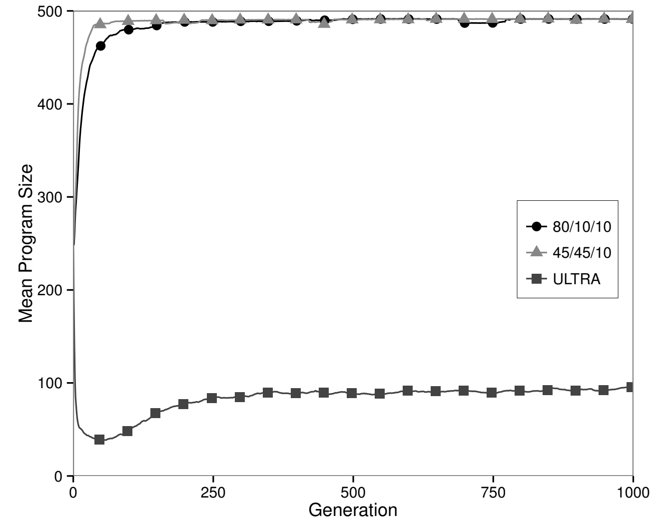
6-Multiplexer Sizes



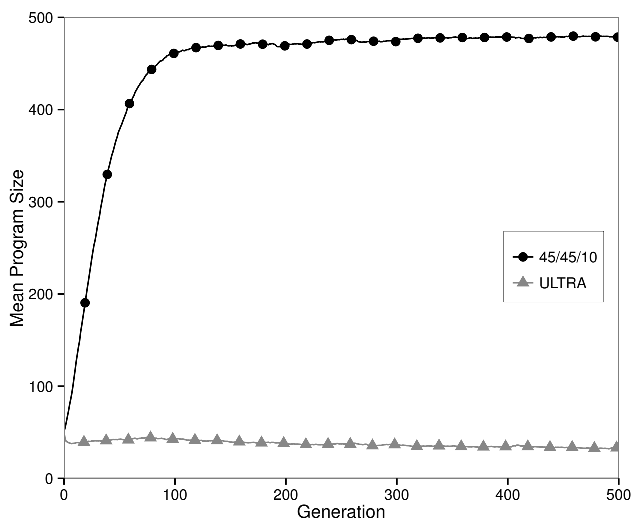
m Sizes



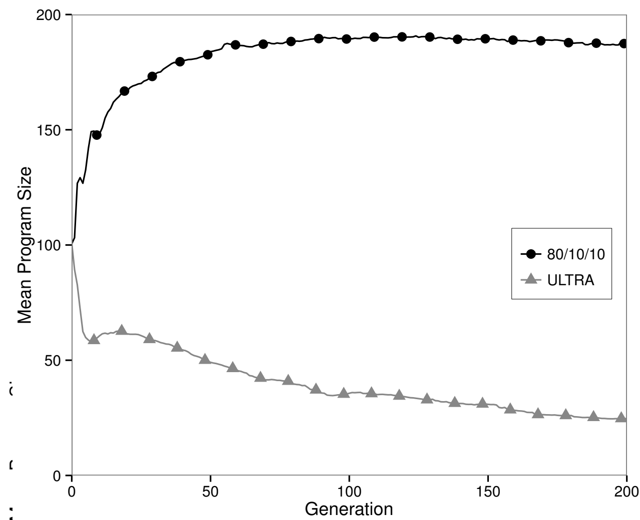
Bioavailability



Pagie-1



Factorial



6-Multiplexer

Conclusions

- Genetic operators that are in some senses "uniform" may perform much better than standard genetic operators in GP
- ULTRA provides an approach to ≈uniform operators based on linearization, alternation, and repair
- ULTRA is easy to implement for PushGP and produces significant improvements in problem solving power and size control
- With somewhat more effort ULTRA should be applicable to other program representations

Digital Multiplier

Table 5: Results on the 2-bit digital multiplier problem. Each condition used 100 runs. CE is the computational effort and MBF is the mean best fitness of the run. The last column gives 2-tailed p-values from unpaired t-tests that examine whether the MBF differs from that of lexicase + ULTRA, shown in the last row of the table.

Condition	Successes	CE	MBF	p-value
Normal	12	6,893,000	0.144	< 0.001
Lexicase	90	595,000	0.006	0.005
ULTRA	57	2,440,000	0.056	< 0.001
Lex+ULTRA	99	192,000	0.0006	-

Future Directions

- ULTRA with tree-based GP (Lisp-style symbolic expressions)
- ULTRA with grammatical evolution, cartesian genetic programming, and other GP representations
- Analysis and manipulation of uniformity properties
- Experimentation with ULTRA parameters