Expressive Genetic Programming

Tutorial
2012 Genetic and Evolutionary Computation Conference (GECCO-2012)

Lee Spector
School of Cognitive Science
Hampshire College
Amherst, MA 01002 USA
lspector@hampshire.edu
http://hampshire.edu/lspector

Copyright is held by the author/owner(s).
GECCO'12 Companion, July 7–11, 2012, Philadelphia, PA, USA.
ACM 978-1-4503-1178-6/12/07.

Instructor

Lee Spector is a Professor of Computer Science in the School of Cognitive Science at Hampshire College in Amherst, Massachusetts, and an adjunct professor in the Department of Computer Science at the University of Massachusetts, Amherst. He received a B.A. in Philosophy from Oberlin College in 1984 and a Ph.D. from the Department of Computer Science at the University of Maryland in 1992. His areas of teaching and research include genetic and evolutionary computation, quantum computation, and a variety of intersections between computer science, cognitive science, evolutionary biology, and the arts. He is the Editor-in-Chief of the journal *Genetic Programming and Evolvable Machines* (published by Springer) and a member of the editorial board of *Evolutionary Computation* (published by MIT Press). He is also a member of the SIGEVO executive committee and he was named a Fellow of the International Society for Genetic and Evolutionary Computation.

Tutorial Description (1)

The language in which evolving programs are expressed can have significant impacts on the problem-solving capabilities of a genetic programming system. These impacts stem both from the absolute computational power of the languages that are used, as elucidated by formal language theory, and from the ease with which various computational structures can be produced by random code generation and by the action of genetic operators. Highly expressive languages can facilitate the evolution of programs for any computable function using, when appropriate, multiple data types, evolved subroutines, evolved control structures, evolved data structures, and evolved modular program and data architectures. In some cases expressive languages can even support the evolution of programs that express methods for their own reproduction and variation (and hence for the evolution of their offspring).

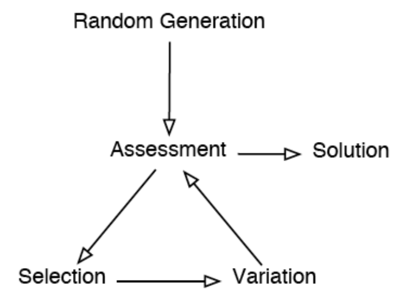
Tutorial Description (2)

This tutorial will begin with a comparative survey of approaches to the evolution of programs in expressive programming languages ranging from machine code to graphical and grammatical representations. Within this context it will then provide a detailed introduction to the Push programming language, which was designed specifically for expressiveness and specifically for use in genetic programming systems. Push programs are syntactically unconstrained but can nonetheless make use of multiple data types and express arbitrary control structures, supporting the evolution of complex, modular programs in a particularly simple and flexible way. The Push language will be described and ten years of Push-based research, including the production of human-competitive results, will be briefly surveyed. The tutorial will conclude with a discussion of recent enhancements to Push that are intended to support the evolution of complex and robust software systems.

Course Agenda

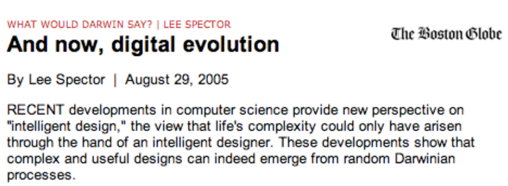
- Genetic Programming refresher
- Why evolve programs in expressive languages?
- Expressivity and evolvability
- Expressive trees, bits, graphs, grammars, stacks
- **Push**
- Expressing the future

Evolutionary Computation



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 (GP)

- Evolutionary computing to produce executable computer programs
- Programs are assessed by executing them
- Automatic programming; producing software
- Potential (?): evolve software at all scales, including and surpassing the most ambitious and successful products of human software engineering

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

A simple example

Given a set of data points, evolve a program that produces *y* from *x*.

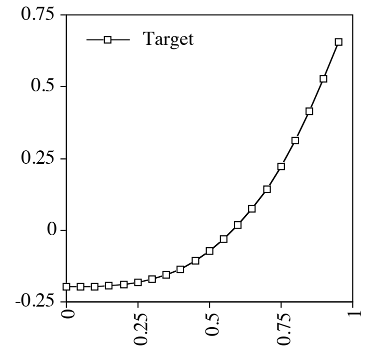
Primordial ooze: +, -, *, %, *x*, 0.1

Fitness = error (smaller is better)

GP Parameters

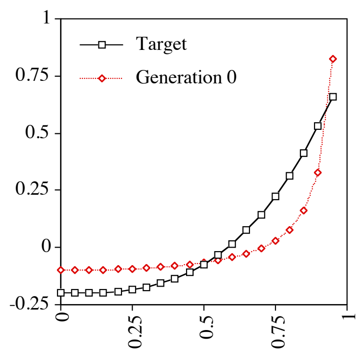
Maximum number of Generations: 51
Size of Population: 1000
Maximum depth of new individuals: 6
Maximum depth of new subtrees for mutants: 4
Maximum depth of individuals after crossover: 17
Fitness-proportionate reproduction fraction: 0.1
Crossover at any point fraction: 0.3
Crossover at function points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

Evolving $y = x^3 - 0.2$



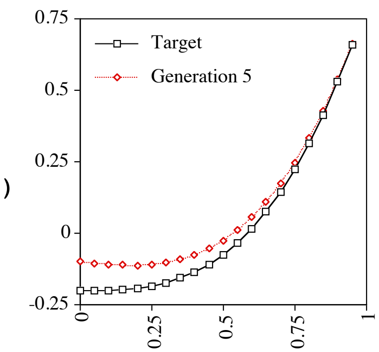
Best Program, Gen 0

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



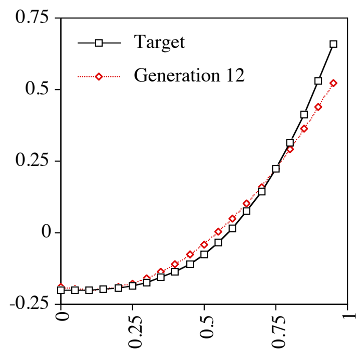
Best Program, Gen 5

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



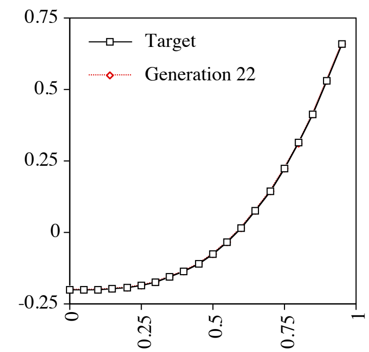
Best Program, Gen 12

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



Expressiveness

- Turing machine tables
- Lambda calculus expressions
- Register machine programs
- Partial recursive functions
- etc.

Pragmatics

The fact that a computation *can* be expressed in a formalism does not imply that a correct expression can be produced in that formalism by a human programmer or by an evolutionary process.

Tricks

- Cars, airplanes, and other complex engineered artifacts...
- Evolved biological organisms...
- Large-scale software systems...

  ... are each composed of millions of specialized parts, chosen, in each case, from a portfolio of domain-specialized components and processes.

Code Tricks

- Data abstraction and organization

  Data types, variables, name spaces, data structures, ...

- Control abstraction and organization

  Conditionals, loops, modules, threads, ...

Tricks via GP (1)

- Specialize GP techniques to directly support "code trick" syntax of human programming languages
- Strongly typed genetic programming
- Automatically defined functions
- Automatically defined macros
- Architecture altering operations

Tricks via GP (2)

- Specialize GP techniques to **indirectly** support "code trick" syntax from human programming languages
- Repair
- Genotype/phenotype mapping
- Grammars

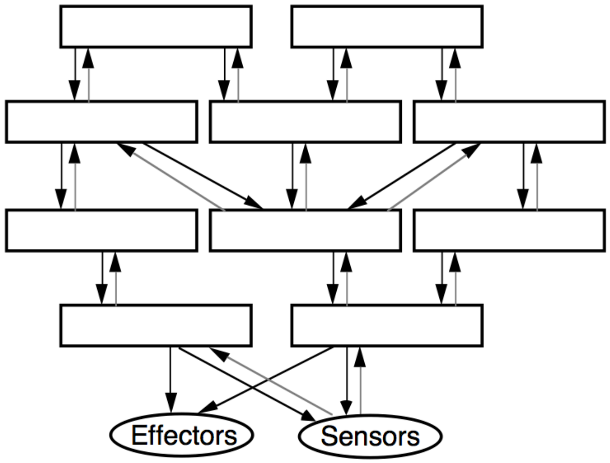
Tricks via GP (3)

- Develop new program encodings, represented most generally as graphs
- Develop analogs of code tricks for these representations
- Specialize GP techniques to directly or indirectly support "code trick" syntax for these new program encodings

Tricks via GP (4)

- Evolve programs in a minimal-syntax language that nonetheless supports a full range of "code tricks"
- For example: orchestrate data flows via stacks, not via syntax
- Push

Modularity is Everywhere



Modularity in Software

- Pervasive and widely acknowledged to be essential
- Modules may be functions, procedures, methods, classes, data structures, interfaces, etc.
- Modularity measures include coupling, cohesion, encapsulation, composability, etc.

Modules via GP

- Automatically-defined functions
- Automatically-defined macros
- Architecture-altering operations
- Module acquisition/encapsulation systems
- Grammars for languages with modules
- Instructions that build/execute modules

Evolving Modular Programs

With "automatically defined functions"

- All programs in the population have the same, pre-specified architecture
- Genetic operators respect that architecture
- Significant implementation costs
- Significant pre-specification
- Architecture-altering operations: more power and higher costs

ADMs

- Macros implement control structures
- ADMs can be implemented via small tweaks to any system that supports ADFs
- Similar pros and cons to ADFs, but provide additional expressive power

Control Structures (1)

Multiple evaluation

```
(defmacro do-twice (code)
  `(progn ,code ,code))

(do-twice (incf x))
```

Control Structures (2)

Conditional evaluation

```
(defmacro numeric-if (exp neg zero pos)
  `(if (< ,exp 0)
     ,neg
     (if (< 0 ,exp) ,pos ,zero)))

(numeric-if (foo) (bar) (baz) (bix))
```

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*)

Why Push?

- 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

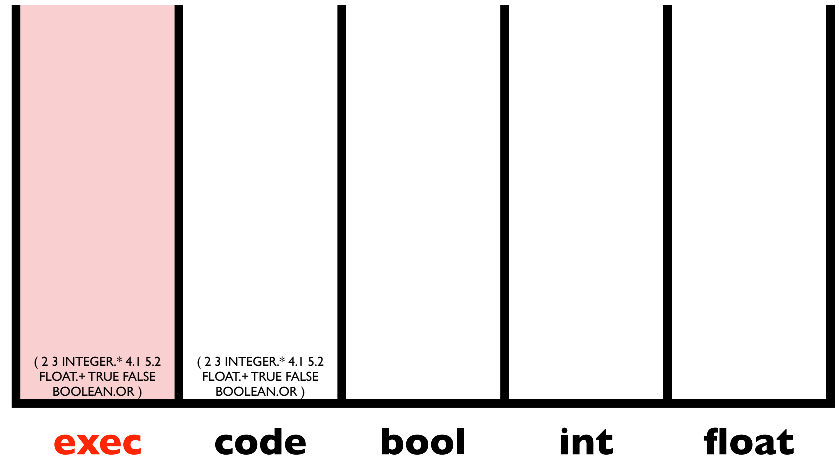
Sample Push Instructions

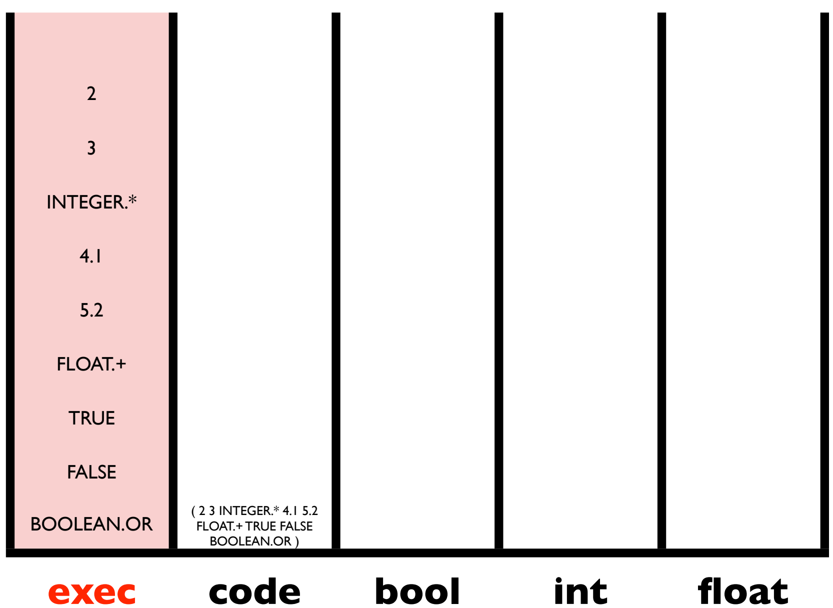
Stack manipulation instructions (all types)	POP, SWAP, YANK, DUP, STACKDEPTH, SHOVE, FLUSH, $=$
Math (INTEGER and FLOAT)	$+, -, /, *, >, <$, MIN, MAX
Logic (BOOLEAN)	AND, OR, NOT, FROMINTEGER
Code manipulation (CODE)	QUOTE, CAR, CDR, CONS, INSERT, LENGTH, LIST, MEMBER, NTH, EXTRACT
Control manipulation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

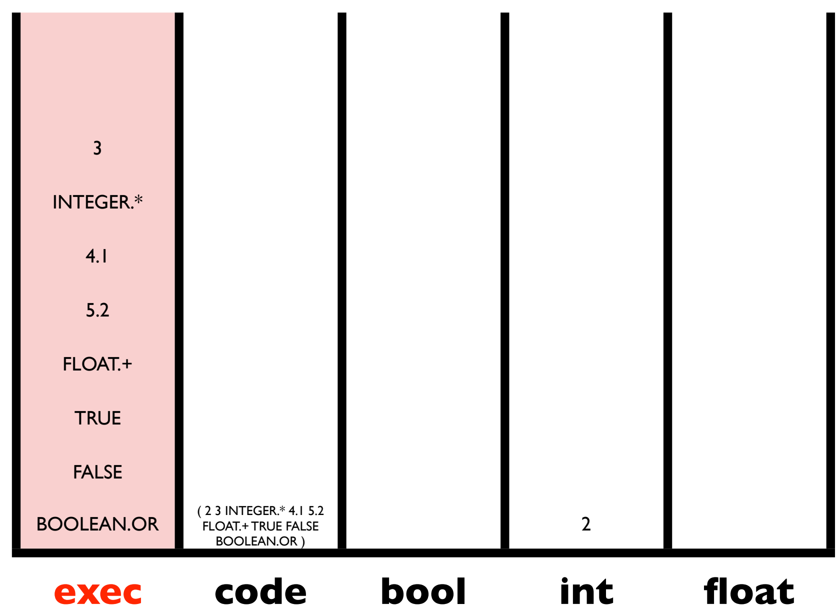
Push(3) Semantics

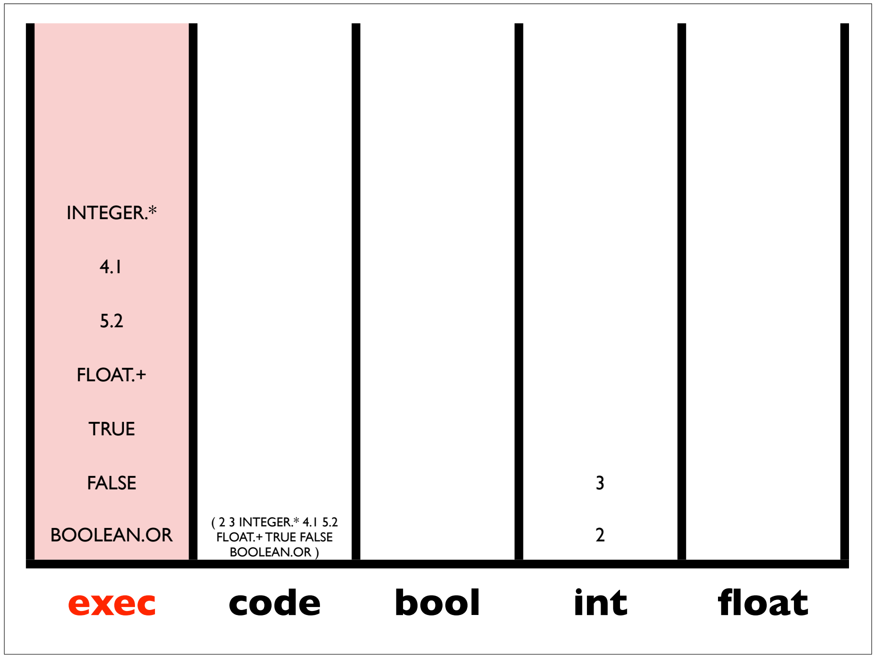
- To execute program P:
  1. Push P onto the `EXEC` stack.
  2. While the `EXEC` stack is not empty, pop and process the top element of the `EXEC` stack, E:
     (a) If E is an instruction: execute E (accessing whatever stacks are required).
     (b) If E is a literal: push E onto the appropriate stack.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

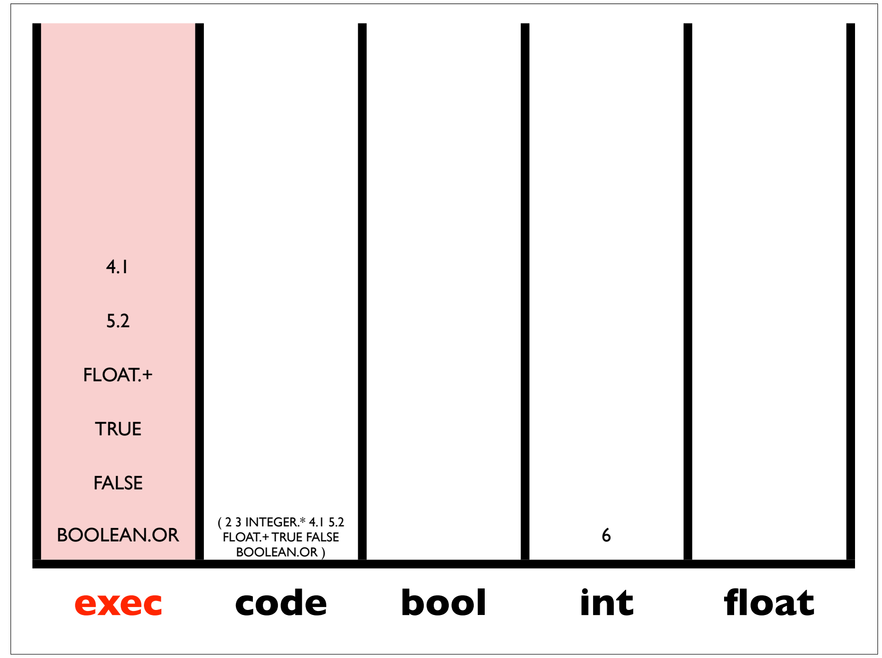
```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+
TRUE FALSE BOOLEAN.OR )
```

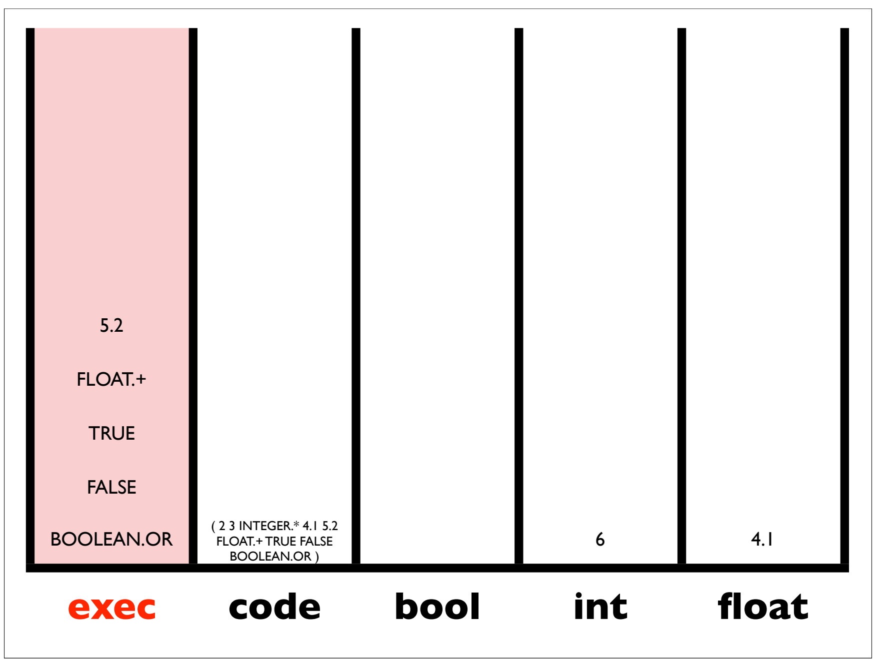


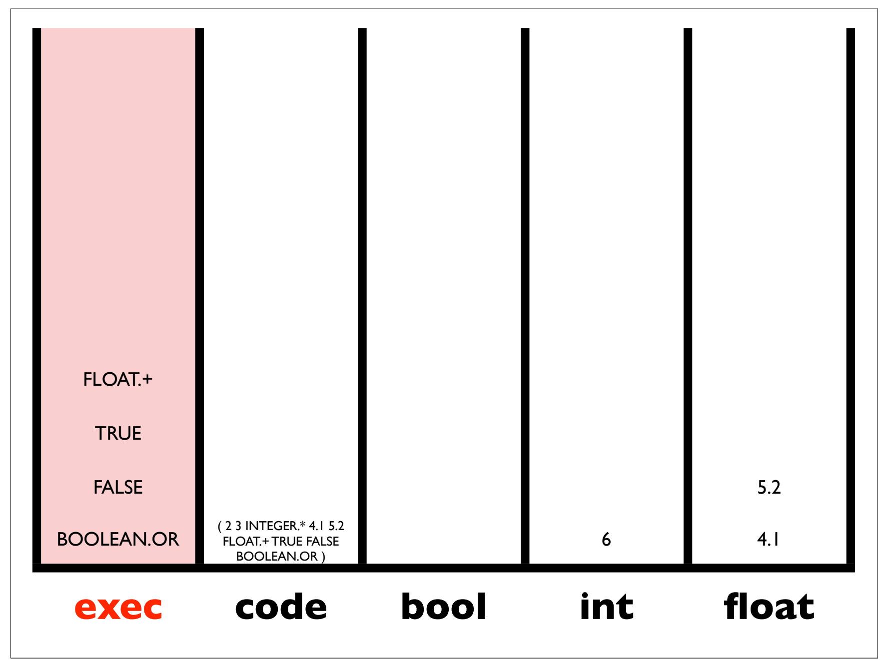





INTEGER.*
4.1
5.2
FLOAT.+
TRUE
FALSE
BOOLEAN.OR
(2 3 INTEGER.* 4.1 5.2 FLOAT.+ TRUE FALSE BOOLEAN.OR)
3
2
exec
code
bool
int
float


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

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

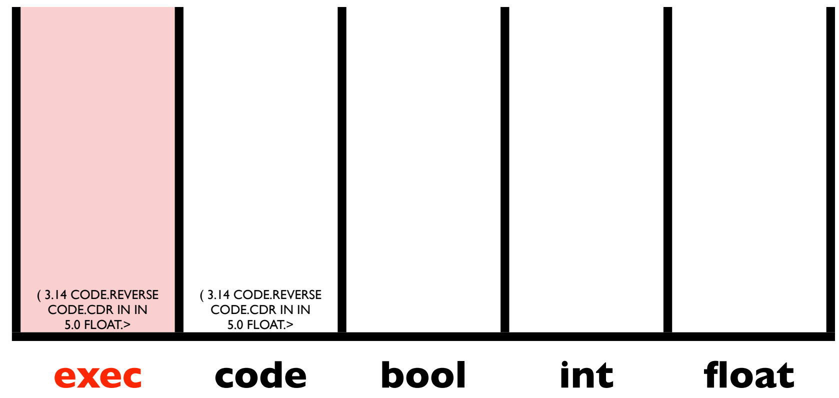
Same Results

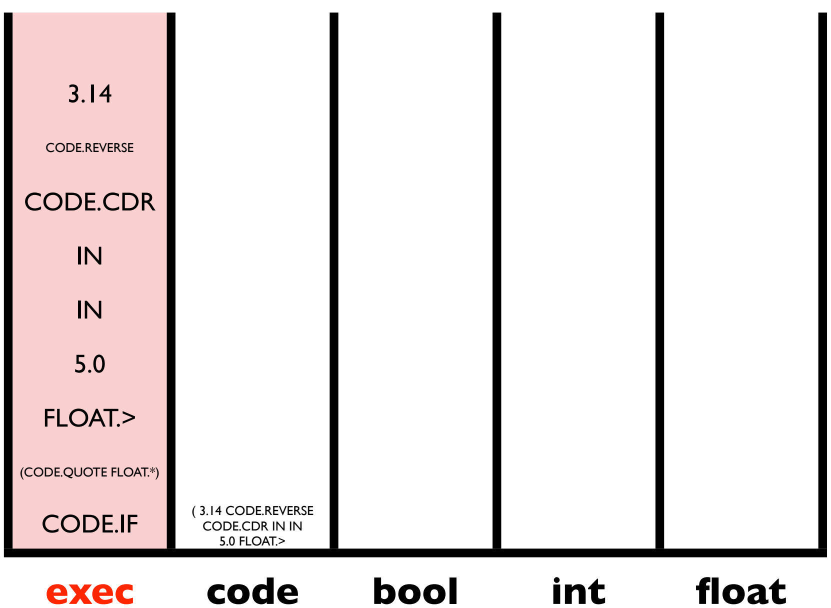
```
( 2 3 INTEGER.* 4.1 5.2 FLOAT.+
    TRUE FALSE BOOLEAN.OR )
```

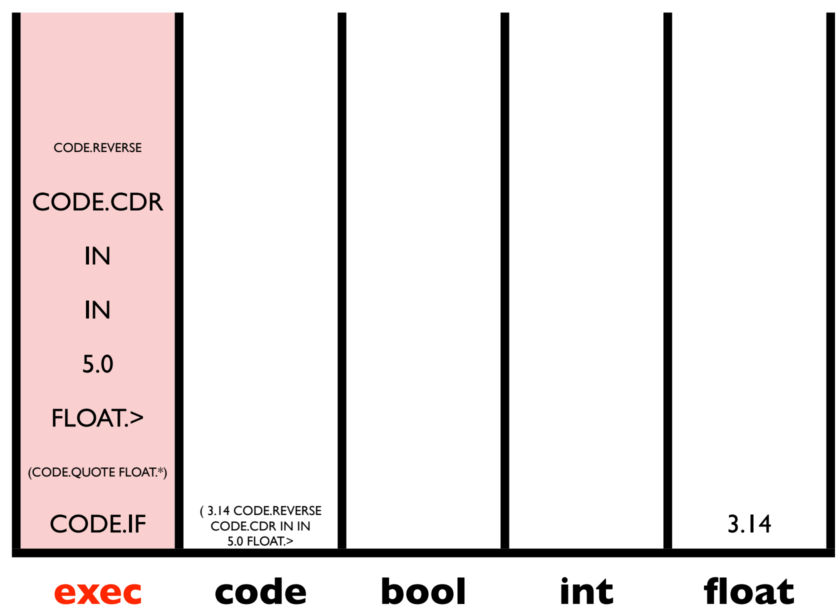
```
( 2 BOOLEAN.AND 4.1 TRUE INTEGER./ FALSE
  3 5.2 BOOLEAN.OR INTEGER.* FLOAT.+ )
```

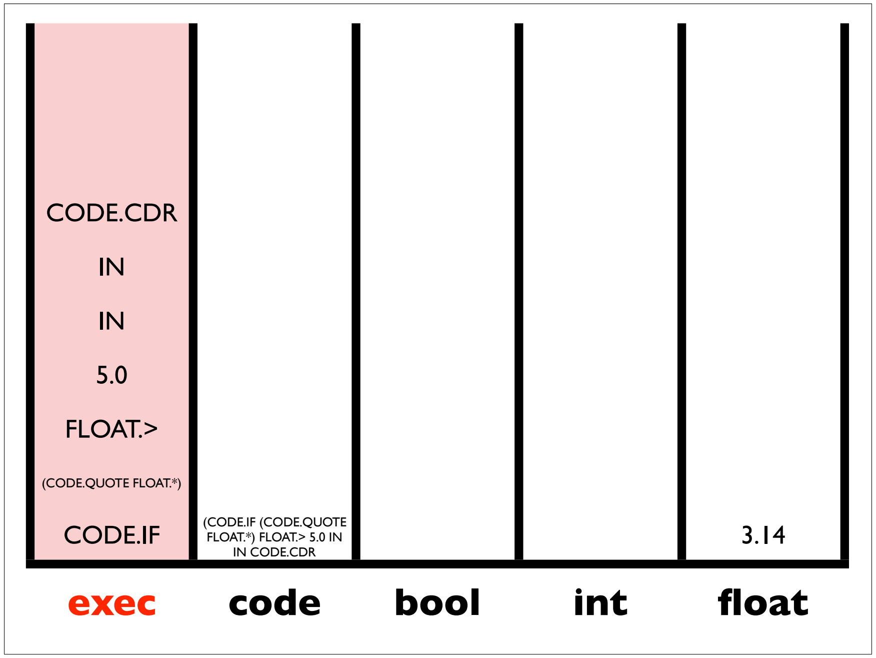
```
( 3.14 CODE.REVERSE CODE.CDR IN IN 5.0
FLOAT.> (CODE.QUOTE FLOAT.*) CODE.IF )
```

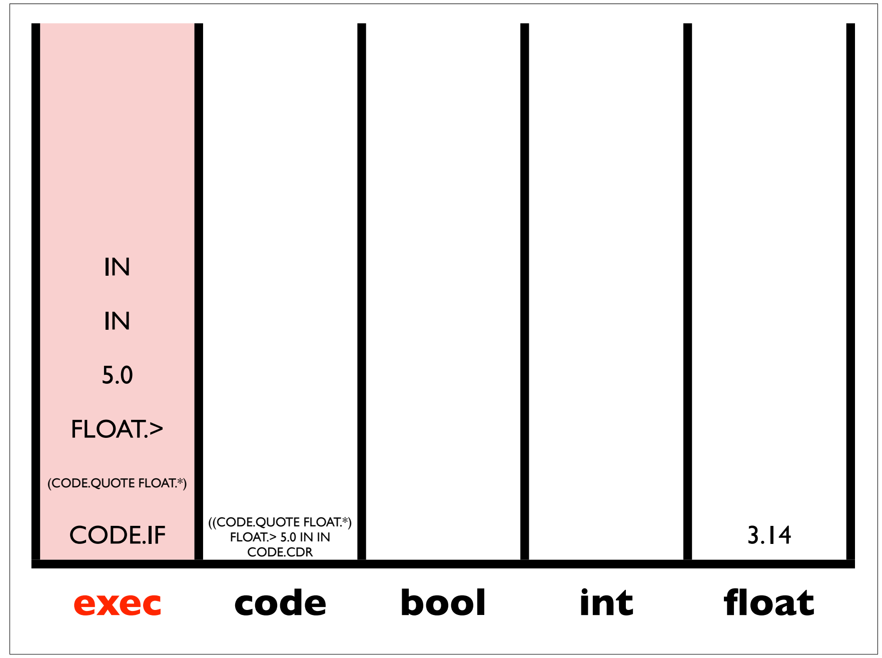
```
IN=4.0
```

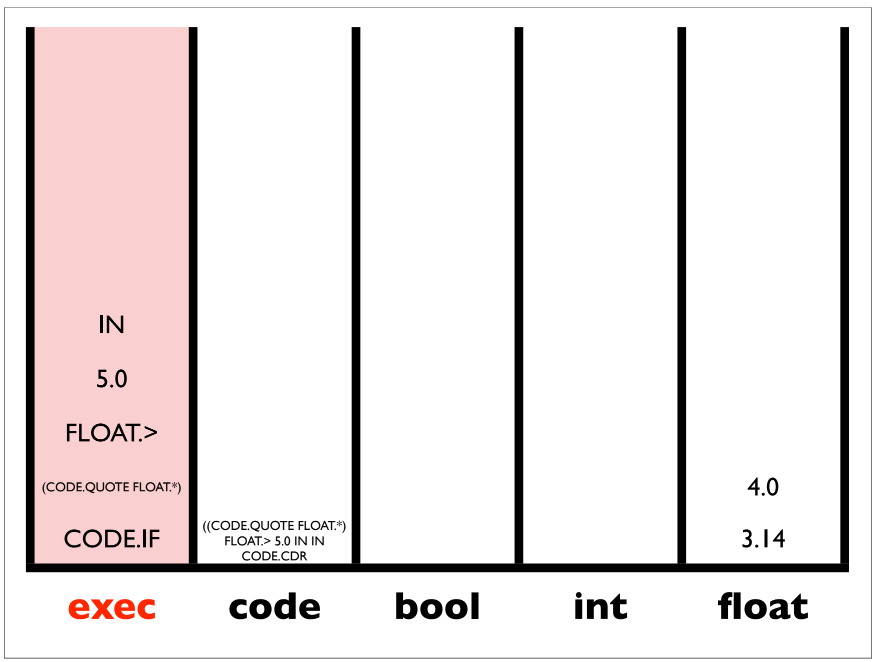


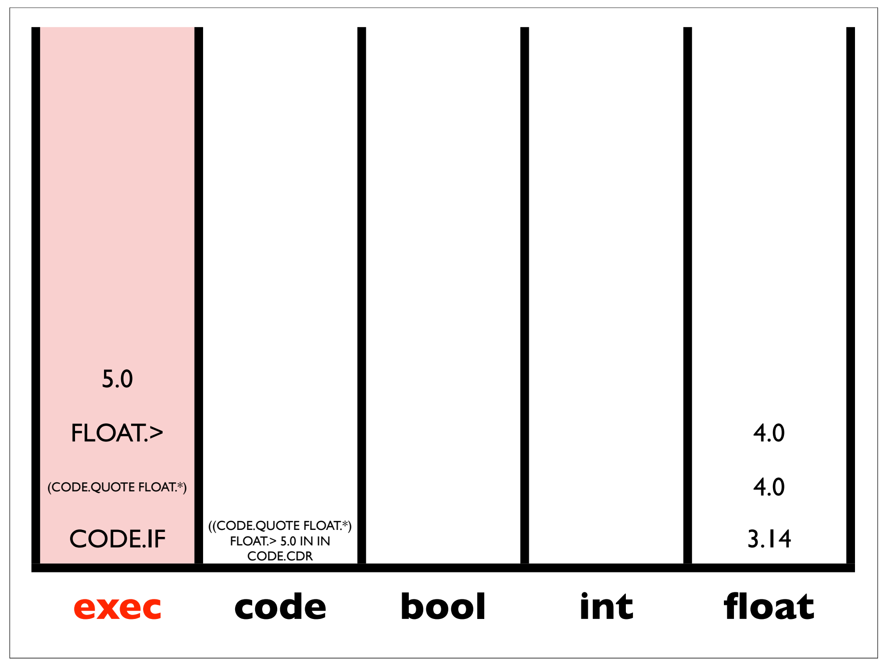


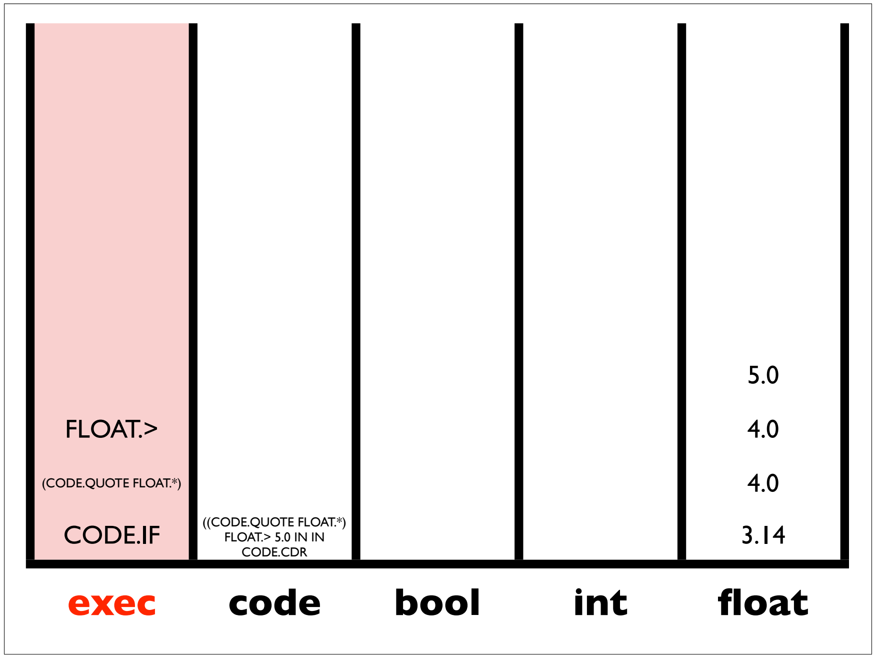


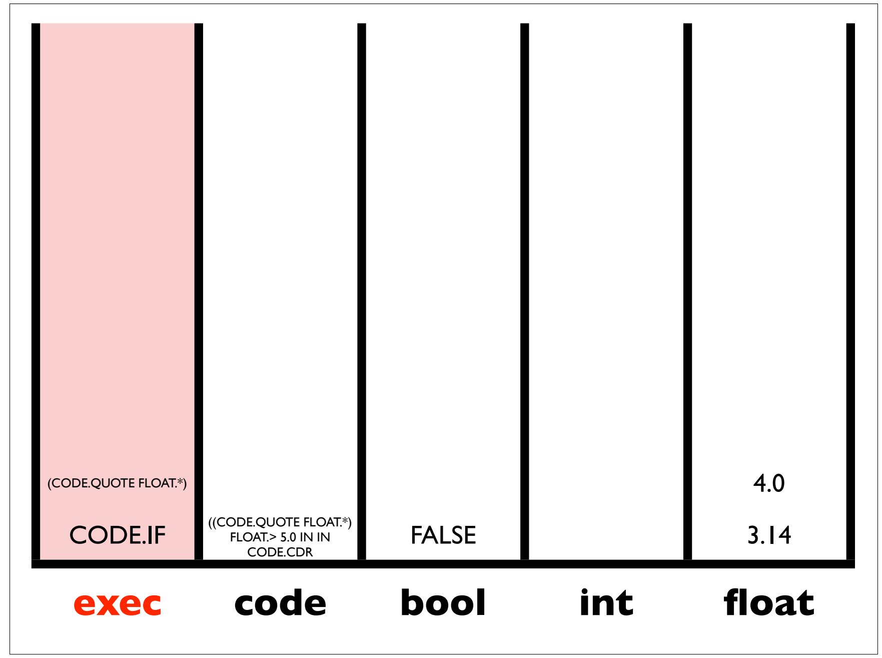

CODE.CDR
IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
(CODE.IF (CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
3.14
exec
code
bool
int
float

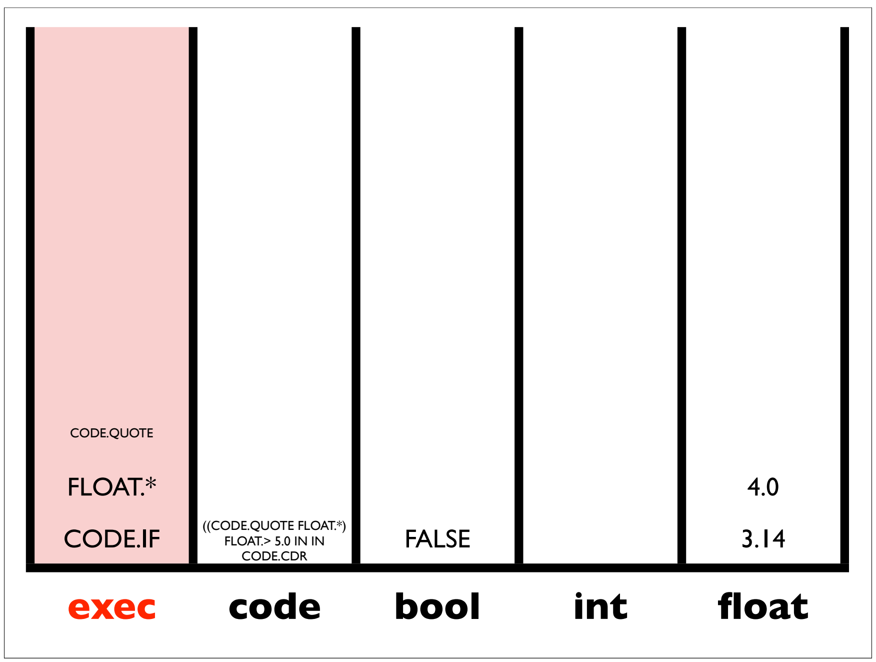

IN
IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
3.14
exec
code
bool
int
float

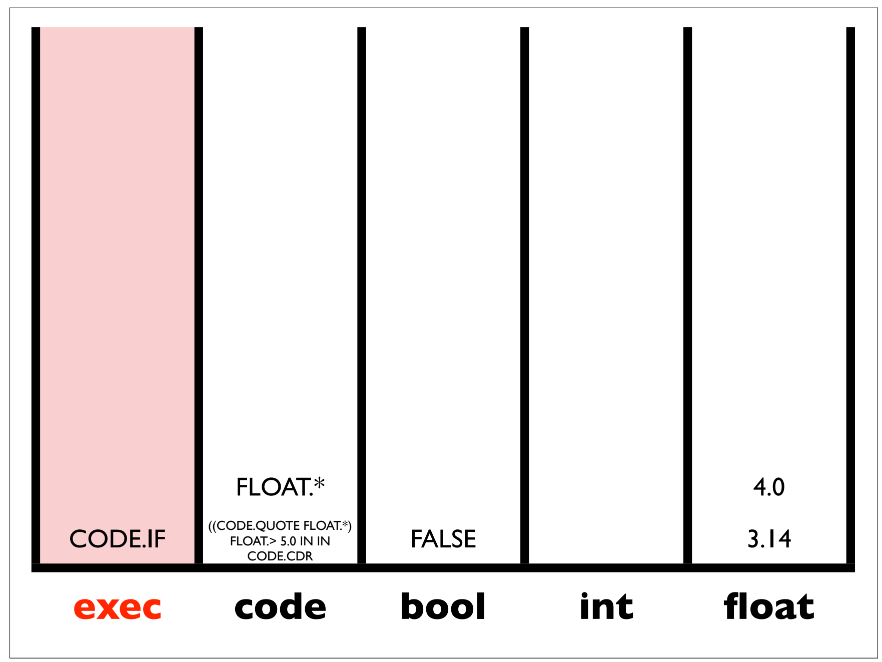

IN
5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
4.0
3.14
exec
code
bool
int
float

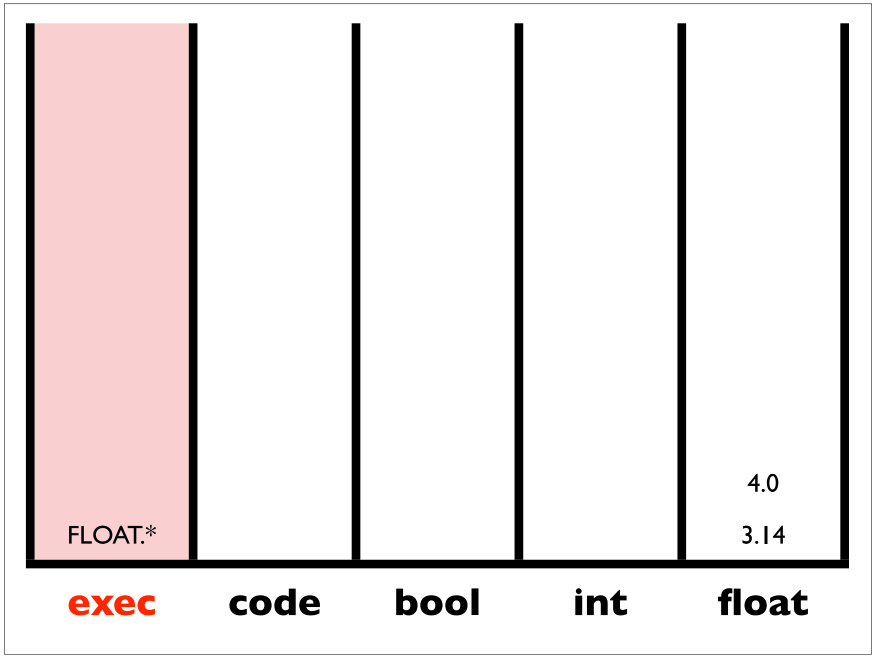

5.0
FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
4.0
4.0
3.14
exec
code
bool
int
float

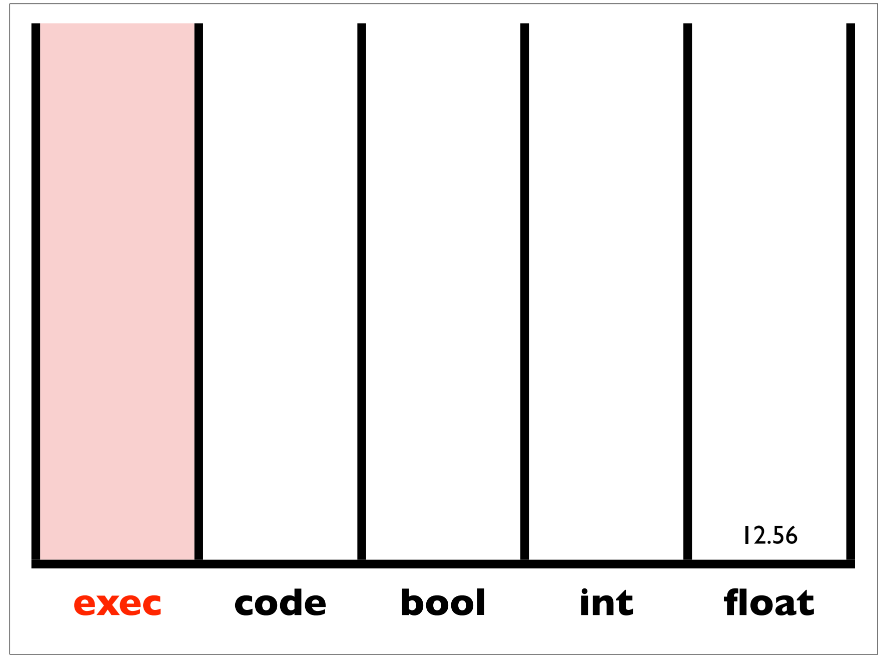

FLOAT.>
(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
5.0
4.0
4.0
3.14
exec
code
bool
int
float

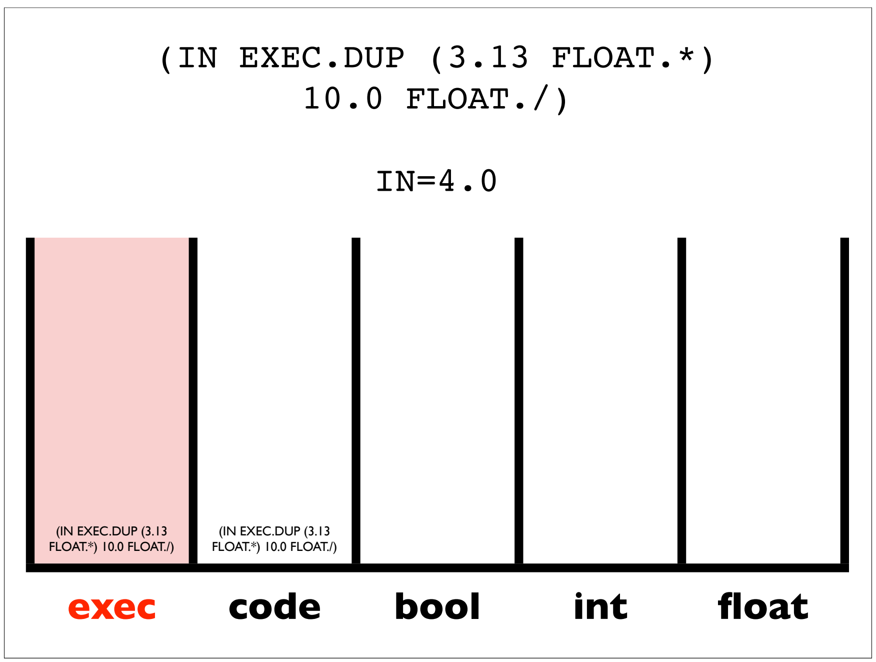

(CODE.QUOTE FLOAT.*)
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
FALSE
4.0
3.14
exec
code
bool
int
float

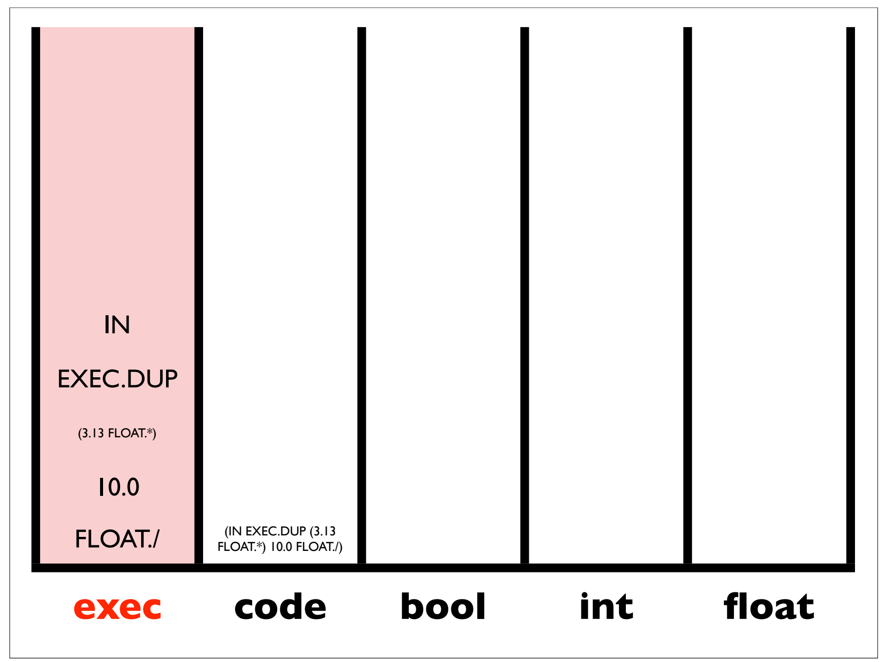

CODE.QUOTE
FLOAT.*
CODE.IF
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
FALSE
4.0
3.14
exec
code
bool
int
float

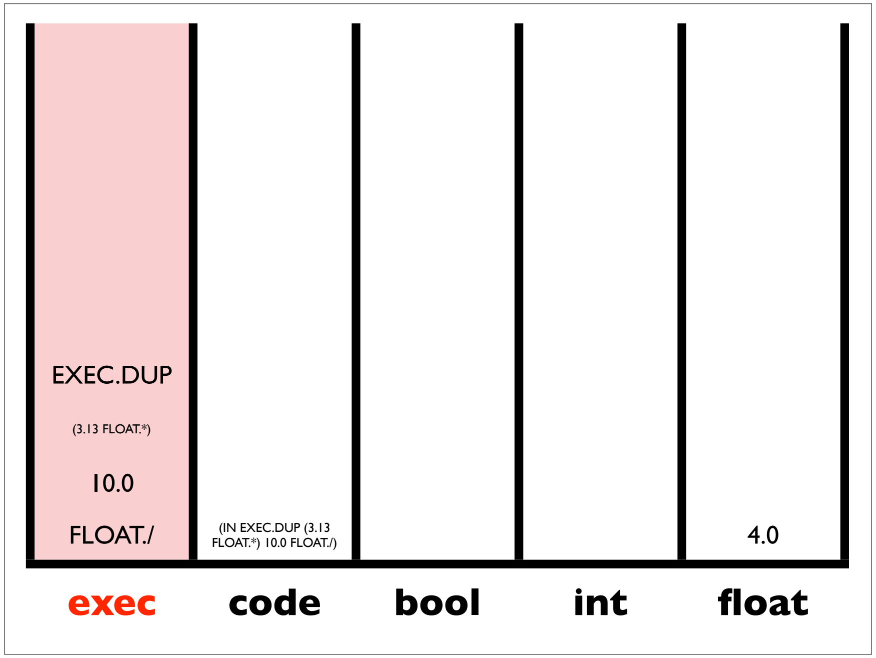

CODE.IF
FLOAT.*
((CODE.QUOTE FLOAT.*) FLOAT.> 5.0 IN IN CODE.CDR
FALSE
4.0
3.14
exec
code
bool
int
float

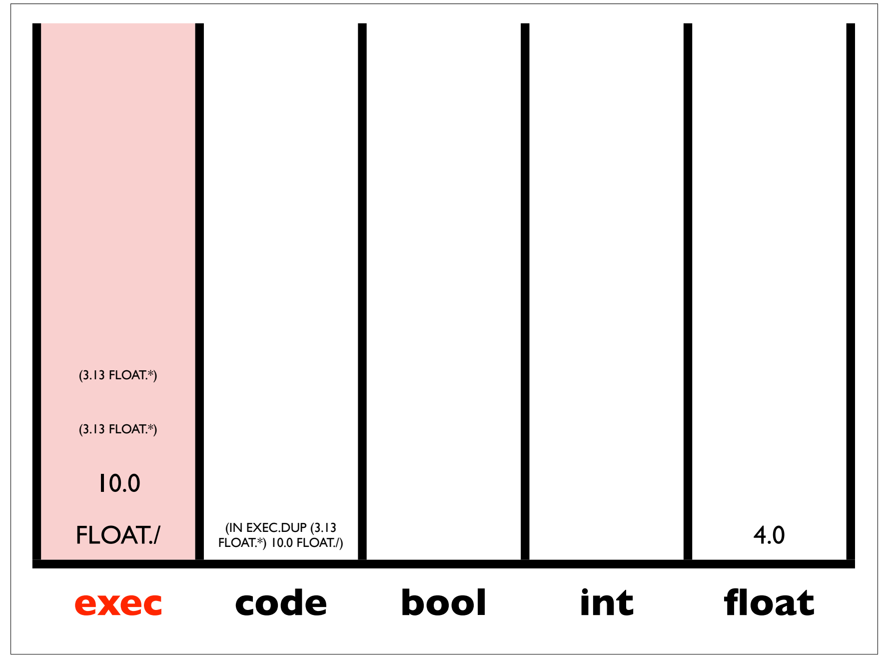

FLOAT.*
4.0
3.14
exec
code
bool
int
float

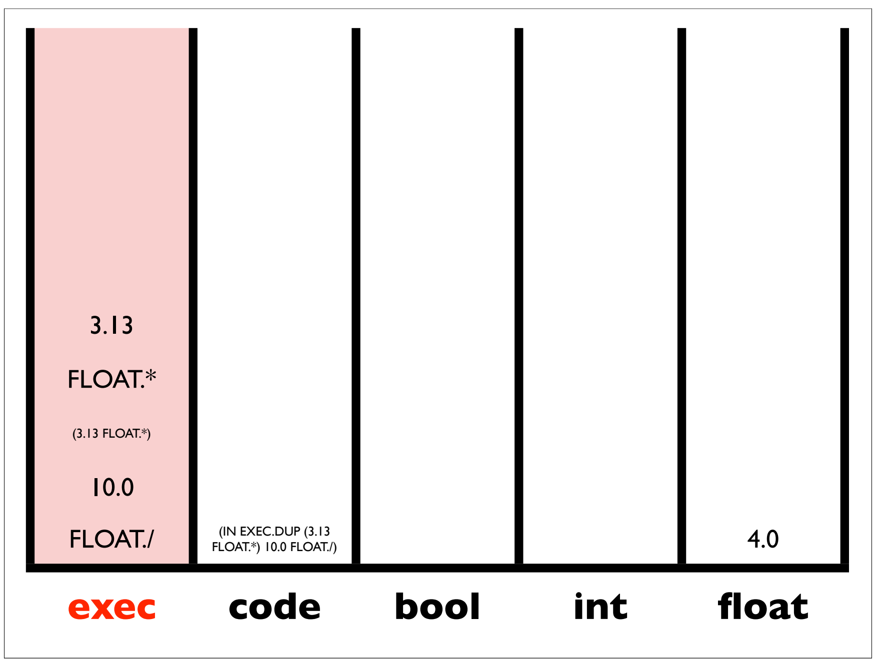

12.56
exec
code
bool
int
float

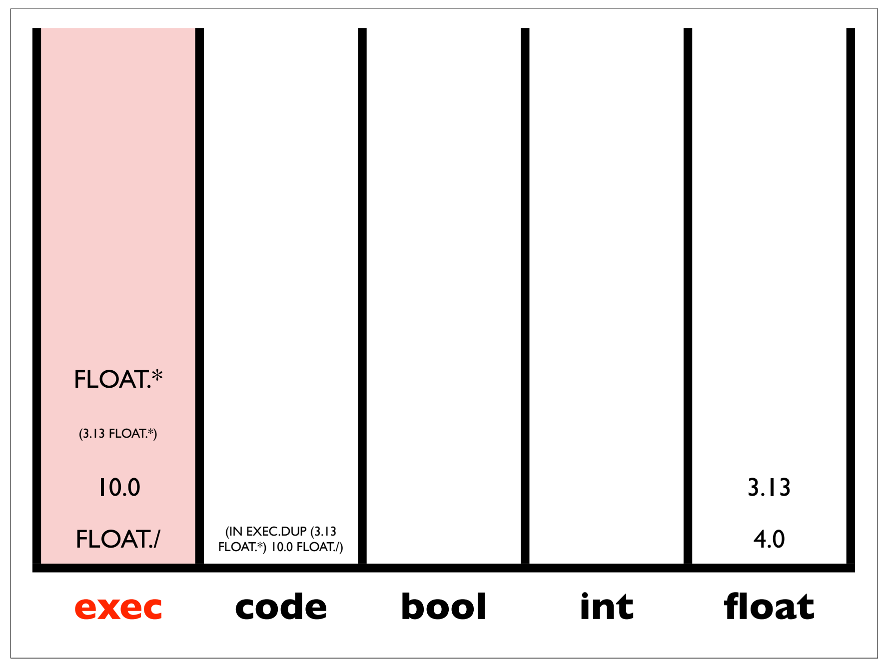

(IN EXEC.DUP (3.13 FLOAT.*)
10.0 FLOAT./)
IN=4.0
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
exec
code
bool
int
float

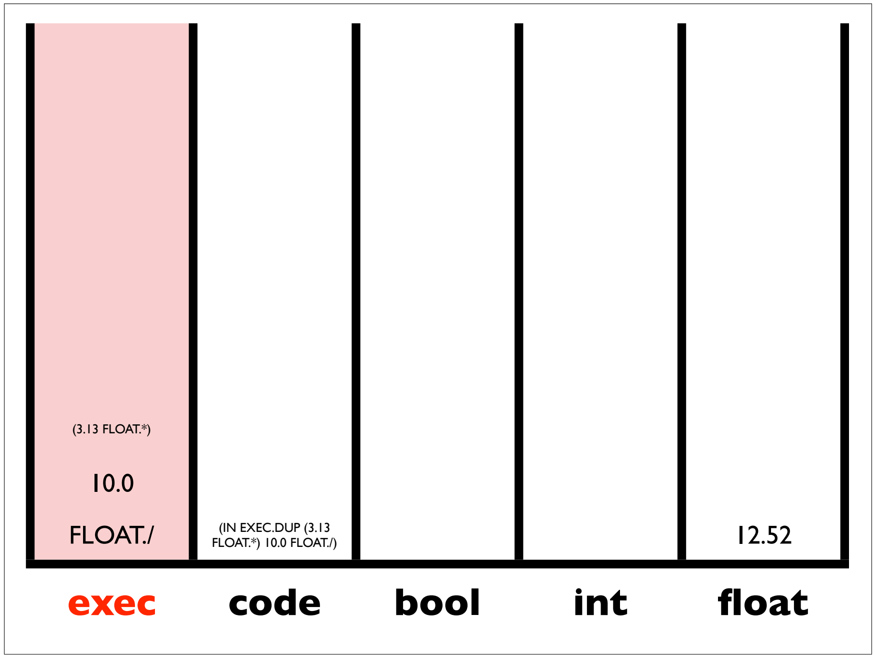

IN
EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
exec
code
bool
int
float

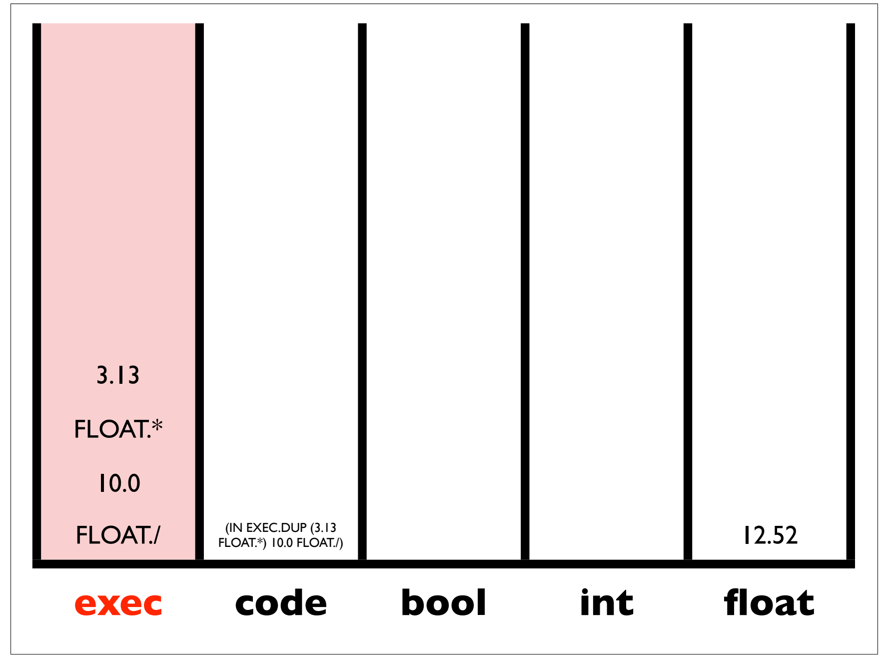

EXEC.DUP
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

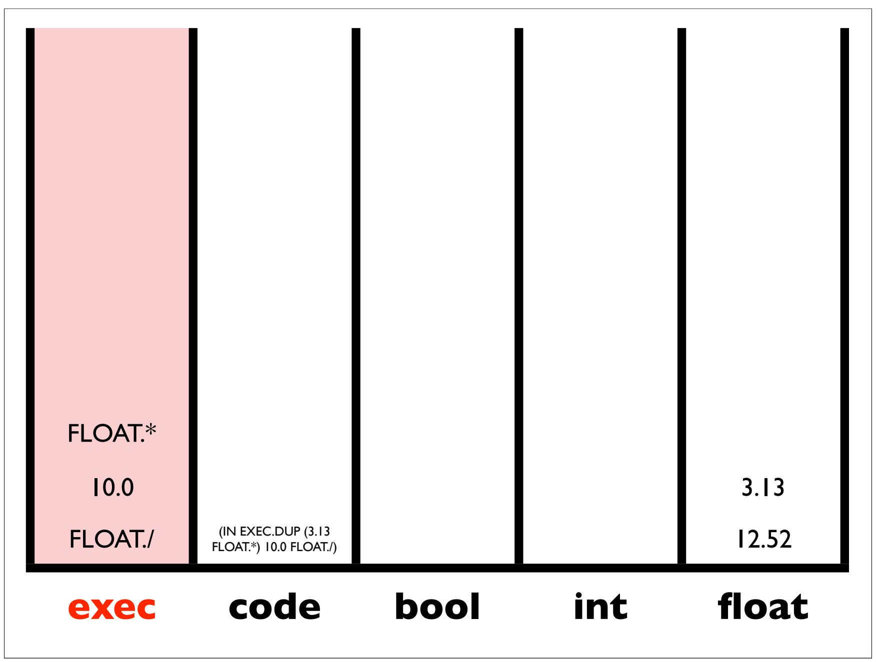

(3.13 FLOAT.*)
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

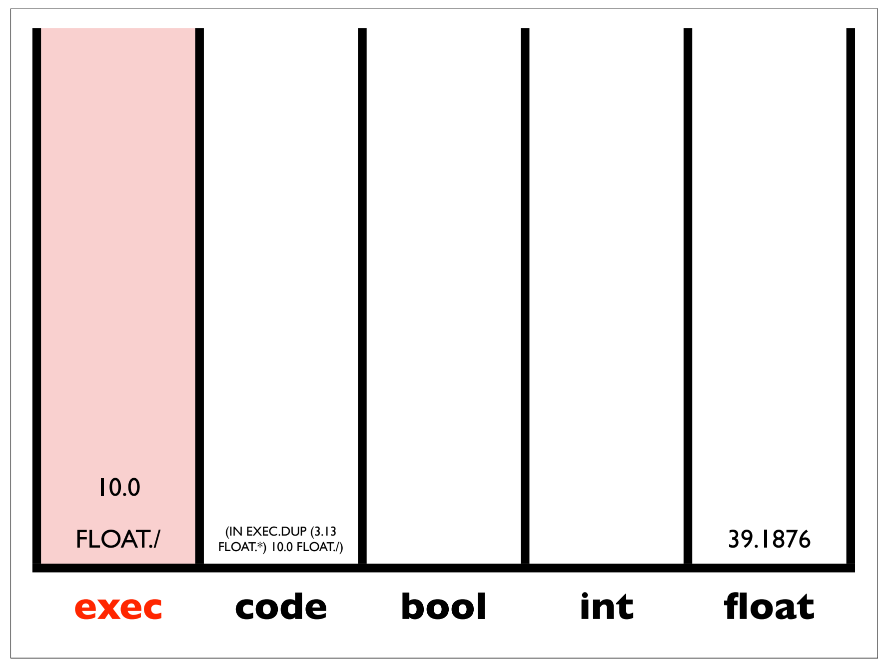

3.13
FLOAT.*
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
4.0
exec
code
bool
int
float

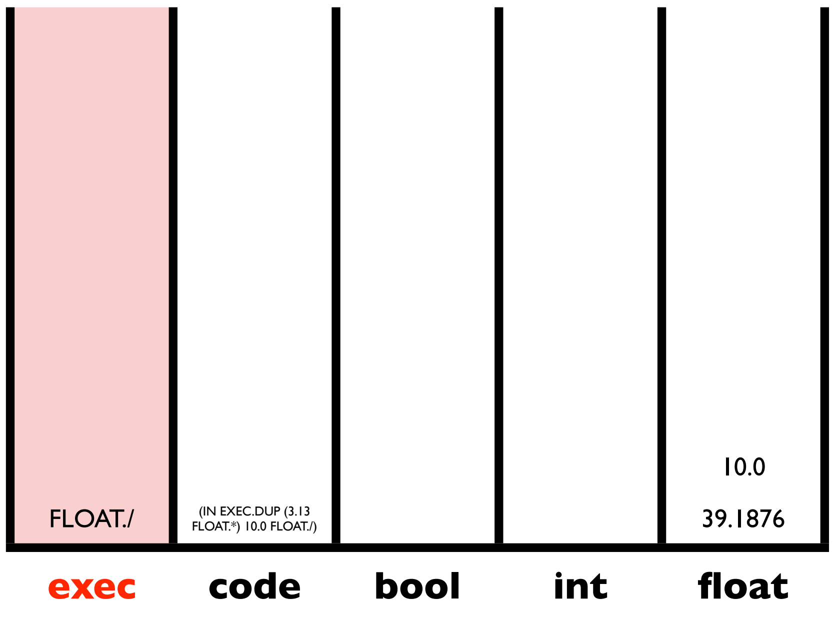

FLOAT.*
(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
3.13
4.0
exec
code
bool
int
float

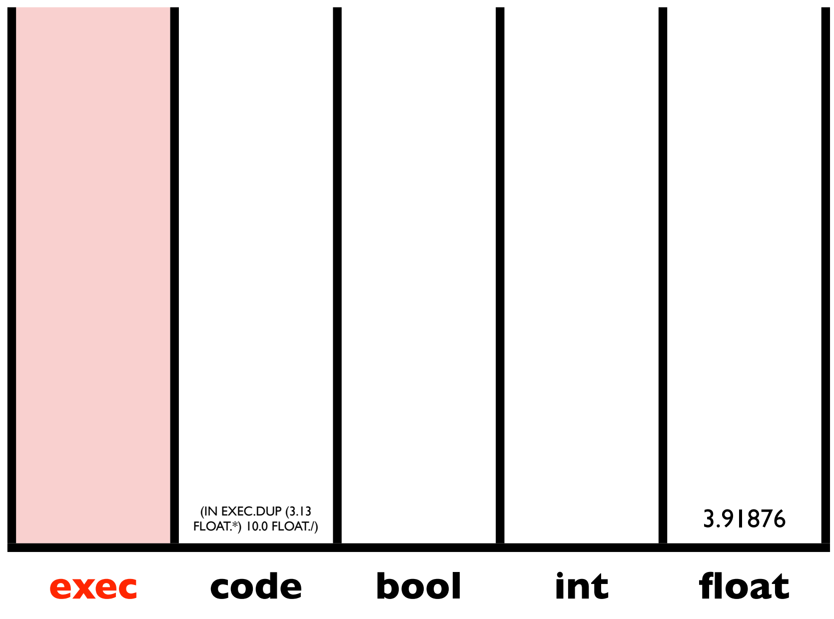

(3.13 FLOAT.*)
10.0
FLOAT./
(IN EXEC.DUP (3.13 FLOAT.*) 10.0 FLOAT./)
12.52
exec
code
bool
int
float


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





The Odd Problem

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

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:

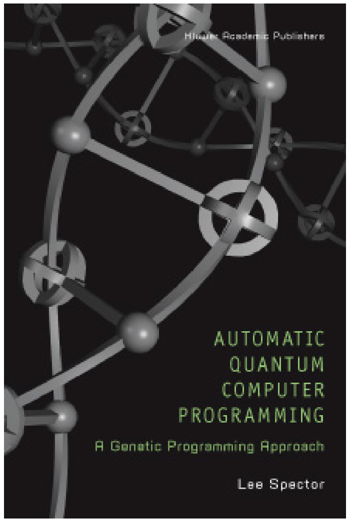
Make it randomly simpler

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



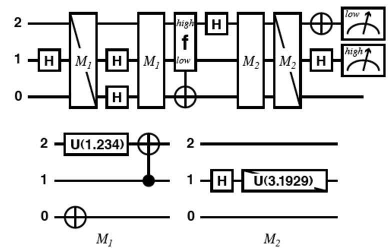


Figure 8.7. A gate array diagram for an evolved version of Grover's database search algorithm for a 4-item database. The full gate array is shown at the top, with M_1 and M_2 standing for the smaller gate arrays shown at the bottom. A diagonal line through a gate symbol indicates that the matrix for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

Humies 2004
GOLD MEDAL

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

Autoconstructive Evolution

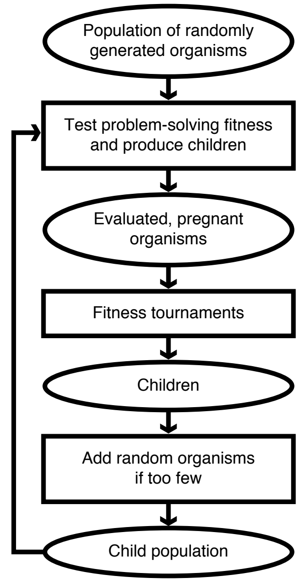
- Individuals make their own children
- Agents thereby control their own mutation rates, sexuality, and reproductive timing
- The machinery of reproduction and diversification (i.e., the machinery of evolution) evolves
- Radical self-adaptation

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)

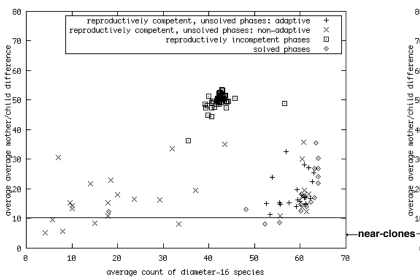
Pushpop

- 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

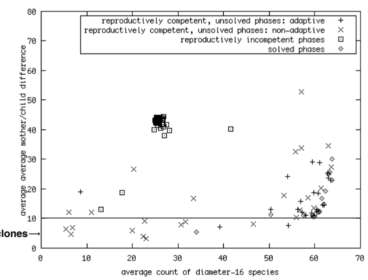


Species vs. Mother/Child Differences

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

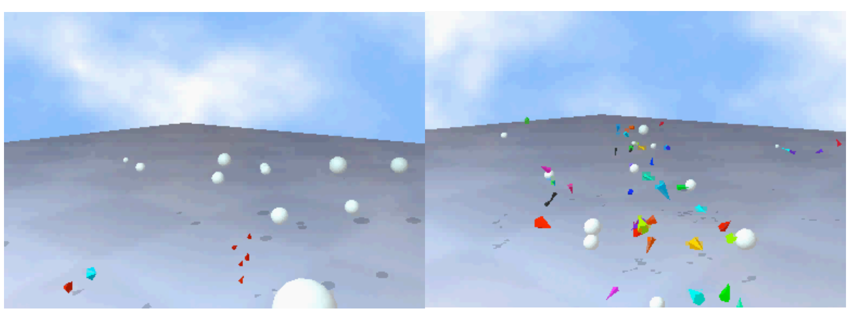
- In adaptive populations:
  - Species are more numerous
  - Diversification processes are more reliable
- Selection can promote diversity
- Provides a possible explanation for the evolution of diversifying reproductive systems

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)

Instruction(s)	Description
DUP, POP, SWAP, REP, =, NOOP, PULL, PULLDUP, CONVERT, CAR, CDR, QUOTE, ATOM, NULL, NTH, +, ∗, /, >, <, NOT, AND, NAND OR, NOR, DO*, IF	Standard Push instructions (See [11])
VectorX, VectorY, VectorZ, VPlus, VMinus, VTimes, VDivide, VectorLength, Make-Vector	Vector access, construction, and manipulation
RandI, RandF, RandV, RandC	Random number, vector, and code generators
SetServoSetpoint, SetServoGain, Servo	Servo-based persistent memory
Mutate, Crossover	Stochastic list manipulation (parameters from stacks)
Spawn	Produce a child with code from code stack
ToFood	Vector to energy source
FoodIntensity	Energy of energy source
MyAge, MyEnergy, MyHue, MyVelocity, MyLocation, MyProgram	Information about self
ToFriend, FriendAge, FriendEnergy, FriendHue, FriendVelocity, FriendLocation, FriendProgram	Information about closest agent of similar hue
ToOther, OtherAge, OtherEnergy, OtherHue, OtherVelocity, OtherLocation, OtherProgram	Information about closest agent of non-similar hue
FeedFriend, FeedOther	Transfer energy to closest agent of indicated category

SwarmEvolve 2.0



Winner, Best Paper Award, AAAA Track, GECCO-2003

AutoPush

- Goals:
  - Superior problem-solving performance
  - Tractable analysis
- Push3
- Asexual
- Children produced on demand (not during fitness testing)
- Constraints on selection and birth
- Still work in progress

Evolving Modular Programs

With Code Manipulation

- Transform code as data on "code" stack
- Execute transformed code with `code.do`, etc.
- Simple uses of modules can be evolved easily
- Does not scale well to large/complex systems

Evolving Modular Programs

With Execution Stack Manipulation

- Code queued for execution is stored on an "execution stack"
- Allow programs to duplicate and manipulate code that on the stack
- Example: `(3 exec.dup (1 integer.+))`
- More parsimonious, but same scaling issue

Evolving Modular Programs

With Named Modules

- Uses Push's "name" stack
- Example:

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

- Coordinating definitions/references is tricky
  and this never arises in evolution!

Module Identity

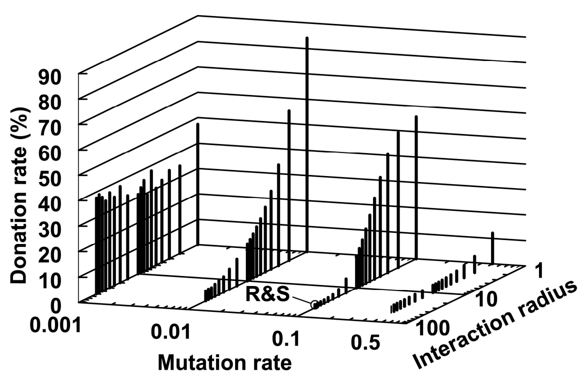
- How are modules recognized by other components of a system?
- Where do module identities come from?
- How can module identity co-evolve with modular architecture?

Holland's Tags

- Initially arbitrary identifiers that come to have meaning over time
- Matches may be inexact
- Appear to be present in some form in many different kinds of complex adaptive systems
- Examples range from immune systems to armies on a battlefield
- A general tool for the support of emergent complexity

Tag-Based Altruism

- Individuals have tags and tag-difference tolerances
- Donate when Δtags $\leq$ tolerance
- Riolo *et al.* (*Nature*, 2001) showed that tag-based altruism can evolve; Roberts & Sherratt (*Nature*, 2002) claimed it would not evolve under more realistic conditions



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

Evolving Modular Programs

With tags

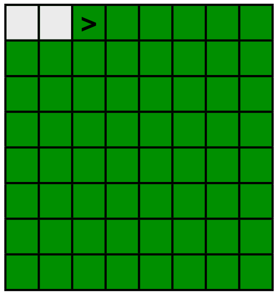
- Include instructions that tag code (modules)
- Include instructions that recall and execute modules by *closest matching* tag
- If a single module has been tagged then all tag references will recall modules
- The number of tagged modules can grow incrementally over evolutionary time
- **Expressive and evolvable**

Tags in Push

- Tags are integers embedded in instruction names
- Instructions like `tag.exec.123` tag values
- Instructions like `tagged.456` recall values by *closest matching* tag
- If a single value has been tagged then all tag references will recall (and execute) values
- The number of tagged values can grow incrementally over evolutionary time

Lawnmower Problem

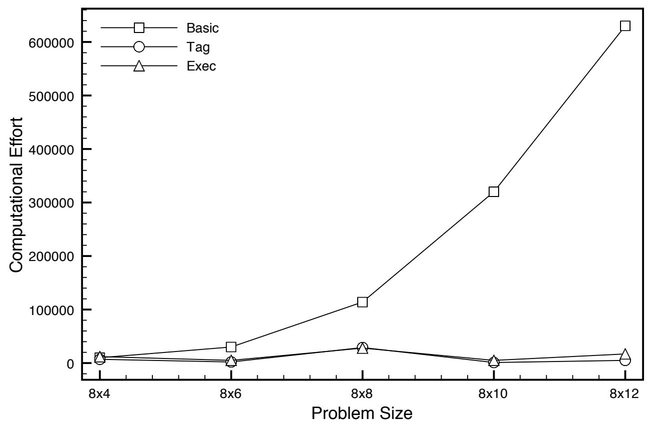
- Used by Koza to demonstrate utility of ADFs for scaling GP up to larger problems



Lawnmower Instructions

Condition	Instructions
Basic	left, mow, v8a, frog, $\mathcal{R}_{v8}$
Tag	left, mow, v8a, frog, $\mathcal{R}_{v8}$, tag.exec.[1000], tagged.[1000]
Exec	left, mow, v8a, frog, $\mathcal{R}_{v8}$, exec.dup, exec.pop, exec.rot, exec.swap, exec.k, exec.s, exec.y

Lawnmower Effort

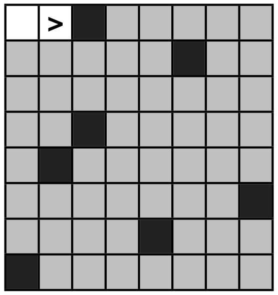


Lawnmower Effort

instr set	8x4	8x6	8x8	8x10	8x12
			problem size		
basic	10000	30000	114000	320000	630000
tag	7000	2000	29000	<1000	5000
exec	12000	5000	28000	5000	17000

Dirt-Sensing, Obstacle-Avoiding Robot Problem

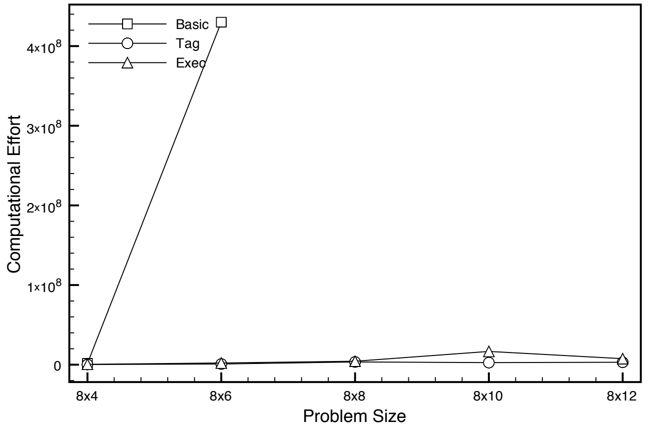
Like the lawnmower problem but harder and less uniform



DSOAR Instructions

Condition	Instructions
Basic	if-dirty, if-obstacle, left, mop, v8a, frog, $\mathcal{R}_{v8}$
Tag	if-dirty, if-obstacle, left, mop, v8a, frog, $\mathcal{R}_{v8}$, tag.exec.[1000], tagged.[1000]
Exec	if-dirty, if-obstacle, left, mop, v8a, frog, $\mathcal{R}_{v8}$, exec.dup, exec.pop, exec.rot, exec.swap, exec.k, exec.s, exec.y

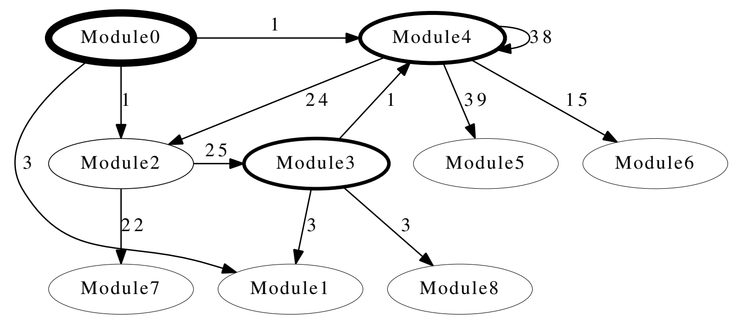
DSOAR Effort



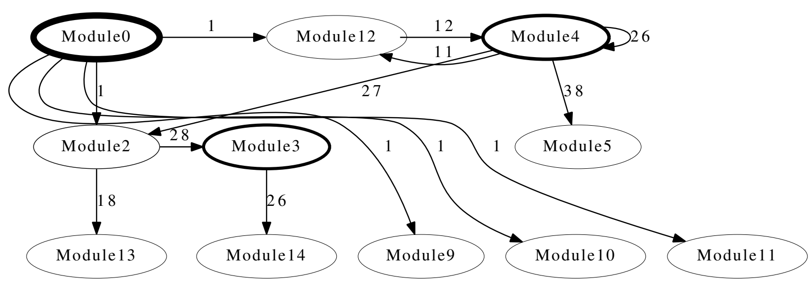
DSOAR Effort

instr set	8x4	8x6	8x8	8x10	8x12
			problem size		
basic	1584000	430083000	inf	inf	inf
tag	216000	864000	3420000	2599000	3051000
exec	450000	2125000	4332000	16644000	7524000

Evolved DSOAR Architecture (in one environment)



Evolved DSOAR Architecture (in another environment)



Tags in Trees

- Example:
```
(progn (tag.123 (+ a b))
        (+ tagged.034 tagged.108))
```
- Must do something about endless recursion
- Must do something about return values of tagging operations and references prior to tagging
- Non-trivial to support arguments in a general way
- Utility not clear from experiments conducted to date

Expressiveness and Assessment

- Expressive languages ease representation of programs that over-fit training sets
- Expressive languages ease representation of programs that work only on subsets of training sets
- Lexicase selection may help: Select parents by starting with a pool of candidates and then filtering by performance on individual fitness cases, considered one at a time

Future Work

- Expression of variable scope and local environments
- Expression of concurrency, parallelism, and time-based structures
- Applications for which expressiveness is likely to be essential, e.g. complete software applications and programs for agents in complex, dynamic, heterogeneous environments

Conclusions

- GP in expressive languages may allow for the evolution of complex software
- Minimal-syntax languages can be expressive, and GP systems that evolve programs in such languages can be simple
- Push is expressive, evolvable, successful, and extensible
- Tags appear to allow for the evolvable expression of program modularity

Thanks

This material is based upon work supported by the National Science Foundation under Grant No. 1017817. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation. Thanks also to Thomas Helmuth, Omri Bernstein, Kyle Harrington, Emma Tosch, Kwaku Yeboah Antwi, and Rebecca S. Neimark for discussions related to this work, to Josiah Erikson for systems support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence.

References

http://hampshire.edu/lspector/push

Spector, L., K. Harrington, and T. Helmuth. 2012. Tag-based Modularity in Tree-based Genetic Programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2012)*. ACM Press. In press.

Spector, L. 2012. Assessment of Problem Multimodality by Differential Performance of Lexicase Selection in Genetic Programming: A Preliminary Report. In *Proceedings of the 1st Workshop on Understanding Problems* at the *Genetic and Evolutionary Computation Conference (GECCO-2012)*. ACM Press. In press.

Harrington, K,. L. Spector, J. Pollack, and U.-M. O'Reilly. 2012. Autoconstructive Evolution for Structural Problems. In *Proceedings of the 2nd Workshop on Evolutionary Computation for the Automated Design of Algorithms* at the *Genetic and Evolutionary Computation Conference (GECCO-2012)*. ACM Press. In press.

Spector, L., K. Harrington, B. Martin, and T. Helmuth. 2011. What's in an Evolved Name? The Evolution of Modularity via Tag-Based Reference. In *Genetic Programming Theory and Practice IX*. New York: Springer. pp. 1-16.

Niekum, S., L. Spector, and A. Barto. 2011. Evolution of Reward Functions for Reinforcement Learning. In *GECCO'11 Posters, Genetic and Evolutionary Computation Conference*. ACM Press. pp. 177-178.

Harrington, K., E. Tosch, L. Spector, and J. Pollack. 2011. Compositional Autoconstructive Dynamics. *Unifying Themes in Complex Systems Volume VIII: Proceedings of the Eighth International Conference on Complex Systems*. New England Complex Systems Institute Series on Complexity. NECSI Knowledge Press. pp. 856-870.

Niekum, S., A. Barto, and L. Spector. 2010. Genetic Programming for Reward Function Search. In *IEEE Transactions on Autonomous Mental Development*, Vol. 2, No. 2, pp. 83-90.

Spector, L. 2010. Towards Practical Autoconstructive Evolution: Self-Evolution of Problem-Solving Genetic Programming Systems. In *Genetic Programming Theory and Practice VIII*, edited by R. L. Riolo, T. McConaghy, and E. Vladislavleva, pp. 17-33. New York: Springer.

Langdon, W. B., R. I. McKay, and L. Spector. 2010. Genetic Programming. In *Handbook of Metaheuristics*, 2nd edition, edited by J.-Y. Potvin and M. Gendreau, pp. 185-226. New York: Springer-Verlag.

Spector, L., and J. Klein. 2008. Machine Invention of Quantum Computing Circuits by Means of Genetic Programming. In *AI-EDAM: Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, Vol. 22, No. 3, pp. 275-283.

Spector, L., D. M. Clark, I. Lindsay, B. Barr, and J. Klein. 2008. Genetic Programming for Finite Algebras. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2008)*. ACM Press.

Klein, J., and L. Spector. 2008. Genetic Programming with Historically Assessed Hardness. In *Genetic Programming Theory and Practice VI*, edited by R. L. Riolo, T. Soule, and B. Worzel. New York: Springer-Verlag. pp. 61-75.

Klein, J., and L. Spector. 2007. Unwitting Distributed Genetic Programming via Asynchronous JavaScript and XML. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2007)*, pp. 1628-1635. ACM Press.

Spector, L., J. Klein, and M. Keijzer. 2005. The Push3 Execution Stack and the Evolution of Control. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2005)*, pp. 1689-1696. Springer-Verlag.

Spector, L., J. Klein, C. Perry, and M. Feinstein. 2005. Emergence of Collective Behavior in Evolving Populations of Flying Agents. In *Genetic Programming and Evolvable Machines*, Vol. 6, No. 1, pp. 111-125.

Spector, L., and J. Klein. 2005. Trivial Geography in Genetic Programming. In *Genetic Programming Theory and Practice III*, edited by T. Yu, R.L. Riolo, and B. Worzel, pp. 109-124. Boston, MA: Kluwer Academic Publishers.

Spector, L., C. Perry, J. Klein, and M. Keijzer. 2004. Push 3.0 Programming Language Description. http://hampshire.edu/lspector/push3-description.html.

Spector, L. 2004. *Automatic Quantum Computer Programming: A Genetic Programming Approach*. Boston, MA: Kluwer Academic Publishers.

Crawford-Marks, R., L. Spector, and J. Klein. 2004. Virtual Witches and Warlocks: A Quidditch Simulator and Quidditch-Playing Teams Coevolved via Genetic Programming. In *Late-Breaking Papers of GECCO-2004, the Genetic and Evolutionary Computation Conference*. Published by the International Society for Genetic and Evolutionary Computation.

Spector, L., J. Klein, and C. Perry. 2004. Tags and the Evolution of Cooperation in Complex Environments. In *Proceedings of the AAAI 2004 Symposium on Artificial Multiagent Learning*. Melno Park, CA: AAAI Press.

Spector, L., C. Perry, and J. Klein. 2003. Push 2.0 Programming Language Description. http://hampshire.edu/lspector/push2-description.html.

Spector, L., J. Klein, C. Perry, and M. Feinstein. 2003. Emergence of Collective Behavior in Evolving Populations of Flying Agents. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*. Springer-Verlag. pp. 61-73.

Spector, L., and H.J. Bernstein. 2003. Communication Capacities of Some Quantum Gates, Discovered in Part through Genetic Programming. In J.H. Shapiro and O. Hirota, Eds., *Proceedings of the Sixth International Conference on Quantum Communication, Measurement, and Computing (QCMC)*. Princeton, NJ: Rinton Press.

Spector, L., and A. Robinson. 2002. Genetic Programming and Autoconstructive Evolution with the Push Programming Language. In *Genetic Programming and Evolvable Machines*, Vol. 3, No. 1, pp. 7-40.

Spector, L. 2002. Adaptive populations of endogenously diversifying Pushpop organisms are reliably diverse. In R. K. Standish, M. A. Bedau, and H. A. Abbass (eds.), *Proceedings of Artificial Life VIII, the 8th International Conference on the Simulation and Synthesis of Living Systems*, pp. 142-145. Cambridge, MA: The MIT Press.

Spector, L., and J. Klein. 2002. Evolutionary Dynamics Discovered via Visualization in the BREVE Simulation Environment. In Bilotta et al. (eds), *Workshop Proceedings of the 8th International Conference on the Simulation and Synthesis of Living Systems*, pp. 163-170. Sydney, Australia: University of New South Wales.

Spector, L., and A. Robinson. 2002. Multi-type, Self-adaptive Genetic Programming as an Agent Creation Tool. In *Proceedings of the Workshop on Evolutionary Computation for Multi-Agent Systems, ECOMAS-2002*, International Society for Genetic and Evolutionary Computation.

Crawford-Marks, R., and L. Spector. 2002. Size Control via Size Fair Genetic Operators in the PushGP Genetic Programming System. In W. B. Langdon et al. (editors), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2002*, pp. 733-739. San Francisco, CA: Morgan Kaufmann Publishers.

Robinson, A., and L. Spector. 2002. Using Genetic Programming with Multiple Data Types and Automatic Modularization to Evolve Decentralized and Coordinated Navigation in Multi-Agent Systems. In *Late-Breaking Papers of GECCO-2002, the Genetic and Evolutionary Computation Conference*. Published by the International Society for Genetic and Evolutionary Computation.

Spector, L. 2001. Autoconstructive Evolution: Push, PushGP, and Pushpop. In Spector, L. et al. (editors), *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO-2001*, 137-146. San Francisco, CA: Morgan Kaufmann Publishers.

Robinson, A. 2001. Genetic Programming: Theory, Implementation, and the Evolution of Unconstrained Solutions. Hampshire College Division III (senior) thesis.

General references on genetic programming

Poli, R., W. B. Langdon, and N. F. McPhee. 2008. *A Field Guide to Genetic Programming*. Lulu Enterprises.

Koza, J. R., M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. 2005. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Springer.

Langdon, W. B., and R. Poli. 2002. *Foundations of Genetic Programming*. Springer.

Koza, J. R., F. H Bennett III, D. Andre, and M. A. Keane. 1999. *Genetic Programming III: Darwinian Invention and Problem Solving*. Morgan Kaufmann.

Banzhaf, W., P. Nordin, R. E. Keller, and F. D. Francone. 1997. *Genetic Programming: An Introduction*. Morgan Kaufmann.

Koza, J. R. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press.

Koza, J. R. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.