

Expressive Genetic Programming: Concepts and Applications

Lee Spector
Cognitive Science, Hampshire College
Computer Science, UMass
Amherst, MA USA
lspector@hampshire.edu

Nicholas Freitag McPhee
Division of Science & Mathematics
University of Minnesota, Morris
Morris, Minnesota USA
mcphee@morris.umn.edu



Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the Owner/Author.

GECCO '18 Companion, July 15–19, 2018, Kyoto, Japan
© 2018 Copyright is held by the owner/author(s).
ACM ISBN 978-1-4503-5764-7/18/07.
<https://doi.org/10.1145/3205651.3207867>

Instructors (1)



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.

More info: <http://hampshire.edu/lspector>

Instructors (2)



Nicholas Freitag McPhee is a Professor of Computer Science in the Division of Science and Mathematics at the University of Minnesota, Morris, in Morris, Minnesota. He received a B.A. in Mathematics from Reed College in 1986 and a Ph.D. from the Department of Computer Sciences at the University of Texas at Austin in 1993. His areas of teaching and research include genetic and evolutionary computation, machine learning, software development, and, when circumstances allow, photography and American roots music. He is a co-author of *A field guide to genetic programming*, and a member of the editorial board of the journal *Genetic Programming and Evolvable Machines* (published by Springer).

More info: <http://facultypages.morris.umn.edu/~mcphee>

Outline

- The Push programming language
- Types and control without syntax
- Evolving Push programs
- Understanding expressive program evolution
- Evolving program evolution

Push

- Programming language for programs that evolve
- Data flows via per-type stacks, not syntax
- Trivial syntax, rich data and control structures
- PushGP: GP system that evolves Push programs
- C++, Clojure, Common Lisp, Elixir, Java, Javascript, Python, Racket, Ruby, Scala, Scheme, Swift; **build your own in any language quickly**
- <http://pushlanguage.org>

Expressive

- Multiple types
- Arbitrary control
- Multiple tasks (use lexicase selection)
- Self reproduction/variation (optional)
- Flexibility, resilience, neutrality

The Push VM

		True		
integer_mult		False		
boolean_and	7	True	"Hello"	
(3 string_dup)	-20	True	"Push"	
integer_add	100	False	"Evolution!"	
Exec	Integer	Boolean	String	...

Push Execution

- Push the program onto the **exec** stack.
- While **exec** isn't empty and we haven't hit the step limit, pop and do the top:
 - If it's an instruction, execute it. (Insufficient arguments? Do nothing.)
 - If it's a literal, push it onto the appropriate stack.
 - If it's a list, push its elements back onto the **exec** stack one at a time.

Example Execution

		True		
integer_mult		False		
boolean_and	7	True	"Hello"	
(3 string_dup)	-20	True	"Push"	
integer_add	100	False	"Evolution!"	
Exec	Integer	Boolean	String	...

		True		
		False		
boolean_and		True	"Hello"	
(3 string_dup)	-140	True	"Push"	
integer_add	100	False	"Evolution!"	
Exec	Integer	Boolean	String	...

		False		
		True	"Hello"	
(3 string_dup)	-140	True	"Push"	
integer_add	100	False	"Evolution!"	
Exec	Integer	Boolean	String	...

		False		
3		True	"Hello"	
string_dup	-140	True	"Push"	
integer_add	100	False	"Evolution!"	
Exec	Integer	Boolean	String	...

		False	
	3	True	"Hello"
string_dup	-140	True	"Push"
integer_add	100	False	"Evolution!"
Exec	Integer	Boolean	String
			...

		False	"Hello"
	3	True	"Hello"
	-140	True	"Push"
integer_add	100	False	"Evolution!"
Exec	Integer	Boolean	String
			...

		False	"Hello"
		True	"Hello"
	-137	True	"Push"
	100	False	"Evolution!"
Exec	Integer	Boolean	String
			...

(1 2 integer_add)			
Exec	Integer	Boolean	String
			...

Full Program

1
2
integer_add

Exec Integer Boolean String ...

2
integer_add

1

Exec Integer Boolean String ...

integer_add 2
1

Exec Integer Boolean String ...

3

Exec Integer Boolean String ...

```
(1 2 integer_add)
```

leaves 3 on the integer stack

```
(True False boolean_or boolean_not)
```

leaves False on the boolean stack

```
(3 5 integer_lte)
```

leaves True on the boolean stack

```
(3 5 integer_lte exec_if (1 "yes") (2 "no"))
```

leaves "yes" on string, 1 on integer

For Most Types

- <type>_dup
- <type>_empty
- <type>_eq
- <type>_flush
- <type>_pop
- <type>_rot
- <type>_shove
- <type>_stackdepth
- <type>_swap
- <type>_yank
- <type>_yankdup

Selected Integer Instructions

```
integer_add integer_dec integer_div  
integer_gt integer_fromstring integer_min  
integer_mult integer_rand
```

Selected Boolean Instructions

```
boolean_and boolean_xor boolean_frominteger
```

Selected String Instructions

```
string_concat string_contains string_length  
string_removechar string_replacechar
```

Exec (selected)

Conditionals:

```
exec_if exec_when
```

General loops:

```
exec_do*while
```

"For" loops:

```
exec_do*range exec_do*times
```

Looping over structures:

```
exec_do*vector_integer exec_string_iterate
```

Combinators:

```
exec_k exec_y exec_s
```

More

```
code atom code car print newline integer_sub integer_inc boolean_stackdepth return_exec pop vector_integer_eq autoconstructive_integer_rand boolean_pop
genome_close_inc string_fromchar vector_string_show zip_yankdup genome_new_vector_float_yankdup exec_yankdup vector_integer_show integer_yankdup string_flush
boolean_swap zip_empty exec_show vector_boolean_yank code_eq exec_y boolean_yank integer_eq genome_silence string_boolean code_contains string_conjchar
code_do_count vector_float_last genome_pop string_substring integer_mult code_length vector_integer_dup boolean_or code_position boolean_empty zip_fromcode
print_vector_string vector_boolean_swap return_frominteger vector_float_pushall char_inhitegap code_cdr exec_do_vector_integer_integer_rand
vector_string_replacefirst string_first vector_boolean_first exec_do_while exec_string_iterate string_indexofchar vector_float_replace integer_fromstring
code_list code_swap char_frominteger genome_gene_randomize vector_integer_emptyvector vector_string_eq vector_float_boolean exec_empty zip_end? exec_fromipnode
string_show vector_boolean_pushall zip_insert_left fromcode exec_rot vector_string_concat vector_float_indexof code_pop vector_string_subvec vector_integer_swap
code_subst char_pop return_string_pop zip_yank exec_dup vector_integer_boolean vector_float_rest vector_flush boolean_fromfloat code_fromziprights
float_glt boolean_float char_laddigit float_the exec_fromiproot vector_integer_empty print_code vector_string_stackdepth string_reverse exec_>
vector_integer_yank float_frominteger char_rot print_char vector_integer_stackdepth vector_boolean_concat boolean_xor integer_gte genome_yankdup
vector_float_show vector_integer_take code_quote string_replacefirst return_fromstring exec_fromziplefts vector_integer_yankdup boolean_show float_lt
vector_string_dup vector_string_occurrencesof vector_integer_replace zip_branch? vector_float_reverse float_not vector_float_subvec string_last print_boolean
boolean_rot vector_string_rest integer_div vector_float_remove integer_fromfloat integer_lte code_fromzipchildren environment_end vector_integer_rot integer_mod
string_concat vector_string_boolean genome_swap code_null exec_do_count vector_float_emptyvector vector_string_yankdup integer_rot float_yankdup
vector_string_rot zip_replace fromexec vector_string_take integer_add vector_integer_occurrencesof integer_show genome_dup return_code pop char_swap integer_max
return_fromexec code_swap return_float_pop code_flush genome_yank zip_show vector_integer_flush vector_integer_subvec vector_boolean_indexof vector_float_pop
vector_string_remove vector_integer_contains zip_remove code_append vector_float_eq vector_integer_conj string_eq zip_leftmost code_yankdup code_rot
integer_stackdepth float_max vector_boolean_set zip_append_child fromexec zip_rest vector_float_conj zip_fromexec string_take zip_left zip_replace fromcode
char_stackdepth return_fromchar genome_eq vector_integer_replacefirst float_stackdepth code_fromiproot float_fromchar float_gt boolean_dup float_fromboolean
code_fromipnode genome_rot vector_float_replacefirst vector_boolean_conj vector_boolean_dup vector_integer_indexof vector_string_swap exec_eq string_emptystring
string_swap integer_yank exec_while float_empty print_vector_boolean integer_glt exec_swap genome_rotate integer_fromchar vector_string_yank string_stackdepth
code_dorange string_replacechar char_atifromstring vector_integer_rest vector_boolean_length char_yank vector_float_empty code_fromfloat genome_parent2
return_fromcode string_pop float_eq vector_boolean_empty zip_insert_child fromexec vector_string_last string_nth code_do? return_zip_pop vector_string_pop
zip_rot vector_integer_nth exec_dorange exec_if char_show zip_down zip_insert_left fromexec code_frominteger vector_boolean_remove vector_integer_remove
boolean_invert_first_then_and genome_flush print_string integer_fromboolean char_yankdup code_do_vector_string_first boolean_frominteger string_setchar
vector_integer_last char_listiter genome_gene_dup vector_integer_concat print_integer code_pop boolean_eq float_gte return_fromfloat genome_gene_copy
string_occurrencesofchar string_replacefirstchar print_float_boolean_rand integer_flush float_show string_replace char_dup float_pop char_eq vector_float_nth
vector_string_conj integer_gt return_integer pop float_sub vector_integer_length vector_float_set vector_string_indexof vector_boolean_rest code_dup
vector_boolean_show zip_eq float_min boolean_not float_mult float_fromstring genome_unsilence code_if vector_integer_pop vector_boolean_last exec_dortimes
zip_pop zip_rightmost float_dec vector_float_contains genome_gene_copy range environment_new exec_do_vector_string code_nthof string_empty char_empty exec_pop
vector_integer_set autoconstructive_boolean_rand vector_float_rot string_yankdup exec_do_vector_float_string_removechar code_extract vector_string_replace
vector_float_first genome_parent return_tagpiece char_flush vector_float_occurrencesof vector_string_emptyvector float_add code_stackdepth exec_s
zip_insert_right fromexec float_dup vector_string_nth zip_stackdepth vector_integer_reverse print_vector_integer char_fromfloat code_dortimes code_noop zip_swap
code_yank integer_lt vector_boolean_eq genome_stackdepth code_fromziplefts noop_open_paren string_containschar string_yank char_rand zip_flush vector_boolean_rot
float_swap exec_fromzipright vector_string_set vector_boolean_flush exec_noop code_glt vector_boolean_stackdepth vector_integer_pushall
vector_boolean_reverse integer_swap string_split vector_boolean_contains string_fromboolean return_boolean_pop vector_float_dup vector_boolean_replace
integer_dup vector_boolean_nth vector_string_length string_rest zip_insert_child fromcode float_tan string_rot string_rand exec_show string_pare to_chars
integer_pop integer_empty vector_float_flush vector_float_yank noop_delete_prev_paren_pair print_exec_zip_append_child fromcode genome_gene_delete code_empty
float_inc zip_right vector_float_length float_rand integer_dec string_contains return_fromboolean vector_float_concat vector_float_stackdepth
exec_do_vector_boolean vector_integer_first genome_show code_rand print_vector_float_float_rot return_char_pop vector_string_contains
vector_boolean_occurrencesof genome_empty zip_prev genome_toggle_silent vector_string_reverse zip_dup code_cons code_number exec_stackdepth float_flush
boolean_and vector_boolean_boolean string_length float_con string_frominteger exec_flush vector_string_empty exec_show vector_float_swap genome_close_dec
code_insert vector_boolean_pop float_div zip_insert_right fromcode code_fromboolean vector_boolean_take code_show environment_begin vector_float_take
boolean_invert_second_then_and code_container code_nth vector_boolean_subvec float_yank zip_up vector_boolean_emptyvector vector_boolean_replacefirst
string_fromfloat vector_boolean_yankdup string_dup boolean_yankdup exec_fromziplefts
```

More

- Input instructions
- Print instructions
- Associative storage via tags
- Environment/return stacks
- Limits, termination modes

Clojush README.md at master · lspector/Clojush

This repository

Pull requests Issues Qist

lspector / Clojush

Unwatch 20 Unstar 193 Fork 62

Code Issues 402 Pull requests 1 Projects 1 Wiki Pulse Graphs Settings

Branch master Clojush / README.md Find file Copy path

lspector lein release:minor b5c882 on Jan 12

5 contributors

341 lines (276 sloc) 17.9 KB Raw Blame History

Clojush

Lee Spector (lspector@shampshire.edu), started 20100227 See version history. Older version history is in old-version-history.txt.

This is the README file accompanying Clojush, an implementation of the Push programming language and the PushGP genetic programming system in the Clojure programming language. Among other features this implementation takes advantage of Clojure's facilities for multi-core concurrency.

Availability

<https://github.com/lspector/Clojush>

Requirements

To use this code you must have a Clojure programming environment; see <http://clojure.org/>. The current version of Clojush requires Clojure 1.7.0.

Clojure is available for most OS platforms. A good starting point for obtaining and using Clojure.

Quickstart

Using Leiningen you can run an example from the OS command line (in the Clojush directory) with a call like:

```
lein run clojush.problems.demos.simple-regression
```

```
;; https://github.com/lspector/Clojush/
```

```
=> (run-push '(1 2 integer_add) (make-push-state))
```

```
:exec ((1 2 integer_add))
:integer ( )
```

```
:exec (1 2 integer_add)
:integer ( )
```

```
:exec (2 integer_add)
:integer (1)
```

```
:exec (integer_add)
:integer (2 1)
```

```
:exec ( )
:integer (3)
```

```
=> (run-push '(2 3 integer_mult 4.1 5.2 float_add
               true false boolean_or)
      (make-push-state))
```

```
:exec ()
:integer (6)
:float (9.3)
:boolean (true)
```

In other words

- Put 2×3 on the integer stack
- Put $4.1 + 5.2$ on the float stack
- Put *true* ∨ *false* on the boolean stack

```
=> (run-push '(2 boolean_and 4.1 true integer_div
               false 3 5.2 boolean_or integer_mult
               float_add)
      (make-push-state))
```

```
:exec ()
:integer (6)
:float (9.3)
:boolean (true)
```

Same as before, but

- Several operations (e.g., *boolean_and*) become NOOPs
- Interleaved operations

```
=> (run-push
    '(4.0 exec_dup (3.13 float_mult) 10.0 float_div)
    (make-push-state))
```

```
:exec ((4.0 exec_dup (3.13 float_mult) 10.0 float_div))
:float ()
```

```
:exec (4.0 exec_dup (3.13 float_mult) 10.0 float_div)
:float ()
```

```
:exec (exec_dup (3.13 float_mult) 10.0 float_div)
:float (4.0)
```

```
:exec((3.13 float_mult) (3.13 float_mult) 10.0 float_div)
:float (4.0)
```

...

```
:exec ()
:float (3.91876)
```

Computes $4.0 \times 3.13 \times 3.13 / 10.0$

```
=> (run-push '(1 8 exec_do*range integer_mult)
      (make-push-state))
```

```
:integer (40320)
```

Computes $8!$ in a fairly “human” way

```
=> (run-push '(code_quote
  (code_quote (integer_pop 1)
    code_quote (code_dup integer_dup
      1 integer_sub code_do
        integer_mult)
      integer_dup 2 integer_lt code_if)
    code_dup
      8
    code_do)
  (make-push-state)))

:code ((code_quote (integer_pop 1) code_quote (code_dup
  integer_dup 1 integer_sub code_do integer_mult)
  integer_dup 2 integer_lt code_if))
:integer (40320)
```

A less “obvious” recursive calculation of 8! achieved by code duplication

```
=> (run-push '(0 true exec_while
  (1 integer_add true))
  (make-push-state))

:exec (1 integer_add true exec_while (1 integer_add
  true))
:integer (199)
:termination :abnormal
```

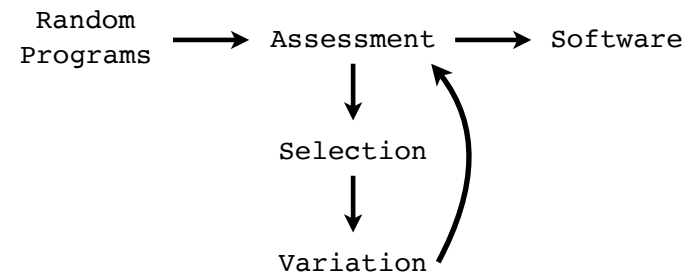
- An infinite loop
- Terminated by eval limit
- Result taken from appropriate stack(s) upon termination

```
=> (run-push '(in1 in1 float_mult 3.141592 float_mult)
  (push-item 2.5 :input (make-push-state)))

:float (19.63495)
:input (2.5)
```

Computes the area of a circle with the given radius: $3.141592 \times in1 \times in1$

Genetic Programming



Linear Genomes

- Support uniform variation
- Structure where we want it, via translation
- ULTRA: Uniform Linear Transformation with Repair and Alternation
- PLUSH: epigenetic markers (Clojush)
- Plushy: close instructions (plushi, propel)

Uniform Variation

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

Structure

- Parentheses matter mostly for defining code blocks for **exec** instructions
- Openings of blocks can be determined by placement of relevant **exec** instructions
- Closings of blocks can be encoded in genomes in several ways

ULTRA

- Linearize just during variation:
 - Linearize two parents
 - Alternate according to *alternation rate*, subject to *alignment deviation*
 - Uniformly mutate, subject to *mutation rate*
 - Repair
- Uniform, but structure anywhere

Plush

Instruction	integer_eq	exec_dup	char_swap	integer_add	exec_if	
Close?	2	0	0	0	1	
Silence?	1	0	0	1	0	

- Linear sequences of instructions and literals
- Instructions specify opens; epigenetic markers specify closes
- Permits uniform linear genetic operators
- Facilitates useful placement of code blocks
- Allows for epigenetic hill-climbing

Plushy

- Instructions specify opens
- "close" pseudo-instructions specify closes
- Used in plushi, propel

Genetic Operators

- Uniform mutation
- Alternation
- Uniform crossover
- Uniform mutation by addition and deletion (UMAD)
- Autoconstruction (genome instructions)

PushGP in Clojush

Set up run for target $x^3 - 2x^2 - x$

```
(pushgp
{:error-function
 (fn [{:keys [program] :as individual}]
  (assoc individual :errors
    (vec
      (for [input (mapv float (range 10))]
        (let [output (-> (make-push-state)
                          (push-item input :input)
                          (run-push program)
                          (top-item :float))]
          (if (number? output)
              (Math/abs (float (- output
                                (- (* input input input)
                                   (* 2 input input)
                                   input))))
              1000000))))))
:atom-generators
'(in1 float_div float_mult float_add float_sub))
```

DEMO

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

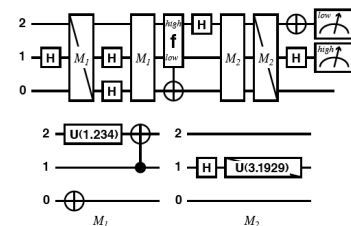
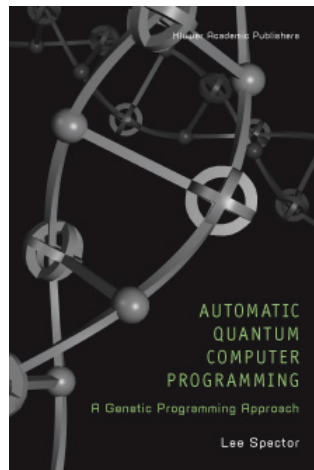


Figure 8.7. A gate array diagram for an evolved version of Grover's database search algorithm for a 4-item database. The full gate array is shown at the top, with M_1 and M_2 standing for the smaller gate arrays shown at the bottom. A diagonal line through a gate symbol indicates that the matrix for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

29 Software Synthesis Benchmarks

- Number IO, Small or Large, For Loop Index, Compare String Lengths, Double Letters, [Collatz Numbers](#), Replace Space with Newline, String Differences, Even Squares, [Wallis Pi](#), String Lengths Backwards, Last Index of Zero, Vector Average, Count Odds, Mirror Image, [Super Anagrams](#), Sum of Squares, Vectors Summed, X-Word Lines, [Pig Latin](#), Negative to Zero, Scrabble Score, [Word Stats](#), Checksum, Digits, Grade, Median, Smallest, Syllables
- PushGP has solved all of these except for the ones in [blue](#)
- Presented in a GECCO-2015 GP track paper

Auto-Simplification

- Loop:
 - Make it randomly simpler
 - Keep simpler if as good or better; otherwise revert
- GECCO-2014 poster: efficiently and reliably reduces the size of the evolved programs
- GECCO-2014 student paper: used as genetic operator
- GECCO-2017 GP best paper nominee: improves generalization

SUCCESS at generation 20

[illegible][illegible]

Auto-simplifying with starting size: 231

...

step: 5000

```
program: (\space \newline in1 string_replacechar print_string "Wx{ "  
string_last in1 string_removechar string_length)
```

[illegible]

```

.....
Total error: 0.0

```

Size: 11

Checksum

Multiple types, looping, and code blocks

Simplified solution:

```

("Check sum is " print_string
 in1 64 exec_string_iterate (integer_fromchar integer_add)
 64 integer_mod
 \space integer_fromchar integer_add char_frominteger
 print char)

```

Checksum

Multiple types, looping, and code blocks

Simplified solution:

```
("Check sum is " print_string
  in1 64 exec_string_iterate (integer_fromchar integer_add)
  64 integer_mod
  \space integer_fromchar integer_add char_frominteger
  print_char)
```

First: Print out the header

Checksum

Multiple types, looping, and code blocks

Simplified solution:

```
("Check sum is " print_string
  in1 64 exec_string_iterate (integer_fromchar integer_add)
  64 integer_mod
  \space integer_fromchar integer_add char_frominteger
  print_char)
```

Second: Convert each character to an integer, sum,
and add to 64.

Checksum

Multiple types, looping, and code blocks

Simplified solution:

```
("Check sum is " print_string
  in1 64 exec_string_iterate (integer_fromchar integer_add)
  64 integer_mod
  \space integer_fromchar integer_add char_frominteger
  print_char)
```

Third: Mod result by 64

Checksum

Multiple types, looping, and code blocks

Simplified solution:

```
("Check sum is " print_string
  in1 64 exec_string_iterate (integer_fromchar integer_add)
  64 integer_mod
  \space integer_fromchar integer add char frominteger
  print_char)
```

Third: Add modulus result to 32 and convert to char

Checksum

Multiple types, looping, and code blocks

Simplified solution:

```
("Check sum is " print_string
  in1 64 exec_string_iterate (integer_fromchar integer_add)
  64 integer_mod
  \space integer_fromchar integer_add char_frominteger
  print_char)
```

Fourth: Print resulting char

Replace Space With Newline

Multiple types, looping, multiple tasks

Simplified solution:

```
(\space char_dup exec_dup in1
  \newline string_replacechar print_string
  string_removechar string_length)
```

Replace Space With Newline

Multiple types, looping, multiple tasks

Simplified solution:

```
(\space char_dup exec_dup in1
  \newline string_replacechar print_string
  string_removechar string_length)
```

First: Duplicate space character and input string for use in both tasks

Replace Space With Newline

Multiple types, looping, multiple tasks

Simplified solution:

```
(\space char_dup exec_dup in1
  \newline string_replacechar print_string
  string_removechar string_length)
```

Second: Replace spaces with newlines and print

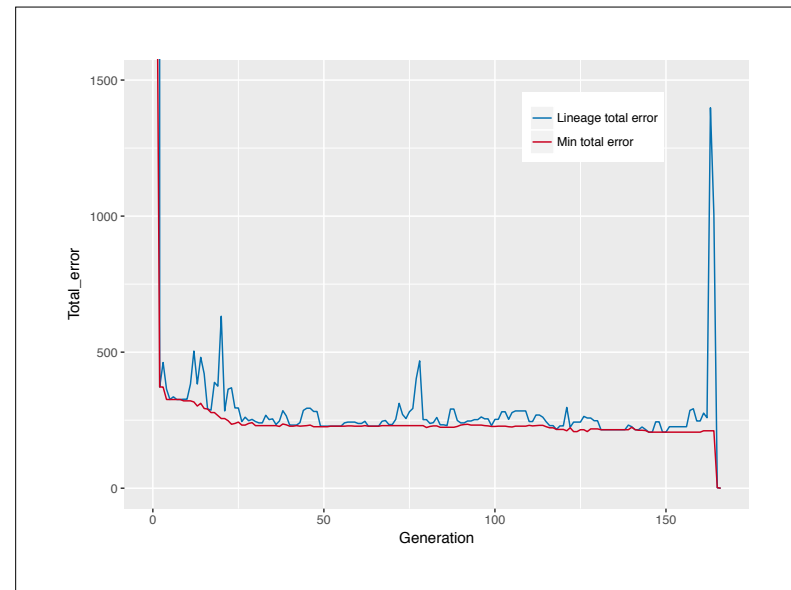
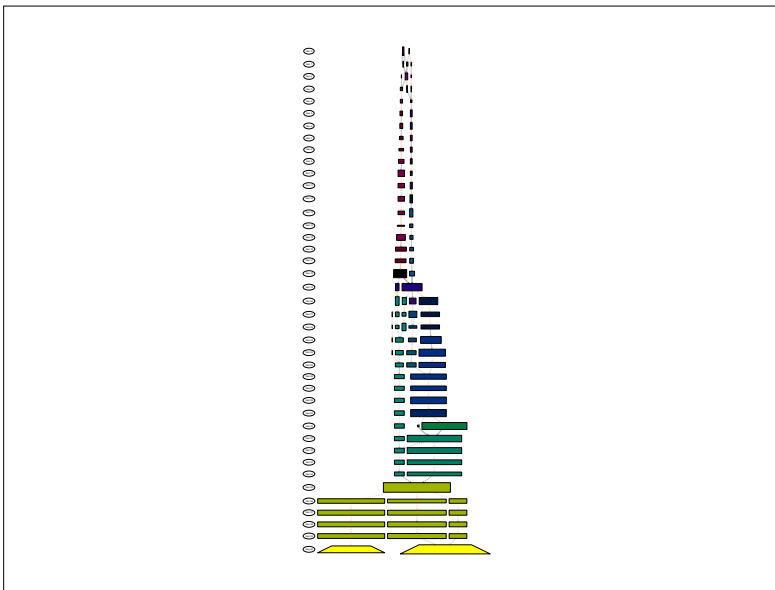
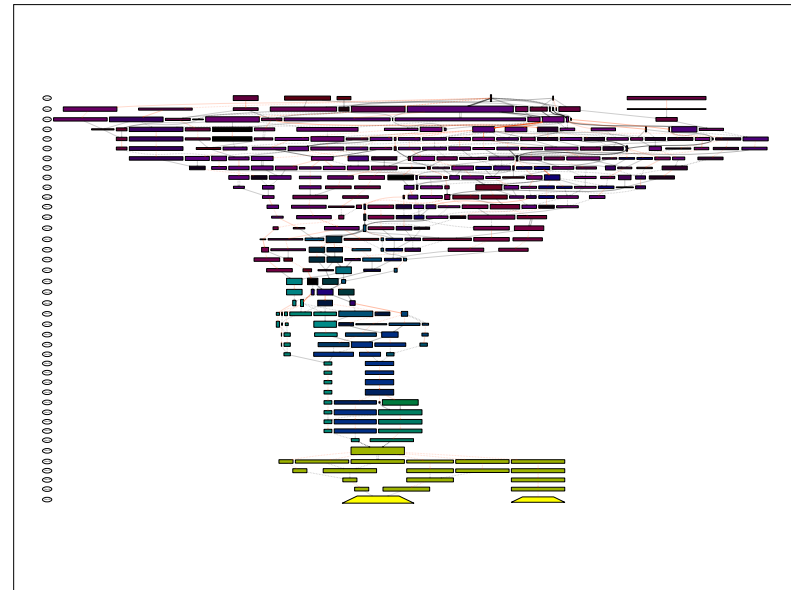
Replace Space With Newline

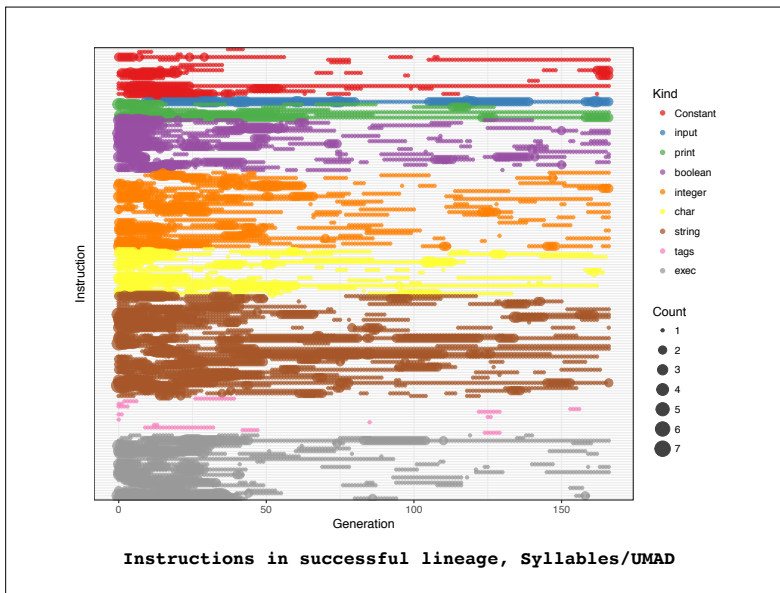
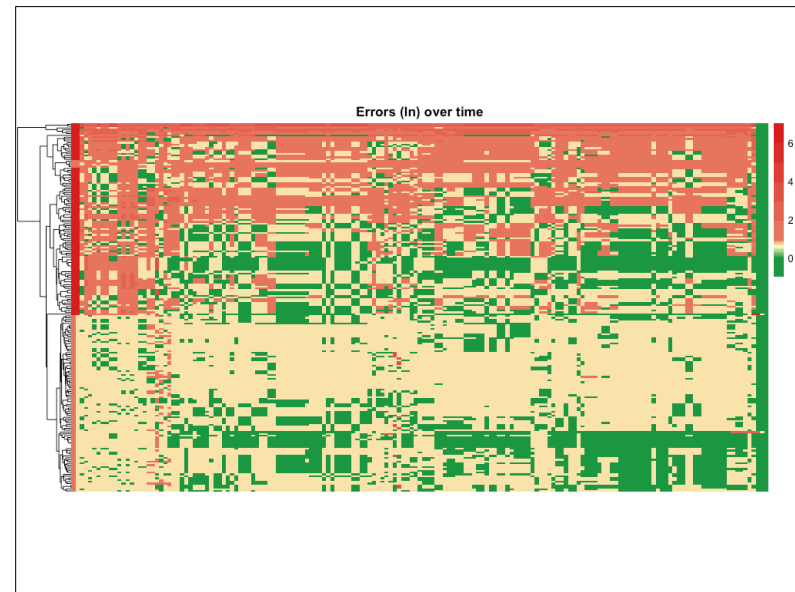
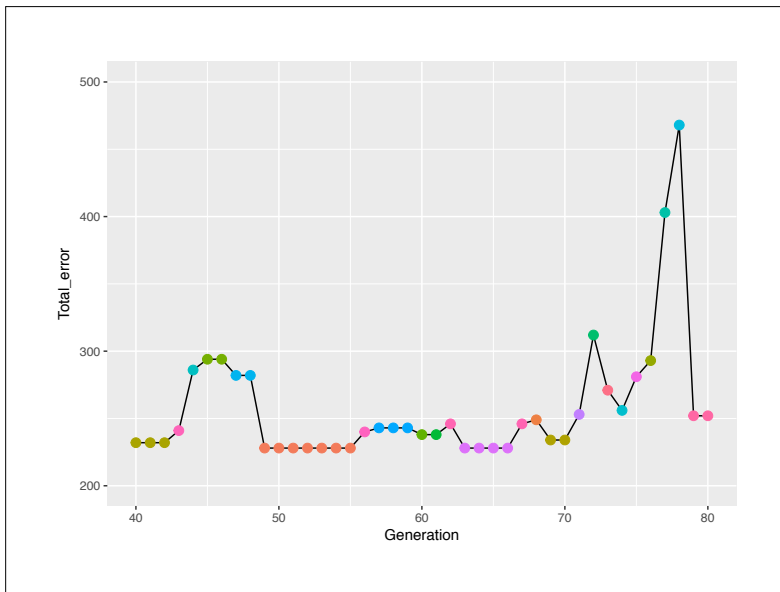
Multiple types, looping, multiple tasks

Simplified solution:

```
(\space char_dup exec_dup inl
 \newline string_replacechar print_string
 string_removechar string_length)
```

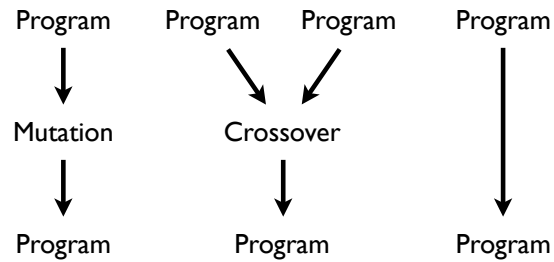
Third: Remove all spaces from second copy of input,
and push length of result on integer stack for return



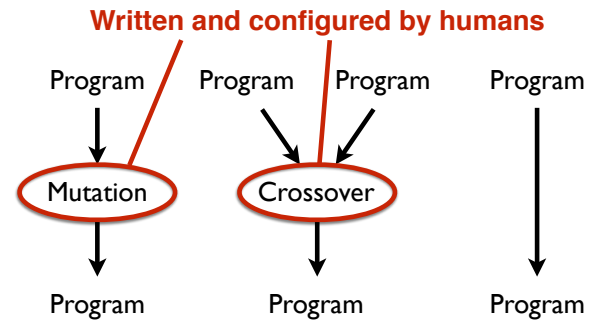


DEMO

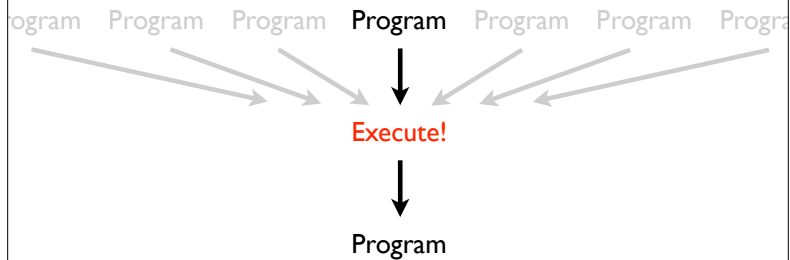
Variation in GP



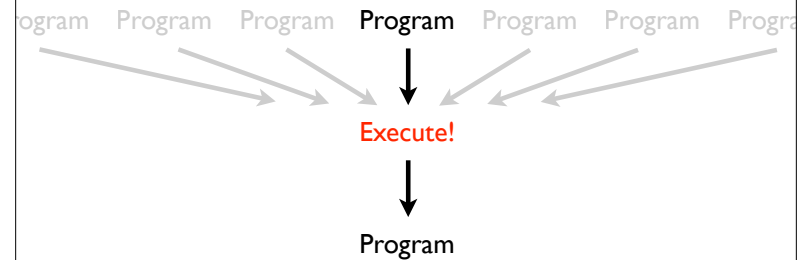
Variation in GP



Autoconstruction



Autoconstruction



A bit more complicated when genomes distinguished from programs

Autoconstruction

- Evolve evolution while evolving solutions
- How? Individuals produce and vary their own children, with methods that are subject to variation
- Requires understanding the evolution of variation
- Hope: May produce EC systems more powerful than we can write by hand

Autoconstruction

- A 15 year old project (building on older and broader-based ideas)
- Like genetic programming, but harder and less successful! But with greater potential?
- Recent versions sometimes solve significant problems, intriguing patterns of evolving evolution

For Evolution²

- Diversity: Individuals vary
- Diversification: Individuals produce descendants that vary, in various ways
- Recursive Variance: Individuals produce descendants that vary in the ways that they vary their offspring

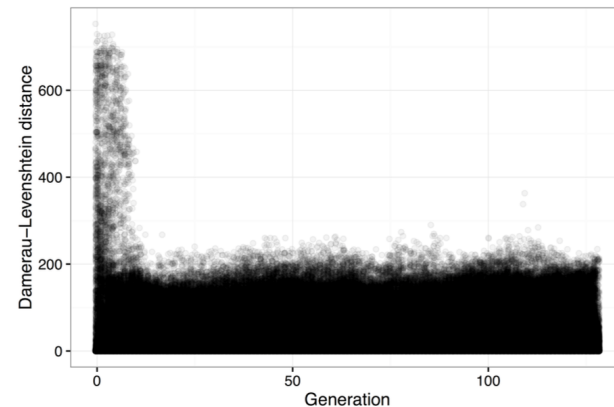


Figure 1: DL-distances between parent and child during a single non-autoconstructive run of GP on the Replace Space With Newline problem

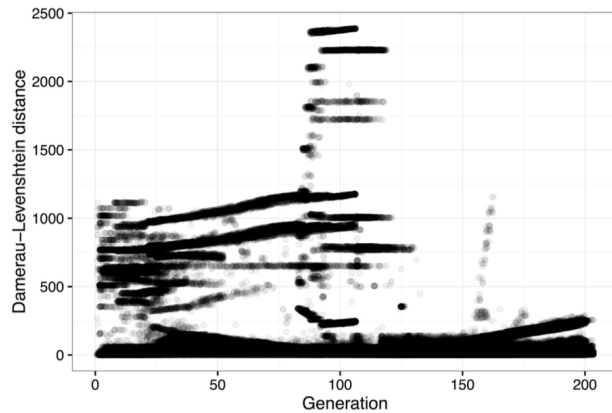


Figure 3: DL-distances between parent and child during a single autoconstructive run of GP on the Replace Space With Newline problem

Evolution Evolving

- Autoconstructive evolution can sometimes succeed as much and as fast as non-autoconstructive evolution
- Autoconstruction found solutions for the string differences software synthesis problem before ordinary GP

Conclusions

- Push supports evolution of expressive programs that use arbitrary types and control structures, possibly to perform multiple tasks
- Push interpreters, and GP systems that evolve Push programs, are easy to write
- Push supports research on expressiveness in genetic programming, for example to support the evolution of modularity

Thanks

Thanks to the members of the Hampshire College Computational Intelligence Lab, to Josiah Erikson for systems support, and to Hampshire College for support for the Hampshire College Institute for Computational Intelligence.

This material is based upon work supported by the National Science Foundation under Grant No. 1617087. 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.

References

- The Push language website: <http://pushlanguage.org>
- Helmuth, Thomas, Nicholas Freitag McPhee, and Lee Spector. 2018. Program Synthesis using Uniform Mutation by Addition and Deletion. In *Proc. Genetic and Evolutionary Computation Conference*. ACM Press.
- Helmuth, Thomas, Nicholas Freitag McPhee, Edward Pantridge, and Lee Spector. 2017. Improving Generalization of Evolved Programs through Automatic Simplification. In *Proc. Genetic and Evolutionary Computation Conference*. ACM Press.
- Helmuth, Thomas, Lee Spector, Nicholas Freitag McPhee, and Saul Shanabrook. Linear Genomes for Structured Programs. In Worzel, William, William Tozier, Brian W. Goldman, and Rick Riolo, Eds., *Genetic Programming Theory and Practice XIV*. New York: Springer.
- McPhee, Nicholas Freitag, Mitchell D. Finzel, Maggie M. Casale, Thomas Helmuth and Lee Spector. A detailed analysis of a PushGP run. In Worzel, William, William Tozier, Brian W. Goldman, and Rick Riolo, Eds., *Genetic Programming Theory and Practice XIV*. New York: Springer.
- Spector, L., N. F. McPhee, T. Helmuth, M. M. Casale, and J. Oks. 2016. Evolution Evolves with Autoconstruction. In *Companion Publication of the 2016 Genetic and Evolutionary Computation Conference*. ACM Press. pp. 1349-1356.
- Helmuth, T., and L. Spector. 2015. General Program Synthesis Benchmark Suite. In *Proc. 2015 Genetic and Evolutionary Computation Conference*. ACM Press. pp. 1039-1046.
- La Cava, W., and L. Spector. 2015. Inheritable Epigenetics in Genetic Programming. In *Genetic Programming Theory and Practice XII*. New York: Springer. pp. 37-51.
- Helmuth, T., L. Spector, and J. Matheson. 2015. Solving Uncompromising Problems with Lexicase Selection. In *IEEE Transactions on Evolutionary Computation* 19(5), pp. 630-643.
-

- Helmuth, T., and L. Spector. 2014. Word Count as a Traditional Programming Benchmark Problem for Genetic Programming. In *Proc. 2014 Genetic and Evolutionary Computation Conference*. ACM Press. pp. 919-926.
- Spector, L., and T. Helmuth. 2014. Effective Simplification of Evolved Push Programs Using a Simple, Stochastic Hill-climber. In *Companion Publication of the 2014 Genetic and Evolutionary Computation Conference*. ACM Press. pp. 147-148.
- Zhan, H. 2014. A quantitative analysis of the simplification genetic operator. In *Companion Publication of the 2014 Genetic and Evolutionary Computation Conference*. ACM Press. pp. 1077-1080.
- Spector, L., K. Harrington, and T. Helmuth. 2012. Tag-based Modularity in Tree-based Genetic Programming. In *Proc. Genetic and Evolutionary Computation Conference*. ACM Press. pp. 815-822.
- 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.
- Spector, L. 2010. Towards Practical Autoconstructive Evolution: Self-Evolution of Problem-Solving Genetic Programming Systems. In *Genetic Programming Theory and Practice VIII*, R. L. Riolo, T. McConaghy, and E. Vladislavleva, eds. Springer. pp. 17-33.
- Spector, L., D. M. Clark, I. Lindsay, B. Barr, and J. Klein. 2008. Genetic Programming for Finite Algebras. In *Proc. Genetic and Evolutionary Computation Conference*. ACM Press. pp. 1291-1298.
- Spector, L., J. Klein, and M. Keijzer. 2005. The Push3 Execution Stack and the Evolution of Control. In *Proc. Genetic and Evolutionary Computation Conference*. Springer-Verlag. pp. 1689-1696.
- Spector, L. 2004. *Automatic Quantum Computer Programming: A Genetic Programming Approach*. Boston, MA: Kluwer Academic Publishers.
- 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. 2001. Autoconstructive Evolution: Push, PushGP, and Pushpop. In *Proc. Genetic and Evolutionary Computation Conference*. Morgan Kaufmann Publishers. pp. 137-146.
- Robinson, A. 2001. Genetic Programming: Theory, Implementation, and the Evolution of Unconstrained Solutions. Hampshire College Division III (senior) thesis.