Genetic Programming 1996: Proceedings of the First Annual Conference, John R. Koza and David E. Goldberg and David B. Fogel and Rick L. Riolo (eds), MIT Press

Using Data Structures within Genetic Programming

W. B. Langdon
Genetic Programming Group,
Dept. of Computer Science,
University College, London,
Gower Street, London WC1E 6BT, UK
Email: W.Langdon@cs.ucl.ac.uk
www: http://www.cs.ucl.ac.uk/staff/W.Langdon/
Tel: +44 (0) 171 380 7214, Fax: +44 (0) 171 387 1397

Abstract

Provision of appropriately structured memory is shown, in some cases, to be advantageous to genetic programming (GP) in comparison with directly addressable indexed memory.

Three "classic" problems are solved. The first two require the GP to distinguish between sentences that are in a context free language and those that are not given positive and negative training examples of the language. The two languages are, correctly nested brackets and a Dyck language (correctly nested brackets of different types). The third problem is to evaluate integer Reverse Polish (postfix) expressions.

Comparisons are made between GP attempting to solve these problems when provided with indexed memory or with stack data structures.

1 Introduction

Human programmers have long recognised that program production, maintenance and testing are eased if the software is written so that it be independent of memory access implementation details. This is achieved by using abstract data structures, such as stacks, queues and lists, to provide an interface between programs and memory.

We seek to show that data abstraction can be beneficially used within genetic programming (GP). So far work by Teller [Teller, 1994], Andre [Andre, 1994] and others shows GP can automatically create programs which explicitly use directly addressable (indexed) memory to solve problems which cannot be solved without memory and we have demonstrated that GP can automatically generate abstract data structures such as stacks, queues [Langdon, 1995] and lists [Langdon, 1996]. In this paper we show that GP can evolve programs which solve problems using such data structures. In two cases we show better GP performance when using data structures compared to directly addressable memory. In the remaining case (which is the first problem presented) the evolved solution uses an unexpected data structure which is appropriate to the problem rather than indexed memory when both are available.

The first two examples (Sections 2 and 3) show GP automatically producing programs which recognize whether a sample of a language is within a specific context free language. The third (Section 4) shows GP can evolve programs which evaluate Reverse Polish (postfix) expressions. These were chosen in the expectation that data structures at least as powerful as stacks would be required to solve them.

2 Balanced Bracket Problem

Other work on GP evolving language recognizers has concentrated upon using GP to evolve tree based specifications for abstract machines, such as finite state machines [Dunay *et al.*, 1994], deterministic push down automata [Zomorodian, 1995], machines composed of simple Turing machines [Dunay and Petry, 1995] or special memory nodes within the tree [Iba *et al.*, 1995]. In contrast we wish to investigate the impact of providing data structures v. indexed memory and so our GP executes the GP tree directly.

The balanced bracket problem is to recognise sentences composed of sequences of two symbols, `(` and `)`, which are correctly nested. E.g. `(())` is correctly nested but `())` is not. A limit of ten symbols per sentence was assumed.

2.1 Architecture

Two automatically defined functions (ADFs) [Koza, 1994] are available to assist the main result producing branch (or tree). The first, adf1, has no arguments and has the same terminal and function sets (except arg1) as the main tree. The second, adf2, has one argument but cannot contain terminals and functions with side effects. This allows a cache of previous values returned by it to be maintained, thus reducing run time.

Table 1 shows the parameters used and the terminals and functions provided, n.b. they include indexed memory but not stacks. Parameters not shown are as [Koza, 1994, page 655]. Table 9 gives the actions of most terminals and functions.

Earlier work with indexed memory had shown even large population had a great tendency to converge to partial solutions which effectively trapped the whole population preventing further progress. Partitioning the population so crossover is restricted to near neighbours has been used to reduced this. In this experiment the population is treated as a 100×100 torus with each member of the population occupying a different square on its surface. Each time a new individual is created a 3×3 square neigbourhood on the torus (known as a deme) is selected at random. Parents and the individual their offspring will replace are selected from this deme rather than from the whole population [Langdon, 1995; Tackett, 1994; Collins, 1992].

2.2 Fitness Function

The fitness of each trial program was evaluated on a fixed set of 175 example sentences containing both correctly nested (positive) and incorrectly nested brackets (negative). The test case includes all the positive cases up to a length of ten symbols and all the negative examples up to a length of four. The number of negative examples grows rapidly with sentence length and so above a length of four a limited number negative examples were chosen at random (see Table 2). The program is run once for each symbol in the sentence. Thus each program is run 1403 times (674 for (and 729 with an argument of)). (This and the two other test cases, Sections 3.2 and 4.2, are available via anonymous ftp node ftp.io.com directory pub/genetic-programming/code in file GPdata_gp96.test.tar.Z).

All stored data within the program initialised to zero at the start of each test sentence. The value returned by the program on the last symbol of the sentence gives its verdict as to whether the sequence is correctly nested.

2.3 Results

In the first run a general solution was produced by generation 19, which contained 88 primitives. A simplified version of it is shown in Figure 1.

In contrast to earlier work [Zomorodian, 1995] where GP was obliged to evolved push down automata the evolved solution is effectively a register machine (n.b. less powerful that a push down automata). The evolved solution (cf. Figure 1) only uses a single integer memory cell (aux1), in which it counts the depth of nesting. At the end of a legal sentence this count must be zero, further should the brackets be unbalanced before the end is reached, this is recognised and aux1 is also used as a flag indicating this. This solution not only passes all the fitness tests and is a general solution to the problem but (given suitable redefinition of max) is a solution for sequences of any length.

Given the readily found general solution did not exhibit stack like behavior it was decided not to repeat this problem with a GP that had stack primitives.

Table 2: Nested Bracket Test Cases

Length	Positive		Negative	
1			all	2
2	all	1	all	3
3			all	8
4	all	2	all	14
5			random	4
6	all	5	random	5
7			random	5
8	all	14	random	14
9			random	14
10	all	42	random	42
Totals		64		111

3 Dyck Problem

Once again the problem is to recognise which sentences are correctly bracketed, however there are now four types of bracket pairs, `(`, `)`, `[`, `]`, {, }, ‘, ’. E.g. {}`[]` is correctly bracketed but `[`} is not. As with the nested brackets problem, a limit of ten symbols per sentence was assumed.

Two sets of experiments have been conducted, the first provided the GP with primitives which implement a stack for it and the second provided primitives like those from which it has been shown GP can evolve stack data structures [Langdon, 1995].

3.1 Architecture

In the first experiments (stack given) no ADFs were used, whilst in the second there are three ADFs, having 0, 1 and 0 arguments. It was hoped that these could evolve to operate like pop, push and top. Each could be called from the main tree, additionally the third (top?) could be called by the first.

The terminals, functions and control parameters used in these two experiments are as Section 2 except where given in Table 3. The differences between the two experiments in this section are shown in the middle and right hand columns of Table 3.

Table 1: Balanced Bracket Problem

Objective:	Find a program that classifies sequences of **(** (represented by 1) and **)** (-1) as being correctly nested or not.
Primitives: (any tree) (rpb, adf1) (rpb, adf2) (rpb only)	ADD, SUB, PROG2, IFLTE, Ifeq, 0, 1, -1, max, forwhile, i0 adf2, aux1, read, write, swap, Set_Aux1 arg1 adf1
Max prog size	$4 \times 50 = 200$. In initial population each tree is limited to 50 primitives.
Fitness cases:	175 fixed test cases.
Fitness:	Number of test cases passed (scalar).
Selection:	Tournament group size of 4 used for both parent selection and selecting programs to be removed from the population. Steady state population (elitist).
Wrapper:	Zero represents False (i.e. not in language) otherwise True.
Parameters:	Pop = 10,000, G = 50, 3×3 demes, no CPU penalty, no aborts.
Success predicate:	A program passes all tests.

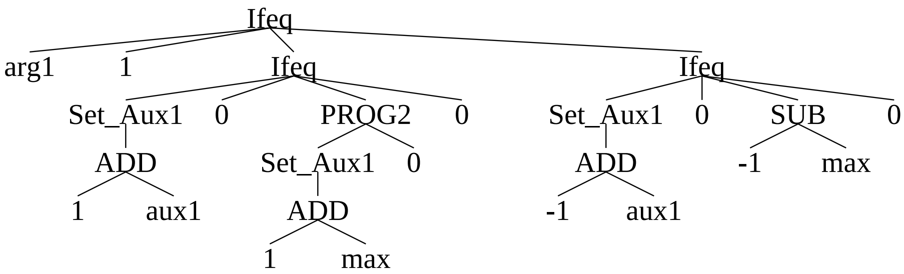

Figure 1: Solution to Bracket Problem (Simplified)

Table 3: Dyck Language Problem

Objective:	Find a program that classifies sequences of four types of bracket (**(** (represented as 5), **)** (71), [(13),] (103), { (31), } (137), ‘ (43) and ’ (167)) as being correctly nested or not.		
Primitives:	Common	Stack Given	Index Memory
All trees:	ADD, SUB, PROG2, IFLTE, Ifeq, 0, 1, max, aux1	Makenull, Empty, Top, Pop, Push	read, write, inc_aux1, dec_aux1
rpb: as all plus adf1: as all plus adf2: as all plus	ifopen, ifmatch, ARG1, Set_Aux1		adf1, adf2, adf3 adf3 arg1, arg2
Max prog size	Initial tree limit 50	50	$4 \times 50 = 200$
Fitness cases:	286 fixed test cases.		
Fitness:	Number of correct answers returned.		
Selection:	Tournament size 4 (After 1^{st} solution CPU penalty used giving a two dimensional fitness value, fitness niching used with a sample of up to 81 (9×9) nearest neighbours.)		
Wrapper:	Zero represents True (i.e. in language) and all other values False.		
Parameters:	Pop = 10,000, G = 50, Pareto, 3×3 demes, CPU penalty only after 1^{st} solution found, Abort on first error in sentence.		
Success predicate:	A program passes all tests.		

3.2 Fitness Function

The fitness of every trial program is determined by presenting it with a series of symbols from test sentences and counting how many times it correctly classifies each as to whether it is the last of a correctly balanced sequence.

This problem is more complex than that in Section 2 and so the test cases are longer. To constrain the increase in runtime, forwhile loops were not used.

The number of possible test cases is much larger than in Section 2 and so it was not practical to include sentences longer than eight symbols and even for lengths of six and eight symbols it was necessary to select (at random) positive test cases to include.

In a correctly matched sentence there will be equal numbers of opening and closing brackets of each type but this is unlikely to be true in a random sequence of brackets. To penalise partial solutions based on a counting mechanisms, like that evolved in Section 2, the test cases include examples where there are equal numbers but which are not correctly nested (referred to as "balanced" in Table 4).

As before it was not practical to include all cases and so longer negative examples (both balanced and not balanced) were selected at random. Even so the fitness tests are much longer than that in Section 2 and so to keep runtime manageable the number of times each program must be run was reduced by:

- Only using the first half of the test cases (i.e. tests up to length six) unless a program passes them all, were upon test sentences of length seven and eight were also used.
- Using the value returned for each symbol, rather than just the last in the sentence, and ignoring the rest of the sentence after the first error. As some shorter sentences are identical to the start of longer ones they need not be tested explicitly and so could be removed from the test cases. The test cases after removing such duplicates are summarised in the right hand side of Table 4.

Initially brackets were coded as $\pm 1, \pm 2, \pm 3, \pm 4$ but general solutions proved difficult to find. Instead, despite the use of "balanced" negative examples, partial solutions based upon summing up symbol values dominated. Since the purpose of the experiment was to investigate learning correct nesting of symbols rather than learning which symbols match each other the problem was simplified by providing the GP with two new primitives (ifmatch and ifopen, cf. Table 9) which say which symbols match each other. To further discourage partial solutions based on summing symbol values the symbols were recoded as prime values with no simple relationships between them (cf. Table 3).

After an individual which passes all the tests is found the GP run is allowed to continue using a modified fitness function which includes a CPU penalty. Each program's fitness now contains two orthogonal terms, the original score and the $\lfloor$average$\rfloor$ number of instructions run per program execution. Tournament selection is still used for reproduction and deletion but now uses Pareto comparison [Goldberg, 1989], so passing tests and using little CPU are equally important.

In the event of no member of the tournament being better than all the others the best members of the tournament are compared with a sample of the adjacent population. The one that is dominated the least by the rest of the population is selected. This introduces a selection pressure to be different from the rest the population so allowing high scoring and low CPU programs to co-exist. This is similar to fitness niching [Horn *et al.*, 1993].

3.3 Results

In three runs given the stack primitives general solutions were evolved by generation seven to 23 (in three identical runs but using simple non-demic (normal) populations, two runs produced solutions in generations 30 and 39). Evolution was allowed to continue after the first individual to pass all the tests was found. Under the influence of the CPU penalty faster but still general solutions were found. Figure 2 shows one of the fastest solutions produced in a run using demes after 50 generations. As in Section 2 the solutions are not only general solutions to the given problem, but given a deep enough stack would work with any sentences of any length.

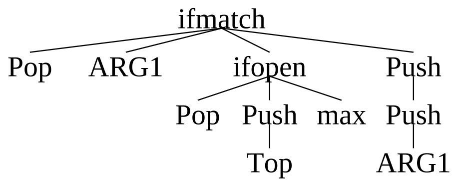

Figure 2: One of the Fastest Solutions to the Dyck Problem evolved after 50 Generation

In contrast none of 15 runs using the indexed memory primitives passed all the tests. Some of the more promising runs were extended beyond 50 generations up to 140 generations without finding a solution. The best (produced after 84 generations) still failed 3 tests (on sequences of up to six symbols). It showed some stack like behaviour which enables it to pass 13 of the tests of length seven and eight but also showed some signs of over fitting to the specific the test cases used rather than having learnt to solve the general problem.

A program which always returns zero (i.e. True) has a fitness of zero because it will always fail on the first symbol of each test sentence (a sentence of odd length must be unbalanced). In contrast a program which never returns zero will always be correct on the first symbol of each sentence and so will get the opportunity to be tested

Table 4: Dyck Language Test Cases

Length	Positive	Negative		After Removing Duplicates			
		Balanced	Random	Positive	Balanced	Random	Score
1		all 8					0
2	all 4	all 60				9	18
3			16			10	30
4	all 32	all 24	16	27	16		172
5			16			16	80
6	random 32	random 32	32	32	32	32	576
7			16			16	112
8	random 32	random 32	32	32	32	32	768
			Totals	91	112	83	1756

on the second symbol which it may also pass. For the actual test cases used a program which never returns zero has a fitness of 714. While aborting a test sentence on the first error reduces the number of times programs are run, it may also make it more difficult to evolve a solution. In both experiments the GP population quickly learns not to return zero, but when using indexed memory it appears to be more difficult than when given a stack to escape this local optima and learn to return zero at some points.

4 Reverse Polish Calculator

In this section the GP evolves a four function (+, −, / and ×) calculator, i.e. evaluates integer arithmetic expression. The problem is simplified by presenting the expression in Reverse Polish Notation (postfix), which avoids consideration of operator precedence, and by avoiding expressions which include divide by zero. No limit on the length of expressions was assumed, however the expressions tested were between three and fifteen symbols long (see Table 6).

Two sets of experiments were made, the first provided the GP with primitives which implement a stack for it and the second provided primitives like those from which it has been shown GP can evolve stack data structures. Since all storage including the supplied stack is initialised before the evolved programs can use it the Makenull operation is not needed. Therefore the terminal set was simplified by not including Makenull and Empty (which is also not needed) in these experiments.

4.1 Architecture

Each individual within the population consists of five separate trees (num, plus, minus, times and div) plus zero or two ADFs. As in sections 2 and 3 each test sentence is presented a symbol at a time to the GP, however in this case the appropriate tree is selected. E.g. if the symbol is an integer, then the num tree is executed with the integer as its argument. Each tree returns a value as the current value of the expression (num's answer is ignored).

In the first experiments (stack given) no ADFs were used, whilst in the second there are two ADFs, having 0 and 1 arguments respectively. It was hoped that these could evolve to operate like pop and push. Each could be called from the five main trees.

The terminals, functions and control parameters are as Section 3 except where given in Table 5. Nb. the long run times encountered with these experiments led to the decision to include a CPU penalty of $\lfloor$average$\rfloor$ number of primitives executed per program run. However initially only programs with long run times are penalised (by setting the penalty to zero for fast programs, i.e. where it was ≤ 50). Should a program be evolved which passes all the fitness test then the CPU penalty is increased by applying it to all programs.

4.2 Fitness Function

In each individual in the population a separate score is maintained for its five operations (num, plus, minus, times and div) plus the CPU penalty. Each time the individual returns the correct answer (and this is checked) the score for each of its operations that has been used since the last time its result was checked is incremented. As in Section 3, these scores are not combined and each contributes as a separate objective in multi-objective Pareto selection tournaments.

The fixed test cases were created before the GP was run. A number of test cases were devised by hand and the remainder were selected at random. However the randomly selected data values (from the range −99...+99) proved to be unsatisfactory for expressions containing "/" because division of two randomly selected integers has a high chance of yielding zero or an integer near it and therefore were changed by hand.

Table 5: Reverse Polish Notation (RPN) Expression Evaluation Problem

Objective:	Find a program that evaluates Reverse Polish (postfix) arithmetic expressions.		
Primitives:	Common	Stack Given	Index Memory
$+-\times/$ trees:	ADD, SUB, MUL, DIV, PROG2, 0, 1, aux1, Set_Aux1	Top, Pop, Push	read, write, inc_aux1, dec_aux1, adf1, adf2
num: as ops plus	arg1		
adf1: as ops but			no adfs
adf2: as ops but			no adfs and add arg1
adf3: as ops but			no adfs and add arg1, arg2
Max prog size	Initial tree limit 50	$5 \times 50 = 250$	$7 \times 50 = 350$
Fitness cases:	127 fixed test cases.		
Fitness:	Number of correct answers returned.		
Selection:	Pareto tournament size 4, CPU penalty (initial threshold 50 per operation), fitness niching used with a sample of up to 81 other members of the population.		
Wrapper:	Value on num ignored. No wrapper on other trees.		
Parameters:	Pop = 10,000, G = 50, Pareto, no demes, no threshold on CPU penalty only after 1^{st} solution found, abort on first wrong answer given in expression.		
Success predicate:	A program passes all tests.		

Table 6: Length of Reverse Polish Expressions at each Point where Answers are Checked in the Fitness Test Cases

length	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	Total
No. of cases			10	3	55	27	44	2	36	1	5		8		3	194

The following rules were used to create the test cases:

- include a higher proportion of minus and divide (cf. Table 7).
- be deeply nested (Table 8).
- divide by zero was avoided.
- Where necessary randomly generated data values were manually changed so that only a few divisions yield values in the range $-3\ldots+3$.
- To avoid problems with overflow, randomly generated expressions did not allow: arguments to addition or subtraction outside the range $-10^8\ldots+10^8$ or arguments to multiplication or division outside the range $-65535\ldots+65535$.
- Data values set by hand were chosen so neither: the product of two arguments of divide nor the square of the second argument exceeded 2,147,483,647.
- In all but one test expression there is one less arithmetic operation than data values. So only the final result should be left after evaluating the expression.

  One case with fewer arithmetic operation was included. In this case there should be multiple data values left after evaluating the expression.

As before it was necessary to constrain run time. This was done by checking answers during the evaluation of each expression and aborting evaluation following the first error detected and removing test cases which essentially duplicated others. This left 127 test expressions which include 194 points where the trial program's answer is checked.

Table 7: Reverse Polish Expression Test Cases

Operation	No.	Max Score
num	550	163
plus	67	58
minus	103	85
times	85	64
divide	156	127
	420	
Totals	970	497

Table 8: Depth of Expression Nesting in RPN Test Cases (depth after each symbol)

depth	1	2	3	4	5	6	Total
number	387	390	149	31	12	1	970

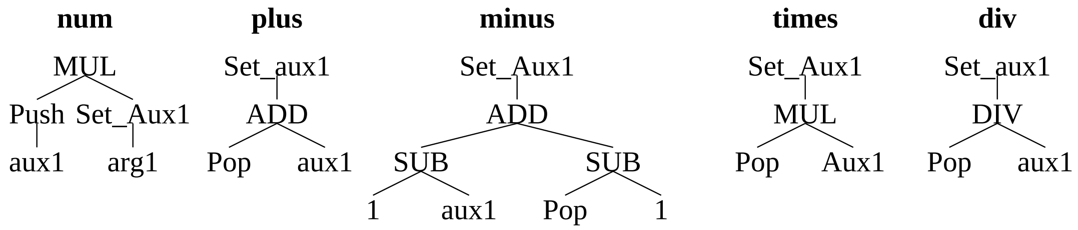

Figure 3: One of the Fastest Solutions to the RPN Problem evolved after 50 generations

4.3 Results

In eleven runs using stack primitives, six produced solutions which passed all the tests, these were found between generations 11 and 23. In four cases the first programs to pass all the tests were also general solutions to the problem. In the other two the first solutions failed special cases such as $1-1$ and $x/y=0$, however in both runs general solutions were evolved before 50 generations.

Under the action of the increased CPU penalty, solutions which took about one third of the CPU time of the first solution found were evolved (see Figure 3).

In 59 runs with stack primitives replaced by indexed memory (see right hand side of Table 5) no program passed all the tests. The highest no. tests passed (148 of 194) was achieved by a program which used the first ADF to implement DIVR (i.e. standard divide but with the arguments in reversed order, see Table 9) and the second to approximate both push and pop on a three level stack. Other unsuccessful trials included adding a third ADF (with two arguments) in the hope that this might evolve the DIVR functionality (best 102 in 33 runs, of which 16 ran out of time before 50 gens) and supplying SUBR and DIVR functions (in place of SUB and DIV) where the best score was 116, in 38 runs.

The non-commutative functions ($-$ and $/$) appear to be more difficult to evolve than commutative ones because the arguments on the stack are in the opposite order to that used by the SUB and DIV functions. (The problem can be readily solved, when given the stack primitives, by replacing SUB and DIV by SUBR and DIVR which process their arguments in the opposite order, i.e. the order on the stack.) The div tree has to use some storage (i.e. aux1) to reverse the order of the stack arguments (sub can simply negate the result of performing the operation in the wrong order to get the right answer). The need to use aux1 makes the problem deceptive, in that many programs can obtain high scores using aux1 as the top of the stack and only fail tests which require deeper use of the stack.

Some of the difficulty the GP with indexed memory found may have been due to trying to use aux1 both as stack pointer (for which inc_aux1 and dec_aux1 make it suitable) and as the top of stack (as evolved in many cases where the stack primitives were given).

5 Conclusions

These experiments have shown GP can solve two new problems (inducing a Dyck language (Section 3) and evaluating Reverse Polish Notation (RPN) expressions, Section 4) and GP can solve the nested bracket problem (Section 2) without requiring an intermediate step generating an abstract machine.

All three examples were solved by GP using the appropriate data structure for the problem. The two more complex examples (Dyck and RPN) proved to be more difficult (no solution has been found to date) for GP when provided with indexed memory than when provided with a stack, despite indexed memory being more powerful.

It has often been argued, e.g. [Kinnear, Jr., 1994, page 12], that functional primitives used with GP should be as powerful as possible, in these examples we have shown appropriate data structures are advantageous, that is GP can benefit from data abstraction.

The failure of GP to solve the problems when provided with the more general (i.e. more powerful) directly addressable memory data structure, is worrying as it shows (at present) data structures should be chosen with care and it may not be sufficient to simply over provide, with more powerful structures than are needed. It was hoped that GP could evolve appropriate data structures as it solved the problem but this has yet to be demonstrated.

A Pareto CPU penalty term to fitness has again been used to curb growth in program size and CPU time.

Acknowledgments

W. B. Langdon is funded by the EPSRC and the National Grid Company plc. I would like to thank my supervisors (M. Levene and P. C. Treleaven), and Andy Singleton for the original version of GP-QUICK on which my code was based.

Table 9: Actions Performed by Terminals and Functions

Primitive	Purpose
DIV(x,y)	**if** $y \neq 0$ **return** x/y **else return** 1
SUBR(x,y) DIVR(x,y)	As SUB and DIV except yield $y-x$ and y/x, i.e. operands reversed.
max	constant 10 ($\geq$ max input size).
PROG2(t,u)	evaluate t; **return** u
ARG1, arg1, arg2	arguments of current operation or ADF
aux1	an auxiliary variable (i.e. in addition to indexed memory).
Set_Aux1(x)	aux1 = x; **return** aux1

forwhile(s,e,l)	**for** i0 = s; i0 $\leq$ e; i0++
	    if timeout (128) **exit loop**
	    if l returns zero **exit loop**
	return i0

i0	Yields value of loop control variable of most deeply nested loop or zero if not in a loop in current tree. N.b. loop control variable in one tree cannot be accessed in another (e.g. an ADF).
IFLTE(x,y,$t1$,$t2$)	**if** $x \leq y$ **return** $t1$ **else return** $t2$
Ifeq(x,y,$t1$,$t2$)	**if** $x = y$ **return** $t1$ **else return** $t2$
ifopen(x,$t1$,$t2$)	**if** $x =$ 5, 13, 31 or 43 **return** $t1$ //i.e. opening symbol **else return** $t2$

ifmatch(x,y,$t1$,$t2$)	**if** $x =$ 5, 13, 31 or 43 evaluate y //i.e. opening symbol
	    if $(x,y) =$ (5,71), (13,103), (31,137) or (43,167) **return** $t1$ //x and y match
	    else return $t2$
	else return $t2$

Makenull	clear stack; **return** 0
Empty	**if** stack is empty **return** 0; **else return** 1
Top	**if** stack is empty **return** 0; **else return** top of stack
Pop	**if** stack is empty **return** 0; **else** pop stack and **return** popped value
Push(x)	Evaluate x; **if** $<$ 99 items on stack push x; **return** x **else return** 0
Indexed memory is held in store[$-l$ ... $+l$], where $l = 63$, i.e. a total of 127 cells.	
read(x)	**if** $\|x\| \leq l$ **return** store[x] **else return** 0
write(x,d)	**if** $\|x\| \leq l$ store[x] = d; **return** original contents of store[x] **else** evaluate d; **return** 0
swap(x,y)	**if** $\|x\| \leq l$ **and** $\|y\| \leq l$ exchange contents of store[x] and store[y] **if** $\|x\| > l$ **and** $\|y\| \leq l$ store[y] = 0 **if** $\|x\| \leq l$ **and** $\|y\| > l$ store[x] = 0 **return** 1

References

[Andre, 1994] David Andre. Evolution of mapmaking ability: Strategies for the evolution of learning, planning, and memory using genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 250–255, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Collins, 1992] Robert J. Collins. *Studies in Artificial Evolution*. PhD thesis, UCLA, Artificial Life Laboratory, Department of Computer Science, University of California, Los Angeles, LA CA 90024, USA, 1992.

[Dunay and Petry, 1995] Bertrand Daniel Dunay and Frederic E. Petry. Solving complex problems with genetic algorithms. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 264–270, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Dunay *et al.*, 1994] B. D. Dunay, F. E. Petry, and W. P Buckles. Regular language induction with genetic programming. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, pages 396–400, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

[Goldberg, 1989] David E. Goldberg. *Genetic Algorithms in Search Optimization and Machine Learning*. Addison-Wesley, 1989.

[Horn *et al.*, 1993] Jeffrey Horn, Nicholas Nafpliotis, and David E. Goldberg. Multiobjective optimization using the niched pareto genetic algorithm. IlliGAL Report no. 93005, Illinois Genetic Algorithm Laboratory, University of Illinois at Urbana-Champaign, 117 Transportation Building, 104 South Mathews Avenue, Urbana, IL 61801-2296, July 1993.

[Iba *et al.*, 1995] Hitoshi Iba, Hugo de Garis, and Taisuke Sato. Temporal data processing using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 279–286, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Kinnear, Jr., 1994] Kenneth E. Kinnear, Jr. A perspective on the work in this book. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 1, pages 3–19. MIT Press, 1994.

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

[Langdon, 1995] W. B. Langdon. Evolving data structures using genetic programming. In L. Eshelman, editor, *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*, pages 295–302, Pittsburgh, PA, USA, 15-19 July 1995. Morgan Kaufmann.

[Langdon, 1996] William B. Langdon. Data structures and genetic programming. In Peter J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, chapter 20. MIT Press, 1996.

[Tackett, 1994] Walter Alden Tackett. *Recombination, Selection, and the Genetic Construction of Computer Programs*. PhD thesis, University of Southern California, Department of Electrical Engineering Systems, 1994.

[Teller, 1994] Astro Teller. The evolution of mental models. In Kenneth E. Kinnear, Jr., editor, *Advances in Genetic Programming*, chapter 9, pages 199–219. MIT Press, 1994.

[Zomorodian, 1995] Afra Zomorodian. Context-free language induction by evolution of deterministic pushdown automata using genetic programming. In E. S. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 127–133, MIT, Cambridge, MA, USA, 10–12 November 1995. AAAI.