7 PROBLEMS SOLVED USING DATA STRUCTURES

In this chapter we show that data abstraction can be beneficially used within genetic programming (GP). Work so far [Teller, 1994a; Andre, 1994b; Brave, 1995; Jannink, 1994] shows GP can automatically create programs which explicitly use directly addressable (indexed) memory to solve problems and Chapters 4, 5 and 6 demonstrate that GP can automatically generate abstract data structures such as stacks, queues and lists. In this chapter 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. Section 7.4 reviews published GP work where explicit memory is used and concludes that in most successful cases data structures appropriate to the problem have been provided for the GP (although the experimenter may not have used the term "data structure").

Three example problems are presented. In each the task is to induce a program which processes a context free language given training samples of the language. We chose problems that should be solvable using stack data structures as stacks were the easiest of the data structures investigated in Chapters 4, 5 and 6 to evolve. In general, data structures at least as powerful as stacks are required to process context free languages.

In Section 7.1 GP evolves a program which classifies sequences of brackets as being correctly or incorrectly nested. Section 7.2 evolves programs which classify sequences

of multiple types of bracket as being correctly nested or not (a Dyck language) and Section 7.3 evolves programs which evaluate Reverse Polish (postfix) expressions. The structure of Sections 7.1, 7.2 and 7.3 is based on the structure of Chapters 4, 5 and 6. For example Sections 7.1.1, 7.2.1 and 7.3.1 each contain the problem statement for one of the three problems. Section 7.5 summarises this chapter.

7.1 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; Longshaw, 1997; Slavov and Nikolaev, 1997], deterministic push-down automata [Zomorodian, 1995], machines composed of simple Turing machines [Dunay and Petry, 1995; Petry and Dunay, 1995] or special memory nodes within the tree [Iba et al., 1995]. While [Falco et al., 1997] uses GP to generate a number of formal languages. However [Koza, 1992, page 442] recasts a simple language recognition problem in terms of classifying DNA sequences as *introns* or *exons* and shows GP can evolve a correct program for this task and [Wyard, 1991; Wyard, 1994; Lucas, 1994] use GAs operating on formal grammar rules of various types to induce grammars for a number of regular and context free languages. In contrast we wish to use the task of evolving a language recogniser to investigate the impact of providing data structures versus indexed memory, and so we follow normal GP practice and our GP executes the GP tree directly i.e. treats it as a program.

In this section we show GP can solve the balanced bracket problem directly when given an appropriate data structure ([Zomorodian, 1995] previously solved this problem using GP to evolve a specification for a pushdown automaton, [Wyard, 1991] used a GA operating on formal grammar rules to induce a grammar for it and [Lankhorst, 1995] used a fixed representation GA to specify a pushdown automaton, while [Sun et al., 1990] solved it by training a neural network in combination with a stack). The balanced bracket language is a context free language and so can be recognised by a pushdown automaton (which implies use of a stack) and not a regular language, which could be recognised by a finite state machine. However a pushdown automaton is not required, the balanced bracket language can be recognised by an intermediate machine, a finite state automaton with a counter. The solution found by GP was of this form. In a run where both index memory and register memory were available, the evolved solution used the register memory, NB GP selected the appropriate data structure for the problem.

7.1.1 Problem Statement

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.

7.1.2 Architecture

Two automatically defined functions (ADFs) (see Section 2.3.2 for an introduction to ADFs) are available to assist the main result producing branch (or tree). The first,

Table 7.1. Tableau for Balanced Bracket Problem

Objective	Find a program that classifies sequences of **(** (represented by 1) and **)** (-1) as being correctly nested or not.
Architecture	Main tree, adf1 (no arguments) and adf2 (one argument)
Primitives (any tree)	ADD, SUB, PROG2, IFLTE, Ifeq, 0, 1, -1, max, forwhile, i0
(rpb, adf1)	adf2, aux1, read, write, swap, Set_Aux1
(rpb, adf2)	arg1
(rpb only)	adf1
Max prog size	$4 \times 50 = 200$. In initial population each tree is limited to 50 primitives.
Fitness case	175 fixed test examples, cf. Table 7.2
Fitness Scaling	Number of test examples correctly classified (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).
Hits	Number test sentences correctly classified
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	Fitness ≥ 175

adf1, has no arguments and has the same terminal and function sets as the main tree. However as it does not have any arguments, it does not use the primitive arg1.

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. (Caches of ADF values were also used in Chapter 5, cf. Table 5.11 (page 118). See also Section D.6).

7.1.3 Choice of Primitives

Table 7.1 shows the parameters used and the terminals and functions provided, NB they include indexed memory but not stacks.

For ease of comparison the same sized indexed memory and stacks were used in all three sets of experiments in this chapter. Both were deliberately generously sized to avoid restricting the GP's use of them. The indexed memory consisted of 127 memory cells, addressed as $-63 \ldots +63$, and the stack allowed up to 99 32-bit signed integers to be pushed. As in the previous chapters, memory primitives had defined behaviour which allows the GP to continue on errors (e.g. popping from an empty stack or writing to a non-existent memory cell). All stored data within the program is initialised to zero before the start of each test sentence. Tables 7.9 and 7.10 (pages 165–166) give the actions of terminals and functions used in this chapter.

Table 7.2. Number of correctly nested and incorrectly nested bracket test sentences of each length used in the nested bracket test case. Longer incorrect sentences were chosen at random from all the possible incorrect sentences of the same length.

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

7.1.4 Fitness Function

The fitness of each trial program was evaluated on a fixed set of 175 example sentences containing both correctly nested (positive tests) and incorrectly nested brackets (negative tests). 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 7.2). 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 `)`). The value returned by the program on the last symbol of the sentence gives its verdict as to whether the sequence is correctly nested, i.e. the value returned by the program is ignored, except on the last symbol of each test sentence.

This test case and the test cases used in Sections 7.2.4 and 7.3.4 are available via anonymous ftp. Section D.9 gives the network addresses.

7.1.5 Parameters

The default values for parameters given in Section D.3 were used except the population size and the maximum program length. The parameters used are summarised in Table 7.1.

Earlier work (cf. Chapter 5) had shown even a large population had a great tendency to converge to partial solutions which effectively trapped the whole population preventing further progress. In this (and the following section) the population was partitioned into demes so crossover is restricted to near neighbours in order to reduce the speed of convergence (see Section 3.8). As in Chapter 5 the population is treated as a 50×100 torus with two members of the population per square on its surface. Each time a new individual is created, a 3×3 square neighbourhood 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 [Tackett, 1994; Collins, 1992].

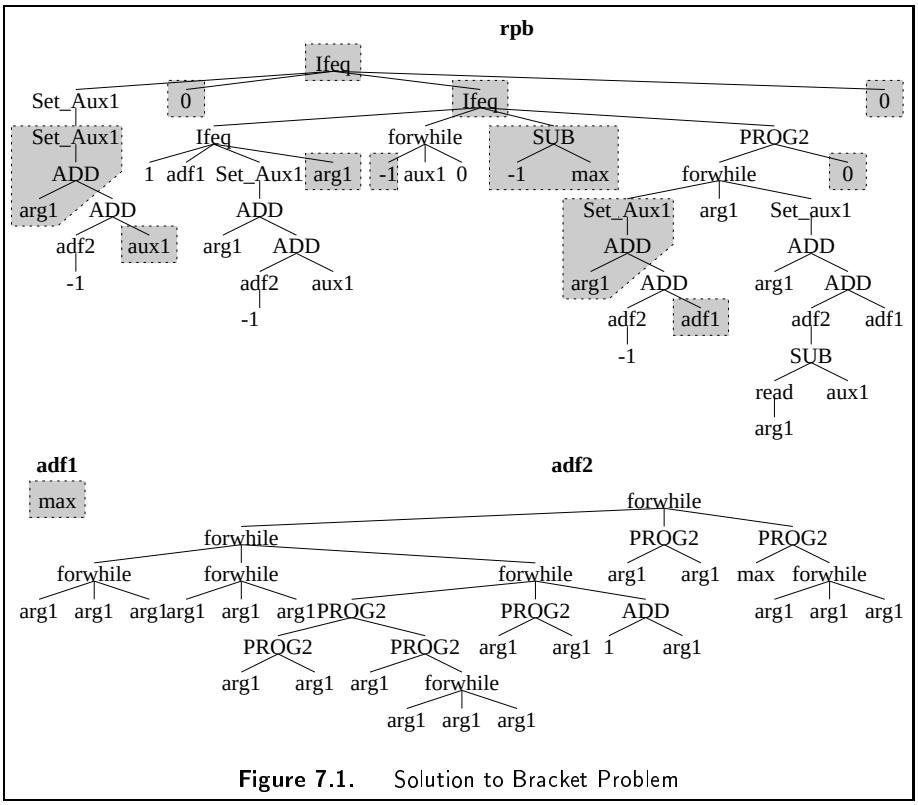

Figure 7.1. Solution to Bracket Problem

7.1.6 Results

In the first run a general solution was produced by generation 19, which contained 88 primitives. This is shown in Figure 7.1 and a simplified version is shown in Figure 7.2.

In contrast to earlier work [Zomorodian, 1995], where GP was obliged to evolve pushdown automata, the evolved solution is effectively a finite state machine with a counter (NB less powerful than a pushdown automaton). The evolved solution (cf. Figure 7.2) 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.

To find the solution given in Figure 7.1 required $19 \times 10,000 = 190,000$ individuals to be processed. This is similar to that required in [Zomorodian, 1995] where a solution was found in generation 24 with a population of 3,000. ($24 \times 3,000 = 72,000$).

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

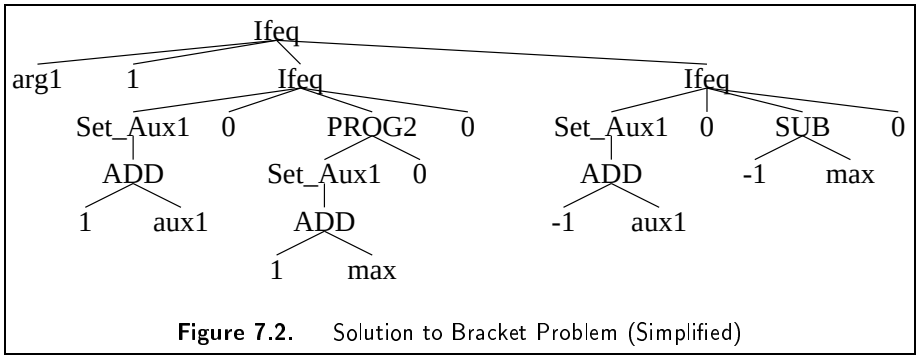

Figure 7.2. Solution to Bracket Problem (Simplified)

7.2 DYCK LANGUAGE

In this section we apply genetic programming to a solve a new problem, that of recognising a Dyck language. Two sets of experiments were conducted, the first provided the GP with primitives which implement a stack for it and the second provided indexed memory and other primitives like those from which it has been shown GP can evolve stack data structures, cf. Chapter 4. The same fitness function, population size and other parameters were used in both sets of experiments. Solutions were readily found when the GP was provided with a stack data structure but no solutions have been found when using indexed memory.

The Dyck problem was chosen as Dyck languages are context free languages and require machines at least as powerful as pushdown automata (i.e. stacks) to solve them. Dyck languages are generalisations of the balanced bracket problem to multiple types of bracket.

7.2.1 Problem Statement

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.

7.2.2 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 (which it was hoped might evolve to act like top) could be called by the first.

7.2.3 Terminals, Functions and Parameters

The terminals, functions and control parameters used in these two experiments are as Section 7.1 except where given in Table 7.3. The differences between the two experiments in this section are shown in the middle and right hand columns of Table 7.3.

The five stack primitives are based on the definition of a stack given in Table 4.1 (page 63), however they have been made more rugged by ensuring their behaviour is defined in all circumstances, i.e. including errors such as popping from an empty stack. Their behaviour is defined at the end of this chapter in Tables 7.9 and 7.10.

This problem is more complex than that in Section 7.1 and so the test case is longer. To constrain the increase in run time, forwhile loops were not used.

7.2.4 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

Table 7.3. Tableau for 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	ifopen, ifmatch, ARG1, Set_Aux1		adf1, adf2, adf3
adf1: as all plus			adf3
adf2: as all plus			arg1, arg2
Max prog size	Initial tree limit 50	50	$4 \times 50 = 200$
Fitness Case	286 fixed test examples, cf. Table 7.4		
Fitness Scaling	Number of correct answers returned.		
Selection	Tournament size 4 (After first solution CPU penalty used giving a two dimensional fitness value, fitness niching used with a sample of up to 81 (9×9) nearest neighbours).		
Hits	Number test symbols correctly classified.		
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 first solution found, Abort on first error in sentence.		
Success predicate	Hits $\geq$ 1756, i.e. all answers correct.		

as to whether it is the last of a correctly balanced sequence. All memory is initialised to zero before the start of each test sentence.

Test Case. The number of possible test sentences of a particular length is much larger than in Section 7.1 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 examples 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. If the only negative examples are random sequences of symbols, a program could correctly guess most answers just by considering if there are equal numbers of each pair of bracket. We anticipate that such programs can be readily evolved, for example the program that evolved in Section 7.1 does this. However it may be anticipated that evolving complete solutions from such partial solutions will be very difficult. (Chapter 8 suggests the evolution of correct stacks is made harder by the presence of "deceptive" partial solutions). To penalise such partial solutions the test case included examples where there are equal numbers but which are not correctly nested (referred to as "Balanced" in Table 7.4).

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 7.1 and so to keep run time manageable the number of times each program must be run was reduced by:

- Only using the first half of the test case (i.e. tests up to length six). However if a program passes all the shorter tests then it was also tested on test sentences of length seven and eight. Thus most of the time the second half of the test case is not used. It is only used by programs that are nearly correct, which evolve later in the GP run.

- In the first experiments in this chapter, each program is only tested at the end of each test sentence. In these experiments the value returned for each symbol is used. If a wrong answer is returned the the rest of the sentence is ignored. This reduces run time as in many cases only part of the test sentence is processed.

  Some shorter sentences are identical to the start of longer ones and so they need not be tested explicitly as the same actions will be performed as part of a longer test. Therefore such duplicates were removed from the test case. The test case after removing such duplicates are summarised in the right hand side of Table 7.4.

Symbol Coding. 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 7.10) 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 7.3).

Table 7.4. Number of correctly and incorrectly nested test sentences in the Dyck language test case. The incorrect test sentences are divided into those with the correct number of each type of bracket but which are in the wrong order (referred to as "Balanced") and others (referred to as "Rand"). Longer sentences were chosen at random. The right hand side of the table gives the number in each category actually used in the Dyck test case, i.e. after removing duplicates.

Length	Positive	Negative		After Removing Duplicates			
		Balanced	Rand	Positive	Balanced	Rand	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	rand 32	rand 32	32	32	32	32	576
7			16			16	112
8	rand 32	rand 32	32	32	32	32	768
			Totals	91	112	83	1756

Evolving Improved Solutions. The combination of Pareto fitness, a CPU penalty and fitness niches introduced in Chapter 6 (Section 6.5.3) was used in these experiments. Briefly 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 $\lfloor \text{mean} \rfloor$ number of instructions run per program execution. Tournament selection is still used for reproduction and deletion but now uses Pareto comparison (see Section 3.9), so passing tests and using little CPU are equally important. The fitness sharing scheme described in Section 6.5.3 was used. This introduces a secondary selection pressure to be different from the rest the population so allowing high scoring and high CPU programs to co-exist with programs with lower scores but using less CPU. This may reduced premature convergence.

7.2.5 Results

In three runs given the stack primitives general solutions were evolved by generation 7 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 (see Figure 7.3). Figure 7.4 shows the first solution to evolve in a run using demes and Figure 7.5 shows one of the fastest solutions produced in the same run after 50 generations. As in Section 7.1 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.

As all runs given stack primitives and using demes succeeded in finding a solution the best (i.e. most likely) estimate of the number of runs required to be assured (with

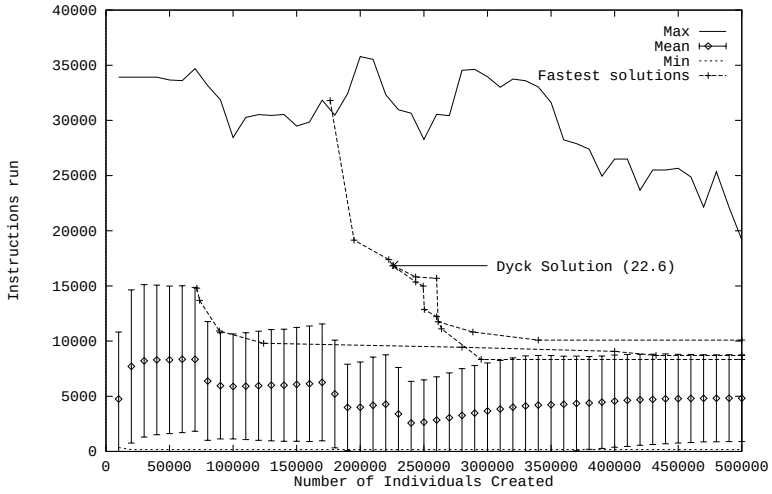

Figure 7.3. Evolution of the number of primitives executed during fitness testing on the Dyck problem, means of 3 runs using demes. Error bars indicate one standard deviation either side of the population mean. The fastest solutions evolved in each run are also plotted. The minimum number of instructions are executed by relatively unsuccessful programs as these are run on few tests.

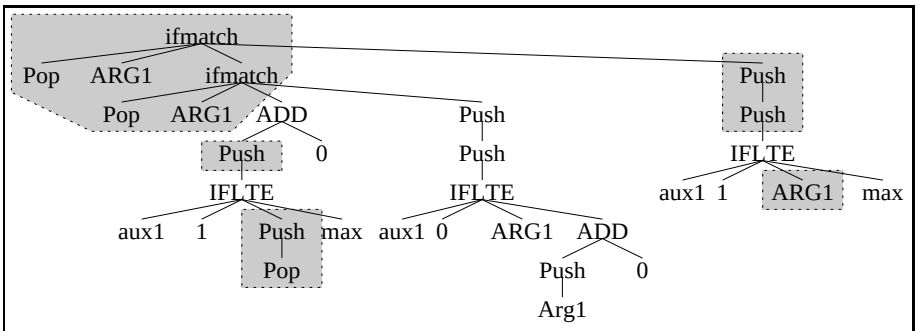

Figure 7.4. First Solution to the Dyck Problem. Evolved after 22.6 Generations

99% probability) of obtaining at least one solution is one. This would require a total of up to $23 \times 10^4 \times 1 = 2.3\,10^5$ trial programs.

In contrast none of 15 runs using the indexed memory primitives passed all the tests. (The probability of the difference between the two experiments being due to chance is $\ll 0.1\%$). 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

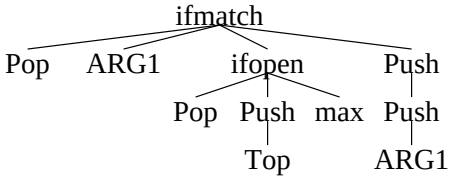

Figure 7.5. One of the Fastest Solutions to the Dyck Problem evolved after 50 Generations

but also showed some signs of over fitting to the specific test case 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 on the second symbol which it may also pass. For the actual test case 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.

7.3 EVALUATING REVERSE POLISH EXPRESSIONS

In this section we describe the final comparison of appropriate data structures and indexed memory. Once again solutions are readily evolved when the appropriate data structure is provided but no solutions have been found when using indexed memory.

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.

7.3.1 Problem Statement

In this section the GP evolves a four function ($+$, $-$, $/$ and $\times$) 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 division by zero. No limit on the length of expressions was assumed, however the expressions tested were between three and fifteen symbols long (see Table 7.6).

7.3.2 Architecture

The multi-tree architecture and multi-tree crossover described in Section 3.6 and employed in Chapters 4, 5 and 6 was used. This allows trees within each individual to evolve to specialise in solving one of the operations that form the complete calculator program. Each individual within the population consists of five separate trees (num,

plus, minus, times and div) plus either zero or two ADFs. As in Sections 7.1 and 7.2 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. Both ADFs could be called from the five main trees.

7.3.3 Terminals, Functions and Parameters

The terminals, functions and control parameters are as Section 7.2 except where given in Table 7.5.

Fears that run time might prove to be excessive led to the decision to remove some unnecessary primitives from the function and terminal sets. 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.

7.3.4 Fitness Function

In each individual in the population a separate score is maintained for its five operations (num, plus, minus, times and div) plus a CPU penalty. Each time the individual returns the correct answer (and it 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 7.2, these scores are not combined and each contributes as a separate objective in multi-objective Pareto selection tournaments.

Test Case. The fixed test case was created before the GP was run. Part of the test case was devised by hand and the remainder was selected at random. However randomly selected data values (from the range $-99 \ldots +99$) 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 data values were changed by hand. (Less than one in eight divisions of randomly chosen values will yield a value of 4 or more or -4 or less).

The following rules were used to create the test case:

- It was expected that as minus and divide are not commutative they would be the most difficult operations to evolve and therefore the test case include a higher proportion of minus and divide than the other two arithmetic operations (cf. Table 7.7).
- The test case was designed to include deeply nested expressions (cf. Table 7.8) as it was anticipated otherwise non-general partial solutions only able to evaluate simple expressions, which could be evaluated without using a stack, would predominate.
- To avoid consideration of exception handling, and its associated complexity, divide by zero was deliberately excluded from the test case.

Table 7.5. Tableau for Reverse Polish Notation (RPN) Expression Evaluation Problem

Objective	Find a program that evaluates integer 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 Case	127 fixed test expressions, cf. Tables 7.6, 7.7 and 7.8.		
Fitness Scaling	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.		
Hits	Number of correct answers returned.		
Wrapper	Value on num ignored. No wrapper on other trees.		
Parameters	Pop = 10,000, G = 100, Pareto, no demes, CPU penalty (increased after 1^{st} solution found), abort on first wrong answer given in expression.		
Success predicate	Fitness $\geq$ 194, i.e. a program passes all tests.		

Table 7.6. Length of reverse polish expressions at each point where answers are checked in the fitness test case.

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

- 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 $-65535 \ldots +65535$.
- Also to avoid overflow problems, 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.
- Most test expressions were well formed, with exactly the right number of data values for the number of operators (and vice-versa). (Since all four operators are binary this means there is one more data value than the number of operators in the expression). However, to test generality, one expression with fewer arithmetic operations 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 examples which essentially duplicated others. This left 127 test expressions which include 194 points where the trial program's answer is checked.

CPU Penalty. The long run times encountered with these experiments led to the decision to include a CPU penalty of $\lfloor \text{mean} \rfloor$ number of primitives executed per program run. Unlike the previous section, this CPU penalty was applied from the start of each run. However initially only programs with long run times are penalised (by ignoring the penalty where it was ≤ 50. This was implemented by setting the penalty is zero for such fast programs). Should a program be evolved which passes the whole fitness test case then the CPU penalty is increased by applying it to all programs.

7.3.5 Results

In eleven runs using stack primitives, six produced solutions which passed all the tests, these were found between generations 11 and 23 (see Figure 7.6). 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 $x/y = 0$ (which were not included in the test case), however in both runs general solutions were evolved less than 12 generations later (before 34 generations).

Table 7.7. Number of times each tree occurs in reverse polish expression (RPN) test case and the score it has when the whole test case is passed.

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

Table 7.8. Number of symbols (i.e. operators or numbers) used in the RPN test case for each level of expression nesting. (Depth of nesting calculated after the symbol has been processed).

depth	1	2	3	4	5	6	Total
No. of cases	387	390	149	31	12	1	970

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. Figure 7.7 shows one of the first general solutions to be evolved and Figure 7.8 shows one of the fastest solutions evolved at the end of the same run.

In 59 runs with stack primitives replaced by indexed memory (see right hand side of Table 7.5) no program passed all the tests. (NB the probability of the difference between the two experiments being due to chance is $\ll 1\%$). The highest number of 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 7.9) 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 leaving the other ADFs free to implement push and pop (best 102 in 33 runs, of which 16 ran out of time before 50 generations) and supplying SUBR and DIVR functions (in place of SUB and DIV) where the best score was 116, in 38 runs.

The probability of a general solution being found by generation 23 when given the stack primitives is best estimated at $4/11$. Using Equation 4.1 (page 75) the number of GP runs required to be assured (with 99% probability) of obtaining at least one solution is 11. This would require a total of up to $23 \times 10^4 \times 11 = 2.53\ 10^6$ trial programs.

Discussion. 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,

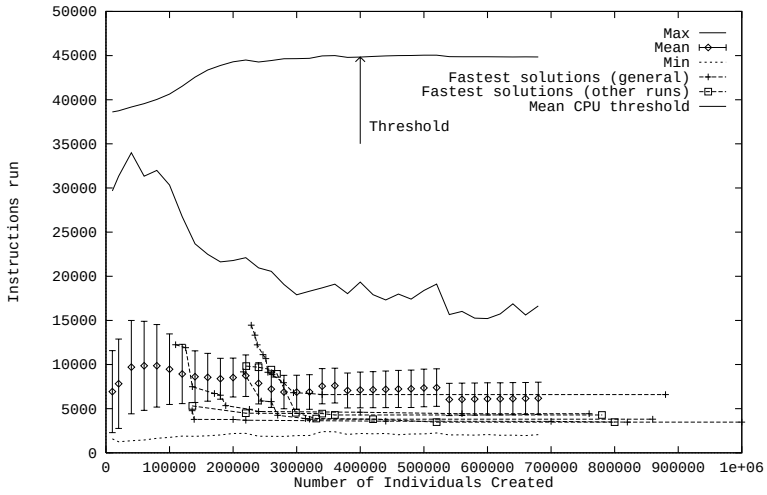

Figure 7.6. Evolution of the number of primitives executed during fitness testing on the calculator problem, means of 11 runs. (Average data is not plotted after generation 70 as several runs run out of time by this point). Error bars indicate one standard deviation either side of the population mean. The fastest solutions evolved during the six successful runs and the population mean of the threshold above which the CPU penalty is applied are also shown. The minimum number of instructions are executed by relatively unsuccessful programs as these are run on few tests.

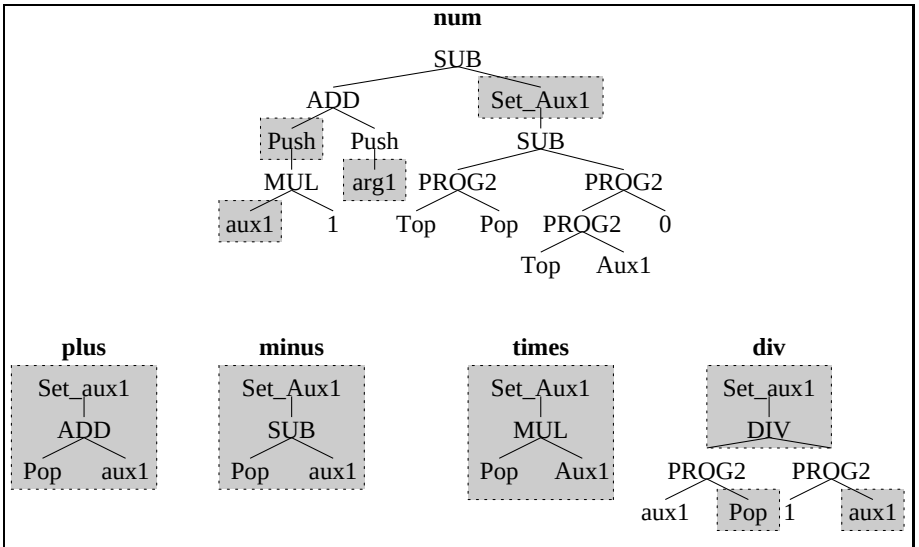

Figure 7.7. First Solution to the RPN Problem. Evolved after 11.0 generations (12240 instructions to complete test case)

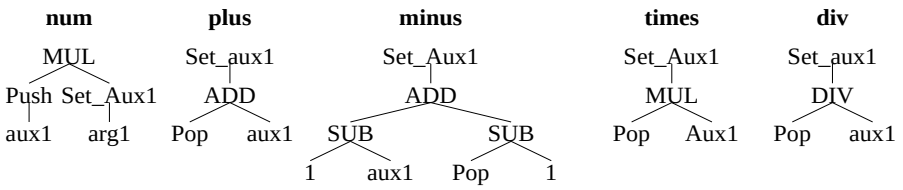

Figure 7.8. One of the Fastest Solutions to the RPN Problem evolved after 76 generations (4842 instructions to complete test case)

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). If this is case better results might be obtained by adding a second auxiliary variable (aux2) so these two uses could be split between two variables.

The top curve on Figure 7.6 shows the initial CPU penalty threshold, i.e. before a solution has been found. This shows on average the CPU threshold is higher than the average maximum CPU used by any individual in the population. While this means the CPU penalty has a small effect, the effect need not be negligible since any program which does exceed the threshold automatically has poor fitness and so is likely to be removed quickly from the population (and so not be included in these statistics). I.e. the penalty may still be effective in constraining growth in elapse time and program size (often described as “bloat”).

Contrasting Figure 7.6 with a similar plot for the list problem (Figure 6.3, page 132) we see in the list the CPU penalty is much more constraining, despite the threshold being set at 120 per test rather than 50. This is probably simply due to the presence of the forwhile primitive in the function set but may also be due in part to the problem requiring more primitives to solve it (fastest evolved solution 29.1 per test versus 7.0 for the calculator).

7.4 WORK BY OTHERS ON SOLVING PROBLEMS WITH MEMORY

This section briefly reviews published work on solving problems using GP which includes memory primitives. In most successful cases data structures appropriate to the problem have been used although the term “data structure” may not have been. The principle exception is Teller’s signal processing system PADO. This section groups publications according to memory structure, starting with the simplest and finishes with consideration of PADO.

7.4.1 Scalars

[Cramer, 1985] showed programs which use simple scalar memory could be evolved, however the paper concentrates upon program representation not use of memory. [Huelsbergen, 1996] solved the same problem, albeit with different primitives etc., but also uses simple scalar memory. Huelsbergen also shows the problem can be solved by random search in a practical time.

[Koza, 1992, page 470] presents an example where a single variable is used to maintain a running total during execution of a loop. While in [Koza, 1994, page 512] a small number of variables are used in a protein classification problem where the program processes proteins sequentially, a residue at a time. The variables provide the ability to store information about previous parts of the protein which is expected to be useful in this classification problem. NB in both cases programs were evolved using memory appropriate to the problem.

7.4.2 One Indexed Memory

Most of the published work on using GP where use of memory is explicitly evolved follows [Teller, 1993; Teller, 1994a] which introduced "indexed memory", i.e. a single multiple celled directly addressable memory, to GP. For example [Raik and Browne, 1996] use indexed memory to show that on a reactive task, GP with explicit memory performs better than GP with implicit memory. Indexed memory, as it allows random access, provides little "structure" and could be problem independent, however in [Andre, 1994b; Andre, 1995b; Andre, 1995a] the indexed memory is made problem specific by treating it as two dimensional and sizing it so that it is isomorphic to a small problem "world". That is the memory is given a structure appropriate to the problem. A similar approach is taken in [Brave, 1996c] where memory is isomorphic to a full binary tree "world".

The simple indexed memory used in [Crepeau, 1995] is not obviously structured in a problem specific manner. The author suggests the success of GP at evolving a "Hello.World" program by manipulating (a subset of) Z80 machine code may in part be due to initialising memory with random 8 bit values. Thus it is "highly probable" [Crepeau, 1995, page 132] that the needed ascii values are initially in the indexed memory.

Another GP system which evolves machine code, based this time on the SUN RISC architecture, allows large amounts of directly addressable memory, however [Nordin and Banzhaf, 1995] does not describe experiments using it. [Nordin and Banzhaf, 1996] describes experiments using the system for sound compression where indexed memory and structured memory (a stack) were tried. In these experiments "programs took longer time to evolve and performed worse in fitness but had a softer sound with less overtones" than experiments without memory. However other changes were simultaneously made which may have made the task more difficult. Therefore it is difficult to draw any conclusions regarding the benefits or otherwise of data structures from this paper.

[Jannink, 1994] includes 16 memory cells in one experiment to evolve programs which generate "random" numbers. This is said to give "the best average validation score", i.e. better than when the programs were not given access to memory. Details

of how the evolved programs use memory are not given and no comparison with other memory sizes or structures is provided.

7.4.3 Case Base

[Spector and Alpern, 1995] presents a system which attempts to evolve music-making programs, specifically producing jazz improvisation responses to supplied "single-measure calls". "While we (Spector and Alpern) have not yet succeeded in inducing and recapitulating the deep structure of jazz melody" promising music generating programs have been evolved and the authors "believe that our framework holds promise for the eventual achievement of this goal."

While the authors refer to their memory system as "indexed memory" it is problem dependent. Consisting of 31 identical data structures, each of which is designed to hold a melody (expressed as 96 MIDI values). One data structure holds the input, another the output (i.e. the program's jazz "response") and the rest form a one dimensional array of 29 elements containing a case base of human written music. Only the output structure may be written to. Various problem dependent functions are provided for cutting and splicing segments of melodies but data values within the data structures cannot be directly manipulated.

7.4.4 Strongly Typed Genetic Programming

[Montana, 1995] presents two examples where GP is provided with local variables which it uses to solve problems (the two other examples don't allow explicit use of evolvable memory). The use of the strong typing framework means the variables must be typed. In both examples the variables are lists, which are either of the same type as the input or the same type as the output. That is with strongly typed GP data structures appropriate to the problem are readily chosen (STGP also prevents some kinds of abuse of the data structures).

7.4.5 Graph Data Structures

[Brave, 1995; Brave, 1996a] shows GP using a graph data structure which provides primitives to connect nodes and follow connections. Using this data structure the GP was able to solve a navigation problem which requires it to form a mental model of its world. This builds on [Andre, 1994b] but replaces a predetermined isomorphism between indexed memory and the problem "world" by a more complex data structure that is appropriate to the problem.

7.4.6 Linked List Data Structure

[Haynes and Wainwright, 1995] requires GP to evolve control programs for agents which have to survive in a simulated world containing mines. The agent's memory is a dynamically allocated linked list, with a new list element representing the current location being automatically allocated each time the agent enters a new location in the world. Read and write access is with respect to the current location, e.g. the current memory cell, the cell representing the location north of here, the cell north-east of that

and so on. The list keeps track of the agent's path allowing it to backtrack along its path. (Since its path lies in a minefield a safe option is always for the agent to retrace its steps). NB the memory is structured in an appropriate fashion for the problem.

7.4.7 Tree Structured Memory for Temporal Data Processing

[Iba et al., 1995] introduces "special 'memory terminals', which point at any nonterminal node within the tree." The value given by a memory terminal is the value at the indicated point in the tree on the previous time step. While this structure is applicable to a range of signal processing problem, once again memory has been constrained for the GP into a structure appropriate to the problem.

[Sharman et al., 1995; Esparcia-Alcazar and Sharman, 1997] similarly use memory terminals to hold values previously calculated at nodes within the program tree, however the mechanism for connecting terminals to inner nodes is different; explicit "psh" functions within the program tree save the value at that point in the tree by pushing it onto a stack. The stack is non-standard as "psh" writes to the current stack whereas "stk*n*" terminals provide a mechanism to read the stack created on the previous time step. The stack is also non-standard in that the "stk*n*" terminals non-destructively read data inside the stack (rather than from just the top of stack).

7.4.8 Object Oriented Programming

Some confirmation of the experimental results of Chapters 4 and 5 is provided by [Bruce, 1995; Bruce, 1996]. Although Bruce casts his work in an object oriented light rather than in terms of data structures there is much that is similar to this work. The details of the data objects in Bruce's experiments on evolving stack and queue data objects are similar to the stack and queue data structures in Chapters 4 and 5. They differ principally by the inclusion of a "`Full?`" object method and the lack of top or front operations. Bruce also considers the evolution of a "priority queue". While this has some similarities with the list data structure evolved in Chapter 6 it is significantly simpler with only five data methods rather than the ten simultaneously evolved in Chapter 6.

The details of the genetic programming system Bruce uses are similar to those used in Chapters 4, 5 and 6. For example one tree per data method (making a total of five trees per individual, see Section 3.6), separating pointers from main indexed memory (cf. Section 3.5), and use of tournament selection (cf. Section 3.2) with a steady state population (cf. Section 3.3). However a population size of 1,000 is used throughout rather than increasing to 10,000 for the more difficult problems.

Bruce conducts six experiments per object type in which he investigates the impact of, evolving the data methods one at a time rather than simultaneously, allowing the inspection of the internal operation of the programs and the impact of using strongly typed genetic programming. As might be expected, evolving one thing at a time, including a comparison of evolved program behaviour with a prescribed ideal implementation in the fitness function, and ensuring the evolved program is type correct, all make the GP's task easier. If all three are avoided (as in our experiments), which he labels experiments "3a", then his GP was unable to evolve the data structure in 20 runs. (Typically the experiments in Chapters 4 to 6 involve about 60 independent runs).

7.4.9 PADO

PADO [Teller and Veloso, 1995c; Teller and Veloso, 1995d; Teller and Veloso, 1996; Teller and Veloso, 1995b; Teller, 1995a; Teller, 1995b; Teller and Veloso, 1995a; Teller, 1996] is a GP based learning architecture for object recognition and has been shown to be able to correctly classify real world images and sounds far better than random guessing (albeit with less than 100% accuracy). PADO is a complex system with many non-standard GP features. For example the classification system is built from a hierarchy of individual programs which may use libraries of evolving code as well as ADFs similar to Koza's, repeated execution of programs within a fixed execution time, programs are represented by a directed graph of execution nodes rather than as trees and the genetic operators used to create new program are themselves evolved, cf. Section 2.4.1. The programs it generates are large and their operation is poorly understood.

PADO was deliberately designed not to use domain knowledge and so only the simplest memory structure (indexed memory) is used. It has been applied to complex ill behaved problems where there is no obvious data structure. GP could in principle build problem specific structures on top of indexed memory which the complexity and size of the evolved programs might conceal, however there is no evidence that this is happening. The better than random performance of PADO may be due to its many other features rather than its simple memory structure.

7.5 SUMMARY

The experiments described in Sections 7.1 to 7.3 (which were reported in part in [Langdon, 1996c]) have shown GP can solve two new problems. In Section 7.2 we showed GP can induce programs which correctly classify test sentences as to whether they are in a Dyck language or not and in Section 7.3 we showed GP evolving code which evaluates Reverse Polish Notation (RPN) expressions. In Section 7.1 we showed GP can solve the nested bracket problem 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 language and RPN) proved to be more difficult for GP when provided with indexed memory rather than when provided with a stack. Despite indexed memory being more powerful than stacks or simple scalars, none of the three problems has been solved using indexed memory.

Section 7.4 reviewed the current GP literature where problems have been solved using evolvable memory. It shows many cases where appropriate data structures have been used to solve problems. The principle counter example, where problem specific data structures have not been provided, is PADO, where better than random performance has been achieved on classification problems with no obvious structure.

It has often been argued, e.g. [Kinnear, Jr., 1994c, 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.

These experiments have not provided evidence that existing GP can scale up and tackle larger problems. If they had shown GP solving problems by evolving the

Table 7.9. 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. NB 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*

required data structures "on the fly" as it needed them this would have been powerful evidence. However this was not demonstrated. The failure of GP to solve the problems when provided with the more general (i.e. more powerful) directly addressable memory data structure shows that data structures should be chosen with care and it may not be sufficient to simply over provide, with more powerful structures than are needed.

Table 7.10. Actions Performed by Terminals and Functions (cont)

`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* **else return** *t2* //*x* and *y* don't match **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[$-l \ldots +l$], 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