Parsing and Translation of Expressions by Genetic Programming

David Jackson
Dept. of Computer Science
University of Liverpool
Liverpool L69 3BX, United Kingdom
Tel. +44 151 794 3678
d.jackson@csc.liv.ac.uk

ABSTRACT
We have investigated the potential for using genetic programming to evolve compiler parsing and translation routines for processing arithmetic and logical expressions as they are used in a typical programming language. Parsing and translation are important and complex real-world problems for which evolved solutions must make use of a range of programming constructs. The exercise also tests the ability of genetic programming to evolve extensive and appropriate use of abstract data types – namely, stacks. Experimentation suggests that the evolution of such code is achievable, provided that program function and terminal sets are judiciously chosen.

Categories and Subject Descriptors
D.1.2 [**Programming Techniques**]: Automatic Programming; D.3.4 [**Programming Languages**]: Processors – *compilers, code generation, parsing*; I.2.6 [**Artificial Intelligence**] Learning – *induction*.

General Terms
Algorithms, Experimentation.

Keywords
Genetic programming, application, software tools.

1. INTRODUCTION
The task of analysing a sentence, program or other text to determine its grammatical structure, and the job of converting such input to another equivalent form, are both important real-world problems that arise in many areas of computing. In a compiler, syntax analysis (parsing) is required to determine the grammatical correctness or otherwise of a computer program, and to establish the syntactical relationships that exist between the constructs which that program encodes. A compiler's translation procedures convert such source code either to an intermediate form or to machine code, usually for direct execution on the host computer.

Genetic programming (GP) is a technique that is used to generate programs automatically by evolutionary means, and has been applied successfully in many problem domains. Since parsers and translators are, after all, just programs (albeit complex ones), an interesting question is whether GP can be used for the induction of these programs too. A step towards answering this is the theme of this paper. Since, at this stage, the sophistication of a real compiler capable of handling a complete programming language seems a little ambitious, we will restrict ourselves to the more manageable realm of arithmetic and logical expressions.

Part of the motivation for this research is that comparatively little appears to have been done in applying evolutionary computing techniques to the compilation process, although other machine learning techniques have been employed [4]. Exceptions include the work of Cooper et al, who used genetic algorithms to solve compiler phase ordering problems [5], and the work of other researchers on the use of GAs to optimize pre-generated microcode [1,3]. Within the particular field of genetic programming lies the research of Stephenson, O'Reilly et al, who used GP to improve the performance of a compiler in respect of its ability to deal with the heuristic problems of predicated hyperblock formation and register allocation [14]. Although not involving programming language compilers, GP has also been applied to the related area of natural language parsing [2].

Another reason for pursuing this line of research is that the problem is a realistic (not toy) one, necessitating a variety of commonly used programming constructs. In the experiments that we shall describe, the three main control flow constructs of sequence, iteration and selection (alternation) will all be needed in combination, and to multiple depth levels. Evolved solutions will also need to make use of sequential input and output. Input will be processed an item at a time, and once an item has been read, the input pointer cannot be moved backwards to re-read it. Similarly, once an output item has been written, it cannot later be erased. Although it is certainly possible to devise other forms of the problem in which random access of the input and output strings is allowed, we wished to make our version as close as possible to real compiler activity, where such inefficiencies must be avoided.

GECCO'05, June 25-29, 2005, Washington, DC, USA.

Another key feature of the problem is that success can only come as a result of the correct use of abstract data types – in this case, stacks. The work of Langdon [11,12] on the application of GP to stacks and other data structures is well known, and he has even evolved programs to manipulate a stack for a postfix calculator (the relevance of which will become clearer in the next section) [13]. Our own work will further test the ability of GP to evolve code that can exploit and manipulate stacks in useful ways.

Following this introduction, the paper contains a section on parsing and another on translation. Within each of those, the problem domain is discussed before the experimental work and results are presented. The paper then ends with some concluding remarks and comments on further work.

2. PARSING

2.1 The Parsing Problem

The syntax of expressions can usually be defined using a simple grammar, and can be written down using BNF notation. The following grammar defines expressions that can be of either arithmetic (a-expr) or Boolean (b-expr) type:

```
<expr>   ::= <a-expr> | <b-expr>


<a-expr> ::= <a-term> | <a-expr> + <a-term>
             | <a-expr> - <a-term>
<a-term> ::= <a-fact> | <a-term> * <a-fact>
             | <a-term> / <a-fact>
<a-fact> ::= <a-prim> | <a-fact> ^ <a-prim>
<a-prim> ::= iden | number


<b-expr> ::= <b-term>
             | <b-expr> "|" <b-term>
<b-term> ::=  <b-fact>
             | <b-term> "&" <b-fact>
<b-fact> ::= iden | true | false
             | <a-expr> <relop> <a-expr>
<relop>  ::=  = | < | > | >= | <=
```

In determining how to evaluate an expression, there is an implicit and commonly-understood set of rules governing the ordering of operator application. In an expression such as

```
a*b+c/d-e
```

we know that we have to perform the multiply first, then the divide, then the add, and finally the subtract. Informally, these rules can be stated as perform multiply and divide before add and subtract, work from left right on equivalent-priority operators, and so on.

A grammar such as the one presented above formalises these rules. For example, an <a-expr> (containing add and subtract) is defined in terms of <a-term> constructs (containing multiply and divide), and so it follows that an <a-expr> can be recognised and evaluated only once its component <a-terms> have been recognised and evaluated. From the grammar given, we can derive a set of priorities for all of the operators that may exist in a valid expression:

```
^                 highest priority
*, /
+,-
<,>,=,…
& (and)
| (or)            lowest priority
```

Based on this prioritisation, a parser can convert an expression written in the standard form to a form which makes the order of operator application completely explicit. An example of a such an alternative form is postfix, or Reverse Polish Notation (RPN). In postfix, operators appear after the operands to which they apply. As an example, the expression

```
a*b+c/d-e
```

converts to

```
a b * c d / + e -
```

A way of thinking about the evaluation of a postfix expression is with the use of a stack: we work from left to right, pushing variables and numeric values onto the stack as they are encountered, and applying each operator to the top two items on the stack, leaving the result on the stack. So, for our example expression, we have:

```
stack a
stack b
multiply top 2 stack items and leave
   result on stack
stack c
stack d
perform divide
perform add
etc.
```

When the end of an expression is reached, the resulting value of that expression is left on the stack.

Since operator ordering is explicit in the postfix expression, parentheses are not needed. For example, an expression such as

```
a*(b+c)
```

converts to the postfix form

```
a b c + *
```

Once it has been converted to postfix by a parser, it is possible to pass an expression on to subsequent phases of compilation; for example, postfix can be used as the basis for code generation. The problem that concerns us for the moment is how to perform the conversion in the first place.

There are two main approaches to syntax analysis: top-down and bottom-up. In the top-down approach, the parser assumes that its input is a valid sentence. It begins by searching for phrases in the input that can be combined to make a valid sentence. For each phrase, it looks for sub-phrases, and so on, until the search reduces to looking for primitive items in the input text.

By contrast, a bottom-up parser tries to find groups of primitive items that correspond to phrases in the grammar. It replaces these groups by the corresponding phrase denotations, then tries to group the phrases into higher-level phrases, and so on, until it is able to make the final replacement that converts the whole lot to a sentence in the language.

In the experiments which follow, we attempt to evolve parsers which are bottom-up. The reason for this is partly to avoid the high degree of mutual recursion usually present in top-down parsers, but also because in many compilers the bottom-up approach is used for expressions, while the top-down approach is reserved for statements and declarations.

2.2 Parsing Experiments

Table 1 presents the parameters used for our initial attempt at solving the parsing problem by genetic programming. The fitness of an individual in the population is assessed by applying that individual's program code to a set of expressions, and comparing the output with the expected postfix version.

Table 1. Initial problem tableau for parsing experiments

Objective	Convert arithmetic expressions to postfix
Terminal set	more-items, stack-not-empty, get-item, item-val, stack-top, pop
Function set	operand, operator, output, push, priority, le, while, progn2, if-then-else
Initial population	Ramped half-and-half, no duplicates
Evolutionary process	Steady-state; 5-candidate tournament selection
Fitness cases	30 arithmetic expressions
Fitness	Number of incorrect postfix expressions
Restrictions	Programs timed-out after 2000 instructions
Success predicate	Zero fitness (all exprs parsed correctly)
Parameters	M=2000; G=51; prob. crossover=0.9; no mutation

Table 2. Expression test data and postfix equivalents

Expr	Postfix	Expr	Postfix	Expr	Postfix
a	a	a+b-c*d	ab+cd*-	a^b*c+d	ab^c*d+
b	b	a+b*c-d	abc*+d-	a^b+c*d	ab^cd*+
a+b	ab+	a*b+c-d	ab*c+d-	a=b+c*d^e	abcde^*+=
a*b	ab*	a*b/c+d	ab*c/d+	a+b=c*d^e	ab+cde^*=
a+b-c	ab+c-	a+b*c/d	abc*d/+	a+b*c=d^e	abc*+de^=
a-b+c	ab-c+	a*b+c/d	ab*cd/+	a+b*c^d=e	abcd^*+e=
a+b*c	abc*+	a+b*c^d	abcd^*+	a+b<c|d>e	ab+c<de>|
a*b+c	ab*c+	a+b^c*d	abc^d*+	a<b&c>d|e	ab<cd>&e|
a/b^c	abc^/	a*b+c^d	ab*cd^+	a=b|c>d&e	ab=cd>e&|
a^b/c	ab^c/	a*b^c+d	abc^*d+	a+b=c-d|e	ab+cd-=e|

The number of test expressions is set at 30, and the fitness value corresponds to the number of tests which are failed by an individual. Thus, zero fitness implies that all expressions are correctly parsed. The full set of test expressions and their postfix equivalents is given in Table 2.

Since the evolved code may contain loops, a program is terminated if it exceeds a maximum number of instruction executions (set at 2000). A single run of the GP system continues until either the maximum number of generations is reached, or a solution (a zero-fitness program) is evolved.

More details of the initial function and terminal sets used for this problem are given in Table 3. Some of these correspond to operations that are typically available for stack-based algorithms: i.e. push, pop, stack-top and stack-not-empty. The meaning of these and most of the others should be fairly obvious. In the case of the priority function, if a valid arithmetic or logical operator is supplied as its argument, the function returns a number representing the position of that operator in the hierarchy of priorities described earlier. Hence, the or operator has priority 1, the and operator has priority 2, and so on. The progn2 function is a connective construct that merely causes each of its two arguments to be evaluated in turn, and is therefore a way of enforcing sequencing in a functional paradigm.

Table 3. Initial function and terminal sets for parser

Node name	Arity	Operation
more-items	0	Return 1 if more items in input, 0 otherwise
stack-not-empty	0	Return 1 if items on stack, 0 otherwise
get-item	0	Fetch next item from input, return its value (zero if empty)
item-val	0	Return value of current item
stack-top	0	Return value of item on top of stack, zero if stack empty
pop	0	Pop item from stack, return its value (zero if stack empty)
operand	1	Return 1 if arg is operand, 0 otherwise
operator	1	Return 1 if arg is operator, 0 otherwise
output	1	Output arg if it is valid operator or operand
push	1	Push arg onto stack
priority	1	Return numeric priority of arg (zero if not valid operator)
le	2	Return 1 if arg1 <= arg2, zero otherwise
while	2	Execute arg2 while arg1 not 0
progn2	2	Execute arg1, then arg2
if-then-else	3	If arg1 not zero, execute arg2, else execute arg3

The results of this first attempt were disappointing. With the node sets just described, it proved impossible to evolve a solution in any run. Figure 1 shows a typical graph of the changes in best and average fitness of a population. The evolutionary process begins well, but soon reaches a point beyond which it cannot progress. Indeed, no run achieved a fitness value lower than 16; that is, the best programs were able to parse 14 of the 30 expressions. Alterations to the population size and the number of generations made no difference to this outcome.

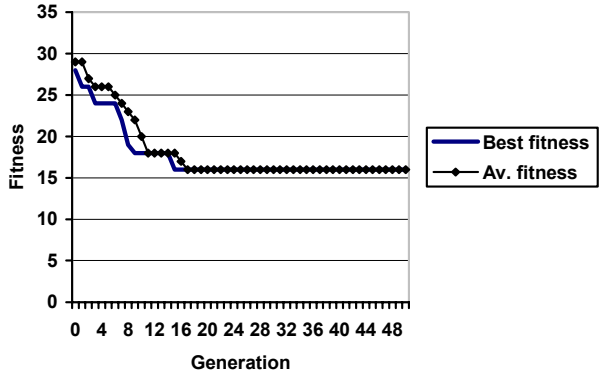

Figure 1. Fitness graph for initial parsing experiment

In an effort to address this problem, two other mechanisms were tried. The first of these involved making the fitness determination more fine-grained. Instead of the all-or-nothing approach for a given expression, the idea is to award a score based on how 'close' a program's output comes to the expected postfix expression. For each item in the expected postfix string, the score is augmented (i.e. the penalty increases) by an amount equal to the distance from the item's position in the program's output string. A fixed penalty is applied if the item is not present at all. Hence, zero (best) fitness is achieved if and only if all items are present in the output string and in their correct positions. The approach is similar in concept to the inversion counting method used by Kinnear in the evolution of sorting programs [7,8].

Unfortunately, this change to the fitness function brought about no improvement, and in fact most runs were made much worse.

The second mechanism attempted was that of complementary phenotype selection [6]. The idea here is to select parents for mating according to how well they offer a combined coverage of successful test cases. The way this proceeds is that the first parent is selected according to its fitness in the customary way; the second parent is chosen not by fitness, but by its ability to maximize the number of distinct test cases passed by both parents. Hence, if parent 1 passes test cases {1, 3, 5, 7}, candidate parent 2A passes cases {1, 5, 7, 8}, and candidate parent 2B passes tests {1, 10, 14}, then 2B should be selected as the second parent despite having a worse fitness score than 2A, since the combined test coverage ({1,3,5,7,10,14}) is better than that achieved with 2A ({1,3,5,7,8}).

Again, however, no improvement was obtained using this method. As before, no individual achieved more than 14 hits. Analysis showed that combined coverage during mate selection never achieved more than 20 out of the 30 test cases. Particularly problematic, it seems, are expressions containing several operators in ascending order of priority (e.g. a+b*c^d), as these require extensive use of the stack. Since these expressions are hardly ever parsed correctly, the chances that complementary phenotype selection will lead to completely correct programs are correspondingly slim.

Table 4. Reduced node sets for parsing experiment

Node name	Arity	Operation
item-val	0	Return value of current item
stack-top	0	Return value of item on top of stack (zero if empty)
operand	0	Return 1 if input item is operand, 0 otherwise
lteq	0	Return 1 if priority of item <= priority of stack-top, else zero
output-item	0	If not at end of input, output current item & advance input ptr
output-stack	0	Output top of stack (if not empty)
push-item	0	If not at end of input, push item onto stack & advance input ptr
while	2	Execute arg2 while arg1 not zero
progn2	2	Execute arg1, then arg2
if-then-else	3	If arg1 not zero, execute arg2, else execute arg3

Following extensive experimentation, it was decided to reduce the number of program node types available. Table 4 shows the function and terminal sets that were eventually settled upon. It can be seen that, while the terminal set has been increased by one member, the number of functions has been significantly decreased. Moreover, many of the terminal set members are more powerful in their functionality, performing operations that previously would have required several nodes. Figure 2 shows the evolutionary progress of best and average fitnesses during a single run using this new node set. In this case, an 844-node solution was evolved at generation 20.

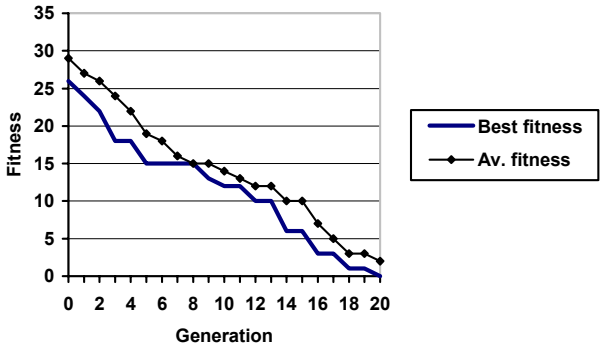

Figure 2. Fitness graph for reduced node sets

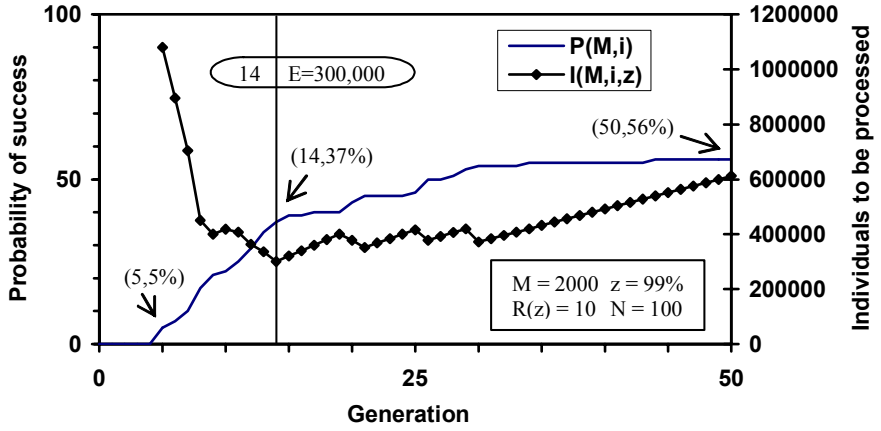

Figure 3. Performance graph for parsing experiment

Following Koza [9,10], we can plot a graph of the probability of success of finding a solution by any generation i, using a population size M. This is shown as the line labelled P(M, i) in Figure 3. For example, at generation 5 the probability of success is only 5%, while by generation 50 it has risen to 56%. Data for the graph is obtained from N=100 runs of the GP system. In the same graph, we can also plot the number of individuals required to be processed in order to achieve a probability of z=99% that a solution will be found by a given generation. This is given by the line labelled I(M, i, z). The minimum value of this line, or the 'computational effort,' is E=300,000, meaning that 300,000 individuals must be processed to achieve a probability of at least 99% that a solution will be obtained. As can be seen from the graph, this value of the computational effort corresponds to R(z) = 10 runs to generation 14.

Two important points about the 'solutions' obtained in these runs must be made. Firstly, the programs were all very large. Secondly, and more importantly, many of them did not generalise. In other words, although they were capable of parsing the expressions present in the input data set, they were not able to parse more complex expressions that they had not previously encountered.

Table 5. Post-evolutionary test data

Expr	Postfix
a*b/c=d|e>f&g	ab*c/d=ef>g&|
a=b&c<d|e>f-1	ab=cd<&ef1->|
a+b=c*d-1|x<y	ab+cd*1-=xy<|
a=b*c|d=x/y&e<f	abc*=dxy/=ef<&|
h*6+m=3+6*2|t	h6*m+362*+=t|

To address this, an additional post-evolutionary test data set was introduced; it can be seen in Table 5. During a run, an individual is adjudged to be correct if and only if it correctly parses the expressions in Table 2 and those in Table 5. However, the latter expressions are evaluated only *after* an individual's fitness has been evaluated. Success or failure on this new data set does not affect the fitness score, and so does not alter the course of evolution except to determine when a solution has been found. It should also be pointed out that, although the second test set is small, its members are more sophisticated, and it was found to be sufficient to distinguish between generalising and non-generalising programs.

As before, the computational effort can be calculated, and was found to have a value of 640,000 individuals, corresponding to 20 runs to generation 15. This is more than twice the computational effort required in the previous experiment, where generality was not a requirement. The following is one example solution, consisting of 19 nodes:

```
WHILE (OPERAND
   IF (PROGN2
         (IF (OUTPUT_ITEM OUTPUT_ITEM LTEQ)
          WHILE (LTEQ OUTPUT_STACK))
   //then
        PROGN2(LTEQ PROGN2(ITEM_VAL
              PUSH_ITEM))
   //else
        PROGN2(ITEM_VAL PUSH_ITEM)))
```

It can be simplified to the following 9-node program:

```
WHILE (OPERAND
   PROGN2
      (PROGN2
         (OUTPUT_ITEM
          WHILE (LTEQ OUTPUT_STACK))
       PUSH_ITEM))
```

A point that should be made about many of the solutions obtained in this experiment is that they contained infinite loops. They

terminated only when they were forced to by the GP system because the maximum instruction count was exceeded. This is obviously an undesirable property of any real-world parsing program, and we therefore performed a further experiment in which finite termination was a necessary condition of program correctness. As might be expected, this pushed the computational effort up still further, to a value of 1,776,000 individuals, representing 24 runs to generation 36.

3. TRANSLATION

3.1 The Translation Problem

The postfix notation that our evolved programs produced in the previous experiments is well-suited to the subsequent generation of target code for stack-based machines. For example, the expression

```
a*b+c/d-e
```

which converts to the postfix string

```
a b * c d /  + e -
```

is then trivially translated to machine code such as the following:

```
PUSH a
PUSH b
MULT
PUSH c
PUSH d
DIV
ADD
PUSH e
SUB
```

However, postfix notation is not a very suitable representation for translation on more general register-based machines. One of the difficulties with it is that it corresponds to a fixed traversal of the abstract tree form of the original expression; as such, it leaves little room for optimization and code generation decisions.

An alternative, more flexible form of representation is provided in the form of triples. When used to encode expressions, each triple comprises a single operator and two operands. The operands may be references to other triples. Consider the following expression:

```
a*b/c = d | e>f & g
```

This may be translated to the following sequence of triples:

```
#1:    *      a      b
#2:    /      #1     c
#3:    =      #2     d
#4:    >      e      f
#5:    &      #4     g
#6:    |      #3     #5
```

This may be thought of as a form of pseudo-assembly code, in which the first instruction is to multiply a and b, leaving the result in register (or temporary variable) 1; the second instruction says to divide the contents of register 1 by variable c, leaving the result in register 2; and so on. On many machines, conversion of triples to the target machine code is relatively straightforward. Triples are sometimes produced as a form of intermediate code, created internally by a compiler before it is passed to a final code generation phase.

Triples can also be viewed as a linear form of the associated expression tree. Each numbered triple corresponds to a labelled node of the tree, with the final result triple forming the root node. The tree for our example expression is shown in Figure 4. The value of such tree structures is that the compiler can make dynamic decisions as to how best to traverse the tree when generating code, and can also perform certain tree-based optimisation procedures.

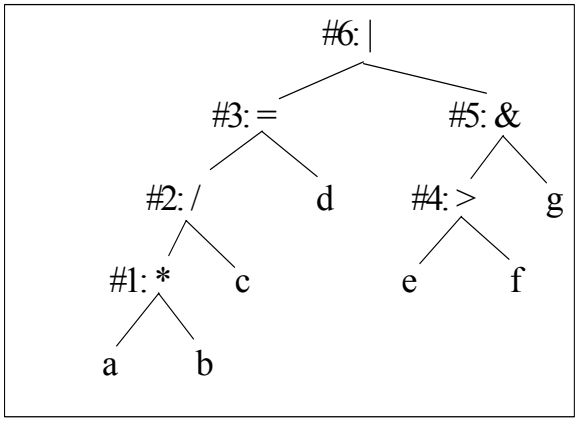

Figure 4. Expression tree for triples from a*b/c = d | e>f & g

It is known that translating an expression directly to triples can be done with the aid of two stacks: one for operators and one for operands. In this next experiment, we wished to investigate whether a GP system could evolve a translator that was capable of this more sophisticated use of stack data structures.

3.2 Translation Experiments

For this problem, the tableau of GP parameters remains much as it did for the parsing experiment. The only real change is to the terminal set, which now becomes:

```
{item-val, stack-top, operand, lteq, output-triple, push-rator, push-rand}
```

The first four members of this set are as before. The single push operation now becomes two: one to push the current input item onto the operator stack, and one to push it onto the operand stack. There is also now an output-triple operator. This forms a triple from the top item of the operator stack and the top two items of the operand stack, assigns the next integer in sequence to the triple, outputs the triple, and leaves the triple number on the operand stack.

The test data also remains as before, except for the two simplest cases consisting of single variables, since these cannot form triples. In performing this experiment we proceeded directly to the evolution of translators that were capable of generalising, i.e. able to solve both the initial training set and the post-evolutionary test set. Some of the test cases drawn from both sets, together with the corresponding triples, are shown in Table 6.

Table 6. Sample test data and triples for translator experiment

Expr	Triples	Expr	Triples
a+b-c	#1: + a b #2: - 1 c	a*b/c=d|e>f&g	#1: * a b #2: / #1 c #3: = #2 d #4: > e f #5: & #4 g #6: | #3 #5
a+b-c*d	#1: + a b #2: * c d #3: - #1 #2	h*6+m=3+6*2|t	#1: * h 6 #2: + #1 m #3: * 6 2 #4: + 3 #3 #5: = #2 #4 #6: | #5 t
a=b+c*d^ e	#1: ^ d e #2: * c #1 #3: + b #2 #4: = a #3		

Over 100 runs of the system, the calculated computational effort was 714,000 individuals, equating to 17 runs to generation 20. This figure is 11.5% higher than the computational effort required for the generalised parsers discussed earlier. An example solution is the following, consisting of 38 nodes:

```
IF ( WHILE ( PUSH_RAND OPERAND ) PROGN2 (
PUSH_RAND OPERAND ) IF ( WHILE ( WHILE (
PUSH_RATOR LTEQ ) PUSH_RAND ) WHILE ( IF (
IF ( PUSH_RAND OPERAND LTEQ ) OPERAND LTEQ )
IF ( OUTPUT_TRIPLE LTEQ STACK_TOP ) ) WHILE
( STACK_TOP IF ( IF ( PUSH_RAND PUSH_RATOR
WHILE ( LTEQ OUTPUT_TRIPLE ) ) OPERAND WHILE
( PUSH_RATOR LTEQ) ) ) ) )
```

This can be simplified to the following:

```
PROGN2 ( PUSH_RAND PROGN2 ( PUSH_RATOR WHILE
( STACK_TOP PROGN2 ( PUSH_RAND PROGN2 (
WHILE ( LTEQ OUTPUT_TRIPLE ) PUSH_RATOR ) )
) ) )
```

As before, many of the solutions are non-terminating. If we insist on termination, the computational effort rises to 2,668,000 individuals, representing 29 runs to generation 45. This is over 50% higher than the effort required for a terminating generalised parser.

4. CONCLUSIONS

In this paper we have described experiments aimed at evolving programs to handle the parsing and translation of arithmetic and logical expressions. In the case of parsing, we have evolved programs to convert expressions to postfix form. Initially, it was hoped that a general set of relatively low-level primitives might be used to induce the parsers, but this proved not to be the case, despite extensive experimentation with varying parameters. That said, there remain a number of approaches that were not attempted: for example, the use of mutation and alternative genetic operators, and the use of encapsulation methods such as automatically defined functions (ADFs) [10]. At present, the problem as stated with our initial function and terminal sets remains an open one, and the author would be interested to learn of any success achieved by researchers applying their own pet methods to it.

Employing a slimmed-down set of more powerful primitives did lead to solutions, although it was found that success on the training set did not always imply generality in the evolved parsers. Use of an additional post-evolution test data set was able to rectify this, at the expense of additional computational effort. Similarly, an insistence that solutions should terminate in a finite number of steps was also achievable at a computational cost.

As mentioned in the introduction, the experiments carried out here act as a good test of the ability of evolving programs to make appropriate use of stack data structures. This is especially true in the case of translation of expressions to triples, where two stacks are required. Success was achieved, but with more computational effort than that required for the postfix-producing parsers – considerably so when finite termination was made a pre-requisite for correctness.

In summary, then, it can be concluded that the experiments showed that there is some potential for the use of genetic programming in the generation of parsers and translators for expressions, albeit with judicious choice of the function and terminal sets made available to the GP system. Expression analysis and processing is, of course, just a small part of what a complete compiler does, and we are a long way from evolving such a complex piece of software. However, one of the advantages of using compilers as a subject of study in this way is that they are highly modular systems, containing many interacting algorithms of varying sophistication. For the future, we will continue to investigate the applicability of genetic programming to some of these other aspects of the compilation process.

5. REFERENCES

[1] Ahmad, I., Dhodhi, M. K. and Saleh, K. A. An Evolutionary Technique for Local Microcode Compaction. *Microprocessors and Microsystems, 19*, 8 (Oct. 1995) 467-474.

[2] Araujo, L. Genetic Programming for Natural Language Parsing. In *Proc. EuroGP 2004, Lecture Notes in Computer Science 3003*, Springer-Verlag, Berlin Heidelberg, 2004, 230-239.

[3] Beaty, S., Whitley, D. and Johnson, G. Motivation and Framework for Using Genetic Algorithms for Microcode Compaction. In *Proc. 23rd Annual Workshop on Microprogramming and Microarchitecture (MICRO-23)*, Orlando, FL, USA, 1990, 117-124.

[4] Calder, N., Grunwald, D., Jones, M., Lindsay, D., Martin, J., Mozer, M. and Zorn, B. Evidence-Based Static Branch Prediction Using Machine Learning. *ACM Trans. Programming Languages and Systems (ToPLaS) 19*, 1 (1997), 188-222.

[5] Cooper, K., Scheilke, P. and Subramanian, D. Optimizing for Reduced Code Space Using Genetic Algorithms. In *Languages, Compilers, Tools for Embedded Systems*, 1999, 1-9.

[6] Dolin, B., Arenas, M. G. and Merelo, J. J. Opposites Attract: Complementary Phenotype Selection for Crossover in Genetic Programming. In *Proc. PPSN VII, Lecture Notes in Computer Science 2439,* Merelo Guervos, J. J. et al (eds), 2002, 142-152.

[7] Kinnear, Jr., K. E. Generality and difficulty in genetic programming: Evolving a sort. In *Proc. Fifth International Conf. on Genetic Algorithms* (University of Illinois), S. Forrest (ed), Morgan Kaufmann, San Mateo, CA, 1993, 287-294.

[8] Kinnear, Jr., K. E. Evolving a sort: Lessons in genetic programming. In *Proc. 1993 International Conf. on Neural Networks* (San Francisco, USA), IEEE Press, 1993, 881-888.

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

[10] Koza, J. R. *Genetic Programming II: Automatic Discovery of Reusable Programs*, MIT Press, Cambridge, MA, 1994.

[11] Langdon, W. B. Data Structures and Genetic Programming. In *Advances in Genetic Programming, vol. 2,* Angeline, P.J. and Kinnear, K.E. (eds), MIT Press, Cambridge MA, 1996, 395-414.

[12] Langdon, W. B. *Genetic Programming and Data Structures*. Kluwer, 1998.

[13] Langdon, W. B. Using Data Structures within Genetic Programming. In *Genetic Programming 1996: Proceedings of the First Annual Conference,* Koza, J. R., Goldberg, D. E., Fogel, D. B. and Riolo, R. L. (eds), MIT Press, Cambridge MA, 1996, 141-149.

[14] Stephenson, M., O'Reilly, U-M., Martin, M. C. and Amarasinghe, S. Genetic Programming Applied to Compiler Heuristic Optimization. In *Proc. EuroGP 2003, Lecture Notes in Computer Science 2610,* Springer-Verlag, Berlin Heidelberg, 2003, 238-253.