# Quick Intro to simple-gp.c

*W. B. Langdon*

Dept of Computer Science,
University College London.

25 April 1994

## ABSTRACT

Wouldn't it be great if computers could actually write the programs? This is the promise of genetic programming. This document gives an introduction to the program simple-gp.c which demonstrates the principles of genetic programming in the C language. This program was designed to show the ideas behind GP, ie to be tinkered with rather than to be a definitive implementation.

The appendix contains a glossary of evolutionary computing terms.

# 1 Quick Introduction to Genetic Programming

This document does not give a complete intoduction to this topic, please see Koza's [Koz92] work.

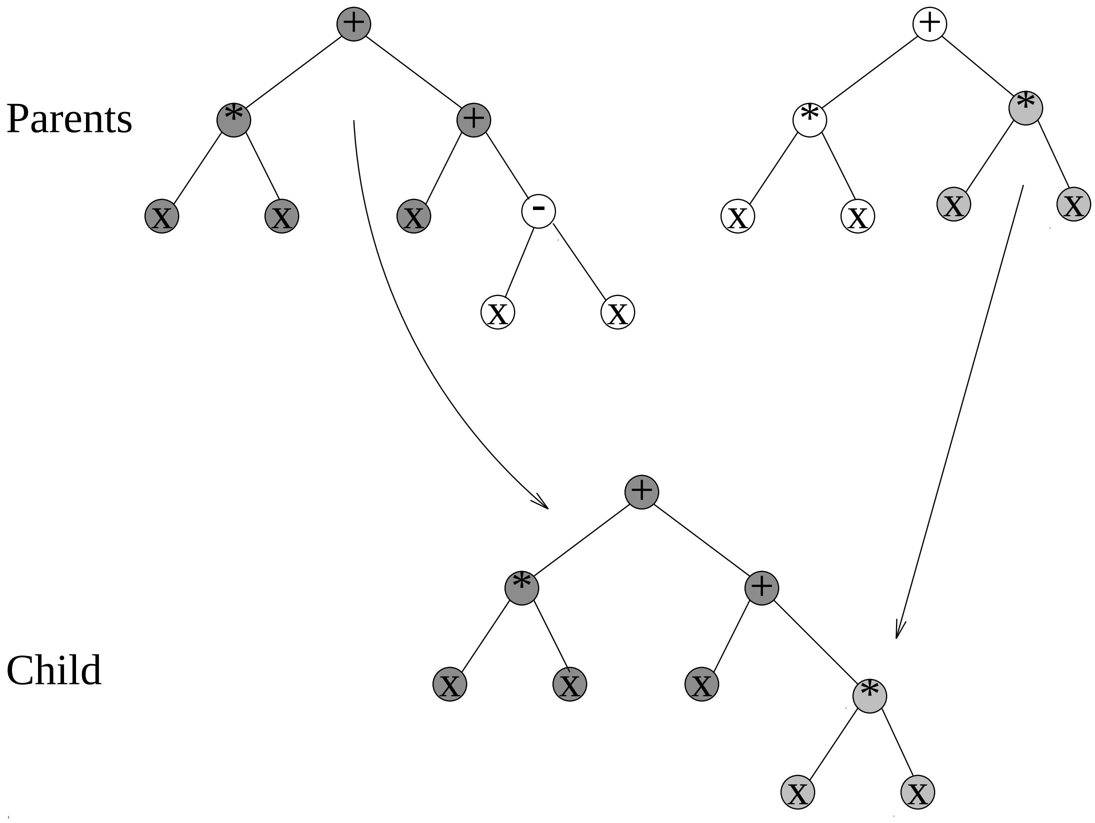
Koza notes there are five[1] preliminary steps before starting a genetic program; choosing the terminals, the functions, the fitness function, control parameters and the termination criterion.

In Koza's terminology, the terminals and the functions are the components of the programs, the functions being junctions in the tree and the terminals the end leaves. For example, functions like + which take two arguments cause a two way fork in the tree, whereas a terminal might be a constant such as 2 or an input such as $x$. Terminals can only appear in the program as arguments to a function.

The choice of components of the program (ie terminals and functions) and the fitness function largely determine the space in which the genetic program will search and consequently how difficult that search is and ultimately how successful it will be.

The control parameters include issues such as the size of the population. The termination criterion is simply a rule for stopping, typically this will include finding a program which solves the problem and a timeout, in case such a solution is not found.

[1]In his later work [Koz94] Koza adds a sixth step; determining the architecture in terms of the programs automatically define functions (ADFs). However he also shows that it is possible (but not necessarily easy) for a genetic program to succeed even with arbitrary choices for his six steps provided it is allowed additional flexibility. We shall not dwell on ADFs here.




Genetic Programming Crossover:
Figure 1: $x^2 + x + x - x$ crossed with $2x^2$ to produce $2x^2 + x$.

## 1.1 A Worked Example of Genetic Programming

In this section we will develop in C a simple example genetic program, which will generate executable C code given only a set of test values which the program is to match. The complete code is available via anonymous ftp from ftp.cs.ucl.ac.uk, directory genetic.

The example is presented in C because of the widespread use of this language but almost any programming language could be used. The programs that are evolved are represented internally in a special purpose interpreted language, this avoids the overhead associated with compiling and linking each program. This is not essential to the method and the interpreter will not be described here. Nb the final program is output as C.

The example problem is to generate C code which will calculate simple polynomials such as $x + x^2$. We then demonstrate this genetic program finding a C program which calculates $x^4 - x^3 + x^2 - x$ [2].

Our method is inspired by rather than identical to Koza's (for example we shall use C rather than Lisp); we shall follow his five steps (see previous section).

### 1.1.1 Choice of the Terminal Set

The only terminal (or leaf) is the input to our program, $x$.

### 1.1.2 Choice of the Function Set

We chose the four floating point arithmetic operators $+$, $-$, $\div$ and $\times$.

Note that we chose the terminal and function sets so that any function can operate upon the result of any function or terminal (Koza calls this property closure. In this case they are all `float`, additionally we define divide by zero to be unity. Also note, all the operators have two operands, so the programs evolved have the form of a binary tree.

### 1.1.3 Choice of the Fitness Function

In our example we shall start by reading in a number of test points which lie upon the curve we are attempting to match. These are stored in arrays `x[]`, `y[]` by function `read_inputs()`. Table 1 lists the points chosen (Figure 2 shows them graphically).

For each program generated, we will evaluate its fitness by calling it with each of the $x$ values and comparing its answer with the desired $y$. The absolute difference (`fabs()`) gives us a value of how bad the program is. We invert this Figure to give a fitness which says how good it is (adding 1.0 before inverting to avoid the possibility of divide by zero). The mean across all the fitness cases is produced by dividing the sum of the differences by the number of points.

If the program is within 1% (or 0.01) we call this a hit, indicating that this is close enough. The hits are not used to select programs but to decide when to stop, which program is the best and for display purposes.

```
test_fitness (int child )
{
  for ( i=0; i < num_x; i++ )
```

[2] Similar polynomials may be produced simply by changing the test points.

| $x$ | $y$ |
|---|---|
| -1.0 | 4.0000 |
| -.5 | .9375 |
| -.2 | .2496 |
| -.1 | .1111 |
| .0 | .0000 |
| .1 | -.0909 |
| .2 | -.1664 |
| .5 | -.3125 |
| 1.0 | .0000 |

Table 1: Test Points for $y = x^4 - x^3 + x^2 - x$

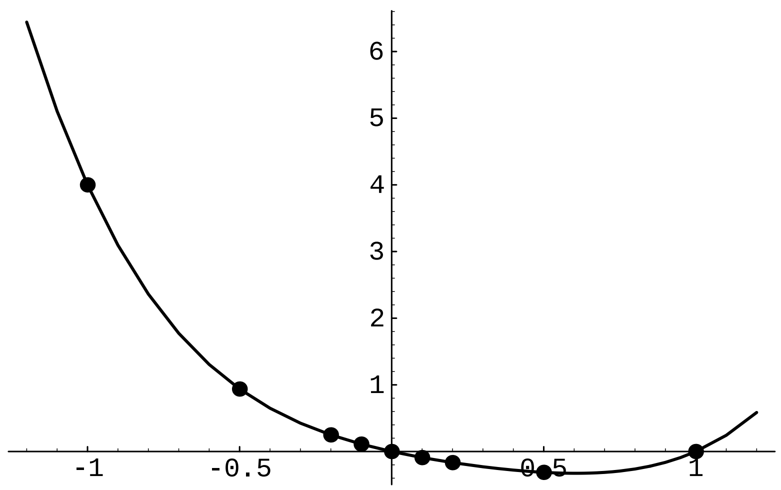



Figure 2: $y = x^4 - x^3 + x^2 - x$

```
{  error = fabs ( y[i] - prog_value( new_population[child].string, x[i] ) );
   if (error <= THRESHOLD || error <= THRESHOLD * fabs(y[i]))
       test_hits++;
   total_error += error;
}/*end for each test point*/

update_hits_etc(num_x/(total_error+num_x), test_hits, child);
```

Function `update_hits_etc()` stores the program's fitness and number of hits in `new_population[child]` and keeps track of the best program found in the generation.

### 1.1.4 Choice of the Control Parameters

We chose, a population size (`POPULATION_SIZE`) of 500; to create 60% of each new generation by cross over (`PCROSS`), of the remainder 99% are direct copies from the previous generation and 1% are mutated copies (`PMUT`). We limit the size of our programs. The limit is chosen to to be high enough not to interfere unduly with the evolutionary process but to restrain its consumption of computer resources. The smallest representation of our target function is $x(x(x^2-x)+x)-x$ which requires 13 nodes in the tree whereas $x^4-x^3+x^2-x$ requires 19. We have chosen the rather arbitrary limit of 25 (`MAX_PROG_SIZE`) nodes (ie no more than 12 operators and 13 operands).

### 1.1.5 Choice of the Termination Criterion

We stop when we have a found a program that correctly matches all the input points (ie they are all hit) or when we reach 50 (`MAX_GEN`) generations.

```
int main()
{
  read_inputs();
  generation = 0;
  initialise_population();

  while ((++generation <= MAX_GEN) && (max_hits < required_hits))
  {
    for (i = 0; i < POPULATION_SIZE; i++)
    {  mum = select();
       if (random() < PCROSS)
         { dad = select();
           crossover (mum, dad, i);
           test_fitness(i);
         }
       else if (random() < PMUT)
         { mutate(mum,i);
           test_fitness(i);
         }
       else
         { copy (mum,i);
```

```
        };
    }/*end for - create new_population */

    replace_old_population();
  } /*end while */

  output_program( population[best_prog].string );
```

Function `initialise_population()` creates 500 random programs within `new_population` by calling `mutate()` and then uses `test_fitness()` to calculates the fitness of each. Finally they are copied to `population[]` by `replace_old_population()`.

### 1.1.6 Implementation

We represent the programs we are generating with strings, each character represents one operator or one operand. Each operator is followed by its operands, which may be themselves be operators, ie the string represents a tree and characters can not be arbitrarily chosen within it (Figure 3, page 10, shows an example program tree and its string representation).

Function `mutate()` is called both to create random initial programs (in which case `mum` is a zero length string) and to replace a subtree (chosen at random) within an existing program with a new subtree created at random. Care is taken that the new program does not exceed the available space, if it does the change is discarded and another is made.

Function `splice()` takes three strings (specified by their start and length) and stores the result of concatenating them in the buffer (denoted by its first two arguments). If the result fits in the buffer it returns zero, otherwise it returns a non-zero value.

```
mutate(int mum, int child)
do {
  mum_end1   = random_point ( population[mum].string );
  mum_start2 = matching_point ( population[mum].string, mum_end1 );
  mum_end2   = strlen ( population[mum].string );

  mut_length = random_tree ( buffer, MAX_PROG_SIZE );

} while (splice (
        new_population[child].string, MAX_PROG_SIZE+1,
        population[mum].string, mum_end1,
        buffer, mut_length,
        &population[mum].string[mum_start2], mum_end2 - mum_start2 ) != 0 );
```

`random_tree()` builds new programs or subtrees starting from the root choosing each new node at random. To ensure the tree has the correct syntax a count is kept of the number of operators which do not yet have operands. As this count gets higher the chance of choosing an operand increases, so preventing the tree getting too big. The algorithm chosen tends to produce asymmetric trees with various length branches. Whilst not identical to the algorithm used by Koza, it is in keeping with later work which used ragged trees (his *half-and-half* method).

```
int random_tree (char buff[], int size)
```

```
{
int dangling_limbs = 1;

  for ( i = 0; (dangling_limbs > 0) && (i <= size) ; i++ )
  {  /*chose function or terminal at random*/
     if( random() > (float)(dangling_limbs*dangling_limbs+1)/(float)(size-i) )
       { buff[i] = random_function();
         dangling_limbs++;          /*all operators have two limbs*/
       }
     else
       { buff[i] = random_terminal();
         dangling_limbs--;
       }
  }
  return (i);
```

To create succeeding generations, parents are chosen in proportion to their fitness from `population[]` by `select()`.

```
int select()
/* return index of parent in old population*/
{
float cumlative_fitness = 0.0;

  r = total_fitness * random();

  for (i = 0; i < POPULATION_SIZE; i++ )
    {   cumlative_fitness += population[i].fitness;
        if ( r <= cumlative_fitness ) return (i);
    }

  return ( POPULATION_SIZE - 1 );
```

New programs are created either by copying old ones, mutating old ones or by crossover between two old ones. The latter is implemented by `crossover()`, which chooses a subtree in one existing program at random and replaces it with another subtree chosen (again at random) from another existing program. As with `mutate()` a check is made that the new program is not too big. If it is the change is discarded and we try again. To keep things simple we only generate one new program; Koza inserts the discarded subtree into the other existing program, so creating two programs in one crossover operation.

```
crossover ( int mum, int dad, int child )
/*choose a random point in program from old population and another
  also from old population to produce a child which is stored in the new
  population */
{
do {
  mum_end1   = random_point ( population[mum].string );
  mum_start2 = matching_point ( population[mum].string, mum_end1 );
```

```
  mum_end2    = strlen ( population[mum].string );

  dad_start = random_point ( population[dad].string );
  dad_end   = matching_point ( population[dad].string, dad_start );

} while (splice (
        new_population[child].string, MAX_PROG_SIZE+1,
        population[mum].string, mum_end1,
        &population[dad].string[dad_start], dad_end - dad_start,
        &population[mum].string[mum_start2], mum_end2 - mum_start2 ) != 0 );
```

Most of the rest of the C code is concerned with displaying the progress of the run or the details of interpreting the internal program representation and will not be described here.

### 1.1.7 Results

3

On one run the following 100% correct code was generated:

```
float div ( float top, float bot )
{if (bot == 0.0)
   return (1.0);
 else
   return ( top/bot );
}

float gp( float x )
{
  return (((x)*((x)-((((x)-(((x)-(x))*(x)))-((x)*(x)))*(x))))-(x));
}
```

By replacing x-x by zero, removing excessive brackets etc the return statement may be reduced to:

```
  return ( x*(x + (x*x -x)*x ) - x );
```

ie C code to calculate $x^2 + x^4 - x^3 - x$.

This simple problem serves merely as an illustration. For comparison programs were created entirely at random, ie without crossover etc, a successful program was generated after 17925 attempts. Genetic programming took between 969 and 7124 on various trials.

In one run the 100% correct program shown in Figure 5 (page 10), was produced by crossover between the two programs shown in Figures 3 (page 10) and 4 (page 10) both of which where of above average fitness. The crossover points chosen (4 and 20) and the following subtrees are shown shaded. It can be seen that the net effect of crossover at these two points is to replace the first terminal ($x$) with a subtree which calculates $x^2$ and this yields a program which is exactly equivalent to $x^4 - x^3 + x^2 - x$.

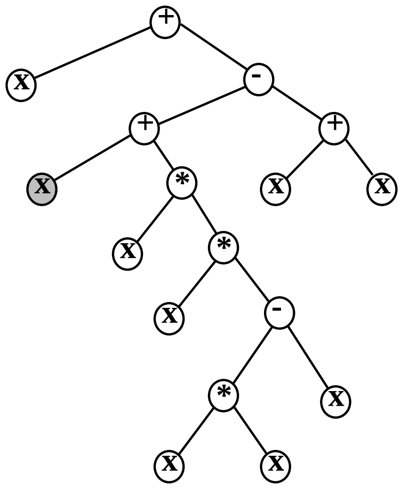


Figure 3: Mum, fitness .71429, $2x + x^2(x^2 - x) - 2x$, Crossover at 4

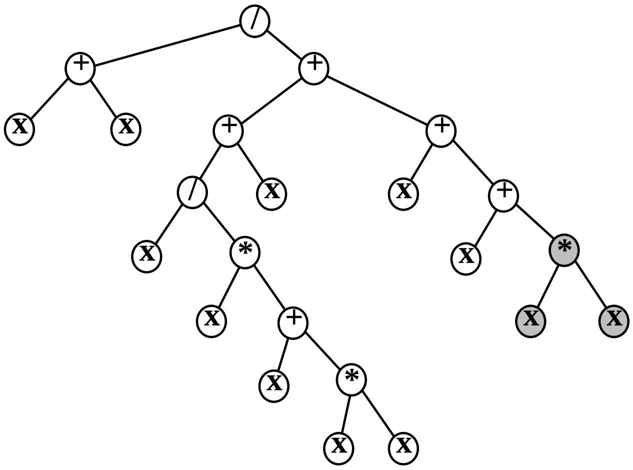


Figure 4: Dad, fitness .66793, $\frac{2x}{\frac{x}{x(x+x^2)}+3x+x^2}$, Crossover at 20

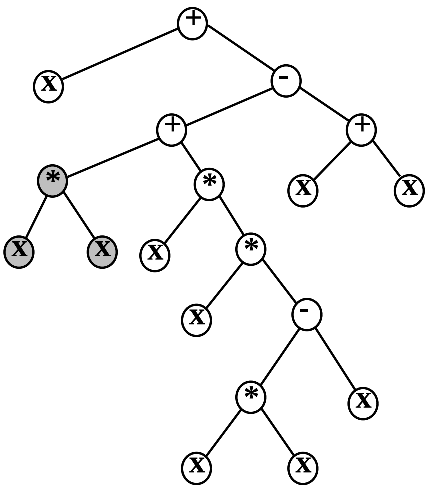


Figure 5: Correct Program: $x + x^2 + x^2(x^2 - x) - 2x$

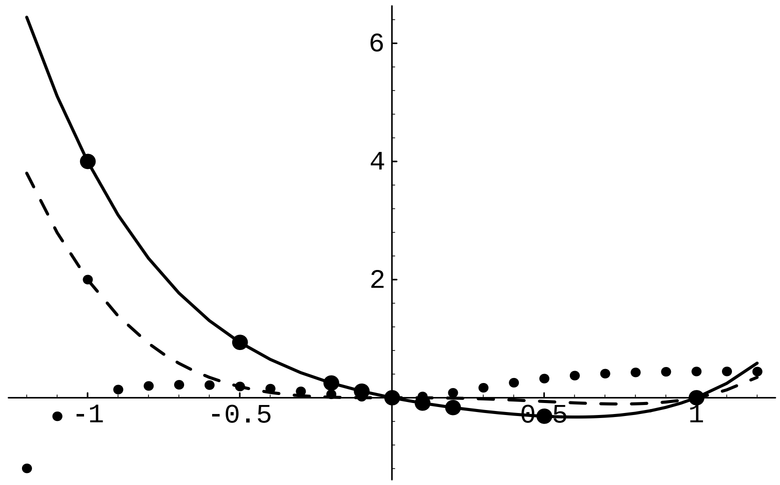



Figure 6: Values returned by *mum* (dashed) and *dad* (small dots)

Figure 6 shows the values of the two parent programs against the test points (which their offspring matches exactly). Note the anomalous behaviour of the dad program at $x = -1$, which is caused by defining divide by zero to be one. This definition gives the program a higher fitness than it would otherwise have had.

[3]Produced by version Feb 1994

# A References

## References


[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Natural Selection.* MIT press, 1992.

[Koz94] John R. Koza. *Genetic Programming II: Scalable Automatic Programming by Means of Automatically Defined Functions.* MIT Press, 1994. Draft.


# B Glossary

**Cellular Automata** A regular array of identical finite state automata whose next state is determined solely by their current state and the state of their neighbours. The most widely seen is the game of *Life* in which complex patterns emerge from a (supposedly infinite) square lattice of simple two state automata whose next state is determined solely the current states of its four closes neighbours and itself.

**Classifiers** A refinement of Genetic Algorithms in which the population consists of a co-operating set of rules which are to learn a problem given a number of test cases. Between each generation the population as a whole is evaluated and a fitness is assigned to each rule using the bucket-brigade algorithm or other credit sharing scheme (eg the Pitt scheme). These schemes aims to reward or punish rules which contribute to a test case according to how good the total solution is by adjusting the individual rules fitness.

At the end of the test data a new generation is created using the Genetic Algorithm as if each rule where independant using its own fitness (measures may be taken are taken to ensure a given rule only appears once in the new population).

**Coevolution** Two or more populations are evolved at the same time. Often the separate populations compete against each other.

**Convergence** Tendency of members of the population to be the same. May be used to mean either their representation or behaviour are identical. Loosely a Genetic algorithm solution has been reached.

**Chromosome** Normally, in Genetic Algorithms the bit string which represents the individual. In genetic programming the individual and its representation are the same both being the program parse tree. In nature many species store their genetic information on more than one chromosome.

**Crossover** Creating a new individual's representation from parts of its parents representations.

**Deme** A separately evolving subset of the whole population. The subsets may be evolved on a different computers. Emigration between subset may be used (see Panmixia).

**Epistasis** A term from biology used to denote that the fitness of an individual depends upon the interaction of a number of their genes. In Genetic Algorithms this would be indicated by the fitness containing a non-linear combination of components of the string.

**Evolution Programming** A population containing a number of trial solutions each of which is evaluated to yield an error. Typically, at the end of each generation, the best half of the population is retained and a new solution is produced from each survivor. The process is continued with the aim that the population should evolve to contain an acceptable solution.

Evolutionary programming like Evolution Strategy produces new children by mutating at random from a single parent solution. The analogue components (eg the connection weights when applied to Artificial Neural Networks) are changed by a gaussian function whose standard deviation is given by a function of the parent's error called its temperature. Digital components (eg presence of a hidden node) are created and destroyed at random.

**Finite State Automata (FSA)** *or Finite State Machine (FSM)* A machine which can be totally described by a finite set of states, it being in one these at any one time, plus a set of rules which determine when it moves from one state to another.

**Fitness Function** A process which evaluates a member of a population and gives it a score or fitness. In most cases the goal is to find an individual with the maximum (or minimum) fitness.

**Function Set** The set of operators used in a genetic program, eg $+ - \times \div$. These act as the branch points in the parse tree, linking other functions or terminals. See also non-terminals.

**Generation** When the children of one population replace their parents in that population. Where some part of the original population is retained, as in steady state GAs, generation typically refers to the interval during which the number of new individuals created is equal to the population size.

**Genetic Algorithm** A population containing a number of trial solutions each of which is evaluated (to yield a fitness) and a new generation is created from the better of them. The process is continued through a number of generations with the aim that the population should evolve to contain an acceptable solution.

GAs are characterised by representing the solution as (often fixed length) string of digital symbols, selecting parents from the current population in proportion to their fitness (or some approximation of this) and the use of crossover as the dominate means of creating new members of the population. The initial population may be created at random or from some known starting point.

**Genetic Operator** An operator in a Genetic Algorithm or genetic programming, which acts upon the chromosome to produce a new individual. Example operators are mutation and crossover.

**Genetic Programming** A subset of Genetic Algorithms. The members of the populations are the parse trees of computer programs whose fitness is evaluated by running them. The reproduction operators (eg crossover) are refined to ensure that the child is syntactically correct (some protection may be given against semantic errors too). This is achieved by acting upon subtrees.

Genetic programming is most easily implemented where the computer language is tree structured so there is no need to explicitly evaluated its parse tree. This is one of the reasons why Lisp is often used for genetic programming.

**Hits** The number of hits an individual scores is the number of test cases for which it returns the correct answer (or close enough to it). This may or may not be a component of the fitness function. When an individual gains the maximum number of hits this may terminate the run.

**Infix Notation** Notation in which the operator separates its operands. Eg $(a + b) \times c$. Infix notation requires the use of brackets to specify the order of evaluation, unlike either prefix or postfix notations.

**Non-Terminal** Functions used to link parse tree together. This name may be used to avoid confusion with functions with no parameters which can only act as end points of the parse tree (ie leaves) and so are part of the terminal set.

**Mutation** Arbitrary change to representation, often at random. In genetic programming, a subtree is replaced by another, all of which is created at random.

**Panmixia** When a population is split into a number of separately evolving populations (demes) but the level of emigration is sufficiently high that they continue to evolve as if a single population.

**Parsimony** Brevity. In GP, this is measured by counting the nodes in the tree. The smaller the program, the smaller the tree, the lower the count and the more parsimonious it is.

**Postfix Notation** *Reverse Polish Notation or Suffix Notation* Notation in which the operator follows its operands. Eg $a + b \times c$ represented as $abc \times +$.

**Prefix Notation** *Polish Notation* Notation in which the operator comes before its operands. Eg $a + b$ represented as $+ab$.

**Premature Convergence** When a Genetic Algorithm's population converges to something which is not the solution you wanted.

**Recombination** as crossover.

**Reproduction** Production of new member of population from existing members. May be used to mean an exact copy of the original member.

**Simulated Annealing** Search technique where a single trial solution is modified at random. An *energy* is defined which represents how good the solution is. The goal is to find the best solution by minimising the energy. Changes which lead to a lower energy are always accepted; an increase is probabilistically accepted. The probability is given by $\exp(-\Delta E/k_B T)$. Where $\Delta E$ is the change in energy, $k_B$ is a constant and T is the *Temperature*. Initially the temperature is high corresponding to a liquid or molten state where large changes are possible and it is progressively reduced using a *cooling schedule* so allowing smaller changes until the system *solidifies* at a low energy solution.

**Terminal Set** A set from which all end (leaf) nodes in the parse trees representing the programs must be drawn. A terminal might be a variable, a constant or a function with no arguments.

**Tournament Selection** A mechanism for choosing individuals from a population. A group (typically between 2 and 7 individuals) are selected at random from the population and the best (normally only one, but possibly more) is chosen.