1 Introduction

The goal of having computers automatically solve problems is central to artificial intelligence, machine learning, and the broad area encompassed by what Turing called "machine intelligence" [415]. Machine learning pioneer Arthur Samuel, in his 1983 talk entitled "AI: Where It Has Been and Where It Is Going" [366], stated that the main goal of the fields of machine learning and artificial intelligence is:

> "to get machines to exhibit behaviour, which if done by humans, would be assumed to involve the use of intelligence."

Genetic programming (GP) is an evolutionary computation (EC) technique that automatically solves problems without having to tell the computer explicitly how to do it. At the most abstract level GP is a *systematic, domain-independent* method for getting computers to *automatically* solve problems starting from a *high-level statement* of what needs to be done.

Over the last decade, GP has attracted the interest of streams of researchers around the globe. This paper is intended to give an overview of the basics of GP, to summarise important work that gave direction and impetus to research in GP as well as to discuss some interesting new directions and applications. Things change fast in this field, as investigators discover new ways of doing things, and new things to do with GP. It is impossible to cover all aspects of this area, even within the generous page limits of this paper. Thus this paper should be seen as a snapshot of the view we, the authors, have at the time of writing.

1.1 GP in a Nutshell

Technically, GP is a special evolutionary algorithm (EA) where the individuals in the population are *computer programs*. So, generation by generation GP *iteratively* transforms populations of programs into other populations of programs as illustrated in Figure 1. During the process, GP constructs new programs by applying genetic operations which are specialised to act on computer programs.

Algorithmically, GP comprises the steps shown in Algorithm 1. The main genetic operations involved in GP (line 5 of Algorithm 1) are the following:

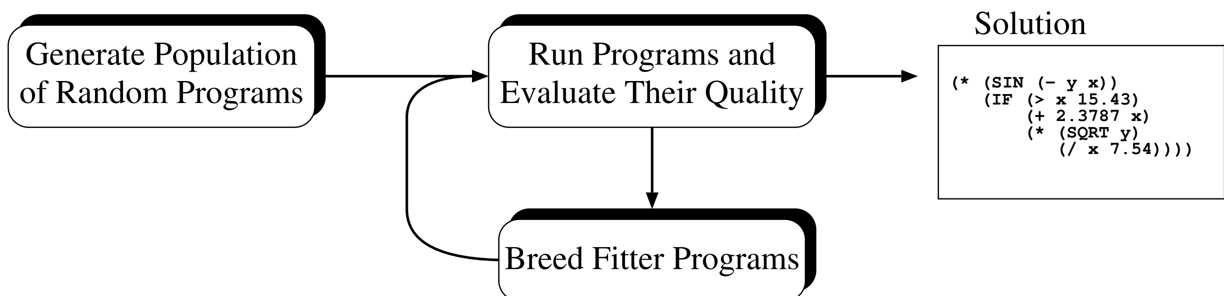

Figure 1: GP's main loop.

Algorithm 1 Abstract GP algorithm.

1: Randomly create an *initial population* of programs from the available primitives (see Section 2.2).
2: **repeat**
3: *Execute* each program and ascertain its fitness.
4: *Select* one or two program(s) from the population with a probability based on fitness to participate in genetic operations (see Section 2.3).
5: Create new individual program(s) by applying *genetic operations* with specified probabilities (see Section 2.4).
6: **until** an acceptable solution is found or some other stopping condition is met (e.g., reaching a maximum number of generations).
7: **return** the best-so-far individual.

- **Crossover:** the creation of one or two offspring programs by recombining randomly chosen parts from two selected programs.

- **Mutation:** the creation of one new offspring program by randomly altering a randomly chosen part of one selected program.

Some GP systems also support structured solutions (see, e.g., Section 5.1), and some of these then include *architecture-altering operations* which randomly alter the architecture (e.g., the number of subroutines) of a program to create a new offspring program. Also, often, in addition of crossover, mutation and the architecture-altering operations, an operation which simply copies selected individuals in the next generation is used. This operation, called *reproduction*, is typically applied only to produce a fraction of the new generation.

1.2 Overview of the Paper

This paper starts with an overview of the key representations and operations in GP (Section 2), a discussion of the decisions that need to be made before running GP (Section 3), and an example of a GP run (Section 4).

This is followed by descriptions of some more advanced GP techniques including: automatically defined functions (Section 5.1) and architecture-altering operations (Section 5.2), the GP problem solver (Section 5.3), systems that constrain the syntax of evolved programs in some way (e.g., using grammars or type systems; Section 5.4) and developmental GP (Section 5.5). Alternative program representations, namely linear GP (Section 6.1) and graph-based GP (Section 6.2) are then discussed.

After this survey of representations, we provide a review of the enormous variety of applications of GP, including curve fitting and data modelling (Section 7.1), human competitive results (Section 7.2) and much more, and a substantial collection of "tricks of the trade" used by experienced GP practitioners (Section 8). We also give an overview of some of the considerable work that has been done on the theory of GP (Section 9).

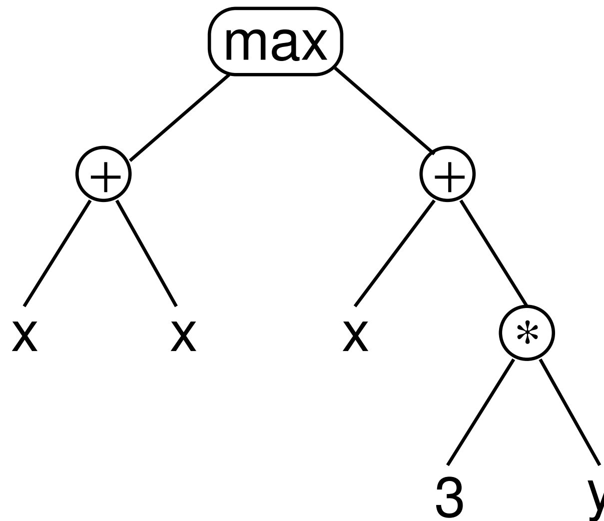

Figure 2: GP syntax tree representing `max(x*x,x+3y)`.

After concluding the paper (Section 10), we provide a resources appendix that reviews the many sources of further information on GP, its applications, and related problem solving systems.

2 Representation, Initialisation and Operators in Tree-based GP

In this section we will introduce the basic tools and terms used in genetic programming. In particular, we will look at how solutions are represented in most GP systems (Section 2.1), how one might construct the initial, random population (Section 2.2), and how selection (Section 2.3) as well as recombination and mutation (Section 2.4) are used to construct new individuals.

2.1 Representation

In GP programs are usually expressed as *syntax trees* rather than as lines of code. Figure 2 shows, for example, the tree representation of the program `max(x*x,x+3*y)`. Note how the variables and constants in the program (`x`, `y`, and `3`), called *terminals* in GP, are leaves of the tree, while the arithmetic operations (`+`, `*`, and `max`) are internal nodes (typically called *functions* in the GP literature). The sets of allowed functions and terminals together form the *primitive set* of a GP system.

In more advanced forms of GP, programs can be composed of multiple components (e.g., subroutines). In this case the representation used in GP is a set of trees (one for each component) grouped together under a special root node that acts as glue, as illustrated in Figure 3. We will call these (sub)trees *branches*. The number and type of the branches in

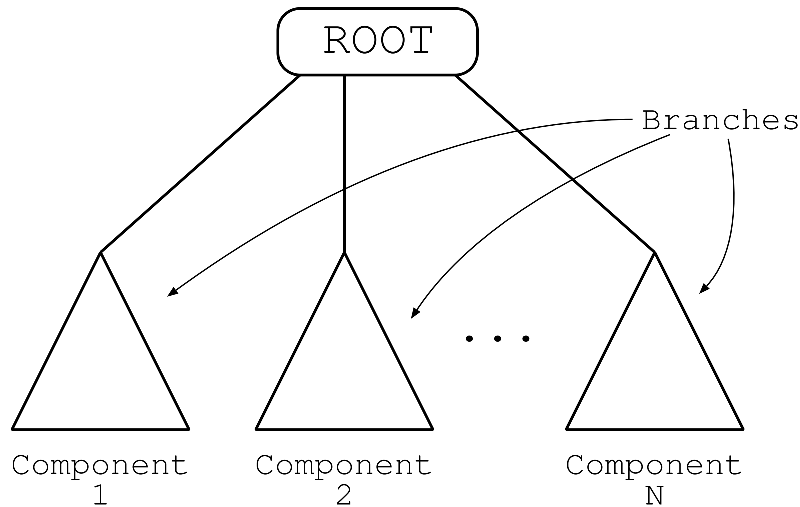

Figure 3: Multi-component program representation.

a program, together with certain other features of the structure of the branches, form the *architecture* of the program.

It is common in the GP literature to represent expressions in a *prefix* notation similar to that used in LISP or Scheme. For example, `max(x*x,x+3*y)` becomes `(max (* x x) (+ x (* 3 y)))`. This notation often makes it easier to see the relationship between (sub)expressions and their corresponding (sub)trees. Therefore, in the following, we will use trees and their corresponding prefix-notation expressions interchangeably.

How one implements GP trees will obviously depend a great deal on the programming languages and libraries being used. Most traditional languages used in AI research (e.g., Lisp and Prolog), many recent languages (e.g., Ruby and Python), and the languages associated with several scientific programming tools (e.g., MATLAB® and Mathemtica®) provide automatic garbage collection and dynamic lists as fundamental data types making it easy to directly implement expression trees and the necessary GP operations. In other languages one may have to implement lists/trees or use libraries that provide such data structures.

In high performance environments, however, the tree-based representation may be too memory-inefficient since it requires the storage and management of numerous pointers. If all functions have a fixed arity (which is extremely common in GP applications) the brackets become redundant in prefix-notation expressions, and the tree can be represented as a simple linear sequence. For example, the expression `(max (* x x) (+ x (* 3 y)))` could be written unambiguously as the sequence `max * x x + x * 3 y`. The choice of whether to use such a linear representation or an explicit tree representation is typically guided by questions of convenience, efficiency, the genetic operations being used (some may be more easily or more efficiently implemented in one representation), and other data one may wish to collect during runs (e.g., it is sometimes useful to attach additional information to nodes, which may require that they be explicitly represented). There are also numerous

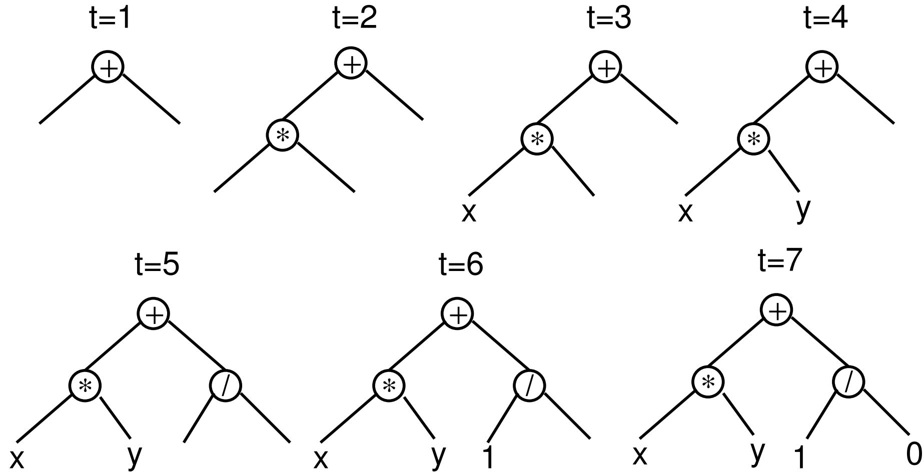

Figure 4: Creation of a full tree having maximum depth 2 (and therefore a total of seven nodes) using the Full initialisation method (t=time).

high-quality, freely available GP implementations (see the resources in the appendix at the end of this paper for more information).

While these tree representations are the most common in GP, there are other important representations, some of which are discussed in Section 6.

2.2 Initialising the Population

Similar to other evolutionary algorithms, in GP the individuals in the initial population are randomly generated. There are a number of different approaches to generating this random initial population. Here we will describe two of the simplest (and earliest) methods (the *Full* and *Grow* methods), and a widely used combination of the two known as *Ramped half-and-half.*

In both the Full and Grow methods, the initial individuals are generated subject to a pre-established maximum depth. In the Full method (so named because it generates full trees) nodes are taken at random from the function set until this maximum tree depth is reached, and beyond that depth only terminals can be chosen. Figure 4 shows snapshots of this process in the construction of a full tree of depth 2. The children of the `*` node, for example, must be leaves, or the resulting tree would be too deep; thus at time $t = 3$ and time $t = 4$ terminals must be chosen (`x` and `y` in this case).

Where the Full method generates trees of a specific size and shape, the Grow method allows for the creation of trees of varying size and shape. Here nodes are selected from the whole primitive set (functions *and* terminals) until the depth limit is reached, below which only terminals may be chosen (as is the case in the Full method). Figure 5 illustrates this process for the construction of a tree with depth limit 2. Here the first child of the root `+` node happens to be a terminal, thus closing off that branch before actually reaching the depth limit. The other child, however, is a function (`-`), but its children are forced to be

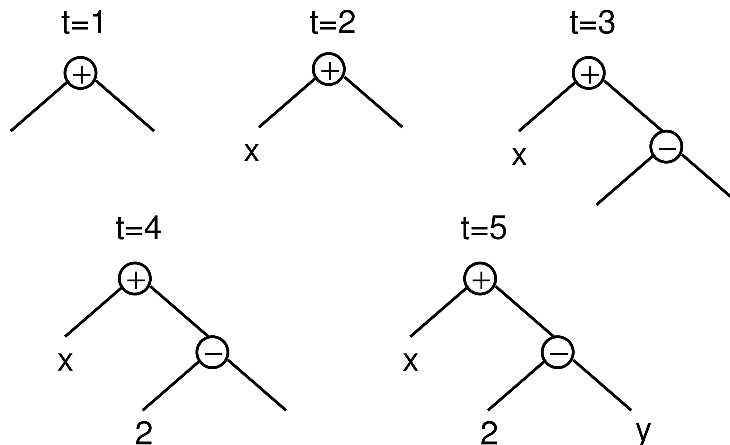

Figure 5: Creation of a five node tree using the Grow initialisation method with a maximum depth of 2 (t=time). A terminal is chosen at $t = 2$, causing the left branch of the root to be closed at that point even though the maximum depth had not been reached.

terminals to ensure that the resulting tree does not exceed the depth limit.

Pseudo code for a recursive implementation of both the Full and Grow methods is given in Algorithm 2.

Note here that the size and shapes of the trees generated via the Grow method are highly sensitive to the sizes of the function and terminal sets. If, for example, one has significantly more terminals than functions, the Grow method will almost always generate very short trees regardless of the depth limit. Similarly, if the number of functions is considerably greater than the number of terminals, then the Grow method will behave quite similarly to the Full method. While this is a particular problem for the Grow method, it illustrates a general issue where small (and often apparently inconsequential) changes such as the addition or removal of a few functions from the function set can in fact have significant implications for the GP system, and potentially introduce important unintended biases.

Because neither the Grow or Full method provide a very wide array of sizes or shapes on their own, Koza [203] proposed a combination called *ramped half-and-half*. Here half the initial population is constructed using Full and half is constructed using Grow. This is done using a range of depth limits (hence the term "ramped") to help ensure that we generate trees having a variety of sizes and shapes.

While these methods are easy to implement and use, they often make it difficult to control the statistical distributions of important properties such as the sizes and shapes of the generated trees. Other initialisation mechanisms, however, have been developed (e.g., [263]) that do allow for closer control of these properties in instances where such control is important.

It is also worth noting that the initial population need not be entirely random. If something is known about likely properties of the desired solution, trees having these properties can be used to seed the initial population. Such seeds might be created by humans based on knowledge of the problem domain, or they could be the results of previous GP runs.

Algorithm 2 Pseudo code for recursive program generation with the "Full" and "Grow" methods.

procedure: gen_rnd_expr(func_set, term_set, max_d, method)

1: **if** max_d = 0 **or** $\left(\text{method} = \text{grow } \textbf{and} \text{ rand()} < \frac{|\text{term_set}|}{|\text{term_set}|+|\text{func_set}|}\right)$ **then**
2: expr = choose_random_element(term_set)
3: **else**
4: func = choose_random_element(func_set)
5: **for** i = 1 **to** arity(func) **do**
6: arg_i = gen_rnd_expr(func_set, term_set, max_d - 1, method);
7: **end for**
8: expr = (func, arg_1, arg_2, ...);
9: **end if**
10: **return** expr

Notes: **func_set** is a function set, **term_set** is a terminal set, **max_d** is the maximum allowed depth for expressions, **method** is either "Full" or "Grow" and **expr** is the generated expression in prefix notation.

However, one needs to be careful not to create a situation where the second generation is dominated by offspring of a single or very small number of seeds. Diversity preserving techniques, such as multi-objective GP (e.g., [313, 374]), demes [242] (see Section 8.6), fitness sharing [125] and the use of multiple seed trees, might be used. In any case, the diversity of the population should be monitored to ensure that there is significant mixing of different initial trees.

2.3 Selection

Like in most other EAs, genetic operators in GP are applied to individuals that are probabilistically selected based on fitness. That is, better individuals are more likely to have more child programs than inferior individuals. The most commonly employed method for selecting individuals in GP is tournament selection, followed by fitness-proportionate selection, but any standard EA selection mechanism can be used. Since selection has been described many times in the EA literature, we will not provide any additional details.

2.4 Recombination and Mutation

Where GP departs significantly from other EAs is in the implementation of the operators of crossover and mutation. The most commonly used form of crossover is *subtree crossover*. Given two parents, subtree crossover randomly selects a crossover point in each parent tree. Then, it creates the offspring by replacing the sub-tree rooted at the crossover point in a copy of the first parent with a copy of the sub-tree rooted at the crossover point in the second parent, as illustrated in Figure 6.

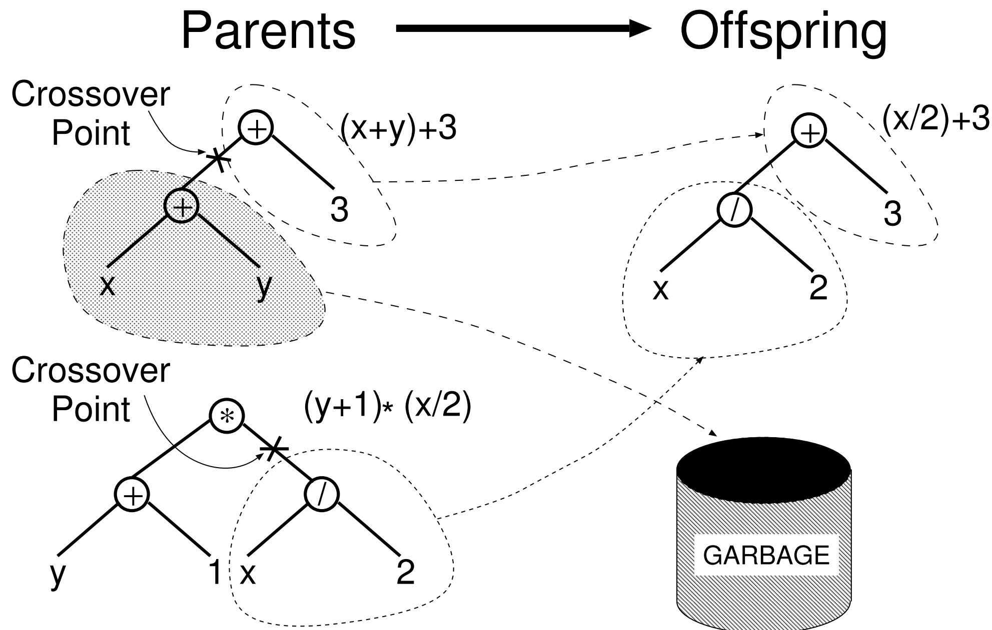

Figure 6: Example of subtree crossover.

Except in technical studies on the behaviour of GP, crossover points are usually *not* selected with uniform probability. Typical GP primitive sets lead to trees with an average branching factor of at least 2, so the majority of the nodes will be leaves. Consequently the uniform selection of crossover points leads to crossover operations frequently exchanging only very small amounts of genetic material (i.e., small subtrees); many crossovers may in fact reduce to simply swapping two leaves. To counter this, Koza suggested the widely used approach of choosing functions 90% of the time, while leaves are selected 10% of the time.

While subtree crossover is the most common version of crossover in tree-based GP, other forms have been defined and used. For example, *one-point crossover* [326, 329, 239] works by selecting a *common* crossover point in the parent programs and then swapping the corresponding subtrees. To account for the possible structural diversity of the two parents, one-point crossover analyses the two trees from the root nodes and considers for the selection of the crossover point only the parts of the two trees, called the *common region*, which have the same topology (i.e. the same arity in the nodes encountered traversing the trees from the root node). In *context-preserving crossover* [87], the crossover points are constrained to have the same coordinates, like in one-point crossover. However, in this case no other constraint is imposed on their selection (i.e., they are not limited to the common region). In *size-fair crossover* [224, 243] the first crossover point is selected randomly like in standard crossover. Then the size of the subtree to be excised from the first parent is calculated. This is used to constrain the choice of the second crossover point so as to guarantee that the subtree excised from the second parent will not be "unfairly" big. Finally, it is worth mentioning that the notion of common region is related to the notion of

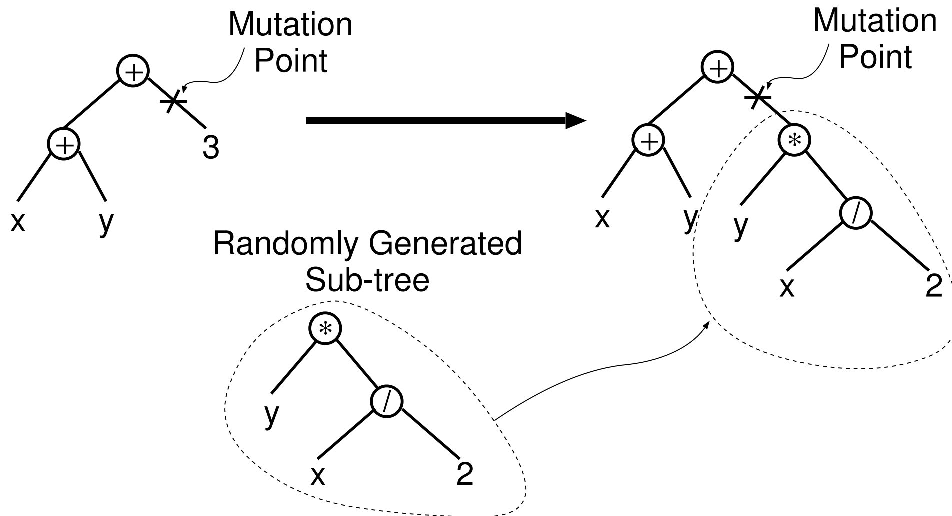

Figure 7: Example of subtree mutation.

homology, in the sense that the common region represents the result of a matching process between parent trees. It is then possible to imagine that within such a region transfer of homologous primitives can happen in very much like the same way as it happens in GAs operating on linear chromosomes. An example of recombination operator that implements this idea is *uniform crossover* for GP [328].

The most commonly used form of mutation in GP (which we will call *subtree mutation*) randomly selects a mutation point in a tree and substitutes the sub-tree rooted there with a randomly generated sub-tree. This is illustrated in Figure 7. Subtree mutation is sometimes implemented as crossover between a program and a newly generated random program; this operation is also known as "headless chicken" crossover [10].

Another common form of mutation is *point mutation*, which is the rough equivalent for GP of the bit-flip mutation used in GAs. In point mutation a random node is selected and the primitive stored there is replaced with a different random primitive of the same arity taken from the primitive set. If no other primitives with that arity exist, nothing happens to that node (but other nodes may still be mutated). Note that, when subtree mutation is applied, this involves the modification of exactly one subtree. Point mutation, on the other hand, is typically applied with a given mutation rate on a per-node basis, allowing multiple nodes to be mutated independently.

There are a number of mutation operators which treat constants in the program as special cases. [371] mutates constants by adding Gaussianly distributed random noise to them. However, others use a variety of potentially expensive optimisation tools to try and fine tune an existing program by finding the "best" value for constants within it. E.g., [369] uses "a numerical partial gradient ascent . . . to reach the nearest local optimum" to modify all constants in a program, while [378] uses simulated annealing to stochastically update numerical values within individuals.

While mutation is not necessary for GP to solve many problems, [307] argues that, in

some cases, GP with mutation alone can perform as well as GP using crossover. While mutation was often used sparsely in early GP work, it is more widely used in GP today, especially in modelling applications.

3 Getting Ready to Run Genetic Programming

To run a GP system to solve a problem a small number of ingredients, often termed *preparatory steps*, need to be specified:

1. the terminal set,
2. the function set,
3. the fitness measure,
4. certain parameters for controlling the run, and
5. the termination criterion and method for designating the result of the run.

In this section we consider these ingredients in more detail.

3.1 Steps 1: Terminal Set

While it is common to describe GP as evolving *programs*, GP is not typically used to evolve programs in the familiar, Turing-complete languages humans normally use for software development. It is instead more common to evolve programs (or expressions or formulae) in a more constrained and often domain-specific language. The first two preparatory steps, the definition of the terminal and function sets, specify such a language, i.e., the ingredients that are available to GP to create computer programs.

The terminal set may consist of:

- *the program's external inputs*, typically taking the form of named variables (e.g., `x`, `y`);
- *functions with no arguments*, which are, therefore, interesting either because they return different values in different invocations (e.g., the function `rand()` that returns random numbers, or a function `dist_to_wall()` that returns the distance from the robot we are controlling to an obstacle) or because the produce side effects (e.g., `go_left()`); and
- *constants*, which can either be pre-specified or randomly generated as part of the tree creation process.

Table 1: Examples of primitives allowed in the GP function and terminal sets.

Function Set	
Kind of Primitive	*Example(s)*
Arithmetic	`+`, `*`, `/`
Mathematical	`sin`, `cos`, `exp`
Boolean	`AND`, `OR`, `NOT`
Conditional	`IF-THEN-ELSE`
Looping	`FOR`, `REPEAT`
⋮	⋮

Terminal Set	
Kind of Primitive	*Example(s)*
Variables	`x`, `y`
Constant values	`3`, `0.45`
0-arity functions	`rand`, `go_left`

Note that using a primitive such as `rand` can cause the behaviour of an individual program to vary every time it is called, even if it is given the same inputs. What we often want instead is a set of fixed random constants that are generated as part of the process of initialising the population. This is typically accomplished by introducing a terminal that represents an *ephemeral random constant*. Every time this terminal is chosen in the construction of an initial tree (or a new subtree to use in an operation like mutation), a different random value is generated which is then used for that *particular* terminal, and which will remain fixed for the rest of the run. The use of ephemeral random constants is typically denoted by including the symbol $\Re$ in the terminal set; see Section 4 for an example.

3.2 Step 2: Function Set

The function set used in GP is typically driven by the nature of the problem domain. In a simple numeric problem, for example, the function set may consist of merely the arithmetic functions (`+`, `-`, `*`, `/`). However, all sorts of other functions and constructs typically encountered in computer programs can be used. Table 1 shows a sample of some of the functions one sees in the GP literature. Also for many problems, the primitive set includes specialised functions and terminals which are expressly designed to solve problems in a specific domain of application. For example, if the goal is to program a robot to mop the floor, then the function set might include such actions as `move`, `turn`, and `swish-the-mop`.

3.2.1 Closure

For GP to work effectively, most function sets are required to have an important property known as *closure* [203], which can in turn be broken down into the properties of *type consistency* and *evaluation safety*.

Type consistency is necessitated by the fact that subtree crossover (as described in Section 2.4) can mix and join nodes quite arbitrarily during the evolutionary process. As a result it is necessary that *any* subtree can be used in any of the argument positions for every function in the function set, because it is always possible that sub-tree crossover

will generate that combination. For functions that return a value (e.g., `+`, `-`, `*`, `/`), it is then common to require that all the functions be type consistent, namely that they all return values of the same type, and that all their arguments be of that type as well. In some cases this requirement can be weakened somewhat by providing an automatic conversion mechanism between types, e.g., converting numbers to Booleans by treating all negative values as false, and non-negative values as true. Conversion mechanisms like this can, however, introduce unexpected biases into the search process, so they should be used thoughtfully.

The requirement of type consistency can seem quite limiting, but often simple restructuring of the functions can resolve apparent problems. An `if` function, for example, would often be defined as taking three arguments: The test, the value to return if the test evaluates to *true*, and the value to return if the test evaluates to *false*. The first of these three arguments is clearly Boolean, which would suggest that `if` can't be used with numeric functions like `+`. This can easily be worked around however by providing a mechanism to automatically convert a numeric value into a Boolean as discussed above. Alternatively, one can replace the traditional `if` with a function of four (numeric) arguments a, b, c, d with the semantics "If $a < b$ then return value c, otherwise return value d". These are obviously just specific examples of general techniques; the details are likely to depend on the particulars of your problem domain.

An alternative to requiring type consistency is to extend the GP system to, for example, explicitly include type information, and constrain operations like crossover so they do not perform "illegal" (from the standpoint of the type system) operations. This is discussed further in Section 5.4.

The other component of closure is evaluation safety, necessitated by the fact that many commonly used functions can fail in various ways. An evolved expression might, for example, divide by 0, or call `MOVE_FORWARD` when facing a wall or precipice. This is typically dealt with by appropriately modifying the standard behaviour of primitives. It is common, for example, to use *protected* versions of numeric functions that can throw exceptions, such as division, logarithm, and square root. The protected version of such a function first tests for potential problems with its input(s) before executing the corresponding instruction, and if a problem is spotted some pre-fixed value is returned. Protected division (often notated with `%`), for example, checks for the case that its second argument is 0, and typically returns 1 if it is (regardless of the value of the first argument).[1] Similarly, `MOVE_AHEAD` can be modified to do nothing if a forward move is illegal for some reason or, if there are no other obstacles, the edges can simply be eliminated by making the world toroidal.

An alternative to protected functions is to trap run-time exceptions and strongly reduce the fitness of programs that generate such errors. If the likelihood of generating invalid expressions is very high, however, this method can lead to all the individuals in the population having nearly the same (very poor) fitness, leaving selection with very little

[1]The decision to return 1 here provides the GP system with a simple and reliable way to generate the constant 1, via an expression of the form `(/ x x)`. This, combined with a similar mechanism for generating 0 via `(- x x)` ensures that GP can easily construct these two important constant.

discriminatory power.

One type of run-time error that is somewhat more difficult to check for is numeric overflow. If the underlying implementation system throws some sort of exception, then this can be handled either by protection or by penalizing as discussed above. If, however, the implementation language quietly ignores the overflow (e.g., the common practice of wrapping around on integer overflow), and if this behavior is seen as unacceptable, then the implementation will need to include appropriate checks to catch and handle such overflows.

3.2.2 Sufficiency

There is one more property that, ideally, primitives sets should have: *sufficiency*. Sufficiency requires that the primitives in the primitive set are capable of expressing the solutions to the problem, i.e., that the set of all the possible recursive compositions of such primitives includes at least one solution. Unfortunately, sufficiency can be guaranteed only for some problems, when theory or experience with other methods tells us that a solution can be obtained by combining the elements of the primitive set.

As an example of a sufficient primitive set let us consider the set {`AND`, `OR`, `NOT`, `x1`, `x2`, ..., `xN`}, which is always sufficient for Boolean function induction problems, since it can produce all Boolean functions of the variables `x1`, `x2`, ..., `xN`. An example of insufficient set is the set {`+`, `-`, `*`, `/`, `x`, `0`, `1`, `2`}, which is insufficient whenever, for example, the target function is transcendental, e.g., $\exp(x)$, and therefore cannot be expressed as a rational function (basically, a ratio of polynomials). When a primitive set is insufficient for a particular application, GP can only develop programs that approximate the desired one, although perhaps very closely.

3.2.3 Evolving Structures other than Programs

There are many problems in the real world where solutions cannot be directly cast as computer programs. For example, in many design problems the solution is an artifact of some type (a bridge, a circuit, etc.). GP has been applied to problems of this kind by using a trick: the primitive set is designed in such a way that, through their execution, programs construct solutions to the problem. This has been viewed as analogous to the development by which an egg grows into an adult. For example, if the goal is the automatic creation of an electronic controller for a plant, the function set might include common components such as `integrator`, `differentiator`, `lead`, `lag`, and `gain`, and the terminal set might contain `reference`, `signal`, and `plant output`. Each of these operations, when executed, then insert the corresponding device into the controller being built. If, on the other hand, the goal is the synthesis of analogue electrical circuits the function set might include components such as transistors, capacitors, resistors, etc. This is further discussed in Section 5.5.

3.3 Step 3: Fitness Function

The first two preparatory steps define the primitive set for GP, and therefore, indirectly define the search space GP will explore. This includes all the programs that can be constructed by composing the primitives in all possible ways. However, at this stage we still do not know which elements or regions of this search space are good (i.e., include programs that solve or approximately solve the problem). This is the task of the fitness measure, which effectively (albeit implicitly) specifies the desired goal of the search process. The fitness measure is our primary (and often sole) mechanism for giving a high-level statement of the problem's requirements to the GP system. For example, if the goal is to get GP to automatically synthesise an amplifier, the fitness function is the mechanism for telling GP to synthesise a circuit that amplifies an incoming signal (as opposed to, say, a circuit that suppresses the low frequencies of an incoming signal or computes its square root).

Depending on the problem at hand, fitness can be measured in terms of the amount of *error* between its output and the desired output, the amount of *time* (fuel, money, etc.) required to bring a system to a desired *target state*, the *accuracy* of the program in recognising patterns or classifying objects into classes, the *payoff* that a game-playing program produces, the *compliance* of a structure with user-specified design criteria, etc.

There is something unusual about the fitness functions used in GP that differentiates them from those used in most other EAs. Because the structures being evolved in GP are computer programs, fitness evaluation normally requires executing all the programs in the population, typically multiple times. While one can compile the GP programs that make up the population, the overhead is usually substantial, so it is much more common to use an interpreter to evaluate the evolved programs.

Interpreting a program tree means executing the nodes in the tree in an order that guarantees that nodes are not executed before the value of their arguments (if any) is known. This is usually done by traversing the tree recursively starting from the root node, and postponing the evaluation of each node until the value of its children (arguments) is known. This process is illustrated in Figure 8, where the number to the right of each internal node represents the result of evaluating the subtree root at that node. In this example, the independent variable `X` evaluates to `-1`. Algorithm 3 gives a pseudo-code implementation of the interpretation procedure. The code assumes that programs are represented as prefix-notation expressions and that such expressions can be treated as lists of components.

In some problems we are interested in the *output* produced by a program, i.e., the value returned when we evaluate starting at the root node. In other problems, however, we are interested in the actions performed by a program. In this case the primitive set will include functions with side effects, i.e., functions that do more than just return a value, but, for example, change some global data structures, print or draw something on the screen or control the motors of a robot. Irrespective of whether we are interested in program outputs or side effects, quite often the fitness of a program depends on the results produced by its execution on many different inputs or under a variety of different conditions. These different test cases typically incrementally contribute to the fitness value

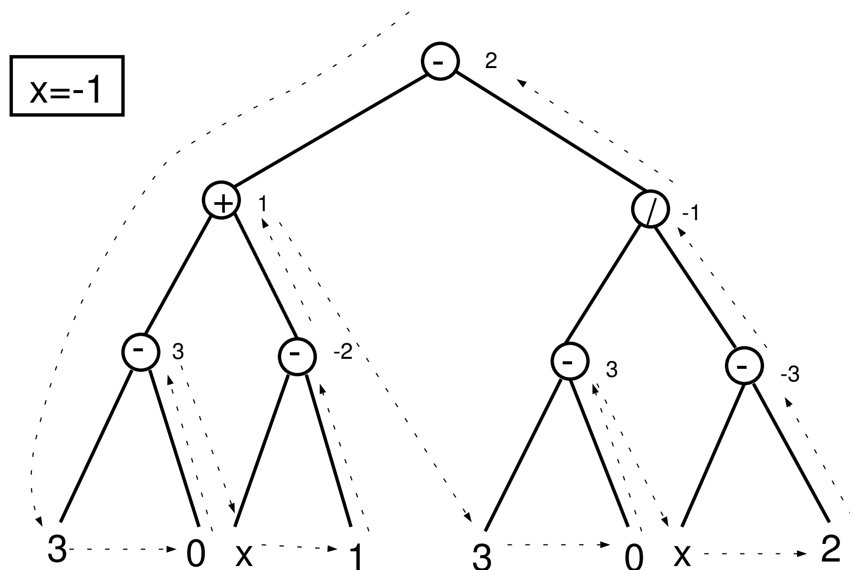

Figure 8: Example interpretation of a syntax tree (the terminal x is a variable has a value of -1). The number to the right of each internal node represents the result of evaluating the subtree root at that node.

of a program, and for this reason are called *fitness cases*.

Another common feature of GP fitness measures is that, for many practical problems, they are *multi-objective,*, i.e., they combine two or more different elements that are often in competition with one another. The area of multi-objective optimization is a complex and active area of research in GP and machine learning in general; see [80], for example, for more.

3.4 Steps 4 and 5: Parameters and Termination

The fourth and fifth preparatory steps are administrative. The fourth preparatory step entails specifying the control parameters for the run. The most important control parameter is the population size. Other control parameters include the probabilities of performing the genetic operations, the maximum size for programs, and other details of the run.

The fifth preparatory step consists of specifying the termination criterion and the method of designating the result of the run. The termination criterion may include a maximum number of generations to be run as well as a problem-specific success predicate. Typically the single best-so-far individual is then harvested and designated as the result of the run, although one might wish to return additional individuals and data as necessary or appropriate for your problem domain.

Algorithm 3 Typical interpreter for GP.

procedure: eval(expr)

```
1:  if expr is a list then
2:     proc = expr(1) {Non-terminal: extract root}
3:     if proc is a function then
4:        value = proc( eval(expr(2)), eval(expr(3)), ... ) {Function: evaluate arguments}
5:     else
6:        value = proc( expr(2), expr(3), ...) {Macro: don't evaluate arguments}
7:     end if
8:  else
9:     if expr is a variable or expr is a constant then
10:       value = expr {Terminal variable or constant: just read the value}
11:    else
12:       value = expr() {Terminal 0-arity function: execute}
13:    end if
14: end if
15: return value
```

Notes: **expr** is an expression in prefix notation, **expr(1)** represents the primitive at the root of the expression, **expr(2)** represents the first argument of that primitive, **expr(3)** represents the second argument, etc.

4 Example of a Run of Genetic Programming

This section provides a concrete, illustrative run of GP in which the goal is to automatically evolve an expression whose values match those of the quadratic polynomial $x^2 + x + 1$ in the range $[-1, +1]$. That is, the goal is to automatically create a computer program that matches certain numerical data. This process is sometimes called *system identification* or *symbolic regression* (see Section 7.1 for more).

We begin with the five preparatory steps from the previous section, and then describe in detail the events in one possible run.

4.1 Preparatory Steps

The purpose of the first two preparatory steps is to specify the ingredients the evolutionary process can use to construct potential solutions. Because the problem is to find a mathematical function of one independent variable, x, the terminal set (the inputs to the to-be-evolved programs) must include this variable. The terminal set also includes ephemeral random constants, drawn from some reasonable range, say from -5.0 to $+5.0$, as described in Section 3.1. Thus the terminal set, T, is

$$T = \{x, \Re\}.$$

The statement of the problem is somewhat flexible in that it does not specify what func-

tions may be employed in the to-be-evolved program. One simple choice for the function set consists of the four ordinary arithmetic functions: addition, subtraction, multiplication, and division. Most numeric regression will include at least these operations, often in conjunction with additional functions such as sin and log. In our example, however, we will restrict ourselves to the simple function set

$$F = \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{\%}\},$$

where `%` is protected division as discussed in Section 3.2.1.

The third preparatory step involves constructing the fitness measure that specifies what the human wants. The high-level goal of this problem is to find a program whose output is equal to the values of the quadratic polynomial x^2+x+1. Therefore, the fitness assigned to a particular individual in the population for this problem must reflect how closely the output of an individual program comes to the target polynomial $x^2 + x + 1$.

The fitness measure *could* be defined as the integral of the absolute value of the differences (errors) between the individual mathematical expression and the target quadratic polynomial x^2+x+1, taken over the range $[-1, +1]$. However, for most symbolic regression problems, it is not practical or possible to analytically compute the value of the integral of the absolute error. Thus it is common to instead define the fitness to be the *sum of absolute errors* measured at different values of the independent variable x in the range $[-1.0, +1.0]$. In particular, we will measure the errors for $x = -1.0, -0.9, \cdots, 0.9, 1.0$. A smaller value of fitness (error) is better; a fitness (error) of zero would indicate a perfect fit. Note that with this definition, our fitness is (approximately) proportional to the area between the parabola $x^2 + x + 1$ and the curve representing the candidate individual (see Figure 10 for examples).

The fourth step is where we set our run parameters. The population size in this small illustrative example will be just four. In actual practice, the population size for a run of GP typically consists of thousands or millions of individuals, but we will use this tiny population size to keep the example manageable. In practice, the crossover operation is commonly used to generate about 90% of the individuals in the population; the reproduction operation (where a fit individual is simply copied from one generation to the next) is used to generate about 8% of the population; the mutation operation is used to generate about 1% of the population; and the architecture-altering operations (see Section 5.2) are used to generate perhaps 1% of the population. Because this example involves an abnormally small population of only four individuals, the crossover operation will be used to generate two individuals, and the mutation and reproduction operations will each be used to generate one individual. For simplicity, the architecture-altering operations are not used for this problem.

In the fifth and final step we need to specify a termination condition. A reasonable termination criterion for this problem is that the run will continue from generation to generation until the fitness (or error) of some individual is less than 0.1. In this contrived example, our example run will (atypically) yield an algebraically perfect solution (with a fitness of zero) after merely one generation.

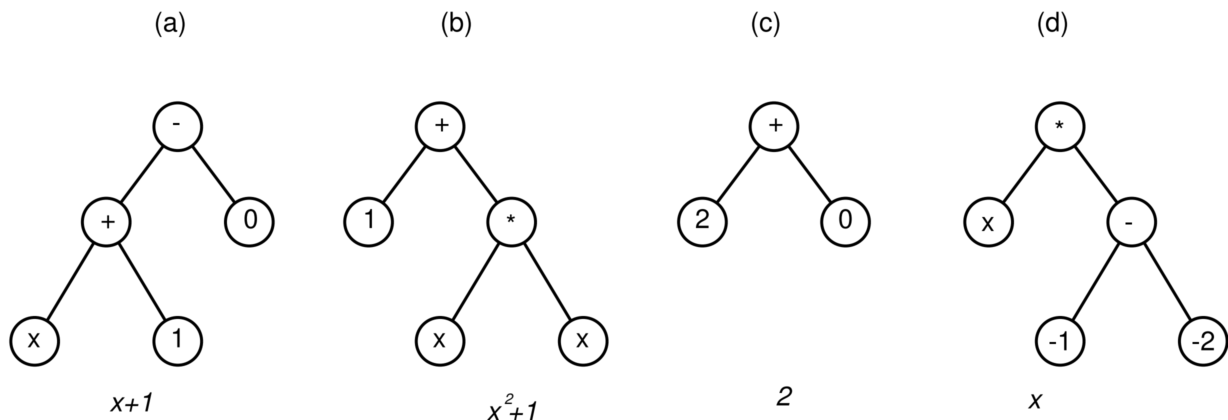

Figure 9: Initial population of four randomly created individuals of generation 0

4.2 Step-by-Step Sample Run

Now that we have performed the five preparatory steps, the run of GP can be launched.

4.2.1 Initialisation

GP starts by randomly creating a population of four individual computer programs. The four programs are shown in Figure 9 in the form of trees.

The first randomly constructed program tree (Figure 9a), and is equivalent to the expression $x + 1$. The second program (Figure 9b) adds the constant terminal 1 to the result of multiplying x by x and is equivalent to x^2+1. The third program (Figure 9c) adds the constant terminal 2 to the constant terminal 0 and is equivalent to the constant value 2. The fourth program (Figure 9d) is equivalent to x.

4.2.2 Fitness Evaluation

Randomly created computer programs will, of course, typically be very poor at solving the problem at hand. However, even in a population of randomly created programs, some programs are better than others. Here, for example, the four random individuals from generation 0 in Figure 9 produce outputs that deviate by different amounts from the target function $x^2 + x + 1$. Figure 10 compares the plots of each of the four individuals in Figure 9 and the target quadratic function $x^2 + x + 1$. The sum of absolute errors for the straight line x+1 (the first individual) is 7.7 (Figure 10a). The sum of absolute errors for the parabola x^2+1 (the second individual) is 11.0 (Figure 10b). The sums of the absolute errors for the remaining two individuals are 17.98 (Figure 10c) and 28.7 (Figure 10d), respectively.

As can be seen in Figure 10, the straight line x+1 (Figure 10a) is closer to the parabola $x^2 + x + 1$ in the range from –1 to +1 than any of three other programs in the population. This straight line is, of course, not equivalent to the parabola $x^2 + x + 1$; it is not even a quadratic function. It is merely the best candidate that happened to emerge from the

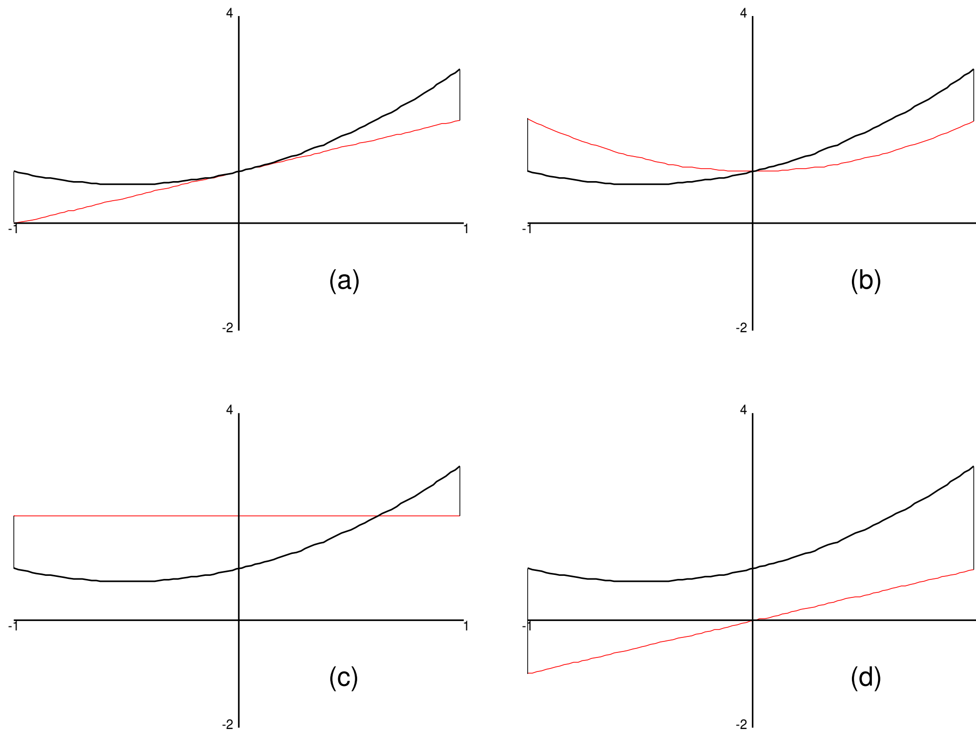

Figure 10: Graphs of the evolved functions from generation 0. The heavy line in each plot is the target function $x^2 + x + 1$, with the other line being the evolved functions from the first generation (see Figure 9). The fitness of each of the four randomly created individuals of generation 0 is approximately proportional to the area between two curves, with the actual fitness values being 7.7, 11.0, 17.98 and 28.7 for individuals (a) through (d), respectively.

blind (and very limited) random search of generation 0. In the valley of the blind, the one-eyed man is king.

4.2.3 Selection, Crossover and Mutation

After the fitness of each individual in the population is ascertained, GP then probabilistically selects relatively more fit programs from the population to act as the parents of the next generation. The genetic operations are applied to the selected individuals to create offspring programs. The important point for our example is that our selection process is not greedy. Individuals that are known to be inferior will be selected to a certain degree. The best individual in the population is not guaranteed to be selected and the worst individual in the population will not necessarily be excluded.

In this example, we will start with the reproduction operation. Because the first individual (Figure 9a) is the most fit individual in the population, it is very likely to be selected to participate in a genetic operation. Let us suppose that this particular individual is, in fact, selected for reproduction. If so, it is copied, without alteration, into the

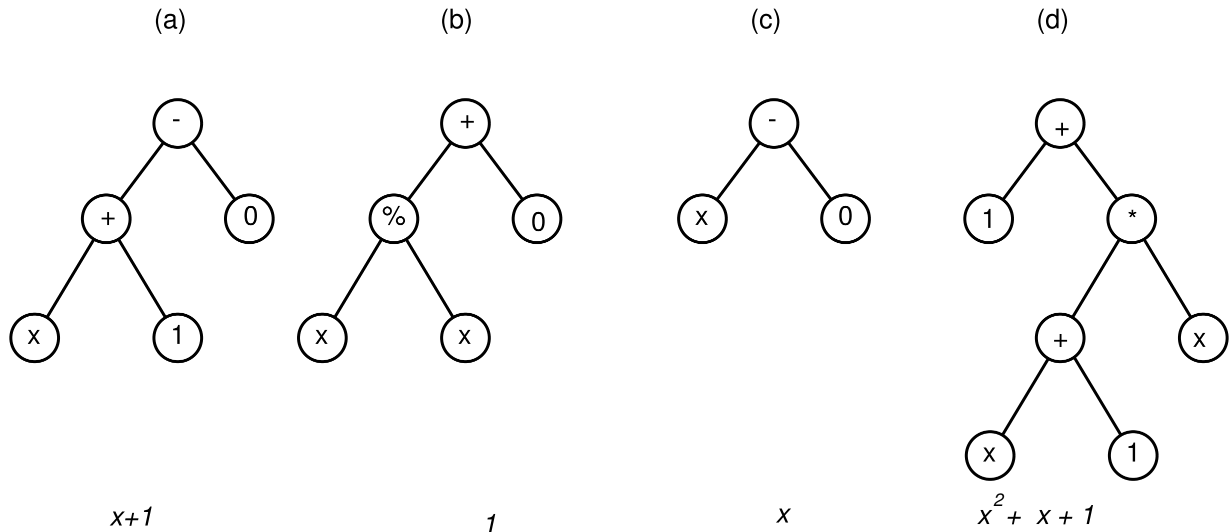

Figure 11: Population of generation 1 (after one reproduction, one mutation, and one two-offspring crossover operation).

next generation (generation 1). It is shown in Figure 11a as part of the population of the new generation.

We next perform the mutation operation. Because selection is probabilistic, it is possible that the third best individual in the population (Figure 9c) is selected. One of the three nodes of this individual is then randomly picked as the site for the mutation. In this example, the constant terminal 2 is picked as the mutation site. This program is then randomly mutated by deleting the entire subtree rooted at the picked point (in this case, just the constant terminal 2) and inserting a subtree that is randomly constructed in the same way that the individuals of the initial random population were originally created. In this particular instance, the randomly grown subtree computes the quotient of x and x using the protected division operation %. The resulting individual is shown in Figure 11b. This particular mutation changes the original individual from one having a constant value of 2 into one having a constant value of 1, improving its fitness from 17.98 to 11.0.

Finally, we use the crossover operation to generate our final two individuals for the next generation. Because the first and second individuals in generation 0 are both relatively fit, they are likely to be selected to participate in crossover. However, selection can always pick suboptimal individuals. So, let us assume that in our first application of crossover the pair of selected parents is composed of the above-average tree in Figures 9a and the below-average tree in Figure 9d. One point of the first parent, namely the + function in Figure 9a, is randomly picked as the crossover point for the first parent. One point of the second parent, namely the leftmost terminal x in Figure 9d, is randomly picked as the crossover point for the second parent. The crossover operation is then performed on the two parents. The offspring (Figure 11c) is equivalent to x and is not particularly noteworthy. Let us now assume, that in our second application of crossover, selection

chooses the two most fit individuals as parents: the individual in Figure 9b as the first parent, and the individual in Figure 9a as the second. Let us further imagine that crossover picks the leftmost terminal x in Figure 9b as a crossover point for the first parent, and the + function in Figure 9a as the crossover point for the second parent. Now the offspring (Figure 11d) is equivalent to $x^2 + x + 1$ and has a fitness (sum of absolute errors) of zero. Because the fitness of this individual is below 0.1, the termination criterion for the run is satisfied and the run is automatically terminated. This best-so-far individual (Figure 11d) is then designated as the result of the run.

Note that the best-of-run individual (Figure 11d) incorporates a good trait (the quadratic term x^2) from the first parent (Figure 9b) with two other good traits (the linear term x and constant term of 1) from the second parent (Figure 9a). The crossover operation thus produced a solution to this problem by recombining good traits from these two relatively fit parents into a superior (indeed, perfect) offspring.

This is, obviously, a highly simplified example, and the dynamics of a real GP run are typically far more complex than what is presented here. Also, in general there is no guarantee that an exact solution like this will be found by GP.

5 Advanced Tree-based GP Techniques

5.1 Automatically Defined Functions

Human programmers organise sequences of repeated steps into reusable components such as subroutines, functions, and classes. They then repeatedly invoke these components — typically with different inputs. Reuse eliminates the need to "reinvent the wheel" every time a particular sequence of steps is needed. Reuse makes it possible to exploit a problem's modularities, symmetries, and regularities (and thereby potentially accelerate the problem-solving process). This can be taken further, as programmers typically organise these components into hierarchies in which top level components call lower level ones, which call still lower levels, etc.

While several different mechanisms for evolving reusable components have been proposed (e.g., [13, 359]), Koza's *Automatically Defined Functions* (ADFs) [204] have been the most successful way of evolving reusable components.

When ADFs are used, a program consists of one (or more) function-defining trees (i.e., ADFs) as well as one or more main result-producing trees (see Figure 3). An ADF may have none, one, or more inputs. The body of an ADF contains its work-performing steps. Each ADF belongs to a particular program in the population. An ADF may be called by the program's main result-producing tree, by another ADF, or by another type of tree (such as the other types of automatically evolved program components described below). Recursion is sometimes allowed. Typically, the ADFs are called with different inputs. The work-performing steps of the program's main result-producing tree and the work-performing steps of each ADF are automatically and simultaneously created by GP. The program's main result-producing tree and its ADFs typically have different function