John R. Koza

This chapter provides an introduction to genetic algorithms, the LISP programming language, genetic programming, and automatic function definition. This chapter also outlines additional sources of information about genetic algorithms and genetic programming.

2.1 Introduction to Genetic Algorithms

John Holland's pioneering book *Adaptation in Natural and Artificial Systems* [1975, 1992] showed how the evolutionary process can be applied to solve a wide variety of problems using a highly parallel technique that is now called the *genetic algorithm.*

The *genetic algorithm* transforms a *population* of individual objects, each with an associated *fitness* value, into a new *generation* of the population using the Darwinian principle of reproduction and survival of the fittest and naturally occurring genetic operations such as *crossover* (*recombination*) and *mutation*. Each *individual* in the population represents a possible solution to a given problem. The genetic algorithm attempts to find a very good or best solution to the problem by genetically breeding the population of individuals.

In preparing to use the conventional genetic algorithm operating on fixed-length character strings to solve a problem, the user must

1. determine the representation scheme,
2. determine the fitness measure,
3. determine the parameters and variables for controlling the algorithm, and
4. determine a way of designating the result and a criterion for terminating a run.

In the conventional genetic algorithm, the individuals in the population are usually fixed-length character strings patterned after chromosome strings. Thus, specification of the *representation scheme* in the conventional genetic algorithm starts with a selection of the string length L and the alphabet size K. Often the alphabet is binary, so K equals 2. The most important part of the representation scheme is the mapping that expresses each possible point in the search space of the problem as a fixed-length character string (i.e., as a *chromosome*) and each chromosome as a point in the search space of the problem. Selecting a representation scheme that facilitates solution of the problem by the genetic algorithm often requires considerable insight into the problem and good judgment.

The evolutionary process is driven by the *fitness measure*. The fitness measure assigns a fitness value to each possible fixed-length character string in the population.

The primary parameters for controlling the genetic algorithm are the population size, M, and the maximum number of generations to be run, G. Populations can consist of hundreds, thousands, tens of thousands or more individuals. There can be dozens, hundreds, thousands, or more generations in a run of the genetic algorithm.

Each run of the genetic algorithm requires specification of a *termination criterion* for deciding when to terminate a run and a method of *result designation*. One frequently used method of result designation for a run of the genetic algorithm is to designate the best individual obtained in any generation of the population during the run (i.e., the *best-so-far individual*) as the result of the run.

Once the four preparatory steps for setting up the genetic algorithm have been completed, the genetic algorithm can be run.

The three steps in executing the genetic algorithm operating on fixed-length character strings are as follows:

I. Randomly create an initial population of individual fixed-length character strings.
II. Iteratively perform the following substeps on the population of strings until the termination criterion has been satisfied:
    A. Assign a fitness value to each individual in the population using the fitness measure.
    B. Create a new population of strings by applying the following three genetic operations. The genetic operations are applied to individual string(s) in the population chosen with a probability based on fitness.
        1 Reproduce an existing individual string by copying it into the new population.
        2 Create two new strings from two existing strings by genetically recombining substrings using the crossover operation (described below) at a randomly chosen crossover point.
        3 Create a new string from an existing string by randomly mutating the character at one randomly chosen position in the string.
III. The string that is identified by the method of result designation (e.g., the best-so-far individual) is designated as the result of the genetic algorithm for the run. This result may represent a solution (or an approximate solution) to the problem.

The genetic algorithm involves probabilistic steps for at least three points in the algorithm, namely

• creating the initial population,

• selecting individuals from the population on which to perform each genetic operation (e.g., reproduction, crossover), and

• choosing a point (i.e., the crossover point or the mutation point) within the selected individual at which to perform the genetic operation.

Moreover, there is often additional randomness involved in the genetic algorithm. For example, the value of fitness may be measured for randomly chosen *fitness cases*.

As a result of the probabilistic nature of the genetic algorithm, it may be necessary to make multiple independent runs of the algorithm in order to obtain a satisfactory result for a given problem. Thus, the above three steps are embedded in an outer loop representing the runs.

Figure 2.1 is a flowchart of these steps for the conventional genetic algorithm. The variable RUN refers to the current run number. The variable GEN refers to the current generation number. The index i refers to the current individual in a population.
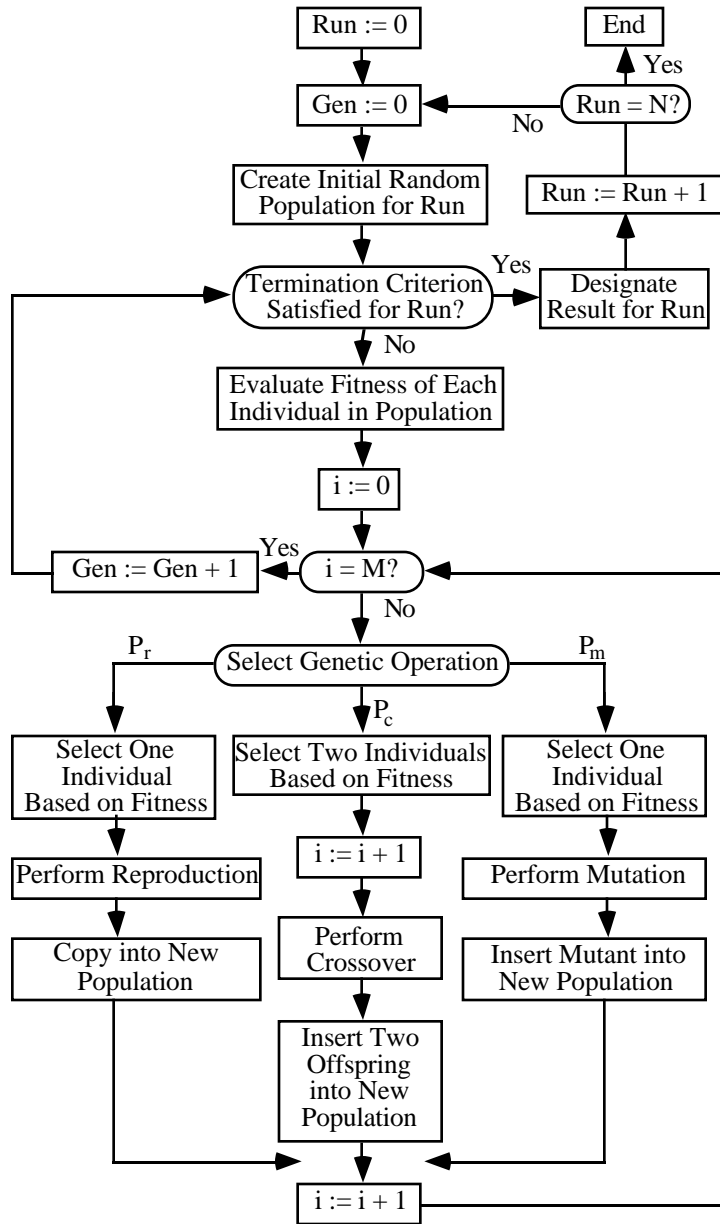
The genetic operation of *reproduction* is based on the Darwinian principle of reproduction and survival of the fittest. In the reproduction operation, an individual is probabilistically selected from the population based on its fitness (with reselection allowed) and then the individual is copied, without change, into the next generation of the population. The selection is done in such a way that the better an individual's fitness, the more likely it is to be selected. An important aspect of this probabilistic selection is that every individual, however poor its fitness, has some probability of selection.

The genetic operation of *crossover* (sexual *recombination*) allows new individuals (i.e., new points in the search space) to be created and tested. The operation of crossover starts with two parents independently selected probabilistically from the population based on their fitness (with reselection allowed). As before, the selection is done in such a way that the better an individual's fitness, the more likely it is to be selected. The crossover operation produces two offspring. Each offspring contains some genetic material from each of its parents.

The table below illustrates the crossover operation being applied to the two parental strings 10110 and 01101 of length $L = 5$ over an alphabet of size $K = 2$.

Parent 1	Parent 2
10110	01101

The crossover operation begins by randomly selecting a number between 1 and $L–1$ using a uniform probability distribution. There are $L–1 = 4$ interstitial locations lying between the positions of a string of length five. Suppose that the third interstitial location is selected. This location becomes the *crossover point*. Each parent is then split at this crossover point into a crossover fragment and a remainder. The table below shows the *crossover fragments* of parents 1 and 2.

```
                                                              ┌───────┐
                    ┌─────────┐                               │  End  │
                    │ Run := 0│                               └───────┘
                    └─────────┘                                   ▲ Yes
                         │                                        │
                         ▼                   No        ┌──────────────┐
                   ┌─────────┐  ◄─────────────────────│  Run = N?    │
                   │ Gen := 0│                         └──────────────┘
                   └─────────┘                                ▲
                         │                           ┌─────────────────┐
                         ▼                           │ Run := Run + 1  │
            ┌──────────────────────┐                 └─────────────────┘
            │ Create Initial Random│                        ▲
            │  Population for Run   │                        │
            └──────────────────────┘                ┌─────────────────┐
                         │                    Yes    │   Designate     │
                         ▼                ─────────► │ Result for Run  │
            ╭──────────────────────╮                 └─────────────────┘
            │ Termination Criterion │
            │  Satisfied for Run?   │
            ╰──────────────────────╯
                         │ No
                         ▼
            ┌──────────────────────┐
            │ Evaluate Fitness of Each│
            │ Individual in Population│
            └──────────────────────┘
                         │
                         ▼
                     ┌───────┐
                     │ i := 0│
                     └───────┘
                         │
         Yes             ▼
   ┌──────────────┐  ╭─────────╮
   │ Gen := Gen + 1│◄─│  i = M? │◄──────────────────┐
   └──────────────┘  ╰─────────╯                    │
                         │ No                        │
                         ▼                           │
      Pr       ╭──────────────────────╮      Pm      │
   ┌──────────│ Select Genetic Operation│──────────┐ │
   │           ╰──────────────────────╯           │ │
   │                   │ Pc                        │ │
   ▼                   ▼                           ▼ │
┌──────────┐  ┌──────────────────────┐  ┌──────────┐ │
│Select One│  │Select Two Individuals│  │Select One│ │
│Individual│  │   Based on Fitness   │  │Individual│ │
│Based on  │  └──────────────────────┘  │Based on  │ │
│Fitness   │          │                 │Fitness   │ │
└──────────┘          ▼                 └──────────┘ │
   │            ┌────────────┐              │         │
   ▼            │ i := i + 1 │              ▼         │
┌──────────────┐└────────────┘     ┌──────────────┐  │
│Perform       │       │           │Perform       │  │
│Reproduction  │       ▼           │Mutation      │  │
└──────────────┘ ┌──────────┐      └──────────────┘  │
   │             │ Perform  │         │               │
   ▼             │Crossover │         ▼               │
┌──────────────┐ └──────────┘  ┌──────────────┐       │
│Copy into New │       │        │Insert Mutant │       │
│ Population   │       ▼        │into New      │       │
└──────────────┘ ┌──────────┐  │Population    │       │
   │             │Insert Two│  └──────────────┘       │
   │             │Offspring │         │               │
   │             │into New  │         │               │
   │             │Population │         │               │
   │             └──────────┘         │               │
   │                   │              │               │
   └──────────────────►│◄─────────────┘               │
                       ▼                              │
                 ┌────────────┐                        │
                 │ i := i + 1 │────────────────────────┘
                 └────────────┘
```

Figure 2.1
Flowchart of the conventional genetic algorithm.

Crossover fragment 1	Crossover fragment 2
101 – –	011 – –

After the crossover fragment is identified, something remains of each parent. The table below shows the *remainders* of parents 1 and 2.

Remainder 1	Remainder 2
– – – 10	– – – 01

The crossover operation combines remainder 1 (i.e., – – – 1 0) with crossover fragment 2 (i.e., 011 – –) to create offspring 2 (i.e., 01110). The crossover operation similarly combines remainder 2 (i.e., – – – 01) with crossover fragment 1 (i.e., 101 – –) to create offspring 1 (i.e., 10101). The following table shows the two offspring.

Offspring 1	Offspring 2
10101	01110

The two offspring are usually different from their two parents and different from each other.

The operation of mutation allows new individuals to be created. It begins by selecting an individual from the population based on its fitness (with reselection allowed). A point along the string is selected at random and the character at that point is randomly changed. The altered individual is then copied into the next generation of the population. Mutation is used very sparingly in genetic algorithm work.

The genetic algorithm works in a domain-independent way on the fixed-length character strings in the population. The genetic algorithm searches the space of possible character strings in an attempt to find high-fitness strings. The fitness landscape may be very rugged and nonlinear. To guide this search, the genetic algorithm uses only the numerical fitness values associated with the explicitly tested strings in the population. Regardless of the particular problem domain, the genetic algorithm carries out its search by performing the same disarmingly simple operations of copying, recombining, and occasionally randomly mutating the strings.

In practice, the genetic algorithm is surprisingly rapid in effectively searching complex, highly nonlinear, multidimensional search spaces. This is all the more surprising because the genetic algorithm does not know anything about the problem domain or the internal workings of the fitness measure being used.

2.2 Program Trees and the LISP Programming Language

All computer programs – whether they are written in FORTRAN, PASCAL, C, assembly code, or any other programming language – can be viewed as a sequence of applications of functions (operations) to arguments (values). Compilers use this fact by first internally translating a given program into a parse tree and then converting the parse tree into the more elementary assembly code instructions that actually run on the computer. However this important commonality underlying all computer programs is usually obscured by the large variety of different types of statements, operations, instructions, syntactic constructions, and grammatical restrictions found in most programming languages.

Genetic programming is most easily understood if one thinks about it in terms of a programming language that overtly and transparently views a computer program as a sequence of applications of functions to arguments.

Moreover, since genetic programming initially creates computer programs at random and then manipulates the programs by various genetically motivated operations, genetic programming may be implemented in a conceptually straightforward way in a programming language that permits a computer program to be easily manipulated as data and then permits the newly created data to be immediately executed as a program.

For these two reasons, the LISP (LISt Processing) programming language [Steele 1990] is especially well suited for genetic programming. However, it should be recognized that genetic programming does not require LISP for its implementation and is not in any way based on LISP or dependent on LISP. Indeed, the majority of the authors in this book implemented genetic programming in C, rather than LISP. Nonetheless, even when researchers do not actually use LISP for writing their programs to implement genetic programming, most (including most of the authors in this book) find it convenient to use the style of the LISP programming language for presenting and discussing the programs evolved by genetic programming.

LISP has only two main types of entities: atoms and lists. The constant 7 and the variable TIME are examples of atoms in LISP. A list in LISP is written as an ordered collection of items inside a pair of parentheses. (A B C D) and (+ 1 2) are examples of lists in LISP.

Both lists and atoms in LISP are called *symbolic expressions* (*S-expressions*). The S-expression is the only syntactic form in pure LISP. There is no syntactic distinction between programs and data in LISP. In particular, all data in LISP are S-expressions and all programs in LISP are S-expressions.

The LISP system works so as to evaluates whatever it sees. When seen by LISP, a constant atom (e.g., 7) evaluates to itself, and a variable atom (e.g., TIME) evaluates to the current value of the variable. When a list is seen by LISP, the list is evaluated by treating the first element of the list (i.e., whatever is just inside the opening parenthesis) as a function and then causing the application of that function to the results of evaluating the remaining elements of the list. That is, the remaining elements are treated as arguments to the function. If an argument is a constant atom or a variable atom, this evaluation is immediate; however, if an argument is a list, the evaluation of such an argument involves a recursive application of the above steps.

For example, in the LISP S-expression (+ 1 2), the addition function + appears just inside the opening parenthesis. The S-expression (+ 1 2) calls for the application of the addition function + to two arguments (i.e., the constant atoms 1 and 2). Since both arguments are atoms, they can be immediately evaluated to their values (i.e., 1 and 2). Thus, the value returned as a result of the evaluation of the entire S-expression (+ 1 2) is 3.

If any of the arguments in an S-expression are themselves lists (rather than constant atoms or variable atoms that can be immediately evaluated), LISP first evaluates these arguments. In Common LISP, this evaluation is done in a recursive, depth-first way, starting from the left). For example, the S-expression (+ (* 2 3) 4) calls for the application of the addition function + to two arguments, namely the sub-S-expression (* 2 3) and the constant atom 4. In order to evaluate the entire S-expression, LISP must first evaluate the sub-S-expression (* 2 3). This argument (* 2 3) calls for the application of the multiplication function * to the two constant atoms 2 and 3, so it evaluates to 6 and the entire S-expression evaluates to 10. LISP S-expressions are examples of prefix notation. FORTRAN, PASCAL, and C are similar to ordinary mathematical notation in that they use ordinary ("infix") notation, so the above program in LISP would be written as 2*3+4 in those languages.

For example, the LISP S-expression

```
(+ 1 2 (IF (> TIME 10) 3 4))
```

further illustrates how LISP views conditional and relational elements of computer programs as applications of functions to arguments. In the sub-S-expression (> TIME 10), the relation > is viewed as a function and > is then applied to the variable atom TIME and the constant atom 10. The subexpression (> TIME 10) then evaluates to either T (True) or NIL (False), depending on the current value of the variable atom TIME. The conditional operator IF is then viewed as a function and IF is then applied to three arguments: the logical value (T or NIL) returned by the subexpression (> TIME 10),
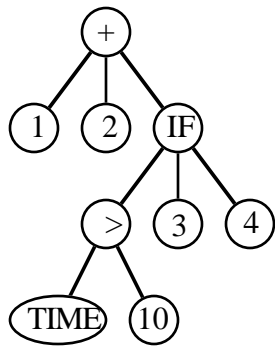
Figure 2.2
The LISP S-expression `(+ 1 2 (IF (> TIME 10) 3 4))` depicted as a rooted, point-labeled tree with ordered branches.

the constant atom `3`, and the constant atom `4`. If its first argument evaluates to `T` (more precisely, anything other than `NIL`), the function `IF` returns the result of evaluating its second argument (i.e., the constant atom `3`), but if its first argument evaluates to `NIL`, the function `IF` returns the result of evaluating its third argument (i.e., the constant atom `4`). Thus, the S-expression evaluates to either `6` or `7`, depending on whether the current value of the variable atom `TIME` is or is not greater than `10`. Most other programming languages create differing syntactic forms and statement types for functions such as `*`, `>`, and `IF` whereas LISP treats all of these functions in the same way.

Any LISP S-expression can be graphically depicted as a rooted point-labeled tree with ordered branches. Figure 2.2 shows the parse tree (program tree) corresponding to the above S-expression.

In this graphical depiction, the three internal points of the tree are labeled with functions (i.e., `+`, `IF`, and `>`). The six external points (leaves) of the tree are labeled with terminals (i.e., the variable atom `TIME` and the constant atoms `1`, `2`, `10`, `3`, and `4`). The root of the tree is labeled with the function appearing just inside the leftmost opening parenthesis of the S-expression (i.e., the `+`).

Note that this tree form of a LISP S-expression is equivalent to the parse tree that many compilers construct internally to represent a given computer program.

An important feature of LISP is that all LISP computer programs have just one syntactic form (i.e., the S-expression). The programs of the LISP programming language are S-expressions, and an S-expression is, in effect, the parse tree of the program.

2.3 Genetic Programming

Genetic programming is an extension of the conventional genetic algorithm in which each individual in the population is a computer program.

The search space in genetic programming is the space of all possible computer programs composed of functions and terminals appropriate to the problem domain. The functions may be standard arithmetic operations, standard programming operations, standard mathematical functions, logical functions, or domain-specific functions.

Genetic programming is an attempt to deal with one of the central questions in computer science, namely
How can computers learn to solve problems without being explicitly programmed? In other words, how can computers be made to do what needs to be done, without being told exactly how to do it?

The book *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [Koza 1992] demonstrated a result that many found surprising and counterintuitive, namely that an automatic, domain-independent method can genetically breed computer programs capable of solving, or approximately solving, a wide variety of problems from a wide variety of fields.

In applying genetic programming to a problem, there are five major preparatory steps. These five steps involve determining
1. the set of terminals,
2. the set of primitive functions,
3. the fitness measure,
4. the parameters for controlling the run, and
5. the method for designating a result and the criterion for terminating a run.

The first major step in preparing to use genetic programming is to identify the set of terminals. The terminals can be viewed as the inputs to the as-yet-undiscovered computer program. The set of terminals (along with the set of functions) are the ingredients from which genetic programming attempts to construct a computer program to solve, or approximately solve, the problem.

The second major step in preparing to use genetic programming is to identify the set of functions that are to be used to generate the mathematical expression that attempts to fit the given finite sample of data.

Each computer program (i.e., mathematical expression, LISP S-expression, parse tree) is a composition of functions from the function set $\mathcal{F}$ and terminals from the terminal set $\mathcal{T}$.

Each of the functions in the function set should be able to accept, as its arguments, any value and data type that may possibly be returned by any function in the function set and any value and data type that may possibly be assumed by any terminal in the terminal set. That is, the function set and terminal set selected should have the closure property.

These first two major steps correspond to the step of specifying the representation scheme for the conventional genetic algorithm. The remaining three major steps for genetic programming correspond to the last three major preparatory steps for the conventional genetic algorithm.

In genetic programming, populations of hundreds, thousands, tens of thousands or more computer programs are genetically bred. This breeding is done using the Darwinian principle of survival and reproduction of the fittest along with a genetic crossover operation appropriate for mating computer programs. As will be seen in the numerous chapters of this book, a computer program that solves (or approximately solves) a given problem may emerge from this combination of Darwinian natural selection and genetic operations.

Genetic programming starts with an initial population of randomly generated computer programs composed of functions and terminals appropriate to the problem domain. The creation of this initial random population is, in effect, a blind random search of the search space of the problem represented as computer programs.

Each individual computer program in the population is measured in terms of how well it performs in the particular problem environment. This measure is called the *fitness measure*. The nature of the fitness measure varies with the problem.

For many problems, fitness is naturally measured by the error produced by the computer program. The closer this error is to zero, the better the computer program. In a problem of optimal control, the fitness of a computer program may be the amount of time (or fuel, or money, etc.) it takes to bring the system to a desired target state. The smaller the amount of time (or fuel, or money, etc.), the better. If one is trying to recognize patterns or classify examples, the fitness of a particular program may be measured by some combination of the number of instances handled correctly (i.e., true positive and true negatives) and the number of instances handled incorrectly (i.e., false positives and false negatives). On the other hand, if one is trying to find a good randomizer, the fitness of a given computer program might be measured by means of entropy, satisfaction of the gap test, satisfaction of the run test, or some combination of these factors. For some problems, it may be appropriate to use a multiobjective fitness measure incorporating a combination of factors such as correctness, parsimony, or efficiency.

Typically, each computer program in the population is run over a number of different *fitness cases* so that its fitness is measured as a sum or an average over a variety of representative different situations. These fitness cases sometimes represent a sampling of different values of an independent variable or a sampling of different initial conditions of a system. For example, the fitness of an individual computer program in the population may be measured in terms of the sum of the absolute value of the differences between the output produced by the program and the correct answer to the problem (i.e., the Minkowski distance) or the square root of the sum of the squares (i.e., Euclidean distance). These sums are taken over a sampling of different inputs (fitness cases) to the program. The fitness cases may be chosen at random or may be chosen in some structured way (e.g., at regular intervals or over a regular grid).

The computer programs in the initial generation (i.e., generation 0) of the process will generally have exceedingly poor fitness. Nonetheless, some individuals in the population will turn out to be somewhat more fit than others. These differences in performance are then exploited.

The Darwinian principle of reproduction and survival of the fittest and the genetic operation of crossover are used to create a new offspring population of individual computer programs from the current population of programs.

The reproduction operation involves selecting a computer program from the current population of programs based on fitness (i.e., the better the fitness, the more likely the individual is to be selected) and allowing it to survive by copying it into the new population.

The crossover operation is used to create new offspring computer programs from two parental programs selected based on fitness. The parental programs in genetic programming are typically of different sizes and shapes. The offspring programs are composed of subexpressions (subtrees, subprograms, subroutines, building blocks) from their parents. These offspring programs are typically of different sizes and shapes than their parents.

Intuitively, if two computer programs are somewhat effective in solving a problem, then some of their parts probably have some merit. By recombining randomly chosen parts of somewhat effective programs, we sometimes produce new computer programs that are even more fit at solving the given problem than either parent.

The mutation operation may also be used in genetic programming.

After the genetic operations are performed on the current population, the population of offspring (i.e., the new generation) replaces the old population (i.e., the old generation).

Each individual in the new population of computer programs is then measured for fitness, and the process is repeated over many generations.

At each stage of this highly parallel, locally controlled, decentralized process, the state of the process will consist only of the current population of individuals.

The force driving this process consists only of the observed fitness of the individuals in the current population in grappling with the problem environment.

As will be seen, this algorithm will produce populations of computer programs which, over many generations, tend to exhibit increasing average fitness in dealing with their environment. In addition, these populations of computer programs can rapidly and effectively adapt to changes in the environment.

The best individual appearing in any generation of a run (i.e., the best-so-far individual) is typically designated as the result produced by the run of genetic programming.

The hierarchical character of the computer programs that are produced is an important feature of genetic programming. The results of genetic programming are inherently hierarchical. In many cases the results produced by genetic programming are default hierarchies, prioritized hierarchies of tasks, or hierarchies in which one behavior subsumes or suppresses another.

The dynamic variability of the computer programs that are developed along the way to a solution is also an important feature of genetic programming. It is often difficult and unnatural to try to specify or restrict the size and shape of the eventual solution in advance. Moreover, advance specification or restriction of the size and shape of the solution to a problem narrows the window by which the system views the world and might well preclude finding the solution to the problem at all.

Another important feature of genetic programming is the absence or relatively minor role of preprocessing of inputs and postprocessing of outputs. The inputs, intermediate results, and outputs are typically expressed directly in terms of the natural terminology of the problem domain. The computer programs produced by genetic programming consist of functions that are natural for the problem domain. The postprocessing of the output of a program, if any, is done by a *wrapper* (*output interface*).

Finally, another important feature of genetic programming is that the structures undergoing adaptation in genetic programming are active. They are not passive encodings (i.e., chromosomes) of the solution to the problem. Instead, given a computer on which to run, the structures in genetic programming are active structures that are capable of being executed in their current form.

In summary, genetic programming breeds computer programs to solve problems by executing the following three steps:

1. Generate an initial population of random compositions of the functions and terminals of the problem (i.e., computer programs).

2. Iteratively perform the following substeps until the termination criterion has been satisfied:
    a. Execute each program in the population and assign it a fitness value using the fitness measure.
    b. Create a new population of computer programs by applying the following two primary operations. The operations are applied to computer program(s) in the population chosen with a probability based on fitness.
        i   Reproduce an existing program by copying it into the new population.
        ii Create two new computer programs from two existing programs by genetically recombining randomly chosen parts of two existing programs using the crossover operation (described below) applied at a randomly chosen crossover point within each program.
3. The program that is identified by the method of result designation (e.g., the best-so-far individual) is designated as the result of the genetic algorithm for the run. This result may represent a solution (or an approximate solution) to the problem.

The genetic crossover (sexual recombination) operation operates on two parental computer programs selected with a probability based on fitness and produces two new offspring programs consisting of parts of each parent.

For example, consider the following computer program (LISP S-expression):

```
(+ (* 0.234 Z) (- X 0.789)),
```

which we would ordinarily write as

$$0.234\ Z + X - 0.789.$$

This program takes two inputs (X and Z) and produces a floating point output.

Also, consider a second program:

```
(* (* Z Y) (+ Y (* 0.314 Z))),
```

which is equivalent to

$$Z\ Y\ (Y + 0.314\ Z).$$

In figure 2.3, these two programs are depicted as rooted, point-labeled trees with ordered branches. Internal points (i.e., nodes) of the tree correspond to functions (i.e., operations) and external points (i.e., leaves, endpoints) correspond to terminals (i.e., input data). The numbers beside the function and terminal points of the tree appear for reference only.

The crossover operation creates new offspring by exchanging sub-trees (i.e., sub-lists, subroutines, subprocedures) between the two parents.

$$0.234Z + X - 0.789 \qquad ZY(Y + 0.314Z)$$
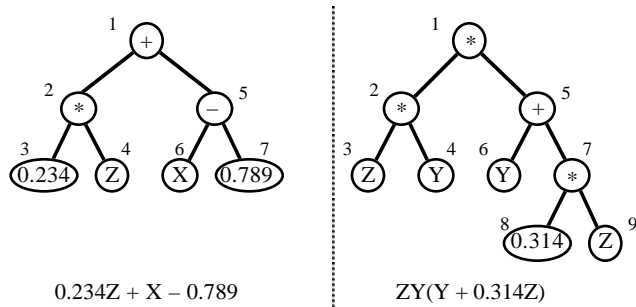
Figure 2.3
Two Parental computer programs.

$$0.234Z \qquad Y + 0.314Z$$
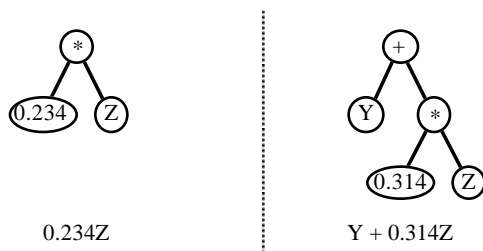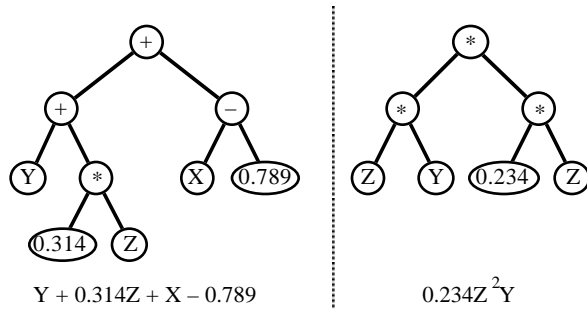
Figure 2.4
Two Crossover Fragments.

Assume that the points of both trees are numbered in a depth-first way starting at the left. Suppose that the point number 2 (out of 7 points of the first parent) is randomly selected as the crossover point for the first parent and that the point number 5 (out of 9 points of the second parent) is randomly selected as the crossover point of the second parent. The crossover points in the trees above are therefore the * in the first parent and the + in the second parent. The two crossover fragments are the two sub-trees shown in figure 2.4.

These two crossover fragments correspond to the underlined sub-programs (sub-lists) in the two parental computer programs.

The two offspring resulting from crossover are

```
(+ (+ Y (* 0.314 Z)) (- X 0.789))
```

and

$$Y + 0.314Z + X - 0.789$$
$$0.234Z^2Y$$

Figure 2.5
Two Offspring.

```
(* (* Z Y) (* 0.234 Z)).
```

The two offspring are shown in figure 2.5.

Thus, crossover creates new computer programs using parts of existing parental programs. Because entire sub-trees are swapped, the crossover operation always produces syntactically and semantically valid programs as offspring regardless of the choice of the two crossover points. Because programs are selected to participate in the crossover operation with a probability based on fitness, crossover allocates future trials to regions of the search space whose programs contains parts from promising programs.

The videotape *Genetic Programming: The Movie* [Koza and Rice 1992] provides a visualization of the genetic programming process and of solutions to various problems.

2.4 Automatic Function Definition in Genetic Programming

Several of the papers in this book deal with the issue of scaling up genetic programming to more difficult problems by means of the automatic discovery of functional subunits. This section discusses one such approach, namely automatically defined functions (ADFs).

A human programmer writing a program containing a common calculation involving the exponential of two numbers would probably write a subroutine (defined function, subprogram, procedure) for the common calculation and then call the subroutine twice from the main program. The six lines of code below show a one-line main LISP program for finding the difference $e^3 - e^2$ and a three-line defined function called `exp-approx` for calculating an approximation to the value of the exponential function:

```
1.  ;;;---main program---
2.  (values (- (exp-approx 3.0) (exp-approx 2.0)))
3.  ;;;---definition of the function "exp-approx"---
4.  (defun exp-approx (arg0)
5.    (values (+ 1.0  arg0  (* 0.5 arg0 arg0)
6.                 (* 0.1667 arg0 arg0 arg0))))
```

Lines 1 and 3 contain comments to identify the main program in line 2 and the definition of the function in lines 4, 5, and 6.

Line 2 contains the main program that calls the function exp-approx twice and then computes the difference (–) between the two. The single value to be returned by the main program in line 2 is highlighted with an explicit invocation of the values function (which ordinarily would not be used).

Lines 4, 5 and 6 contain the definition of the function exp-approx. This function definition (called a defun in LISP) does four things.

First, this defun (line 4) assigns a name, exp-approx, to the function being defined. The name permits subsequent reference to this function by the main (calling) program (line 2).

Second, this defun identifies the argument list (line 4) of the function being defined. In this defun, the argument list is (arg0) containing one dummy variable (formal parameter) arg0.

Third, this defun contains a body (lines 5 and 6) that performs the work of the function. In this defun, the work consists of the addition of the first three terms of the Taylor series approximation for e^x using the four-argument primitive function of addition (+) and the three- and four-argument primitive function of multiplication (*). Note that the body of the function uses the dummy variable arg0 that is localized within the defun. This dummy variable does not appear in the main program.

Fourth, this defun identifies the value to be returned by the function. In this defun, the single value to be returned is emphasized and highlighted with an explicit invocation of the values function (line 5).

This particular illustrative defun has one local dummy variable, returns only a single value, has no side effects, and refers only to its one local dummy variable (i.e., it does not refer to any of the actual or "global" variables of the overall problem). However, in general, defined functions may have any number of arguments (including no arguments), may return multiple values (or no values), may or may not perform side effects, and may or may not explicitly refer to the actual (global) variables of the overall problem.

Automatic function definition can be implemented within the context of genetic programming by establishing a constrained syntactic structure for the individual programs
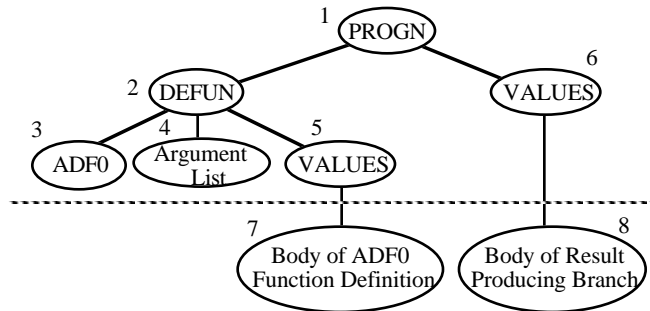
Figure 2.6
An overall computer program consisting of one function-defining branch and one result-producing branch.

in the population as described in [Koza 1992]. Each program in the population contains one (or more) function-defining branches and one (or more) main result-producing branches. The result-producing branch usually has the ability to call one or more of the defined functions. A function-defining branch may have the ability to refer hierarchically to another already-defined function (and potentially even itself). When the result-producing branch returns only a single value, we sometimes refer to it as the *value-returning branch.*

Figure 2.6 shows the overall structure of a program consisting of one function-defining branch and one main result-producing branch. The function-defining branch appears in the left part of this figure and the result-producing branch appears on the right.

There are eight different "types" of points in this overall program. The first six types are invariant and appear above the horizontal dotted line in this figure. The eight types are as follows:

1. the root of the tree (which consists of the place-holding PROGN connective function),
2. the top point, DEFUN, of the function-defining branch,
3. the name, ADF0, of the automatically defined function,
4. the argument list of the automatically defined function,
5. the VALUES function of the function-defining branch identifying, for emphasis, the value(s) to be returned by the automatically defined function,
6. the VALUES function of the result-producing branch identifying, for emphasis, the value(s) to be returned by the result-producing branch,
7. the body (i.e., work) of the automatically defined function ADF0, and
8. the body of the result-producing branch.

When the overall program is evaluated, the PROGN causes the sequential evaluation of the two branches. The PROGN starts by evaluating the first branch, namely the function-defining branch. The function-defining branch merely defines the automatically defined function ADF0. When a defun is evaluated by the PROGN, the result returned (which happens in LISP to be the name of the function being defined) is irrelevant. Since the PROGN returns only the result of the evaluation of its last argument, the evaluation of the function-defining branch does not cause the return of any value for the PROGN as a whole. The PROGN now evaluates its second branch, namely the result-producing branch. The body of the result-producing branch may refer to the automatically defined function ADF0. The value returned by the overall program consists only of the value returned by the VALUES function associated with the result-producing branch. Note that in this formulation all references to an automatically defined function are to the automatically defined function within the same overall program (and not to the automatically defined functions of other programs in the population).

Genetic programming will evolve a population of programs, each consisting of a function definition in the function-defining branch and a result-producing branch. The structures of both the function-defining branches and the result-producing branch are determined by the combined effect, over many generations, of the selective pressure exerted by the fitness measure and by the effects of the operations of Darwinian fitness proportionate reproduction and crossover. The function defined by the function-defining branch is available for use by the result-producing branch. Whether or not the defined function will be actually called is not predetermined, but instead, determined by the evolutionary process.

Since each individual program in the population of this example consists of one function-defining branch and one result-producing branch, the initial random generation must be created so that every individual program in the population has this particular constrained syntactic structure. Specifically, every individual program in generation 0 must have the invariant structure represented by the six points of types 1 through 6 described above. Each function and terminal in the function-defining branch is of type 7. The function-defining branch is a random composition of functions from the function set $\mathcal{F}_{\mathrm{fd}}$ for the function-defining branch and terminals from the terminal set $\mathcal{T}_{\mathrm{fd}}$ for the function-defining branch. The terminal set $\mathcal{T}_{\mathrm{fd}}$ for the function-defining branch typically contains the localized dummy variables (e.g., ARG0). Each function and terminal in the result-producing branch is of type 8. The result-producing branch is a random composition of functions from the function set $\mathcal{F}_{\mathrm{rp}}$ and terminals from the terminal set $\mathcal{T}_{\mathrm{rp}}$. The function set for the result-producing branch typically contains the available defined functions (e.g., ADF0) and does not contain the dummy variables of the defined

function (e.g., `ARG0`). The result-producing branch typically contains the actual variables of the problem. The actual variables of the problem usually do not appear in the function-defining branch, although they may.

Since a constrained syntactic structure is involved, crossover must be performed so as to preserve this syntactic structure in all offspring. Since each program must have the invariant structure represented by the six points of types 1 through 6, crossover is limited to points of types 7 and 8. Structure-preserving crossover is implemented by allowing any point of type 7 or 8 to be the crossover point in the first parent. However, once the crossover point in the first parent has been selected, the crossover point of the second parent must be of the same type (i.e. type 7 or 8). In the context of this example, this means that crossover will only exchange a sub-tree from the function-defining branch of one parent with a sub-tree from the function-defining branch of the other parent or that crossover will exchange a sub-tree from the result-producing branch of one parent with a sub-tree from the result-producing branch of the other parent. This restriction on the selection of the crossover point of the second parent ensures the preservation of the constrained syntactic structure (originally created in generation 0) for all offspring in all subsequent generations.

Automatic function definition is the main subject of the forthcoming book *Genetic Programming II* [Koza 1994]. A visualization of automatic function definition and numerous example problems can be found in the forthcoming videotape *Genetic Programming II Videotape: The Next Generation* [Koza and Rice 1994]. There is one example of automatic function definition in *Genetic Programming: The Movie* [Koza and Rice 1992].

Angeline and Pollack [1993a, 1993b] have developed a "module acquisition" strategy which is discussed elsewhere in this book.

2.5   Sources of Additional Information about Genetic Programming

An electronic mail mailing list on genetic programming has been established and is currently maintained by James P. Rice of the Knowledge Systems Laboratory of Stanford University. You may subscribe to this on-line mailing list at no charge by sending a subscription request consisting of the message `subscribe  genetic-programming` to `genetic-programming-request@cs.stanford.edu` by electronic mail.

An on-line public repository and FTP (file transfer protocol) site containing computer code, papers on genetic programming, and frequently asked questions (FAQs) has been established and is currently maintained by James McCoy of the Computation Center at

the University of Texas at Austin. This FTP site may be accessed by electronic mail by anonymous FTP from the `pub/genetic-programming` directory from the site `ftp.cc.utexas.edu`. This FTP site contains the original "Little LISP" computer code written in Common LISP for genetic programming as described in *Genetic Programming: On the Programming of Computers by Means of Natural Selection* [Koza 1992], the SGPC ("Simple Genetic Programming in C") computer code written in the C programming language by Walter Alden Tackett and Aviram Carmi of Hughes Aircraft Corporation for genetic programming, and computer code appearing in *Genetic Programming II* [Koza 1994] in Common LISP for implementing automatic function definition and constrained syntactic structures within genetic programming.

The proceedings of the Fifth International Conference on Genetic Algorithms contains recent work on genetic programming by Angeline and Pollack [1993a, 1993b], Banzhaf [1993], Gruau [1993], Handley [1993], Iba et. al [1993], Kinnear [1993b], Koza [1993], Spencer [1993], and Tackett [1993]. The proceedings of the Second International Conference on Simulation of Adaptive Behavior contains work by Reynolds [1993]. The proceedings of the *1993 IEEE International Conference on Neural Networks* contains work by Kinnear [1993a]. The proceedings of the Third Workshop on Artificial Life contains work by Reynolds [1994].

In addition, the proceedings of the IEEE World Congress on Computational Intelligence to be held in Florida in June 1994 will contain additional papers on genetic programming.

2.6 Sources of Additional Information about Genetic Algorithms

Additional information on genetic algorithms can be found in Goldberg [1989], Davis [1987, 1991], Michalewicz [1992], Maenner and Manderick [1992], and Buckles and Petry [1992]. Conference proceedings include Grefenstette [1985, 1987], Schaffer [1989], Belew and Booker [1991], Forrest [1993], Rawlins [1991], and Whitley [1992]. Stender [1993] describes parallelization of genetic algorithms. Davidor [1992] describes application of genetic algorithms to robotics. Schaffer and Whitley [1992] and Albrecht, Reeves, and Steele [1993] describe work on combinations of genetic algorithms and neural networks. Forrest [1991] describes application of genetic classifier systems to semantic nets. Schwefel and Maenner [1991] and Maenner and Manderick [1992] contain recent work on evolutionary strategies (ES). Fogel and Atmar [1992, 1993] contain recent work on evolutionary programming (EP).

Additional information about genetic algorithms may be obtained from the GA-LIST electronic mailing list to which you may subscribe at no charge by sending a subscription request to `GA-List-Request@AIC.NRL.NAVY.MIL`. Additional information about evolutionary programming may be obtained from the EP-LIST electronic mailing list to

which you may subscribe at no charge by sending a subscription request to `EP-List-Request@Magenta.Me.Fau.Edu`.

Bibliography

Albrecht, R. F. (1993), C. R. Reeves, and N. C. Steele, *Artificial Neural Nets and Genetic Algorithms*. Springer-Verlag.

Angeline, P. J. (1993a) and J. B. Pollack. *Coevolving high-level representations*. Technical report 92-PA-COEVOLVE. Laboratory for Artificial Intelligence. Ohio State University. July 1993.

Angeline, Peter J. and Pollack, Jordan B. Coevolving high-level representations. In C. G. Langton, Ed. *Artificial Life III*. In Press.

Angeline, P. J. (1993b) and J. B. Pollack, Competitive environments evolve better solutions for complex tasks. In S. Forrest, Ed. *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. Pages 264-270.

Banzhaf, W (1993) Genetic programming for pedestrians. In S. Forrest, Ed. *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc.. Page 628.

Belew, R. (1991) and L. Booker, Eds., *Proceedings of the Fourth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann.

Buckles, B. P. (1992) and F. E. Petry, *Genetic Algorithms*. Los Alamitos, CA: The IEEE Computer Society Press.

Davidor, Y. (1991) *Genetic Algorithms and Robotics*. Singapore: World Scientific.

Davis, L., Ed. (1987) *Genetic Algorithms and Simulated Annealing*. London: Pittman.

Davis, L. (1991), *Handbook of Genetic Algorithms*. New York: Van Nostrand Reinhold.

Fogel, D. B. (1992) and W. Atmar, Eds., *Proceedings of the First Annual Conference on Evolutionary Programming*. San Diego, CA: Evolutionary Programming Society .

Fogel, D. B. (1993) and W. Atmar, Eds., *Proceedings of the Second Annual Conference on Evolutionary Programming*. San Diego, CA: Evolutionary Programming Society.

Forrest, S. (1991) *Parallelism and Programming in Classifier Systems*. London: Pittman.

Forrest, S. (1993) *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc.

Goldberg, D. E. (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.

Grefenstette, J. J., Ed. (1985) *Proceedings of an International Conference on Genetic Algorithms and Their Applications*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Grefenstette, J. J., Ed. (1987) *Genetic Algorithms and Their Applications: Proceedings of the Second International Conference on Genetic Algorithms*. Hillsdale, NJ: Lawrence Erlbaum Associates.

Gruau, F. (1993) Genetic synthesis of modular neural networks. In S. Forrest, Ed. *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. Pages 318–325.

Handley, S. (1993) Automated learning of a detector for a-helices in protein sequences via genetic programming. In S. Forrest, Ed. *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. Pages 271-278.

Holland, J. H. (1975) *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. Ann Arbor, MI: University of Michigan Press 1975. Also Cambridge, MA: The MIT Press 1992.

Iba, H. (1993), T. Jurita, H. de Garis, and T. Sat, System identification using structured genetic algorithms. In S. Forrest, Ed. *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. Pages 279-286.

Kinnear, K. E. Jr. (1993a) Evolving a sort: Lessons in genetic programming. *1993 IEEE International Conference on Neural Networks, San Francisco.* Piscataway, NJ: IEEE 1993. Volume 2. Pages 881-888.

Kinnear, K. E. Jr. (1993b) Generality and difficulty in genetic programming: Evolving a sort. In S. Forrest. Ed. *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. Pages 287–294.

Koza, J. R. (1992) *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA: The MIT Press.

Koza, J. R. (1993) Simultaneous discovery of reusable detectors and subroutines using genetic programming. In S. Forrest, Ed. *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. Pages 295–302.

Koza, J. R. (1994) *Genetic Programming II.* Cambridge, MA: The MIT Press. In Press.

Koza, J. R.(1992), and J. P. Rice *Genetic Programming: The Movie*. Cambridge, MA: The MIT Press.

Koza, J. R. (1994), and J. P. Rice *Genetic Programming II Videotape: The Next Generation* Cambridge, MA: The MIT Press. In Press.

Maenner, R. (1992), and B. Manderick, *Proceedings of the Second International Conference on Parallel Problem Solving from Nature.* North Holland.

Michalewicz, Z. (1992). *Genetic Algorithms + Data Structures = Evolution Programs*. Berlin: Springer-Verlag.

Rawlins, G., Ed. (1991) *Foundations of Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc.

Reynolds, C. W. (1993) An evolved vision-based behavioral model of coordinated group motion. In Meyer, Jean-Arcady, Roitblat, Herbert L. and S. Wilson, Ed. *From Animals to Animats 2: Proceedings of the Second International Conference on Simulation of Adaptive Behavior*. Cambridge, MA: The MIT Press. Pages 384-392..

Reynolds, C. W. (1994) An evolved vision-based model of obstacle avoidance behavior. In C. G. Langton, Ed. *Artificial Life III*. In Press.

Schaffer, J. D., Ed. (1989). *Proceedings of the Third International Conference on Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann Publishers Inc.

Schaffer, J. D. (1992) and D. Whitley, Eds., *Proceedings of the Workshop on Combinations of Genetic Algorithms and Neural Networks 1992.* Los Alamitos, CA: The IEEE Computer Society Press.

Schwefel, H.-P. (1991) and R. Maenner, Eds., *Parallel Problem Solving from Nature.* Berlin: Springer-Verlag.

Spencer, G. (1993) Automatic generation of programs for crawling and walking. In S. Forrest, Ed. *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. Page 654.

Steele, Guy L. Jr. (1990) *Common LISP*. Digital Press. Second Edition.

Stender, J., Ed. *Parallel Genetic Algorithms.* IOS Publishing.

Tackett, W. A. (1993) Genetic programming for feature discovery and image discrimination. In S. Forrest, Ed. *Proceedings of the Fifth International Conference on Genetic Algorithms*. San Mateo, CA: Morgan Kaufmann Publishers Inc. Pages 303–309.

Whitley, D. (editor) (1992) *Foundations of Genetic Algorithms and Classifier Systems 2*, Vail, Colorado 1992. San Mateo, CA: Morgan Kaufmann Publishers Inc.