*Evolutionary Computation*, **5**, no. 2, pp. 143-180.

# Evolutionary program induction directed by logic grammars

Man Leung Wong
Department of Computing and Decision Sciences
Lingnan University, Tuen Mun
Hong Kong
mlwong@ln.edu.hk

Kwong Sak Leung
Department of Computer Science and Engineering
The Chinese University of Hong Kong
Hong Kong
ksleung@cs.cuhk.edu.hk

## Abstract

Program induction generates a computer program that can produce the desired behavior for a given set of situations. Two of the approaches in program induction are inductive logic programming (ILP) and genetic programming (GP). Since their formalisms are so different, these two approaches cannot be integrated easily though they share many common goals and functionalities. A unification will greatly enhance their problem solving power. Moreover, they are restricted in the computer languages in which programs can be induced. In this paper, we present a flexible system called LOGENPRO (LOgic grammar based GENetic PROgramming) that uses some of the techniques of GP and ILP. It is based on a formalism of logic grammars. The system applies logic grammars to control the evolution of programs in various programming languages and represent context-sensitive information and domain-dependent knowledge. Experiments have been performed to demonstrate that LOGENPRO can emulate GP and GP with ADF (Automatically Defined Functions). Moreover, LOGENPRO can employ knowledge such as argument types in a unified framework. The experiments show that LOGENPRO has superior performance to that of GP and GP with ADF when more domain-dependent knowledge is available. We have applied LOGENPRO to evolve general recursive functions for the even-n-parity problem from noisy training examples. A number of experiments have been performed to determine the impact of domain-specific knowledge and noise in training examples on the speed of learning.

**Area**: Machine Learning, Genetic Programming, Logic grammars

# 1. Introduction

Researchers in the field of automatic programming investigate how to automate the tasks of the software life cycle. In this paper, we present a flexible system that evolves programs automatically and overcomes some of the problems of existing automatic programming systems which are briefly described as follows.

The users of automatic programming systems are only required to write specifications for what they want and the systems systematically generates programs satisfying these specifications. Deductive, transformational, inspection, and inductive methods can be employed in these systems (Rich and Waters, 1988).

If the specification can be formulated as a theorem stating the relation between the inputs and the corresponding outputs, then a program can be obtained by finding a constructive proof of the satisfiability of the specification. Deductive methods search for an inference path from some initial states to a goal representing the specification. Since the search space is extremely large and the current deductive systems cannot control the search process effectively, these systems cannot discover complex programs.

Transformational methods search for a sequence of transformations to convert a specification into a low-level implementation. The three components of a transformation are a pattern, a set of logical applicability conditions, and an action. When an instance of the pattern is found in the specification, the conditions are checked to determine whether the transformations can be employed. If the conditions are satisfied, the action is evaluated to compute a new section of code, which is used to replace the code matched by the pattern. At each step, a transformation is selected and applied to a specification to produce a modified specification. The above process is repeated until some condition is satisfied. However, transformational systems suffer from the same problem of deductive systems.

Inspection methods construct a program by recognizing clichés in the specification and then choosing among various implementations of the identified cliché. The process of identifying clichés and selecting an implementation can be viewed as a difficult search problem. Consequently, a fully automatic, inspection system suffers from the same control problem in deductive and transformational systems (Rich and Waters, 1990).

Inductive methods perform inductive inference which generalizes partial specifications, such as training examples, to produce programs exhibiting the desired behavior. Program induction systems can be classified by the kinds of information employed in the specifications (Olsson, 1995). Some systems use traces of computation or sets of positive and negative examples. Biermann (1972) demonstrated that flowcharts and Turing machines can be induced from example traces. Summers (1977), Biermann and Smith (1979) described systems that create programs in Lisp from examples of their behaviors. Inductive logic programming (ILP) systems construct logic programs from examples and background knowledge (Muggleton, 1992, 1994; Quinlan, 1990, 1991; Lavrac and Dzeroski, 1994; De Raedt, 1992; De Raedt and Bruynooghe, 1992; Muggleton and De Raedt, 1994). They are more powerful than traditional learning systems because they use an expressive first-order logic framework and background knowledge. On the other hand, genetic programming (GP) systems use specifications represented as fitness functions to drive

the evolution of programs in Lisp (Cramer, 1985; Koza, 1992, 1994; Koza et al., 1996; Kinnear 1994; Angeline and Kinnear, 1996).

Since the formalisms of ILP and GP are so different, these two approaches cannot be integrated easily although their properties and goals are similar. If they can be combined in a common framework, then many of the techniques and theories obtained in one approach can be applied in the other one. The combination can greatly enhance the information exchange between these fields. Moreover, they are restricted in the computer languages in which programs can be induced. ILP systems can only learn logic programs. On the other hand, programs evolved by GP systems are usually expressed in Lisp.

This paper presents a flexible system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) that employs some of the techniques of GP and ILP to learn programs in various programming languages. The system is also powerful enough to represent context-sensitive information and domain-dependent knowledge. This knowledge can be used to accelerate the learning speed and/or improve the quality of the programs induced (Wong and Leung, 1995a, 1995b, 1995c, 1996).

The details of LOGENPRO are presented in the next section. In section three, we illustrate the application of LOGENPRO in learning S-expressions in Lisp. We demonstrate the application of various knowledge to accelerate the learning of sub-functions in the forth section. In section five, we apply LOGENPRO to learn recursive functions from noisy training examples. The last section is the conclusion.

## 2. LOGENPRO (The LOgic grammar based GENetic PROgramming system)

Since LOGENPRO can induce programs in various programming languages such as Lisp and Prolog, it must be able to accept grammars of different languages to generate programs in them. Most modern programming languages are specified in the notation of BNF (Backus-Naur form) which is a kind of context-free grammars (CFGs). However, LOGENPRO is based on logic grammars because CFGs (Hopcroft and Ullman, 1979; Lewis and Rapadimitrion, 1981) are not expressive enough to represent context-sensitive information for some languages and domain-dependent knowledge of the target program being induced. The idea of using formal grammars to direct search for programs in the hypothesis space or to reduce the size of the space has also been independently studied by other researcher recently (Cohen, 1992; Gruau, 1996; Whigham, 1995, 1996). This section first introduces the formalism of logic grammars followed by the descriptions of LOGENPRO.

### 2.1. Introduction to logic grammars

Logic grammars are the generalizations of CFGs. They are more expressive than CFGs, but equally amenable to efficient execution. In this paper, logic grammars are described in a notation similar to that of definite clause grammars (Pereira and Warren, 1980; Pereira and Shieber, 1987; Sterling and Shapiro 1986). The logic grammar for some simple S-expressions in table 1 will be used throughout this section.

A logic grammar differs from a CFG in that the logic grammar symbols, whether terminal or non-terminal, may include arguments. The arguments can be any term in the grammar. A term is either a logic variable, a function, or a constant. A variable is represented by a question mark ? followed by a string of letters and/or digits. A function is a grammar symbol followed by a bracketed n-tuple of terms and a constant is simply a 0-arity function. Arguments can be used in a logic grammar to enforce context-dependency and to represent the semantics of the program.

The terminal symbols enclosed in square brackets correspond to the set of words of the language specified. For example, the terminal `[(- ?x ?y)]` creates the constituent `(- 1.0 2.0)` of a program if `?x` and `?y` are instantiated respectively to 1.0 and 2.0. Non-terminal symbols are similar to literals in Prolog, `exp-1(?x)` in table 1 is an example of non-terminal symbols. Commas denote concatenation and each grammar rule ends with a full stop.

The right-hand side of a grammar rule may contain logic goals and grammar symbols. The goals are pure logical predicates for which logical definitions have been given. They specify the conditions that must be satisfied before the rule can be applied. For example, the goal `member(?x, [W, Z])` in table 1 instantiates the variable `?x` to either `W` or `Z` if `?x` has not been instantiated, otherwise it checks whether the value of `?x` is either `W` or `Z`. If the variable `?y` has not been bound, the goal `random(1, 2, ?y)` instantiates `?y` to a random floating point number between 1 and 2. Otherwise, the goal checks whether the value of `?y` is between 1 and 2.

Domain-dependent knowledge can be represented in logic goals. For example, consider the following grammar rule:

```
a-useful-program    ->   first-component(?X),
                         {is-useful(?X, ?Y)},
                         second-component(?Y).
```

This rule states that a useful program is composed of two components. The first component is generated from the non-terminal `first-component(?X)`. The logic variable `?X` is used to store semantic information about the first component produced. The logic goal then determines whether the first component is useful according to the semantic information stored in `?X`. Domain-dependent knowledge about which program fragments are useful is represented in the logical definition of this predicate. If the first component is useful, the logic goal `is-useful(?X, ?Y)` is satisfied and some semantic information is stored into the logic variable `?Y`. This information will be used in the non-terminal `second-component(?Y)` to guide the search for a good program fragment as the second component of a useful program.

The special non-terminal `start` corresponds to a program of the language. In table 1, some grammar symbols are shown in bold-face to identify the constituents that cannot be manipulated by genetic operators. For example, the last terminal symbol **`[)]`** of the second rule is revealed in bold-face because every S-expression must be ended with a ')'. The number before each rule is a label for later discussions. It is not part of the grammar.

### 2.2. Representations of programs

Since LOGENPRO operates on a population of programs and generates a new population of offspring programs by using genetic operators, it must produce the

initial population and guarantee that only valid offspring programs will be created. Moreover, it is necessary to find a universal representation of programs, because programs in various programming languages have different syntax and semantics. One of the fundamental contributions of LOGENPRO is in the representation of programs. This representation facilitates the generation of the initial population and the operations of various genetic operators such as reproduction, mutation, and crossover. In this sub-section, we introduce the representation and describe the method of generating the initial population. The genetic operators are described in sub-sections 2.3 and 2.4.

A program can be represented as a derivation tree that shows how the program has been derived from the logic grammar. LOGENPRO applies deduction to randomly generate programs and their derivation trees in the language declared by the given grammar. These programs form the initial population. For example, the program `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))` can be generated by LOGENPRO given the logic grammar in table 1. It is derived from the following sequence of derivations:

```
start     =>   [(*] exp(W) exp(W) exp(W) [)]
            =>   [(*] [(/ W 1.5)] exp(W) exp(W) [)]
            =>   [(*] [(/ W 1.5)] [(/ W 1.5)]
                 exp(W) [)]
            =>   [(*] [(/ W 1.5)] [(/ W 1.5)]
                 [(/ W 1.5)] [)]
            =>   [(* (/ W 1.5) (/ W 1.5) (/ W 1.5))]
```

This sequence of derivations can be represented as the derivation tree depicted in figure 1. In literature, the terms derivation trees and parse trees are usually used interchangeably. However, we will use the term derivation trees to refer to the tree structures in our framework and the term parse trees to refer to those in GP. The bindings of logic variables are shown in italic font and enclosed in a pair of braces. The sub-trees enclosed in a dashed rectangular are frozen. In other words, they are generated by bold-faced grammar symbols and they cannot be modified by genetic operators.

An advantage of logic grammars is that they specify what is a legal program without any explicit reference to the process of program generation and parsing. Furthermore, a logic grammar can be translated into an efficient logic program that can generate and parse the programs in the language declared by the logic grammar (Pereira and Warren, 1980; Pereira and Shieber, 1987; Abramson and Dahl 1989). In other words, the process of program generation and parsing can be achieved by performing deduction using the translated logic program. Consequently, the program generation and analysis mechanisms of LOGENPRO can be implemented using a deduction mechanism based on the logic programs translated from the grammars. The method of implementing LOGENPRO using a Prolog like logic programming language is described in Appendix A.

Initial programs can also be induced by other learning systems such as FOIL (Quinlan, 1990, 1991) or given by the user. LOGENPRO analyzes each program and creates the corresponding derivation tree. If the language is ambiguous, more than one derivation tree can represent a given program. In this case, LOGENPRO selects one tree from the set of possible trees to represent the program. For example, the program `(* (/ W 1.5) (/ W 1.5) (/ W 1.5)`) can also be derived from the following sequence of derivations:

```
start     =>   {member(?x, [W, Z])} [(*] exp-1(?x)
               exp-1(?x) exp-1(?x) [)]
          =>   [(*] exp-1(W) exp-1(W) exp-1(W) [)]
          =>   [(*] {random(1, 2, ?y)} [(/ W ?y)]
               exp-1(W) exp-1(W) [)]
          =>   [(*] [(/ W 1.5)] exp-1(W) exp-1(W)
               [)]
          =>   [(*] [(/ W 1.5)] {random(1, 2, ?y)}
               [(/ W ?y)] exp-1(W) [)]
          =>   [(*] [(/ W 1.5)] [(/ W 1.5)]
               exp-1(W) [)]
          =>   [(*] [(/ W 1.5)] [(/ W 1.5)]
               {random(1, 2, ?y)} [(/ W ?y)] [)]
          =>   [(*] [(/ W 1.5)] [(/ W 1.5)]
               [(/ W 1.5)] [)]
          =>   [(* (/ W 1.5) (/ W 1.5) (/ W 1.5))]
```

The derivation tree of this sequence of derivations is depicted in figure 2. The `?y1`, `?y2`, and `?y3` in the figure are different instances of the logic variable `?y` appearing in the same or different rules in the grammar.

### 2.3. Crossover of programs

The crossover is a sexual operation that starts with two parental programs and the corresponding derivation trees. One program is designated as the primary parent and the other one as the secondary parent. Their derivation trees are called the primary and secondary derivation trees respectively. The following steps are used to produce an offspring program:

1. If there are sub-trees in the primary derivation tree that have not been selected previously, select randomly a sub-tree (primary sub-tree) from these sub-trees using a uniform distribution. The root of the selected sub-tree is called the primary crossover point. Otherwise, terminate the algorithm without generating any offspring.
2. Select another sub-tree (secondary sub-tree) in the secondary derivation tree under the constraint that the offspring produced must be valid according to the grammar. In Appendix B, we present the method of selecting sub-tree satisfying the constraint.
3. If a sub-tree can be found in step 2, create and return the offspring, which is obtained by deleting the primary sub-tree and then inserting the secondary sub-tree at the primary crossover point. Otherwise, go to step 1.

Consider two parental programs generated randomly from the grammar in table 1. The primary parent is `(+ (- Z 3.5) (- Z 3.8) (/ Z 1.5))` and the secondary parent is `(* (/ W 1.5) (+ (- W 11) 12) (- W 3.5))`. The corresponding derivation trees are depicted in figures 3 and 4 respectively. In the figures, the plain numbers identify the sub-trees of these derivation trees, while the underlined numbers indicate the grammar rules used in deducing the corresponding sub-trees.

For example, if the primary and secondary sub-trees are respectively 2 and 15. The valid offspring is `(* (- Z 3.5) (- Z 3.8) (/ Z 1.5))` is obtained and its derivation tree is shown in figure 5. It is interesting to find that the sub-tree 25 has a label 2. This indicates that the sub-tree is generated by the second grammar rule

rather than the third rule applied to the primary parent. The second rule must be used because the terminal symbol `[(/]` is changed to `[(*]` and only the second rule can create the terminal `[(*]`.

In another example, the primary and secondary sub-trees are 3 and 16 respectively. The valid offspring `(+ (/ Z 1.5) (- Z 3.8) (/ Z 1.5))` is produced and the derivation tree is shown in figure 6. It should be emphasized that the constituent from the secondary parent is changed from `(/ W 1.5)` to `(/ Z 1.5)` in the offspring. This must be modified because the logic variable `?x` in sub-tree 41 is instantiated to `Z` in sub-tree 39. This example demonstrates the use of logic grammars to enforce contextual-dependency between different constituents of a program.

LOGENPRO disallows the crossover between the primary sub-tree 6 and the secondary sub-tree 19. The sub-tree 19 requires the variable `?x` to be instantiated to `W`, But, `?x` must be instantiated to `Z` in the context of the primary parent. Since `W` and `Z` cannot be unified, these two sub-trees cannot be crossed over.

LOGENPRO has an efficient algorithm to check these conditions before performing any crossover. Thus, only valid offspring are produced and this operation can be achieved effectively and efficiently. The detailed algorithm for implementing the crossover operation is discussed in Appendix B.

### 2.4. Mutation of programs

The mutation operation in LOGENPRO introduces random modifications to programs in the population. A program in the population is selected as the parental program. The selection is based on various methods such as fitness proportionate and tournament selections. The following steps are used to produce an offspring program:

1. If there are sub-trees in the derivation tree of the parental program that have not been selected previously, select randomly a sub-tree from these sub-trees using a uniform distribution. The root of the selected sub-tree is called the mutation point. Otherwise, terminate the algorithm without generating any offspring.
2. Generate a new derivation tree using the deduction mechanism produced by LOGENPRO. The new derivation tree is created under the constraint that the offspring produced must be valid according to the grammar. In Appendix C, we describe the method of generating derivation tree satisfying the constraint.
3. If a new derivation-tree can be found in step 2, create and return the offspring, which is obtained by deleting the selected sub-tree and then inserting the new derivation tree at the mutation point. Otherwise, go to step 1.

For example, assume that the program being mutated is `(+ (- Z 3.5) (- Z 3.8) (/ Z 1.5))` and the corresponding derivation tree is depicted in figure 3. If the sub-tree 3, MUTATED-SUB-TREE, is selected to be modified and the root of the MUTATED-SUB-TREE is designated as the MUTATE-POINT. Then a new derivation tree, NEW-SUB-TREE, for the S-expression `(/ Z 1.9)` can be obtained from the non-terminal symbol `exp-1(Z)` using the fifth rule of the grammar. The derivation tree is shown in figure 7. A new offspring is obtained by duplicating the genetic materials of its parental derivation tree, followed by deleting the MUTATED-SUB-TREE from the duplication, and then pasting the NEW-SUB-TREE at the

MUTATE-POINT. The derivation tree of the offspring `(+ (/ Z 1.9) (- Z 3.8) (/ Z 1.5))` can be found in figure 8.

LOGENPRO has an efficient implementation of the mutation algorithm. Moreover, an inference engine has been developed for deducing derivation trees (or programs) from a given logic grammar. Thus, only valid mutations can be performed and this operation can be achieved effectively and efficiently. The mutation algorithm is given in Appendix C.

## 2.5. The evolution process of LOGENPRO

The problem of inducing programs can be reformulated as a search for a highly fit program (according to the fitness function) in the space of all possible programs in the language specified by the logic grammar (Mitchell, 1982). In LOGENPRO, populations of programs are genetically bred (Holland, 1992; Goldberg, 1989; Davis 1991) using the Darwinian principle of survival and reproduction of the fittest along with genetic operations appropriate for creating programs. LOGENPRO starts with an initial population of programs generated randomly, induced by other learning systems, or provided by the user. Logic grammars provide declarative descriptions of the valid programs that can appear in the initial population. A fitness function must be defined by the user to evaluate the fitness values of the programs. Typically, each program is run over a set of fitness cases and the fitness function estimates its fitness by performing some statistical operations (e.g. average) to the values returned by this program.

Since each program generated in the evolution process must be executed, a compiler or interpreter for the corresponding programming language must be available. This compiler or interpreter is called by the fitness function to compile or interpret the created programs. LOGENPRO can only guarantee that valid programs in the language specified by the logic grammar will be generated. However, to ensure that the produced programs can be successfully compiled or interpreted, an appropriate compiler/interpreter must be provided. Thus, the user must be very careful in designing the logic grammar and the fitness function.

The initial programs in generation 0 normally have poor performances. However, some programs in the population will be fitter than others. Fitness of each program in the generation is estimated and the following process is iterated over many generations until the termination criterion is satisfied. The reproduction, sexual crossover, and asexual mutation are used to create new generation of programs from the current one. The reproduction involves selecting a program from the current generation and allowing it to survive by copying it into the next generation. Either fitness proportionate or tournament selection can be used.

The crossover is used to create a single offspring program from two parental programs selected. Mutation creates a modified offspring program from a parental program selected. Unlike crossover, the offspring program is usually similar to the parent program. Logic grammars are used to constraint the offspring programs that can be produced by these genetic operations.

This algorithm will produce populations of programs which tend to exhibit increasing average of fitness. LOGENPRO returns the best program found in any generation of a run as the result.

## 3. Learning functional program

In this section, we describe how to use LOGENPRO to emulate traditional GP (Koza, 1992, 1994). GP has a limitation that all the variables, constants, arguments for functions, and values returned from functions must be of the same data type. This limitation leads to the difficulty of inducing even some rather simple and straightforward functional programs. For example, one of these programs calculates the dot product of two given numeric vectors of the same size. Let `X` and `Y` be the two input vectors, then the dot product is obtained by the following S-expression:

```
(apply (function +)  (mapcar (function *) X Y))
```

Let us use this example for illustrative comparison below. To induce a functional program using LOGENPRO, we have to determine the logic grammar, fitness cases, fitness functions, and termination criterion. The logic grammar for learning functional programs is given in table 2. In this grammar, we employ the argument of the grammar symbol `s-expr` to designate the data type of the result returned by the S-expression generated from the grammar symbol. For example,

```
(mapcar (function +) X (mapcar (function *) X Y))
```

is generated from the grammar symbol `s-expr([list, number, n])` because it returns a numeric vector of size n. Similarly, the symbol `s-expr(number)` can produce `(apply (function *) X)` that returns a number.

The terminal symbols `[ + ]`, `[ - ]`, and `[ * ]` represent functions that perform ordinary addition, subtraction, and multiplication respectively. The symbol `[ % ]` represents function that normally returns the quotient. However, if division by zero is attempted, the function returns 1.0. The symbol `protected-log` is a function that calculates the logarithm of the input argument `x` if `x` is larger than zero, otherwise it returns 1.0. The logic goal `random(-10, 10, ?a)` generates a random floating point number between -10 and 10 and instantiates `?a` to the random number generated.

Ten random fitness cases are used for training. Each case is a 3-tuples $\langle X_i, Y_i, Z_i \rangle$, where $1 \le i \le 10$, $X_i$ and $Y_i$ are vectors of size 3, and $Z_i$ is the corresponding dot product. The fitness function calculates the sum, taken over the ten fitness cases, of the absolute values of the difference between $Z_i$ and the value returned by the S-expression for $X_i$ and $Y_i$. Let S be an S-expression and $S(X_i, Y_i)$ be the value returned by the S-expression for $X_i$ and $Y_i$. The fitness function Val is defined as follows,

$$\mathrm{Val}(S) = \sum_{i=1}^{10} |S(X_i, Y_i) - Z_i|$$

A fitness case is said to be covered by an S-expression if the value returned by it is within 0.01 of the desired value. An S-expression that covers all training cases is further evaluated on a testing set containing 1000 random fitness cases. LOGENPRO will stop if the maximum number of generations of 100 is reached or a S-expression that covers all testing fitness cases is found.

For traditional GP, the terminal set T is {`X`, `Y`, `R`} where `R` is the ephemeral random floating point constant. `R` takes on a different random floating point value between -10.0 and 10.0 whenever it appears in an individual program in the initial population (Koza 1992). The function set F is {`protected+`, `protected-`,

protected*, protected%, protected-log, vector+, vector-, vector*, vector%, vector-log, apply+, apply-, apply*, apply%}, taking 2, 2, 2, 2, 1, 2, 2, 2, 2, 1, 1, 1, 1, and 1 arguments respectively.

The primitive functions protected+, protected-, and protected* respectively perform addition, subtraction, and multiplication if the two input arguments X and Y are both numbers. Otherwise, they return 0. The function protected% returns the quotient. However, if division by zero is attempted or the two arguments are not numbers, protected% returns 1.0. The function protected-log finds the logarithm of the argument X if X is a number larger than zero. Otherwise, protected-log returns 1.0.

The functions vector+, vector-, vector*, and vector% respectively perform vector addition, subtract, multiplication and, division if the two input arguments X and Y are numeric vectors with the same size, otherwise they return zero. The primitive function vector-log performs the S-expression:

```
(mapcar (function protected-log) X)
```

if the input argument X is a numeric vector, otherwise it returns zero. The functions apply+, apply-, apply*, and apply% respectively perform the following S-expressions if the input argument X is a numeric vector:

```
(apply (function protected+) X),
(apply (function protected-) X),
(apply (function protected*) X), and
(apply (function protected%) X),
```

otherwise they return zero.

It should be emphasized that the primitive functions vector+, vector-, vector*, and vector% can be emulated by using the grammar rules 11, 18, 19, 20, and 21. The primitive funcition vector-log can be emulated by using the grammar rules 12 and 22. The primitive functions apply+, apply-, apply*, and apply% can be emulated by using the grammar rules 15, 18, 19, 20, and 21. Thus the set of effective functions represented by the grammar in table 2 is equivalent to the set used in traditional GP. The functions mapcar and apply cannot be used in traditional GP because the first argument of these functions must be a valid operators such as +, -, *, or %. But traditional GP cannot enforce this constraint, thus we have to create some special functions such as vector+, apply+ etc. to handle this problem.

The fitness cases, the fitness function, and the termination criterion are the same as those used by LOGENPRO. Three experiments are performed. The first one evaluates the performance of LOGENPRO using a population of 100 programs. The other two experiments evaluate the performance of GP using respectively populations of 100 and 1000 programs. In each experiment, sixty trials are attempted and the results are summarized in figure 9. From the curves in figure 9, the lower values are better, thus LOGENPRO has superior performance than that of GP.

The curves in figure 10(a) show the experimentally observed cumulative probability of success P(M, i) of solving the problem by generation i using a population of M programs. The curves in figure 10(b) show the number of programs I(M, i, z) that must be processed to produce a solution by generation i with a probability z. Throughout this section, the probability z is set to 0.99. The curve for GP with a population of 100 programs is not depicted because the values is extremely large. For the LOGENPRO curve, I(M, i, z) reaches a minimum value of 8800 at

generation 21. On the other hand, the minimum value of I(M, i, z) for GP with population size of 1000 is 66000 at generation 1. LOGENPRO can find a solution much faster than GP and the computation (i.e. I(M, i, z)) required by LOGENPRO is much smaller than that of GP.

The performance of LOGENPRO is better because knowledge of data type has been encoded in the grammar. Consequent, invalid programs such as

```
(+ (apply (function +) 9) 9)
```

cannot be produced. On the other hand, traditional GP may create the equivalent invalid program `(+ (apply+ 9) 9)`. In other words, the search space of traditional GP is larger than that of LOGENPRO. But, the former contains many invalid programs.

The idea of applying knowledge of data type to accelerate learning has been investigated independently by Montana (1995) in his Strongly Typed Genetic Programming (STGP). He presents many examples involving vector and matrix manipulation to illustrate the operation of STGP. However, he has not compared the performance between traditional GP and STGP. Although it is commonly believed that knowledge can accelerate the speed of learning, Pazzani and Kibler (1992) showed that inappropriate and/or redundant knowledge can sometimes degrade the performance of a learning system. We show that knowledge of data type can be represented in a logic grammar and thus LOGENPRO can emulate the effect of STGP easily[1]. Moreover, more natural primitive functions such as `mapcar` and `apply` can be used in LOGENPRO, rather than using some special primitive functions such as `vector+` and `apply+` found in traditional GP.

## 4. Learning sub-functions using LOGENPRO

Automatic discovery of problem representation primitives is certainly one of the most challenging research areas in genetic programming. GP with ADF (Automatically Defined Functions) is one of the approaches that have been proposed to acquire problem representation primitives automatically (Koza, 1994). In this approach, each program in the population contains an expression, called the result producing branch, and definitions of one or more sub-functions which may be invoked by the result producing branch. The result producing branch is evaluated to produce the fitness of the program. A constrained syntactic structure and some modified genetic operators are required for the evolution of the programs. To employ GP with ADF, the user must provide explicit knowledge about the number of available automatically defined sub-functions, the number of arguments of each sub-functions, and the allowable terminal and function sets for each sub-function.

In this section, we demonstrate how to use LOGENPRO to emulate GP with ADF. LOGENPRO is employed to learn a sub-function that calculates dot product and employ this sub-function in the main program. In other words, it is expected to induce the following S-expression:

[1] In addition to emulating STGP, LOGENPRO can also emulate GP with ADF as described in section 4.

```
(progn
  (defun ADF0 (arg0 arg1)
     (apply (function +)
        (mapcar (function *) arg0 arg1)))
  (+ (ADF0 X Y) (ADF0 Y Z)))
```

In the logic grammar for this problem (table 3), we employ the argument of the grammar symbol `s-expr` to designate the data type of the result returned by the S-expression generated from the grammar symbol. The terminal symbols `[ + ]`, `[ - ]`, and `[ * ]` represent functions that perform ordinary addition, subtraction, and multiplication respectively.

Ten random fitness cases are used for training. Each case is a 4-tuples $\langle X_i, Y_i, Z_i, R_i \rangle$, where $1 \le i \le 10$, $X_i$, $Y_i$ and $Z_i$ are vectors of size 3, and $R_i$ is the corresponding desired result. The fitness function calculates the sum, taken over the ten fitness cases, of the absolute values of the difference between $R_i$ and the value returned by the S-expression for $X_i$, $Y_i$ and $Z_i$. Let S be an S-expression and $S(X_i, Y_i, Z_i)$ be the value returned by the S-expression for $X_i$, $Y_i$ and $Z_i$. The fitness function Val is defined as follows,

$$\mathrm{Val}(S) = \sum_{i=1}^{10} |S(X_i, Y_i, Z_i) - R_i|$$

A fitness case is said to be covered by an S-expression if the value returned by it is within 0.01 of the desired value. An S-expression that covers all training cases is further evaluated on a testing set containing 1000 random fitness cases. LOGENPRO will stop if the maximum number of generations is reached or an S-expression that covers all testing fitness cases is found.

For GP with ADF (with the modified genetic operator), the terminal set $T_0$ for the automatically defined function (ADF0) is {`arg0`, `arg1`} and the function set $F_0$ is {`protected+`, `protected-`, `protected*`, `vector+`, `vector-`, `vector*`, `apply+`, `apply-`, `apply*`}, taking 2, 2, 2, 2, 2, 2, 1, 1, and 1 arguments respectively. The terminal set $T_r$ for the result producing branch is {`X`, `Y`, `Z`} and the function set $F_r$ is {`protected+`, `protected-`, `protected*`, `vector+`, `vector-`, `vector*`, `apply+`, `apply-`, `apply*`, `ADF0`}, taking 2, 2, 2, 2, 2, 2, 1, 1, 1, and 2 arguments respectively. The primitive functions have already been defined in the previous section. The fitness cases, the fitness function, and the termination criterion are the same as the ones used by LOGENPRO. We evaluate the performance of LOGENPRO and the ADF using populations of 100 and 1000 programs respectively.

Thirty trials are attempted and the results are summarized in figures 11 and 12. Figure 11 shows, by generation, the fitness (error) of the best program in a population. These curves are found by averaging the results obtained in thirty different runs using various random number seeds and fitness cases. From these curves, LOGENPRO has superior performance than that of GP with ADF. The curves in figure 12(a) show the experimentally observed cumulative probability of success, P(M, i), of solving the problem by generation i using a population of M programs. The curves in figure 12(b) show the number of programs I(M, i, z) that must be processed to produce a solution by generation i with a probability z of 0.99. The curve for LOGENPRO reaches a minimum value of 4900 at generation 6. On the other hand, the minimum value of

I(M, i, z) for GP with ADF is 5712000 at generation 41. This experiment clearly shows the advantage of LOGENPRO. By employing various knowledge about the problem being solved, LOGENPRO can find a solution much faster than GP with ADF and the computation (i.e. I(M, i, z)) required by LOGENPRO is much smaller than that of GP with ADF. Moreover, LOGENPRO can emulate the effects of STGP and GP with ADF simultaneously and easily.

# 5. Learning recursive functions from noisy examples

An approach to make a large problem more tractable is to discover problem representations automatically. Koza (1994) uses the even-n-parity problem to demonstrate that his approach of hierarchical Automatically Defined Functions (ADF) can facilitate the solving of the problem.

Koza shows that the even-7-parity problem can be solved using GP with hierarchical ADF. He finds that about 1440000 functions, I(M, i, z), should be evaluated to obtain at least one solution with 99% probability. In each fitness calculation, 128 fitness cases must be evaluated. Thus, 1440000*128 = 184320000 fitness cases should be processed. Unfortunately, the solutions found can only solve the even-n-parity problem with a particular value of n. If a different value of n is used, GP with hierarchical ADF must be applied again to find other programs that can solve the new even-n-parity problem.

Clearly, the solution found is not general enough to solve all instances of the even-n-parity problem for all $n \geq 0$. In this section, we use LOGENPRO to evolve general recursive functions for the even-n-parity problem from noisy training examples.

The even-n-parity problem is first presented in sub-section 5.1, followed by the sub-section describing the difficulties of learning recursive functions for the problem. The method of representing domain-specific knowledge using logic grammars is discussed in sub-section 5.3. Next, we describe a number of experiments that evaluate the impact of knowledge and noise in examples on the rate of inducing general recursive functions. The last sub-section is a discussion.

## 5.1. The even-n-parity problem

The boolean even-n-parity function of n boolean input arguments returns true (`T`) if an even number of the arguments are true, otherwise it returns false (`nil`). Koza (1994) used GP with hierarchical ADF to induce the function. The training set contains all $2^n$ combinations of the n boolean input arguments. The standardized fitness of an S-expression is the sum of the error between the value returned by the S-expression and the correct value of the even-n-parity function. The maximum number of generations is 51. The population sizes for the even-3-, 4-, 5-, and 6-parity problems are 16000. On the other hand, the population size for the even-7-parity problem is 4000.

## 5.2. The recursive even-n-parity function

Since all $2^n$ fitness cases, for a particular value of n, are used as the training examples, it is unclear whether GP can discover the regularities of the even-n-parity problem and induce a general function. A more general recursive function which can solve all instances of the problem for all $n \geq 0$ is shown in table 4.

The recursive function is composed of two components. The first component is the base statement:

```
if (null L) T
```

specifying that the function must return true (T) if the input argument L is an empty list. The second component is the recursive statement:

```
(AND (OR (first L) (parity (rest L)))
     (NAND (first L) (parity (rest L))))
```

In general, a recursive function consists of one or more base statements and a number of recursive statements. It is difficult to evolve a recursive function because appropriate base and recursive statements and correct ordering of them must be evolved simultaneously.

For example, the following function:

```
(defun parity (L)
  (AND (or (first L) (parity (rest L)))
       (if (null L) T
        (AND (OR (first L) (parity (rest L)))
             (NAND (first L) (parity (rest L)))))))
```

is incorrect, although the second component of the outermost AND function is the target recursive function to be evolved.

Moreover, consider the problem of inducing a function from all fitness cases of the even-3-parity problem, the standardized fitness value of the function:

```
(defun parity (L)
  (if (null L) T (first L)))
```

is only 4, although its base statement is correct. The standardized fitness value of the function:

```
(defun parity (L)
  (if (null L) nil
    (AND (OR (first L) (parity (rest L)))
    (NAND (first L) (parity (rest L))))))
```

is 8 (the worst value), although its recursive statement is correct. These examples illustrate that the problem of inducing recursive functions is difficult, because the properties of the problem obstruct the construction and combination of good building blocks and it is hard to design fitness functions that are able to assign partial credit.

### 5.3. Representing knowledge using logic grammars

To evolve recursive functions using LOGENPRO, we have to determine the terminals, the primitive functions, the fitness cases, the fitness function, and the termination criterion. The terminal set is {L, T, nil} where L is the input argument of the recursive function to be learned, T and nil are boolean truth values. The argument L is a list of boolean values and any number of boolean values can exist in the list L. The set of primitive functions is {AND, OR, NAND, NOR, ifnil, first, rest, parity}

The boolean functions AND, OR, NAND, and NOR take two boolean input arguments and return one boolean value. The function ifnil takes three arguments. The first argument must be an S-expression that returns a list of boolean values. The last two arguments must be S-expressions that produce boolean output values. The function ifnil checks whether the first input argument returns an empty list. If the list is empty, ifnil returns the boolean value passed into the second S-expression as

the output value of the function, otherwise it returns the boolean value of the third S-expression.

The primitive function `first` takes a list of boolean values as its argument and returns the first boolean value of the list if the list is not empty. Otherwise, the function generates an exception signal to indicate that an illegal operation has been attempted to get the first element from an empty list. The primitive function `rest` must take a list of n boolean values, for any value of $n \geq 0$, as its argument. If the input list is not empty, it returns a list containing the last n-1 elements of the input list, otherwise, the function generates an exception signal to indicate an illegal action has been tried. The primitive function `parity` takes a list of boolean values as its input and returns a boolean value. This function recursively calls the recursive function being evolved by LOGENPRO.

There are three data types: BOOLEAN, LIST, and SMALLER-LIST, that can be used to specify the primitive functions. The data type LIST contains lists of n boolean values, for any value of $n \geq 0$. The data type SMALLER-LIST contains lists of m boolean values for any value of $m < n$. The arities, the data types of the input arguments, and the data types of the output values of all primitive functions are summarized in table 5.

It can be observed from table 5 and the function in table 4 that three kinds of knowledge can be used to facilitate the learning of recursive functions. They are:

- *Knowledge_1*: data types of input arguments and output value,
- *Knowledge_2*: the recursive function to be evolved must return `T` if the input argument `L` is an empty list, and
- *Knowledge_3*: the outermost statement of the recursive function to be evolved must be the base statement, i.e. `if (null L) t`.

The terminal set, the primitive functions, and different kinds of knowledge can be represented easily using logic grammars. Three logic grammars: Grammar_1, Grammar_2, and Grammar_3; have been developed to encode different amounts of knowledge. The kinds of knowledge represented in the three grammars are summarized as follows:

| | *Knowledge_1* | *Knowledge_2* | *Knowledge_3* |
|---|---|---|---|
| Grammar_1 | yes | yes | yes |
| Grammar_2 | yes | yes | no |
| Grammar_3 | yes | no | no |

The rules of Grammar_1 are given in table 6. As described in section 3, we employ the argument of the non-terminal grammar symbol `s-expr` to designate the data type of the result returned by the S-expression generated from the grammar symbol.

The terminal grammar symbols **`[ T ]`**, **`[ nil ]`**, and **`[ L ]`** in rules 32, 33, and 37 of Grammar_1 form the terminal set of the problem. The terminal grammar symbols **`[ AND ]`**, **`[ OR ]`**, **`[ NAND ]`**, and **`[ NOR ]`** in rules 40, 41, 42, and 43 represent primitive functions that perform ordinary boolean operations. Rule 34 of Grammar_1 specifies that these primitive functions take two boolean input arguments and return one boolean value.

The terminal symbol **`[ first ]`** in rule 36 represents the primitive function that returns the first element of a list of boolean values. Rule 36 also specifies that the primitive function takes a list of boolean values as its argument and returns a boolean value. The terminal symbol **`[ rest ]`** in rule 39 represents the primitive function that takes a list of n boolean values as its argument and returns a list containing the last n - 1 elements of the input list. Rule 39 also declares the arity of the function and the data types of its argument and output value.

Since an item of the SMALLER-LIST data type is also a list of boolean values, it should belong to the LIST data type. This fact is specified in rule 38. This example shows that a hierarchy of data types can be declared easily using grammar rules.

The primitive function represented by the terminal symbol **`[ parity ]`** in rule 35 must take a list of the SMALLER-LIST data type. This rule avoids a non-terminating recursive function such as:

```
(defun parity (L) (ifnil L T (parity L)))
```

to be evolved by LOGENPRO.

The first rule of Grammar_1 represents the domain-specific knowledge that the recursive function to be evolved must return `T` if the input argument `L` is an empty list, and the outermost statement of the function must be the correct base statement.

Grammar_2 is similar to Grammar_1, the main difference between them is that rule 31 of Grammar_1 has been changed to rule 51:

```
51:  start      ->   [ (defun parity (L) ],
                     s-expr(boolean), [ ) ].
```

Moreover, Grammar_2 has one additional rule (rule 52):

```
52:  s-expr(BOOLEAN) ->  [ ( ], [ ifnil L T ],
                         s-expr(BOOLEAN), [ ) ].
```

representing the domain-specific knowledge that the recursive function to be evolved should return `T` if the input argument `L` is an empty list.

Grammar_3 is similar to Grammar_2, the difference between them is that rule 52 of Grammar_2 has been changed to rule 53 of Grammar_3:

```
53:  s-expr(BOOLEAN) ->  [ ( ], [ ifnil ],
                         s-expr(LIST),
                         s-expr(BOOLEAN),
                         s-expr(BOOLEAN), [ ) ].
```

To determine the impact of domain-specific knowledge on the speed of learning, a number of experiments have been performed using different grammars. In this section, LOGENRPO with Grammar_1, Grammar_2, and Grammar_3 are designated as LOGENPRO_1, LOGENPRO_2, and LOGENPRO_3 respectively. The results of the experiments are discussed in the next sub-section.

### 5.4. Experiments

Two series of experiments have been done. They differ in the fitness functions and the fitness cases used. In each experiment, the population size is 500 and the maximum number of generations is 50. The probabilities of performing crossover and mutation are respectively 0.7 and 0.1. The maximum depth of derivation trees generated in the initial population is 12. The maximum depth of trees produced by crossover and mutation is also 12. The two series of experiments are presented in sub-sections 5.4.1 and 5.4.2 respectively.

#### 5.4.1. The first series of experiments

Three experiments have been performed repeatedly for 60 times to evaluate the abilities of LOGENPRO_1, LOGENPRO_2, and LOGENPRO_3 in inducing recursive functions for the even-n-parity problem. In these experiments, the even-0, 2, and 3- parity problems are used in the training process. The training set contains all 13 fitness cases from these even-parity problems. The standardized fitness value of an evolved function is the total number of misclassifications on the 13 fitness cases. The evolution terminates if the maximum number of generations of 50 is reached or a function that classifies all fitness cases correctly is found. In order to avoid the problem caused by an inefficient recursive function, an execution time limit is enforced. After executing 100 primitive functions, if the evolved function fails to find a result for a fitness case, it will be terminated. In this case, it is assumed that the function will misclassify the corresponding fitness case.

It is possible that an evolved function will generate exceptions during its execution for some fitness cases, because it is illegal to perform the first/rest operation on an empty list. If the function produces an exception, it is assumed that it will misclassify the corresponding fitness cases.

In the 60 trials, LOGENPRO_1 successfully evolves 16 functions that classify all fitness cases correctly. The generated functions are then tested on the even-i-parity problems, where $i \in \{0, 1, 2, 4, 5, 6, 7, 8, 9, 10\}$. All of them can successfully solve all the problems. They are further analyzed manually and it is found that these 16 functions are correct recursive functions for the general even-n-parity problem.

The I(M, i, z) for z of 99% reaches a minimum value of 335000 at generation 9 (Koza 1992). Since there are only 13 fitness cases, 335000*13 = 4355000 fitness cases should be processed. Thus, LOGENPRO can find a general, recursive function for the even-n-parity problem very efficiently. On the other hand, GP with hierarchical ADF evaluates 184320000 fitness cases to find a function that solves the even-7-parity problem only. In other words, LOGENPRO_1 can solve the even-7-parity problem about 42 times faster.

LOGENPRO_2 successfully evolves 16 functions that classify all fitness cases correctly. All of them are correct recursive functions for the general even-n-parity problem. The I(M, i, z) for z of 99% reaches a minimum value of 287500 at generation 24, and 287500*13 = 3737500 fitness cases should be processed. It is found that LOGENPRO_2 can solve the even-7-parity problem about 49.3 times faster than GP with hierarchical ADF. By comparing the performance of LOGENPRO_1 and LOGENPRO_2, it is interesting to find that LOGENPRO_2 can solve the problem using less computation effort than LOGENPRO_1. This

observation implies that *Knowledge_3* is not very effective in accelerate the learning if the training examples are not noisy[2].

LOGENPRO_3 successfully evolves 5 functions that classify all fitness cases correctly. All of them are correct recursive functions for the general even-n-parity problem. The I(M, i, z) for z of 99% reaches a minimum value of 1272000 at generation 47, and 1272000*13 = 16536000 fitness cases should be processed. It is found that LOGENPRO_3 can solve the even-7-parity problem about 11 times faster than GP with hierarchical ADF. By comparing the performance of LOGENPRO_1 and LOGENPRO_3, a 3.8 times speedup is observed.

### 5.4.2. The second series of experiments

Three experiments have been performed repeatedly for 60 times to evaluate the abilities of LOGENPRO_1, LOGENPRO_2, and LOGENPRO_3 in inducing recursive functions for the even-n-parity problem from noisy training examples. The even-0-, 2-, and 3- parity problems are used in the training process. The training set contains all 13 fitness cases from these even-parity problems. To introduce noise into the training examples, one of them is randomly selected and the result of the selected example is modified from `T` to `nil` or from `nil` to `T`. The fitness function and the termination criterion are the same as those of the first series of experiments.

LOGENPRO_1 successfully evolves 10 correct recursive functions for the general even-n-parity problem[3]. The I(M, i, z) for z of 99% reaches a minimum value of 450000 at generation 9. Since there are 13 fitness cases in the training set, 450000*13 = 5850000 fitness cases should be processed. It is found that 1.34 times effort must be used if a noisy training set is employed compared to the corresponding experiment in the first series. On the other hand, LOGENPRO_1 can solve the even-7-parity problem about 32 times faster than GP with hierarchical ADF.

LOGENPRO_2 successfully evolves 5 correct recursive functions for the general even-n-parity problem. The I(M, i, z) for z of 99% reaches a minimum value of 1215000 at generation 26, and 1215000*13 = 15795000 fitness cases should be processed. It is found that LOGENPRO_2 can solve the even-7-parity problem about 12 times faster than GP with hierarchical ADF. However, about 4 times effort must be used if a noisy training set is employed. By comparing the performance of LOGENPRO_1 and LOGENPRO_2, it can be observed that a 2.7 times speedup is achieved if knowledge_3 is available. This observation implies that knowledge_3 is effective in handling noisy training examples.

LOGENPRO_3 successfully evolves 2 correct recursive functions for the general even-n-parity problem. The I(M, i, z) for z of 99% reaches a minimum value of 2475000 at generation 47, and 2475000*13 = 32175000 fitness cases should be processed. It is found that LOGENPRO_3 can solve the even-7-parity problem about 6 times faster than GP with hierarchical ADF. By comparing the performance of

---

[2] One possible explanation is that the search space of LOGENPRO_1 is too restrictive to hinder the evolution of good recursive programs such as `(defun parity (L) (AND (ifnil L t (ifnil L t (OR (first L) (parity (rest L))))) (ifnil L t (NAND (first L) (parity (rest L))))))`.

[3] To determine whether a recursive function is correct, a number of even-i-parity problems without introduced noise, where $i \in \{0, 1, 2, 4, 5, 6, 7, 8, 9, 10\}$, are used to test the function. If the function can pass the test, it is further analyzed manually to confirm its correctness.

LOGENPRO_1 and LOGENPRO_3, a 5.5 times speedup is observed. However, about 2 times effort must be used if a noisy training set is employed.

### 5.5. Discussion

In this section, we employ LOGENPRO to evolve recursive functions for the general even-n-parity problem from training examples with or without noise. Two series of experiments have been performed to study the impact of domain-specific knowledge and noise in the training examples on the speed of learning recursive functions. The numbers of fitness cases processed to induce general recursive functions with 99% probability for the two series of experiments are summarized in table 7. These experiments demonstrate that knowledge may accelerate the speed of learning while more computation effort may be required if noisy training examples are employed. Moreover, LOGENPRO can solve the even-7-parity problem much faster than GP with hierarchical ADF.

## 6. Conclusion

We have presented a flexible system called LOGENPRO (The LOgic grammar based GENetic PROgramming system) that uses some of the techniques of genetic programming and inductive logic programming. It is based on a formalism of logic grammars. The system can learn programs in various programming languages and represent context-sensitive information and domain-dependent knowledge. An experiment that employs LOGENPRO to induce an S-expression for calculating dot product has been performed. This experiment illustrates that LOGENPRO, when used with domain knowledge, accelerates the learning of programs.

Automatic discovery of sub-functions is one of the most important research areas in genetic programming. In GP with ADF, the user must provide explicit knowledge about the number of available sub-functions, the number of arguments of each sub-functions, and the allowable terminal and function sets for each sub-function. An experiment has been performed to demonstrate that LOGENPRO can emulate GP with ADF and represent the knowledge easily. Moreover, LOGENPRO can employ other knowledge such as argument types in a unified framework. This experiment shows that LOGENPRO has superior performance to that of GP with ADF when more domain-dependent knowledge is available.

We have applied LOGENPRO to evolve general recursive functions for the even-n-parity problem. Two series of experiments have been performed to study the impact of domain-specific knowledge and noise in the training examples on the speed of learning recursive functions. These experiments show that knowledge may accelerate the speed of learning. But more computation effort may be required if noisy training sets were employed in the training process.

For future work, we plan to evolve hierarchical recursive functions for the even-n-parity problem. There are many inductive learning systems such as THESYS (Summers, 1977) and ADATE (Olsson, 1995) that can induce recursive functional programs efficiently. Therefore, we will implement these techniques on LOGENPRO. Moreover, reusable templates of logic grammars could be provided to eliminate the potential difficulties in developing the right grammar for a given problem.

## Acknowledgments

The research is sponsored by the RGC Earmarked research grant of UGC reference number CUHK 486/95E.

## Reference

Abramson, H. and Dahl, V. (1989). *Logic Grammars*. Berlin: Springer-Verlag.
Angeline, P. J. and Kinnear, K. E. Jr., editors (1996). *Advances in Genetic Programming 2*. MA: MIT Press.
Biermann, A. W. (1972). On the Inference of Turing Machines from Sample Computations. *Artificial Intelligence*, **3**, pp. 181-198.
Biermann, A. W. and Smith, D. R. (1979). A Production Rule Mechanism for Generating LISP Code. *IEEE Trans. on Systems, Man, and Cybernetics*, **9**, pp. 260-276.
Cohen, W. (1992). Compiling Prior Knowledge into an Explicit Bias. In *Proceedings of the Ninth International Workshop on Machine Learning*, 102-110. CA: Morgan Kaufmann.
Cramer, N. L. (1985). A Representation for the Adaptive Generation of Simple Sequential Programs. In J. J. Grefenstette (ed.), *Proceedings of the First International Conference on Genetic Algorithms and Their Applications*. Hillsdale NJ: Lawrence Erlbaum.
Davis, L. D. (1991). *Handbook of Genetic Algorithms*. Van Nostrand Reinhold.
DeJong, G. F., editor (1993). *Investigating Explanation-Based Learning*. Boston: Kluwer Academic Publishers.
DeJong, G. F. and Mooney, R. (1986). Explanation-Based Learning: An Alternative View. *Machine Learning*, **1**, pp. 145-176.
De Raedt, L. (1992). *Interactive Theory Revision: An Inductive Logic Programming Approach*. London: Academic Press.
De Raedt, L. and Bruynooghe, M. (1992). Interactive Concept Learning and Constructive Induction by Analogy. *Machine Learning*, **8**, pp. 251-269.
Goldberg, D. (1989). *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading MA: Addison-Wesley.
Gruau, F. (1996). On Using Syntactic Constraints with Genetic Programming. In P. J. Angeline and K. E. Kinnear, Jr. (Eds.) *Advances in Genetic Programming 2*, 402-417. MA: MIT Press.
Holland, J. (1992). *Adaptation in Natural and Artificial Systems*. Cambridge MA: MIT Press.
Hopcroft, J. E. and Ullman, J. D. (1979). *Introduction to automata theory, languages, and computation*. MA: Addison-Wesley.
Kinnear, K. E. Jr., editor (1994). *Advances in Genetic Programming*. Cambridge MA: MIT Press.
Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. Cambridge MA: MIT Press.
Koza, J. R. (1992). *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. Cambridge MA: MIT Press.
Koza, J. R., Goldberg, D. E., Fogel, D. B., and Riolo, R. L. editors (1996). *Proceedings of the 1996 Genetic Programming Conference*. Cambridge MA: MIT Press.
Lavrac, N. and Dzeroski, S. (1994). *Inductive Logic Programming: Techniques and Applications*. London: Ellis Horword.
Lewis, H. R. and Rapadimitrion, C. H. (1981). *Elements of the theory of computation*. NJ: Prentice Hall.

Mitchell, T. M. (1982). Generalization as Search. *Artificial Intelligence*, **18**, pp. 203-226.
Mitchell, T. M., Keller, R. M. and Kedar-Cabelli, S. T. (1986). Explanation-Based Generalization: A Unifying View. *Machine Learning*, **1**, pp. 47-80.
Montana, D. J. (1995). Strongly Typed Genetic Programming. *Evolutionary Computation*, **3**, pp. 199-230.
Muggletion, S. (1992). Inductive Logic Programming. In S. Muggletion (ed.), *Inductive Logic Programming*, pp. 3-27. London: Academic Press.
Muggletion, S. (1994). Inductive Logic Programming. *SIGART Bulletin*, **5** (1), pp. 5-11.
Muggleton, S. and De Raedt, L. (1994). Inductive Logic Programming: Theory and Methods. *J. Logic Programming*, **19-20**, pp. 629-679.
Olsson, R. (1995). Inductive Functional Programming using Incremental Program Transformation. *Artificial Intelligence*, **74**, pp. 55-81.
Pazzani, M. and Kibler, D. (1992). The utility of knowledge in Inductive learning. *Machine Learning*, **9**, pp. 57-94.
Pereira, F. C. N. and Shieber, S. M. (1987). *Prolog and Natural-Language Analysis*. CA: CSLI.
Pereira, F. C. N. and Warren, D. H. D. (1980). Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks. *Artificial Intelligence*, **13**, pp. 231-278.
Quinlan, J. R. (1991). Knowledge Acquisition from Structured Data - using Determinate Literals to Assist Search. *IEEE Expert*, **6**, pp. 32-37.
Quinlan, J. R. (1990). Learning Logical Definitions from Relations. *Machine Learning*, **5**, pp. 239-266.
Rich, C. and Waters, R. C. (1990). *The Programmer's Apprentice*. New York: Addison-Wesley..
Rich, C. and Waters, R. C. (1988). Automatic Programming: Myths and Prospects. *IEEE Computer*, **21**(8), pp. 40-51.
Sterling, L. and Shapiro, E. (1986). *The Art of Prolog*. MA: MIT Press.
Summers, P. D. (1977). A Methodology for LISP Program Construction from Examples. *JACM*, **24**, pp. 161-175.
Whigham, P. A. (1996). Search Bias, Language Bias and Genetic Programming. In *Proceedings of the First Genetic Programming Conference*, 230-237. Cambridge, MA: MIT Press.
Whigham, P. A. (1995). Inductive Bias and Genetic Programming. In *Proceedings of the First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, 461-466. UK: IEE.
Wong, M. L. and Leung, K. S. (1996). Learning Recursive Functions from Noisy Examples using Generic Genetic Programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo (Eds) *Proceedings of Genetic Programming 1996 Conference***.** pp. 238-246. MA: MIT Press.
Wong, M. L. and Leung, K. S. (1995a). An Induction System that Learns Programs in different Programming Languages using Genetic Programming and Logic Grammars. In *Proceedings of the 7th IEEE International Conference on Tools with Artificial Intelligence*. pp. 380-387. CA: IEEE Computer Society Press.
Wong, M. L. and Leung, K. S. (1995b). Combining Genetic Programming and Inductive Logic Programming using Logic Grammars. In *Proceedings of the 1995 IEEE International Conference on Evolutionary Computing*. pp. 733-736. CA: IEEE Computer Society Press.
Wong, M. L. and Leung, K. S. (1995c). Applying Logic Grammars to Induce sub-functions in Genetic Programming. In *Proceedings of the 1995 IEEE International*

*Conference on Evolutionary Computing*. pp. 737-740. CA: IEEE Computer Society Press.

## Appendix A

In this appendix, we discuss the method of implementing LOGENPRO using a Prolog-like logic programming language. The differences between the logic programming language used and Prolog are listed as follows:

- A variable is represented by a question mark ? followed by a string of letters and/or digits.
- The elements of a list can be separated by either commas or spaces. For example, `[a b c]` and `[a, b, c]` are equivalent.
- A pair of '`|`' is used to represent a frozen terminal symbol. For example, the symbol **`[)]`** in the second rule of the grammar in table 1 is translated into `|)|`.
- A pair of braces encloses a sequence of logic goals appearing in a logic grammar.
- If there are a number of clauses $C_1$, $C_2$, ..., $C_n$ that match a goal G, the ordering of evaluating these clauses is determined randomly.

Using the difference list approach (Sterling and Shapiro, 1986), a grammar rule of the form:

$A_0$ -> $A_1$, $A_2$, ..., $A_n$.

is translated into a logic program clause of the form:

$A_0$' :- $A_1$', $A_2$', ..., $A_n$'.

in the logic programming language. Here, if $A_i$ , for some `i` between 0 and n, is a non-terminal with M arguments, then $A_i$' is a literal with M+3 arguments. The predicate symbols of $A_i$ and $A_i$' are the same. For example, $A_i$ is translated into `exp(?X, ?Tree, ?`$S_j$`, ?`$S_{j+1}$`)`, for some `j`, if $A_i$ is `exp(?X)`. The literal `exp(?X, ?Tree, ?`$S_j$`, ?`$S_{j+1}$`)` states that the sequence of symbols between `?`$S_j$ and `?`$S_{j+1}$ is a sentence of the category represented by the non-terminal symbol `exp(?X)`. The derivation tree of the sentence is stored in the logic variable `?Tree`.

A terminal symbol such as `[a b c]` is translated to a literal with 3 arguments: `connect([a b c], ?`$S_j$`, ?`$S_{j+1}$`)`, for some `j`. The predicate `connect` is defined as:

```
connect(?A, ?S0, ?S1) :- append(?A, ?S1, ?S0).
```

This predicate declares that the list of symbols stored in the logic variable `?A` can be found in the sequence of symbols between `?S0` and `?S1`.

If $A_k$, for some `k` between 1 and n, is a sequence of pure logic goals enclosed by a pair of braces, i.e., $A_k$ has the form of {$G_0$, $G_1$, ...., $G_m$}, then $A_k$' is obtained from $A_k$ by removing the pair of braces.

For example, the grammar depicted in table 1 can be translated into the logic program presented in table 8. In the clause 1' of the logic program shown in table 8, the compound term `tree(start, [(*], ?E1, ?E2, frozen(?E3), |)|)` indicates that it is a tree with a root labeled as `start`. The children of the root include the terminal symbol `[(*]`, a tree created from the non-terminal `exp(W)`, another tree created from the non-terminal `exp(W)`, a frozen tree generated from the non-terminal **`exp(W)`**, and the frozen terminal `|)|`.

Thus, a derivation tree can be generated randomly by issuing the following query:

```
?- start(?T, ?S, []).
```

This goal can be satisfied by deducing a sentence that is in the language specified by the grammar. One of the solutions is:

```
?S = [(* (/ W 1.5) (/ W 1.5) (/ W 1.5))]
```

and the corresponding derivation tree is:

```
?T = tree(start, [(*],
          tree(exp(W), [(/ W 1.5)]),
          tree(exp(W), [(/ W 1.5)]),
          frozen(tree(exp(W), [(/ W 1.5)])),
          |)|)
```

This is exactly a representation of the derivation tree shown in figure 1. In fact, the bindings of all logic variables and other information are also maintained in the derivation trees to facilitate the genetic operations that will be performed on the derivation trees.

## Appendix B.

The crossover algorithm is described in tables 9, 10, and 11.

Consider two parental programs generated randomly from the grammar in table 1. The primary parent is `(+ (- Z 3.5) (- Z 3.8) (/ Z 1.5))` and the secondary parent is `(* (/ W 1.5) (+ (- W 11) 12) (- W 3.5))`. The corresponding derivation trees are depicted in figures 3 and 4 respectively.

In step 1 of the crossover algorithm in table 9, the global variable PRIMARY-SUB-TREES contains the sub-trees 2, 3, 5, 6, and 8. The primary derivation tree (i.e. the sub-tree 0), the sub-trees 1, 4, 7, 10 that contain logic goals, and the frozen sub-trees 9, 10, 11, and 12 are excluded. The whole primary derivation tree cannot be mated because it must be generated from the grammar symbol `start`. If the symbol `start` is not recursive (i.e. `start` does not appear on the right hand side of a rule), the whole secondary derivation tree must be chosen for crossover. Thus, the offspring program must be a copy of the secondary parental program. In fact, the same effect can be obtained by reproducing the secondary parental program.

The sub-trees containing logic goals are excluded from crossover operations for two reasons. Firstly, the crossover algorithm can be greatly simplified if logic goals are prevented from performing crossover. Secondly, logic goals specify the conditions that must be satisfied before the rule can be applied and/or the computations that should be done. Hence, from the viewpoint of natural selection and reproduction, the interpretation of crossover between logic goals is unclear and unnatural. Thus this kind of operations is avoided.

Similarly, the sub-trees 13, 15, 16, 18, 19, and 20 are assigned to the global variable SECONDARY-SUB-TREES in step 2. In the next step, a sub-tree in the variable PRIMARY-SUB-TREES is selected randomly using a uniform distribution if the variable is not empty. Assume that the sub-tree 2 is selected as the SEL-PRIMARY-SUB-TREE. Thus, it is removed from the variable PRIMARY-SUB-TREES in step 4. A copy of the variable SECONDARY-SUB-TREES is made and stored into the global variable TEMP-SECONDARY-SUB-TREES in step 5.

Steps 6 to 8 form a loop that finds an appropriate sub-tree from the variable TEMP-SECONDARY-SUB-TREES. A sub-tree, SEL-SECONDARY-SUB-TREE, is appropriate if a valid offspring can be obtained by executing crossover between the SEL-PRIMARY-SUB-TREE and the SEL-SECONDARY-SUB-TREE. If no appropriate sub-tree can be found in this loop, the algorithm returns back to step 3 to find another SEL-PRIMARY-SUB-TREE. Assume that the sub-tree 15 is chosen as the SEL-SECONDARY-SUB-TREE. Step 8 determines whether a valid offspring can be obtained. It is the most complicate procedure in this algorithm and it is delineated in table 10 and explained in the following paragraphs.

In step 11 of the algorithm shown in table 10, the sub-trees 1, 3, 6, 9, and 12 are found to be the siblings of the SEL-PRIMARY-SUB-TREE 2 and stored into the global variable SIBLINGS. The SIBLINGS can be thought as the context around the PRIMARY-CROSSOVER-POINT and the context's consistency has to be checked and computed. The purpose of step 12 is to remove the bindings established solely by the SEL-PRIMARY-SUB-TREE which will be deleted by the crossover operation. To achieve this goal, the bindings of each sub-tree in the variable SIBLINGS is modified so that only the bindings established by itself is retained. The bindings instantiated by

a sub-tree can be found easily using the techniques of explanation-based learning (DeJong, 1993; Mitchell et al., 1986; DeJong and Mooney, 1986). For example, the bindings `{?x/Z}` of the sub-tree 1 need not be modified because the logic variable `?x` is instantiated to the value `Z` by the logic goal `member(?x, [W, Z])`. The bindings `{?x/Z}` of the sub-tree 3 is changed to an empty list because the logic variable `?x` is bound to the value `Z` by the sub-tree 1. Similarly, the bindings `{?x/Z}` of the sub-trees 6 and 9 are changed to empty lists. The bindings of the sub-tree 12 is not changed because it is already empty.

In step 13, the bindings of the SEL-SECONDARY-SUB-TREE is modified so that only the bindings established by itself is retained. The purpose is to identify the effect of the sub-tree on the logic variables. In this example, since the grammar symbol of the SEL-SECONDARY-SUB-TREE 15 has no argument, its bindings is empty. In fact, the primary and secondary derivation trees are pre-processed by LOGENPRO using an algorithm based on the techniques of Explanation-Based Learning (EBL). The algorithm finds the bindings established solely by the corresponding sub-trees of the derivation trees. The results are stored in the sub-trees so that they can be retrieved in constant time $C_r$. Thus the time complexity of step 12 is O(n) where n is the number of sub-trees in the global variable SIBLINGS. Similarly, the time complexity of step 13 is O(1).

In step 14, the second grammar rule is satisfied by the sub-trees in SIBLINGS and the SEL-SECONDARY-SUB-TREE. Moreover, this rule reaches the conclusion `start` which is consistent with the requirement of the sub-tree 0, the parent of the SEL-PRIMARY-SUB-TREE. Thus, the offspring generated is valid. The procedure that checks whether a conclusion is consistent is presented in table 11.

In step 9 of the crossover algorithm in table 9, the offspring is generated. In the next step, it is returned as the solution after some house-keeping tasks have been performed. The house-keeping tasks update the bindings and the rule numbers of the sub-trees of the offspring. The offspring program of this example is `(* (- Z 3.5) (- Z 3.8) (/ Z 1.5))` and its derivation tree is shown in figure 5. It is interesting to find that the sub-tree 25 has the rule number 2. This indicates that the sub-tree is generated by the second grammar rule rather than the third rule applied to the primary parent. The second rule must be used because the terminal symbol `[(+]` is changed to `[(*]` and only the second rule can create the terminal `[(*]`. In fact, this situation is identified in step 14 of the function `is-valid` and a record is maintained so that the rule number can be changed to 2 by the house-keeping procedure.

In another example, the same primary and secondary parents are used. Assume that the SEL-PRIMARY-SUB-TREE 3 is selected in step 3 and the SEL-SECONDARY-SUB-TREE 16 is chosen in step 7 of the crossover algorithm. Now, the siblings of the SEL-PRIMARY-SUB-TREE 3 are the sub-trees 1, 2, 6, 9, and 12. Although the SEL-PRIMARY-SUB-TREE has the bindings `{?x/Z}`, the instantiation of the logic variable `?x` to value `Z` is done by the sub-tree 1. In other words, the SEL-PRIMARY-SUB-TREE has not established any binding. In step 12 of the function `is-valid`, the bindings `{?x/Z}` of the sub-tree 1 is not modified because the logic variable `?x` is instantiated to the value `Z` by the logic goal `member(?x, [W, Z])`. The bindings of the sub-trees 2 and 12 are not changed because they are already empty. The bindings `{?x/Z}` of the sub-trees 6 is changed

to an empty list because the logic variable `?x` is bound to the value `Z` by the sub-tree 1. Similarly, the bindings `{?x/Z}` of the sub-tree 9 is changed to an empty list.

The SEL-SECONDARY-SUB-TREE has the bindings `{?x/W}`, but the instantiation of `?x` is performed by the sub-tree 14. Thus, the bindings of the SEL-SECONDARY-SUB-TREE is changed in step 13 to an empty list (i.e. the logic variable `?x` is not instantiated). In step 14, since the third rule satisfies all requirements, a valid offspring `(+ (/ Z 1.5) (- Z 3.8) (/ Z 1.5))` as depicted in figure 6 is created.

As a further example, the same primary and secondary parents are used. Assume that the SEL-PRIMARY-SUB-TREE 6 is selected in step 3 of the crossover algorithm and the SEL-SECONDARY-SUB-TREE 19 is chosen in step 7. The variable SIBLINGS contains the sub-trees 1, 2, 3, 9, and 12. In step 12 of the function `is-valid` (table 10), the bindings `{?x/Z}` of the sub-tree 1 is not modified. The bindings of the sub-trees 2 and 12 are not modified because they are already empty. The bindings `{?x/Z}` of the sub-trees 3 and 9 are changed to empty lists because the logic variable `?x` is bound to the value `Z` by the sub-tree 1.

The SEL-SECONDARY-SUB-TREE 19 has the bindings `{?x/W}`. This sub-tree is generated from the rule 7 and the application of this rule will instantiate the logic variable `?x` to the value `W`. In other words, the SEL-SECONDARY-SUB-TREE performs the instantiation of `?x` to `W`. Thus, the bindings of the SEL-SECONDARY-SUB-TREE is not changed in step 13. It must be mentioned that the sub-tree 14 also instantiates `?x` to `W`. Since the two sub-trees bind `?x` to the same value `W`, this situation is valid. In step 14, no rule can be satisfied by the sub-trees in the variable SIBLINGS and the SEL-SECONDARY-SUB-TREE. Thus, the two sub-trees 6 and 19 cannot be mated. The reason is that the same logic variable `?x` must be instantiated to different values `Z` and `W`: the sub-tree 19 requires the variable `?x` to be instantiated to `W` while `?x` must be instantiated to `Z` in the context of the primary parent. The function `is-valid` can determine this situation and avoid the crossover algorithm from generating an offspring.

The crossover algorithm guarantees that only valid offspring can be produced and this operation can be achieved effectively. It has been estimated that the worst case time complexity of the crossover algorithm is $O(N_p*N_s*D_p)$, where $N_p$ and $N_s$ are respectively the numbers of sub-trees in the global variables PRIMARY-SUB-TREES and SECONDARY-SUB-TREES, $D_p$ is the depth of the primary derivation tree (Depth starts from 0).

Since the computation time consumed by performing crossover is insignificant when compare with the time used in evaluating the fitness of each program in the population. The issue of computational complexities of various crossover algorithms has not been addressed by other researchers in the field of Genetic Programming. In fact, it is easy to calculate that the worst case time complexity of the structure-preserving crossover algorithm of GP (Koza 1994) is $O(N_{p1}*N_{p2})$, where $N_{p1}$ and $N_{p2}$ are respectively the sizes of the parental parse trees. Similarly, the crossover algorithm of STGP (Montana 1995) has the same complexity. Although the crossover algorithm of LOGENPRO is slightly slower than other algorithms by $O(D_p)$, it is more general and powerful than other algorithms.

## Appendix C.

The algorithm in table 12 is used to produce an offspring program by mutation. For example, assume that the program being mutated is `(+ (- Z 3.5) (- Z 3.8) (/ Z 1.5))` and the corresponding derivation tree is depicted in figure 3. In step 1 of the mutation algorithm, the global variable SUB-TREES contains the sub-trees 0, 3, and 6. The frozen sub-trees 9, 10, 11, and 12 are excluded. The sub-trees 1, 4, and 7 are also excluded because they contain logic goals of the grammar and thus should not be modified by genetic operations. The sub-trees 2, 5, and 8 containing terminal symbols are eliminated for two reasons. First, the mutation algorithm is significantly simplified if terminal symbol need not be modified. Second, the effect of mutating terminal symbols can be emulated by the crossover operation. Recalling the example described in Appendix B, the primary sub-tree 2 are crossed with the secondary sub-tree 15 to generate the offspring `(* (- Z 3.5) (- Z 3.8) (/ Z 1.5))`. This offspring can be seen as the result of mutating the terminal symbol `[(+]` to the `[(*]`.

In step 2, a sub-tree in the variable SUB-TREES is selected randomly using a uniform distribution if the SUB-TREES is not empty. Otherwise, the mutation algorithm terminates without generating any modified program because no valid mutation can be found. In normal situation, this should not occur because it is almost always possible to select the whole derivation tree as the one to be mutated. The whole tree cannot be chosen only if it is frozen. The effect of mutating the whole tree, the sub-tree 0 in this example, is equivalent to generating a new program from scratch. A new program can be created successfully if the language specified by the grammar contains at least one program (this must be true for a grammar to be useful). Thus, the algorithm will fail to find a mutation only if the whole derivation tree is frozen.

Assume that the sub-tree 3 is selected as the MUTATED-SUB-TREE in step 2. In the next step, the sub-tree 3 is removed from the variable SUB-TREES. The NON-TERMINAL and the ARGS are `exp-1(?x)` and `{?x}` respectively. Since the logic variable `?x` is instantiated to `Z` in the sub-tree 1 by the logic goal `member(?x, [W, Z])`, the bindings `{?x/Z}` are stored into the variable NEW-BINDINGS in step 4.

In step 5, the new non-terminal NEW-NON-TERMINAL `exp-1(Z)` is created. Using this mechanism, contextual-dependent information can be exchanged amongst different parts of a program. In step 6, a new derivation tree for the S-expression `(/ Z 1.9)` can be obtained from the non-terminal symbol `exp-1(Z)` using the fifth rule of the grammar. This derivation tree is displayed in figure 7.

Since the NEW-SUB-TREE can be found, a new offspring is obtained by duplicating the genetic materials of its parental derivation tree, followed by deleting the MUTATED-SUB-TREE from the duplication, and then pasting the NEW-SUB-TREE at the MUTATE-POINT. The derivation tree of the offspring `(+ (/ Z 1.9) (- Z 3.8) (/ Z 1.5))` can be found in figure 8.

# TABLES

```
1:    start       ->    [(*], exp(W), exp(W), exp(W), [)].
2:    start       ->    {member(?x,[W, Z])}, [(*], exp-1(?x),
                        exp-1(?x), exp-1(?x), [)].
3:    start       ->    {member(?x,[W, Z])}, [(+], exp-1(?x),
                        exp-1(?x), exp-1(?x), [)].
4:    exp(?x)     ->    [(/ ?x 1.5)].
5:    exp-1(?x)   ->    {random(1,2,?y)}, [(/ ?x ?y)].
6:    exp-1(?x)   ->    {random(3,4,?y)}, [(- ?x ?y)].
7:    exp-1(W)    ->    [(+ (- W 11) 12)].
```

**Table 1: A logic grammar**

```
10:   start       ->    s-expr(number).
11:   s-expr([list, number, ?n]) ->
                        [ (mapcar (function ] ,op2, [ ) ] ,
                        s-expr([list, number, ?n]),
                        s-expr([list, number, ?n]),[)].
12:   s-expr([list, number, ?n]) ->
                        [ (mapcar (function ], op1, [ ) ] ,
                        s-expr([list, number, ?n]),[)].
13:   s-expr([list, number, ?n]) -> term([list, number, ?n]).
14:   s-expr(number)    ->    term(number).
15:   s-expr(number)    ->    [(apply (function ], op2, [)] ,
                              s-expr([list, number, ?n]),[)].
16:   s-expr(number)    ->    [ ( ], op2, s-expr(number),
                              s-expr(number), [ ) ].
17:   s-expr(number)    ->    [ ( ],op1,s-expr(number), [)].
18:   op2               ->    [ + ].
19:   op2               ->    [ - ].
20:   op2               ->    [ * ].
21:   op2               ->    [ % ].
22:   op1               ->    [ protected-log ].
23:   term( [list, number, n] )     ->    X.
24:   term( [list, number, n] )     ->    Y.
25:   term( number )    ->    { random(-10, 10, ?a) }, [?a ].
```

**Table 2: The logic grammar for the Dot Product problem**

```
start                        ->    [(progn (defun ADF0 (arg0 arg1)],
                                   s-expr2(number), [)],
                                   s-expr(number), [)].
s-expr([list, number, ?n])->  [ (mapcar (function ], op2, [ ) ] ,
                                   s-expr([list, number, ?n]),
                                   s-expr([list, number, ?n]),[ ) ].
s-expr([list, number, ?n])->  term([list, number, ?n]).
s-expr(number)               ->    [ (apply (function ], op2, [ ) ] ,
                                   s-expr([list, number, ?n]),[ ) ].
s-expr(number)               ->    [ ( ], op2, s-expr(number),
                                   s-expr(number), [ ) ].
s-expr(number)               ->    [ (ADF0 ], s-expr([list, number, ?n]),
                                   s-expr([list, number, ?n]), [ ) ].
term([list, number, n]) ->    X.
term([list, number, n]) ->    Y.
term([list, number, n]) ->    Z.
s-expr2([list, number, ?n])-> [ (mapcar (function ], op2, [ ) ],
                                   s-expr2([list, number, ?n]),
                                   s-expr2([list, number, ?n]),[ ) ].
s-expr2([list, number, ?n])-> term2([list, number, ?n]).
s-expr2(number)              ->    [ (apply (function ], op2, [ ) ] ,
                                   s-expr2([list, number, ?n]),[ ) ].
s-expr2(number)              ->    [ ( ], op2, s-expr2(number),
                                   s-expr2(number), [ ) ].
term2([list, number, n])->    arg0.
term2([list, number, n])->    arg1.
op2                          ->    [ + ].
op2                          ->    [ - ].
op2                          ->    [ * ].
```

**Table 3: Logic grammar for the sub-function problem**

```
(defun parity (L)
  (if (null L) T
    (AND (OR (first L) (parity (rest L)))
    (NAND (first L) (parity (rest L))))))
```

**Table 4: A recursive function for the even-n-parity problem**

| | Arity | Data types of input arguments | Data type of the output value |
|---|---|---|---|
| AND | 2 | BOOLEAN BOOLEAN | BOOLEAN |
| OR | 2 | BOOLEAN BOOLEAN | BOOLEAN |
| NAND | 2 | BOOLEAN BOOLEAN | BOOLEAN |
| NOR | 2 | BOOLEAN BOOLEAN | BOOLEAN |
| ifnil | 3 | LIST BOOLEAN BOOLEAN | BOOLEAN |
| first | 1 | LIST | BOOLEAN |
| rest | 1 | LIST | SMALLER-LIST |
| parity | 1 | LIST | BOOLEAN |

**Table 5: The arities, the data types of the input arguments, and the data types of the output value of all primitive functions**

```
31:   start                 ->   [ (defun parity (L) ],
                                 [ (ifnil L T ],
                                 s-expr(boolean), [ )) ].
32:   s-expr(BOOLEAN)       ->   [ T ].
33:   s-expr(BOOLEAN)       ->   [ nil ].
34:   s-expr(BOOLEAN)       ->   [ ( ], op, s-expr(BOOLEAN),
                                 s-expr(BOOLEAN), [ ) ].
35:   s-expr(BOOLEAN)       ->   [ ( ], [ parity ],
                                 s-expr(SMALLER-LIST), [ ) ].
36:   s-expr(BOOLEAN)       ->   [ ( ], [ first ],
                                 s-expr(LIST), [ ) ].
37:   s-expr(LIST)          ->   [ L ].
38:   s-expr(LIST)          ->   s-expr(SMALLER-LIST).
39:   s-expr(SMALLER-LIST)->   [ ( ],[ rest ], s-expr(LIST),[)].
40:   op                    ->   [ AND ].
41:   op                    ->   [ OR ].
42:   op                    ->   [ NAND ].
43:   op                    ->   [ NOR ].
```

**Table 6: The grammar rules of Grammar_1**

| | LOGENPRO_1 | LOGENPRO_2 | LOGENPRO_3 |
|---|---|---|---|
| The 1st series of experiments | 4355000 | 3737500 | 16536000 |
| The 2nd series of experiments | 5850000 | 15795000 | 32175000 |

**Table 7: The numbers of fitness cases processed to induce general recursive functions with 99% probability**

```
1':   start(tree(start, [(*], ?E1, ?E2, frozen(?E3), |)|), ?S0, ?S5)
            :-    connect([(*], ?S0, ?S1), exp(W, ?E1, ?S1, ?S2),
                  exp(W, ?E2, ?S2, ?S3), exp(W, ?E3, ?S3, ?S4),
                  connect([)], ?S4, ?S5).

2':   start(tree(start, {member(?x, [W, Z])}, [(*], ?E1, ?E2,
                  frozen(?E3), |)|),?S0, ?S5)
            :-    member(?x, [W, Z]), connect([(*], ?S0, ?S1),
                  exp-1(?x, ?E1, ?S1, ?S2), exp-1(?x, ?E2, ?S2, ?S3),
                  exp-1(?x, ?E3, ?S3, ?S4), connect([)], ?S4, ?S5).

3':   start(tree(start, {member(?x, [W, Z])}, [(+], ?E1, ?E2,
                  frozen(?E3), |)|),?S0, ?S5)
            :-    member(?x, [W, Z]), connect([(+], ?S0, ?S1),
                  exp-1(?x, ?E1, ?S1, ?S2), exp-1(?x, ?E2, ?S2, ?S3),
                  exp-1(?x, ?E3, ?S3, ?S4), connect([)], ?S4, ?S5).

4':   exp(?x, tree(exp(?x), [(/ ?x 1.5)]),?S0, ?S1)
            :-    connect([(/ ?x 1.5)], ?S0, ?S1).

5':   exp-1(?x, tree(exp-1(?x), {random(1,2,?y)}, [(/ ?x ?y)]),?S0,
                     ?S1)
            :-    random(1, 2, ?y), connect([(/ ?x ?y)], ?S0, ?S1).

6':   exp-1(?x, tree(exp-1(?x), {random(3,4,?y)}, [(- ?x ?y)]),?S0,
                     ?S1)
            :-    random(3, 4, ?y), connect([(- ?x ?y)], ?S0, ?S1).

7':   exp-1(W, tree(exp-1(W), [(+ (- W 11) 12)]),?S0, ?S1)
            :-    connect([(+ (- W 11) 12)], ?S0, ?S1).
```

**Table 8: A logic program obtained from translating the logic grammar presented in table 1**

```
Input:
   P:              The primary derivation tree.
   S:              The secondary derivation tree.

Output:
   Return a new derivation tree if a valid offspring can be obtained
   by performing crossover, otherwise return false.

Function crossover(P, S)
{

      1.    Find all sub-trees of the primary derivation tree P and
            store them into a global variable PRIMARY-SUB-TREES,
            excluding the primary derivation tree, all logic goals,
            and frozen sub-trees.
      2.    Find all sub-trees of the secondary derivation tree S and
            store them into a global variable SECONDARY-SUB-TREES,
            excluding all logic goals and frozen sub-trees.
      3.    If  the variable PRIMARY-SUB-TREES is not empty, select
            randomly a sub-tree from it using a uniform distribution.
            Otherwise, terminate the algorithm without generating any
            offspring program.
      4.    Designate the sub-tree selected as the SEL-PRIMARY-SUB-
            TREE and the root of it as the PRIMARY-CROSSOVER-POINT.
            Remove the SEL-PRIMARY-SUB-TREE from the variable
            PRIMARY-SUB-TREES.
      5.    Copy the variable SECONDARY-SUB-TREES to the temporary
            variable TEMP-SECONDARY-SUB-TREES.
      6     If the variable TEMP-SECONDARY-SUB-TREES is not empty,
            select randomly a sub-tree from it using a uniform
            distribution. Otherwise, go to step 3.
      7.    Designate the sub-tree selected in step 6 as the SEL-
            SECONDARY-SUB-TREE. Remove it from the variable TEMP-
            SECONDARY-SUB-TREES.
      8.    If the offspring produced by performing crossover between
            the SEL-PRIMARY-SUB-TREE and the SEL-SECONDARY-SUB-TREE
            is invalid according to the grammar or the depth of it
            exceeds the maximum depth of trees produced by crossover,
            go to step 6. The validity of the offspring generated can
            be checked by the procedure is-valid(P, SEL-PRIMARY-SUB-
            TREE, SEL-SECONDARY-SUB-TREE).
      9.    Copy the genetic materials of the primary parent P to the
            offspring, remove the SEL-PRIMARY-SUB-TREE from it and
            then impregnating a copy of the SEL-SECONDARY-SUB-TREE at
            the PRIMARY-CROSSOVER-POINT.
      10.   Perform some house-keeping tasks and return the
            offspring program.
}
```

**Table 9: The crossover algorithm of LOGENPRO**

```
Input:
   P:              The primary derivation tree
   P-sub-tree:     The sub-tree in the primary derivation tree that is
                   selected to be crossed over.
   S-sub-tree:     The sub-tree in the secondary derivation tree that
                   is selected to be crossed over.

Output:
   Return true if the offspring generated is valid, otherwise return
   false.

Function is-valid(P, P-sub-tree, S-sub-tree)
{
      11.   Find all siblings of the P-sub-tree in P and store them
            into the global variable SIBLINGS.
      12.   For each sub-tree in the variable SIBLINGS, perform the
            following sub-steps:
               • Store the bindings of the sub-tree to the global
                 variable BINDINGS.
               • For each logic variable in the variable BINDINGS
                 that is not instantiated by the sub-tree,  remove
                 it from the variable BINDINGS.
               • Modify the bindings of the sub-tree.
      13.   Modify the bindings of the S-sub-tree. A logic variable
            is retained only if it is instantiated in the S-sub-tree.
      14.   If there is a rule in the grammar such that:
               • it is satisfied by the sub-trees in the variable
                 SIBLINGS and the S-sub-tree,
               • the sub-trees in the variable SIBLINGS and the S-
                 sub-tree are used exactly once,
               • the sub-trees are applied in the same order as that
                 in the original rule of the primary derivation
                 tree, and
               • a consistent conclusion C is deduced from the rule.
                 The conclusion is consistent if the function
                 is-consistent(P, PARENT, C) returns true where
                 PARENT is the parent of the P-sub-tree. The
                 function is-consistent is presented in table 11.
                 then the offspring generated will be valid.
            Otherwise, the offspring will be invalid.
}
```

**Table 10: The algorithm that checks whether the offspring produced by LOGENPRO is valid.**

```
Input:
   P:        The primary derivation tree.
   PARENT:   The parent of the primary sub-tree.
   C:        The conclusion.

Output:
   Return true if the conclusion C is consistent, otherwise return false.

Comment:
   This operation can be viewed as performing a tentative crossover between
   PARENT and C and then determining whether the tentative offspring
   produced is valid. Here, PARENT is treated as the primary sub-tree while
   C is treated as the secondary sub-tree of the tentative crossover
   operation. The main difference between this algorithm and that in table
   10 is that all rule applications in all ancestors of PARENT must be
   maintained.

Function is-consistent(P, PARENT, C)
{
      15.   If PARENT is the root of P then
                  if C is labeled with the symbol start then
                        return true
                  else false.
      16.   Find all siblings of PARENT in P and store them into the global
            variable SIBLINGS.
      17.   For each sub-tree in the variable SIBLINGS, perform the
            following sub-steps:
               • Store the bindings of the sub-tree to the global
                 variable BINDINGS.
               • For each logic variable in the variable BINDINGS that is
                 not instantiated by the sub-tree,  remove it from the
                 variable BINDINGS.
               • Modify the bindings of the sub-tree.
      18.   Let the grammar rule applied in the parent node of PARENT as
            RULE.
            If the following conditions are satisfied:
               • RULE is satisfied by the sub-trees in the variable
                 SIBLINGS and C,
               • the sub-trees in SIBLINGS and C are used exactly once
                 and the ordering of applications is maintained, and
               • a consistent conclusion C' is deduced from RULE. The
                 conclusion is consistent if the function
                 is-consistent(P, GRANDPARENT, C') returns true where
                 GRANDPARENT is the parent node of PARENT.
                 then
                        return true
                 else
                        return false.
}
```

**Table 11: The algorithm that checks whether a conclusion deduced from a rule is consistent with the direct parent of the primary sub-tree.**

```
Input:
   P:       The derivation tree of the parental program

Output:
   Return a new derivation tree if a valid offspring can be obtained
   by performing mutation, otherwise return false.

Function mutation(P)
{
      1.    Find all sub-trees of the derivation tree P of the
            parental program and store them into a global variable
            SUB-TREES, excluding all frozen sub-trees, logic goals,
            and terminal symbols
      2.    If SUB-TREES is not empty, select randomly a sub-tree
            from the SUB-TREES using a uniform distribution.
            Otherwise, terminate the algorithm without generating any
            offspring.
      3.    Designate the sub-tree selected as MUTATED-SUB-TREE. The
            root of the MUTATED-SUB-TREE is called the MUTATE-POINT.
            Remove the MUTATED-SUB-TREE from the variable SUB-TREES.
            The MUTATED-SUB-TREE must be generated from a non-
            terminal symbol of the grammar. Designate this non-
            terminal symbol as NON-TERMINAL. The NON-TERMINAL may
            have a list of arguments called ARGS.
      4.    For each argument in the ARGS, if it contains some logic
            variables, determine whether these variables are
            instantiated by other constituent of the derivation tree.
            If they are, bind the instantiated value to the variable.
            Otherwise, the variable is unbounded. Store the modified
            bindings to a global variable NEW-BINDINGS.
      5.    Create a new non-terminal symbol NEW-NON-TERMINAL from
            the NON-TERMINAL and the bindings in the variable NEW-
            BINDINGS.
      6.    Try to generate a new derivation tree NEW-SUB-TREE from
            the NEW-NON-TERMINAL using the deduction mechanism
            provided by LOGENPRO. The depth of the new derivation
            tree NEW-SUB-TREE is restricted so that the depth of the
            mutated derivation tree created in step 7 does not exceed
            the maximum depth of trees produced by mutation.
      7.    If a new derivation tree can be successfully created, the
            offspring is obtained by deleting the MUTATED-SUB-TREE
            from a copy of the parental derivation tree P and then
            impregnating the NEW-SUB-TREE at the MUTATE-POINT.
            Otherwise, go to step 3.
}
```

**Table 12: The mutation algorithm**

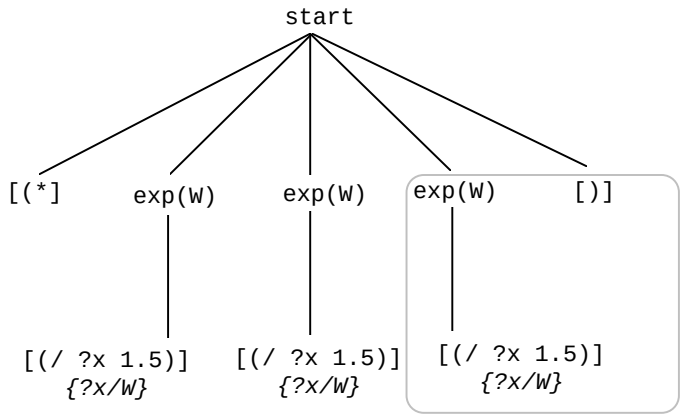



**Figure 1: A derivation tree of the S-expression in Lisp `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))`**

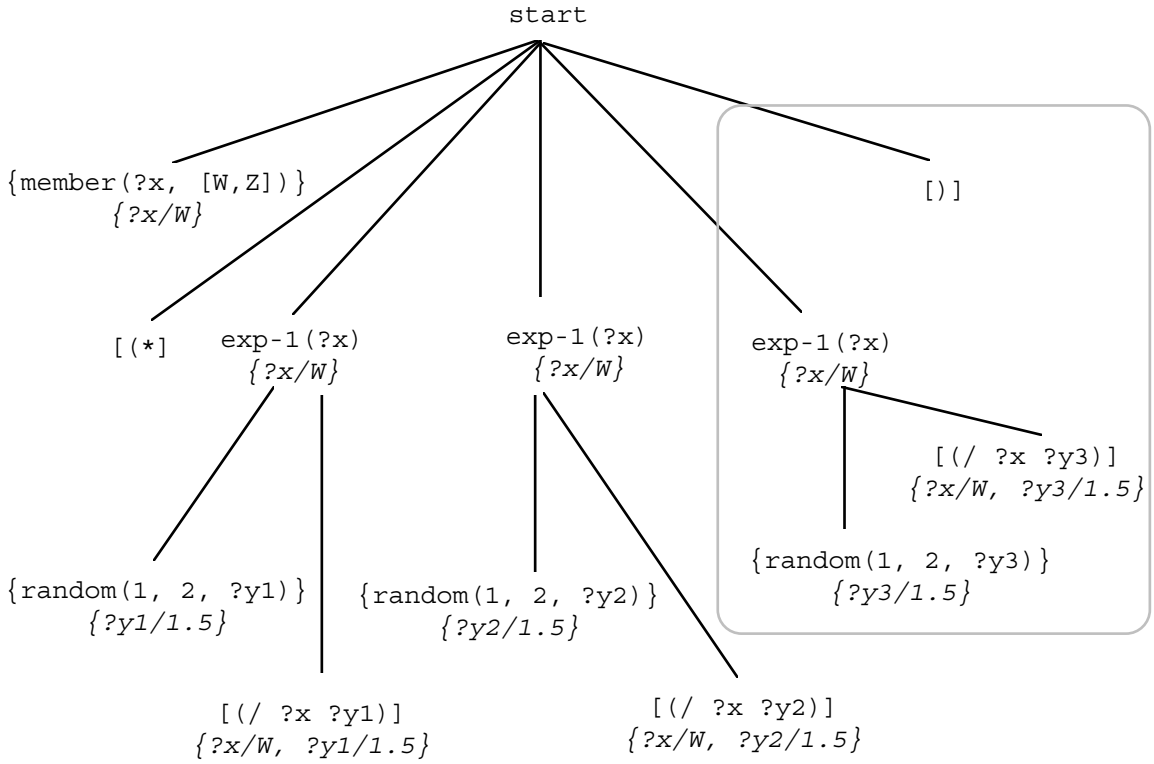



**Figure 2: Another derivation tree of the S-expression in Lisp `(* (/ W 1.5) (/ W 1.5) (/ W 1.5))`**

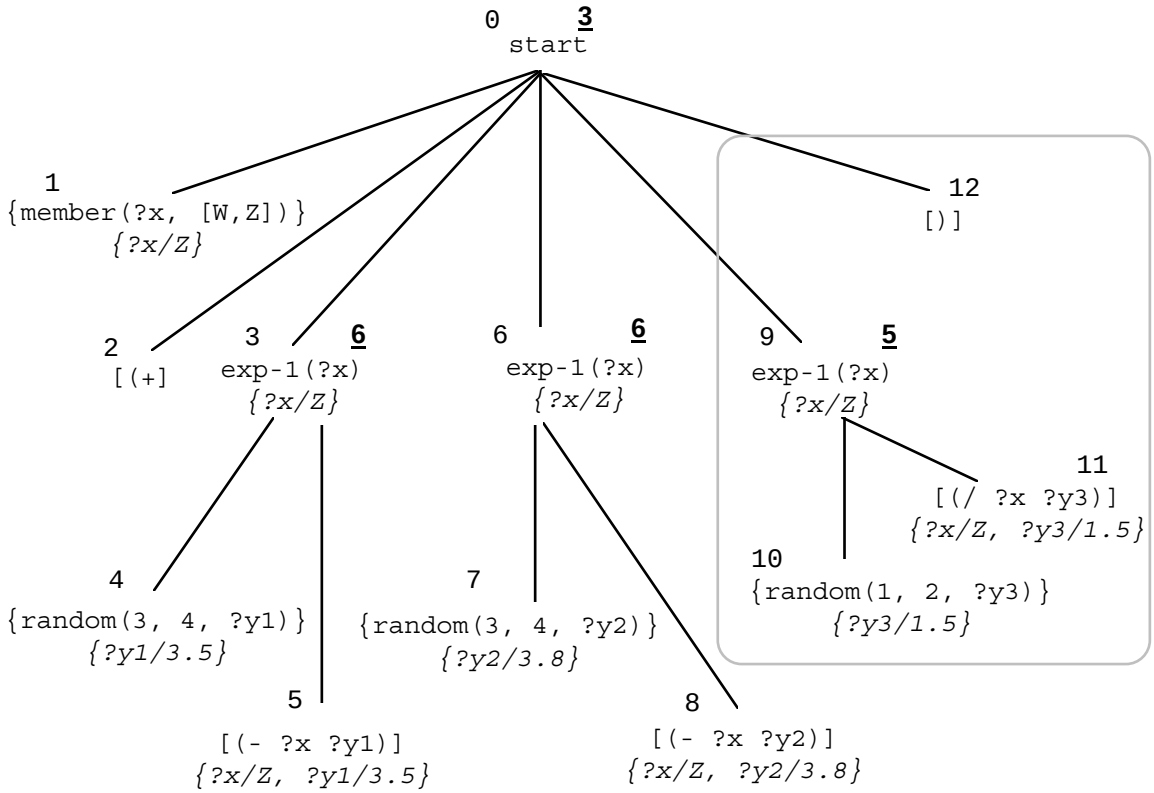



**Figure 3: The derivations tree of the primary parental program (+ (- Z 3.5) (- Z 3.8) (/ Z 1.5)).**

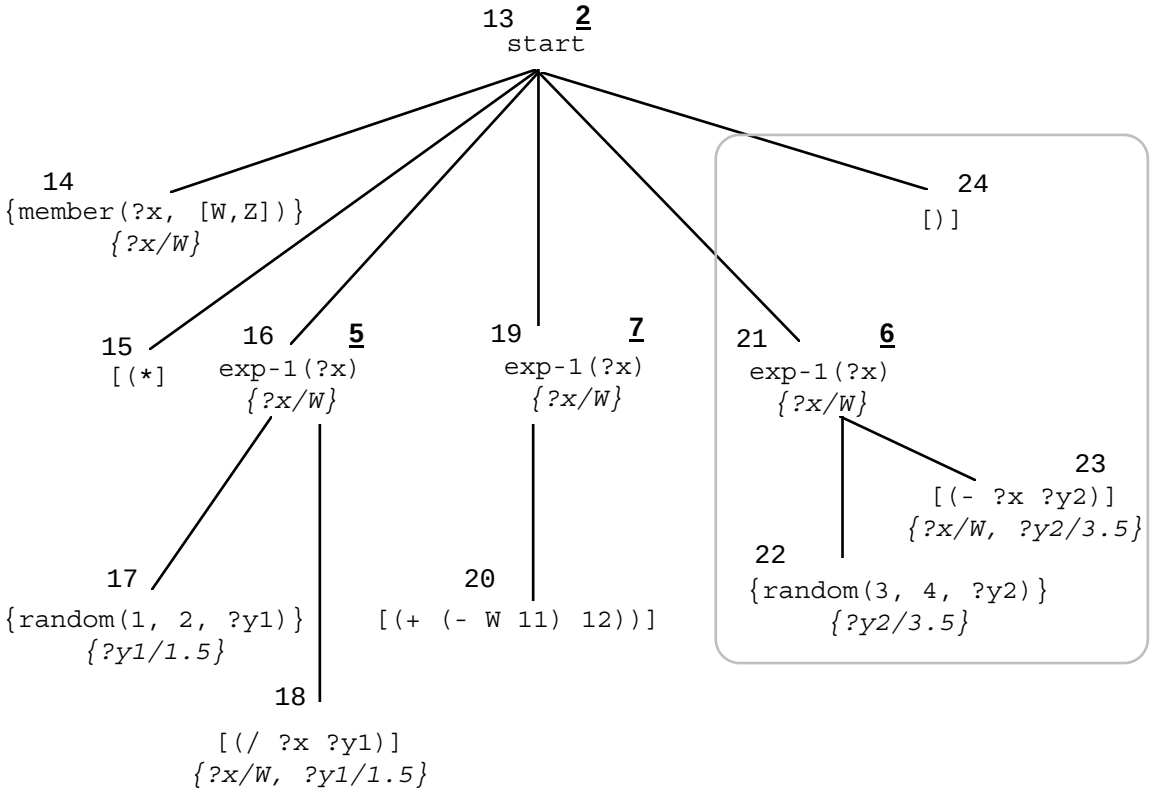



**Figure 4:** **The derivations tree of the secondary parental program `(* (/ W 1.5) (+ (- W 11) 12) (- W 3.5))`.**

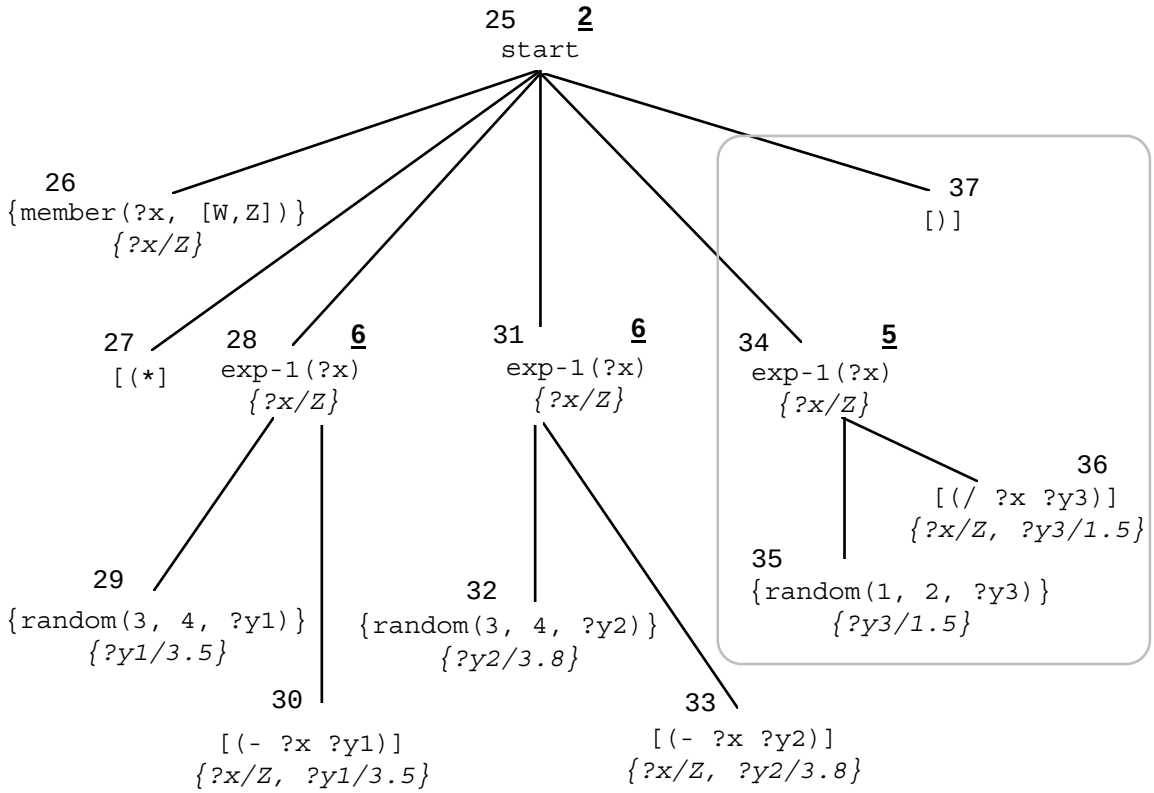



**Figure 5:** **A derivation tree of the offspring produced by performing crossover between the primary sub-tree 2 of the tree in figure 3 and the secondary sub-tree 15 of the tree in figure 4.**

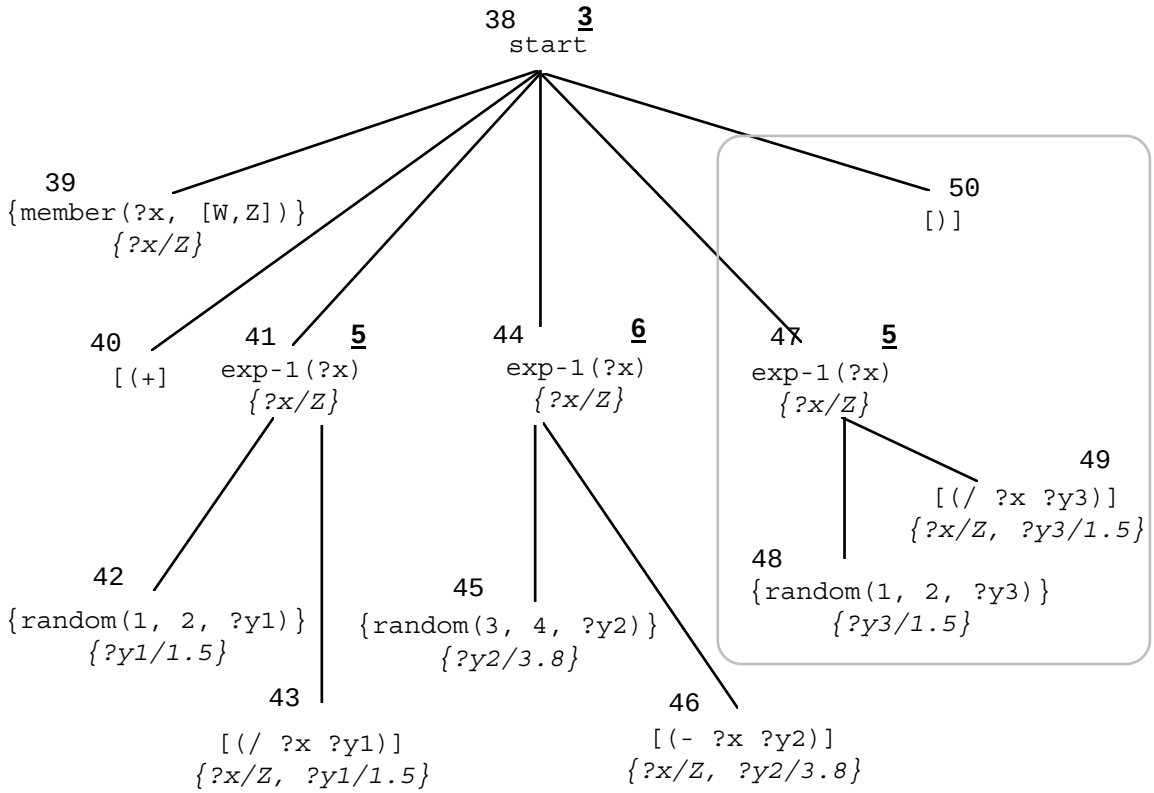



**Figure 6:** **A derivation tree of the offspring produced by performing crossover between the primary sub-tree 3 of the tree in figure 3 and the secondary sub-tree 16 of the tree in figure 4.**

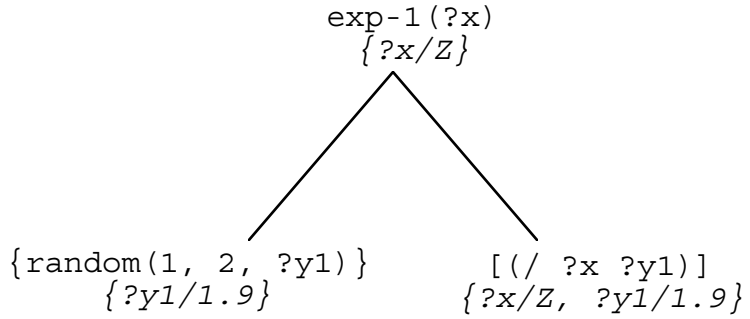



**Figure 7:** **A derivation tree generated from the non-terminal `exp-1(Z)`**

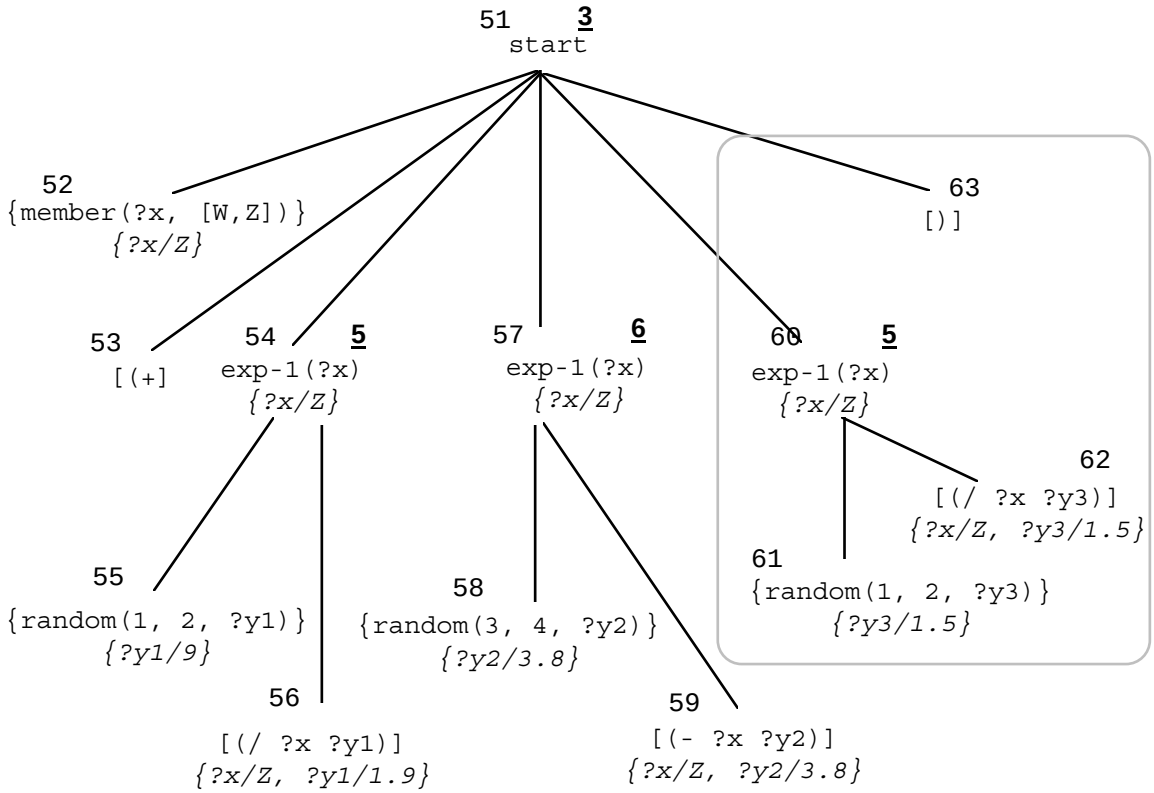



**Figure 8:** **A derivation tree of the offspring produced by performing mutation of the tree in figure 3 at the sub-tree 3**

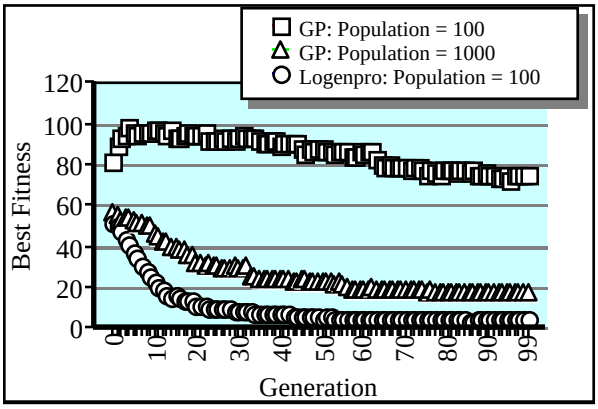



**Figure 9:** **Fitness curves showing best fitness for the Dot Product problem**

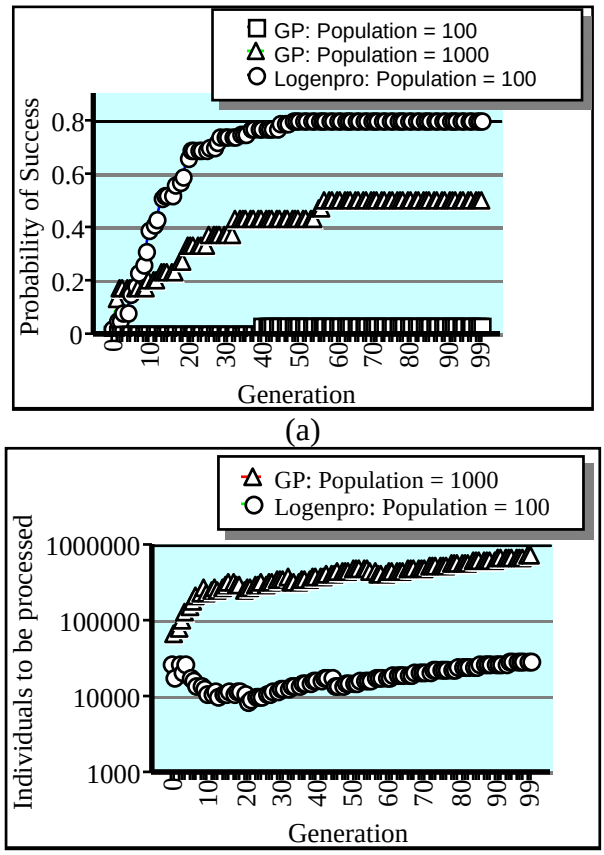



**Figure 10:** **The performance curves showing (a) cumulative probability of success P(M, i) and (b) I(M, i, z) for the DOT PRODUCT problem.**

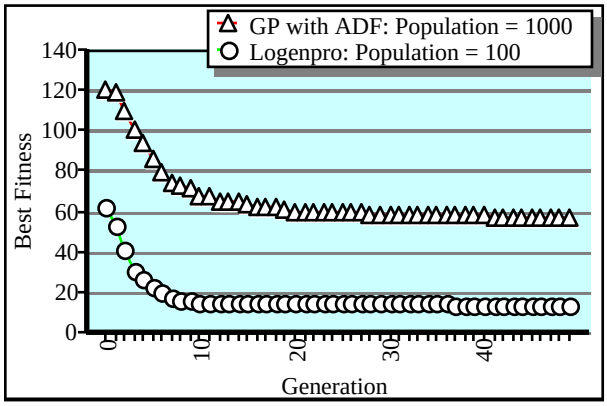
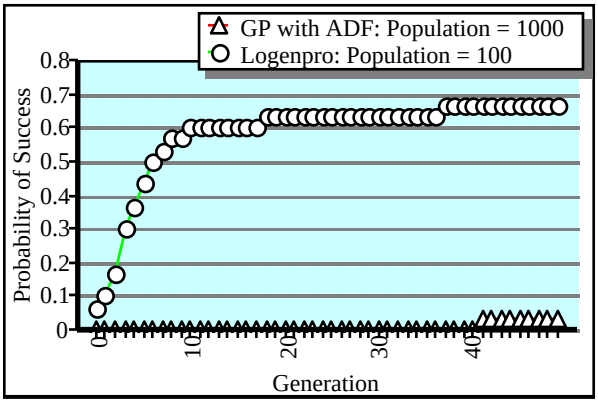
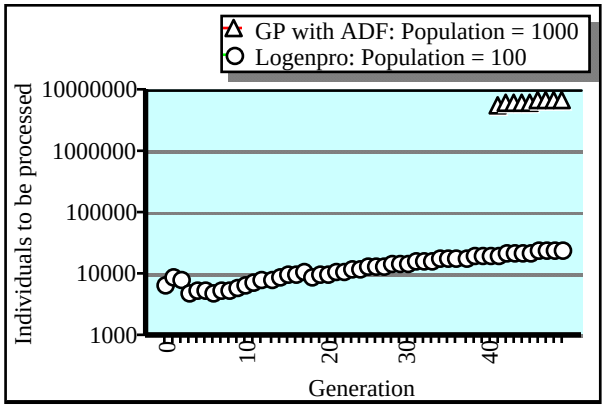



**Figure 11: Fitness curves showing best fitness for the sub-function problem**




(a)




(b)

**Figure 12: Performance curves showing (a) cumulative probability of success P(M, i) and (b) I(M, i, z) for the sub-function problem**