GENETIC PROGRAMMING WITH CONTEXT-SENSITIVE GRAMMARS

Norman Paterson

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews

2003

Genetic programming with context-sensitive grammars

Thesis submitted for the degree of Doctor of Philosophy

by

Norman Paterson

School of Computer Science

University of St Andrews

Scotland

September 2002

Th
E 378

Dedication

To my parents

John Campbell Paterson, 1908 — 1975

Marcelle Berthe Hütler, 1910 — 2000

Abstract

This thesis presents Genetic Algorithm for Deriving Software (Gads), a new technique for genetic programming. Gads combines a conventional genetic algorithm with a context-sensitive grammar. The key to Gads is the ontogenic mapping, which converts a genome from an array of integers to a correctly typed program in the phenotype language defined by the grammar. A new type of grammar, the reflective attribute grammar (rag), is introduced. The rag is an extension of the conventional attribute grammar, which is designed to produce valid sentences, not to recognise or parse them. Together, Gads and rags provide a scalable solution for evolving type-correct software in independently-chosen context-sensitive languages. The statistics of performance comparison is investigated. A method for representing a set of genetic programming systems or problems on a cladogram is presented. A method for comparing genetic programming systems or problems on a single rational scale is proposed.

Declaration

(i) I, Norman Paterson, hereby certify that this thesis, which is approximately 70 000 words in length, has been written by me, that it is the record of work carried out by me and that it has not been submitted in any previous application for a higher degree.

date 24.4.2003 *signature of candidate* [illegible]

(ii) I was admitted as a research student in October 1995 and as a candidate for the degree of Doctor of Philosophy in October 1995; the higher study for which this is a record was carried out in the University of St Andrews between 1995 and 2002.

date 24.4.2003 *signature of candidate* [illegible]

(iii) I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

date 24.4.2003 *signature of supervisor* [illegible]

Copyright

date 24·4·2003 *signature of candidate* ______________

Acknowledgements

Many people deserve acknowledgement for the help they gave in the execution of this work. Michael Livesey, as supervisor, was always able to show interest and listen to my endless and sometimes pointless thought processes. Without his support, this work would quite simply not have been possible. Ishbel Duncan offered to proofread: not a task to be undertaken lightly.

The staff and students of the School of Computer Science unwittingly gave support in that some of the time I spent on this part-time work can only have come from the time I should have spent on my full-time job as Senior Scientific Officer, which must have suffered in consequence.

Conor Ryan and Michael O'Neil from the University of Limerick showed that the original ideas had some merit, and their friendship helped me to feel less alone in a large world.

My wife and friend, Margaret McCabe, encouraged me relentlessly.

Acronyms & notations

But the scene of this narrative is laid in the South of England and takes place in and around Knotacentinum Towers (pronounced as if written Nosham Taws), the seat of Lord Knotacent (pronounced as if written Nosh).

But it is not necessary to pronounce either of these names in reading them.

Nonsense Novels, Stephen Leacock, 1911.

The following acronyms and notations are used without further elaboration throughout this thesis. I have tried to adopt the rule that acronym letters are named individually if in upper case, and pronounced phonetically if in lower case.

α

Probability of a type 1 error; level of significance.

δ

Least d which can be detected 95% reliably.

ADF

Automatically defined function. A function whose definition is evolved automatically, and which may be used by other parts of the individual it is part of.

anova

Analysis of variance.

BNF

Backus-Naur form or Backus Normal form.

CFG

Context free grammar.

CSG

Context sensitive grammar.

d

Displacement; difference in means of two simulated populations.

ECJ

A Java-based evolutionary computation and GP research system [Luke, 2001].

ES

Evolution strategy.

EPF

Empirical (or estimated) power function.

ET

Expression tree. A tree showing an expression such as a Lisp S-expression. All nodes are terminal symbols. Internal nodes have arity greater than 0. Leaf nodes have arity equal to 0. Contrast with PT.

GA

Genetic algorithm.

Gads

Genetic algorithm for deriving software. Two editions of Gads are presented in this thesis. Gads 1 was previously described in [Paterson, 1996]. Gads 2 is described in this thesis for the first time. When neither edition is specified, *Gads* is a more general reference to any-edition in the series.

GE

Grammatical Evolution, a GP system desribed in [Ryan, 1998a]. Historically, GE lies between Gads 1 and Gads 2.

GCL

Global confidence level, 95%.

GEP

Gene Expression Programming [Ferreira, 2001].

GP

Genetic programming.

LHS

Left hand side (of a BNF production).

PT

Parse tree. A tree showing the derivation or structure of a sentence according to a grammar. Internal nodes are nonterminal symbols and leaf nodes are terminal symbols. Also called *derivation tree* or *syntax tree*. Contrast with ET.

rag

Reflective attribute grammar.

RHS

Right hand side (of a BNF production).

RMS

Root mean square.

RNG

Random number generator.

SA

Simulated annealing.

sag

Standard (ie non-reflective) attribute grammar.

SGP

Standard (ie Koza-style) genetic programming.

STGP

Strongly Typed GP [Montana, 1995] and [Clack, 1997].

V

Coefficient of variation = standard deviation ÷ mean.

Contents

1 Introduction

Genetic programming (GP) is a method for automatically developing computer programs. *Automatic* means that programs are produced by specifying what they are to do, not how they are to do it. For example, a function *f* which computes the square root of its argument may be specified by

$$(f(x))^2 = x$$

This describes what *f* does, but it says nothing about how it might do it.

The current explosion of interest in GP was triggered by the publication of *Genetic Programming: on the programming of computers by means of natural selection* [Koza, 1992], and most work in GP since then has been a development of Koza's ideas. Koza's approach is the GP mainstream, and as the *de facto* standard it is sometimes referred to as standard genetic programming (SGP).

The remaining subsections of this introduction summarise this thesis and place it the context of other GP work, as follows:

§1.1 Outline of GP
Presents a general description of GP.

§1.2 Outline of Genetic Algorithm for Deriving Software (Gads)
Introduces the main topic of this thesis: Gads.

§1.3 Related work
Places this thesis in context of current GP systems.

§1.4 Motivation
Explains the aim of this work.

§1.5 Contribution
Summarises the main contributions of this thesis.

§1.6 Reading guide
Summarises the main sections of this thesis.

1.1 Outline of GP

GP is a child of the genetic algorithm (GA). The seminal work on GAs is *Adaptation in natural and artificial systems* [Holland, 1992]. In outline, GAs work as follows.

1 Initialisation
Generate an initial population at random.

2 Evaluation
Compute the fitness of each individual in the population, which is a measure of how well it meets the requirements.

3 Breeding
Breed a new population, favouring fitter individuals as parents.

4 Termination
If the population contains an individual which meets the requirements, or if some other limit is reached, then stop. Otherwise continue from step 2.

For example, suppose we wish to design a roof truss in the form of a triangle with 2 braces on each side:

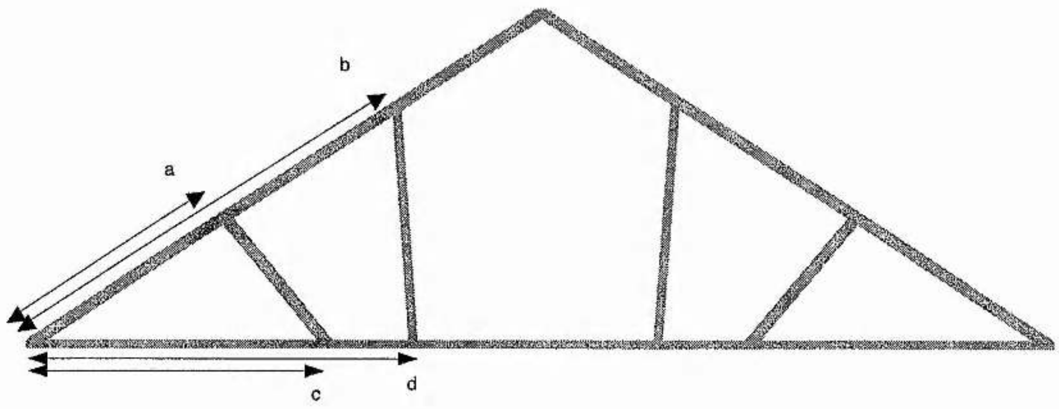

Figure 1-1: Roof truss

We can reasonably expect the truss to be symmetrical, so for a given height and width there are 4 variables, shown in the figure as *a*, *b*, *c* and *d*. These are real values within a limited range. We can thus represent any individual design by a real array of length 4, and an initial population of designs could easily be generated with a random number generator (RNG). An array of 4 reals permits designs where the braces cross each other or cross the centre of the truss. If we want to exclude these or similar possibilities, then a different representation might be necessary.

The fitness of an individual design is computed by means of a *fitness function.* The fitness function is problem-specific, and is essentially a definition of the problem to be solved. Continuing with the truss example, the fitness function could be a procedure which takes the 4 real numbers as arguments and computes properties such as cost, weight, or strength. Which properties are computed depends on the aim of the exercise. The result is a scalar value representing how good the individual is. The individuals in the initial population can be expected to be of poor quality, but they will not all be equally poor. By chance, some will be better than others.

Parents are chosen to breed a new generation in such a way that fitter parents are more likely to be selected. The crossover technique is used to produce children. Each parent is split in two at some position along its length, and a child is formed by joining the first part of one parent with the second part of the other. By joining the other two parts, a second child can be produced at the same time for little extra effort. If the representation is fixed length, then both parents must be split at the same position. Various modifications of this simple form of crossover, and various other genetic operations such as mutation, are used. With the truss example, since a parent consists of only 4 reals, there are only 3 internal crossover points.

Notice that GA does not guarantee to find a solution or stop within any specified number of steps. In practice it continues until a satisfactory solution is found, or some resource is exhausted.

For evolution to work, two things are necessary: the fitness of the children must correlate with the fitness of the parents, and there must be variation in the children's fitness. These two conditions result in a situation where children sometimes outperform their parents. In this environment, selective breeding acts like a ratchet, evolving ever fitter individuals.

There are countless possible variations to each step. For example, the data type of individuals is not limited to one-dimensional arrays; multi-dimensional arrays, graphs and other data types have been used [Michalewicz, 1994]. Fitness may be a single scalar value, or there may be multiple objectives. There may be multiple subpopulations, with migration from one subpopulation to another. There may be different species in competition with each other. The best individuals may be guaranteed a place in the new population (elitism). Crossover need not be limited to word boundaries. A degree of random mutation may be introduced. The probability of an individual being selected for breeding may be proportional to its fitness value (roulette or fitness-proportionate selection) or the fitness values may only be used for ranking (tournament selection). The entire population may be replaced on each cycle (generational) or one individual at a time (steady state).

To apply the GA model to the evolution of programs, some changes are needed. If programs were represented as character strings, it would be possible to use the GA model directly, with an array of characters rather than real numbers. But in such a scheme, children would very rarely be syntactically valid. For example, the probability that even the opening and closing parentheses would match in a child formed by concatenating parts of two parent programs split at random would be infeasibly small. SGP therefore uses a different form of representation for its individuals: the expression tree (ET). Figure 1-2 shows the ET for the Lisp expression "(GT X (+ X V))". In this expression, "GT" stands for *greater than*, and the whole expression is Lisp for $x > (x + v)$. All nodes of the ET are terminal symbols of the grammar. Internal nodes are functions or operators which take arguments. External nodes are variables or constants which take no arguments. Crossover in SGP is defined to swap whole subtrees between two parents. Mutation can also be redefined in terms of subtrees. The language of the individuals can be defined in such a way that this form of crossover always produces children which are syntactically and semantically valid.

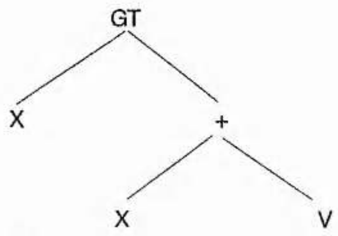

Figure 1-2: Expression tree for "(GT X (+ X V))"

SGP programs are usually S-expressions in a first-order subset of Lisp augmented with special-purpose functions relevant to the problem domain. Lisp has the advantage that S-expressions are themselves Lisp objects. It is therefore possible to manipulate individuals for operations such as crossover, and then to evaluate them to compute their fitness.

An example of a fitness function for a square root function is as follows. We first define the root mean square (RMS) error in f as:

$$RMS(f) = \sqrt{\frac{1}{|X|}\sum_{X}\left(\left(f(x)\right)^2 - x\right)^2}$$

where X is a sample of real values x. *RMS* ranges from 0 to infinity, with 0 corresponding to the ideal. For fitness-proportionate or fitness-ranked selection we need a fitness function in which higher values are better.

This can be done by an expression such as:

$$fitness(f) = \frac{1}{1 + RMS(f)}$$

It should be noted that applying a GA to a data type other than a linear list or array — even to trees — is not peculiar to SGP. [Michalewicz, 1994] describes the application of GAs to many different data types. What is unique to GP in general and SGP in particular is the principle that the individual — however represented — represents an executable program.

Given the above outline it is perhaps not surprising that GA and GP are often described with reference to *On the origin of species by means of natural selection* [Darwin, 1859]. Although this is understandable, it is not accurate, because neither GA nor GP are concerned with the evolution of new species, nor do they use natural selection. Even multi-species GP systems do not aim to produce new species. So far from using natural selection, in which there is no distant goal, both GA and GP require an explicit fitness function to be provided by the user to direct the selection towards the user's goal. It follows that GA and GP are based not so much on Darwin as on the selective breeding programmes for animals and plants which date from prehistorical times. [Koza, 1996a] refers to *animal husbandry*. What is different is that following Darwin we have a better understanding of the principles by which such programmes work. Genesis chapter 30 verses 37–42 describes a selective breeding programme in existence before the principles were understood.

1.2 Outline of Gads

Gads evolves type-correct sentences in a given context-sensitive language. The language is defined by a formal grammar which is separate from the underlying evolutionary system. If the grammar defines a programming language such as Lisp, C or Java, then Gads evolves type-correct programs. If the grammar defines a language for describing electronic circuits, neural nets, or bin packing, then Gads evolves electronic circuits, neural nets, or bin packing solutions. However, this thesis only demonstrates the evolution of programs in one context-sensitive programming language.

The main components of a Gads system are:

1 An evolutionary system such as a GA;

2 A formal grammar for the desired language;

3 An ontogenic mapping from genotype to phenotype;

4 A fitness function which defines the problem to be solved.

Of these, the evolutionary system and the fitness function are standard and are not explained further here. The use of a formal grammar and the ontogenic mapping from genotype to phenotype require explanation. An outline explanation is given below.

Formal grammars have been used to specify programming languages for many years. [Naur, 1963] introduced Backus-Naur form (BNF) to give a context-free definition of Algol 60. Context-free grammars (CFGs) are simple to understand, but to represent the context-sensitive aspects of a language, it is necessary to use a context-sensitive grammar (CSG). The most commonly used type of CSG is the sag [Pagan, 1981].

Grammars are commonly said to *generate* sentences in the language they define, but in practice, they are more often used to *parse* sentences, in the process of compiling or interpreting them. Each sentence in a language has a parse tree (PT). Figure 1-3 shows a PT for the same Lisp expression used for the ET in figure 1-2. In the PT, internal nodes are nonterminal symbols, and leaf nodes are terminal symbols, of the language. (The actual grammar is not given here. There are infinitely many possible grammars for a given language. An example of a grammar corresponding to this PT is syntax A in table 2-3.)

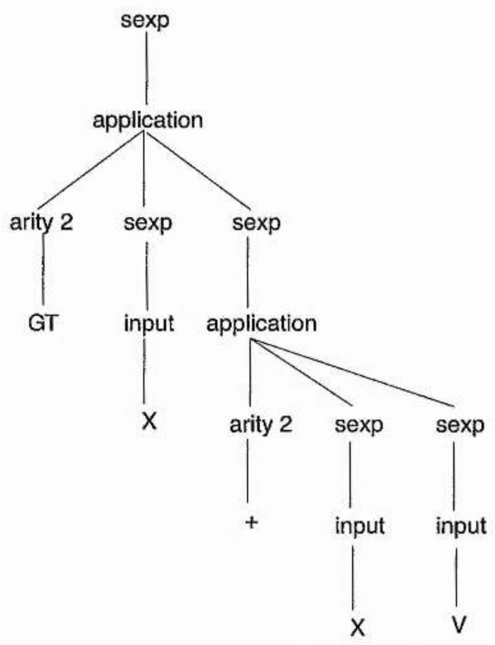

Figure 1-3: Parse tree for "(GT X (+ X V))"

Using a grammar to generate a sentence involves growing a PT, starting from a single node for the language start symbol, and repeatedly using the grammar rules to expand nonterminal nodes, until all the leaves of the PT are terminal symbols. With a CFG this is relatively straightforward, but with a sag it is so inefficient as to be infeasible. This thesis introduces the *reflective attribute grammar*, which is an enhanced attribute grammar that can be used to generate sentences in a context-sensitive language efficiently.

Generating a PT is a matter of choosing which rules to apply and which order to apply them in. This choice can be represented as a list of rules. Since a grammar can only have a finite number of rules, it is possible to number them, so a list of rules can be represented as a list of rule numbers. A list or array of numbers is precisely the kind of data that GAs operate on. Thus, we have a route from GA individuals, as lists of numbers, to type-correct sentences in any context-sensitive language we can define by means of a reflective attribute grammar (rag). This is called the *ontogenic mapping.*

In fact it is not quite as simple as the above outline suggests. The numbers in a GA individual are gene values; the numbers which direct the grammar to grow a PT are rule numbers. Although these are both numbers, they have different ranges, and a translation step is necessary. Beginning with the translation from gene values to rule numbers, the ontogenic mapping uses the gene values to select rules from the grammar and construct a PT, from which a type-correct sentence can be extracted.

Also, although the ontogenic mapping cannot produce invalid sentences, because it uses the grammar which defines the language in question, it can fail to complete the process, resulting in a PT in which one or more leaves are nonterminals. This can happen if the GA individual does not have sufficient genes, or if the gene values select rules which lead to further PT growth instead of PT termination. However it happens, a repair mechanism is necessary. This is implemented by specifying a default rule for each nonterminal in the grammar.

Gads therefore makes a clear distinction between an individual's *genotype*, which is the list or array of numbers by which the individual is represented inside a GA or other evolutionary system, and the *phenotype*, which is the sentence (ie program or other object) that is the desired solution. The *phenotype language* is the language in which the phenotype is written. Key properties of the ontogenic mapping appear to be (1) whether it supports neutral evolution, and (2) whether it is many-to-one.

1.3 Related work

[Ryan, 1998a] describes Grammatical Evolution (GE), which is closely related to Gads. GE develops the idea of Gads 1 and addresses some of its shortcomings. This thesis, in turn, adopts some of the GE ideas to develop Gads 2. The relationship between Gads and GE is described more fully in §5 *Gads2*.

1.3.1 Using formal languages

There is a progression of phenotype languages from SGP's original small untyped expression languages to large context-sensitive languages now possible with Gads.

In the original form of SGP [Koza, 1992], there is only one data type (usually floating point), and there are no unbound variables. Steps must be taken to force all calculations into this mould. For example, to avoid divide-by-zero exceptions, a protected division operator (`%`) is used which always returns a valid floating point value even if its denominator is zero; to obtain Boolean values, floating point values are compared to zero; and so on. The net effect is that SGP produces programs in an untyped expression language.

This phenotype language is simple enough to describe by a CFG. Syntax A (table 2-3) is an example of a CFG which defines the language of the cart-centering problem.

Although untyped expression languages are versatile, there has been much interest in extending SGP beyond the limitations of untyped expression languages. Sequencing, iteration, abstraction (automatically defined functions, or ADFs) and data types have been added to the basic model. But the methods of achieving these extensions have been inelegant, incomplete and restrictive, especially when compared with the facilities available to human programmers using everyday programming languages. For example, sequencing [Koza, 1992] was initially achieved by having a sequence operator PROGN2 that took two arguments. It evaluated the arguments in order. This allowed a two-step sequence; for a longer sequence it was necessary to nest or cascade the two-step operator. In [Koza, 1994] abstraction was initially achieved by having the user prescribe how many procedures and how many parameters each should take, in advance of any evolution; in chapters 21–25 a method of evolving this architecture is presented. However, it is still within the context of an untyped expression language.

Despite their simplicity, untyped expression languages are versatile. As well as solutions to numerical or symbolic problems, untyped expressions can represent the construction of neural nets, electronic circuits and other objects [Whitley, 1995], [Koza, 1998c]. Thus it has been possible to apply SGP to a surprisingly wide range of problems.

[Montana, 1995] and [Clack, 1997] introduce Strongly Typed GP (STGP), an enhanced form of SGP which supports multiple data types. STGP uses protective crossover to enforce data type constraints. The types are specified by the user, not evolved by the system. Specifying these details requires that the user has insight into how the problem might best be solved, which slightly weakens any claim to be an *automatic* programming method.

A criticism of these extensions to the original untyped expression language is that they are all somewhat *ad hoc*. Each extension addresses one particular aspect. Although abstraction, sequencing, iteration, data types etc have been exhaustively studied as part of programming language design, it is difficult to take advantage of this by piecemeal extensions. Several researchers have integrated formal grammars with SGP.

[Whigham, 1996] introduces CFG-GP, which uses an explicit CFG so that the phenotype may be in any desired context free language. The tree is a PT rather than an ET. Crossover is modified so that only subtrees with the same nonterminal root may be swapped. Mutation is modified in a similar way. The effect of this is to advance GP's range from untyped expression languages to context free languages.

[Hörner, 1996] describes a system which is similar to [Whigham, 1996].

[Bruhn, 2002] describes a system also based on PT representation but for a tiny language (3 nonterminals) specialised for the knapsack problem. The PT is extended by adding attributes to the PT nodes which represent the linear constraints of the knapsack problem. The usual SGP genetic operators are extended to ensure not only that the context-free grammar is satisfied but also that the linear constraints are satisfied. In effect this is a special-purpose attribute grammar, though it is not named as such. Bruhn's system performs better than the corresponding GA in [Michalewicz, 1994].

These systems maintain SGP's approach of representing individuals as trees, but use PTs rather than ETs. The parsing information in a PT is used by the genetic operators such as crossover to ensure that only valid offspring are produced. This extends the range of phenotype language to context-free languages. However, it is not obvious how these systems could be extended to CSGs. If a formal grammar is not used, it would be difficult to represent the rules of the language that must be enforced if the individual is to be valid; and while a CFG can be used to construct and maintain the integrity of PTs, the requirements of a CSG would be difficult to implement as a form of protected crossover.

Gads evolves type-correct sentences in a given context-sensitive language. It is thus a major step in the direction of increasing the range of phenotype languages which can be evolved.

1.3.2 Mapping genotype to phenotype

In most GP systems, the ontogenic mapping is one-to-one, and is so simple that it is easily overlooked. It is commonly said that in SGP, genotype and phenotype are not distinguished. Strictly speaking this is not correct. The SGP genotype is a Lisp ET, while the phenotype is a Lisp program. The ontogenic mapping in SGP is therefore a tree traversal, which is a one-to-one mapping.

In Compiling Genetic Programming System [Nordin, 1994a], individuals are arrays of machine code instructions. This system is unusual in that genotype and phenotype truly are not distinguished, even by a one-to-one mapping. The genotype and phenotype are one and the same. Since machine code does not have the complexity of a high level language, simple GA crossover can be used. However, minor modifications to the crossover and mutation are introduced to ensure that individual instructions are viable and do not, for example, have invalid operation codes. This approach offers great advantages in efficiency, since no translation or interpretation of the phenotype is necessary.

[Keller, 1996] describes a system with an ontogenic mapping. The mapping is initially from codons (small integers) to the terminal symbols of the phenotype language. A repair method is needed to repair any invalid sequences that result. The system does not use formal grammars.

[Ferreira, 2001] introduces Gene Expression Programming (GEP), in which genotype and phenotype are separate, in that the genotype and phenotype operators can be partitioned. The genotype is essentially tree-structured, but it is not a PT or a conventional ET. There is no translation process which maps genotype elements into phenotype elements. (Section 3.4.1 *ibid* emphasises this), ie phenotype elements are present in their final form in the genotype. Ontogenesis consists of a breadth-first traversal of the genotype tree. The main difference between GEP and SGP is in the genotype representation, which allows unused elements to accumulate in the genotype without appearing in the phenotype. While GEP is shown to perform well on some problems, it is not obvious how it might be scaled to provide sequencing, iteration, abstraction or data types.

1.4 Motivation

The aim of this work is to address some of the limitations identified in §1.3 *Related work.*

1.4.1 Formal grammars

In the early days of programming, languages were defined by their compilers. This led to so many inconsistencies that the benefit of a separate formal definition of the language was quite obvious, and nowadays languages are almost always defined by formal grammars. Given that the business of GP is to produce programs, it is something of an anomaly that the phenotype language is not represented by a formal grammar in SGP. SGP embodies mechanisms to define a language but which have little resemblance to a formal grammar. Those GP systems that have incorporated grammars have used CFGs, which are not powerful enough to represent the full complexity of a programming language.

By incorporating full-size CSGs into GP, I hope to make the benefits of programming language design such as abstraction, sequencing, iteration, data types — not to mention future programming language developments — available to GP at a stroke. Further, by separating the language from the GP system, it should be easy to change the language to evolve programs in any language. By using languages for timetables, electric circuits, or molecules, we should be able to evolve these kinds of object, using the same underlying evolutionary engine. Decoupling the language from the evolution should make many different kinds of GP simpler to achieve.

1.4.2 Simple genotype

SGP and most GP systems use a tree-based genotype. Although this was a key development which made GP possible, it has some disadvantages. It makes GP so distinct from other forms of evolutionary programming that advances in one field may not be so easily transferable to another.

By using a genotype which is a simple list or array of integers, Gads brings GP back to the GA. This makes it possible to use existing GA software, to use existing GA theory, and to avoid existing patents on tree-based systems.

1.4.3 Change in representation

It is well known that changes in the representation of the genotype can have a major effect on the performance of GP, even with the same problem and fitness function. The Travelling Salesman Problem is an example of this [Michalewicz, 1994]. Indeed, the basis of SGP is to represent programs as ETs and not as character strings [Koza, 1992].

Therefore, it is reasonable to expect that Gads would offer improved performance for at least some classes of problem, simply by virtue of a change of representation; although by the same reasoning, Gads should have worse performance on others. An example of improved performance is given in §2 *Gads 1*, where solutions to the cart-centering problem are found in generation 0 — that is, by random search.

1.5 Contribution

The main contributions of this thesis are:

Extending the range of GP to context-sensitive phenotype languages
Gads technique involves a formal grammar as part of the specification of the ontogenic mapping. By changing the grammar, the phenotype may be produced in Lisp, Java or any other language. The range of languages need not be limited to programming languages. For example, if it is possible to devise a language for electronic circuits, timetables or bin packing, then it should be possible for Gads to find solutions to these problems.

Exporting GP to other evolutionary systems
The Gads genotype is a list or array of integers. The simplicity of this data structure means that Gads could be fitted on to a range of base technologies, such as GAs, simulated annealing (SA), or even evolution strategies (ESs), though to date, only GAs have been used. Thus, Gads is not closely coupled to any particular underlying technology. This brings several advantages. One, Gads is able to leverage existing technologies. Two, Gads is able to draw on existing theories. Three, since Gads does not use tree-based representations, existing GP patents are avoidable.

1.6 Reading guide

The main sections of this thesis are as follows. §2 *Gads 1* describes the first implementation of Gads, which used a small CFG and a primitive ontogenic mapping. §3 *Statistics* and §4 *Grammars* address the main limitations of §2 *Gads 1*. §5 *Gads 2* describes the second implementation of Gads, which uses a full-size CSG and a more advanced ontogenic mapping. §6 *Conclusions* discusses the strengths and weaknesses of the work, and where it leads.

2 Gads 1

This section describes Gads 1. Much of the material in this section is taken from [Paterson, 1996], which reports the results of the first experiments to investigate the Gads technique.

The aim of the investigation was to decide whether Gads was feasible. For this reason, many aspects of the implementation are far from optimal. A technique that only works with careful tuning is not as valuable as one which works, however inefficiently, without it. Also, having found that Gads work without careful tuning suggests that there is room to improve its performance at a later date.

2.1 Introduction

Gads 1 is an implementation of GP that uses array genotypes. The Gads 1 genotype is fixed-length array of integers which, when read by a suitable generator, causes that generator to write the program that is the corresponding phenotype. There is therefore a clear distinction between the genotype and the phenotype. The mapping from genotype to phenotype is called the ontogenic mapping. The genotype is operated on by the genetic operators — crossover, mutation and so on — in the usual range of ways available to GAs. To evaluate the fitness of a genotype, Gads 1 tests the phenotype in the environment in which it is to operate. The result of the testing gives the fitness of the phenotype, and by implication, of the underlying genotype.

Using an array genotype has the immediate advantage that conventional GA engines can be used. Using a generator to convert from genotypes to phenotypes is not new. [Michalewicz, 1994] gives many examples, though not using the terms *genotype* and *phenotype.*

The reason for investigating alternatives to ET genotypes is that the behaviour of a GA system (and GA is taken here to include GP as a particular case) depends greatly on the way the genotypes represent the phenotypes. Thus, we should expect some change in performance to result from a change of design. Whether the change is for the better or worse is only to be found by experiment, but it would be reasonable to expect that there are some classes of problem for which one approach is more suitable than the other. The range of programs that can be produced by list- or tree-based GP should, however, be identical.

2.2 Principles of Gads 1

2.2.1 The phenotype language

The *phenotype language* is the programming language in which the phenotypes produced by Gads 1 are written. The choice of phenotype language therefore depends on the problem domain which Gads 1 is addressing. Lisp is an obvious candidate to begin with, because so much GP has already been done using it, and results are available for comparison; but Gads 1 is not limited to Lisp.

2.2.2 The ontogenic mapping

There are many candidates for the ontogenic mapping. For example, a feature common among text editors is the ability to accept a sequence of mark-up commands to apply to a file. We can assign an identifying integer to each mark-up command, so that an editing sequence can be coded as a list of integers. Given an initial file to edit, the editor is then an implementation of an ontogenic mapping. [Keller, 1996] describes a many-to-one mapping from genes to the terminal symbols of the phenotype language. Program transformations offer yet another possible class of genotype mappings. Gads 1 uses a BNF definition of the phenotype language syntax as the basis of the ontogenic mapping. The syntax of the phenotype language is written in BNF as a set of productions of the form:

```
LHS ::= RHS
```

where the left hand side (LHS) is one nonterminal symbol of the language, and the right hand side (RHS) is a concatenation of zero or more symbols (terminal or nonterminal) of the language. The productions are numbered from 0 to *n*, so that any production can be represented by a number in the range [0, *n*]. For example:

#	production		
0	`<sexp>`	`::=`	`<input>`
1	`<sexp>`	`::=`	`(GT <sexp> <sexp>)`
2	`<sexp>`	`::=`	`(+ <sexp> <sexp>)`
3	`<sexp>`	`::=`	`(- <sexp> <sexp>)`
4	`<input>`	`::=`	`X`
5	`<input>`	`::=`	`V`

Thus, beginning with the start symbol <sexp>, a production sequence results in a particular program. For example, the sequence 1, 0, 4, 2, 0, 4, 0, 5 progressively transforms the start symbol <sexp> into the expression `(GT X (+ X V))`:

```
        <sexp>
1       (GT <sexp> <sexp>)
0       (GT <input> <sexp>)
4       (GT X <sexp>)
2       (GT X (+ <sexp> <sexp>))
0       (GT X (+ <input> <sexp>))
4       (GT X (+ X <sexp>))
0       (GT X (+ X <input>))
5       (GT X (+ X V))
```

Every well-formed program in a language has a derivation according to the syntax of that language. A derivation is a sequence of productions which, when applied in turn, transform the language's start symbol into the program. Thus, any program can be represented by a derivation, and any derivation can be represented by the sequence of integers which correspond to the productions in its derivation. In short, any program can be represented as a sequence of integers in the range [0, *n*]. This is the basis for representation of programs in Gads 1.

Provded the phenotype language is context-free, using BNF brings many benefits:

- BNF is well-established.
- The BNF syntax for many languages is readily available.
- A BNF generator produces all programs in a language. There are no programs which, due to some unintentional peculiarity of the generator, cannot be produced.
- A BNF generator produces only well-formed programs. Unviable genotypes do not occur. Expensive repair mechanisms are not necessary.
- It is feasible to include the BNF syntax of the phenotype as part of the input to Gads 1. Thus the phenotype language need not be hard coded into Gads 1.

The use of | to indicate alternatives in the RHS is a commonly-used BNF shorthand for several productions which have the same LHS. In Gads 1, this shorthand notation is not used. Each RHS consists of a single concatenation.

2.2.3 The initial population

Each gene in the initial population is generated as a random number, distributed uniformly over the range [0, *n*]. This means that each production in the phenotype syntax is equally likely to occur at each gene in the genotype.

There are, however, infinitely many syntaxes for a given phenotype language. One syntax may be a simple renaming of another, or the relationship between them may be more subtle. Distinct syntaxes give rise in general to different distributions of programs in the initial population. The initial distribution can be controlled by the choice of syntax. The question of using a distribution other than uniform to generate the initial population, so that certain productions are favoured at the expense of others, is not discussed here. Nor is the question of using different distributions for different gene positions along the chromosome.

2.2.4 Generating the phenotype

A BNF definition of the phenotype language is needed. For this investigation, simple BNF definitions of Lisp were used, with about 10 productions. Phenotypes may be generated as strings or ETs. Gads 1 uses ETs.

A string phenotype is a character string containing embedded nonterminal symbols. Applying a production involves searching the string for an occurrence of the nonterminal which is the left hand side of the current production. Although simple, this involves much linear searching, and care is needed to avoid creating spurious nonterminals by the juxtaposition of terminals. A data structure can be used to keep a record of the position and type of all nonterminals in the string, removing the need for linear searches and so improving performance.

An ET phenotype requires a data structure for ET nodes. Each node of the ET represents one symbol of the phenotype language, and has links to zero or more child nodes. The number of links a node has depends on the symbol it represents. When a nonterminal node is expanded, links to child nodes are added, depending on the RHS of the production. For example, the partially developed phenotype

```
(GT (- X <sexp>) <sexp>)
```

is represented by the ET shown below:

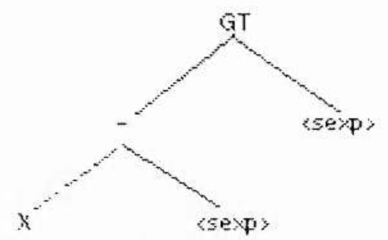

Figure 2-1: A partially developed ET

Whichever form of phenotype is used, the generator begins by initialising a new phenotype, and then applies the productions identified by the genes in the order they occur along the chromosome. The initial value of the phenotype is usually the start symbol of the language; for Lisp, this is <sexp>. In more general terms, the initial value may be any partially-generated program such as

```
(GT <sexp> <sexp>)
```

Using a partially generated program limits Gads 1 phenotypes to a subset of the programs in the phenotype language.

2.2.5 Inapplicable productions

As ontogenesis proceeds, each gene in turn identifies the next production to apply. But a given production can only be applied if the developing phenotype contains the nonterminal which is the production's LHS. For example, given the partially generated phenotype:

```
(GT <sexp> <numeral>)
```

only a production with <sexp> or <numeral> on its LHS can be applied. If a gene selects a production which cannot be applied, the generator passes over that gene, and moves on to the next gene.

2.2.6 Residual nonterminals

After all the productions in the genotype have been applied, there may still be some nonterminals in the phenotype. This could happen because the ontogenic mapping reached the end of the chromosome before all the nonterminals were expanded, or perhaps because the necessary genes for the nonterminal were not present in the chromosome. However it happens, the end result is a phenotype which is not fully developed, and whose fitness therefore cannot be evaluated in the usual way.

There are several ways of dealing with this. The phenotype can be rejected as unviable, or penalised with a very low fitness, without further evaluation.

It is also possible to repair the phenotype. This has the advantage that the resulting phenotype is well-formed, so it can be evaluated in the ordinary way, and can contribute to the search for a good solution. Gads 1 uses a simple repair method, which is to expand remaining nonterminals to a default terminal value. For example, every remaining <sexp> can be replaced by 0, every <atom letter> by A, and so on. However, deciding what the default value should be for any given nonterminal would not be entirely trivial for a large language. The default could even be a random value, but this raises the question of endless iterations and non-repeatability.

2.2.7 Evaluation

Evaluating the individual requires either that it be compiled, linked and executed; or that it is interpreted.

Compiling and linking introduces a considerable overhead, but if the fitness evaluation involves much processing, the overhead is likely to be recovered in more efficient execution.

Given that Gads 1 phenotypes are ETs, it was simple to evaluate them by means of an ET interpreter. The interpreter was designed to give any residual nonterminals default values during interpretation, so that the repair mechanism was actually implemented in the interpreter.

2.3 Experimental design

This section describes an experiment to test the Gads 1 technique by applying it to the cart-centering problem. The aim of the experiment is firstly to discover whether Gads 1 works at all, and secondly to begin to discover how various conditions affect its performance.

2.3.1 The problem

The problem is to find a program to control the motion of a cart. The cart can move to left or right along a straight frictionless track. At any time t, the position of the cart is x and it is moving at velocity v. The cart is subject to a force applied either in the positive or negative direction, so that the cart accelerates uniformly to the left or right. The control program can control the direction of the force, but not its magnitude; in particular, it cannot be switched off. By choosing appropriate units of measurement, mass of cart and magnitude of force, we can arrange that the optimal solution has a simple form.

The control program controls the motion of the cart by computing whether to apply the force to the left or to the right. That is, the program calculates a given x and v. The aim of the control program is to bring the cart to rest (ie $v = 0$) at the origin (ie $x = 0$) in the shortest possible time. This problem has an analytical solution, which is:

$$\text{if } \left(x < v \times abs(v)\right) \text{ then } a = +0.5 \text{ else } a = -0.5$$

This problem is chosen as a test case for Gads 1 because it is well-known, and gives a basis for comparison with tree based GP. [Koza, 1992] refers to further details of this problem in [Macki, 1982] and [Bryson, 1975].

2.3.2 Experimental conditions

This section outlines which conditions are held constant, and which are varied.

The experimental conditions which are held constant are outlined in table 2-1.

Value	Description
GAGS-0.95	A conventional GA engine is used. See §2.3.3.
500	The population is fixed at 500 individuals.
uniform	The genes of the initial population are randomly generated with a uniform distribution over [0, n] where n is the number of the last production in the syntax.
best of run	The best individual found in the course of the run is designated as the solution. If several individuals have the same best fitness, the first one found is kept.
full term	The run terminates after 50 generations.
ET	Phenotypes are generated as ETs which can be interpreted. See §2.3.4.
Simulation	Phenotypes are evaluated by simulating control of a cart for 20 test cases. See §2.3.5

Table 2-1: Experimental constants

The experimental conditions which are varied are outlined in table 2-2.

Name	Values	Description
Syn	A, B	Two syntaxes (ie genotype mappings) for the same phenotype language are compared. See §2.3.6.
Init	<sexp>, <application>	Two initial values for generating the phenotype are compared. See §2.3.7.
Sel	ELITE 10%, ROULETTE	Two selection methods for the GA are compared. See §2.3.8.
Len	50, 200	Two chromosome lengths are compared. See §2.3.9.

Table 2-2: Experimental variables

The above four experimental variables give a total of 16 sets of experimental conditions. The experimental variables are described in more detail below.

In addition to the experimental variables, three different sets of test data are used for each of the 16 conditions, making 48 runs in all. A different random seed is used to generate the test data in each of the 48 runs.

2.3.3 The GA engine

The GA engine used for this experiment is GAGS-0.95 [Merelo, 1994]. GAGS-0.95 was initially selected partly because it supports variable-length chromosomes. However, this facility is not used in the experiments.

GAGS is a conventional genetic algorithm, not especially tailored or customised for Gads 1. In fact it is in some ways less than ideal because the genes are real numbers rather than integers. They must be mapped from the real range [0, 1] into the integer range [0, *n*] to identify a production. However, this illustrates an advantage of Gads 1 over tree-based systems, namely that it does not need a specialised GP engine.

2.3.4 Generating the phenotype

The phenotype is generated as an ET. Each node of the ET is implemented as a tuple of 3 integers:

```
(SYMBOL, LINK1, LINK2)
```

SYMBOL is a simple encoding which specifies a terminal or nonterminal of the phenotype language. LINK1 and LINK2 are pointers to other nodes in the tree. The NIL value is used when an actual pointer is not required. Zero, one or two of the links in a node may actually be used, depending on SYMBOL. For example, the partially developed phenotype

```
(GT (- X <sexp>) <sexp>)
```

is implemented as shown below:

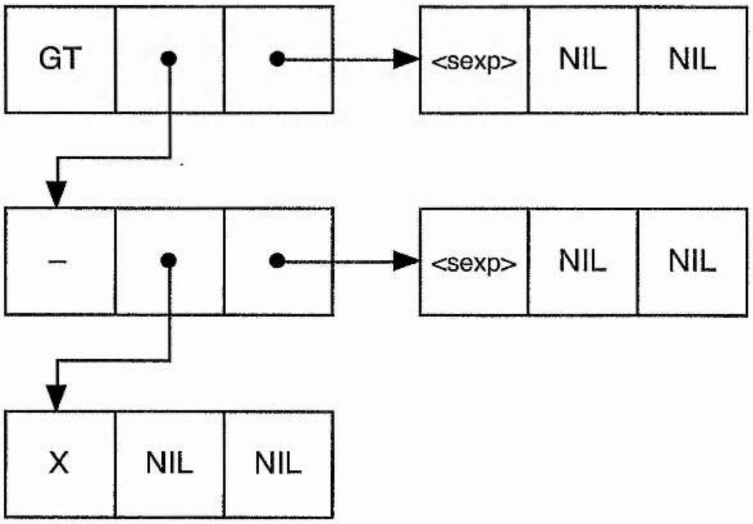

Figure 2-2: Implementation of an ET

This is the same example as is used in figure 2-1. The structure of the ET is identical, but in figure 2-2 more of the implementation is apparent. Since the nodes have three fields, the ET can be written as triples:

```
(GT,
        (-, (X, NIL, NIL), (<sexp>, NIL, NIL) ),
        (<sexp>, NIL, NIL)
)
```

The generator begins by creating a new ET as a single node. The first field of the node, SYMBOL, is specified by the experimental variables. The second and third fields, LINK1 and LINK2, are set to NIL. Generation then continues by applying the productions corresponding to the genes along the chromosome in turn. To apply a production, the generator traverses the ET from the root, in infix order, looking for a node whose SYMBOL is the same as the LHS of the production. If no such node can be found, the production is ignored and the generator passes on to the next gene.

Once a suitable gene has been found, the SYMBOL of the node is updated. If the production has only one symbol in its RHS this application is complete. If the production has more than one symbol in its RHS, one or both of the link fields are also used. For example, consider productions 5 and 6 of syntax A (table 2-3 below) being applied to the following node:

```
(<application>,
      NIL,
      NIL
)
```

For production 5, the result is:

```
(<arity1>,
      (<sexp>, NIL, NIL),
      NIL
)
```

That is, a new <sexp> node is created, and connected to the tree by the first link in the node being updated. For production 6, the result is:

```
(<arity2>,
      (<sexp>, NIL, NIL),
      (<sexp>, NIL, NIL)
)
```

That is, two new <sexp> nodes are created and connected to the tree.

2.3.5 Evaluating the phenotype

The fitness of the phenotype ET is measured by simulating control of the cart for 20 random (x, v) starting conditions uniformly distributed over the range [(-0.75, -0.75), (+0.75, +0.75)]. A different random seed is used for each of the 48 runs. The same test cases are used throughout each run.

The repair mechanism for residual nonterminals (which are discovered during evaluation of the ET) is to interpret them as 0. If the residual nonterminal occupies the SYMBOL field of a node, then the entire subtree of the ET is interpreted as 0, whether the links are nonterminals or not. If the residual nonterminal occupies the LINK1 or LINK2 field of a node, then that field is interpreted as 0.

Simulation requires a wrapper to convert the real number returned from the evaluation of the phenotype into an acceleration of +0.5 or -0.5. The wrapper used is:

```
if (evaluate > 0.0) then +0.5 else -0.5
```

where `evaluate` is the result of evaluating the phenotype in the environment of x and v.

The simulation equations of motion are:

$$v(t+\tau) = v(t) + \tau a(t) \tag{1}$$

$$x(t+\tau) = x(t) + \tau v(t) + \tau^2 a(t)/2 \tag{2}$$

Equation (1) says that the velocity at time $(t + \tau)$ is the same as the velocity at time (t), plus any change due to acceleration a at time (t) for the duration τ. If the acceleration is zero, there is no change in velocity. If the acceleration is greater than zero, the velocity at time $(t + \tau)$ is greater than the velocity at time (t), and if the acceleration is less than zero, the velocity at time $(t + \tau)$ is less than the velocity at time (t).

Equation (2) says that the position at time $(t + \tau)$ is the same as the position at time (t), plus any change due to velocity v at time (t) for the duration τ, and for the acceleration a at time (t) for the duration τ. Of course if there is a non-zero acceleration, the velocity at the start of the time quantum will not be the same as the velocity at the end, but by making the time quantum small enough this error can be kept to an acceptable limit.

Taken together, the equations compute the velocity v and the position x of the cart at time $(t + \tau)$ in terms of its velocity and position at time (t). The symbol τ is the time quantum, which is set at 0.02 s. By computing these equations repeatedly the simulation computes the velocity and position of the cart at any desired time.

Simulation continues until either the simulated time runs out (at 10 s) or the cart is close enough to the origin of the (x, v) plane (ie $x^2 + v^2 \leq r^2$), where r is the target radius, set at 0.1 m.

Thus it is not necessary for the simulated cart to come exactly to rest exactly at the origin. An approximate solution will do; the degree of approximation being controlled by the target radius *r* and the time quantum τ. By contrast, the theoretical solution to this problem, given below in §2.3.9 *Chromosome length*, is the exact solution. The exact solution satisfies the approximate problem, but the converse is not true. The importance of this is that if the GP system is left to evolve after the theoretical solution has been found, it can go on to find solutions which are better than the theoretical optimum, which is puzzling unless you realize that GP and theory are not solving precisely the same problem.

The raw fitness is the sum of the simulated time over all 20 test cases. The adjusted fitness is 1.0/(1.0+raw fitness).

2.3.6 Syntax

Two syntaxes are compared: A and B. They are shown in tables 2-3 and 2-4.

#	production
0	`<sexp> ::= <input>`
1	`<sexp> ::= <application>`
2	`<input> ::= X`
3	`<input> ::= V`
4	`<input> ::= -1`
5	`<application> ::= (<arity1> <sexp>)`
6	`<application> ::= (<arity2> <sexp> <sexp>)`
7	`<arity1> ::= ABS`
8	`<arity2> ::= +`
9	`<arity2> ::= -`
10	`<arity2> ::= *`
11	`<arity2> ::= %`
12	`<arity2> ::= GT`

Table 2-3: Syntax A

#	production
0	`<sexp> ::= <input>`
1	`<sexp> ::= <application>`
2	`<input> ::= X`
3	`<input> ::= V`
4	`<input> ::= -1`
5	`<application> ::= (ABS <sexp>)`
6	`<application> ::= (+ <sexp> <sexp>)`
7	`<application> ::= (- <sexp> <sexp>)`
8	`<application> ::= (* <sexp> <sexp>)`
9	`<application> ::= (% <sexp> <sexp>)`
10	`<application> ::= (GT <sexp> <sexp>)`

Table 2-4: Syntax B

Both syntaxes define the same phenotype language: a stripped-down first-order Lisp, containing only nested arithmetic expressions. All values are real numbers. The meaning of the terminal symbols is as follows:

X

A variable whose value is the position of the cart.

V

A variable whose value is the velocity of the cart.

-1

The constant -1.

ABS

A function of one argument, which returns its absolute value.

+

A function of two arguments, which returns their sum.

-

A function of two arguments, which returns the first less the second.

*

A function of two arguments, which returns their product.

%

A function of two arguments, which returns the first divided by the second. If the second is within ±0.000001, the value 1 is returned. This avoids the risk of dividing by zero.

GT

A function of two arguments, which returns 1 if the first is greater than the second, and 0 otherwise.

Although both syntaxes define the same phenotype language, the probability of any given program being produced varies between the two syntaxes. The difference in program probabilities can be shown for the case of function applications. The relative frequency of the various functions in programs generated using the two syntaxes is shown in table 2-5.

Function	**Syntax A**	**Syntax B**
ABS	50%	17%
+	10%	17%
-	10%	17%
*	10%	17%
%	10%	17%
GT	10%	17%

Table 2-5: Relative frequency of functions

Syntax A is skewed towards ABS, at the expense of the other functions. Although ABS is involved in the optimal solution, it does not comprise 50% of the functions. Syntax B's phenotype distribution should be richer in likely solutions to the cart-centering problem.

2.3.7 Initial value

Two initial values for generating the phenotype are compared: <sexp> and <application>.

<sexp> is the start symbol of Lisp. Using this as the initial value for phenotype generation means that the whole of the stripped-down Lisp is the range of phenotypes that can be generated. In particular, x, v and -1 are possible phenotypes.

In order to avoid these over-simple phenotypes, and force Gads 1 to generate at least one function application, we use <application> as an alternative initial value for program generation. This nonterminal occurs in both syntaxes, with the same syntactic meaning, although the resulting phenotype distribution is different.

2.3.8 Selection method

GAGS supports a range of selection methods. Two methods are compared.

ROULETTE

GAGS forms a gene pool by selecting individuals from the old population in proportion to their fitness, using the roulette wheel algorithm. Pairs of parents are chosen at random from the gene pool and mated using uniform crossover. The offspring form the new population.

ELITE 10%

GAGS removes the worst 10% of the population, and replaces them by breeding. For each breeding pair, one parent is chosen by fitness proportionate selection, and the second is chosen by uniform random selection. The parents are mated using uniform crossover and added to the population.

2.3.9 Crossover

GAGS uses uniform crossover. Given that the Gads 1 chromosome is an array, with a definite beginning and a clear ordering of the genes along its length, one-point crossover might be expected to produce much better results. However, this is not an option in GAGS 0.95. Rather than develop yet another GA, it was decided to stick with GAGS, with the intention of discovering whether Gads would work with a less-than-optimal GA.

If uniform crossover is actually counter-productive, in the sense that it disrupts useful gene subsequences, then roulette selection is likely to perform badly. Elitism is likely to take more generations to produce any result at all, although the meaning of *generation* is different.

2.3.10 Chromosome length

Two chromosome lengths are compared: 50 genes and 200 genes.

The optimal program can be represented as:

```
(GT (* -1 X) (* V (ABS V)))
```

and can be represented (using syntax A) by as few as 21 genes:

```
        <sexp>
1       <application>
6       (<arity2> <sexp> <sexp>)
12      (GT <sexp> <sexp>)
1       (GT <application> <sexp>)
6       (GT (<arity2> <sexp> <sexp>) <sexp>)
10      (GT (* <sexp> <sexp>) <sexp>)
0       (GT (* <input> <sexp>) <sexp>)
4       (GT (* -1 <sexp>) <sexp>)
0       (GT (* -1 <input>) <sexp>)
2       (GT (* -1 X) <sexp>)
1       (GT (* -1 X) <application>)
6       (GT (* -1 X) (<arity2> <sexp> <sexp>))
10      (GT (* -1 X) (* <sexp> <sexp>))
0       (GT (* -1 X) (* <input> <sexp>))
3       (GT (* -1 X) (* V <sexp>))
1       (GT (* -1 X) (* V <application>))
5       (GT (* -1 X) (* V (<arity1> <sexp>)))
7       (GT (* -1 X) (* V (ABS <sexp>)))
0       (GT (* -1 X) (* V (ABS <input>)))
3       (GT (* -1 X) (* V (ABS V)))
```

However, it might not be reasonable to expect such a compact chromosome to arise in practice. The question is, how long must the chromosome be? In general, the shorter the chromosome the better, since shorter chromosomes require fewer computing resources.

A few simple experiments were carried out to show the relation between chromosome length and program length. 1 000 chromosomes of length 50, 100 and 200 uniformly random genes were generated, and used to generate programs, beginning with <sexp>. The lengths of the programs were measured by counting the symbols (ie variables, constants and functions, but not parentheses or spaces).

This question can also be answered analytically by the following method. Suppose we have an infinite chromosome with a uniform distribution of gene values. Rewrite the CFG as a set of simultaneous equations, according to these rules:

1 Merge rules which have the same LHS into a single rule using the bar symbol (|) to separate alternatives. For example, rewrite syntax A as:

```
<sexp>           ::=   <input> | <application>
<input>          ::=   X | V | -1
<application>    ::=   (<arity1> <sexp>)
                 |     (<arity2> <sexp> <sexp>)
<arity1>         ::=   ABS
<arity2>         ::=   + | - | * | % | GT
```

2 Convert each rule into an equation by making the following substitutions:

2.1 Replace each nonterminal by an algebraic unknown which represents the expected length of sentences derived from the nonterminal. (The simplest way to do this is to remove the angled brackets and italicise the name.)

2.2 Replace each terminal by the value 1 (which is its expected length). If you don't want to count a certain terminal, replace it by the value 0. This is done below with the terminal symbols (and).

2.3 Replace each ::= with =.

2.4 Replace concatenation in the RHS by addition.

2.5 Replace alternation by an averaging function; for example, replace:

```
P | Q | R
```

by:

```
average (P , Q , R)
```

or:

```
(P + Q + R) / 3
```

The substitutions must be done with care to avoid accidentally confusing terminal symbols and variable names. Rewriting syntax A leads to the following system of simultaneous linear equations:

```
sexp          =      average(input, application)
input         =      average(1, 1, 1)
application  =      average(0 + arity1 + sexp + 0,
                     0 + arity2 + sexp + sexp + 0)
arity1        =      1
arity2        =      average(1, 1, 1, 1, 1)
```

which can be solved for *sexp* to give:

```
    sexp        =       4
```

Table 2-6 shows the mean program length for both syntaxes, both theoretical and empirical:

Genes	Syntax A	Syntax B
50	2.669	3.934
100	3.345	5.206
200	3.648	6.718
Theoretical	4	12

Table 2-6: Mean program length

In both syntaxes, the empirical programs are well short of the theoretical limit. This implies that the ontogenic mapping runs out of genes before all nonterminals have been expanded. Given that we are dealing with very small grammars, the fact that 200 genes only generates programs half as long as they could be is cause for concern. It suggests that the ontogenic mapping uses genes inefficiently, and would not scale to larger grammars.

With a slight modification, the method of converting the CFG to a system of equations can compute the expected number of genes needed to produce a program in a given language. The first method, given above, works by computing the expected number of leaf nodes in a PT. Each leaf is a terminal symbol and is given the value 1 (unless it is a terminal you don't wish to count, in which case it is given the value 0). To compute the expected number of genes needed we modify the equations to count internal nodes instead of leaves, since each internal node requires one gene to expand it.

The modifcations are (1) the RHS of each equation begins with "`1 +` " to count the gene for the nonterminal being expanded, and (2) each terminal symbol is replaced by 0. The system of equations for syntax A is:

```
sexp        =    1 + average(input, application)
input       =    1 + average(0, 0, 0)
application =    1 + average(0 + arity1 + sexp + 0,
                 0 + arity2 + sexp + sexp + 0)
arity1      =    1 + 0
arity2      =    1 + average(0, 0, 0, 0, 0)
```

which can be solved for *sexp* to give:

```
sexp        =        10
```

That is, it takes on average 10 genes to derive a program from <sexp>. This is low compared to the chromosome lengths of 50, 100 and 200 that are investigated empirically above, which is consistent with the conclusion that the ontogenic mapping is inefficient.

The maximum length of program produced increases with the chromosome length. It is also apparent that syntax B produces noticeably longer programs than syntax A.

The number of residual nonterminals drops as the chromosome length increases. This is of interest, since it suggests an upper bound on chromosome length, which there is no advantage in passing.

It must be remembered that the above discussion applies only to the first generation. After that, the effect of evolution — selection and recombination — change the average lengths of programs in the population., as shown below in table 2-8. For the purposes of the experiment, chromosome lengths of 50 and 200 genes are used.

2.4 Experimental results

Several measurements were made for each of the 48 runs, and other quantities can be calculated from them. The effect of the experimental parameters on each of these is discussed in a separate section below. Table 2-7 summarises the measurements and the effect of the experimental parameters on them.

Value	Description
Phenotype length (Sym)	The size of the best-of-run's phenotype, as generated, measured by counting symbols. *Major factors: Len and Syn.* See §2.4.1.
Number of generations (Gen)	The generation (starting at 0) where the best individual was found. *Major factors: Syn and Sel.* See §2.4.2.
Number of individuals (Ind)	The number of individuals generated up to the best-of-run. *Major factors: Sel; others minor.* See §2.4.3.
Time to center the cart (Std)	The best individual's average time to center the cart, in simulated seconds. *Minor factors: Sel and Init.* See §2.4.4.
Optimal program	Gads 1 finds the optimal program. See §2.4.5.

Table 2-7: Experimental measurements

To gauge the effect of an experimental variable on a measurement, we divide the 48 runs into two groups, one group for each value of the variable. We then compare the mean measurement value of both groups. This is as much statistical analysis as seems justified, given the small population size and our ignorance of the underlying distributions.

2.4.1 Phenotype length (Sym)

The length of generated programs measured as the number of nodes of the ET. This ignores parentheses, and counts each name, constant or non-terminal as one symbol. The shortest best-of-run program, with length 4 symbols, is:

```
(GT (ABS X) V)
```

The mean program length has clearly increased, as can be seen by comparing tables 2-6 and 2-8. Short length helps reduce the load of calculating fitness, because there is less computation necessary to evaluate a smaller ET than a larger one. Short length also makes programs easier for a person to understand, as can be seen by examining the three equivalent programs shown in §2.4.5 *Optimal program* below. But longer length is necessary to represent more complicated algorithms. The distribution of program length over all 48 runs is shown in figure 2-3.

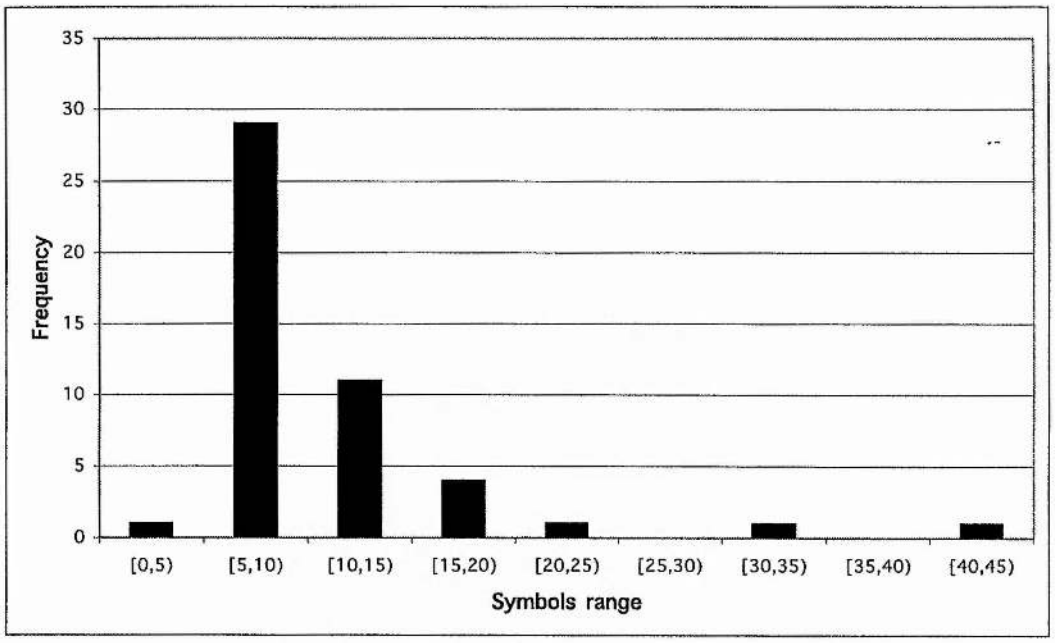

Figure 2-3: Distribution of phenotype length (Sym)

The effect of experimental variables on program length is shown in table 2-8.

Var	Value	Mean	Value	Mean
Syn	A	7.5	B	11.3
Init	<sexp>	9.0	<application>	9.7
Sel	ELITE 10%	8.8	ROULETTE	9.9
Len	50	6.6	200	12.1

Table 2-8: Factors affecting phenotype length (Sym)

Chromosome length and syntax have the greatest effect on Sym. Initial symbol and selection method do not appear to affect length significantly.

2.4.2 Number of generations (Gen)

By itself, Gen gives an idea of how much evolution is taking place. The number of generations necessary to find the solution in each run is relatively small. On average overall 48 runs only 8 generations are needed, and in 25% of runs, the best result is found in generation 0. This suggests that the population does not evolve particularly well.

On the other hand, the optimal solution was found in generation 0 of run 30. This is very much at odds with tree-based chromosomes. Chapters 7 and 9 of [Koza, 1992] suggest (but do not state explicitly) that SGP very rarely solves the cart-centring problem in generation 0. That one run in 30 of Gads 1 should do is therefore intriguing, and worth investigating further.

The distribution of Gen over all 48 runs is shown in figure 2-4.

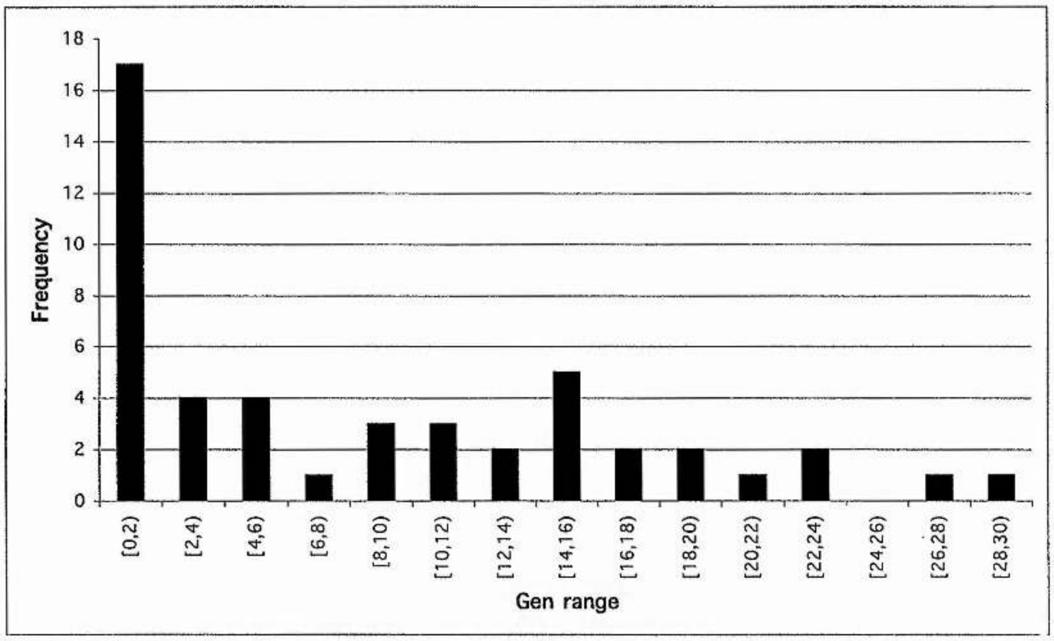

Figure 2-4: Distribution of number of generations (Gen)

The effect of experimental variables on Gen is shown in table 2-9.

Var	Value	Mean	Value	Mean
Syn	A	6.9	B	9.0
Init	<sexp>	8.1	<application>	7.8
Sel	ELITE 10%	6.9	ROULETTE	9.0
Len	50	7.8	200	8.1

Table 2-9: Factors affecting number of generations (Gen)

Syntax and selection method have the greatest effect on Gen. Initial symbol and chromosome length do not appear to affect Gen significantly.

2.4.3 Number of individuals (Ind)

The number of individuals generated up to and including the best of the run gives an idea of how efficient the search is.

The number depends on the selection method and the generation in which the best individual was found. The actual number of evaluations is not available from GAGS, so we assume that on average half of the new individuals in the successful generation are generated before the best is discovered. However, even a pessimistic assumption here (ie that the entire population is evaluated before the best is discovered) makes little difference to the number of evaluations.

ROULETTE

Each generation is completely new. The number of individuals considered is:

$$Gen \times 500 + 250$$

ELITE 10%

Only 10% of the population is replaced at each generation after the first. The total number of individuals considered is:

$$\textit{if}\;\; Gen = 0 \;\; \textit{then} \;\; 250 \;\; \textit{else} \;\; 500 + Gen \times 50$$

The distribution of Ind over all 48 runs is shown in figure 2-5.

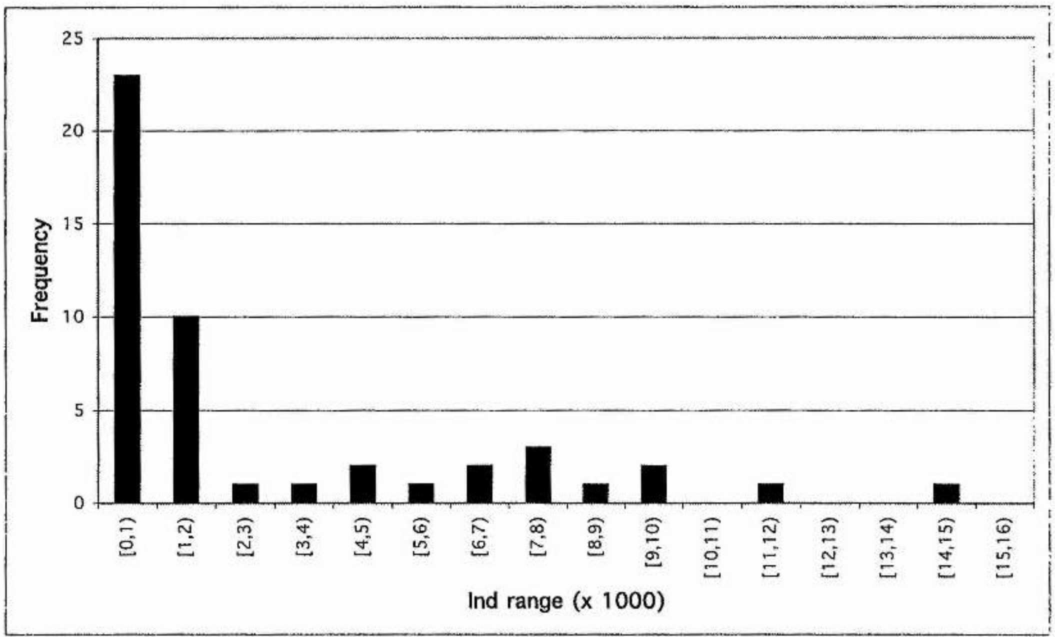

Figure 2-5: Distribution of number of individuals (Ind)

The effect of experimental variables on Ind is shown in table 2-10.

Var	Value	Mean	Value	Mean
Syn	A	2637.5	B	2852.1
Init	<sexp>	2829.2	<application>	2660.4
Sel	ELITE 10%	739.6	ROULETTE	4750.0
Len	50	2568.8	200	2920.8

Table 2-10: Factors affecting number of individuals (Ind)

As might be expected, selection method has the most significant effect on Ind, but all four experimental parameters appear to affect Ind significantly.

2.4.4 Time to center the cart (Std)

Since every run used a different random seed for its 20 test cases, the time from the runs has a large random component, and is not useful for comparison. Instead, a fixed sample of 1 000 random test points was generated and used to compare phenotypes after all runs were complete. The time to center the cart using the standard test data is called the standard time, or Std.

Two outlying values of Std (5.7 s and 5.8 s) are excluded from the following analysis. The distribution of Std over the remaining 46 runs is shown in figure 2-6. The most frequent time is 2.418 s, corresponding to the phenotype:

```
(GT (* X -1) V)
```

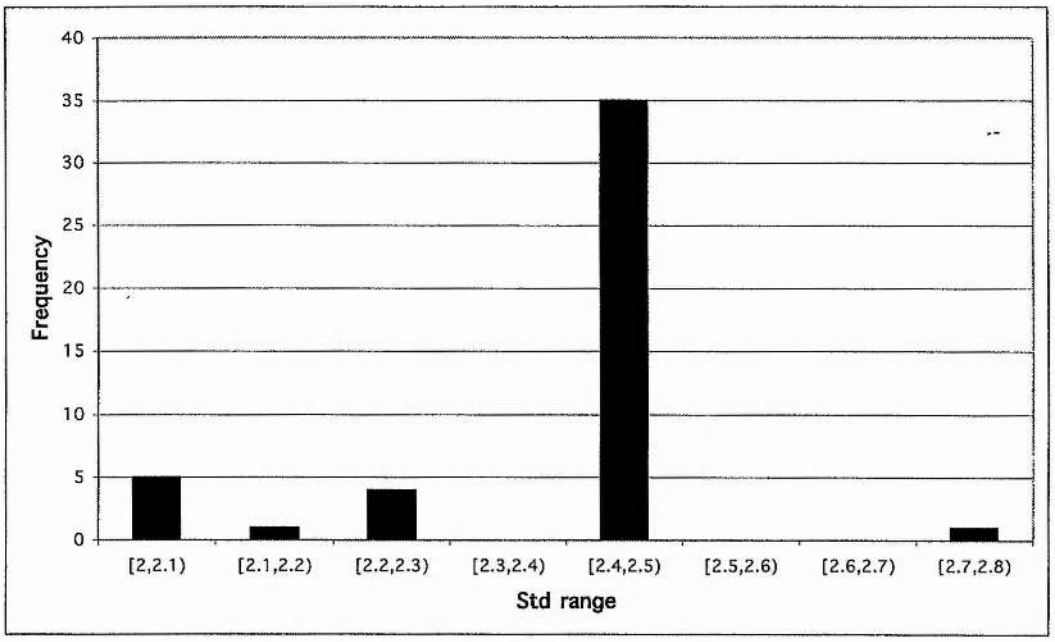

Figure 2-6: Distribution of time to center (Std)

The effect of experimental variables on the 46 values of Std is shown in table 2-11.

Var	Value	Mean	Value	Mean
Syn	A	2.363	B	2.368
Init	<sexp>	2.404	<application>	2.327
Sel	ELITE 10%	2.406	ROULETTE	2.325
Len	50	2.389	200	2.342

Table 2-11: Factors affecting time to center (Std)

Selection method and initial symbol have the greatest effect on Std, but the effect is slight. The other experimental variables do not appear to affect Std significantly.

2.4.5 Optimal program

A vital question is whether Gads 1 discovers the optimal program. The optimal program can be written as:

```
(GT (* -1 X) (* V (ABS V)))
```

Gads 1 finds the optimal program in runs 10, 22 and 30. The run details are shown in table 2-12.

Nº	Syn	Init	Sel	Len	Gen	Std
10	A	appl	roul	200	7	2.019
22	A	sexp	roul	200	28	2.019
30	B	appl	elit	200	0	2.019

Table 2-12: Runs leading to optimal program

The phenotypes are as follows:

Run 10

```
(GT (* V -1) (% X (ABS V)))
```

Run 22

```
(- (- (% -1 -1) (% X X)) (+ (% X (ABS V)) V))
```

Run 30

```
(% (* X X) (+ -1 (- (- (GT (* (- (GT V (- (* (- V (+ X -1)) (%
(GT V (ABS (- X X))) V)) V)) V) (ABS V)) X) (GT (* <sexp>
<sexp>) <sexp>)) (- <sexp> <sexp>))))
```

Although these phenotypes are syntactically quite different from each other and from the optimal program, the fact that they all have the same Std value over a sample of 20 initial conditions, and that they are all arithmetic expressions without any mechanism such as conditionals to define special cases in their input, strongly suggests that they are functionally identical. Based on Std values we claim that 14 different algorithms are discovered in the 48 runs.

2.5 Comparisons

The experiment was designed partly with the aim of comparing the performance of Gads 1 with that of SGP. The cart-centering problem is the first example of GP described in [Koza, 1992].

2.5.1 Phenotype length

On the whole the Gads 1 phenotypes do not resemble those of [Koza, 1992]. Koza's phenotypes are around 60 symbols long, Gads 1 phenotypes are around 10 symbols long.

2.5.2 Number of individuals

Koza's optimal solution is found in generation 33. With the same principles used to calculate Ind, the number of individuals is:

$$500 + 33 \times 450 = 15350 \ \textit{individuals}$$

The comparable Gads 1 figures from runs that lead to the optimal program are 3 750, 14 250 and 250 individuals (table 2-12). Run 30 can reasonably be discounted as a fluke that could only occur with an extremely simple problem.

2.5.3 Initial population

The discovery of the optimal program by pure chance in generation 0 of run 30 is noteworthy. Tree-based GP experimenters do not generally expect anything of value to arise in generation 0.

It is not obvious what, if anything, to make of this case. Clearly, no evolution was involved, because the effect is in the initial population. A possible explanation is that the discovery is connected with the relatively short length of programs Gads 1 generates. But the phenotype in run 30 is 41 symbols long — the longest phenotype over all runs.

2.6 Questions raised

The study has raised a number of interesting questions. These are outlined below. As well as these issues, it is clear that Gads 1 should be applied to a further range of problems.

2.6.1 Specifying sentence distribution

Many grammars exist for the same language. Gads 1 used different grammars as a way to specify the distribution of language strings. There are other ways of doing this; for example, by specifying a probability or weight for each alternative in a rule.

However, Gads 1 can achieve the same effect with a grammar in which the rules are replicated in proportion to their relative probabilities. A purist might object that a grammar is a set of rules so that replicating set members is futile, but this can easily be overcome by renaming each of the duplicates. The point is that both methods are equivalent, in terms of the distributions they can produce. Are all 'sensible' methods equivalent? Which is the most efficient in terms of programmer effort?

2.6.2 Moving away from Lisp

Much work has been done using Lisp as the phenotype language. The use of other languages should be investigated. A possible difficulty here would be the cost of evaluating fitness.

2.6.3 Functions, work variables etc

The definition of re-usable functions has been shown to increase the effectiveness of genetic programming. Methods for doing this with Gads 1 should be investigated. In addition, the use of variables other than input variables should be investigated. Such *work variables* are essential in real programs.

2.6.4 Choosing sentence distribution

Given a method of specifying a sentence distribution, how do we choose a good sentence distribution? A study of the rule frequencies used to produce real programs would be of interest.

2.6.5 Statistical analysis

The extent to which a statistical analysis of the result can be carried out is limited because we do not yet know much about the distribution the measurements are likely to have. For example, an analysis of variance to discover the relative importance of the experimental variables in determining the experimental measurements would be of value. This requires that the distribution of the measurements is known to be approximately normal. We hope to collect sufficient data to be able to use more powerful statistical tools in future analyses.

2.6.6 Sequential chromosomes

The essence of a sequence is that earlier genes can affect later ones but not vice-versa. However, this only becomes visible during the evaluation of fitness. Thus, in some sense, sequentiality is in the eye of the beholder. Is there an objective way to measure the amount of sequentiality in a chromosome?

2.6.7 Gene effectiveness

In the Gads 1 experiment, chromosomes of 200 genes were used, but nothing like that length of program was generated. It would appear that many of the genes are unused. Why is this? Is there a way to increase gene effectiveness? Is it a good thing to do?

As the population evolves, how does the pattern of active and inactive genes change? We might expect that as evolution proceeds, convergence advances along the length of the chromosome. Does this actually happen?

2.6.8 Genetic operations

GAGS uses uniform crossover, where each parent has an equal chance to contribute each gene to a child. We might expect that one-point crossover would be much more effective for Gads 1, given that it has a sequential chromosome. This may be related to the observation that many of the Gads 1 runs did not find a better solution than was found by random search in generation 0. One-point and other crossover techniques should be investigated.

2.6.9 Initial distribution

Gads 1 was able to discover the optimal solution in one of its initial generations. It would be interesting to compare the initial population produced by Gads 1 to that produced by [Koza, 1992].

2.7 Conclusions

This section draws conclusions from the experiment.

The experiment was to decide (i) whether Gads 1 is feasible and (ii) whether Gads 1 is worth developing further.

Gads 1 was implemented using a general-purpose GA engine that was not customised for Gads 1 in any way. In some ways it was far from what might be expected to be optimally tuned to Gads 1 requirements. The implementation of Gads 1 was about 520 lines of C, including comments and cpp directives. The implementation shows that the technique is simple and feasible.

The performance of Gads 1 on the cart-centering problem is good enough to confirm the feasibility of Gads 1. The three runs which discovered the optimal solution were reasonably efficient.

The conclusion is that Gads 1 is feasible and is worth developing further. The main limitations revealed by the investigation are in the areas of statistics and scalability to full-size languages. These issues are dealt with in §3 *Statistics* and §4 *Grammars.*

3 Statistics

This section deals with issues of statistical analysis which are raised in §2 *Gads 1*. The aim of this section is to identify and develop the statistical tools necessary for designing GP experiments and analysing their results. It is in 3 main sections.

The first section, §3.1 *Statistical perspective*, analyses GP from a statistical perspective, leading to a correspondence between statistical notions such as *experiment*, *population*, *sample* and *random variable* and GP notions such as *run*, *population*, *individual* and *fitness*. The criteria for a well-formed GP experiment are thereby established.

The second section, §3.2 *Performance comparison*, uses a conventional approach to investigate whether standard statistical techniques can reliably be used to compare performance measurements produced by GP systems. This puts performance comparison on a sound footing. It also reveals that the most important factor affecting performance is the problem.

The third section, §3.3 *Visualisation*, develops a technique for visualising a collection of points in a multi-dimensional space. The points can represent problems or GP system configurations. The technique, which is adapted from biology, uses path-length trees known as cladograms.

3.1 Statistical perspective

3.1.1 Introduction

This section presents a description of the key GP features in statistical terms. This is a necessary prerequisite to being able to apply statistical techniques to GP experiments, and it leads to some important conclusions. The introductory statistics presented here can be found in any statistics textbook, and is taken mostly from [Freund, 1979].

3.1.2 Populations and samples

In statistical terms, a *population* is a set of all conceivable or hypothetically possible observations of a phenomenon. Measures of populations (eg mean, standard deviation) are called *population parameters*. A *sample* is part of a population. Measures of samples (corresponding to population parameters) are called *sample statistics*.

A *sample design* is a scheme by which members of a population are selected for a sample. The simplest sample design is the random sample, where every member of the population has the same chance of being selected. Random samples need about 30 members to avoid the need for small sample techniques when comparing sample means. An important property of sample designs is that the members must be chosen independently of one another.

Where there is no theoretical basis for a sample design, the best that can be done is to select members of the population by educated guesswork. This produces a sample which can be called a *benchmark suite*. Random samples and benchmark suites are at opposite ends of a range of sample designs.

In GP terms the terms *population* and *generation* are used interchangeably, except when *generation* refers to the activity of creating new individuals. To avoid confusion, we henceforth use the term *population* only in its statistical sense, and use *generation* for the GP notion.

There are several populations of interest in this investigation:

- Configurations
- Runs
- Individuals in generation 0
- Individuals in generation $n > 0$

They are described below in more detail.

3.1.2.1 Configurations

3.1.2.1.1 Populations

The population of configurations is the population of GP engines and problems, in all their variety, with all their parameters. The extent, even the precise definition, of this population is unknown.

3.1.2.1.2 Samples

In choosing a sample of configurations, the best that can be done is to make an educated guess.

3.1.2.2 Runs

3.1.2.2.1 Populations

Given a configuration, each run is characterized by its RNG seed. The population of runs for a given configuration is thus equivalent to the population of RNG seeds. This is very convenient, since the population of RNG seeds is the natural numbers from 1 to whatever limit the RNG imposes.

3.1.2.2.2 Samples

It is thus simple to choose a random sample of runs for any given configuration. All that is needed is to seed the RNG for each run with a value derived, say, from the system clock.

3.1.2.3 Individuals in generation 0

3.1.2.3.1 Populations

The population of which generation 0 of a run is a subset is typically the set of all programs in a certain language (eg first-order Lisp) which satisfy certain properties (eg the number of nodes is in a certain range, the atoms are in a certain set, and so on). Individuals in GP are said to be *generated* rather than selected, typically according to a scheme such as ramped half-and-half.

Ramped half-and-half (chapter 6.2 in [Koza, 1992]) generates Lisp ETs. The term *ramped* means that equal numbers of ETs are generated with depths from 2 to the specified maximum. (The *depth* of a tree is the number of arcs from root to furthest leaf.) For example, if the maximum depth is 6, then there are equal numbers of ETs with depths 2, 3, 4, 5 and 6. The term *half-and-half* means that within the ETs of each depth, half are generated by the *full* method, and half by the *grow* method. The full method generates ETs in which all leaves are at the same depth. The grow method generates ETs in which leaves may be at any depth up to the maximum. The reason for choosing a scheme such as ramped half-and-half is pragmatic: it produces a wide variety of trees of various shapes and sizes.

3.1.2.3.2 Samples

In the case of generation 0 populations, the sample design is a generative technique such as ramped half-and-half. Generation 0 is therefore not a random sample, and it is not valid, for example, to use the mean fitness of generation 0 as an estimate of the mean fitness of the population. It is hard to estimate population parameters on the basis of a sample drawn with an intricate sample design such as half-and-half.

A way round this difficulty is as follows. We have a non-random sample drawn from a population by some sample design. This sample could equally well have been a random sample drawn from a different population. We can define this hypothetical population implicitly in terms of the generative technique. There is little point in describing the hypothetical population explicitly, because the only way we can actually draw samples from it is to generate them. This change of standpoint enables us to treat generation 0 as a random sample, provided we specify the generative technique.

The quality of the sample may be questioned on the grounds that all the individuals have been produced from one seed for the random number generator. There is some sense in this but ultimately it rests on a confusion. An RNG has two essential properties: one, that it is algorithmic and repeatable, and two, that the numbers it produces satisfy statistical tests for randomness. Consider simulating 1 000 throws of a die. We would not hesitate to use one seed to generate all 1 000 throws. It would make no sense to argue that each throw should have its own seed, since this would effectively mean doing without the RNG. The fact that each run starts from one seed is irrelevant: each call of an RNG returns a random number. The individuals in generation 0 are guaranteed to be statistically (but not algorithmically) independent of each other by the quality of the RNG.

An experiment comprises many runs, each of which has its own generation 0. Each of these can be treated as a separate sample, or they can be combined into a single sample. There is a benefit in separate samples, because it is possible to test the claim that they are all drawn from the same population. This can be used, for example, to test the hypothesis that the generation software in the engine is working correctly.

3.1.2.4 Individuals in generation $n > 0$

3.1.2.4.1 Populations

Each successive generation of a run is generated from the previous one using various operators, involving the fitness of the individuals. The population of individuals of which the generation is a subset is, like the population for generation 0, typically the set of all programs in a certain language (eg first-order Lisp) which satisfy certain properties (eg the number of nodes is in a certain range, the atoms are in a certain set, and so on). However the individuals in generation $n > 0$ are not generated in the same way as those in generation 0, and the *certain properties* they satisfy need not be the same as those in generation 0. For example, individuals in generation $n > 0$ may be allowed to have more nodes than those in generation 0.

3.1.2.4.2 Samples

If we attempt to treat generation $n > 0$ as a sample in the same way as we did for generation 0, we run into serious problems, because the individuals are not independent of each other.

To see why this is so, consider two runs of a configuration, each of which converges completely by the same generation, but each run converging to a different genotype. If these generations are samples, what population are they drawn from?

One approach is to say they are drawn from the same population but the sample design is different in each case. This implies that we have a different sample design for each generation number and each RNG seed.

Another approach is to change our standpoint as we did for generation 0. We say that the populations are different, but the samples are random. This simplification lets us estimate population parameters; but the conclusion is not very useful, because we have a different population for each generation number and each RNG seed.

Analysis on the basis of either of these approaches would be difficult, to say the least. The source of the difficulty is that individuals after generation 0 are not independent — no surprise, they have been interbreeding for n generations. We conclude that generation $n > 0$ is not a sample of any population we can deal with.

To deal with generation $n > 0$ as a sample it is necessary to treat each generation $n > 0$ as one individual, which is a random variable derived from the run's RNG seed. A study of a generation $n > 0$ is essentially a study of a sample of 1. It is possible to compute, say, the mean of the *n*th generation fitness, but that mean is not very informative, because another run will produce a different mean. To study the *n*th generation mean, the experimenter should produce a sample of *n*th generation means, and study them as a sample.

3.1.3 Experiments, outcomes, units and treatments

In statistical terms, any process of observation or measurement is referred to as an *experiment*, and the results, whether made through simple observations or extensive calculations, are the *outcomes* of the experiment. Experiments are often described in terms of applying *treatments* to *units*. (This terminology arose from agricultural experiments, for example, applying different levels of fertilizer were applied to plots of land.)

From the discussion of §3.1.2 *Populations and samples*, it is clear that the soundest choice for a statistical experiment is the *run*. A run is characterised by a *configuration* consisting of:

- A GP engine, in a particular configuration. For example, Lil-GP, with a generation size of 50, ramped half-and-half initialisation, depth limit 30, and genetic operators replicate 9%, crossover 90% and mutate 1%. The engine is the treatment.
- A problem, defined by an objective (or fitness) function, in a particular configuration. For example, the lawnmower problem, for a particular size and shape of lawn. The problem is the unit.

A run also needs an RNG seed. The RNG seed is not considered to be part of the configuration. This is discussed in the next section.

A configuration is thus a collection of experimental conditions, which can be a data structure of arbitrary complexity. The work described in this thesis consists of many such experiments.

It is tempting to partition the configuration into a pair of sub-configurations, one for the problem and one for the engine, perhaps by defining a parameter to be a problem parameter if it can be observed in the solution; and otherwise to be an engine parameter. For example, the terminal set is a problem parameter, but the genetic operators are not. But it is not always clear which parameters are associated with the problem and which with the engine. For example, it might seem natural to associate generation size with the engine. But even in [Koza, 1992], where there is a deliberate attempt to use the same engine parameters for all of the examples (so that the same engine configuration is used throughout), it is necessary to give the Boolean 11-multiplexer a generation size of 4000 instead of the usual 500. Does this mean that generation size is a problem parameter? Certainly not; using different technique we can solve the Boolean 11-multiplexer problem with no generation size parameter at all. For example, random search or human design are methods which have no generation size parameter. This parameter therefore belongs to the engine. Knowing what value to set it to requires a very special kind of knowledge about the problem, and this is one area of weakness in GP paradigm. The existence of these problem-specific engine parameters means that for the time being, we cannot usefully split a configuration into a problem part and an engine part. It is however reasonable to identify a configuration by the problem and engine involved.

In addition to the problem and engine, the operating system, hardware and so on may be necessary for a complete description of an experiment. These factors are constant for this investigation.

The termination criterion is an experimental condition, in that it affects the resulting observations, such as the fitness of the best-of-run individual. But it is unlike other experimental conditions, in that it does not alter the way a run progresses. If a run is like a sequence, then the termination criterion establishes how many terms there are, but it does not affect the rule for computing each term from the previous one. Ideally, therefore, termination criteria should not be treated in the same way as other configuration components, but it has not been given special treatment in published papers before now, so I do not give it special treatment here either. This is a topic for future investigation.

3.1.4 Random variation

Much of traditional statistics is concerned with the natural variation that occurs between units (problems, in GP) and treatments (engines, in GP). For example, if you apply the same level of fertilizer to two seeds, they won't produce the same yield. This is due both to differences in the treatment (it is not possible to get it exactly the same twice) and to differences in the unit (no two seeds are identical). Nor is it possible to apply a treatment to the same unit more than once.

These limitations do not apply to computer simulations such as GP. It is possible, for example, to re-use a sequence of random numbers. This technique is known as CRN and is one of several methods of variance reduction described in [Bratley, 1983]. When CRN is used, the experiment produces a sample of pairs rather than a pair of samples. A sample comprising the (signed) differences between the two values of each pair is computed, and tested against the null hypothesis that it is a sample from a population whose mean is zero. This form of testing is called a *paired* test.

For CRN to be of use certain pre-requisite conditions must be met, the main one being that the random numbers are used for the same purpose in the models being compared. This can require the use of multiple random number streams so that each stream is used for one purpose.

For example, consider the design of an experiment to investigate the effect of doubling chromosome length by comparing the performance of two GP engines over a range of problems, using one random number stream, with the same seed in each pair of configurations (one with double the chromosome length of the other). This experimental design is not sound, because the same random numbers are not used for the same purposes. The first use of the random numbers is to produce generation 0. But the configuration with the double chromosome length must use twice as many random numbers as the other configuration. By the time the first individual in generation 0 has been produced, the random number streams are out of step. Despite using the same RNG seed the experiment is not a CRN design. In order to use CRN, this experiment would have to be redesigned, for example, by having one random number stream to produce genes, and another to drive the selection of individuals for breeding.

It can be seen that CRN is an option only if the model is well-understood, particularly in its use of the random numbers. The possibility of error is high because it depends on algorithmic analysis, which is a difficult subject, not least when analysing a system as complex as a GP engine developed by a third party. The chance of detecting error is low because an error of this sort does not manifest itself in output that is blatantly wrong. The effects are subtle and may be revealed only by careful testing.

In this thesis, CRN is not used. One RNG stream is used for both treatments and units. Provided the RNG works, and is being used correctly, this should not introduce any artifacts into the results. The function of an RNG is precisely to produce a stream of unrelated numbers. There is no advantage in using two RNG processes instead of one, and in fact, doing so introduces the risk of spurious results.

We therefore define the RNG seed as separate from the configuration.

3.1.5 Confidence level

Statistical results are generally not 100% certain because of the effects of random variation. For example, the mean of a sample can be calculated with 100% accuracy, but it is not the mean of the underlying population. The most we can say about the population mean is that it probably lies within certain limits. We can be precise about the limits and about just how probable it is.

The measure of probability is the *confidence level.* A confidence level of 95% is usual for experimental work in the natural sciences, where it is seen as a compromise between a desire for increased confidence and a desire to keep the confidence intervals small enough to be useful.

In order to make the results of different parts of this thesis compatible, we assume that the same confidence level applies to all observations and in all hypothesis testing. This level is called the global confidence level (GCL), and is set at 95%.

3.2 Performance comparison

This section presents the results of an investigation into the reliability of statistical tests. Much of the material in this section is taken from [Paterson, 2000].

3.2.1 Introduction

A weakness of the Gads 1 experimental analysis is the way in which the new technology is compared with the old, namely, by comparing the means of two rather small samples. While this may be sufficient to indicate that further investigation of Gads 1 is justified, it is not sufficient to enable us to draw reliable, quantified conclusions.

The reason for this weakness is that we know next to nothing about the statistical properties of performance as a random variable. In this section, we report the results of an investigation into the reliability of statistical tests (Student's, Smirnov's and Randomisation) when they are used to detect differences in GP performance data. Lil-GP and SGPC are used to provide test data. The main conclusions are: (a) that parametric tests perform better, and non-parametric tests worse, than expected; (b) that the reliability of the tests depends mostly on the problem being solved; and (c) that no test can reliably detect a difference less than 1.5 × coefficient of variation with GP data.

Performance comparison is central to research in GP. For example, 22 of the 55 papers in [pp3–343, Koza, 1997a] compare performance of rival systems. In practice, performance comparison usually comes down to deciding whether the difference between the means of two samples of observations is significant. Many methods are used to reach this decision. Of the 22 papers mentioned above, 16 used visual comparison of graphs, two used Student's T test, two used Koza's cumulative probability of success [Koza, 1992], 1 used the 2-sample Z test, and 1 used the Mann-Whitney U test.

A common error is to compare sample means as if they were exact population means, not estimates of population means. There is a 95% probability that the interval *sample mean* ± (1.96 × *standard error*) contains the population mean. Standard error is described in statistics textbooks, eg [Freund, 1979]. Only two of the 22 papers gave standard error or enough information to work it out. Treating sample means as point estimates is the main flaw in Koza's metric *cumulative probability of success* [Angeline, 1996a].

4 of the 22 papers used recognised statistical tests. However, T and Z tests have preconditions. Student's T test requires that the two populations have close to normal distributions, and that their variances are equal. The

sample size can be small. The Z test requires a sample size of at least 30, that the two populations have close to normal distributions, and that their variances are equal.

We find that normal distributions are the exception. An example of a performance distribution is shown in figure 3-1. Performance is on the x-axis, and frequency with which that performance occurs, on the y-axis. The code LOAO identifies the engine and problem configuration, as explained in §3.2.2.2 *Factors* below. This configuration is at the mid-range of normality among the distributions we examined (see §3.2.3.1 *Normality* below) and is rejected at the 5% level as non-normal.

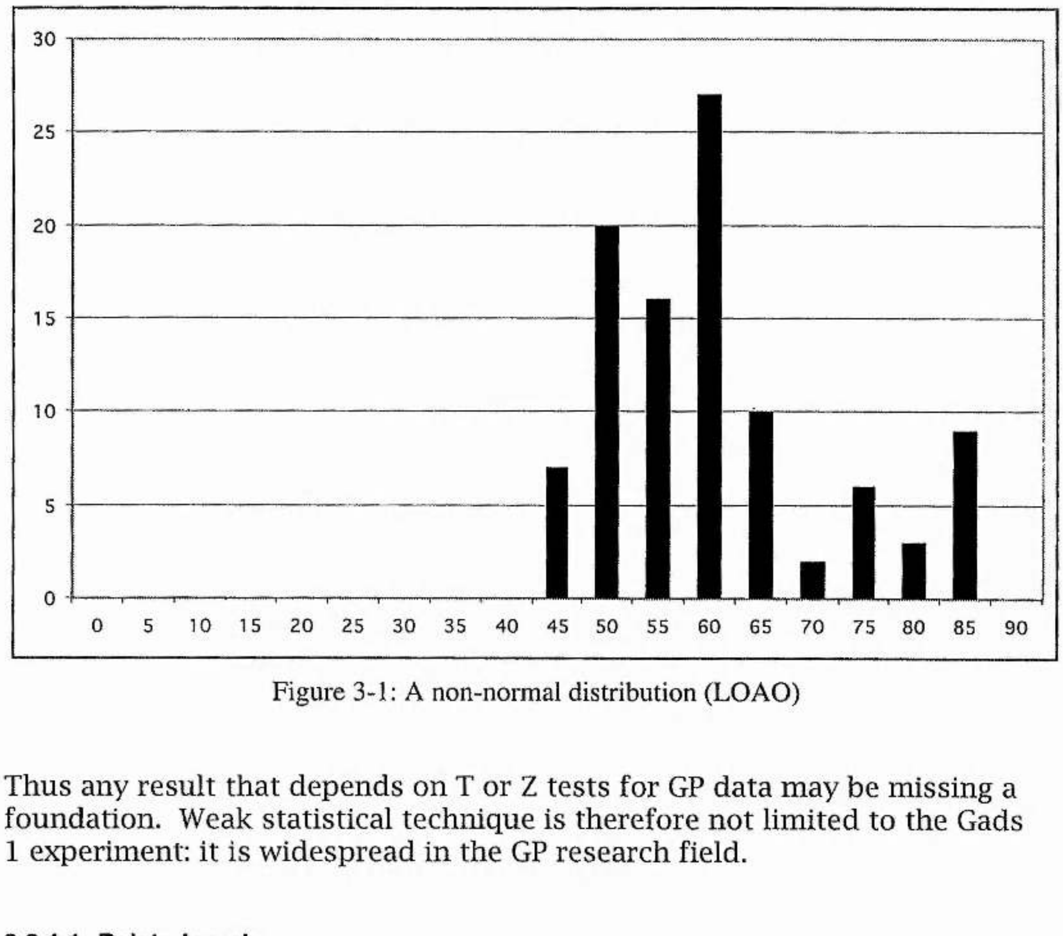

Figure 3-1: A non-normal distribution (LOAO)

Thus any result that depends on T or Z tests for GP data may be missing a foundation. Weak statistical technique is therefore not limited to the Gads 1 experiment: it is widespread in the GP research field.

3.2.1.1 Related work

[Daida, 1997] addresses general issues of the difficulty of comparing results or repeating others' work. In their own comparison, they use the Mann-Whitney U test, which is not so dependent on population properties as the T or Z tests; but they do not attempt to investigate the population properties.

[Lawrence, 1997] considers distributions of performance in neural network simulations, and how to present results for a fair interpretation.

[Luke, 1997] reports an investigation of about 12 000 independent runs of a range of engine-problem configurations, with a sample size of 25. The configurations were chosen to compare crossover and mutation. The results are described as "nonlinear," and because the effects are shown to be significant by Student's T test, the paper argues that they cannot be attributed to noise. However, the large number of comparisons means that some of the results (5% of them, if a confidence level of 95% is used) are almost certainly caused by noise. It is impossible to say whether the noisy results are the nonlinear results. However, their study does bequeath a large corpus of runs.

[Luke, 1998] is a revision of [Luke, 1997], in which a number of statistical flaws are put right.

3.2.2 Experimental design

The experimental design is at two levels. At the upper level we investigate the reliability of statistical tests. At the lower level we investigate GP performance data. The performance data is produced in such a way that we know what the results of the lower level investigation should be. The observed results of the lower level investigation are input to the upper level investigation. The two levels are summarized in table 3-1.

	Upper level	**Lower level**
Phenomenon	Statistical testing.	GP.
Factors	Test, sample size.	Engine, RNG, problem, parameters.
Trial	Application of a statistical test.	Execution of a GP system.
Outcome	Acceptance or rejection of a given null hypothesis.	Files summarizing evolution process.
Random variable	*Accept* or *reject.*	Performance, ie raw fitness of solution.
Experiment	30 trials, same except for RNG seed.	10, 20 or 30 trials, same except for RNG seed.
Statistics	α and δ	(none)

Table 3-1: Outline of Experimental Design

The description in the following subsections starts at the upper level, drills down to the bottom of the lower level, and then returns to the upper level to fit pieces together. The experiment involves several different kinds of sample, taken for different purposes. To help avoid confusion, these are referred to as type A-, B- or C-samples as follows. A-samples are size 100, and are used for bootstrapping as described in §3.2.2.5 *Bootstrapping.* B-samples are size 30, and are used for the upper level experiment. C-samples are size 10, 20 or 30 and are used for the lower level experiment.

The various components of the investigation are identified by codes, eg *T* for *Student's T test.* These codes are given in parentheses in the next two subsections.

3.2.2.1 Tests and C-sample sizes

We chose to investigate 3 tests that span a range of statistical techniques:

Student's T test (T)
The classic experimenter's tool for comparing small samples. The samples must be from approximately normal populations with the same variance.

Smirnov's 2-sample test (S)
A non-parametric test which does not assume that the populations have any particular distribution.

Randomisation test (R)
A modern computer-intensive test.

Each test was used to compare two equal-sized C-samples of 10, 20 or 30 observations.

3.2.2.2 Factors

We chose to investigate the effect of the following factors on the tests:

engine
Either Lil-GP (L) or SGPC (S).

RNG
Either the engine's own (O), or the Solaris RNG (S).

problem
One of the problems distributed with the engine. Lil-GP has Artificial Ant (A), Lawnmower (L), Multiplexer (M), Regression (R) and Two Box (T). SGPC has Classifier (C), Donut (D), Regression (R) and Sin (S). (Regression (R) occurs twice.)

parameters
Each engine's own default settings (O) were used where possible. In addition, a parameter set was devised with the aim of replicating the Regression problem in [Koza, 1992] on both Lil-GP and SGPC (K).

A complete set of all 4 factors is called a *configuration.* Each configuration has a 4-letter code, eg LOAO. A configuration specifies all the information necessary to run a GP system, except the RNG seed. Although 64 configurations are possible, only 22 actually occur. This defines a relation between the factors, ie the factors are not independent.

3.2.2.3 Trials and outcomes

One lower-level trial is the execution of one GP system with one configuration and one RNG seed. The outcome of a trial is the collection of files which it produces. This varies from one engine to another, but typically contains a summary of the simulation which has been carried out.

For any given configuration, the C-sample space (ie population of possible outcomes) is therefore equivalent to the population of RNG seeds. It is simple to choose a random C-sample of RNG seeds.

3.2.2.4 Random variables

We chose to investigate just one random variable at the lower level, namely *performance.* This term is defined as *the raw fitness of the first best-of-run individual.* It is extracted from the outcome of each lower-level trial.

There is no deep theoretical basis for this choice. *The decision to measure one thing or another as a performance variable is often intuitive, and rarely is a strong argument made for or against a particular metric.* — [Cohen, 1995]. We chose fitness because the fitness of the best individual found in a run is surely the primary goal of the run. Raw fitness is in some sense closest to the source.

3.2.2.5 Bootstrapping

Rather than use the experimental apparatus described above to produce C-samples directly, we chose to use bootstrapping to reduce the computational effort. Bootstrapping simulates C-sampling from a large or infinite population by C-sampling-with-replacement from an A-sample of that population.

Bootstrapping has two advantages. One, the expense of producing C-samples is greatly reduced, since C-sampling-with-replacement re-uses values that are expensive to compute. Two, the properties of the simulated population can be manipulated and known with certainty. The disadvantage is that we are dealing with a simulation of a population, not with the real thing.

To support the bootstrapping procedure, we made 100 trials for each configuration, and extracted the performance data from each. This gave us 22 A-samples of 100 observations. These A-samples were named with the 4-letter configuration code, eg LOAO. C-samples of 10, 20 or 30 performance values were drawn with replacement from the A-samples.

3.2.2.6 Hypotheses

Each test was used to accept or reject the following null hypothesis at a 5% level of significance:

H0 *That there is no difference in the means of the populations Π1 and Π2 from which the C-samples were drawn.*

on the basis of two equal-sized C-samples of performance data. Note that Π1 and Π2 refer to the simulated populations rather than the A-samples, since the use of bootstrapping is transparent to the tests.

The alternative hypothesis, H1, was always the simple negation of H0. We did not attempt to decide which of the populations has the larger mean, only that there is a difference. Thus, the two-tailed form of each test was always used. H1 is a composite hypothesis, since there are many ways in which two populations can differ.

To produce a pair of C-samples for which H0 is true, we draw them both from the same A-sample.

To produce a pair of C-samples for which H0 is false, we first draw the first C-sample in the ordinary way. We draw the second C-sample from the same A-sample, but add a displacement d to each observation. The second C-sample is thus effectively from a population which is translated by a known amount. We measure d as a percentage of the mean of the A-sample. For example $d = 5$ implies that the C-samples are drawn from populations Π1 and Π2 whose means are $\mu 1$ and $\mu 2 = \mu 1 \times 1.05$. If $d = 0$, then $\Pi 1 = \Pi 2$ in all respects. Populations Π1 and Π2 always have equal variance. We chose the following range of values for d: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100}.

We chose only to investigate differences in means, because that is the difference researchers are most often interested in detecting. It is also the only difference that Student's T test is intended to detect, and in some sense it is the simplest difference.

A combination of a test, a C-sample size, a configuration and a value for d is called a *scenario*. A scenario specifies everything that is needed to apply a test, except for an RNG seed to drive the bootstrapping process. A scenario is a trial at the upper level, the outcome of which is either *accept* or *reject*.

3.2.2.7 Power functions

There are two ways in which a test can fail. To reject H0 when it is true (a false negative) is a type 1 error. To accept H0 when it is false (a false positive) is a type 2 error. We chose to compare the tests on the basis of the occurrence of these errors.

The probability of a type 1 error is conventionally denoted α and, if the test's preconditions are satisfied, is equal to the level of significance, in this case 5%. Since the test's preconditions are in doubt, we estimated the probability of a type 1 error empirically and compared it with the theoretical value of 5%.

The probability of a type 2 error is conventionally denoted β. β is not a unique number, because H1 is composite: there is a different value of β for each $\Pi2$. However, since we restricted ourselves to $\Pi2$s that are characterized by a parameter d, β reduces to a function of d. Such a function is called a *power function* [Conover, 1971]. Figure 3-2 shows a typical power function shape.

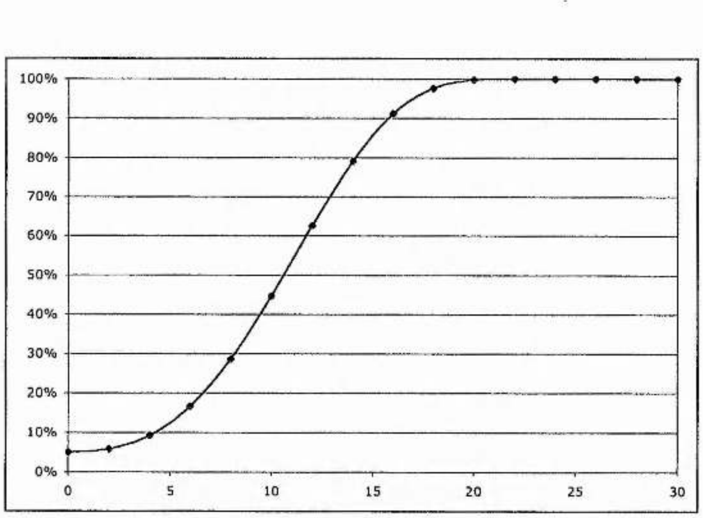

Figure 3-2: Typical power function shape

The power function shows the probability of rejecting H0 as a function of d. For $d = 0$, the power function is α. For $d > 0$, the power function is $1 - \beta$.

The power function is thus a single concept that describes the behaviour of a test over the range of conditions we are interested in. The true or theoretical power function is not within our grasp. We estimate the power functions empirically by executing each scenario 30 times (with different

RNG seeds) and recording how often H0 was rejected. The proportion of rejections, as a percentage in [0, 100], is the value of the EPF at that scenario.

3.2.2.8 Alpha and delta

Power functions, though informative, do not make for simple comparison of tests. All the power functions examined are well-formed. They have approximately the same shape as figure 3-2, but may be more or less extended along the d-axis. It is therefore feasible to use a scalar metric to distinguish them. We introduce the metric δ, defined to be the least value of d for which β is less than or equal to 5%. That is, as d increases from 0, the test becomes more reliable in detecting a difference of d% of the mean. δ is the value of d at which that reliability first reaches 95% (corresponding to a 5% level of significance). Putting it the other way round, the test cannot reliably detect a difference of less than δ.

Thus, α and δ are the upper-level random variables. The aim of the investigation is to discover how they depend on the experimental factors.

3.2.3 Results

These results are necessarily a summary. Full details are at ftp.dcs.st-and.ac.uk/pub/norman/StatDist.tar.gz.

4 of the A-samples (SOCO, SORO, SSCO and SSRO) contained only zeroes. This appeared not to be a bug but arose because the perfect solution (with fitness 0) was discovered every time. Since this effectively removed the need for statistical analyses at the same time as rendering it impossible, we removed these A-samples from further consideration and continued with the remaining 18 configurations.

3.2.3.1 Normality

The Kolmogorov-Smirnov normality test was applied to each A-sample, against the alternative hypothesis that the sample was not drawn from a Normal population. Table 3-2 shows the value of the Kolmogorov-Smirnov test statistic (k) for each configuration. The rows are sorted by k. The critical value of the statistic is 0.136 at the 5% level. Configurations are on the left or right of the table depending on how their k value compares to 0.136.

Accept		Reject	
Config	k	Config	k
LOTO	0.05	LSRO	0.14
LSTO	0.06	SODO	0.15
SSSO	0.10	LORO	0.15
SOSO	0.12	LSLO	0.17
LSAO	0.12	LOAO	0.17
SSDO	0.12	LOLO	0.17
		SSRK	0.18
		LSRK	0.20
		LORK	0.20
		SORK	0.21
		LOMO	0.27
		LSMO	0.30

Table 3-2: Kolmogorov-Smirnov normality test results

3.2.3.2 Alpha and delta

The observations of α and δ are shown in tables 3-3 and 3-4.

	T			S			R		
	10	**20**	**30**	**10**	**20**	**30**	**10**	**20**	**30**
LOAO	2	1	0	2	0	3	1	3	1
LOLO	2	3	2	2	6	16	1	1	1
LOMO	2	0	0	14	20	27	3	0	2
LORO	2	4	3	2	2	5	2	1	1
LORK	1	0	0	0	2	7	1	5	2
LOTO	1	1	1	3	3	0	1	1	1
LSAO	2	1	2	1	1	2	0	0	3
LSLO	1	0	1	2	5	14	0	1	1
LSMO	0	1	2	13	21	27	1	2	1
LSRO	2	4	2	0	1	2	1	2	2
LSRK	2	2	0	2	0	6	1	0	1
LSTO	2	2	2	2	0	2	1	3	0
SODO	1	0	0	0	0	1	1	1	2
SORK	2	0	3	1	3	9	0	2	1
SOSO	2	1	1	0	0	0	0	0	0
SSDO	3	5	3	0	0	0	2	1	2
SSRK	2	0	3	2	1	2	0	1	2
SSSO	3	1	1	1	1	1	1	1	1

Table 3-3: Observed values of α (actually rejection counts in B-samples of 30)

	T			S			R		
	10	**20**	**30**	**10**	**20**	**30**	**10**	**20**	**30**
LOAO	40	30	20	40	20	20	40	20	20
LOLO	5	3	3	7	4	2	6	4	3
LOMO	4	3	2	4	2	1	5	3	2
LORO	200	100	100	200	90	80	200	200	100
LORK	200	200	200	200	90	70	200	200	200
LOTO	80	40	40	80	60	40	70	50	40
LSAO	30	20	20	40	30	20	40	20	20
LSLO	5	3	3	5	3	2	5	3	3
LSMO	4	3	3	4	3	1	4	4	3
LSRO	200	200	90	200	200	80	200	200	100
LSRK	200	200	200	200	100	80	200	200	200
LSTO	80	60	40	80	50	40	80	50	40
SODO	60	40	30	70	40	30	50	40	30
SORK	200	200	200	200	90	60	200	200	200
SOSO	20	20	10	20	10	7	20	20	8
SSDO	30	20	30	30	30	20	40	30	20
SSRK	200	200	200	200	90	70	200	200	90
SSSO	20	10	8	20	10	8	20	20	10

Table 3-4: Observed values of δ

The columns are by test and C-sample size. The rows are by configuration. The values of table 3-3 are numbers of rejections in a B-sample of 30 tests with the given configuration. The values in table 3-4 are percentages. For some configurations, the empirical power function (EPF) value of the test never reached 95% . For these, δ was set to an arbitrary 200%.

No standard error is given for the values in tables 3-3 and 3-4, but each value is based on a B-sample of 30 observations.

3.2.3.3 Alpha

We tested the hypothesis that the observed values of α from table 3-3 are not significantly different from the theoretical value of 5% as follows.

First, recap the basis of the experiment. Two equal-sized C-samples were drawn from the same population and a test was applied to decide whether the C-sample means were significantly different. A *rejection* resulted if the test concluded that the C-samples were drawn from different populations. This was repeated 30 times for each of 162 cases: 18 populations (LOAO to SSSO), 3 tests (T, S and R) and 3 C-sample sizes (10, 20 and 30). The number of rejections was counted in each case, being a value in the range 0 to 30. To convert a rejection count to an α value, divide by 30.

Under the null hypothesis, the only reason for a rejection is random variation within the 5% confidence level at which the tests were applied. Therefore, we can expect the number of rejections to have a binomial distribution with a size $n = 30$ and probability $p = 5\%$, and we can use a χ^2 test to compare the observed distribution of rejections with the expected distribution. If the difference is too great, we shall have reason to reject the null hypothesis on which the expectations were based.

The complete set of observations in table 3-3 gives the following results:

#	o	e	$(o-e)^2/e$
0	37	34.7714798	0.14282689
1	50	54.9023365	0.43773916
2	42	41.8991515	0.00024274
3	15	20.5820393	1.51390067
4+	18	9.8449929	6.75512321
Total	162	162	8.84983268
		DF	4
		χ^2 5%	9.488
		Conclusion	Accept

Table 3-5: χ^2 test for T, S and R

Column # gives the category labels **0** to **4+** which are the number of rejections in each B-sample of 30. Category **4+** includes 4 or more rejections, which are combined to keep the expected number in each category at 5 or more. Column **o** shows the observed frequency of the number of rejections in each category. For example, 37 out of 162 cases had 0 rejections, 50 had 1 rejection, and so on. Column **e** shows the corresponding expected frequency. The final column **$(o-e)^2/e$** shows the computation of the test statistic for each category.

The **Total** row gives the total of the columns above it. The rightmost entry in this row is the test statistic. **DF** is the degrees of freedom, which is 1 less than the number of categories. The χ^2 figure is the critical value at 5%.

Since the test statistic 8.84983268 is less than the critical value 9.488, the conclusion is that we have no reason to reject the null hypothesis.

Tables 3-6, 3-7 and 3-8 below give the corresponding analysis for each of the three tests separately:

#	o	e	$(o-e)^2/e$
0	12	11.5904933	0.01446839
1	14	18.3007788	1.01070554
2	18	13.9663838	1.16494431
3+	10	10.1423441	0.00199775
Total	54	54	2.19211598
		DF	3
		χ^2 5%	7.815
		Conclusion	Accept

Table 3-6: χ^2 test for T alone

#	o	e	$(o-e)^2/e$
0	14	11.5904933	0.50090386
1	9	18.3007788	4.72681996
2	13	13.9663838	0.06686754
3+	18	10.1423441	6.08762194
Total	54	54	11.38221330
		DF	3
		χ^2 5%	7.815
		Conclusion	Reject

Table 3-7: χ^2 test for S alone

#	o	e	$(o-e)^2/e$
0	11	11.5904933	0.03008347
1	27	18.3007788	4.13514910
2	11	13.9663838	0.63004377
3+	5	10.1423441	2.60725750
Total	54	54	7.40253385
		DF	3
		χ^2 5%	7.815
		Conclusion	Accept

Table 3-8: χ^2 test for R alone

Table 3-5 shows that we cannot reject the null hypothesis overall, but not by much. The test statistic is close to the critical value. When we look at the performance of the three tests individually, it is clear that Student's T test performs well, randomisation is borderline, and Smirnov's test fails.

Although the overall result is not to reject the null hypothesis, it is based on a kind of average that has no real meaning. In practice, we would use one test, not an average of three. The investigation shows that the choice of test is critical. We conclude that in general, the populations are not sufficiently normal to meet the requirements of the tests.

3.2.3.4 Delta

Factor	Fα	Fδ	F	α	δ	αδ
Engine	5.45	0.04	3.84	*		
RNG	0.02	0.00	3.84			
problem	6.94	121.03	2.10	*	*	*
parameters	0.76	164.41	3.84		*	
Test	10.32	1.04	3.00	*		
Sample size	1.69	2.83	3.00			

Table 3-9: Analysis of variance of α and δ

In table 3-9, Fα is the F value resulting from the analysis of variance applied to the α values of table 3-3. Fδ is the corresponding figure from table 3-4. F is the critical value at 5% for the appropriate degrees of freedom. A * in columns headed α, δ and αδ shows which factors have an effect on α, δ or both.

These results cannot be taken at face value because the factors are not independent. Only 18 of 64 configurations exist, which defines a relation between engine, RNG, problem and parameters. If only one of these factors is to be considered, then table 3-9 suggests it should be problem, which is the only factor that affects both α and δ. The rows for engine and parameters are therefore discounted.

The only effective factors are the problem and the test; and of these, the problem has the most effect. The importance of the problem as the determining factor suggests that some of the results should be revisited to take that into account. For example, if the Configurations in table 3-2 are pooled by problem, we find that two (S, T) pass, two (A, D) are borderline, and 3 (R, L, M) fail the normality test.

Despite our misgivings about the use of anova, it appears to have performed well. The results agree with our subjective view of tables 3-3 and 3-4, and correctly identified related factors.

3.2.3.5 Coefficient of variation

To discover whether a cheaper predictor of δ could be found, we investigated several measures of dispersion of the large samples. Using the coefficient of variation (V), we found a correlation of 99%. Figure 3-3 shows the relation with the least squares fit:

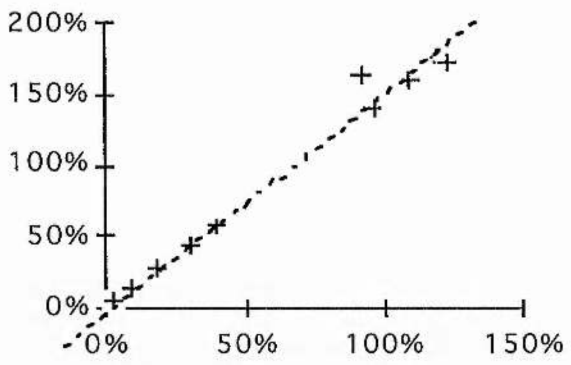

Figure 3-3: mean δ (y axis) vs V (x axis)

V is taken from the large sample of a configuration; and δ is the corresponding row mean from table 3-4. (Standard error of the row means is not shown but can be computed. It grows from about 0 to about 19%.) The graph suggests a linear relationship, with an outlier at V=92%. A least squares fit gives:

$$\delta = 0.0103 + 1.47 \times V$$

This result must be treated with some caution because it uses the arbitrary value of 200% for missing data when calculating δ. However, using any value from 125% to 477% for missing data gives a correlation of 99% so it is not particularly sensitive.

It is obviously no surprise that V correlates with δ. It is not possible to detect a difference that is small relative to the variability of the sample. To have a context for this result, we also generated 3 normally distributed large samples with mean 50 and standard deviations of 1, 5 and 50. For the normal data, the factor was about 1.7, compared to the 1.47 for GP data. That is, it was easier to detect differences in GP data than in normal data.

3.2.4 Conclusions

The aim of this study was to investigate whether GP perfomance data is normally distributed, and if not, how statistical tests behave when faced with such data.

Some allowance must be made in these conclusions for aspects of the experiment. First, the use of bootstrapping and displacement of the mean to generate data means that the data is partly simulated. Second, an arbitrary value of 200% was used for missing values in the calculation of δ. Third, analysis of variance, used to produce table 3-9, assumes that the populations are normally distributed with equal variance. This assumption was not confirmed, though the results of the anova (analysis of variance) appear to be reasonable.

3.2.4.1 Normality

GP performance data is not always normally distributed; as suggested by table 3-2, it may be the case that normal distributions are the exception.

3.2.4.2 Comparison of tests

The α metric appears to be a useful measure of a test's reliability. The δ metric does not have such a sound theoretical foundation, and although plausible, should be viewed with caution.

In broad terms, there is not as much difference between the tests as we expected. Table 3-9 shows that only α is affected by the test, but α appears to be affected more by the problem than by the test. There is a wide variation in δ, but again, table 3-9 shows that these are effects of the problem, not of the test.

Smirnov's test performed surprisingly poorly. This is probably due to the nature of the Multiplexer performance distribution. This distribution has several large spikes (eg at 1280) which result in many duplicate values in samples drawn from it. Smirnov's test expects a continuous distribution. (A continuous distribution is one in which the probability of any one outcome is vanishingly small, and the probability of an outcome falling within a range can be determined by integration. For example, the probability that a car is travelling at exactly 30 miles per hour is infinitesimal, but the probability that it is travelling between 25 and 35 miles per hour is finite.) The Multiplexer distribution is discrete, not continuous, as evidenced by the many duplicate integer values. Note that for a distribution to be continuous, it is not sufficient for the outcomes to be real-valued. Performance measurements of many GP systems are real-valued, but the values are drawn from a relatively small set, so they are in fact discrete. Strictly speaking, all digital simulations are of finite accuracy and are therefore discrete. Whether this leads to problems with a test such as Smirnov's may depend on the number of duplicate values in the samples.

Compared to either Smirnov's or Student's tests, the Randomisation test is expensive to compute, and difficult to program in such a way that the program can be independently tested. To obtain an independent test of the software (ie to ensure that your implementation is correct) it is necessary to produce output in test conditions such that you know what the output should be. With an implementation of, say, Student's test, this is fairly straightforward. You can set up test data and compute the test statistic in a few lines of code, or on a spreadsheet. To achieve better independence, it is easy to do this on a different computer, and even to engage the help of a colleague to guard against errors in your own understanding. With Randomisation, it is very hard to achieve this degree of independence. To compare outputs, it is necessary that the outputs are identical to the bit - level. This requires that the computers use the same RNG and have the same floating point representation. Detecting errors in stochastic data is hard, because they are usually not visible to the naked eye. The net result is that implementing Randomisation twice, independently, for the purpose of testing the implementation, is considerably harder than can be justified when simpler alternatives are available.

On the basis of this study, Student's test is our preferred test.

3.2.4.3 Comparison of problems

The problem was the main factor in determining the reliability of any test, and table 3-5 shows that its main effect was on the test's δ value.

The 99% correlation between V and δ shown in figure 3-3 suggests that almost all of the effect of the problem can be explained in terms of the dispersion of the performance distribution.

In short, how well a test can detect a difference between C-sample means depends almost entirely on the problem used to produce the C-samples. As a rule of thumb, no test can reliably detect a difference between means less than $1.5 \times V$. Observed values of V range from 2% to 123%.

3.2.4.4 Summary

We had expected that the non-normality of the data would adversely affect all parametric tests, specifically Student's T test and anova. We had expected to be able to demonstrate the superiority of a non-parametric test such as Smirnov's when dealing with GP data. In fact, we found the exact opposite. Not only does Smirnov's test perform unacceptably, but Student's test performs better with GP data than with normal data. Although anova was not the main subject of the study, we have kept the anova result because its behaviour seems to be well within the acceptable range.

While this may appear to be good news it is unsettling. When a test's assumptions are not met, we cannot be sure how the test will perform. It may reject a null hypothesis not because it is false, but because the data show an assumption to be false.

3.3 Visualisation

This section presents a method for comparing GP systems or problems on a single rational scale.

3.3.1 Introduction

§3.2 *Performance comparison* shows how to make pair-wise comparisons of configurations. It also shows that the choice of problem is the most important factor affecting the performance of a configuration. Ideally, we should like to be able to average out the problem and obtain a measure of the engine's power; ideally, this would be a number on a rational scale. Unfortunately there are sound reasons why this ideal cannot be achieved in full; but there is no reason it cannot be approached, and this section presents a method for doing so.

The reasons the ideal cannot be achieved are (1) the No Free Lunch theorems [Wolpert, 1995], [Wolpert, 1997] and (2) the sample design for configurations. The No Free Lunch theorem tells us that all engines perform equally well when their performance is averaged over all possible problems. Of course, we are only dealing with a sample of the problem population, which is presumably why there is such variation in engine performance. The problem, as explained in §3.1.2 *Populations and samples*, is that our sample design for problems is the benchmark suite, so our observations of engine performance cannot be generalised to the whole problem population. There appears to be much more variation in the difficulty of problems than in the ability of engines to find solutions. This makes averaging over problems extremely suspect. If a biologist were given a sample of life forms consisting of a dried pea, a wasp and a blue whale, and asked to make observations of them, we would not expect the average of these observations to be very enlightening.

This predicament is colloquially referred to as *horses for courses* and generally treated as having no satisfactory solution. The colloquial expression refers to race horses and race courses. It means that how fast a horse runs depends on which course it runs on. If horse A beats horse B on 10 courses, but B beats A on a different 10 courses, you cannot say which horse is better. A horse's performance must be represented by a matrix with one element for each course. In our case, the horses are engines and the courses are problems.

Nonetheless, horses are bought and sold, and some horses are worth more than others. Money value is a scalar, so it appears that performance can be converted into a scalar by people who know about horses (and courses). This section presents an algorithm for the horses for courses problem. The algorithm is, to a certain extent, arbitrary. It might be possible to test it by using it to compute the price of real horses, and comparing the result with actual prices, but sadly that is beyond the scope of this investigation. The algorithm is presented here as an aid to visualisation, not as a tool for testing hypotheses.

An important property of this algorithm is that it applies equally well to engines and problems. It therefore gives us a means of deciding how good a sample a benchmark suite of problems is.

3.3.2 Cost and benefit

We begin by defining *cost* and *benefit*. We follow the usual practice of measuring cost by counting fitness evaluations. We introduce the term *benefit* as a measure of how effective an engine is at finding solutions. The novel idea here is that we also measure benefit in evaluations. Cost and benefit are thus both measured in the same units, which goes some way to making comparisons, averages, etc meaningful.

3.3.2.1 Measures of cost

The cost of using a GP engine is computational effort, so the measure of cost centers on the amount of computing work done.

There are various ways to measure computational effort, but counting fitness evaluations is the most often used. Other measures such as wall clock time, CPU time, or system accounting costs could be used but they generally add to the complexity and parochialism of the cost measure without improving its quality.

Fitness evaluation is generally computationally expensive and the number of evaluations is a measure of the efficiency of the GP engine. It is simply the number of individuals evaluated, up to but not including the solution. (The reason for not counting the solution is for consistency with the ideal GP engine that produces the solution first time. That ideal would have zero cost.)

An argument can be made for not counting individuals which duplicate earlier ones, since these have already been evaluated. The business of identifying duplicates — memoisation — has its own cost. If the cost of memoisation is comparable to the cost of fitness evaluation then it makes little sense to memoise, and the measure then becomes a simple count of evaluations as described above. If memoisation is worth doing then a compromise measure could be used, with a memoised evaluation counting as some fixed proportion of an evaluation. It is not difficult to imagine more complicated measures, but it is difficult to justify them. The simplest way to treat memoisation is to give it zero cost. In this thesis, memoisation is not considered.

The measure of cost used in this thesis is therefore *fitness evaluation count.*

3.3.2.2 Measures of benefit

The purpose of a GP engine is to find solutions to problems, so the measure of benefit centers on the quality of solutions found.

A *solution* is defined in this thesis as the first fittest individual found in a run. This individual may be discovered in any generation (including 0) and remains the solution unless and until a strictly fitter individual is found. There are many possible measures of the quality of a solution, but the fitness of the solution is the most obvious choice.

In order to measure cost and benefit in the same units, we calibrate the objective function in the same units as the cost function, namely, in evaluations. For example, we might say that a good solution is worth 100M evaluations, and a poor one is worth only 1k evaluations. This may appear to be an unnatural thing to do, in the sense that it feels strange. But it is only making explicit in a different form what is implicit when the user decides to stop searching before a perfect solution is found.

Existing objective functions are not always trivially simple to calibrate in evaluations. Some objective functions measure fitness as a number greater than or equal to zero, with a higher value representing a better solution. Some have a natural upper limit; some do not. Some measure error rather than fitness, so that the GP system is required to minimise the objective function rather than maximise it. None of these differences is of great significance but they contribute to mild confusion and unnecessary complexity.

To avoid adding to the confusion, we introduce the term *benefit* to mean the worth of a solution, measured in evaluations. The lowest benefit is zero, and the higher the benefit the better. Some problems may have a natural limit on the benefit, and others not.

3.3.2.3 Horses for courses

The horses for courses problem is as follows. We have e engines, $E_1 \dots E_e$, and p problems, $P_1 \dots P_p$. Each engine is tried with each problem, giving ep pairs. Each pair results in a benefit, which is the performance measure for the engine, and a cost, which is the performance measure for the problem.

The data can be represented as a benefit matrix *B* with *e* rows and *p* columns, and a cost matrix *C*, with *p* rows and *e* columns. The cost and benefit matrices are arranged so that the rows correspond to the entities whose performance is in the body of the matrix.

The data is sample means. In the simple form of horses for courses the means are treated as points instead of estimates, which is an over-simplification. A more robust method would take the standard error of the mean into account when producing the cladogram and again when the performance of engines and problems is reduced to a rational scale (§3.3.2.3.5 *Path length tree*, and §3.3.2.3.6 *Constructing the path length tree*, below). The final result should be that each engine or problem is represented by an interval on the rational scale, not by a point.

Suppose we have $e = 5$ engines and $p = 3$ problems. Typical data might look like this:

$$B = \begin{vmatrix} 27 & 39 & 21 \\ 22 & 23 & 32 \\ 17 & 39 & 29 \\ 24 & 26 & 25 \\ 17 & 37 & 26 \end{vmatrix}$$

$$C = \begin{vmatrix} 11 & 14 & 13 & 13 & 14 \\ 14 & 16 & 20 & 11 & 15 \\ 14 & 13 & 18 & 15 & 16 \end{vmatrix}$$

The solution to the horses for courses problem is a column vector *B'* of *e* rows, ie one element per engine, which is that engine's overall performance measure in the context of the problem sample. By symmetry there should also be a column vector *C'* of *p* rows from *C*, with one element per problem, representing that problem's difficulty in the context of the engine sample.

3.3.2.3.1 Arithmetic mean

There are countless functions which could produce a column vector as required, but there is no reason (yet) to look further than the arithmetic mean, and there are good reasons not to look much further.

In the arithmetic mean scheme, each element of *B'* is the mean of the corresponding row of *B*, giving the following values (to one decimal place):

$$B' = \begin{vmatrix} 29.0 \\ 25.7 \\ 28.3 \\ 25.0 \\ 26.7 \end{vmatrix}$$

And corresponding to C:

$$C' = \begin{vmatrix} 13.0 \\ 15.2 \\ 15.2 \end{vmatrix}$$

According to this scheme, engine 1 is the best with 29.0, engine 4 the worst with 25.0. Problem 1 is the easiest with 13.0, and problems two and 3 equally hard with 15.2.

The arithmetic mean is simple, but raises two concerns.

3.3.2.3.2 Two kinds of difficulty

To compute the mean we add the observed benefits of one engine working on several different problems. The first concern is that this may not be a meaningful thing to do.

Cost and benefit are measured in the same units, namely, evaluations. A consequence of this is that benefits and costs are commensurate across all fitness functions.

This is not a play on words. A problem has two kinds of difficulty or computational effort. The first kind is the difficulty of evaluating points in the search space. Evaluating a point may involve any amount of computational effort. The end result of this effort is one fitness value. The second kind is the difficulty of using these fitness values to direct a search, through the space defined by the fitness function and the genetic operators, to find a better point. The first kind of difficulty may be called *internal*, the second *external.*

Benefit and cost are measured in evaluations. They are not concerned with how much effort each evaluation involves, only in how well the search is conducted. They only measure external difficulty. Because they ignore the internal difficulty of evaluating each point's fitness, they measure only the difficulty that is visible to the GP engines. Cost and benefit are therefore commensurate, and the arithmetic mean is a valid computation. Indeed any linear sum of benefits and costs would be meaningful in these terms.

3.3.2.3.3 Equal importance

The second concern is that computing the arithmetic mean of a row of *B* implies that the columns of *B* are all of equal importance. This would be fair if the problems that the columns represent were a random sample of all problems. This, however, is not the case. First, the problems are a benchmark suite, chosen by educated guesswork. This is also true of the engines. Second, in any investigation, the columns are liable to contain not only the problems of the benchmark suite, but also variations of these, which are included to see what the effect of certain minor changes in problem parameters might be. This is more common when comparing engines than problems, since we are used to tweaking engine parameters, but it can arise in either case.

We have no real assurance that the benchmark suite is fair to begin with. The inclusion of minor variations pretty well ensures that the resulting mix is far from fair. We therefore upgrade from the arithmetic mean to at least a weighted arithmetic mean. Since cost and benefit are commensurate, the weighted arithmetic mean is as meaningful as the plain arithmetic mean. The issue is how to determine the weights.

The case of exact duplicates provides the basis of a thought-experiment which, although somewhat artificial, is quite informative. Suppose we have 5 problems A, B, C, D and E which are a fair sample (whatever that means). For simplicity, let us express the weights as percentages. Then each problem will deserve the same weight, say 20%. Suppose we now add to the set a sixth problem E' which is identical to E. The presence of E' adds no new information and therefore should not cause any change in the weighted means. The weights of A, B, C and D in the new set must still be equal, and also equal to the sum of the weights for E and E'. By symmetry, the weights of E and E' should be equal to each other. This implies that the weights of A, B, C and D remain at 20% while E and E' are at 10%. The weight which was assigned to E is now shared equally between E and E'.

Consider now a slightly different case. Suppose we have the same 5 problems as before, but instead of adding an exact duplicate, we add to the set a sixth problem E" which is very similar to but distinct from E. In this case we should expect that the weights of A, B C and D must still be equal but the weights of E and E" together should be slightly more than that. Whether the weights of E and E" should be equal depends on the differences of E and E" from A, B, C and D. We might find, for example, that A, B, C and D each have weight 19%, while E and E" have weights of 12% each.

To take this approach any further we need first a measure of the difference between two problems (or engines) and second, an algorithm to input this measure and output weights. With these weights we can compute the column matrices B' and C'.

3.3.2.3.4 Distance between problems

For the sake of simple language, we deal only with assigning weights to engines, so that we can compute the weighted mean of each problem's cost; remembering always that the same method must be able to assign weights to problems so that we can compute the weighted mean of each engine's benefit. For the purpose of explaining the method, it is better to have 5 weights to compute than 3.

There are many ways in which we could define the difference between engines, but arguably the simplest is to use the benefit matrix *B*. This matrix gives us the problems' view of the engines. If we want to consider the engines as black boxes, then the only external property we have is how well they solve problems, and that is exactly the information that *B* contains. Each row of *B* identifies a point in a *p*-dimensional space. We can therefore use Euclidean distance to measure the difference between any two engines.

For example, with the matrix *B* already given, we can calculate the engine distance matrix D_E as follows. Engine 1 is at (27, 39, 21) and engine two is at (22, 23, 32). The distance between these points is $\sqrt{[(27-22)^2 + (39-23)^2 + (21-32)^2]} = \sqrt{402} = 20.05$ approximately. Computing the distances between all pairs (to two decimal places) gives the following *e* by *e* matrix:

$$D_E = \begin{vmatrix} 0 & & & & \\ 20.05 & 0 & & & \\ 12.81 & 17.03 & 0 & & \\ 13.93 & 7.87 & 15.30 & 0 & \\ 11.36 & 16.03 & 3.61 & 13.08 & 0 \end{vmatrix}$$

The main diagonal is zero, showing that every engine is identical to itself, and the upper triangle (not shown) is a reflection of the lower triangle. This matrix shows that engines 3 and 5, with difference 3.61, are very similar, in terms of how they handle the 3 problems we are considering. Engines two and 4, with difference 7.87, are also quite similar, while engines 1 and 2, with difference 20.05, are the most distinct. A similar matrix, of *p* rows by *p* columns, can be computed for problems.

3.3.2.3.5 Path length tree

Now that we have a distance measure between engines, we need an algorithm to compute the weights to assign to them. What should the results of the algorithm be? If all 5 engines were equidistant, they should all receive equal weight, say 20%. But engines 3 and 5 are relatively close, so they should count more as one engine than two. This should raise the average weight per engine to about 25%, and engines 3 and 5 should get about 12.5% each. Engines two and 4 must be taken into account, and so on. We might expect engine 1 to get the most weight on account of its distance from engine 2.

To compute the weights we first represent the distance matrix as a PLT or cladogram. (The method of producing the PLT is given later.) A PLT is a tree composed of nodes and arcs. A node can have 1, two or 3 arcs. A node with 1 arc is a leaf node. The leaf nodes correspond to the points (engines in this case) we are measuring distance between. Exactly one node has two arcs: the root node. A node with three arcs is an internal or branch node. Each arc has a length. The PLT represents the distance matrix in the sense that the sum of the arc lengths along the path from one leaf node to another is equal to the distance between the corresponding points in the distance matrix.

The PLT for the distance matrix given above is as follows:

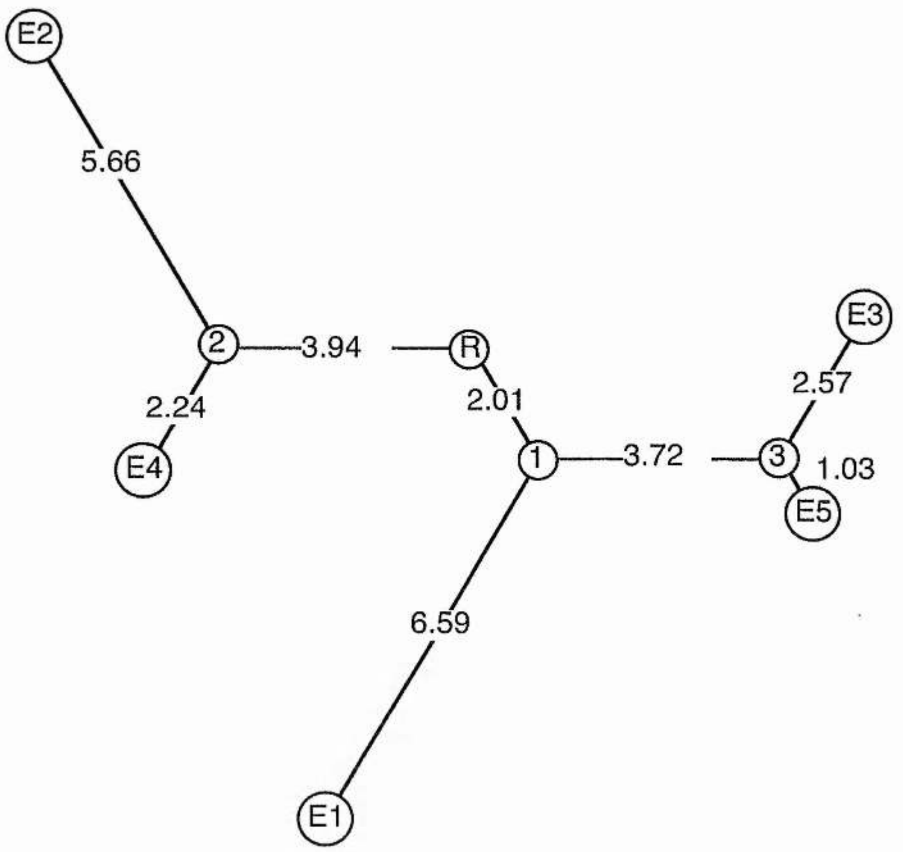

Figure 3-4: Path length tree

Node (R) is the root. Nodes (1), (2), (3) are internal nodes. Nodes (E1), (E2) etc correspond to the engines 1 ... 5. The arcs are labelled with distances. For example, the distance from (E2) to (E4) is 5.66 + 2.24 = 7.9, as compared to 7.87 in the distance matrix. In practice the PLT does not represent the distance matrix exactly, because a PLT has fewer degrees of freedom than a distance matrix for the same number of points. The construction algorithm aims to minimise the error.

The PLT in figure 3-4 is easy to comprehend, but the drawing style is not easy to use with larger numbers of nodes. A more conventional representation is as follows:

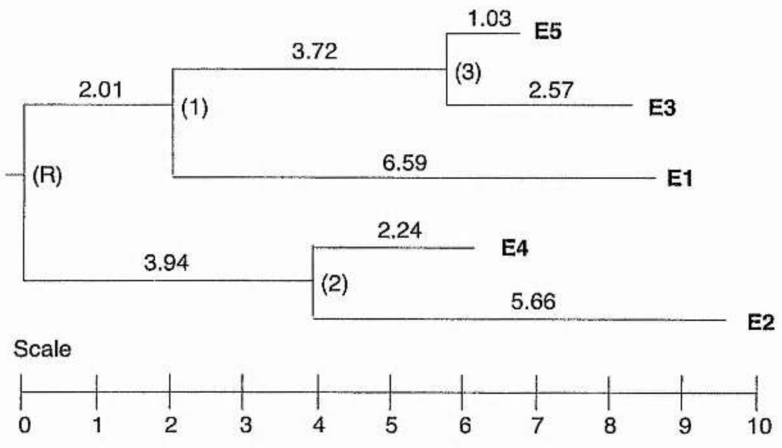

Figure 3-5: Cladogram

The diagram is a 2-dimensional representation of a set of points in an *n*-dimensional space. The points in the *n*-dimensional space are the engines; the 3-dimensional space is defined by the benefit (in evaluations) measured by the three problems.

The horizontal dimension of the diagram uses a rational scale to represent distance. The horizontal scale is shown at the foot of the diagram. The length of each horizontal arc is given above the arc.

The vertical dimension is categorical, and simply places each point in its own category. Vertical lines do not represent distance. The vertical categories are sorted by horizontal length, with shortest length first. Thus, **E4** lies above **E2**, and the entire **E5-E3-E1** subtree lies above the entire **E4-E2** subtree.

The numbers in parentheses are the internal node numbers. They are not significant but have been left in to identify the nodes.

Both forms of tree are equivalent. The cladogram form is easier to scale and to produce automatically. The PLT form is perhaps easier to read, so I use it for the rest of this exposition.

Leaf nodes which are close (as defined by the distance matrix) are also close in the PLT and cladogram. Consider the paths from the root to the leaf nodes. Leaf nodes for engines which are close have paths with arcs in common. The closer the leaf nodes, the more common arcs. Identical engines would appear as nodes separated by an arc of length zero, so that the entire path length from the root would be common.

Representing the distance matrix as a PLT enables us to assign weights to engines in a systematic way. If the engines were equally diverse, they should be equally weighted. In this case the PLT would be star-shaped, and all arc lengths would be equal. The PLT is not star-shaped to the extent that the engines are not equally diverse. Firstly, the root acts as a kind of centroid, so the distance of a leaf from the root is a reasonable measure of the weight to attach to the leaf. If the PLT were star-shaped with a different arc length to each engine, the weight attached to each engine would be its arc length. But the PLT is tree-shaped, which is to say, some leaf nodes share part of the path to the root. For example, (E2) and (E4) share the arc (2)–(R). This is a measure of the similarity between (E2) and (E4). The more similar such nodes are, the more path they share. We take account of this by dividing the weight of the shared path between all the leaf nodes that share it. In this way, leaf nodes which are similar are made to share weight. Applying this to the PLT given above results in the following:

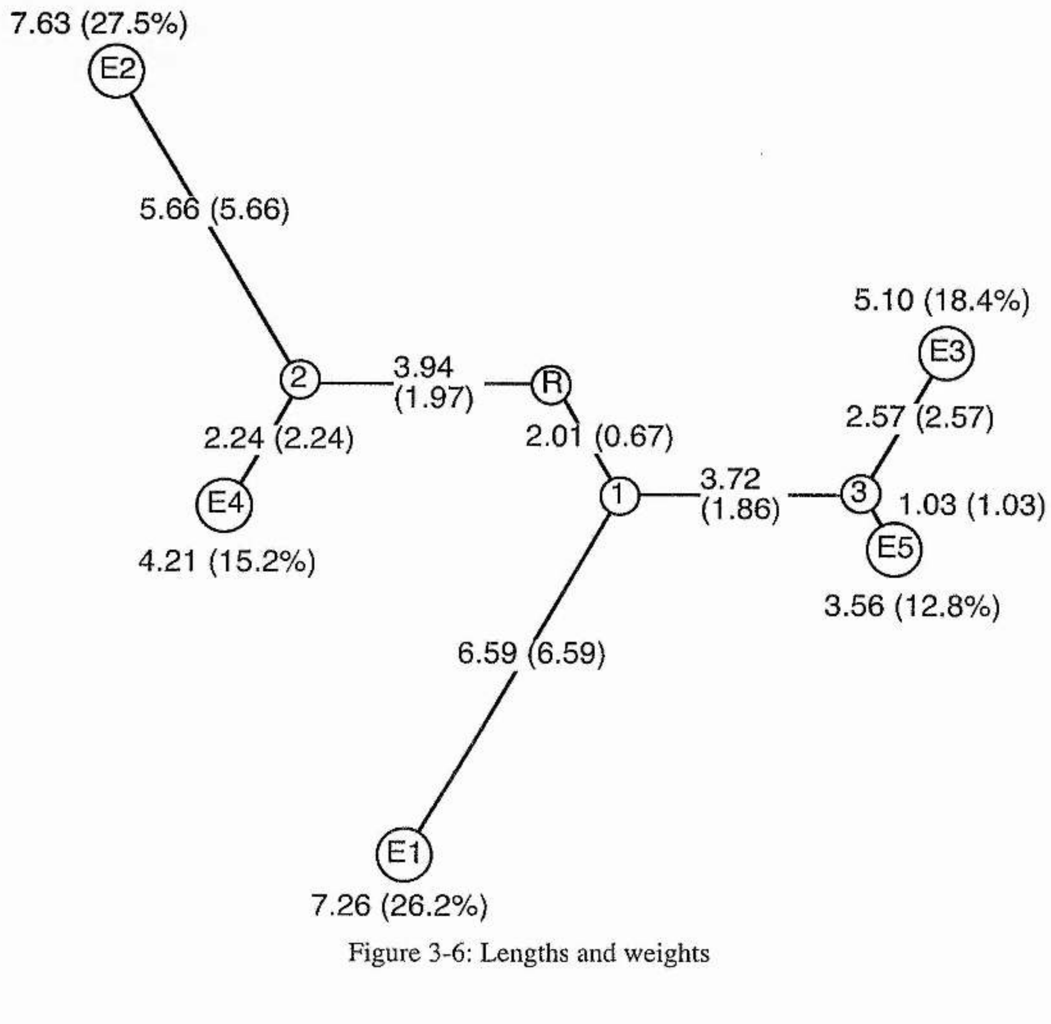

Figure 3-6: Lengths and weights

The weight of each arc is shown in parentheses after its length. For example, the arc (R)–(2) is shared by (E2) and (E4) so its length has been divided by 2, giving 1.97. Similarly arc (R)–(1) was divided by 3, and arc (1)–(3) by 2.

The weight of node (E2) is now computed as 5.66 + 1.97 = 7.63. This is shown next to node (E2). The figure 27.5% is the weight as a percentage of the weights of all the leaf nodes. (The percentages do not add to exactly 100% because of rounding error.)

This gives us engine weights W_E as follows:

$$W_E = \begin{vmatrix} 7.26 \\ 7.63 \\ 5.10 \\ 4.21 \\ 3.56 \end{vmatrix}$$

or, as percentages:

$$W_E = \begin{vmatrix} 26.2 \\ 27.5 \\ 18.4 \\ 15.2 \\ 12.8 \end{vmatrix}$$

from which we can compute the weighted mean of the problem costs as:

$$C' = \begin{vmatrix} 12.9 \\ 15.3 \\ 14.9 \end{vmatrix}$$

For comparison, the unweighted mean (from §3.3.2.3.1 *Arithmetic mean*) is:

$$C' = \begin{vmatrix} 13.0 \\ 15.2 \\ 15.2 \end{vmatrix}$$

The effect of the weights is slight but noticeable.

3.3.2.3.6 Constructing the path length tree

PLTs are used in biology to represent distance relationships between living creatures in a process called phylogenetic inference.

The Fitch-Margoliash algorithm [Fitch, 1967] for producing unrooted PLTs is available as part of the PHYLIP software package [Felsenstein, 1995]. It is a heuristic algorithm which searches for a tree that minimises the error expression:

$$\sum_i \sum_j \frac{\left(D_{ij} - d_{ij}\right)^2}{D_{ij}^2}$$

where *D* is the distance accoding to the distance matrix, and *d* is the distance according to the PLT. The search builds a tree by finding the two entities (engines or problems, in our case) which can best be combined into a single entity which averages their distances. This reduces the number of entities by one. The process is repeated until no entities remain. The phylogenetic tree that results is simply a record of the order in which entities were combined. The first tree so created is not necessarily the best. The algorithm creates a series of trees by varying the order of combination slightly, and retaining the tree with the least error.

The unrooted PLT for our example data looks like this:

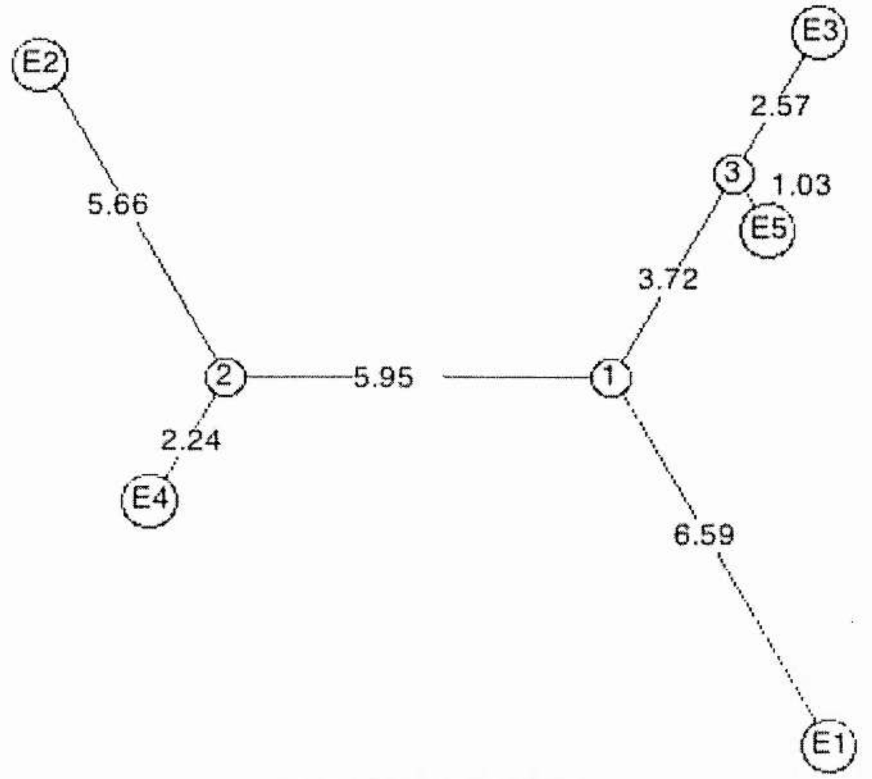

Figure 3-7: Unrooted PLT

An unrooted tree is unusual in Computer Science, and it is easy to make the mistake of thinking of one node as 'obviously' being the root, especially if the representation of the tree artificially distinguishes one node from the others. One way of establishing a root is to choose an existing node. However, it is not always obvious which node to choose, and experiments with pencil and paper soon show that fair results sometimes cannot be achieved with any existing node as root. We need a reliable method of establishing a root before we can use the PLT to compute weights.

The following method is used find a root node. We create a new, virtual point which is extremely and equally distant from all the real points. This requires a new row and column in the distance matrix, with all elements (except the main diagonal element) set to the same high value. The modified distance matrix is shown below. (Note that the top half reflects of the lower half.)

$$\begin{vmatrix}
0 & & & & & \\
20.05 & 0 & & & & \\
12.81 & 17.03 & 0 & & & \\
13.93 & 7.87 & 15.30 & 0 & & \\
11.36 & 16.03 & 3.61 & 13.08 & 0 & \\
1000.00 & 1000.00 & 1000.00 & 1000.00 & 1000.00 & 0
\end{vmatrix}$$

This virtual point is like a fixed star, so far away that all the real points are equidistant from it. The Fitch-Margoliash algorithm is not sensitive to the exact distance to the fixed star, provided it is large enough. In the example data given above, a distance to the fixed star of 1 000 is sufficient; further increases result in very small changes to the resulting PLT. The PLT produced by Fitch-Margoliash for a tree with a fixed star at distance 1 000 is shown below.

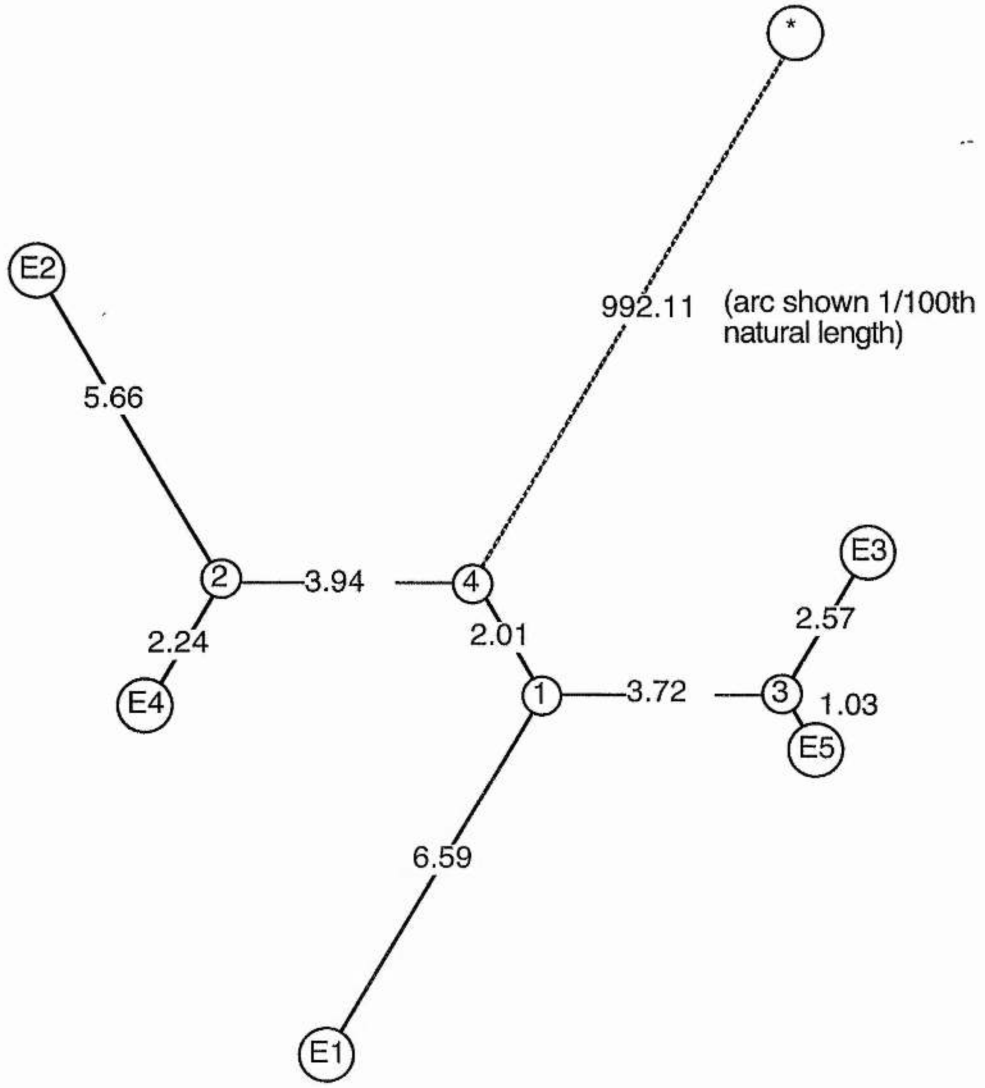

Figure 3-8: PLT with fixed star at distance 1 000

Compare the PLT without a fixed star (figure 3-7) to the PLT with a fixed star (figure 3-8). Both are unrooted trees. The difference is that the PLT with fixed star has one more leaf node (*) for the fixed star and one more internal node (4). Apart from the new arcs (2)-(4) and (4)-(1), the previous arcs are unaffected by the change. Further, the old arc (2)-(1) has simply been split in two by node (4), since 3.94 + 2.01 = 5.95. While this is not a proof of anything, it is typical of the effect of adding a fixed star. Node (4), which is where the fixed star attaches to the PLT, is chosen as the root. The arc (4)-(*) is discarded, leaving us with a rooted PLT.

3.3.2.3.7 Performance-based weights

We have described a procedure for computing weights based on engine differences. Are other methods possible? One approach that seems attractive is to have weights proportional to performance. The idea here is that the problems are not equally difficult, so we should give more weight to difficult problems than easy ones, when averaging an engine's performance.

Unfortunately, this approach does not work. If the weight for a problem is proportional to the problem's cost, then by symmetry, the weight for an engine is proportional to the engine's benefit. This leads to simultaneous equations which reduce to $0 = 0$.

The conclusion is that we cannot treat benefit and cost as multiplicative inverses of one another in this way, and that to try to do so is a conceptual error.

4 Grammars

This section deals with issues of scalability which are raised in §2 *Gads 1*. The essence of the scalability problem is that the Gads 1 experiment used an unrealistically small CFG. To generate a wider range of programs, the technique must be extended to use a realistically large CSG.

CSGs are necessary because real programming languages such as C and Java have context-sensitive rules. For example, it is only valid to refer to an identifier in an expression if the identifier has been declared, is of the correct type, includes the expression in its scope, and (depending on the language) has been initialised. Only if these conditions have been met may the identifer be used. But it is beyond the power of a CFG to represent these conditions or to test whether they have been met before allowing the relevant productions to be invoked. Where CFGs have productions whose LHS is a single nonterminal, CSGs have productions of the form:

```
xAy   ::=   xBy
```

that is, `A` may be rewritten as `B` if it is in the context defined by `x` and `y`. That is the formal approach, but in practice it is more usual to use other representations for the grammar, as it would not be practicable to produce a definition for a real language in that form. The most common and practical forms of CSG are the two-level grammar and the attribute grammar [Pagan, 1981], [Deransart, 1988].

Whichever type of grammar is used, the problem remains that grammars are designed for analysing sentences, not for synthesizing them. Chapter 1 of [Pagan, 1981] lists seven uses of a formal grammar. Synthesis is not among them. The historical reasons for this are obvious. Until the invention of GP there has been next to no automatic programming and therefore no possibility of synthesizing programs.

4.1 Existing CSGs

This section considers existing forms of CSG for use in Gads. The key feature that Gads requires is that the grammar can be used to synthesize sentences without computationally expensive backtracking or other searching. As shown in §2 *Gads 1*, CFGs have these properties, but as explained above, they are not suitable for deriving more realistic programs.

4.1.1 Two-level grammars

We examine two-level grammars only in enough depth to decide they are not suitable. Suppose we have a two-level grammar defined as follows:

```
Protonotions
     None.

Notions (rules)
     program:                    series.
     series:                     statement sequence.

Hyper rules
     SEQITEM sequence:           SEQITEM;
                                      SEQITEM sequence,
                                      SEQITEM.

Meta notions (meta rules)
     SEQITEM::                   declaration;
                                      statement;
                                      letter.
```

Now consider using this grammar to generate a sentence beginning with the start symbol `program` and applying productions:

```
        program
=>      series
=>      statement sequence
=>      ?
```

At this point we must pause. What does *`statement sequence`* change to? It is a notion, but there is no rule for it, because it is an instance of the more general *`SEQITEM sequence`* hyper rule.

How do we find a hyper rule for *`statement sequence`*? In this tiny grammar, we can see that *`statement sequence`* matches *`SEQITEM sequence`*, because *`statement`* matches *`SEQITEM`*. But in a realistically large grammar, this matching is not likely to be simple. Even after a successful matching, and identifying a hyper rule, we are not finished. In general, a hyper rule may define any number of *`statement sequences`*. We will need a further method to choose one in particular.

To summarise, synthesis in two-level grammars is much more complicated than in one-level grammars. It's not immediately obvious how to do it, or even if it's possible. We therefore put two-level grammars to one side.

4.1.2 Attribute grammars

Standard attribute grammars (sags) are not suitable for Gads 2, for reasons that come down to the bias in favour of analysis over synthesis. This bias leads to grammars that can recognise a valid sentence, but cannot be efficiently directed to produce one. Sags are a powerful computational paradigm. To say that they are not suitable for Gads 2 therefore requires careful justification. That justification now follows. Although it is closely argued, it is not so much a formal mathematical proof as a methodical exploration of system design.

Suppose we are using a sag to generate a PT in some language, beginning with the start symbol, and expanding node after node. Suppose we are using the basic Gads 1 mechanism for this: that is, we have a stream of genes and use them to choose one production at a time from the set of applicable productions (ie the set of productions whose LHS labels the PT node we are expanding). Suppose we are now about to expand a node which should generate an identifier, and suppose that out of all possible identifiers in this language, only a finite subset are valid in this context.

For example, we may have already generated:

```
let s = "a string"
let x = 3
let y = x * 5
let z = x * x +
```

The language requires an expression after the last + sign; we have made choices as to the type of expression, and have decided that the expression is to be an identifier. Possible identifiers include `x`, `y`, and perhaps some predeclared identifiers such as `pi`. But an undeclared identifier in this context is not valid. Nor is an identifier if its type is not compatible. An identifier of type `string`, for example, would not be valid here.

We don't need to be specific about the productions in the grammar or the specific language. At some point in the derivation of identifiers we must reach a stage where some choices are valid and others are not; and we suppose we are at that stage.

The usual way to design a sag that deals with this is to use the attributes to maintain an *environment*: that is, a data structure which records a description of the context at each node in the PT. For example, at the PT node corresponding to the declaration of identifier `s`, the environment would typically be augmented by a representation of the fact that `s` is in scope, that it has type `string`, and that it has been initialised. Similarly the environment is augmented by the declarations of `x` and `y` by the time we reach the situation in question. The specifics of the environment do not affect this argument: what is important is simply that there is an environment recording the context.

In analysis (ie when parsing a string of symbols which may be a sentence in the language), an environment is used to test whether a given identifier is valid by looking up the identifier in the environment and confirming that certain conditions hold. Typical conditions in this case would be such as (1) the identifier must exist in the environment, (2) it must be of a scalar arithmetic type, and (3) it must have been initialised. The conditions are usually formalised as a predicate which is associated with the production that produces the identifier. On encountering a predicate that evaluates to *false*, the analyser takes some exceptional action, such as backtracking to try other parsing possibilities, or, if this is not possible, issuing a message indicating that it has discovered an error in the input.

In synthesis (ie when producing a valid sentence in the language), we would like to use the environment to direct the choice of productions, so that a valid identifier could be produced. There is nothing in the productions themselves which can assist in this. Each production consists of an LHS which is a nonterminal symbol, and an RHS which is a sequence of one or more symbols. These symbols do not refer to attribute values. Also, since the set of productions in a sag is fixed, the productions must be capable of producing all possible identifiers in the language at any node in the PT.

The following subsections explore ways in which the synthesis might proceed, that is, the ways in which a sag could be used for Gads 2.

4.1.2.1 Calculation after selection

The simplest approach is to use the gene stream to select productions in the usual way. After each production is selected and applied, we evaluate the attributes and predicates associated with it. So long as no predicate evaluates to *false*, this process continues. It terminates when the PT is complete, in which case it will represent a type-correct program.

Difficulties arise if a predicate evaluates to *false*. If this happens, it means that a production was chosen in a context where it was not a valid choice. The difficulty is to know what to do about it.

The simplest option is to give up: mark the individual as unviable and perhaps give it the lowest possible fitness value. But this would lead to an unacceptably high mortality rate, as the chances of producing a type-correct individual by this method are fairly low.

Another option is to *backtrack*, that is, to revisit an earlier decision and choose a different path forwards. (We could save genes by skipping over the gene that made a bad decision, and re-use the gene stream, but displaced by one position.) This time we may succeed; if not, we backtrack and try again. Provided we methodically backtrack to recent decision points first and try them, before backtracking to earlier decisions, backtracking carries out an exhaustive search of a tree of possibilities. However, because we are using the gene stream to direct the search, it is not exhaustive. Our searching is limited by the genes that are available in the current individual. We are more likely to exhaust our supply of genes than exhaust the possibilities in the search space. Also, even if we could exhaust the search space, backtracking could be enormously expensive. For example, suppose under the direction of the gene stream, we decided to generate an identifier in a context where *no* suitable identifier was available. The fact that this particular alley is blind does not show up until we reach the end of the alley: we generate an identifier and a predicate evaluates to *false*. At this point we try another identifer — and it too fails. By the principle of backtracking to most recent decisions first, we must generate and reject all possible identifiers before we revisit the decision that is the cause of the rejections. Given that the productions can produce all possible identifiers in the language at any node in the PT, this is likely to take an unacceptably long time.

The conclusion is that any approach in which we choose a production before testing the consequences of that choice, is not feasible.

4.1.2.2 Calculation before selection

In this approach we investigate the possibility of using current environment as well as the gene stream to direct the choice of the next production.

It is clear from the analysis above that what is required is a method by which we can restrict the choice of productions at any given node in the PT. We need to exclude from consideration those productions which must lead to failure. This avoids blind alleys. We don't need to exclude productions that lead to a mixture of successes and failures, provided that they have at least one successful outcome.

For example, suppose we have a grammar that generates identifiers as letter sequences, and that we are in a context where the only valid identifiers are `apple` and `banana`. In this context we should exclude productions that lead to identifiers beginning with anything other than `a` or `b`, as they *must* fail. We should consider only productions that lead to `a` or `b`, even though they could lead to identifiers such as `avocado` or `bramble` which would fail. Paths leading to certain failure are excluded later.

The environment already has a representation of the set of identifiers which are valid at this point in the PT. What is required is a function which takes as its parameters a grammar and an environment, and from these returns the set of production rules (more precisely, their identifying numbers) which are valid at this point. More formally, we seek a function *f* such that:

$$f:(G,e)\mapsto\left\{p \mid L(p)\cap I\neq\Phi\right\}$$

where:

G	is the attribute grammar
e	is the environment at this stage in the PT
p	is a production in G (more precisely, its identifying number)
$L(p)$	is the language of p, ie the set of identifiers that can be derived from p
I	is the set of valid identifiers at this point in the PT, extracted from e
Φ	is the empty set.

In English, *f* identifies the productions which *may* lead to valid identifiers. The only productions which are excluded are those which *could not possibly* lead to a valid identifier. Having used *f* to compute the set of valid productions in the context of the current node, we use the next gene from the gene stream to choose one of these productions. From the definition of *f* we know that no matter which production we choose, we will not necessarily end in failure. Every production returned by *f* can lead to success. By computing *f* at each PT node, and only choosing from the set of productions it returns, we can generate a program which is type-correct, without ever needing to backtrack.

The essence of this approach is that *f* is computed *before* we choose a production, which guarantees that we never need to backtrack. However, this method has drawbacks.

First, *f* may be unacceptably expensive to compute. To compute *f* we need to test each production p in the grammar which has the LHS for the current PT node. For each p, we need to decide whether there is at least one identifier i in the environment, such that i can be derived from p. If so, we add p to the set of possible productions to be returned by *f*. If not, p cannot lead to a valid identifier and we exclude it excluded. The number of identifiers we examine to decide whether to allow or exclude p is not fixed. We only need to derive one identifier to allow p, but we can only exclude p after trying every valid identifier in the environment. Thus, to test a given p, *f* analyses i using a grammar G' which is equal to G except that it has p as its start symbol. To compute *f* we carry out many such analyses.

Second, unless *f* is to be used at every node of the PT, a mechanism is needed to indicate when to use *f*, and when to go ahead without it. This could probably be done without deviating from the sag model by listing all affected productions, or giving certain nonterminals names in a certain form, for example.

Third, for Gads 2 to use *f*, the grammar must be written to implement an environment in a particular way, and the environment must be accessible by Gads 2. Gads 2 would have to be closely coupled with the grammar, which is not a good design principle. Grammar design is not trivial, so this may not always be an easy thing to accomplish.

Fourth, this approach requires that every time an identifier or piece of context-sensitive code is required, it has to be generated by a directed grammatical derivation.

The conclusion is that any approach in which we test productions to see if they could lead to a valid PT would be expensive to compute, difficult to program, and inefficient.

4.1.2.3 The third way

A weakness of both the previous approaches is that they use productions to replicate the information available in the environment. This leads to inefficiency. It is simple to produce a set of valid identifiers for the current context from the environment. Instead of using the gene stream to direct a grammatical derivation that leads to one of them, it would be much simpler to use just one gene to choose one directly from the set. An identifier chosen in this way is valid, so backtracking is not necessary.

The advantage of this approach is that it is simple and efficient. In fact, given that it consumes just one gene per identifier, has no backtracking, and it is computationally inexpensive, it is hard to see how it could be improved upon.

The disadvantage of the third way is that we no longer have a standard attribute grammar. The third way requires that at a certain stage in the growth of the PT, we stop growing the PT by choosing from the set of applicable productions and expanding a PT node, and instead choose from a set of identifiers computed from an attribute and use that to expand the node. It may be efficient, but it is not a standard attribute grammar. Also, this approach requires close coupling of Gads 2 and grammar.

The next section presents a method of obtaining the advantages of this approach with none of the disadvantages.

4.2 Reflective attribute grammars

The example of selecting an identifier is obviously just one example of context sensitivity. For full context sensitivity, we must be able to choose types, procedures, control structures, and all the panoply of a full context sensitive language. The ideal grammar for Gads 2 ontogenesis would have the set of all acceptable choices in any given context, and use one gene in the genome to choose one of them. I call this *ideal* because it uses one unit of genetic information to make one choice in the ontogenesis, and no backtracking is ever necessary.

An extended form of attribute grammar called a *rag*, which reaches the ideal, is introduced below. I have used S-algol [Morrison, 1979] below to illustrate the ideas. S-algol combines a pleasant grammar with a full range of imperative language features such as basic types, procedures, arrays, and structures. S-algol is also used as the phenotype language in the experiments which follow. The S-algol rag is given in §B.

4.2.1 Overview

This section develops the idea of the rag, placing it in the context of CFGs and sags. The explanation is given in terms of formal grammars and also in terms of a possible object-oriented implementation of them. The actual implementation was in Java, so as to be able to make use of ECJ [Luke, 2001].

4.2.1.1 Grammatical notation

Grammars are described using a form of BNF based on the description given in §2.4.1 of [Pagan, 1981]. Briefly, a *symbol* is a sequence of lowercase letters or digits beginning with a letter. Spaces are not significant. Thus the following all identify the same symbol:

```
no table
notable
not able
```

A symbol ending in "`symbol`" defines the concrete representation of a terminal symbol. Each terminal symbol must have exactly one rule that defines it. The rule names the terminal symbol on the left, terminated by a colon. The right hand side is a string of characters delimited by a nonspace character (normally the quote). Spaces are significant within the delimited string. For example:

```
caret symbol:               "^"
quote symbol:               /"/
newline symbol:             "
"
numeral 4 symbol:           "4"
while symbol:               " while"
```

The terminal symbols are usually collected in a *representation table* which is separate from the rest of the grammar. Separating the concrete syntax from the abstract helps keep the grammar organised and easier to read.

A symbol which does not end in "`symbol`" is a nonterminal. Each nonterminal must have at least one rule that defines it. The rule names the nonterminal symbol on the left, terminated by a colon. The right hand side is a series of one or more *alternatives*, separated by semicolons. The whole rule is terminated by a period. Each alternative consists of a comma-separated sequence of symbols. For example:

```
literal int:        space symbol, numeral 0 symbol;
                    space symbol, digits.


digits:             digit;
                    digit, digits.


digit:              numeral 0 symbol;
                    numeral 1 symbol;
                    numeral 2 symbol;
                    numeral 3 symbol;
                    numeral 4 symbol;
                    numeral 5 symbol;
                    numeral 6 symbol;
                    numeral 7 symbol;
                    numeral 8 symbol;
                    numeral 9 symbol.
```

As in Gads 1 it is convenient to place the alternatives in different productions. The definition of `literal int` above is equivalent to:

```
literal int:        space symbol, numeral 0 symbol.


literal int:        space symbol, digits.
```

The alternatives for a nonterminal are numbered sequentially from 0. The order of the alternatives in a production is not significant as far as the language (ie the set of sentences and their structure) which the grammar defines is concerned, but it does affect the ontogenesis, because the first production for a nonterminal (ie production 0) is used as the default production for that nonterminal.

We generate a sentence by growing a PT, starting with the root, and expanding node by node until each path ends with a terminal symbol. Gads 2 always expands the leftmost node first. We describe this process either as growing a PT by expanding nodes or as rewriting nonterminal symbols in a string, providing the meaning is clear from the surrounding text.

4.2.1.2 Context free grammars

When the grammar is a CFG, expanding a non-terminal node consists of using the next gene to choose a production alternative from the grammar which has that non-terminal symbol as its LHS. In Gads 1, the alternative chosen is the one whose number equals the gene value. In Gads 1 this was acceptable but for a large grammar it would lead to many genes having no effect, as only a few alternatives would be applicable at any time.

[O'Neill, 2001b] describes the GE solution to this, which is to translate the gene value into a production number by a simple arithmetic operation. In GE this is done by computing the gene value *modulo* the number of applicable production alternatives. In Gads, we scale the gene value from the gene range to the range of applicable alternatives. For example suppose genes are integers in the range [0, 999] and there are 4 alternatives for the leftmost nonterminal, numbered 0, 1, 2 and 3. Then any gene value in the range [0, 249] will select alternative 0, any gene value in the range [250, 499] will select alternative 1, and so on. Thus every gene is effective. The choice of translation scheme does have an effect, but this implementation is concerned with feasibility rather than performance.

Although the order in which alternatives are numbered does not affect the language of the grammar, this version of ontogenesis uses the first alternative (ie alternative 0) as the *default* definition for the symbol. The default is used if ontogenesis has exhausted the genes in the genome, and unexpanded nonterminal symbols remain in the developing phenotype. For example (using the above definitions), if an unexpanded `literal int` remains in the developing phenotype, it defaults to `<space> 0`. Similarly, `digits` defaults to `digit` and `digit` defaults to `0`. Thus, by ordering the alternatives with care, all unexpanded symbols can be expanded automatically to sensible defaults. However, if the order of the alternatives for `digits` were reversed, then an unexpanded `digits` symbol would default to an endless string of `0`s.

In object-oriented terms, nodes are objects of class Node. A Node has an array of child Nodes.

4.2.1.3 Standard attribute grammars

When the grammar is a sag, attribute values are inherited or synthesized at each node. (Note that Gads 2 does not use a sag — this step in the explanation is included just to fill a gap on the way to rags.) Inheritance and synthesis corresponds to passing parameters to, and receiving results from, the node-expanding procedure. *Attribute evaluation rules* are associated with each production alternative to specify the inheritance or synthesis is to be done, in the event that that alternative is used to expand a node. Generally, an expression assignment language is all that is needed for the evaluation rules. For example, if `val` is an attribute associated with `digits` and `digit`, we might have:

```
digits:          digit;
      {digits.val := digit.val}


digits:          digit, digits.
      {digits1.val := digits2.val*10+digit.val}
```

The subscripts in the second evaluation rule are necessary to distinguish between the two occurrences of `digits`. In object-oriented terms, these two occurrences are distinct objects, each of which has a `val` attribute.

The numerical value of a string of digits in a sentence of the language can thus be referred to in the grammar. This makes it possible, for example, to specify in the grammar that no string of digits was greater than some limit. Without attributes, it is not really feasible to do such a thing, so specification of limits is either not done at all, or is done in an ad-hoc manner. When attribute grammars are used for parsing, this can be achieved using a boolean attribute `OK` which is synthesized at each nonterminal, and which is true if and only if the input sentence is recognised. If a false value is ever synthesized, the non-terminal responsible may give an indication of the error that has been detected. For example, to ensure that `digits` never represents a value greater than the machine limit `MAXINT`, we could extend the definition of `digits` to read:

```
digits:           digit;
      {
            OK := digit.val <= MAXINT
            digits.val := digit.val
      }


digits:           digit, digits.
      {
            OK := digits₂.val <=
                        (MAXINT-digit.val)/10
            digits₁.val :=
                        digits₂.val*10+digit.val
      }
```

In this example, we must take care that no intermediate value exceeds `MAXINT`, during the calculation of `OK`. However, since ontogenesis synthesises sentences rather than analysing them, there is no need for an attribute such as `OK` in Gads 2.

4.2.1.4 Reflective attribute grammars

When the grammar is a rag, the grammar itself is treated as an attribute. It is passed to the node-expanding procedure. The default behaviour when expanding a nonterminal node is to invoke the node-forming procedure recursively to produce a child node for each symbol on the RHS of the production alternative, and pass it the grammar, unchanged, as a parameter. If the rag never deviates from this default behaviour it is equivalent to an ordinary attribute grammar.

During the expansion of the node, the grammar may be modified, and the modified grammar passed down to the children of this node. (A modified grammar could conceivably be returned — ie synthesized — though this was not necessary in any of the examples I studied.) The expansion of the node is directed by the evaluation rules associated with the production alternative.

Rag evaluation rules are more complex than sag evaluation rules. The default rag evaluation rules have the following effect.

- Inherit: the rag *r*, the symbol to expand *s*, and the genome *g*.

1 Record the symbol *s* for this node.

2 If *s* is a terminal symbol, there is nothing to do. Return.

3 Use the next gene in *g* to choose an alternative from the productions which have this symbol on the LHS. (The genome object *g* is modified each time a gene is consumed.) If the genes are exhausted, choose the default alternative.

4 Create an array of child nodes, with one array element for each symbol on the RHS of the chosen production. Let the symbols on the RHS be $s_0 \dots s_n$.

5 For $i = 0$ to n, expand child node i, passing in the values r, s_i and g.

6 Return.

These are the steps which are carried out each time a node is expanded in the ordinary way. The grammar g is not modified and the effect is equivalent to a sag.

The S-algol *for* clause provides a slightly more complex example. *For* is one of several alternatives for `clause void`, and is defined as follows:

```
clause void:        for symbol,
                    id int new,
                    space symbol,
                    equals symbol,
                    clause int,
                    to symbol,
                    clause int,
                    do symbol,
                    clause void.
```

For example:

```
for x = 1 to 3 do write "x: ", x, "'n"
```

which, when executed, produces the following output:

```
x: 1
x: 2
x: 3
```

The *for* node thus has 9 children, numbered from 0 to 8, corresponding to the symbols on the RHS of the production. According to the scope rules of S-algol, a *for* clause introduces a new identifier (x in the example above) whose scope is the `clause void` at the end of the *for* (`write` in the example above). It follows that only two of the 9 children raise any issues of context sensitivity, namely #1 (`id int new`) and #8 (`clause void`). All the other children can be expanded according to the default evaluation rules given above. They inherit the context of the *for* as a whole. For children #1 and #8, the rag has two goals: (i) that child #1 has to introduce a new identifier, and (ii) that the identifier introduced by child #1 has to be available for the expansion of child #8.

The following diagram illustrates the situation after child #1 has been expanded; the curved arrow shows the data flow of the attribute synthesized by child #1 and about to be inherited by child #8. The diagram is somewhat impressionistic: data does not flow into the *for* clause node, but into the process which is constructing that node. A fuller description of the events depicted in the diagram is given below.

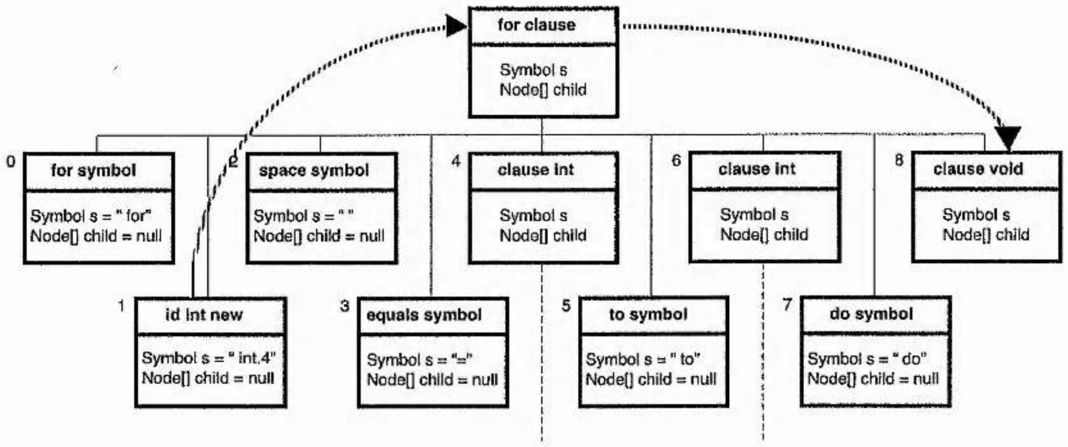

Figure 4-1: Expansion of a *for* node

The mechanism for introducing new identifiers is embodied in the evaluation rules for certain nonterminal symbols such as `id int new`, and is invoked whenever a node for such a symbol is constructed. The evaluation rules associated with `id int new` are identical to the default rules given above, for steps 1 through 4. Steps from 5 onwards are as follows:

5 Generate a new identifier as a character string. Identifiers must be unique. As an aid to readability they are constructed in the form *t.n* where *t* is the type of the identifier and *n* is a serial number. For example, the identifier of a *for* clause control variable might be `int.4`.

6 Declare a new terminal symbol whose value is the concrete representation of the new identifier. This makes the identifier available as a single symbol in the grammar.

7 Convert the current node from a nonterminal node to a terminal node, with the new identifier as its value. Set the array of children to null.

This raises the question: what production has the nonterminal symbol `id int new` as its LHS? This is addressed in more detail below; in passing note that the definition of this symbol is not relevant, since the effect of evaluation rule 7 is explicitly to ignore the RHS. In order to keep the grammar valid, however, every nonterminal symbol must be the LHS of at least one production, so we simply introduce a terminal symbol as a convenient stop:

```
id int new:              id int new symbol.


id int new symbol:       " idIntNew"
```

The evaluation rules associated with the *for* clause are identical to the default rules given above, for steps 1 through 4. Steps from 5 onwards are as follows:

5 For $i = 0$ to 7, expand child node i, passing in the values r, s_i and g. That is, expand all children in the default way except the final `clause void`.

6 Obtain the identifier synthesized by child #2. The evaluation rules for new identifiers ensure that this is in the form of a terminal symbol which we may name t. (That is, t is the name of a name. The scope of t is the evaluation rules of the *for* clause.)

7 Construct a new production of the form:

```
id int c:   t.
```

The LHS of this production (`id int c`) is the symbol for identifiers of constant integer variables (ie variables whose values may be read, but which may not be assigned to).

8 Push the new production on to the grammar r, producing grammar r'. This adds t to the set of integer identifiers that are available at this point in the PT. There may or may not be other identifiers in scope at this point. The term *push* is used to suggest a stack-like operation in r. From a theoretical point of view, r' may be a completely new and separate object from r, but in practice, it is much cheaper to modify r.

9 Expand child node #8, passing in the values r', s_8 and g. That is, expand the final `clause void` with a grammar that is extended by the addition of a production that allows the *for* clause's control variable to be used in any context that requires an `id int c`.

10 Pop the new production from the grammar. This undoes the effect of step 8, and once again the comments about theory and practice apply.

11 Return.

Thus the evaluation rules, working in concert, create a new identifier of the correct type, integrate it into a local extension of the grammar, use that extension to develop a clause, so that the new identifier is available in its correct scope in the PT.

Evaluation rules for a rag must be written in a language that is capable of accessing attributes synthesized by children, creating productions, pushing productions onto a grammar, synthesizing new attributes, and so on. Experiments suggest that a simple machine-code-like language would be adequate, with operations like *expand node* or *push production onto grammar.* The node-constructing procedure would include an interpreter for this language, and carry out the evaluation instructions associated with the alternative as it expanded the node.

A specification for such a language is not included in this thesis. Instead, a pragmatic but no less general approach was used, and a node-constructing procedure specific to S-algol was developed. Rather than a general interpreter, a numbered set of specialised methods was developed in Java. Method 0 is the default method, method 1 defines new identifiers, method two deals with *for* clauses, and so on. In all, 10 methods were developed. The evaluation rules associated with an alternative then reduce to a number specifying which method to apply when this alternative is chosen. Thus the methods were compiled rather than interpreted. From a theoretical basis the evaluation rules are written in a language that has an extremely simple syntax (namely, the set of integers from 0 to 10) and a rather complex semantics (namely, the 10 compiled methods).

The sag term *evaluation rules* does not quite fit the rag model. Instead of a set of simultaneous equations defining each synthesized attribute, we now have a procedure carrying out various operations. For this reason, I use the term *production method* instead. A production method is a procedure, written in a suitable language, associated with an alternative RHS of a rag production. If an alternative in a rag has no production method, it is associated with a default production method.

To start ontogenesis, a PT is constructed by calling the node constructor with the start symbol of the language, a fresh genotype, and the rag for the language. The rag at this point is called the *root grammar.* It is special in that it defines the language before any program-specific declarations have been made, and when the implications of doing so are embodied in its production methods.

4.2.2 The S-algol rag

This section presents a rag for S-algol. In so doing, it addresses various language issues concerned with the development of a rag.

The rag developed here does not encompass the entire S-algol language. For the purposes of this thesis, it was considered sufficient to demonstrate a rag capable of handling basic types, variables, iteration, and procedures. Language features such as arrays, data types, and various specialised types such as pixels and files, should be within the capabilities of rags, but would not add significantly to the value of this work.

The S-algol rag is given in §B.

4.2.2.1 Basic types

The S-algol basic types are *void, int, real, bool* and *string.* There is also a *constant modifier* so that, for example, *creal* is the type whose value is *real,* but which cannot be assigned to.

The [Morrison, 1979] definition of S-algol uses a CFG plus a set of type rules which first define various type categories (eg *arith* is *int* or *real*) and then define how types and syntax interact. For example, the rule for a *while* clause is given as:

```
while <clause>: bool
do <clause>: void => void
```

This means that the test expression in a *while* must be *boolean*; the object of the while must be *void,* and the whole *while* clause is *void.*

A rag is locally context free, and must be able to represent types in such a way that only options of the correct types are available. The only way to do this is to include type information in the nonterminal symbols. Thus, where the S-algol CFG defines an expression, the rag must distinguish between *void, int, real, bool* and *string* expressions. This is done by appending the type name to the symbol name. For example, the top level S-algol expression nonterminal is *exp0.* The rag defines 5 versions of this, namely *exp0 void, exp0 int, exp0 real, exp0 bool* and *exp0 string.*

4.2.2.2 Spaces

As is usual for a language recogniser, S-algol's CFG does not specify how spaces are used. This does not cause problems in parsing, but can in generating. For example, S-algol defines:

```
<exp3>      ::=    <exp4>[<add_op><exp4>]*
<exp4>      ::=    <exp5>[<mult_op><exp5>]*
<exp5>      ::=    [<add_op>]<exp6>]
```

which permits:

```
5 + + 6
```

as an int expression. But without spaces, this becomes:

```
5++6
```

which generates a type error, because ++ is a string operator. Similarly, spaces affect the following expressions:

```
7 rem 3
```

and

```
7rem3
```

which generates a syntax error because there appears to be an operator missing between the int `7` and the identifier `rem3`.

It is not sufficient to put spaces around every terminal symbol, because this leads to errors like:

```
1 2 3
```

instead of

```
123
```

The solution strategy is a combination of tactics. First, use spaces as a prefix, not as a suffix. (It could just as well be the other way around: the important thing is consistency.) Second, include spaces in the representation table wherever possible. For example:

```
div symbol: " div"
```

This is the most efficient way to supply spaces. Essentially, a space is used to prefix any symbol which we know must be a token in its own right. Thus `div symbol` has a space, because it is only used as an integer operator. But `plus symbol` cannot have a space, because it may be used in a string. Third, and last, we add `space symbol` into language productions wherever else a space is needed. For example:

```
eq op:      space symbol, equals symbol;
            tilde equals symbol.
```

Bottom-level symbols (`space symbol` and `equals symbol`) which have no specific meaning of their own are being combined to make a higher level symbol (`eq op`) which has a more specific meaning. But `tilde equals symbol` is specific enough (it is always used to define a *not equals* relation) to be defined with its own prefix space. In some cases, new nonterminals have to be defined to carry out this strategy completely.

4.2.2.3 Identifiers

Declarations introduce new identifiers which are available for use within their own scopes.

As is usual, S-algol's CFG defines an identifier as a sequence of letters, digits and periods, beginning with a letter. However, this makes no distinction between variable names and reserved words; for example, `while` is a valid identifier according to the CFG. Provided the language only has a finite number of reserved words, this problem can easily be avoided by generating identifiers in a form which ensures they cannot clash with any reserved word.

The same argument applies to predeclared identifiers. S-algol predeclares and initialises the following:

```
int   r.w         !width of real output field
int   i.w         !width of int output field
int   s.w         !spaces between output fields
cint  maxint      !largest int
creal maxreal     !largest real
creal epsilon     !smallest 1+epsilon > 1
creal pi          !mathematical constant
```

plus a few others of types not implemented in this work.

In this thesis, identifiers are generated in the form `t.`*n* where `t` is the type of the identifier and *n* is a serial number. For example, the identifier of a *for* clause control variable might be `int.4`. The serial number starts at 0 and is incremented by 1 for every identifier that is issued (ie all types draw from the same serial number stream). Note that the dot symbol is not an operator in S-algol: the dot in `int.4` is simply a character in the identifier.

4.2.2.4 Scope

The scope of an identifier is the region of the program in which the identifier can be used to refer to the object it identifies. Because of the way a rag works, it is most convenient if the scope of an identifier coincides with a subtree of the PT, and preferably a subtree whose root is close to where the identifier is introduced. Provided this is the case, it is fairly simple to add the appropriate productions to the grammar which is applied at the root of the subtree, so that the new identifier is incorporated into the language in precisely that part of the program which is its scope. Difficulties can arise if the scope rules are not well matched to the syntax.

Matching scope rules with syntax requires some care in designing the grammar. For example, suppose a language defines a program to be a sequence of declarations followed by a sequence of statements, where the scope of an identifier begins after its declaration and continues to the end of the enclosing block. (This is the case for S-algol identifiers.) Consider the following sample program:

```
let x = 3
let y = x * 5
let z = x * x + y
if z/x > x/y
      then write "yes"
      else write "no"
```

We might expect a CFG along the following lines:

```
program:            decls, stmts.
decls:              decl; decl, decls.
stmts:              stmt; stmt, stmts.
decl   :            decl real;
                          decl int;
                          ...
stmt   :            stmt assign;
                          stmt if;
                          stmt while
                          ...
```

where ... in the definitions of `decl` and `stmt` mean that a variety of different types of declaration and statement are defined. (The representation table is omitted to save space.) Using this grammar, the PT for the sample program is:

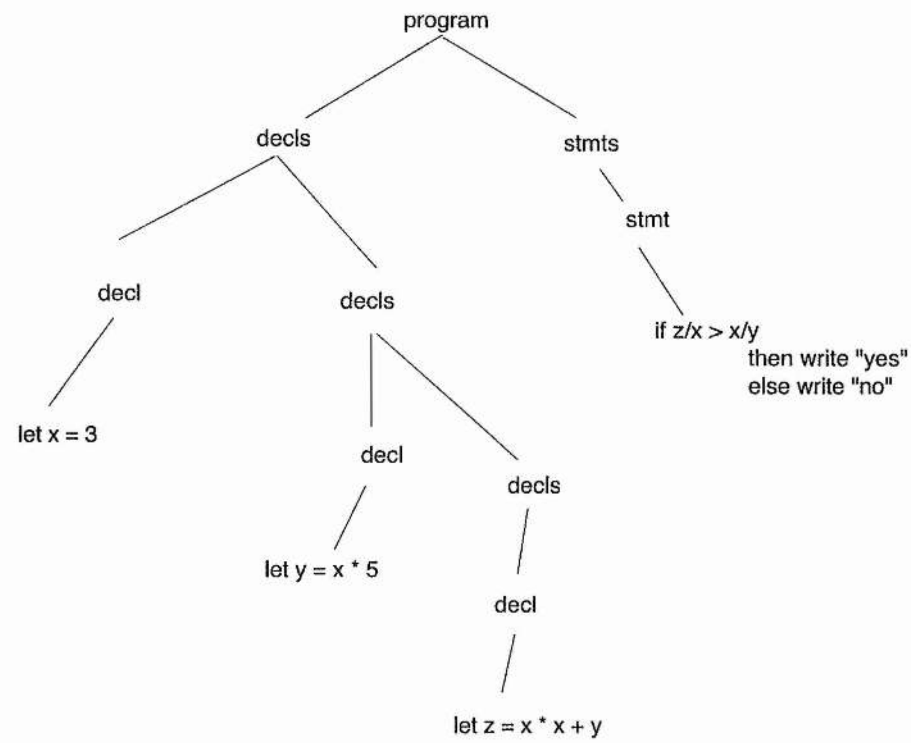

Figure 4-2: PT not amenable to scope rules

This PT reveals a poor grammar design, at least as far as scope is concerned. The scope of x is the rest of the program, but the rest of the program is not a single subtree. The scope of z is the *if* statement, which is about as far away in the PT as it is possible to be in this little example.

By using the following grammar, we can avoid this problem:

```
program:          decl, scope.
scope:            decl, scope;
                        stmts.
stmts:            stmt; stmt, stmts.
decl  :           decl real;
                        decl int;
                        ...
stmt  :           stmt assign;
                        stmt if;
                        stmt while
                        ...
```

Here, the nonterminal `decls` has been removed and replaced by a nonterminal `scope` which explicitly structures the PT to conform to the scope rules. There is no need to do the same for `stmts` because statements do not introduce new identifiers (though you might choose to restructure `stmts` for other reasons, eg cosmetics or consistency). Using this grammar, the PT for the same program is:

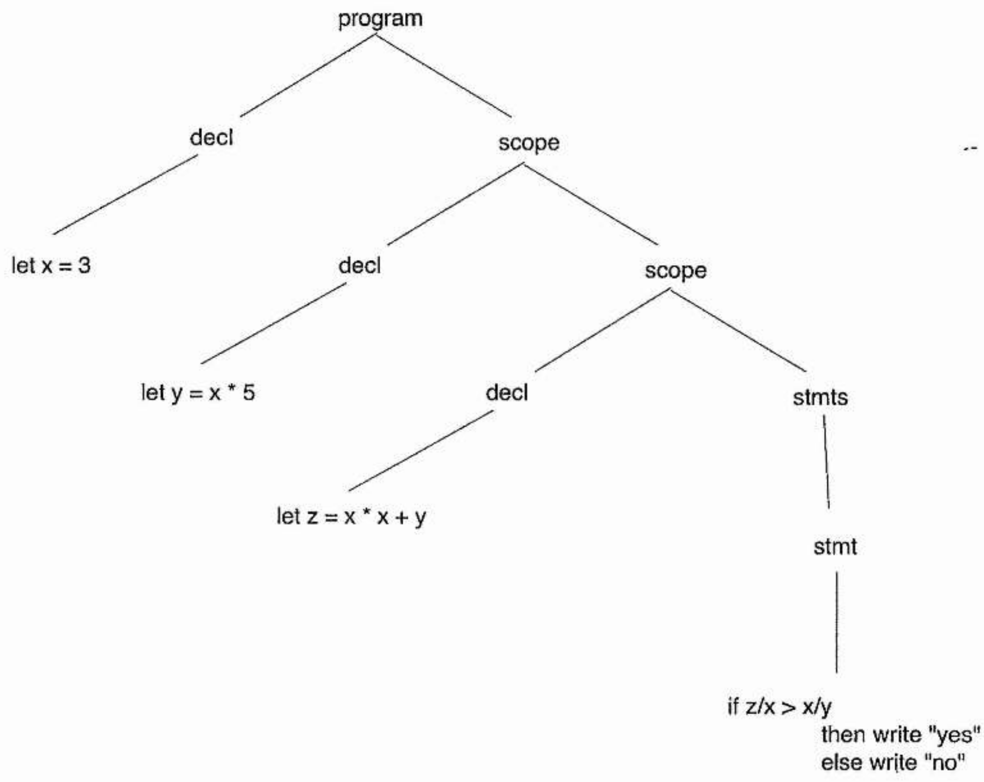

Figure 4-3: PT amenable to scope rules

In this PT, the production methods needed for the `program` and `scope` nodes are similar: get the identifier from the left hand child and push it onto the grammar for expanding the right hand child.

Thus, it may be advisable to modify a grammar to make it simpler to implement as a rag, if there is poor correspondence between the syntax and the scope rules.

In human programming, holes in scope are an issue. A hole occurs when a variable is declared with the same name as one in an outer block. The inner declaration masks the outer declaration. However, holes in scope do not add to the power of the language to represent algorithms; they merely make life easier for programmers. The following programs are equivalent:

```
!program 1: hole
let x = 3
for i = 1 to 10 do {
      let x = i * x
      write i, x, "'n"
}


!program 2: no hole
let x = 3
for i = 1 to 10 do {
      let y = i * x
      write i, y, "'n"
}
```

Program 1 declares two variables with identifier **x**. Program 2 declares the same variables with identifiers **x** and **y**. Program 1 creates a hole in the scope of the first variable **x**. Program 2 creates no hole in any scope. The point is that by renaming variables, it is possible to avoid holes in scope. Since the scheme for constructing identifiers used here ensures that all identifiers are unique, holes in scope cannot arise, and as the above example shows, we don't lose anything by this.

S-algol has simple scope rules — that is, they are simple to define, to understand and to implement, and they are well matched to the syntax. Other languages such as Java have more complex scope rules which allow, for example, an attribute to be used in any method of the class, whatever order the attribute and the class are declared in. However, our aim is (1) to synthesize programs, not to analyze them, and (2) to be able to generate all algorithms, not all programs. Consequently a rag for Java could generate all attributes before all classes, so that all attribute declarations are available in the PT when they are needed.

4.2.2.5 *For* clause

The *for* clause is the simplest example of a declaration in S-algol. It declares a control variable of type *int*, whose scope is a single void clause. S-algol provides two forms of *for*, one with an implied increment of 1, and one with an explicit increment. Only the first form is implemented here.

The implementation of *for* in a rag is described in §4.2.1.4 *Reflective attribute grammars.*

4.2.2.6 *Let* declarations

S-algol has 4 basic types (*int*, *real*, *bool* and *string*) which can be used in a let declaration. These double to 8 if the constant form is allowed; but constant declarations are not necessary for the purposes of this thesis and are therefore not implemented. There are predeclared constants in S-algol, so it is clear that rags are capable of handling them if it were desired.

The mechanism is similar to the *for* clause except that there are 4 types and the introduced identifier is less accessible, being several nodes away from where it is needed in the PT. A more complex production method is needed to access it and apply it. The S-algol productions involved in an *int* declaration are given in [Morrison, 1979] as:

```
<sequence>          ::=   <declaration>[;<sequence>]|
                                 <clause>[;sequence>]

<declaration>       ::= <let_decl>|
                                 <structure_decl>|
                                 <proc_decl>|
                                 <forward>
```

(It is possible to have a sequence which ends with a declaration. At first sight this appears to be pointless, but it is possible for the declaration to have side effects. However, a final declaration has an empty scope.) The definition of `<sequence>` is rewritten for the rag as follows:

```
sequence void:     clause void;
                   clause void,
                         sequence separator,
                         sequence void;
                   decl let;
                   decl let,
                         sequence separator,
                         sequence void;
                   decl proc;
                   decl proc,
                         sequence separator,
                         sequence void.

sequence int:      clause int;
                   sequence void,
                         sequence separator,
                         clause int.

sequence real:     clause real;
                   sequence void,
                         sequence separator,
                         clause real.

sequence bool:     clause bool;
                   sequence void,
                         sequence separator,
                         clause bool.

sequence string:   clause string;
                   sequence void,
                         sequence separator,
                               clause string.

sequence separator:
                   space symbol,
                         semicolon symbol,
                         newline symbol.
```

The rag form is much longer, for several reasons. First, the untyped S-algol `sequence` and `clause` expand to 5 typed rag `sequences` and `clauses`. Second, S-algol's form of BNF allows metasymbols like `[ ]` and `|`. Third, the rag must make the punctuation explicit, by stating where sequence separators occur.

The production method for *procedure* declarations is quite different to that for *let* declarations, so the two types of declaration are separated at the top level. *Procedure* declarations are dealt with in the next section.

The definition of `sequence separator` includes a newline. Unless newlines are added explicitly (here and elsewhere) the generated program is a single line which is much longer than a sane person would write. These long lines exposed a bug in the compiler which caused compilation to fail.

Rather than fix the compiler, it was simpler to insert newlines at suitable points. They also made the phenotype programs easier to read.

The definition of `<declaration>` includes structures and forward declarations which are not implemented in this thesis. The definition of let declarations is rewritten as follows:

```
decl let:               decl let int;
                        decl let real;
                        decl let bool;
                        decl let string.

decl let int:
                        let symbol,
                              id int new,
                              assignment symbol,
                              clause int.

decl let real:
                        let symbol,
                              id real new,
                              assignment symbol,
                              clause real.

decl let bool:
                        let symbol,
                              id bool new,
                              assignment symbol,
                              clause bool.

decl let string:
                        let symbol,
                              id string new,
                              assignment symbol,
                              clause string.

id int new symbol:        " idIntNew"
id real new symbol:       " idRealNew"
id bool new symbol:       " idBoolNew"
id string new symbol:     " idStringNew"
```

Again, the rag is much longer than the original version. The new identifier nonterminals (`id int new`, etc) are defined by corresponding symbols (`id int new symbol`, etc). As explained in the outline above, these terminal symbols are essentially placeholders, which are necessary only so that the grammar is well-formed. The production method for the new identifier nonterminals ignores them and converts their own node from nonterminal to terminal. However, the placeholder symbols are occasionally useful in development or debugging.

Only alternatives #3 and #5 of sequence void need production methods. Alternative #3 introduces a variable identifier, and #5 introduces a procedure identifier. The other alternatives either don't introduce new identifiers or introduce identifiers which have empty scopes.

Alternative #5 is dealt with in the next section.

The production method for alternative #3 (`decl let, sequence separator, sequence void`) digs down two levels to the declaration and retrieves the new identifier. It then creates a production of the appropriate type, pushes it onto the grammar, expands the `sequence void`, and pops the identifier from the grammar. Digging down more than one level is not the best design: it would have been cleaner to have the method for `decl let` do some of the work. The production method must find the type of the declaration in order to generate an appropriate production to push onto the grammar.

S-algol does not contain any predeclared variables of type *real*, *bool* or *string*. At face value this means the root grammar has no productions to define these identifiers:

```
id real
id bool
id string
```

and of course these nonterminals don't appear in any RHS either. Thus, while the following `int` productions exist in the root grammar:

```
exp6 int:          id int.
id int:            real width symbol;
                   string width symbol;
                   int width symbol.
```

The corresponding `real` productions don't exist:

```
exp6 real:         id real.
```

because there are no predeclared `real` identifiers. (There are predeclared *constant* `reals`, but constant `real` is not the same as `real`.)

This means that when a *real*, *bool* or *string* identifier is introduced, it is necessary not only to add a production for the identifier in question, but also to add a production for identifiers of that type in general. For example, if we introduce `real.0`, we need two productions:

```
exp6 real:          id real.
id real:            real.0 symbol.
```

As we continue to expand nodes and grow the PT, it is possible that a further *real* identifier is introduced, in the scope of `real.0`. In this case, we only need to add an alternative to the `id real` production, so that it becomes:

```
id real:            real.0 symbol;
                    real.1 symbol.
```

The proper way to do this would be for the production method to test the grammar to see whether `id real` was already defined, and take either the first course of action or the second as appropriate. In this thesis, however, I took a pragmatic course of action and predeclared a *seed variable* of each type, including `int` for consistency. That is, I modified the language slightly to sidestep the problem. The modifications are:

Add a declaration for each type to the standard preamble:

```
let INT := 0
let REAL := 0.0
let BOOL := false
let STRING := ""
```

Add productions for assignment for *real, bool* and *string* to `clause void` (*int* assignment is already declared):

```
clause void:
                    id real,
                          assignment symbol,
                          clause real;
                    id bool,
                          assignment symbol,
                          clause bool;
                    id string,
                          assignment symbol,
                          clause string.
```

Add productions for each type of id, with RHS equal to the predeclared symbol:

```
id int:             variable int symbol.
id real:            variable real symbol.
id bool:            variable bool symbol.
id string:          variable string symbol.
```

Add productions to allow level 6 expressions to produce identifiers of each type:

```
exp6 real:          id real.
exp6 bool:          id bool.
exp6 string:        id string.
```

Add productions to produce the terminal symbols:

```
variable int symbol:           " INT"
variable real symbol:          " REAL"
variable bool symbol:          " BOOL"
variable string symbol:        " STRING"
```

Having added these seed variables, the production method for sequence void is always able to push a new identifier of any type onto the grammar without further ado. However, a disadvantage of this approach is that the name space is slightly polluted, or diluted, by the seeds, since they provide extra routes to generate various forms of zero. This presumably makes it slightly more difficult to find genes that generate values other than zero. It also makes phenotypes more wordy than they need be.

4.2.2.7 *Procedure* declarations

S-algol procedures may be recursive (ie the scope of a procedure includes its own body). Mutual recursion requires a *forward* declaration. Higher-order procedures — ie procedures which take procedures as parameters — are also supported.

For this thesis, only first-order procedures and self-recursion (ie not mutual recursion) was implemented. Two production methods for procedure declarations were implemented, one which allowed recursion, and one which did not, for comparison.

The S-algol syntax for parameter strings allows adjacent parameters of the same type to dispense with the second and subsequent type identifiers. For example, the following prototypes are equivalent:

```
procedure m (real x, y; int z -> real)
procedure m (real x; real y; int z -> real)
```

This is just syntactic sugar; the rag generates parameter lists in the second form only. That is, a parameter list is a semicolon-separated list of parameters, each of which consists of a type identifier and a variable identifier. The rag syntax for procedure declarations is as follows:

```
decl proc:               decl proc void;
                         decl proc int;
                         decl proc real;
                         decl proc bool;
                         decl proc string.

decl proc void:
                         proc symbol,
                               id proc new,
                               sequence separator,
                               clause void;
                         proc symbol,
                               id proc new,
                               round l symbol,
                               parameter list,
                               round r symbol,
                               sequence separator,
                               clause void.

decl proc int:
                         proc symbol,
                               id proc new,
                               round l symbol,
                               arrow symbol,
                               type int symbol,
                               round r symbol,
                               sequence separator,
                               clause int;
                         proc symbol,
                               id proc new,
                               round l symbol,
                               parameter list,
                               arrow symbol,
                               type int symbol,
                               round r symbol,
                               sequence separator,
                               clause int.

parameter list:          parameter;
                         parameter,
                               parameter separator,
                               parameter list.

parameter:               type int symbol,
                               id int new;
                         type real symbol,
                               id real new;
                         type bool symbol,
                               id bool new;
                         type string symbol,
                               id string new.
```

For brevity, only `decl proc void` and `decl proc int` are shown in full. The productions for *real*, *bool* and *string* are similar to the *int* version.

When a procedure is declared there are two scopes of interest:

- The body of the procedure.
- The `sequence void` following the procedure body.

I refer to these as the *internal* and the *external* scope respectively. The scope of the parameter declarations is the internal scope. The scope of the procedure is both the internal and external scopes (if we wish to enable recursion) or just the external scope (if we do not wish to enable recursion).

The mechanism for introducing a new procedure identifier is the same as for variable identifiers — that is, the production method for nonterminal `id proc new` generates an identifier of the form `proc.`*n* and converts the node from nonterminal to terminal. For the same reason that seed variables of each type are required, it is necessary to define seed procedures of each type. Identity procedures which return their arguments are used for this: For example:

```
    procedure PROC.INT (int x -> int); x
```

To manage the information involved in procedure declarations, three new attributes are required. These are conventional synthetic attributes in the sag sense. In terms of the implementation, three new object classes are introduced, and three fields of these types are added to the Node class. The three object classes are Prototype, Parameter, and ParameterList.

A Prototype object is synthesized when any type of procedure declaration node is created. It serves to represent the prototype that is declared. Once the Prototype object is complete, it is used by the parent `sequence void` node to generate internal and external productions.

A ParameterList object is synthesized for each parameter list node. If the child is another `parameter list` node than the child's list becomes the tail of the parent's list. If the child is just a `parameter`, then a new ParameterList object is started, with one element. The topmost `parameter list` node is the child of a procedure declaration. This takes the list and incorporates it into the Prototype object.

4.2.2.8 Literals

Literals are generated by microsyntax, ie by productions like:

```
literal int:                space symbol,
                                 numeral 0 symbol;
                            space symbol,
                                 digits.

digits:                     digit;
                            digit,
                                 digits.

digit:                      numeral 0 symbol;
                            numeral 1 symbol;
                            numeral 2 symbol;
                            numeral 3 symbol;
                            numeral 4 symbol;
                            numeral 5 symbol;
                            numeral 6 symbol;
                            numeral 7 symbol;
                            numeral 8 symbol;
                            numeral 9 symbol.
```

This is expensive in that many genes are needed. However, the benefit is that each genotype can evolve whatever literals it requires. There is no convenient way to generate SGP's random ephemeral constants. Generating random literals in production methods would be possible, but then the value of the literals cannot be replicated.

4.2.2.9 Preamble, postamble, program

Although not part of the S-algol language, these items are necessary to use the language in Gads 2. Each phenotype is generated in an S-algol wrapper comprising a preamble and a postamble.

The same preamble is used in all problems. It contains debugging control, seed variables, seed procedures, protected procedures (eg to avoid divide-by-zero errors), enhancements (`FLOOR` and `CEILING` procedures are not provided by S-algol), RNG setup, a procedure to map fitness into a standardised scale, and code to initialise the IO environment.

The postamble is problem-specific. In all problems except Annie the phenotype is coded as a procedure body, since this makes it simple to evaluate the fitness over a set of test cases. This is often done by comparing the observed value (from the phenotype) with an expected value, and computing the RMS error. The procedure prototype, if any, is problem-specific.

In the course of the investigation, eight problems and some variations of.. them were investigated. Each problem was designed to exercise a different aspect of the rag system, and consequently each problem required a different subset of the S-algol language. To avoid having to maintain a suite of almost identical rags, a single rag that provided for all the problems was developed, with a system of top-level nonterminals to make it simple to switch between them. Thus, to obtain the Monkey problem, the first line of the rag (which defines the start symbol) is:

```
program:            program monkey.
```

while to obtain the multiplexer program, the first line becomes:

```
program:            program multiplexer.
```

And so on. Each program has its own section where it redefines the symbols with the values it requires, such as the phenotype. For example, the phenotype for monkey is a `literal string`, while the phenotype for cart is `exp3 real`. These definitions, coming after the standard ones, replace them. All other problem-specific program sections are commented out, so that the rag is specialised for just one problem at a time.

5 Gads 2

This section describes the Gads 2 experiments. The aims of the experiments were:

- To demonstrate the evolution of type-correct programs in a context sensitive language independent of the GA implementation.
- To obtain preliminary performance measurements of the system, including a rational scalar comparisons of problems and engines.
- To demonstrate visualisation techniques.

Like Gads 1, Gads 2 is not optimised for performance. Similarly, the implementation of rags is not intended as a polished product. It is a single-use design consistent with the advice given in [Brooks, 1995], namely *plan to throw one away.*

The main subsections below are as follows:

§5.1 The origins of Gads 2
Describes the relationship between Gads 1, GE and Gads 2.

§5.2 Systems
Describes the software and hardware used for the experiments, and the general experimental setup that was common to all the problems investigated.

§5.3 Problems and individual results
This section describes the problems and the results on an individual basis.

§5.4 Comparative results
This section compares the results from different problems.

5.1 The origins of Gads 2

The main precursors of Gads 2 are Gads 1 and GE [Ryan, 1998a], [O'Neill, 2001b]. The chronological order of these systems is:

$$Gads1 \rightarrow GE \rightarrow Gads2$$

This shows the essential relationship between Gads and GE but ignores influences which other systems also had on their design. The following subsections discuss the main differences between Gads 1, GE and Gads 2.

5.1.1 Translation: mapping genes to productions

In Gads 1 a gene selects the production with the same number. That is, the mapping from gene value to production number is the identity function: gene value 5 always selects production 5. This leads to unacceptably large numbers of introns, because the probability of selecting a production that can actually be applied falls as the number of productions rises.

In GE a gene is mapped from the set of gene values (say, [0, 255]) to the set of applicable production numbers (say [0, 3]), by computing the gene value *modulo* the number of relevant production alternatives. That is, GE computes the remainder when the gene value is divided by the number of production alternatives. For example, to map the gene value 121 into [0, 3] GE computes 121 modulo 4 = 1.

In Gads 2, the mapping method is to scale the gene by a linear transformation. For example, scaling [0, 255] into [0, 3] means that genes with values in [0, 63] map to 0, genes in [64, 127] map to 1, genes in [128, 191] map to 2 and genes in [192, 255] map to 3.

The analogous biological process is *translation*, which matches codons with amino acids during protein syntehsis. Whatever function is used, the important property is that the same genotype always produces the same phenotype. In both biology and GE the translation is many-to-one. That is, there may be more than one gene value which corresponds to an amino acid or production alternative. In both biology and GE this fact — referred to as genetic code degeneracy — promotes genotypic diversity.

5.1.2 Repair: wrapping and defaults

In Gads 1 the chromosome is scanned exactly once. This can leave the phenotype in an incompletely developed state, with nonterminals still unexpanded.

Gads 1 dealt with this by a rudimentary repair mechanism that assigned default values (generally zero) to undeveloped parts of the phenotype during evaluation. This approach was possible in Gads 1 because the language was small (so zero was always a legal option) and because the phenotype was evaluated by a purpose-built interpreter which could detect when default values were needed and supply them. At the time of Gads 1 it was thought that the defaults system would not be easy to scale to a large grammar.

In GE the chromosome is scanned multiple times, up to a prescribed limit. This technique is called *wrapping*. It means that any given gene may be used multiple times. The effect of a gene depends on the nonterminal it is applied to. Given the same nonterminal, with the same set of production alternatives, the same gene always translates to the same alternative. But with wrapping, the gene may be used to select an alternative from a different production in each pass. Translation ensures that the gene will always make a valid choice. The main effect of wrapping is to make the ontogenic mapping more likely to produce a completely developed individual. It also affects the actual phenotypes that are produced by the ontogenic mapping.

Wrapping is analogous to the chromosome being in the form of a ring buffer, not a list or string; this is in fact the case in prokaryotic bacteria.

Wrapping is not sufficient to guarantee the complete development of the individual. It is possible to have a combination of genes which never complete development, no matter how many times they are wrapped. For example, using Syntax A (of table 2-3) a chromosome composed only of genes which select productions 1 and 5 will produce a phenotype that grows without limit. In GE, an individual which is incompletely developed after the wrapping limit has been reached is given the lowest possible fitness value.

Thus wrapping is not a complete solution to incomplete development, though by improving the completion rate of the ontogenic mapping, it reduces the need for a repair mechanism in the first place. Wrapping should be seen as part of the ontogenic mapping, not as a repair mechanism. By using wrapping, GE reduces the need for a repair mechanism to such an extent that GE does not have a repair mechanism.

Informal experiments for Gads 2 suggested that wrapping would not be so effective with a full-size grammar. The reason for this appears to be not so much the larger size of the grammar as its construction. For example, the definition of an expression is recursive and it is always possible to start a new sub-expression in parentheses. This is typical for a realistic language. Wrapping, instead of completing development, simply causes deeper and deeper nesting of new sub expressions, without limit.

Given a high proportion of incomplete individuals it would not be practical to assign them all minimum fitness values, as in GE. A repair mechanism is necessary. For this reason the defaults mechanism of Gads 1 was improved, as outlined in §4.2.1.2. In Gads 2, the first production alternative for a given nonterminal that is listed in the grammar is the default for that nonterminal. By ordering the production alternatives from simplest to most complex, it is easy to set up a system of defaults that ensures every nonterminal has a sensible default, and avoids the mistake of having a recursive rule as the default. The concern in Gads 1 that defaults would be difficult to organise for a larger grammar proved to be groundless.

The defaults repair mechanism means that every individual develops completely, so it can then have its fitness evaluated in the ordinary way. This is valuable, since there are differences in the performance of these individuals, even though they are not completely developed using the ontogenetic mapping, and their genetic contribution may be lost if the individuals are given unfairly low fitness values.

In Gads 2, wrapping was not used at all (more correctly, the wrapping limit was set to zero so that the chromosome was scanned only once). It would have been possible to have a higher wrapping limit in Gads 2, but optimising the performance of Gads 2 was not the aim of the experiment, and this possibility was not explored.

5.1.3 Genotype: fixed length or variable length

Gads 1 used fixed length chromosomes, for simplicity. Three different lengths were investigated.

GE uses variable-length chromosomes. It is not clear that variability is essential for GE to work, though using variable length chromosomes allows GE to employ genetic operators that are not available to fixed length systems. *Duplicate* randomly selects and copies a part of the chromosome, and inserts the copy immediately before the last gene in the chromosome. *Prune* discards the unused tail of a chromosome, in the event that the ontogenic mapping completes before all the genes have been scanned. These operators change the length of the chromosomes. Steps must be taken in any variable length system to ensure that the chromosomes do not grow to unreasonable sizes.

Gads 2, again for simplicity, uses fixed length chromosomes. A length of 1 000 genes per chromosome is used in the experiments. This size was chosen, following informal experiments, as being large enough to produce all the phenotypes that were to be investigated, but not so large as to cause any system problems.

5.1.4 Genetic operators: crossover

In Gads and GE, genetic operators work on the genotype, which is an array or list of integers, rather than a tree as in SGP. Because the ontogenic mapping ensures that any genotype can be expressed as a phenotype, issues of syntax, type-correctness, etc do not arise. However, the Gads/GE genotype does have an implicit internal structure, because of the way it is interpreted during ontogenesis. The following paragraphs explain how the structure arises and its interaction with simple one-point crossover.

First, there is an implied sequence in the genotype. A gene nearer the start of the chromosome may affect the operation of a gene nearer the end of the chromosome, but not vice-versa. For this reason it would be more accurate to describe the genotype data structure as a list or string than an array. This is true even if the genotype is implemented as an array.

Second, ontogenesis may produce a fully developed individual before it reaches the end of the chromosome. In such a case, the chromosome consists of an active head, which is expressed in the phenotype, and an inactive tail, which is not expressed. We can always describe a chromosome as a head and a tail if we allow that the tail may be of zero length. Suppose we now carry out a one-point crossover of two same-length parents P1 and P2. The crossover point may lie in the head or in the tail, in either parent, giving three different possibilities: X1, X2 or X3 (figure 5-1):

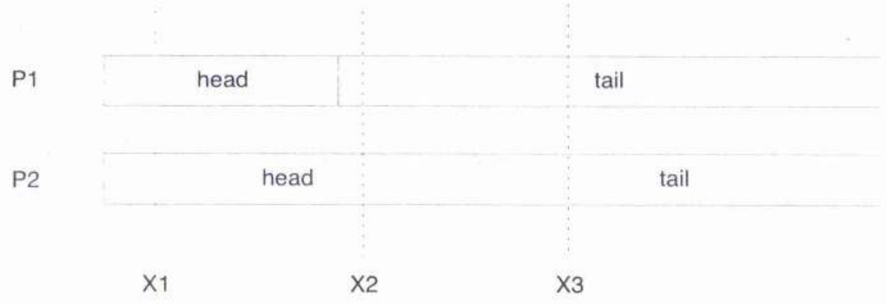

Figure 5-1: One-point crossover

The kind of children produced depends on the crossover location, as summarised in table 5-1.

	Crossover X1	**Crossover X2**	**Crossover X3**
Child C1	P1[..X1] + P2[X1..]	P1[..X2] + P2[X2..]	P1[..X3] + P2[X3..]
Child C2	P2[..X1] + P1[X1..]	P2[..X2] + P1[X2..]	P2[..X3] + P1[X3..]

Table 5-1

In terms of the amount of genetic mixing that is produced, it does not matter whether the crossover is of type X1, X2 or X3. But the phenotypic effect (if we ignore genetic code degeneracy, and the possibility of parents having identical genes, for the moment) does depend on the crossover type. Children produced by X1 are distinct from both parents. Child C1 produced by X2 is phenotypically identical to P1 because it inherits the entire P1 head. Child C2 produced by X2 is phenotypically distinct from both parents. Both children produced by X3 are phenotypically identical to their parents. Thus simple crossover may not produce as much change to the phenotype population as might be expected.

Despite this apparent inefficiency, producing phenotype clones is not a waste of time, because the genetic material in a tail can become active at a later time. For example, child C2 by X1 inherits the shortest combination of parental heads: P2[..X1] followed by P1[X1..]. When this combination is used in ontogenic mapping, the effect of P1[X1..] depends on the effect of P2[..X1], because of the implicit genotype sequencing. This may result in the length of C2's head being less than, equal to or greater than the length of P1's head. If C2's head is longer than P1's, then genes from P1's tail are expressed in C2's phenotype. Thus, what was inactive can become active.

Gads 1 used uniform crossover, rather than one-point crossover, and made no allowance for the chromosome structure. In uniform crossover each gene in a child is equally likely to have come from either parent. It is possible that this form of crossover might be very disruptive to the implicit sequence of the genes in Gads, though the ontogenic mapping ensures that no invalid genotypes can result.

GE's default crossover operator is a simple one-point crossover much as described above, though the chromosomes are variable length and each parent's crossover point is chosen independently.

[O'Neill, 2001b] refers to various forms of homologous crossover, in which there is a deliberate attempt to ensure that the genetic material that is exchanged is in some sense equivalent. The aim is to avoid overly destructive crossover which might result from exchanging material that is entirely unrelated and unsuitable for the context into which it is placed. (In biology, *homologous chromosomes* have the same or allelic genes with genetic loci usually arranged in the same order, so that DNA in matching positions on two chromosomes serves a similar purpose in each.) [O'Neill, 2001b] introduces a new form of two-point homologous crossover for GE. In the standard form of this operation, the two parents are first compared, gene by gene, from the start of the chromosome. Genes are considered to match if they translate to the same production. The first crossover point is immediately before the first pair of genes which do not match. The second crossover points are chosen randomly and independently in both parents, in the region to the right of the first crossover points. There is also a variant form of crossover in which the second crossover points are at the same locus in both parents, so that the children are same lengths as the parents.

Gads 2 uses a one-point crossover, modified to ensure that the crossover point is always in the head of both parents. That is, it is restricted to type X1 of figure 5-1. Since Gads chromosomes are fixed length, the crossover position is the same in both parents. The reason for this choice is that the large size of Gads 2 chromosomes (1 000 genes), when combined with parsimony, can result in relatively short heads. If, in a given run, the typical head length is just 100 genes, then choosing a crossover point uniformly in the 1 000-gene chromosome will result in type X3 crossovers 90% of the time, and slower phenotype evolution.

5.1.5 Hobson's choice genes

In the definition of a language it is often desirable to define a production which has just one alternative on the RHS. For example, in Syntax A (table 2-3) there is only one rule for <arity 1>, namely production #7. In [Ryan, 1998a] the BNF definition of a C function has only one alternative for each LHS except <expr>. If, in the course of ontogenesis, such a nonterminal is produced, there is no choice about what it must expand to, no need to translate a gene to make that choice, and consequently no need to use up a gene at that point.

In Gads 1, which has an identity translation function, this is not an issue. In order to select production #7, it is necessary to have a gene with value 7.

In GE, as described in [Ryan, 1998a], genes are not used up unless necessary.

In the Gads 2 implementation of ontogenesis, these "Hobson's choice" genes are used up. They must therefore use up some resources. For example, they may weaken the effect of crossover by diluting the crossover point space with ineffective crossover points. Any such effect could trivially be avoided by modifiying the ontogenic mapping to only use up a gene for a real choice.

5.2 Systems

This section gives a top-down description of the software and hardware used for the experiments, and the general experimental setup that was common to all the problems investigated.

5.2.1 Computing facilities

The experiments were carried out on a laboratory with a server and about 60 client systems. Each run was started with 33 clients in parallel so that even if a few clients failed there would be a reasonable-sized sample of results. The runs were coordinated from the laboratory server.

The server ran Solaris 7; the clients ran RedHat Linux 7.1. In addition, the clients had Java 1.3.1 and S-algol [Kirby, 2000].

Each client also had the ECJ GA engine [Luke, 2001]. This is a well-built and well-documented GA system, written in Java, and with numerous hooks for the experimenter to extend in Java.

5.2.2 Gads 2 implementation

I developed Gads 2 as an ECJ "experiment", as suggested in [Luke, 2001]. This means that Gads 2 is entirely contained in a directory subtree of ECJ. A copy of the subtree is available at

```
ftp://ftp.dcs.st-and.ac.uk/pub/norman
```

The implementation is in two main parts: Java classes for a general rag; and extensions to specialise this for S-algol. The programming quality is admittedly rather rough, as my aim was to demonstrate the validity of the concept, not to develop an elegant or efficient implementation of it.

The main interface between ECJ and Gads 2 was provided by:

RAGInitializer
Called by ECJ at the start of the run. Sets up the rag.

RAGProblem4
Called by ECJ to evaluate an individual.

RAGStatistics
Called by ECJ to output information about the run.

RAGIndividual
The Gads 2 genotype was an ECJ IntegerVector with a modified crossover. A simple crossover works but may be less efficient.

5.2.3 ECJ parameters

Four engine configurations — ie ECJ parameter sets — were used. They were named Koz_0, Koz_1, Pat_0 and Pat_1. All configurations shared ECJ's *simple.params* with the following deviations:

Species
Integer vector

Population size
1 000 individuals

Genome size
1 000 genes

Gene value range
[0, 100]

Crossover points
1

Crossover probability
0.9

Mutation probability
0.01

Two further configuration settings were used, each with two values, making a total of 4 engine configurations. The intention of this was simply to ensure that there was some variety in the experimental setup, and that results were not specific to one configuration. The settings were:

Style
Koza or Paterson (see below).

Fitness computation
Either functionality alone, or functionality and parsimony.

Koza-style parameters were intended to be similar to those in [Koza, 1994]. They include:

Selection method
Tournament size 7

Generations
100

Paterson-style parameters were intended to be quite different from Koza's:

Elite
900

Selection method
Tournament size 1 (ie random selection)

Generations
991

The different numbers of generations were contrived so that both Koza- and Paterson-style had the same number of evaluations, namely 100 000. These limits — 1 000 individuals and 100 000 evaluations — are probably too small if your purpose is to find solutions to real problems.

Parsimony pressure was applied in 50% of the configurations. Configurations which ignored parsimony were given subscript 0. Parsimonious configurations were given subscript 1. The application of parsimony pressure is described below in §5.2.5 *Fitness: functionality and parsimony.*

5.2.4 Raw and transformed data scales

It was necessary to design scales for measuring fitness, and its components, functionality and parsimony. There were two basic reasons for this.

The first was that the range of the raw measure was sometimes as large as [0, `Float.MAX_VALUE`]. The value of Java's `Float.MAX_VALUE` is about 10^{38}. This number is so large that it is very difficult to comprehend. For example, a value of just one-tenth of one percent of this is 10^{35}, which is still enormous by human standards. It is very difficult to interpret values in so large a range.

The second was that the raw measures were error measures, in which 0 is the best and bigger values are worse. These had to be reversed so that zero was the worst and bigger values were better. There are two obvious ways to do this: take the reciprocal, or take the negative.

These two factors interact. For example, if the raw functionality is very small compared to the theoretical maximum, reciprocating or negating will lead to values which are almost indistinguishable, so that fitness-based selection will be jeopardised.

I adopted a standardised set of scale transformations to deal with this, with the aim of producing scales that were understandable and which produced a spread of values when real individuals were evaluated. The transformations are written as the procedure `map` in the preamble. The input to `map` is three boolean switches which specify the transformations required, an observed value, and the range it is in. The procedure returns a transformed value as follows:

logarithms

If the `LOG` option is selected, the scale and the observed value are replaced by their natural logarithms. This is normally used if the upper bound of the raw scale is `Float.MAX_VALUE` or `Integer.MAX_VALUE`.

For example, if the raw value r is in the range $[x, \; y]$ then it is transformed into the value $log\ r$ in the range $[log \; x, \; log \; y]$.

reverse

If the `REV` option is selected, the sense of the scale and observed value (which may by now be logarithmed) are reversed. This is used when the raw value is an error measure, such as RMS difference between observed and expected values.

For example, if the raw value r is in the range $[x, \; y]$ then it is transformed into the value $x + y - r$ in the range $[y, \; x]$.

scale

If the `SCA` option is selected, the observed (which may by now be logarithmed and/or reversed) are scaled into a specific range. This is used when the raw value is not in the standard range.

For example, if the raw value r is in the range $[x, y]$ then it is transformed into the value $(r - x) / (y - x) * m$ in the range $[0, m]$.

The final value of the transformed observation is returned.

5.2.5 Fitness: functionality and parsimony

Fitness is represented in ECJ as a real value in [0, `Float.MAX_VALUE`], with 0 being the worst possible.

Raw functionality was a problem-specific measure of how well the individual performed the objective task, as measured by the evaluation wrapper. The most common measure was the RMS difference between observed values (from the individual being evaluated) and expected values (from an ideal solution). (The term *expected* does not imply any actual expectation — it is an echo of statistical terminology.)

Raw parsimony was measured in terms of the leaf count in the phenotype. (An early version used the gene count. I switched to a phenotype-based measure because it seemed more consistent to base fitness measures on the phenotype than on the genome.) Leaf count is in the range [1, `Integer.MAX_VALUE`], but in practice is unlikely to be anywhere near `Integer.MAX_VALUE`.

I wanted to be able to read a fitness value in terms of functionality and parsimony scores. A simple way to do this is to use decimal digits before the decimal point to represent functionality, and those after the decimal point to represent parsimony. This also ensures that functionality takes precedence over parsimony. A Java float has a 24-bit mantissa, with an implied 1 bit in front after normalisation. This makes 25 bits, which can represent 2^{25} fixed-point values, or just over 7 decimal digits. I therefore represented fitness by a fixed-point decimal number of the form:

```
ffff.ppp
```

where `ffff` represents the functionality, and `ppp` the parsimony. Thus, functionality was transformed into a value in the range [0, 9999], and parsimony into a value in the range [0, 999], using the transformation procedure described above.

Using this method it is simple to interpret a fitness of, say 9825.881 as scoring 98.25% for functionality and 88.1% for parsimony. There is a slight difference between functionality and parsimony scores. The maximum functionality score of 9999 is usually achievable, since in the problems considered, the ideal result is known. The maximum parsimony of 999 is not usually achievable, because it corresponds to a phenotype of length 1. Of course we do not usually know the minimum phenotype length in advance. For example, the best solutions in the Annie problem scored 9999.876 and this is probably the highest achievable score.

It would have been possible to use multiobjective fitness, and might have allowed for a smaller grain, which might be helpful. However, parsimony and functionality are not independent and so Pareto optimisation quickly results in convergence on solutions with good parsimony but poor functionality, because parsimony is much easier to achieve. Multiobjective fitness is therefore not a simple option.

5.2.6 Crossover

Although all genomes were the same size, 1 000 genes, not all genes were needed for every ontogenic mapping. A parsimonious individual might only use 100 genes. The genome in such a case consists of an active head of 100 genes followed by an inactive tail of 900 genes. If crossover treats all possible crossover points as equally likely, then 9 times out of 10 the crossover point will be in the tail. The child which inherits the head of the genome will then be almost equivalent to the parent with the shorter head. I say *almost* equivalent because later crossovers might show up differences. In order to make crossover more effective, I introduced a version where the crossover points are guaranteed to be in the head of both parents.

This form of crossover is similar to the pruning technique described in [Ryan, 1998a]. However pruning is an operator that permanently modifies genotypes — genes are thrown away. The form of crossover here simply ensures that the crossover point is in the active region.

5.2.7 Hobson's choice

In each of the problems, several genes were used up in generating the wrapper. This requires no evolution because there are no choices to make, so all gene values select the same production. Nonetheless these "Hobson's choice" genes must exist in this implementation of the ontogenic mapping, and must therefore use up some resources. For example, they may weaken the effect of crossover by diluting the crossover point space with ineffective crossover points. Any such effect is expected to be slight, though it could trivially be avoided by modifiying the ontogenic mapping to only use up a gene for a real choice.

The ontogenic mapping in [Ryan, 1998a] does not use up genes unless it is necessary.

5.2.8 S-algol

The phenotype language chosen for the Gads 2 experiments is S-algol [Morrison, 1979] [Kirby, 2000]. S-algol is a general-purpose programming language which has the benefits of a pleasant grammar that makes it particularly suitable as a teaching language. In structure, it follows the general style of the Algol-like languages. Scalar data types include integers, reals, booleans and strings.

Integer operators `div` and `rem` compute integer division and remainder, respectively. For the purposes of GP, various procedures were defined. Procedures `DIV` and `REM` are forms of `div` and `rem`, protected against division by zero. Procedure `SLASH` is the real equivalent. Procedure `SUBSTR` provides a form of substring extraction that is protected against indices beyond either end of the string. Procedures `FLOOR` and `CEILING` were also added for the Tile problem. These are defined in the wrapper, near the end of §B.1 *Syntax*.

5.3 Problems and individual results

This section describes the problems and the results on an individual basis.

A range of problems was chosen to exercise different aspects of Gads 2.

Monkey
: Evolve a string literal to test that the basic system is working.

Cart
: The conventional cart-centering problem.

Tile
: Mixed mode arithmetic.

Multiplexer
: A standard example, involving boolean expressions and conditionals.

Power
: Variable declaration, assignment and iteration.

Two Box
: Procedure declaration and use.

Fact
: Recursive procedure declaration and use.

Annie
: Evolution of a main program.

Each of these problems is described in the following subsections. The description gives the aim of the problem, the objective of the problem, the subset of the S-algol rag that was used, how functionality was calculated, and the individual qualitative results.

5.3.1 Monkey

The aim of the monkey problem was to test the Gads 2 implementation.

The objective was to evolve the literal string `"Hello, world!'n"`.

The monkey problem was named after the famous quotation:

"If an army of monkeys were strumming on typewriters, they might write all the books in the British Museum."

—Sir Arthur Eddington, *The nature of the physical world*, 1928

The virtual monkeys were not given such a difficult task, but the task is not as simple as it appears. The kernel of the phenotype was an S-algol `literal string`, so that the effective grammar was reduced to the following subset of S-algol:

```
program: programmonkey.
programmonkey: preamblesymbol, phenotypemonkey,
postamblemonkeysymbol, endofprogram.
phenotypemonkey: phenotypemonkeybeginsymbol, literalstring,
phenotypemonkeyendsymbol.
endofprogram: spacesymbol, questionsymbol.
literalstring: spacesymbol, quotesymbol, quotesymbol; spacesymbol,
quotesymbol, chars, quotesymbol.
chars: character; character, chars.
character: ascii; special.
ascii: letter; digit; punctuation.
special: apostrophersymbol, specialfollow.
letter: letterlowerasymbol; letterlowerbsymbol; letterlowercsymbol;
letterlowerdsymbol; letterloweresymbol; letterlowerfsymbol;
letterlowergsymbol; letterlowerhsymbol; letterlowerisymbol;
letterlowerjsymbol; letterlowerksymbol; letterlowerlsymbol;
letterlowermsymbol; letterlowernsymbol; letterlowerosymbol;
letterlowerpsymbol; letterlowerqsymbol; letterlowerrsymbol;
letterlowerssymbol; letterlowertsymbol; letterlowerusymbol;
letterlowervsymbol; letterlowerwsymbol; letterlowerxsymbol;
letterlowerysymbol; letterlowerzsymbol; letterupperasymbol;
letterupperbsymbol; letteruppercsymbol; letterupperdsymbol;
letteruppersymbol; letterupperfsymbol; letteruppergsymbol;
letterupperhsymbol; letterupperisymbol; letterupperjsymbol;
letterupperksymbol; letterupperlsymbol; letteruppermsymbol;
letteruppernsymbol; letterupperosymbol; letterupperpsymbol;
letterupperqsymbol; letterupperrsymbol; letteruppersssymbol;
```

```
letteruppertsymbol; letterupperusymbol; letteruppervsymbol;
letterupperwsymbol; letterupperxsymbol; letterupperysymbol;
letterupperzsymbol.
digit: numeral0symbol; numeral1symbol; numeral2symbol;
numeral3symbol; numeral4symbol; numeral5symbol; numeral6symbol;
numeral7symbol; numeral8symbol; numeral9symbol.
punctuation: spacesymbol; exclamationsymbol; hashsymbol;
dollarsymbol; percentsymbol; ampersandsymbol; roundlsymbol;
roundrsymbol; asterisksymbol; plussymbol; commasymbol; hyphensymbol;
periodsymbol; slashsymbol; colonsymbol; semicolonsymbol;
anglelsymbol; equalssymbol; anglersymbol; questionsymbol; atsymbol;
squarelsymbol; backslashsymbol; squarersymbol; caretsymbol;
underscoresymbol; apostrophelsymbol; curlylsymbol; barsymbol;
curlyrsymbol; tildesymbol.
specialfollow: letterlowernsymbol; letterlowerpsymbol;
letterlowerosymbol; letterlowertsymbol; letterlowerbsymbol;
apostrophersymbol; quotesymbol.
```

Monkey thus comes down to evolving `chars`. For each character, the genome must first choose between `ascii` and `special`. If `ascii` is chosen, the genome must then choose `letter`, `digit` or `punctuation`, and then the individual character. Similarly, a `special` character involves several choices. The ideal genome must produce a string of 15 characters in several categories. This is not an efficient way to discover strings of characters, but it is not intended as an exercise in efficiency.

Functionality was computed as follows. The observed value was the literal string. The expected value was the target string, `"Hello, world!'n"`. Raw functionality was the RMS of the difference between ascii values of observed characters and expected characters, with the shorter of the observed and expected strings extended with null characters to the same length as the longer string. For example, suppose the target string was `"Hello"` and the observed string was `"Hdlpij"`:

Observed:	H	d	l	p	i	j
Ascii:	72	100	108	112	105	106
Expected:	H	e	l	l	o	NUL
Ascii:	72	101	108	108	111	0
Differences:	0	1	0	4	6	106
Squares:	0	1	0	16	36	11236

```
Sum of squares:        1 + 16 + 36 + 11236
                       = 11289
Mean square:           11289/6
                       = 1881.5
Root mean square:      sqrt (1881.5)
                       = 43.4
```

The least possible value of raw functionality is 0. This is achieved when the observed and expected strings are equal. The maximum value is S-algol's *maxreal*, but this value is extremely unlikely because the probability of a long string is vanishingly small due to the definition of `chars`. Consequently the maximum value of the raw functionality was capped at a value equal to the raw functionality of the empty string.

Adjusted functionality was computed from the raw functionality first by reversing the scale (so that 0.0 represented the worst, not the best), and then by scaling it into the range [0, 9999].

A typical run (res_pat_0/log.atholl.0)[1] produced the following series of best-of-run phenotypes, starting in generation 0:

```
"''q[A836L'b8"
"'t;4E''''p[='p'"0"
"2;4E''''p[='p'"0"
"`{4{r't'"'t>cn"
"&v4E''''p[878'o'b'p'""
"''Q4E1't'"E>cn"
"''q[AF't6mp]'o"
"6{2{^'''p[878'o'b'p'""
"'o}mi8'o'p:eJ`7"
"''>gD7'''pmpJ`7"
"7Qoh''7'pmpJ`7"
"7Qgh{'''pJ_m|0"
"<tgh{7'p[eJS0'p"
"0Xmo]'''pJp]`7"
"8Qo}`3'pvp]|7"
"1}m}`3'pvp]|_"
"1{m}`3'pvp]|_"
"7Qgh`3'pvp]|_"
"7Qge{'''p[pc`c2"
"2^[{]'''pmpc`c2"
"1sgh{36mpc`c2"
"7tgh{'''pmpc`c2"
"7_gh{1'pmp]kc2"
"N_gh{1'pmp]kc2"
"Ntoh`''1vpc`c2"
"7hge{'''pmpmkc2"
"8Xge{7''|py`c2"
"7fgh{'''pxpmkc4"
"7fgh{'''pxpmkc2"
"N_gh{''1mpm`c2"
"N_ohv'''pvqckc2"
"5^mh{7''|pmjc2"
"Tfgh{'''pxpmkc2"
"N_ohv'''pvqmkc4"
"N_ghv'''pvqmkc2"
"7bgh{''''xpmkc2"
"N_gb{''''|pmkc2"
"N_ph{''''xpmkc2"
"N_ohv''''xpmkc2"
```

[1] The run log is *log.athol.0* and can be found in the results directory *res_pat_0*. The results directory identifies the parameters as run 0 of the client "atholl" with Paterson-style with no parsimony.

```
"N_gb{''''|pmk_#"
"N_ghv7''|pmj_#"
"Ndoh{''''xpmkc)"
"N_oh{''''|pmkc#"
"Ndmh{''''xpmkc)"
"Ndmhv''''|pmkc)"
"N_ohv''''xpmkc#"
"N_mhv''''xpmkc#"
"N_ohv'''"xpmkc#"
"Ndmhv'''"xpmkc#"
"E^ohv''''xpmk_#'t"
"Ndmhv''''xpmk_#'t"
"Nbohv'''"vpmk_#'t"
"Ndmhv'''"xpmk_#'t"
"Ndmhl''''xpmkc#'t"
"Ndmhv'''"xpmkc#'b"
"Gdghm'''"vpmk_#'t"
"Gdghm'''"xppkc#'t"
"Idmom'''"vppkc#'t"
"Gdmom'''"xprkc#'t"
"Gdmmm'''"vppkc#'t"
"Gdmkm'''"xprkc#'t"
"Idmkn'''"xprkc#'t"
"Gdmkn'''"xprkd#'t"
"Gdmkn'''"xprkd#'n"
"Idmln'''"xprld#'t"
"Gdmkn'''"xprld!'t"
"Gdmln'''"xprld!'t"
"Gdmln'''"xorld!'t"
"Gdmlo'''"xprld!'n"
"Gemln'''"xorld!'n"
"Gdmlo'''"world!'n"
"Gemlo'''"world!'n"
"Hemlo'''"world!'n"
"Hello'''"world!'n"
```

The length of the phenotype is always close to the target length of 15 characters. This aspect of the functionality is evidently quite effective.

The phenotype string does not appear remotely similar to the target until very near the end. For example, it is not at all obvious that moving from "`Tfgh{'''pxpmkc2`" to "`Ndmhv''''|pmkc)`" represents an improvement. This is presumably because the improvement is in terms of ascii values, which are not immediately apparent to human vision. The distance from `H` to `T` is greater than the distance from `H` to `N`, so we know that the first letter has improved, even though there is no subjective impression of it. I mention this because to begin with, I thought the system was not actually working at all.

The target is not actually reached, because the run ends in a local minimum. Between `Hello` and `world` the expected value is `commasymbol, spacesymbol`. Both of these are in the `punctuation` category. The observed value is a pair of characters in the `specialfollow` category. (In S-algol, single quote acts as an escape character.) Thus, to change either of these characters requires two gene mutations: one for category and one for the character. It is most unlikely that both gene changes will occur during the production of one individual, so the changes must occur one at a time or not at all. But doing either change results in a worse functionality. Given the unfortunate choice of `specialfollow` characters in the first place; the system has actually done the best it can:

```
Observed:           '      "
Ascii:              39     34
Expected:           ,      SP
Ascii:              44     32
Differences:        5      2
Squares:            25     4


Sum of squares:     29
```

Mutating to choose `punctuation` rather than `specialfollow` would mean that a punctuation character would be selected; but not one so close to the expected value. Hence the system finds a less-than-perfect solution.

The Monkey experiment demonstrates that Gads 2 produces phenotypes according to the grammar, and demonstrates evolution in the GA system.

5.3.2 Cart

The aim of the cart problem was to revisit a standard problem.

The objective was to evolve a control expression for a bang-bang rocket cart as described in § 2 *Gads 1*. A human-designed solution to this problem is shown below:

```
procedure expected (real x, v -> real)
      -x - v*rabs(v)
```

The kernel of the phenotype was an S-algol `exp3 real`. To keep the phenotype language comparable to the original problem, several S-algol nonterminals were redefined. For example, *int, bool, string, conditional expression, procedure*, and so on were removed, leaving a reduced grammar as follows:

```
program: programcart.
programcart: preamblesymbol, phenotypecart, postamblecartsymbol,
endofprogram.
endofprogram: spacesymbol, questionsymbol.
phenotypecart: phenotypecartbeginsymbol, exp3real,
phenotypecartendsymbol.
clausereal: exp0real.
```

```
clausel: spacesymbol, roundlsymbol.
clauser: spacesymbol, roundrsymbol.
exp0real: exp1real.
clauseseparator: spacesymbol, commasymbol.
exp1real: exp2real.
exp2real: exp3real.
exp3real: exp4real; exp4real, addop, exp4real.
addop: spacesymbol, plussymbol, newlinesymbol; spacesymbol,
hyphensymbol, newlinesymbol.
exp4real: exp5real; procslashsymbol, clausel, exp5real,
clauseseparator, exp6real, clauser; exp5real, multopreal, exp6real.
exp5real: exp6real.
exp6real: literalreal; applreal; idrealc; clausel, clausereal,
clauser.
multopreal: spacesymbol, asterisksymbol.
literalreal: spacesymbol, roundlsymbol, hyphensymbol,
numeral1symbol, roundrsymbol.
applreal: applrabs.
idrealc: spacesymbol, letterlowerxsymbol; spacesymbol,
letterlowervsymbol.
applrabs: procrabssymbol, clausel, exp0real, clauser.
```

Functionality was computed as follows. A set of 20 test cases was generated. Each test case was a starting position *x* and velocity *v* for the cart, both uniformly distributed in the range (-0.75, +0.75). The same cases were used throughout. The observed value was the simulated time taken to centre the cart using the evolved control expression. The expected value was the simulated time taken using the human-designed control expression. Raw functionality was the RMS of the difference between observed time and expected time, over the sample of test cases.

The least possible value of raw functionality is 0. This is achieved when the observed and expected times are equal. (It is worth mentioning that the optimal time of 2.02 seconds given in [Koza, 1992] is an artifact of the simulation parameters. The simulation ends when the cart is within a certain non-zero capture radius of the origin. An interaction of the capture radius and the time quantum mean that it is possible to center the cart in less than the theoretical minimum time.) The maximum value is the simulated time limit, 10 seconds.

Adjusted functionality was computed from the raw functionality first by reversing the scale (so that 0.0 represented the worst, not the best), and then by scaling it into the range [0, 9999].

A typical run (res_pat_0/log.atholl.0) produced the following best-of-run phenotype:

```
    - x + SLASH ( - v , rabs ( SLASH ( rabs ( SLASH ( - x , x ) )
    , v ) ) )
```

The Cart experiment demonstrates that Gads 2 can produce phenotypes comparable with those produced by Gads 1.

5.3.3 Tile 1, Tile 2

The aim of the tile problem was to demonstrate the evolution of an expression involving more than one type.

The objective was to evolve an integer-valued expression for the number of square unit tiles needed to cover a given right-angled triangle, with real-valued sides. The tiles cover the triangle from the right angle. For example, a triangle of sides 4.5 and 7.3 needs 23 tiles:

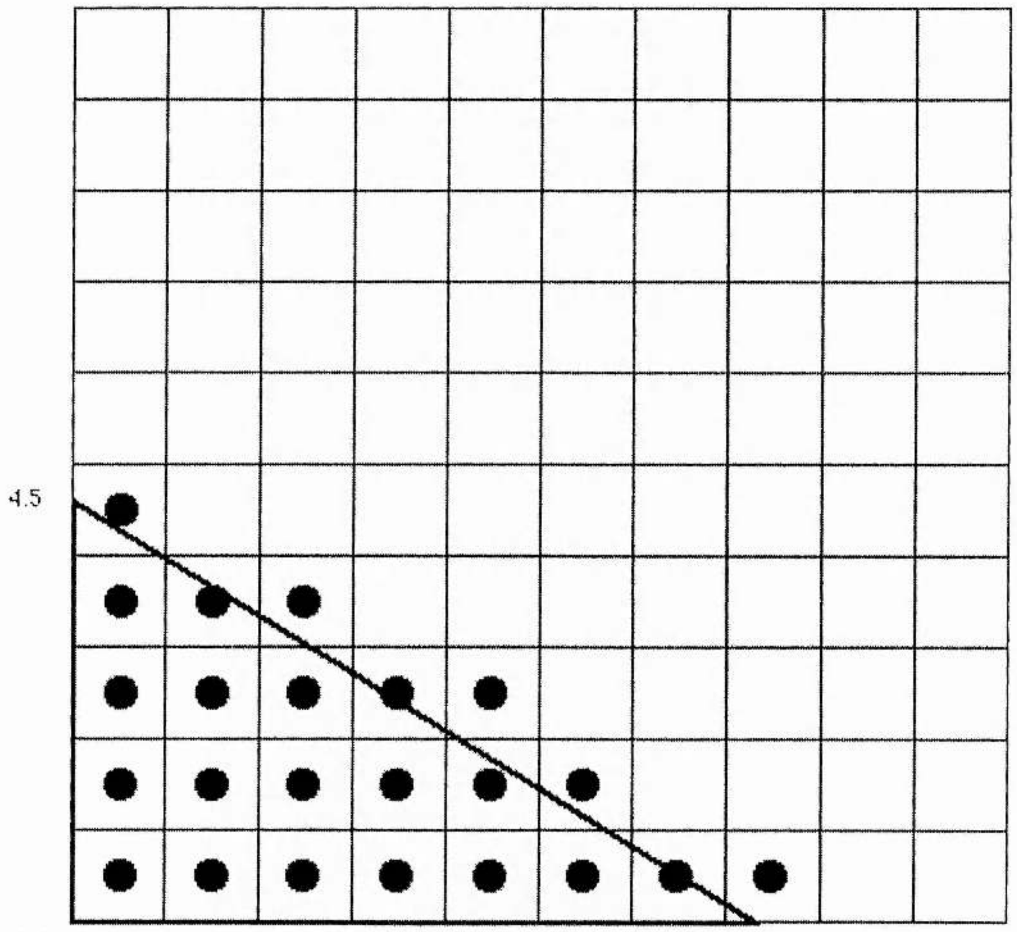

Figure 5-2: How many tiles?

As far as I know there is no closed expression for the number of tiles: it is necessary to compute the number by adding up the number of tiles in each column, which involves iteration. A human-designed procedure for this is as follows:

```
procedure expected (real x, y -> int)
{
        let t := 0
        for i = 0 to FLOOR(x) do {
                t := t + CEILING(y-i/x*y)
        }
        t
}
```

The kernel of the phenotype was an S-algol `exp0 int`. As usual, several S-algol nonterminals were redefined, to reduce the grammar to the relevant subset of S-algol. Also, procedures `FLOOR` and `CEILING` were introduced, because it seemed fairly likely they would be useful for this problem.

By mistake, version 1 of tile included `void clause` in the grammar. This is valid S-algol, but introduced the unintended capacity for the tile phenotype to write directly to standard output. Since the functionality of the phenotype was passed to the GA engine by executing the phenotype and reading its standard output, this meant that the phenotype could write its own functionality, independently of how well it computed the number of tiles. This mistake was fixed for version 2, which is shown below.

```
program: programtile.
programtile: preamblesymbol, phenotypetile, postambletilesymbol,
endofprogram.
endofprogram: spacesymbol, questionsymbol.
phenotypetile: phenotypetilebeginsymbol, exp0int,
phenotypetileendsymbol.
clauseint: exp0int.
clausereal: exp0real.
clausel: spacesymbol, roundlsymbol.
exp0int: exp1int.
clauser: spacesymbol, roundrsymbol.
exp0real: exp1real.
clauseseparator: spacesymbol, commasymbol.
exp1int: exp2int.
exp1real: exp2real.
exp2int: exp3int.
exp2real: exp3real.
exp3int: exp4int; exp4int, addop, exp4int.
exp3real: exp4real; exp4real, addop, exp4real; exp4int, addop,
exp4real; exp4real, addop, exp4int.
exp4int: exp5int; procdivsymbol, clausel, exp5int, clauseseparator,
exp6int, clauser; procremsymbol, clausel, ex
p5int, clauseseparator, exp6int, clauser; exp5int, multopint,
exp6int.
addop: spacesymbol, plussymbol, newlinesymbol; spacesymbol,
hyphensymbol, newlinesymbol.
```

```
exp4real: exp5real; procslashsymbol, clausel, exp5real,
clauseseparator, exp6real, clauser; exp5real, multopreal
, exp6real.
exp5int: exp6int.
exp6int: literalint; clausel, clauseint, clauser; applint.
multopint: spacesymbol, asterisksymbol.
exp5real: exp6real.
exp6real: literalreal; clausel, clausereal, clauser; applreal;
idrealc.
multopreal: spacesymbol, asterisksymbol.
literalint: spacesymbol, numeral0symbol; spacesymbol, digits.
applint: applabs; applfloor; applceiling.
literalreal: spacesymbol, numeral0symbol, periodsymbol,
numeral0symbol; literalint, periodsymbol; literalint, pe
riodsymbol, digits; literalint, periodsymbol, digits,
letterloweresymbol, digits; literalint, periodsymbol, digi
ts, letterloweresymbol, addop, digits.
applreal: applrabs; applsqrt.
idrealc: spacesymbol, letterlowerxsymbol; spacesymbol,
letterlowerysymbol.
digits: digit; digit, digits.
digit: numeral0symbol; numeral1symbol; numeral2symbol;
numeral3symbol; numeral4symbol; numeral5symbol; numeral6s
ymbol; numeral7symbol; numeral8symbol; numeral9symbol.
applabs: procabssymbol, clausel, exp0int, clauser.
applrabs: procrabssymbol, clausel, exp0real, clauser.
applsqrt: procsqrtsymbol, clausel, exp0real, clauser.
applceiling: procceilingsymbol, clausel, exp0real, clauser.
applfloor: procfloorsymbol, clausel, exp0real, clauser.
```

Functionality was computed as follows. A set of 20 test cases was generated. Each test case was a pair of real values *x* and *y* for a triangle, both uniformly distributed in the range (1, 100). The same cases were used throughout. The observed value was the number of tiles necessary according to the evolved expression. The expected value was the actual number of tiles necessary computed using the human-designed procedure. Raw functionality was the RMS of the difference between observed value and expected value, over the sample of test cases.

The least possible value of raw functionality is 0. This is achieved when the observed and expected number of tiles are equal. The maximum value is S-algol's *maxint*.

Adjusted functionality was computed from the raw functionality by taking logs, reversing the scale (so that 0.0 represented the worst, not the best), and then by scaling it into the range [0, 9999].

The phenotypes produced ranged from the short:

```
    1 + 8 * 212
```

to the long and impenetrable:

```
 ( 1508 +
 FLOOR ( CEILING ( SLASH ( -
 0.4e04 , y ) +
 +
 x ) * 9 -
 -
 x * 9.856 ) ) +
 ( DIV ( ( REM ( ( REM ( ( REM ( 9 , ( ( 0 * 4 ) ) ) +
 REM ( 7 , 4 ) ) , 18707 ) +
 DIV ( ( DIV ( 0 , ( DIV ( ( REM ( abs ( REM ( ( REM ( CEILING
( ( REM ( 0 , ( ( DIV ( 0 , ( DIV ( 0
 , ( 374 -
 DIV ( ( REM ( ( DIV ( FLOOR ( +
 257.4e +
8 +
 SLASH ( +
 ( -
 ( -
 0.0 +
 DIV ( ( FLOOR ( +
 sqrt ( SLASH ( ( SLASH ( 0.6 , rabs ( SLASH ( rabs ( DIV (
FLOOR ( +
 sqrt ( SLASH ( sqrt ( SLASH ( +
 y , sqrt ( SLASH ( -
 sqrt ( SLASH ( rabs ( ( SLASH ( rabs ( SLASH ( +
 sqrt ( SLASH ( y , ( ( REM ( ( 6 ) , 0 ) -
 DIV ( 57069 , 0 ) ) * 86 +
 -
 rabs ( FLOOR ( rabs ( y ) * rabs ( REM ( CEILING ( SLASH ( -
 0.0 , ( x * ( SLASH ( -
 x , rabs ( REM ( 7 , ( REM ( abs ( DIV ( ( REM ( FLOOR ( 27 *
4 -
 SLASH ( ( SLASH ( ( DIV ( ( DIV ( abs ( REM ( ( DIV ( 0 , ( (
REM ( 8 , 0 ) +
 abs ( DIV ( 9 , 0 ) ) * 0 ) ) ) ) , 0 ) ) , FLOOR ( DIV ( 0 ,
( 8 +
 DIV ( CEILING ( 713 -
 SLASH ( +
 rabs ( 0.0 ) , 0.0 ) ) , 0 ) ) ) +
 0.0 ) ) +
 0 ) , 0 ) +
 0.0 ) , 0.0 ) +
 0 ) , 0.0 ) ) , 0 ) +
 0 ) , 0 ) +
 0 ) , 0 ) ) ) +
 0.0 ) ) +
 0.0 ) +
 0.0 ) ) ) , 0 ) +
 0.0 ) +
 0.0 ) * 0 +
 0.0 ) * 0.0 ) ) +
 0 ) , 0.0 ) +
 0 ) , 0.0 ) +
 0.0 ) * 0.0 +
 0.0 ) , 0.0 ) +
 0.0 ) , 0.0 ) +
 0 ) ) +
 0.0 ) , 0.0 ) ) +
 0 ) , 0 ) +
 0.0 ) , 0.0 ) ) ) +
 0.0 ) , 0.0 ) +
 0.0 ) * 0.0 +
 0 ) ) , 0 ) ) +
0.0 ) , 0.0 ) ) , 0 ) ) , 0 ) +
 0 ) , 0 ) ) ) +
 0 ) ) +
 0 ) * 0 ) ) +
 0 ) +
 0.0 ) , 0 ) ) , 0 ) +
 0 ) , 0 ) ) , 0 ) ) ) +
 0 ) , 0 ) ) , 0 ) +
 0 ) , 0 ) +
 0 ) * 0
```

The long phenotype has a tail of zeroes. This indicates that the ontogenic mapping has used up all the genes in the genome. When this happens, and the ontogenic mapping process requests the next gene to choose between alternatives, it is given the value zero. This value results in the first alternative of the production being chosen. The first production is arranged to be the simplest; for example `exp0 int` is expanded via a series of productions to the literal 0. Thus, when the genes have all been used, a tail of default values results. This ensures that all expressions, parameter lists, etc are finished off in the simplest grammatically correct fashion.

The most interesting phenotype which resulted from this problem, with fitness 9258.0, (in version 1, res_koz_0, log.strathspey.0) was:

```
DIV ( CEILING ( DIV ( 0 , 0 ) +
 y * x ) , 2 ) -
 DIV ( FLOOR ( -
 y +
 SLASH ( x , 0.91e -
2 ) ) , 2 )
```

Simplifying `0.91e` to 1, and using the identities:

$$floor(-x) = -ceiling(x)$$
$$ceiling(-x) = -floor(x)$$

this reduces to:

```
DIV ( CEILING ( x * y ) , 2 ) +
DIV ( CEILING ( x + y ) , 2 )
```

In the example above, $(x, y) = (7.333, 4.611)$, giving an evolved value of 21 tiles, as compared with the true value of 23 tiles. The evolved expression can be seen to be composed of two subexpressions added together. The first approximates the area of the triangle, and the second adds a correction. In further tests, this expression was often exactly correct, never more than a few tiles wrong, and always wrong in the same direction (ie underestimating rather than overestimating).

The Tile experiment demonstrated that mixed mode expressions were satisfactorily evolved by Gads 2, and the discovery of interesting algorithms.

5.3.4 Multiplexer

The aim of the multiplexer problem was to demonstrate the evolution of boolean expressions involving *if* clauses and boolean operators.

The problem is described in section 7.4.1 of [Koza, 1992]. The objective was to evolve a 3-bit multiplexer. A human-designed expression for this is shown below:

```
procedure expected (bool a2, a1, a0,
      d7, d6, d5 ,d4 ,d3, d2, d1, d0 -> bool)
{
      if    a2
      then  if    a1
            then  if    a0
                  then  d7
                  else  d6
            else  if    a0
                  then  d5
                  else  d4
      else  if    a1
            then  if    a0
                  then  d3
                  else  d2
            else  if    a0
                  then  d1
                  else  d0
}
```

The kernel of the phenotype was an S-algol `exp0 bool`. As usual, several S-algol nonterminals were redefined, to reduce the grammar to the relevant subset of S-algol, which is shown below:

```
program: programmultiplexer.
programmultiplexer: preamblesymbol, phenotypemultiplexer,
postamblemultiplexersymbol, endofprogram.
endofprogram: spacesymbol, questionsymbol.
```

```
phenotypemultiplexer: phenotypemultiplexerbeginsymbol, exp0bool,
phenotypemultiplexerendsymbol.
clausebool: exp0bool; ifsymbol, clausebool, thensymbol, clausebool,
elsesymbol, clausebool.
clausel: spacesymbol, roundlsymbol.
clauser: spacesymbol, roundrsymbol.
exp0bool: exp1bool; exp1bool, orsymbol, newlinesymbol, exp1bool.
exp1bool: exp2bool; exp2bool, andsymbol, newlinesymbol, exp2bool.
exp2bool: exp3bool; notop, exp3bool; exp3bool, eqop, exp3bool;
notop, clausel, exp3bool, eqop, exp3bool, clauser.
exp3bool: exp4bool.
notop: spacesymbol, tildesymbol.
eqop: spacesymbol, equalssymbol; tildeequalssymbol.
exp4bool: exp5bool.
exp5bool: exp6bool.
exp6bool: literalbool; clausel, clausebool, clauser; idboolc.
literalbool: truesymbol; falsesymbol.
idboolc: spacesymbol, letterlowerasymbol, numeral2symbol;
spacesymbol, letterlowerasymbol, numeral1symbol; spacesymbol,
letterlowerasymbol, numeral0symbol; spacesymbol, letterlowerdsymbol,
numeral7symbol; spacesymbol, letterlowerdsymbol, numeral6symbol;
spacesymbol, letterlowerdsymbol, numeral5symbol; spacesymbol,
letterlowerdsymbol, numeral4symbol; spacesymbol, letterlowerdsymbol,
numeral3symbol; spacesymbol, letterlowerdsymbol, numeral2symbol;
spacesymbol, letterlowerdsymbol, numeral1symbol; spacesymbol,
letterlowerdsymbol, numeral0symbol.
```

Functionality was computed as follows. All 2048 combinations of the 11 input bits were generated. The observed value was the boolean value of the evolved expression. The expected value was the correct value computed using a perfect expression. Raw functionality was the count of the times where the observed and expected values were equal, over the sample of test cases.

The least possible value of raw functionality is 0. The maximum value is 2048.

Adjusted functionality was computed from the raw functionality by scaling it into the range [0, 9999].

A typical phenotype is as follows:

```
d7 and
 a1 = d7 or
 ~ a1 and
 ~ ( d5 ~= ( if if ~ ( true ~= ( ~ true and
 ( ~ d3 and
 ~ ( ~ ( if if ~ false or
 ~ d7 then if if ~ ( ( if ~ ( ~ ( d7 = ( if if ( if if if if
if if ~ ( ( false = ( ~ d4 or
 ( if ( a1 and
 ~ ( true = ( ( ~ ( ( ~ ( ~ ( if if if ( if ~ ( ( ~ ( ~ ( ( (
~ ( ( true ~= ( ( if if if if a0 ~= (
~ ( false ~= a1 ) or
 ~ ( ( ~ ( false ~= ( ~ false or
 false and
 ~ ( ( ~ true and
 ~ ( ( if if ~ true and
 ( if if ~ ( if if if ~ false and
 ~ ( ~ false ) then if ( ~ ( ( if if ~ ( ( if if if if if ( if
if if if ~ ( ~ d7 ) and
 ~ ( if ~ ( false = true ) and
 ( if if false and
 (.if ~ ( ( ~ a0 and
 ( ( true ~= true and
 ~ ( ( if if if if ~ false and
 ( ~ ( if ~ ( d2 = a2 ) and
 ~ ( ( ~ ( d0 ~= d3 ) ) ~= ( ~ true and
 ~ ( false ~= ( ~ ( ~ ( if if if if ~ ( ~ false and
 ~ ( if if if false = false and
 d0 = d4 or
 ~ ( ( if false ~= ( if true then true else true ) and
 true then true else true ) = true ) and
 true then true else true then true else true then true else
true ) or
 true ) or
 true then true else true then true else true then true else
true then true else true ) and
 true or
 true ) and
 true ) ) or
 true ) ) or
 true then true else true ) and
 true ) = true then true else true then true else true then
true else true then true else true ) = t
rue ) ) or
 true ) = true ) = true ) and
 true then true else true ) = true then true else true then
true else true ) = true then true else t
rue ) or
 true then true else true then true else true then true else
true then true else true ) = true and
 true or
 true then true else true then true else true then true else
true then true else true then true else
 true ) = true and
 true ) and
 true then true else true then true else true ) = true ) and
 true or
 true ) = true or
 true then true else true else true then true else true then
true else true ) and
 true then true else true then true else true ) = true or
 true then true else true then true else true ) = true ) ) =
true and
 true ) ) ) ) = true ) ) or
 true then true else true then true else true then true else
true then true else true ) or
 true ) or
 true ) = true ) and
 true ) = true and
true ) = true ) ) or
 true ) or
 true ) and
 true or
 true then true else true ) and
 true then true else true then true else true then true else
true ) ) ) = true ) or
 true ) = true ) ) ) = true then true else true ) and
 true ) ) and
 true or
 true ) or
 true then true else true then true else true then true else
true then true else true then true else
 true then true else true ) = true and
 true then true else true then true else true ) ) and
 true ) and
 true or
 true then true else true ) = true or
 true ) or
 true then true else true then true else true else true then
true else true ) or
 true ) ) or
 true ) ) and
 true or
 true then true else true then true else true ) )
```

Once again we can see the tail, this time composed of many occurrences of `true`, which is the default boolean expression.

The Multiplexer experiment demonstrates that Gads 2 successfully generates boolean expressions, involving *if* clauses and boolean operators.

5.3.5 Power

The aim of the power problem was to demonstrate the evolution of a program involving the declaration and use of typed variables and iteration.

The objective was to evolve the body of a procedure to raise a real value x to an integer power n. A human-designed solution to this is as follows:

```
procedure expected (real x; int n -> real)
{
      let result := 1.0
      for i = 1 to n do
            result := result * x
      result
}
```

The kernel of the phenotype was an artificial nonterminal defined to be a sequence of *void* clauses and declarations, followed by a *real* expression which was the value of the procedure. The human designed solution above requires only one declaration (of *result*), one void clause (the *for*) and a simple real expression (*result*). The phenotype kernel is therefore considerably more general than is required. As usual, several S-algol nonterminals were redefined, to reduce the grammar to the relevant subset of S-algol, which is shown below:

```
program: programpower.
programpower: preamblesymbol, phenotypepower, postamblepowersymbol,
endofprogram.
endofprogram: spacesymbol, questionsymbol.
phenotypepower: phenotypepowerbeginsymbol, sequencevoid,
sequenceseparator, clausereal, phenotypepowerendsymbol.
sequencevoid: clausevoid; clausevoid, sequenceseparator,
sequencevoid; decllet; decllet, sequenceseparator, sequencevoid.
clausevoid: exp0void; forsymbol, idintnew, spacesymbol,
equalssymbol, clauseint, tosymbol, clauseint, dosymbol, clausevoid;
idint, assignmentsymbol, clauseint; idreal, assignmentsymbol,
clausereal.
sequenceseparator: spacesymbol, semicolonsymbol, newlinesymbol.
decllet: declletint; declletreal.
clauseint: exp0int.
clausereal: exp0real.
clausel: spacesymbol, roundlsymbol.
exp0int: exp1int.
clauser: spacesymbol, roundrsymbol.
exp0real: exp1real.
declletint: letsymbol, idintnew, assignmentsymbol, clauseint.
declletreal: letsymbol, idrealnew, assignmentsymbol, clausereal.
idintnew: idintnewsymbol.
idrealnew: idrealnewsymbol.
exp0void: exp1void.
idint: spacesymbol, letterlowernsymbol.
idreal: spacesymbol, letterlowerxsymbol.
clauseseparator: spacesymbol, commasymbol.
exp1void: exp2void.
exp1int: exp2int.
exp1real: exp2real.
exp2void: exp3void.
exp2int: exp3int.
exp2real: exp3real.
exp3void: exp4void.
exp3int: exp4int; exp4int, addop, exp4int.
exp3real: exp4real; exp4real, addop, exp4real; exp4int, addop,
exp4real; exp4real, addop, exp4int.
exp4void: exp5void.
exp4int: exp5int; procdivsymbol, clausel, exp5int, clauseseparator,
exp6int, clauser; procremsymbol, clausel, exp5int, clauseseparator,
exp6int, clauser; exp5int, multopint, exp6int.
addop: spacesymbol, plussymbol, newlinesymbol; spacesymbol,
hyphensymbol, newlinesymbol.
exp4real: exp5real; procslashsymbol, clausel, exp5real,
clauseseparator, exp6real, clauser; exp5real, multopreal, exp6real.
exp5void: exp6void.
exp5int: exp6int; addop, exp6int.
exp6int: literalint; clausel, clauseint, clauser; idint.
multopint: spacesymbol, asterisksymbol.
exp5real: exp6real.
exp6real: literalreal; clausel, clausereal, clauser; idint; idreal.
```

```
multopreal: spacesymbol, asterisksymbol.
exp6void: sequencel, sequencer; clausel, clausevoid, clauser;
sequencel, sequencevoid, sequencer.
sequencel: spacesymbol, curlylsymbol.
sequencer: spacesymbol, curlyrsymbol.
literalint: spacesymbol, numeral0symbol; spacesymbol, digits.
literalreal: spacesymbol, numeral0symbol, periodsymbol,
numeral0symbol; literalint, periodsymbol; literalint, periodsymbol,
digits; literalint, periodsymbol, digits, letterloweresymbol,
digits; literalint, periodsymbol, digits, letterloweresymbol, addop,
digits.
digits: digit; digit, digits.
digit: numeral0symbol; numeral1symbol; numeral2symbol;
numeral3symbol; numeral4symbol; numeral5symbol; numeral6symbol;
numeral7symbol; numeral8symbol; numeral9symbol.
```

Functionality was computed as follows. A set of 30 test cases was generated. Each test case was a pair of real values *x* and *n*, both uniformly distributed, *x* in the range (0, 10) and *n* in the range [0, 10]. The same cases were used throughout. The observed value was the value of x^n computed according to the evolved expression. The expected value was computed using the human-designed solution above. Raw functionality was the RMS of the difference between observed value and expected value, over the sample of test cases.

The least possible value of raw functionality is 0. This is achieved when the observed and expected times are equal. The maximum value is *maxreal.*

Adjusted functionality was computed from the raw functionality by taking logs, reversing the scale (so that 0.0 represented the worst, not the best), and then by scaling it into the range [0, 9999].

A typical phenotype (res_pat_0/log.dufftown.0) is as follows:

```
x := 0.4e7 -  n * n ;
n := DIV ( + n , 0 ) + DIV ( n , ( REM ( 811 , n ) - REM ( n ,
0 ) ) ) ;
( SLASH ( n , n ) +  SLASH ( n , 0.76 ) ) * x + SLASH ( 6.557e
+ 0 , n )
```

As can be seen in this example, no variables are declared and no iteration takes place. The phenotype does not require either of these grammatical constructions, it merely enables them. Many phenotypes declare variables but never use them. For example (res_koz_0/log.atholl.0):

```
let real.0 := DIV ( - n , 0 ) - SLASH ( - x , 0.0 ) ;
SLASH ( n , ( SLASH ( x , 42. ) ) ) + + 7578471 * n
```

While (res_koz_0/log.kinclaith.0) shows that more than one declaration is possible:

```
let int.0 := DIV ( n , ( REM ( 1 , n ) ) ) ;
let real.1 := SLASH ( 09.0222 , 7.6 ) + ( x * 0.5719072e + 01
+ REM ( 0 , n ) ) ;
+ n + REM ( - ( DIV ( 0 , ( REM ( - n , n ) ) ) + ( n * ( n -
DIV ( + n , n ) ) ) ) , 0 )
```

No best-of-run phenotype declared more than two variables. Use of declared variables was rare. For example, (res_koz_0/log.coe.0) begins:

```
let real.0 := SLASH ( - ( SLASH ( ( SLASH ( - 0. , n ) ) , x )
) , n ) - n ;
real.0 := - n + - real.0 ;
```

showing that the identifier `real.0` was correctly added to the grammar following its declaration.

Evolution of *for* clauses was much rarer. For example (res_pat_0/log.ransom.0) shows both an evolved *for* clause and use of a previously declared variable:

```
let int.0 := DIV ( 0 , n ) ;
for int.1 = ( int.0 - DIV ( int.0 , n ) ) * int.0 to n * n do
int.0 := REM ( - 0 , int.0 ) ;
n * 0.7e7
```

The grammar for this problem would support a nested *for*, though none was produced as a best-of-run phenotype. None of the best-of-run phenotypes used any declared variable in their final *real* expression.

Although this experiment failed to produce anything remotely like the human-designed solution, it succeeded in its aim of demonstrating the declaration and use of type-correct variables and iteration.

5.3.6 Two box

The aim of the two box problem was to demonstrate the evolution of a procedure declaration and use. The problem is described in section 4 of [Koza, 1994]. Four variants of the problem were investigated. None are identical in all respects to the original Koza problems, but they do aim to capture the spirit of the original. The variants present a progression towards less prescriptive procedure declarations.

The objective in all variants is to evolve an expression for the difference in volume of two cuboids. The cuboid dimensions are given as two sets of three real values. A human-designed solution to this problem is as follows:

```
procedure expected
      (real L0, W0, H0, L1, W1, H1 -> real)
{
      procedure volume
                  (real ARG0, ARG1, ARG2 -> real)
            ARG0 * ARG1 * ARG2

      volume (L0, W0, H0) -
      volume (L1, W1, H1)
}
```

The variants of the problem are distinguished by their phenotype grammars as follows:

Two Box 1 (Koza style: without ADF)
A real expression involving only the *real* parameters `L0`, `W0`, `H0`, `L1`, `W1` and `H1` and *real* arithmetic operations +, –, * and `SLASH`.

Two Box 2 (Koza style: with ADF)
A non-recursive procedure of type (*real, real, real* -> *real*) and a *real* expression as for Two Box 1, plus the declared procedure.

Two Box 3 (Paterson style: with unprescribed, non-recursive ADF)
A non-recursive procedure of type (*real** -> *real*) (where *real** means zero or more *real*s) and a *real* expression as for Two Box 1, plus the declared procedure.

S-algol supports recursion by default, so for this variant it was necessary to implement a form of S-algol which did not support recursion. This is described in §4.2.2.7 *Procedure declarations.*

Two Box 4 (Paterson style: with unprescribed, recursive ADF)
As for Two Box 3, but with recursion enabled.

The Two Box 1 kernel was a `clause real`. As usual, several S-algol nonterminals were redefined, to reduce the grammar to the relevant subset of S-algol, which is shown below:

```
program: programtwobox1.
programtwobox1: preamblesymbol, phenotypetwobox1,
postambletwoboxsymbol, endofprogram.
endofprogram: spacesymbol, questionsymbol.
```

```
phenotypetwobox1: phenotypetwoboxbeginsymbol, clausereal,
phenotypetwoboxendsymbol.
clausereal: exp0real.
clausel: spacesymbol, roundlsymbol.
clauser: spacesymbol, roundrsymbol.
exp0real: exp1real.
clauseseparator: spacesymbol, commasymbol.
exp1real: exp2real.
exp2real: exp3real.
exp3real: exp4real; exp4real, addop, exp4real.
addop: spacesymbol, plussymbol, newlinesymbol; spacesymbol,
hyphensymbol, newlinesymbol.
exp4real: exp5real; procslashsymbol, clausel, exp5real,
clauseseparator, exp6real, clauser; exp5real, multopreal, exp6real.
exp5real: exp6real.
exp6real: idrealc; clausel, clausereal, clauser.
multopreal: spacesymbol, asterisksymbol.
idrealc: spacesymbol, letterupperlsymbol, numeral0symbol;
spacesymbol, letterupperwsymbol, numeral0symbol; spacesymbol,
letterupperhsymbol, numeral0symbol; spacesymbol, letterupperlsymbol,
numeral1symbol; spacesymbol, letterupperwsymbol, numeral1symbol;
spacesymbol, letterupperhsymbol, numeral1symbol.
```

The Two Box 2 kernel was a procedure declaration followed by a `clause real`. The procedure declaration was constrained to be of the specified type. The prescribed parameter list of 3 reals was obtained by "unwinding" the definition of `parameterlist` so that the procedure methods already developed for the general parameter list could be re-used without modification. The seed procedure `PROC.REAL` was included so that the ADF could be added as an alternative to an existing production, without having to introduce the whole production. As usual, several S-algol nonterminals were redefined, to reduce the grammar to the relevant subset of S-algol, which is shown below:

```
program: programtwobox2.
programtwobox2: preamblesymbol, phenotypetwobox2,
postambletwoboxsymbol, endofprogram.
endofprogram: spacesymbol, questionsymbol.
phenotypetwobox2: phenotypetwoboxbeginsymbol, sequencevoid,
phenotypetwoboxendsymbol.
sequencevoid: declproc, sequenceseparator, clausereal.
sequenceseparator: spacesymbol, semicolonsymbol, newlinesymbol.
declproc: declprocreal.
clausereal: exp0real.
clausel: spacesymbol, roundlsymbol.
clauser: spacesymbol, roundrsymbol.
applproctypereal: proctyperealsymbol, clausel, exp0real, clauser.
exp0real: exp1real.
declprocreal: procsymbol, idprocnew, roundlsymbol, parameterlist3,
arrowsymbol, typerealsymbol, roundrsymbol, sequenceseparator,
clausereal.
idrealnew: idrealnewsymbol.
idprocnew: idprocnewsymbol.
parameter: typerealsymbol, idrealnew.
```

```
parameterseparator: spacesymbol, semicolonsymbol.
idreal: spacesymbol, letterupperlsymbol, numeral0symbol;
spacesymbol, letterupperwsymbol, numeral0symbol; spacesymbol,
letterupperhsymbol, numeral0symbol; spacesymbol, letterupperlsymbol,
numeral1symbol; spacesymbol, letterupperwsymbol, numeral1symbol;
spacesymbol, letterupperhsymbol, numeral1symbol.
clauseseparator: spacesymbol, commasymbol.
exp1real: exp2real.
exp2real: exp3real.
exp3real: exp4real; exp4real, addop, exp4real.
addop: spacesymbol, plussymbol, newlinesymbol; spacesymbol,
hyphensymbol, newlinesymbol.
exp4real: exp5real; procslashsymbol, clausel, exp5real,
clauseseparator, exp6real, clauser; exp5real, multopreal, exp6real.
exp5real: exp6real.
exp6real: idreal; applreal; clausel, clausereal, clauser.
multopreal: spacesymbol, asterisksymbol.
applreal: applproctypereal.
parameterlist3: parameter, parameterseparator, parameterlist2.
parameterlist2: parameter, parameterseparator, parameterlist1.
parameterlist1: parameter.
```

The Two Box 3 kernel was, like the Two Box 2 kernel, a procedure declaration followed by a `clause real`. However, the procedure declaration was less constrained: the parameter list was zero or more real arguments. The seed procedure PROC.REAL was included so that the ADF could be added as an alternative to an existing production, without having to introduce the whole production. As usual, several S-algol nonterminals were redefined, to reduce the grammar to the relevant subset of S-algol, which is shown below:

```
program: programtwobox3.
programtwobox3: preamblesymbol, phenotypetwobox3,
postambletwoboxsymbol, endofprogram.
endofprogram: spacesymbol, questionsymbol.
phenotypetwobox3: phenotypetwoboxbeginsymbol, sequencevoid,
phenotypetwoboxendsymbol.
sequencevoid: declproc, sequenceseparator, clausereal.
sequenceseparator: spacesymbol, semicolonsymbol, newlinesymbol.
declproc: declprocreal.
clausereal: exp0real.
clausel: spacesymbol, roundlsymbol.
clauser: spacesymbol, roundrsymbol.
applproctypereal: proctyperealsymbol, clausel, exp0real, clauser.
exp0real: exp1real.
declprocreal: procsymbol, idprocnew, roundlsymbol, arrowsymbol,
typerealsymbol, roundrsymbol, sequenceseparator, clausereal;
procsymbol, idprocnew, roundlsymbol, parameterlist, arrowsymbol,
typerealsymbol, roundrsymbol, sequenceseparator, clausereal.
idrealnew: idrealnewsymbol.
idprocnew: idprocnewsymbol.
parameterlist: parameter; parameter, parameterseparator,
parameterlist.
parameter: typerealsymbol, idrealnew.
```

```
parameterseparator: spacesymbol, semicolonsymbol.
idreal: spacesymbol, letterupperlsymbol, numeral0symbol;
spacesymbol, letterupperwsymbol, numeral0symbol; spacesymbol,
letterupperhsymbol, numeral0symbol; spacesymbol, letterupperlsymbol,
numeral1symbol; spacesymbol, letterupperwsymbol, numeral1symbol;
spacesymbol, letterupperhsymbol, numeral1symbol.
clauseseparator: spacesymbol, commasymbol.
exp1real: exp2real.
exp2real: exp3real.
exp3real: exp4real; exp4real, addop, exp4real.
addop: spacesymbol, plussymbol, newlinesymbol; spacesymbol,
hyphensymbol, newlinesymbol.
exp4real: exp5real; procslashsymbol, clausel, exp5real,
clauseseparator, exp6real, clauser; exp5real, multopreal, exp6real.
exp5real: exp6real.
exp6real: idreal; applreal; clausel, clausereal, clauser.
multopreal: spacesymbol, asterisksymbol.
applreal: applproctypereal.
```

The Two Box 4 kernel (and consequently the context-free grammar) was identical to the Two Box 3 kernel. The difference, namely the fact that recursion was enabled in Two Box 4, was achieved by using a different production method.

Functionality was computed for all variants as follows. A set of 10 test cases was generated. Each test case was a set of 6 integer values `L0`, `W0`, `H0`, `L1`, `W1`, and `H1`, all uniformly distributed in the range [1, 10]. The same cases were used throughout. The observed value was the value of the evolved expression. The expected value was the correct difference in the cuboid volumes computed using the human-designed solution above. Raw functionality was the RMS of the difference between observed value and expected value, over the sample of test cases.

The least possible value of raw functionality is 0. This is achieved when the observed and expected times are equal. The maximum value is *maxreal*.

Adjusted functionality was computed from the raw functionality by taking logs, reversing the scale (so that 0.0 represented the worst, not the best), and then by scaling it into the range [0, 9999].

A typical Two Box 1 phenotype (res_koz_0/log.atholl.0) is as follows:

```
        ( ( ( H0 -
        SLASH ( W1 , W0 ) ) * W0 -
        SLASH ( ( SLASH ( ( H1 * ( L1 * W1 ) +
        ( SLASH ( ( W1 ) , ( SLASH ( ( SLASH ( L1 , W0 ) -
        SLASH ( H1 , W1 ) ) , W0 ) -
        L1 ) ) ) * W1 ) , ( SLASH ( H0 , H0 ) +
        SLASH ( H0 , ( H1 * ( SLASH ( ( ( H0 * W1 +
        L1 * H0 ) * L1 ) , ( L0 * ( SLASH ( ( SLASH ( ( W1 * L0 +
 H1 ) , ( H1 * ( SLASH ( W1 , W1 ) ) -
 SLASH ( W0 , ( L1 * W0 +
 ( ( L1 ) * ( SLASH ( W0 , ( ( ( SLASH ( L1 , L0 ) -
 SLASH ( ( SLASH ( ( ( SLASH ( ( SLASH ( ( SLASH ( ( ( ( (
SLASH ( L1 , ( SLASH ( ( W1 - SLASH ( W1 , L0 ) ) , W1 ) ) ) )
) * ( SLASH ( ( SLASH ( W0 , L0 ) -
 ( W0 * ( ( SLASH ( H1 , L1 ) +
 ( L0 * L1 +
 ( H1 ) ) ) * ( SLASH ( ( SLASH ( ( ( ( L0 * ( W0 * ( ( SLASH
( L1 , ( SLASH ( ( W1 * ( ( H0 * W0 ) -
 SLASH ( ( ( W0 ) +
 H0 ) , ( ( SLASH ( H0 , L1 ) ) * L1 ) ) ) ) , W1 ) +
 W0 ) ) ) -
 ( SLASH ( L0 , ( SLASH ( W1 , W0 ) ) ) ) ) ) -
 SLASH ( ( ( SLASH ( ( L1 +
 SLASH ( L1 , W0 ) ) , ( ( W1 ) -
 W0 ) ) +
 SLASH ( ( H1 * L0 ) , L0 ) ) +
 L0 ) , L0 ) ) * L0 ) ) , L0 ) +
 L0 ) , L0 ) ) ) ) ) , L0 ) +
 L0 ) +
 L0 ) * L0 ) , L0 ) +
 L0 ) , L0 ) ) , L0 ) ) ) , L0 ) +
 L0 ) , L0 ) ) * L0 ) ) ) ) ) * L0 ) ) ) ) ) , L0 ) +
 L0 ) +
 L0 ) ) ) ) ) ) ) +
 L0 ) , L0 ) ) * L0 +
 L0 ) +
 L0
```

The tail consists of L0 which is the default *real* expression in this grammar.

A typical Two Box 2 phenotype (res_koz_1/log.inchmurrin.0) is shown below.

```
procedure proc.0( real real.1 ; real real.2 ; real real.3 ->
real) ;
 H0 * real.2 ;
 ( SLASH ( ( proc.0( PROC.REAL ( PROC.REAL ( W0 * W0 ) *
PROC.REAL ( SLASH ( PROC.REAL ( L0 ) , PROC.REAL ( H1 ) ) ) -
 W0 ) * W1 , ( SLASH ( H0 , H1 ) ) * ( SLASH ( H0 , ( W1 *
PROC.REAL ( SLASH ( ( W1 -
 SLASH ( PROC.REAL ( SLASH ( proc.0( PROC.REAL ( SLASH (
PROC.REAL ( W0 * H0 ) , W1 ) ) , PROC.REAL ( W1 ) , PROC.REAL
( PROC.REAL ( PROC.REAL ( SLASH ( H1 , PROC.REAL ( W0 ) ) ) )
) * ( H1 +
 ( ( proc.0( SLASH ( proc.0( ( SLASH ( W0 , ( SLASH ( H0 , W1
) -
 ( SLASH ( H0 , W0 ) ) ) ) +
 W0 ) * proc.0( SLASH ( proc.0( SLASH ( L0 , H0 ) , W1 , L1 *
( W0 * proc.0( SLASH ( PROC.REAL ( W1 ) , ( ( PROC.REAL (
SLASH ( W0 , L0 ) ) ) * ( L1 * PROC.REAL ( SLASH ( PROC.REAL (
PROC.REAL ( ( PROC.REAL ( PROC.REAL ( H0 ) ) ) ) ) , ( ( SLASH
( ( ( PROC.REAL ( ( L1 ) ) +
 H0 ) * ( ( ( SLASH ( L1 , W1 ) ) +
 ( PROC.REAL ( W1 ) * W1 ) * H0 ) ) ) , ( H0 * ( PROC.REAL (
W1 ) ) +
 L0 ) ) ) +
 PROC.REAL ( SLASH ( proc.0( ( H0 * ( ( proc.0( SLASH ( ( H0 +
 W0 ) , PROC.REAL ( SLASH ( proc.0( H0 , PROC.REAL ( H1 ) ,
SLASH ( ( ( ( W0 ) ) ) , W0 ) +
 ( L0 )) , L0 ) +
 L0 ) ) , L0 , L0) +
 L0 ) ) +
 L0 ) , L0 , L0) , L0 ) ) * L0 ) ) ) +
 L0 ) ) ) , L0 , L0) )) , L0 ) , L0 , L0) +
 L0 , L0 , L0) , L0 ) , L0 , L0) * L0 ) ) )) , L0 ) +
 L0 ) , L0 ) ) , L0 ) ) ) ) +
 L0 ) , L0) ) , L0 ) ) * L0
```

The phenotype begins with the prototype for procedure `proc.0`, with the three *real* parameters, named `real.1`, `real.2` and `real.3`. The body refers to `H0` and `real.2`, showing that the external parameters and the procedure's own parameters are in scope. The phenotype then ends with a multi-line real expression involving the cuboid dimensions and the newly-defined procedure `proc.0`. Each call of the new procedure has the correct number of parameters. The procedure parameters are not in scope in the expression and so do not appear. The expression ends with a default tail.

Two Box 3 allows any number of *real*s in the procedure declaration. As expected from the grammar, the most common number of *real*s is zero, followed by 1, 2 and so on. The most observed was 6. A typical Two Box 3 phenotype (res_pat_1/log.brackla.0) with a procedure of type (-> *real*) is shown below:

```
procedure proc.0( -> real) ;
 W1 * ( SLASH ( H1 , ( W0 * W0 +
 W0 ) ) ) ;
 PROC.REAL ( SLASH ( H0 , PROC.REAL ( PROC.REAL ( proc.0 ) ) )
+
 SLASH ( PROC.REAL ( SLASH ( PROC.REAL ( SLASH ( W0 , proc.0 )
-
 ( W1 ) * ( PROC.REAL ( proc.0 * L1 -
 PROC.REAL ( SLASH ( ( proc.0 * ( SLASH ( ( SLASH ( ( SLASH (
PROC.REAL ( W0 * PROC.REAL ( ( SLASH ( ( SLASH ( L0 , proc.0 )
+
 SLASH ( proc.0 , proc.0 ) ) , proc.0 ) -
 H1 ) +
 proc.0 * W0 ) -
 proc.0 * L1 ) , proc.0 ) -
 W1 ) , PROC.REAL ( ( proc.0 +
 SLASH ( PROC.REAL ( ( SLASH ( PROC.REAL ( proc.0 +
 SLASH ( PROC.REAL ( L0 -
 PROC.REAL ( PROC.REAL ( SLASH ( proc.0 , W0 ) +
 H1 ) * PROC.REAL ( SLASH ( ( SLASH ( PROC.REAL ( ( SLASH (
proc.0 , ( ( ( SLASH ( PROC.REAL ( SLASH ( PROC.REAL ( ( L0 *
H1 -
 L0 ) * proc.0 ) , L1 ) ) , ( ( SLASH ( proc.0 , proc.0 ) ) +
 proc.0 ) ) +
 SLASH ( PROC.REAL ( SLASH ( PROC.REAL ( ( proc.0 ) * ( ( ( L0
) +
 PROC.REAL ( proc.0 ) * ( W0 * ( proc.0 -
 ( H0 +
 L1 * ( proc.0 +
 SLASH ( PROC.REAL ( H0 -
 ( PROC.REAL ( SLASH ( PROC.REAL ( L1 ) , ( ( SLASH (
PROC.REAL ( SLASH ( H0 , proc.0 ) -
 proc.0 ) , W0 ) ) * ( SLASH ( H0 , L1 ) ) +
 ( SLASH ( ( L0 * W1 ) , proc.0 ) ) ) ) ) ) ) , ( ( SLASH ( H1
, PROC.REAL ( SLASH ( W0 , proc.0 ) -
 PROC.REAL ( SLASH ( proc.0 , PROC.REAL ( SLASH ( ( SLASH (
proc.0 , H0 ) +
 proc.0 ) , W0 ) ) ) ) ) ) ) * ( ( L0 ) ) ) ) ) ) ) ) ) ) ) ,
L0 ) ) , L0 ) ) * L0 ) +
 L0 ) ) +
 L0 ) ) , L0 ) +
 L0 ) , L0 ) ) ) * L0 ) , L0 ) ) , L0 ) +
 L0 ) * L0 +
 L0 ) , L0 ) ) +
 L0 ) ) +
 L0 ) , L0 ) ) +
 L0 ) , L0 ) ) ) * L0 +
 L0 ) ) , L0 ) +
 L0 ) , L0 ) ) * L0 +
 L0
```

A typical Two Box 3 phenotype (res_pat_1/log.garioch.0) with a procedure of type (*real*, *real* -> *real*) is shown below:

```
procedure proc.0( real real.1 ; real real.2 -> real) ;
 SLASH ( real.1 , ( L1 ) ) ;
 proc.0( W0 * proc.0( PROC.REAL ( ( SLASH ( W1 , H0 ) +
 SLASH ( ( W0 ) , proc.0( SLASH ( PROC.REAL ( ( SLASH ( W1 ,
L0 ) +
 H1 ) ) , H0 ) , SLASH ( PROC.REAL ( W0 * proc.0( PROC.REAL (
PROC.REAL ( ( ( H0 +
 ( proc.0( PROC.REAL ( L0 ) , SLASH ( W1 , PROC.REAL ( SLASH (
W1 , ( PROC.REAL ( SLASH ( PROC.REAL ( ( W0 * PROC.REAL (
SLASH ( H0 , ( PROC.REAL ( H0 ) ) ) ) -
 W1 ) ) , PROC.REAL ( ( SLASH ( H0 , PROC.REAL ( PROC.REAL (
SLASH ( W0 , ( PROC.REAL ( PROC.REAL ( PROC.REAL ( L1 * L0 ) )
-
 W0 ) * L1 ) ) ) * PROC.REAL ( H0 +
 PROC.REAL ( W1 +
 PROC.REAL ( PROC.REAL ( ( L1 +
 W0 ) ) ) * PROC.REAL ( SLASH ( ( ( PROC.REAL ( H0 ) -
 W1 ) ) , ( proc.0( PROC.REAL ( PROC.REAL ( ( proc.0(
PROC.REAL ( PROC.REAL ( W0 ) * PROC.REAL ( SLASH ( L1 , ( H1 *
PROC.REAL ( H1 ) ) ) ) ) , ( ( SLASH ( W1 , PROC.REAL ( ( W1 *
W1 -
 SLASH ( ( W0 ) , H1 ) ) * PROC.REAL ( PROC.REAL ( ( L1 ) ) )
+
 ( H1 ) ) ) ) * ( ( L1 -
 SLASH ( PROC.REAL ( SLASH ( ( L1 ) , L1 ) ) , H0 ) ) ) )) *
PROC.REAL ( SLASH ( PROC.REAL ( proc.0( L1 -
 H1 , H1) ) , PROC.REAL ( PROC.REAL ( SLASH ( proc.0( proc.0(
W1 -
 W0 , L0) * L0 , L0) , L0 ) ) ) ) +
 L0 ) ) +
 L0 ) ) +
 L0 , L0) ) ) +
 L0 ) ) * L0 ) ) ) ) * L0 ) ) ) * L0 +
 L0 ) ) ) ) +
 L0) * L0 ) ) ) ) ) , L0) ) , L0 ) +
 L0) ) ) ) * L0 , L0) , L0)
```

The procedure's formal arguments are in scope only in the procedure body. The procedure itself is in scope in the final clause, where each call has two *real* arguments as required. There is a short tail.

The best individual found (res_pat_1/log.glenhaven.0) was:

```
procedure proc.0( -> real) ;
 L0 ;
 ( proc.0 * ( W0 * H0 ) -
 L1 * PROC.REAL ( W1 * H1 ) )
```

Given that **PROC.REAL** is an identity procedure, this can be seen to be equivalent to the correct value.

Two Box 4 allows the procedure to be recursive. However it was not until the experiment had been concluded that it was realised that without boolean expressions, any call of the recursive procedure, whether from inside itself or from the final real clause, must lead to endless recursion, resulting in a timeout and a functionality score of zero. Thus recursion, while syntactically sound, was a semantic trap. (The next problem, Fact, deals with recursion more fairly.) As a consequence examples of a Two Box 4 phenotype showing a recursive call of the procedure are rare, and they all have the prototype (-> *real*). For example (res_koz_0/log.chivas.0):

```
procedure proc.0( -> real) ;
 L1 +
 proc.0 ;
 ( L0 ) * PROC.REAL ( ( SLASH ( H0 , W1 ) +
 PROC.REAL ( PROC.REAL ( SLASH ( W0 , L1 ) ) ) * PROC.REAL (
PROC.REAL ( ( PROC.REAL ( SLASH ( PROC.REAL ( H0 +
 SLASH ( PROC.REAL ( SLASH ( PROC.REAL ( PROC.REAL ( PROC.REAL
( PROC.REAL ( H0 * ( W0 - SLASH ( H0 , PROC.REAL ( SLASH (
PROC.REAL ( SLASH ( L0 , ( SLASH ( L0 , W1 ) -
 SLASH ( PROC.REAL ( ( SLASH ( PROC.REAL ( SLASH ( PROC.REAL (
SLASH ( PROC.REAL ( PROC.REAL ( H1 ) * ( SLASH ( PROC.REAL (
SLASH ( W0 , H0 ) ) , PROC.REAL ( SLASH ( PROC.REAL ( SLASH (
PROC.REAL ( H1 ) , ( PROC.REAL ( SLASH ( L1 , PROC.REAL (
SLASH ( PROC.REAL ( PROC.REAL ( ( SLASH ( PROC.REAL ( ( W1 *
L1 ) * ( PROC.REAL ( PROC.REAL ( SLASH ( PROC.REAL ( PROC.REAL
( PROC.REAL ( SLASH ( ( SLASH ( PROC.REAL ( PROC.REAL ( SLASH
( PROC.REAL ( H0 -
 PROC.REAL ( SLASH ( PROC.REAL ( SLASH ( W1 , PROC.REAL (
PROC.REAL ( SLASH ( W0 , PROC.REAL ( SLASH ( PROC.REAL ( ( (
PROC.REAL ( SLASH ( ( ( SLASH ( H1 , ( PROC.REAL ( ( SLASH ( (
PROC.REAL ( H0 * ( SLASH ( ( SLASH ( ( W0 * PROC.REAL ( (
SLASH ( PROC.REAL ( ( SLASH ( PROC.REAL ( ( SLASH ( ( SLASH (
PROC.REAL ( SLASH ( H1 , W1 ) +
 SLASH ( PROC.REAL ( PROC.REAL ( SLASH ( PROC.REAL ( SLASH (
PROC.REAL ( SLASH ( L0 , PROC.REAL ( SLASH ( ( ( SLASH ( ( ..
PROC.REAL ( L1 +
 PROC.REAL ( L0 ) ) * L0 +
 L0 ) , L0 ) +
 L0 ) * L0 ) , L0 ) ) ) ) , L0 ) ) , L0 ) ) * L0 ) , L0 ) ) ,
L0 ) ) , L0 ) ) ) , L0 ) ) +
 L0 ) , L0 ) ) +
 L0 ) +
 L0 ) , L0 ) ) , L0 ) ) ) * L0 ) , L0 ) +
 L0 ) * L0 +
 L0 ) * L0 +
 L0 ) ) ) ) , L0 ) ) +
 L0 ) ) * L0 +
 L0 ) , L0 ) +
 L0 ) ) ) * L0 +
 L0 ) ) +
 L0 ) , L0 ) ) ) , L0 ) ) * L0 ) , L0 ) +
 L0 ) , L0 ) +
 L0 ) * L0 +
 L0 ) * L0 ) , L0 ) +
 L0 ) * L0 +
 L0 ) * L0 ) +
 L0 ) , L0 ) ) ) +
 L0 ) , L0 ) +
 L0 ) ) +
 L0 ) * L0 ) ) ) , L0 ) ) ) ) ) , L0 ) ) , L0 ) +
 L0 ) , L0 ) ) +
 L0 ) , L0 ) ) ) +
 L0 ) , L0 ) ) ) ) +
 L0 ) +
 L0 ) ) * L0 ) , L0 ) +
 L0 ) , L0 ) ) , L0 ) ) ) ) +
 L0 ) ) )
```

The procedure `proc.0` is endlessly recursive. To call it results in a zero functionality score, and indeed it is not called in the final `clause real`.

In all Two Box examples, the procedure body was typically one or two lines long, while the final clause was around twenty lines (not including the tail). Given that both of these are derived from `clause real`, this difference is surprising. A possible explanation — apart from error — is that the CFG components of the grammar are not equal, since in the second case, the defined procedure `proc.0` is available. In Two Box 4 the CFG components are equal, but the semantics render comparison impossible.

Two Box 1 essentially replicates the earlier experiments such as Cart.

Two Box 2 demonstrates that Gads 2 can evolve a procedure body for a given prototype and use it in an expression.

Two Box 3 demonstrates that Gads 2 can evolve a procedure prototype, a non-recursive body for it, and an expression which uses it.

Two Box 4 demonstrates that Gads 2 can evolve a procedure prototype, a recursive body for it, and an expression which uses it. The demonstration of recursion is not entirely satisfactory because recursion was (by accident) impossible to achieve. However, Gads 2 was able both to evolve recursive programs, and to learn that doing so was ineffective.

5.3.7 Fact

The aim of the Fact problem was to demonstrate the evolution and use of a recursive procedure.

The objective was to evolve a recursive factorial procedure. However, this simple objective required a less-than-obvious wrapper to avoid making it too specific. Real types were used to avoid integer overflow. The objective was couched as follows: to evolve a procedure *p* of type (*int -> real*); and then to evolve a real expression, possibly involving *p* and a given integer *n*; the value of the real expression to be equal to *factorial* (*n*). A human-designed solution is as follows:

```
procedure expected (int n -> real);
{
      procedure factorial (int n -> real)
            if n <= 0
                  then 1
                  else factorial(n-1)*n

      factorial (n)
}
```

Here, `factorial` is the evolved procedure, and `factorial (n)` is the evolved expression. As can be seen, the evolved program involves three types: *int*, *real* and *bool*.

The kernel of the phenotype was an artificial nonterminal defined to be a procedure declaration followed by a `clause real`. The procedure declaration was constrained to be of type (*int -> real*), and recursion was enabled. The procedure would therefore be in scope in its own body and in the following `clause real`. As usual, several S-algol nonterminals were redefined, to reduce the grammar to the relevant subset of S-algol, which is shown below:

```
program: programfact.
```

```
programfact: preamblesymbol, phenotypefact, postamblefactsymbol,
endofprogram.
endofprogram: spacesymbol, questionsymbol.
phenotypefact: phenotypefactbeginsymbol, sequencevoid,
phenotypefactendsymbol.
sequencevoid: declproc, sequenceseparator, clausereal.
sequenceseparator: spacesymbol, semicolonsymbol, newlinesymbol.
declproc: declprocreal.
clauseint: exp0int.
clausereal: exp0real; ifsymbol, clausebool, thensymbol, clausereal,
elsesymbol, clausereal.
sequencebool: clausebool; sequencevoid, sequenceseparator,
clausebool.
clausebool: exp0bool; ifsymbol, clausebool, thensymbol, clausebool,
elsesymbol, clausebool.
sequencestring: clausestring; sequencevoid, sequenceseparator,
clausestring.
clausestring: exp0string; ifsymbol, clausebool, thensymbol,
clausestring, elsesymbol, clausestring.
clausel: spacesymbol, roundlsymbol.
exp0int: exp1int.
clauser: spacesymbol, roundrsymbol.
applproctypereal: proctyperealsymbol, clausel, exp0real, clauser.
exp0real: exp1real.
applproctypebool: proctypeboolsymbol, clausel, exp0bool, clauser.
exp0bool: exp1bool; exp1bool, orsymbol, newlinesymbol, exp1bool.
applproctypestring: proctypestringsymbol, clausel, exp0string,
clauser.
exp0string: exp1string.
declprocreal: procsymbol, idprocnew, roundlsymbol, parameterlist,
arrowsymbol, typerealsymbol, roundrsymbol, sequenceseparator,
clausereal.
idintnew: idintnewsymbol.
idprocnew: idprocnewsymbol.
parameterlist: parameter.
parameter: typeintsymbol, idintnew.
idint: spacesymbol, letterlowernsymbol.
idreal: variablerealsymbol.
idbool: variableboolsymbol.
idstring: variablestringsymbol.
clauseseparator: spacesymbol, commasymbol.
exp1int: exp2int.
exp1real: exp2real.
exp1bool: exp2bool; exp2bool, andsymbol, newlinesymbol, exp2bool.
exp1string: exp2string.
exp2int: exp3int.
exp2real: exp3real.
exp2bool: exp3bool; notop, exp3bool; exp3int, eqop, exp3int;
exp3real, eqop, exp3real; exp3int, comparop, exp3int; exp3real,
comparop, exp3real; notop, clausel, exp3int, eqop, exp3int, clauser;
notop, clausel, exp3real, eqop, exp3real, clauser; notop, clausel,
exp3int, comparop, exp3int, clauser; notop, clausel, exp3real,
comparop, exp3real, clauser.
exp2string: exp3string.
exp3int: exp4int; exp4int, addop, exp4int.
exp3real: exp4real; exp4real, addop, exp4real; exp4int, addop,
exp4real; exp4real, addop, exp4int.
exp3bool: exp4bool.
notop: spacesymbol, tildesymbol.
eqop: spacesymbol, equalssymbol; tildeequalssymbol.
```

```
exp3string: exp4string.
comparop: spacesymbol, anglelsymbol; anglelequalssymbol;
spacesymbol, anglersymbol; anglerequalssymbol.
exp4int: exp5int; procdivsymbol, clausel, exp5int, clauseseparator,
exp6int, clauser; procremsymbol, clausel, exp5int, clauseseparator,
exp6int, clauser; exp5int, multopint, exp6int.
addop: spacesymbol, plussymbol, newlinesymbol; spacesymbol,
hyphensymbol, newlinesymbol.
exp4real: exp5real; procslashsymbol, clausel, exp5real,
clauseseparator, exp6real, clauser; exp5real, multopreal, exp6real.
exp4bool: exp5bool.
exp4string: exp5string; exp5string, multopstring, exp5string.
exp5int: exp6int; addop, exp6int.
exp6int: literalint; clausel, clauseint, clauser; idint.
multopint: spacesymbol, asterisksymbol.
exp5real: exp6real.
exp6real: idint; idreal; applreal; clausel, clausereal, clauser.
multopreal: spacesymbol, asterisksymbol.
exp5bool: exp6bool.
exp5string: exp6string.
multopstring: concatsymbol.
exp6bool: literalbool; clausel, clausebool, clauser; sequencel,
sequencebool, sequencer; applbool; idbool.
exp6string: literalstring; clausel, clausestring, clauser;
sequencel, sequencestring, sequencer; procsubstrsymbol, clausel,
exp0string, clauseseparator, exp0int, clauseseparator, exp0int,
clauser; applstring; idstring.
sequencel: spacesymbol, curlylsymbol.
sequencer: spacesymbol, curlyrsymbol.
literalint: spacesymbol, numeral0symbol; spacesymbol, digits.
applreal: applproctypereal.
literalbool: truesymbol; falsesymbol.
applbool: applproctypebool; appldigit; applletter.
literalstring: spacesymbol, quotesymbol, quotesymbol; spacesymbol,
quotesymbol, chars, quotesymbol.
applstring: applproctypestring; applcode; appliformat.
digits: digit; digit, digits.
chars: character; character, chars.
character: ascii; special.
ascii: letter; digit; punctuation.
special: apostrophersymbol, specialfollow.
letter: letterlowerasymbol; letterlowerbsymbol; letterlowercsymbol;
letterlowerdsymbol; letterloweresymbol; letterlowerfsymbol;
letterlowergsymbol; letterlowerhsymbol; letterlowerisymbol;
letterlowerjsymbol; letterlowerksymbol; letterlowerlsymbol;
letterlowermsymbol; letterlowernsymbol; letterlowerosymbol;
letterlowerpsymbol; letterlowerqsymbol; letterlowerrsymbol;
letterlowerssymbol; letterlowertsymbol; letterlowerusymbol;
letterlowervsymbol; letterlowerwsymbol; letterlowerxsymbol;
letterlowerysymbol; letterlowerzsymbol; letterupperasymbol;
letterupperbsymbol; letteruppercsymbol; letterupperdsymbol;
letteruppereSymbol; letterupperfsymbol; letteruppergsymbol;
letterupperhsymbol; letterupperisymbol; letterupperjsymbol;
letterupperksymbol; letterupperlsymbol; letteruppermsymbol;
letteruppernsymbol; letterupperosymbol; letterupperpsymbol;
letterupperqsymbol; letterupperrsymbol; letteruppersSymbol;
letteruppertsymbol; letterupperusymbol; letteruppervsymbol;
letterupperwsymbol; letterupperxsymbol; letterupperysymbol;
letterupperzsymbol.
```

```
digit: numeral0symbol; numeral1symbol; numeral2symbol;
numeral3symbol; numeral4symbol; numeral5symbol; numeral6symbol;
numeral7symbol; numeral8symbol; numeral9symbol.
punctuation: spacesymbol; exclamationsymbol; hashsymbol;
dollarsymbol; percentsymbol; ampersandsymbol; roundlsymbol;
roundrsymbol; asterisksymbol; plussymbol; commasymbol; hyphensymbol;
periodsymbol; slashsymbol; colonsymbol; semicolonsymbol;
anglelsymbol; equalssymbol; anglersymbol; questionsymbol; atsymbol;
squarelsymbol; backslashsymbol; squarersymbol; caretsymbol;
underscoresymbol; apostrophelsymbol; curlylsymbol; barsymbol;
curlyrsymbol; tildesymbol.
specialfollow: letterlowernsymbol; letterlowerpsymbol;
letterlowerosymbol; letterlowertsymbol; letterlowerbsymbol;
apostrophersymbol; quotesymbol.
appldigit: procdigitsymbol, clausel, exp0string, clauser.
applletter: proclettersymbol, clausel, exp0string, clauser.
applcode: proccodesymbol, clausel, exp0int, clauser.
appliformat: prociformatsymbol, clausel, exp0int, clauser.
```

Inspection of the above grammar reveals that it includes support for the *string* type. This was unintentional; but has been left in for future comparison. Modifying grammars is not simple: it is easy to include unwanted features and hard to find out where the leak is. In the above grammar, the leak is `appl bool`. If `exp6 bool` is redefined without `appl bool` on its RHS, the grammar supports boolean expressions but not boolean procedures.

Functionality was computed as follows. A set of 10 test cases was generated, evenly distributed over the range [1, 95] which is the largest range that could not cause overflow in the evaluation process. The observed value was the logarithm of the value returned by the evolved expression. The expected value was the logarithm of the value returned by the the human-designed solution above. Raw functionality was the RMS of the difference between observed value and expected value, over the sample of test cases.

The least possible value of raw functionality is 0. This is achieved when the observed and expected times are equal. The maximum value is *log maxreal.*

Adjusted functionality was computed from the raw functionality by reversing the scale (so that 0.0 represented the worst, not the best), and then by scaling it into the range [0, 9999].

None of the best-of-run phenotypes produced a conditional expression, though it is not obvious why they were selected out. The example phenotype below is not best-of-anything, but it shows the features of interest:

```
procedure observed (int n -> real)
(
!<<<<< phenotype begins
 procedure proc.0( int int.1 -> real) ;
 if if ~ ( PROC.REAL ( proc.0( DIV ( +
 ( DIV ( -
 6 , ( REM ( -
 ( -
 ( REM ( n , ( REM ( ( REM ( +
 2 , ( REM ( n , ( DIV ( +
 0 , ( REM ( -
 n , ( -
 int.1 * ( DIV ( ( REM ( +
 n , ( REM ( ( REM ( +
 ( ( ( 6 +
 n ) * ( ( REM ( -
 int.1 , int.1 ) -
 REM ( +
 ( DIV ( ( +
 int.1 * n +
 REM ( int.1 , 47 ) ) , int.1 ) +
 DIV ( 73 , n ) ) , n ) ) ) ) ) , ( REM ( +
 ( REM ( 65 , ( +
 4 * ( REM ( int.1 , ( REM ( +
 ( REM ( +
 n , 0 ) ) , ( DIV ( +
 n , 0 ) -
 0 ) ) ) ) +
 REM ( ( REM ( +
 ( REM ( int.1 , ( DIV ( ( ( REM ( ( DIV ( ( int.1 ) , n ) -
 +
 ( -
 ( REM ( -
 0 , ( REM ( +
 n , ( 0 * int.1 -
 -
 n * ( REM ( n , ( DIV ( ( n * 3 -
 DIV ( ( REM ( +
 n , ( REM ( -
 ( REM ( int.1 , ( ( +
 ( DIV ( int.1 , 7 ) +
 DIV ( -
 ( -
 52 ) , 0 ) ) * ( DIV ( +
 ( REM ( -
 ( REM ( +
 ( -
 ( REM ( -
 8 , ( ( DIV ( ( REM ( -
 n , ( ( -
 4 * int.1 ) ) ) -
 0 ) , 252 ) ) ) ) +
 REM ( ( REM ( -
 n , 1013 ) +
 REM ( n , 0 ) ) , n ) ) -
+
 ( ( REM ( -
 1 , n ) -
 REM ( int.1 , ( +
 ( DIV ( 0 , 0 ) +
 0 ) ) ) ) * 0 ) ) , 0 ) +
 0 ) , 0 ) ) , 0 ) +
0 ) +
0 ) * 0 ) ) ) , 0 ) +
0 ) ) +
0 ) , 0 ) ) , 0 ) +
0 ) ) +
0 ) ) ) ) ) +
0 ) +
0 ) ) , 0 ) ) * 0 +
0 ) , 0 ) ) ) +
0 ) , 0 ) +
0 ) , 0 ) ) +
0 ) ) ) , 0 ) +
0 ) ) +
0 ) , 0 ) ) ) ) , 0 ) ) ) ) +
0 ) ) +
0 ) ) ) ) ) , 0 ) +
0 ) ) +
0 ) +
0 ) , 0 ) ) ) ) , 0 ) +
0) +
n ) +
n = n ) and
true or
true then true else true then n else n ;
n
!>>>>> phenotype ends
}
```

The first 4 and last 2 lines are in italics because they are prescribed; all else is evolved. The body of procedure `proc.0` extends to the semicolon 4 lines from the end. The next line is the `clause real` that is returned as the value of procedure `observed`.

The body of `proc.0` contains boolean expressions and uses the parameters `n` and `int.1` as it should. It is also recursive.

This experiment demonstrates that Gads 2 can evolve a recursive procedure.

5.3.8 Annie

The aim of the Annie problem was to demonstrate the evolution of a main program with a full-sized context-sensitive grammar, that is, that Gads 2 successfully deals with the scalability problems of Gads 1.

In order for the main program to fit into the same scheme as the other experiments, it was necessary that it had no wrapper. In effect it was its own wrapper, and had the power to write its own functionality. The maximum functionality was 9999, so a human-designed solution is as follows:

```
write 9999, "'n"
```

This is about the simplest program imaginable. However, given the size of the search space, it is not a trivial problem to solve. The kernel of the phenotype was `sequence void`. No S-algol nonterminals were redefined. The full S-algol grammar as shown in §B was used, comprising 129 nonterminals and 165 terminals.

The Annie problem was named after the character who sings:

"Anything you can do, I can do better."

—Irving Berlin, *Annie Get Your Gun*, 1946

Functionality was computed as follows. Whatever output the program produced was read. If it could be interpreted as a real value using standard input procedures, that was the functionality. A value less than 0 or greater than 9999 was set to zero, so that it was not simply a matter of outputting *maxreal.* Annie has to get as close to 9999 as possible — but no more.

As might be expected the range of solutions was large. The most parsimonious and highest functionality score was 9999.876, which was achieved in three runs. This corresponds exactly to the human-designed individual above. A more typical phenotype (res_pat_0/log.atholl.0) is shown below:

```
let int.0 := REM ( length ( "" ++ PROC.STRING ( code ( ( REM (
-
 maxint , INT ) -
 length ( ( "52'o" ++ STRING ) ) ) ) ++ STRING ) ) , ( DIV ( +
 { DECODE ( "'p9" ) * maxint } , { write 9998. -
 -
 length ( PROC.STRING ( code ( 0 -
 REM ( -
 DECODE ( { { ( iformat ( maxint * s.w -
 0 * 2 ) ) } ++ ( { { STRING ++ { ( code ( DIV ( 7 , s.w ) ) )
} } } ++ ( { let real.1 := DIV ( -
 maxint , r.w ) -
 REAL ;
 STRING ++ PROC.STRING ( ( iformat ( +
 0 ) ++ STRING ) ++ { PROC.STRING ( { let real.2 := REAL +
 maxint * sqrt ( -
 s.w +
 r.w * pi ) ;
 PROC.STRING ( ( "''65" ) ++ { ":'n'"]'py" ++ ( STRING ++ "'o"
) } ) } ++ "'n8}" ) } ) } ++ code ( DIV ( +
 s.w , 04 ) ) ) ) } ) , INT ) ) ) ++ { "" } ) ,"'n" ;
 procedure proc.3( real real.4 -> int) ;
 REM ( i.w , 0 ) ;
DIV ( -
maxint , r.w ) } ) ) )
```

The write clause begins on the 4th line and ends with the string " 'n" on the 5th-from-last line. It's obvious that the value 9998 has been evolved because it is valuable. What the rest of the write statement does is not obvious, and is probably not worth examining in detail.

This experiment demonstrates that Gads 2 can evolve main programs in a fully-featured context sensitive language.

5.4 Comparative results

This section examines the results on a comparative basis.

The main observations of each run were the fitness of the best-of-run and the number of evaluations needed to reach it. Each run was characterized by a configuration and a random seed which was different for each run. The configuration was partitioned into problem parameters and engine parameters. Problem parameters were those which directly affected the solution phenotype; the engine parameters were those which affected the performance of the evolutionary process. On this basis, the grammar, which includes the evaluation function and the wrapper, is the embodiment of the problem parameters, while the GA system and the evolutionary parameters, such as population size, are the engine.

The following tables give the main observations.

B-N	annie	cart	fact	monkey	multi-plexer	power	tile1	tile2	twobox	twobox	twobox	twobox
koz_0	33	32	33	33	33	33	33	33	33	33	33	33
koz_1	33	33	33	33	33	33	32	33	33	33	33	33
pat_0	30	33	33	30	33	33	33	33	33	33	33	33
pat_1	33	32	33	31	33	33	33	33	33	33	33	33

Table 5-2: B-N: Sample size of benefit samples

B-MN	annie	cart	fact	monkey	multi-plexer	power	tile1	tile2	twobox	twobox	twobox	twobox
koz_0	9923.97	9999	8330	9753	6586.636	9741.758	7231.818	7083.697	9951.576	9935	9940.424	9942.212
koz_1	9661.062	9999.836	8373.995	9768.431	6416.986	9741.596	7094.389	7003.607	9950.212	9940.046	9945.447	9946.443
pat_0	9815.267	9999	8230.121	9868	6459.394	9740.818	7006.97	7129.788	9947.758	9935.061	9938.848	9943.333
pat_1	9970.032	9999.841	8210.695	9835.082	6290.305	9741.142	7020.398	7131.305	9951.795	9936.411	9945.874	9945.378

Table 5-3: B-MN: Mean of benefit samples

B-SD	annie	cart	fact	monkey	multi-plexer	power	tile1	tile2	twobox	twobox	twobox..	twobox
koz_0	349.205	0	326.381	145.542	395.883	3.553	469.365	499.661	12.191	4.969	7.71	11.858
koz_1	1706.174	0.012	296.792	144.929	418.472	3.807	583.392	493.339	14.43	5.297	11.299	14.985
pat_0	974.369	0	217.999	106.504	285.05	3.737	375.457	434.528	6.722	4.962	6.548	11.277
pat_1	140.959	0.011	214.833	112.008	281.655	2.474	408.65	604.171	16.82	4.368	11.028	12.039

Table 5-4: B-SD: Standard deviation of benefit samples

B-V	annie	cart	fact	monkey	multi-plexer	power	tile1	tile2	twobox	twobox	twobox	twobox
koz_0	0.035	0	0.039	0.015	0.06	0	0.065	0.071	0.001	0.001	0.001	0.001
koz_1	0.177	0	0.035	0.015	0.065	0	0.082	0.07	0.001	0.001	0.001	0.002
pat_0	0.099	0	0.026	0.011	0.044	0	0.054	0.061	0.001	0	0.001	0.001
pat_1	0.014	0	0.026	0.011	0.045	0	0.058	0.085	0.002	0	0.001	0.001

Table 5-5: B-V: Coefficient of variation of benefit samples

C-N	koz_0	koz_1	pat_0	pat_1
annie	33	33	30	33
cart	32	33	33	32
fact	33	33	33	33
monkey	33	33	30	31
multiplexer	33	33	33	33
power	33	33	33	33
tile1	33	32	33	33
tile2	33	33	33	33
twobox1	33	33	33	33
twobox2	33	33	33	33
twobox3	33	33	33	33
twobox4	33	33	33	33

Table 5-6: C-N: Sample size of cost samples

C-MN	koz_0	koz_1	pat_0	pat_1
annie	52348.485	78954.545	54956.667	80616.667
cart	4562.500	29621.212	7295.455	28003.125
fact	88924.242	91621.212	94871.212	94813.636
monkey	72924.242	72227.273	95240.000	95304.839
multiplexer	56863.636	51560.606	55416.667	66680.303
power	46772.727	47045.455	40234.848	59968.182
tile1	82469.697	54031.250	62386.364	41619.697
tile2	65166.667	50196.970	77380.303	44287.879
twobox1	62863.636	62166.667	85513.636	84037.879
twobox2	48045.455	94590.909	50407.576	95007.576
twobox3	60833.333	78196.970	64183.333	86959.091
twobox4	59954.545	81560.606	80501.515	94243.939

Table 5-7: C-MN: Mean of cost samples

C-SD	koz_0	koz_1	pat_0	pat_1
annie	27423.896	26004.916	18702.184	19895.864
cart	2906.472	33992.423	3038.924	13625.071
fact	21440.368	19900.938	13188.176	13029.769
monkey	10553.525	16388.397	4135.661	4796.168
multiplexer	29137.410	32297.387	32504.737	29283.159
power	39335.315	33837.009	33358.323	32194.783
tile1	27698.697	42757.368	39453.317	40757.242
tile2	36116.709	39833.627	31675.271	40217.804
twobox1	25191.539	28398.577	16603.818	24141.727
twobox2	30364.546	5658.863	28393.441	4870.449
twobox3	28688.703	22040.424	28319.667	18115.866
twobox4	24259.136	17489.174	20794.923	7890.261

Table 5-8: C-SD: Standard deviation of cost samples

C-V	koz_0	koz_1	pat_0	pat_1
annie	0.524	0.329	0.340	0.247
cart	0.637	1.148	0.417	0.487
fact	0.241	0.217	0.139	0.137
monkey	0.145	0.227	0.043	0.050
multiplexer	0.512	0.626	0.587	0.439
power	0.841	0.719	0.829	0.537
tile1	0.336	0.791	0.632	0.979
tile2	0.554	0.794	0.409	0.908
twobox1	0.401	0.457	0.194	0.287
twobox2	0.632	0.060	0.563	0.051
twobox3	0.472	0.282	0.441	0.208
twobox4	0.405	0.214	0.258	0.084

Table 5-9: C-V: Coefficient of variation of cost samples

Table names begin with *B* or *C*, for *benefit* or *cost* respectively. In fact, no attempt was made to render cost and benefit comensurate; there seemed to be no reasonable way to do so. Benefit in these tables is simply fitness.

Table names end with *N*, *MN*, *SD* or *V* for *sample size*, *sample mean*, *sample standard deviation*, or *sample coefficient of variation* respectively.

Annie through *Twobox4* are problems in alphabetical order. *Koz_0* through *Pat_1* are engine configurations.

5.4.1 Pairwise comparison

Using the data in the above tables it is straightforward to compare any pair of samples. But it is immediately apparent that pairwise comparison is of limited use, because there are so many pairs. One obvious choice is to compare Tile 1 and Tile 2, in each of the 4 engine configurations, for cost and benefit. Given that the sample sizes are all over 30 there is no need to use Student's T test — the usual Z test will do [Freund, 1979].

The test statistic for comparing two sample means is

$$Z = \frac{m_1 - m_2}{\sqrt{\frac{s_1^{\,2}}{n_1} + \frac{s_2^{\,2}}{n_2}}}$$

where *m* is the sample mean, *s* the sample standard deviation, and *n* the sample size. The critical value for *Z* in a two-sided alternative is ±1.96 at 5%. Applying this to the benefit and cost measurements for the tile problems, we have:

		Tile 1			Tile 2			
	config	n	m	s	n	m	s	Z
B	koz_0	33	7231.818	469.365	33	7083.697	499.661	1.24
B	koz_1	32	7094.389	583.392	33	7003.607	493.339	0.68
B	pat_0	33	7006.970	375.457	33	7129.788	434.528	-1.23
B	pat_1	33	7020.398	408.650	33	7131.305	604.171	-0.87
C	koz_0	33	82469.697	27698.697	33	65166.667	36116.709	2.18
C	koz_1	32	54031.250	42757.368	33	50196.970	39833.627	0.37
C	pat_0	33	62386.364	39453.317	33	77380.303	31675.271	-1.70
C	pat_1	33	41619.697	40757.242	33	44287.879	40217.804	-0.27

Table 5-10: comparison of tile versions

The leftmost column has B or C for benefit or cost. The rightmost column shows the *Z* value for each row. For example, the *Z* value 1.24 is the test statistic for comparing the benefit observations of tile 1 and tile 2, under the koz_0 configurations.

Only one *Z* value exceeds the critical value — the koz_0 cost comparison. But given the 5% confidence level it is probably reasonable to conclude that 1 rejection of the null hypothesis in 8 tests is not significant, and that the two versions of the tile problem behave indistinguishably.

I have little doubt that other pairwise comparisons would show that other pairs of problems and possibly of engines are similarly indistinguishable. For a rigorous analysis this should be taken into account, for example by pooling the relevant samples. However, I have kept the samples distinct for the purposes of demonstrating the visualisation techniques in action.

5.4.2 Visualisation

Using the method described in §3.3 *Visualisation*, and the data given above, we can produce cladograms for problems and engines.

The problem cladogram is shown below:

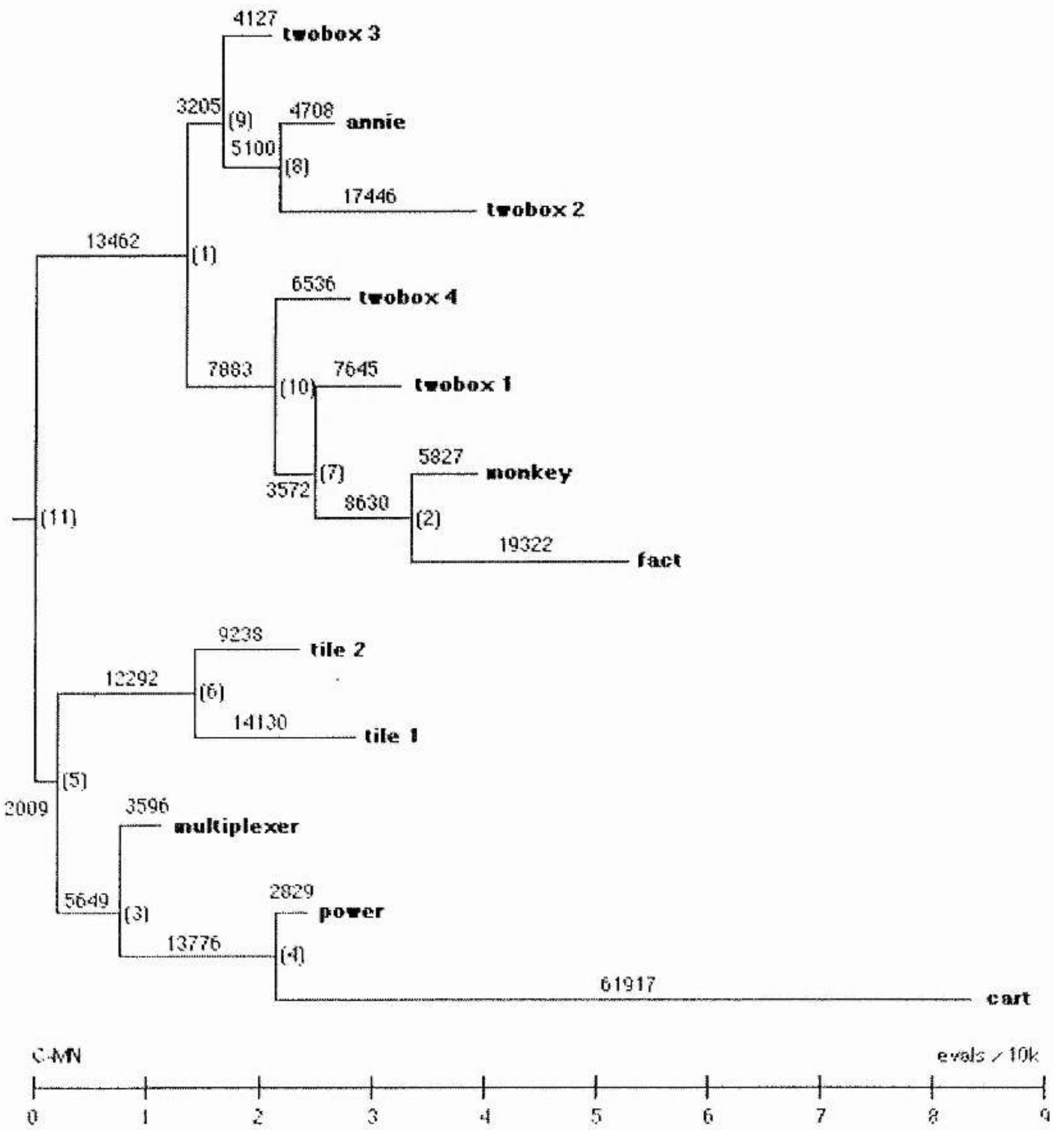

Figure 5-3: Problem cladogram (mean)

As already explained the diagram is a 2-dimensional representation of a set of points in an *n*-dimensional space. The points in the *n*-dimensional space are the problems; the 4-d space is defined by the cost (in evaluations) measured by the four engine configurations.

The horizontal dimension of the diagram uses a rational scale to represent distance. For example, the distance between the points **power** and **cart** is 2 829 + 61 917 = 64 746, which approximates the distance between these points in the original 4-dimensional space. The horizontal scale is shown at the foot of the diagram. The length of each horizontal arc is given above the arc.

The vertical dimension is categorical, and simply places each point in its own category. Vertical lines do not represent distance. The vertical categories are sorted by horizontal length, with shortest length first. Thus, **tile 2** lies above **tile 1**, and the entire **tile** subtree lies above the entire **multiplexer-power-cart** subtree.

The numbers in parentheses are the node numbers assigned by Phylip. They are not significant but have been left in to identifiy the nodes.

For comparison, the following cladogram is based on the same data but uses median rather than mean:

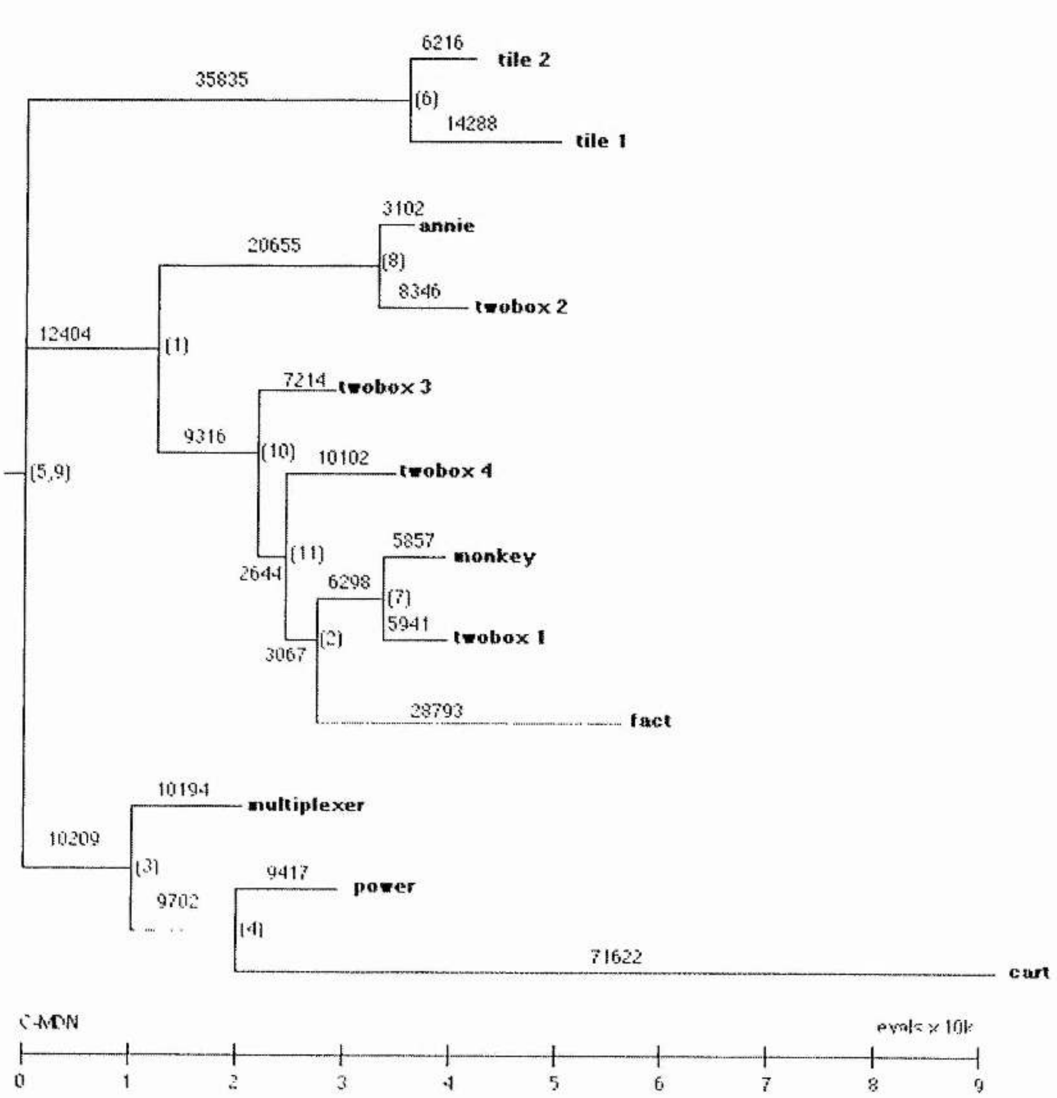

Figure 5-4: Problem cladogram (median)

At first glance figures 5-3 and 5-4 appear quite different, but on closer inspection it turns out that much of the difference is not significant. At the root, two numbers in parentheses indicate that two Phylip nodes (nodes 5 and 9) coincide. In the main body of the diagram, three main subtrees match those of the mean version. The vertical ordering is different, but as the vertical scale is catergorical, that is not significant. The only structural differences are in the middle subtree of the median cladogram, where there has been some rearrangement of the clades.

The engine cladogram is shown below:

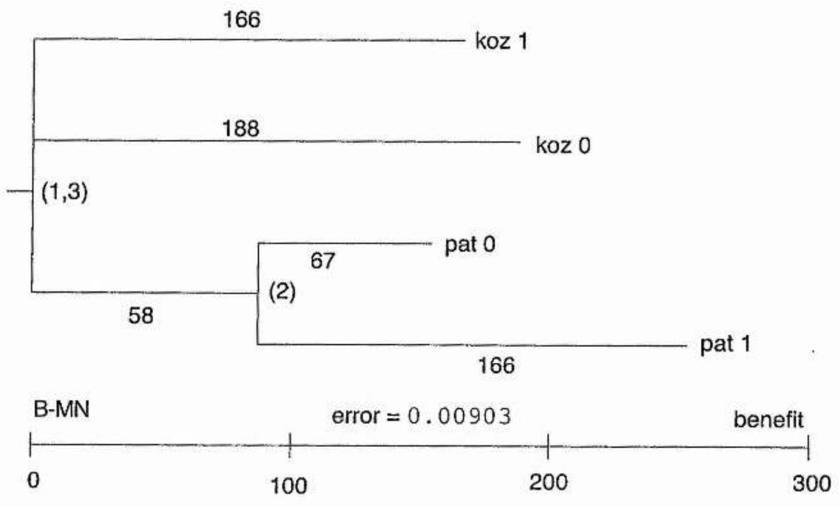

Figure 5-5: Engine cladogram (mean)

For comparison, the median version is also given:

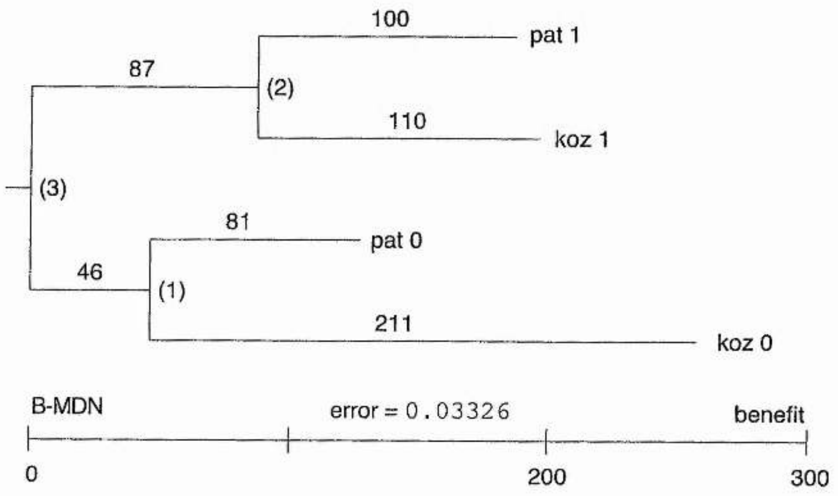

Figure 5-6: Engine cladogram (median)

These diagrams represent the 4 points in the 12-dimensional engine space. The unit of measure is benefit (ie fitness). With only 4 engines it is not possible to learn much from these diagrams, but it is noteworthy that the diagrams do detect the two ECJ parameter sets. The main clades of the mean version correspond to the style, and the main clades of the median version correspond to the parsimony option.

5.4.3 Rational scale comparison

The mean versions of the cladograms were used to compute weights for the problems and engines, and then to compute the weighted mean performance of problems and engines.

5.4.3.1 Problem weights — engine performance

Engine performance is a measure of how effective each engine was, in terms of how fit the best-of-run was after 100k evaluations, averaged over the 12 problems.

The problem weights were as follows:

Problem	Weight (%)	Weight (absolute)
annie	4.4%	10249
cart	30.5%	71090
fact	12.3%	28722
monkey	6.5%	15227
multiplexer	2.5%	5881
power	5.2%	12002
tile1	8.9%	20678
tile2	6.8%	15786
twobox1	5.5%	12730
twobox2	9.9%	22988
twobox3	3.1%	7118
twobox4	4.5%	10430

Table 5-11: Problem weights

The weighted and unweighted average and ranked engine performance is given in the following table:

Engine	Weighted mean	Unweighted mean	Weighted rank	Unweighted rank
koz_0	9228	9035	1	1
koz_1	9202	8987	2	4
pat_0	9198	9001	3=	2
pat_1	9198	8998	3=	3

Table 5-12: Engine performance

The mean values are shown on the chart below:

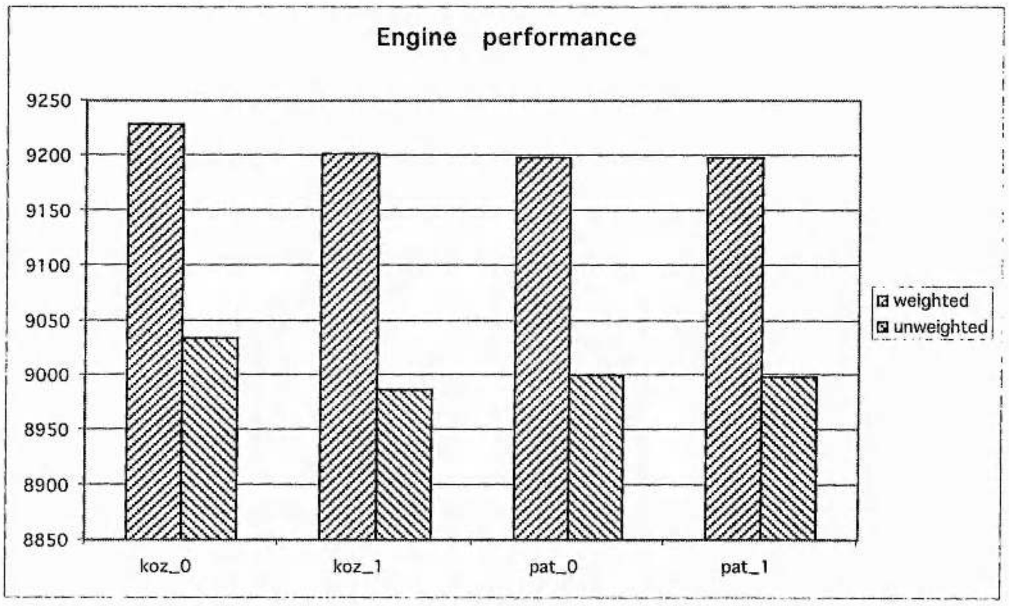

Figure 5-7: Engine performance

The overall increase in the weighted performance is due to the large weight (30.5%) given to the Cart problem. The Cart problem is easy — the ideal solution is always found — and the weighted engine performance reflects this.

The effect of the weights on the ranked performance is to change the obvious conclusion from *parsimony pressure improves performance* to *Koza style parameters are better than Paterson style.* Of course these conclusions are simplistic, given the variation in the data.

5.4.3.2 Engine weights — problem performance

Problem performance is a measure of how difficult each problem is, in terms of how many evaluations were necessary to reach the best-of-run in 100k evaluations, averaged over the 4 engines.

The engine weights were as follows:

Engine	Weight (%)	Weight (absolute)
koz_0	29.2%	188
koz_1	25.7%	166
pat_0	14.8%	95
pat_1	30.3%	195

Table 5-13: Engine weights

The weighted and unweighted average and ranked problem performance is given in the following table:

Problem	Weighted mean	Unweighted mean	Weighted rank	Unweighted rank
annie	68138	66719	7	7
cart	18510	17371	12	12
fact	92282	92558	1	1
monkey	82829	83924	2	2
multiplexer	58261	57630	9	10
power	49873	48505	11	11
tile1	59811	60127	8	8
tile2	56801	59258	10	9
twobox1	72453	73645	6	4
twobox2	74587	72013	4	6
twobox3	73708	72543	5	5
twobox4	78938	79065	3	3

Table 5-14: Problem performance

The mean values are shown on the chart below:

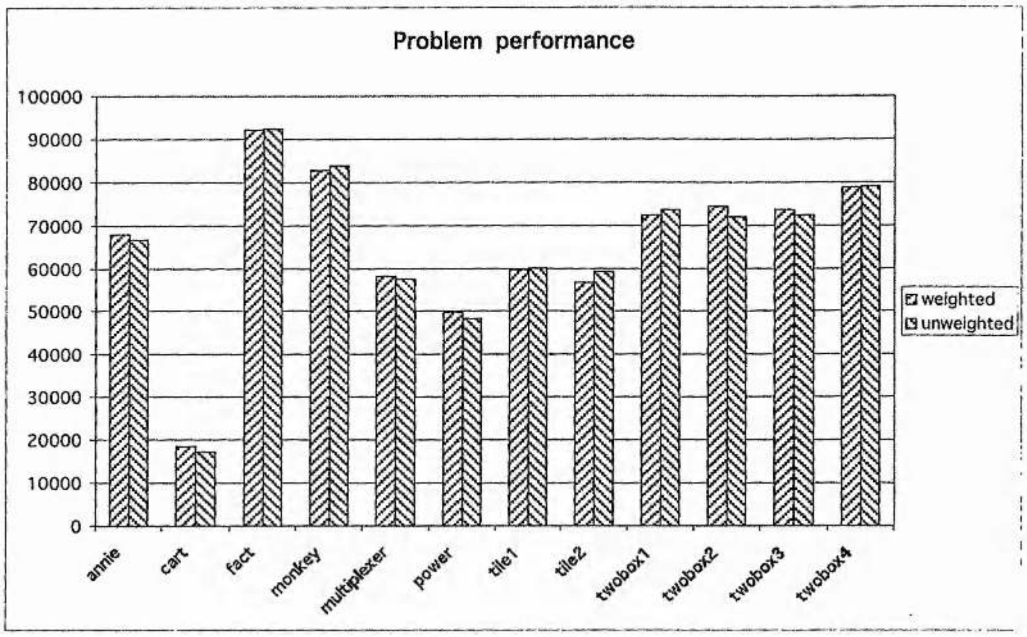

Figure 5-8: Problem performance

There is little difference between the weighted and unweighted means.

Differences are more evident in the rankings, where pairs (9, 10) and (4, 6) are swapped. These pairs are (**multiplexer**, **tile2**) and (**twobox1**, **twobox2**) respectively. Given the slight effect of the weighting and the variance in the underlying data it would be unwise to attach any significance to this change in ranking.

6 Conclusions

This section discusses the contribution of this research to GP, its limitations, and suggests directions for future work.

6.1 Contribution

The contributions of this thesis are shown below in order of significance as it appears at the time of writing.

6.1.1 Context sensitivity

Gads provides a scalable solution for evolving type-correct software in independently-chosen context-sensitive languages.

Up till now, small phenotype languages have been the only option. The lack of a general mechanism for context sensitivity meant that only small grammars were feasible, whether they were tightly or loosely coupled to the evolutionary engine. Gads extends GP to loosely coupled full-size context-sensitive phenotype languages. The implications of this remain to be explored. The obvious possibility is that more complex programs in human programming languages such as Java or C, including data structures and procedures, can be evolved entirely from scratch. Gad also presents the possibility of defining new languages for circuit design, scheduling, or other problem domains, and evolving solutions in these languages.

6.1.2 Non genetic search

The ontogenic mapping enables non-genetic optimisation systems to perform automatic programming.

Any system that can search for optimal solutions in a space represented by a list or array of integers, directed by an objective function, can now search for sentences in context-sensitive languages. For example SA, ES or possibly even neural nets could evolve programs, and thus open other avenues to automatic programming.

6.1.3 Performance

Separation of genotype and phenotype, with the ontogenic mapping, provides improvement in evolutionary performance over SGP. This was noted in §2 *Gads 1* and has been independently confirmed in [Freeman, 1998] and [O'Neill, 2001c].

6.1.4 Statistics

The initial discussion in §3.1 *Statistical perspective* gives useful insights into what can be investigated. Matching the components of the GP system with the corresponding components of the statistical model is non-trivial, and helps to avoid conceptual blunders.

§3.2 *Performance comparison* gives a much-needed foundation to the use of statistical tests in comparing performance of GP systems.

§3.3 *Visualisation* suggests new ways to represent experimental GP data.

6.1.5 Languages and compilers

Rags reveals many features of the phenotype language that were probably unintentional. For example, 5 + + – + 6 is valid S-algol, but it probably should not be. Rags could be a useful tool in the design of programming languages and in the testing of compilers.

6.1.6 Solution

The solution to the tile problem is a minor contribution. This solution is a good example of GP at work. Faced with a problem for which there may well be no closed-form solution, GP is nonetheless able to find an approximate solution. This particular example must also be counted a success for Gads 2, as it demonstrates the evolution of an expression involving mixed types.

6.2 Limitations

The main limitation is that this thesis is broad rather than deep. It covers a wide range of topics, but does not deal with any of them in great depth. Experimental results are given, but little or no theoretical analysis.

6.2.1 Statistics

Some of the analysis in §3.2 *Performance comparison* has little theoretical foundation. Measures such as delta, although plausible, may have flaws as yet undiscovered.

The data underlying §3.3 *Visualisation* is sample means. The means are treated as points instead of estimates, which is an over-simplification. A more robust method would take the standard error of the mean into account when producing the cladogram and again when the performance of engines and problems is reduced to a rational scale. The final result should be that each engine or problem is represented by an interval on the rational scale, not a by point.

6.2.2 Rags implementation

The implementation of rags is suitable for a proof-of-concept, but it is not a polished product. It is a single-use design consistent with Brooks' advice: *plan to throw one away* [Brooks, 1995].

An aspect of this is that Gads 2 is not optimised for performance. Its performance as recorded here provides a baseline; we may expect tuning to produce better performance though there is as yet no concrete evidence on which to base this expectation.

6.3 Questions from Gads 1

This section revisits §2.6 *Questions raised.*

6.3.1 Specifying sentence distribution

The issue of biasing the grammar to make some sentences more likely than others is still to be investigated. Would it be helpful to attach weights to production alternatives so that some were more likely to be chosen than others? If so, how would the weights be set? Could they be evolved? Would they apply to the entire run or could they vary from one individual to another?

6.3.2 Moving away from Lisp

The goal of using a language other than Lisp is achieved in §5 *Gads 2.*

6.3.3 Functions, work variables, etc

The goal of evolving re-usable functions and work variables is achieved in §5 *Gads 2.*

6.3.4 Choosing sentence distribution

A study of the rule frequencies in the derivation of real programs has not been carried out.

6.3.5 Statistical analysis

The weakness of the analysis in §2 *Gads 1* is addressed satisfactorily in §3 *Statistics.*

6.3.6 Sequential chromosomes

This question has been addressed to some extent in [Keijzer, 2001].

6.3.7 Gene effectiveness

This question is specific to Gads 1. As Gads 2 uses translation the question is no longer of interest.

6.3.8 Genetic operations

Different crossover techniques are discussed in [O'Neill, 2001b] and [Keijzer, 2001].

6.3.9 Initial distribution

A comparison of generation 0 from Gads and SGP is still to be made.

6.4 Future work

This section presents a number of questions raised by this thesis, including those still open in §6.3 *Questions from Gads 1* above.

6.4.1 Statistics

The main theme of §3 *Statistics* is the application of statistics to GP. GP systems exhibit behaviour that is much more complex than traditional computer science is used to dealing with, possibly more like biology than engineering. For this reason, we should look to statistics and the life sciences to see what tools and techniques can be adapted for GP.

There is a sizeable body of statistics (eg [Morrison, 2001], [Felsenstein, 1995]) dedicated to problems such as visualisation, which should be investigated. Latent variable models (eg [Loehlin, 1992]) offer another avenue which should be investigated.

§3.1.2 *Populations and samples* makes a claim that is empirically testable, namely that generation 0 can be treated as a sample but generation $n > 0$ cannot, and the effect is greater as n increases. According to the Central Limit Theorem, the sampling distribution of the mean approximates the normal distribution [Freund, 1979]. That is, if we take large (≥ 30) samples of a population and compute their means, these means are distributed normally. This is true whatever the population distribution. To test the claim, proceed as follows. For a range of problems and engines, and n taking a suitable range of values, say 0, 10, 20, 30, 40, and 50 in turn, let the GP system run to generation n, and measure the mean fitness (or size, or any other performance measure you like). (The range of values for n is not strictly necessary: only the largest value really needs to be tested. But a range should show a trend which is more convincing than a single point.) Do this 30 times with the same configuration and a different RNG seed. Use a goodness-of-fit test to decide whether the means of a configuration are normally distributed. If the argument in §3.1.2 is sound, the means should always be normally distributed when $n = 0$, and in general less so as n increases.

It could be argued that there is no need to test this claim as it has been mathematically proven. This is not quite so. The Central Limit Theorem is a theorem, and may be taken as proven, but that is not what is being tested. The test is whether the analysis made in §3.1.2 *Populations and samples* is sound. That analysis makes a prediction which can be tested by experiment. The behaviour of GP systems is sufficiently complex that empirical verification of theoretical predictions is valuable.

What is needed is a reference work entitled *Statistical techniques for evolutionary computation.*

6.4.2 Grammars

Theoretical aspects of rags should be explored. For example: what kind of language is necessary for the production methods? Do rags restrict how scope is defined? Are there useful subclasses of CFGs?

An implementation of rags designed for other researchers to use, probably in Java, would make this a useful tool. Rags for other languages (such as C, C++, Java) could be developed.

A method to avoid having to generate an individual's PT and flattening it into a character string, only for a compiler to have to read it and reconstruct the PT, would greatly improve wall-clock performance.

All the languages considered so far have in common the property that identifiers are introduced and then used. Not all problems fit this model. An example of a different model is the children's game in which a set of randomly chosen integers have to be combined into an expression whose value is equal to some goal. For example, if the set was {5, 7, 19, 20, 56} and the goal was 44 then a solution would be:

$$56-(5+7)*(20-19)$$

The player starts with the complete set of integers, but each one can only be used once. When any given integer is used, it creates a context in which the set of available integers has been reduced. For a rag, this requires removing alternatives from the language, not adding them.

6.4.3 Performance comparison

§5 *Gads 2* aims to carry out a deeper investigation of Gads 2. An original aim was to compare the performance of Gads 2 with that of an SGP system. This was not done, and §5 *Gads 2* was reduced to a demonstration of Gads 2 and a rough baseline of its performance. A comparison remains necessary.

The notion of comensurate cost and benefit, introduced in §3.3 *Visualisation,* is not used in §5 *Gads 2.* The need for this concept, and its use in practice, therefore remains to be demonstrated.

See also §6.3.9 *Initial distribution.*

6.4.4 Analysis of Gads

See also §6.3.6 *Sequential chromosomes* and §6.3.8 *Genetic operations.*

6.4.5 GP system design

GP is expected to discover every wheel from scratch. This is more apparent in Gads 2 than in SGP, since Gads 2 can use a standard programming language, with all the usual programming concepts. But the conceptual level of the constructs in a programming language such as S-algol or Java is extremely low compared to the level at which human programmers work. We don't expect a totally inexperienced programmer to discover and implement concepts like stacks or linked lists, every time they have to write a program; so why should GP? What is needed is a library of programming strategies, which the GP system can draw on. Strategies might include, for example, divide-and-conquer, induction, tail recursion, or to-do-lists. It is not immediately obvious which strategies to include, or how they could be represented. Something like this has been done in functional languages, which have the expressive power to represent a strategy like divide-and-conquer as a (very) high-order function.

See also §6.3.1 *Specifying sentence distribution* and §6.3.4 *Choosing sentence distribution.*

6.4.6 Biological analogy

[de Jong, 2001] describes an experiment in which the population diversity is explicitly maintained. This illustrates a point which is often made, namely that the pressure behind biological evolution is replication, not adaptation. Adaptation is possible because of replication, not vice versa. In biological terms, programs like cart-centring are probably the most successful, because so many copies of this program exist. The purpose of GP for humans is not replication but adaptation. (It is permissible to use the term *purpose* here; biological evolution has no purpose.) It is therefore a mistake to aim to simulate biological evolution in all its aspects. GP should identify and adopt just those aspects of biological evolution which result in adaptation.

An analogy between GP and biological evolution implies a correspondence between the two sides. There seems to me to be a mismatch between the information content of the supposedly matched terms. The biological side carries much more information than the GP side; in other words, GP uses terms that imply more information content than they actually contain. The most obvious example is perhaps the GP use of the term *gene*, which in GP usage can contain as little as 1 bit of information, while in biology might contain several hundred bits of information.

I would therefore propose an analogy as follows, with the aim of achieving a closer fit in terms of information content, and of not giving a biological word a radically different meaning in GP.. First, the biological term *base* should correspond to the GP term *bit*. These are close in information capacity, and both are units at the lowest end of their respective scales. By direct analogy, a GP *codon* should then be a contiguous group of bits, and a GP *gene* should be a sequence of codons. (This is a change from current GP usage, where *gene* is often synonymous with *codon*.)

The biological term *translation*, which associates codons with amino acids (the full biological process is not relevant here) should correspond to the GE or Gads *translation* which maps codons to rules in the grammar. The Gads term *ontogenesis* should be replaced by *procedure synthesis*, corresponding to *protein synthesis*.

In terms of these definitions, most GP systems deal with single-gene individuals. Further, the typical GP individual corresponds not to a whole biological organism but to a single protein. In terms of information content, matching GP individuals with proteins is more reasonable than matching them with entire organisms.

Clarifying the analogy is not an exercise in pedantry. The purpose of an analogy is to carry ideas from one discipline to another. I have suggested that GP currently operates at the level of the single gene and protein synthesis. Perhaps this is the natural limit of GP. But if we wish to consider evolving software orders of magnitude more complex than GP can currently produce, we will need new ways to go about it. The biological analogy suggests a way forward. In biology, a chromosome consists of several genes, and an entire eukaryotic genotype consists of several chromosomes. These organisational structures could correspond to the organisational structures in an object-oriented program. Each gene could represent a method or attribute; each chromosome a class, and the entire genotype a complete program. By organising GP in this way, we might be able to tackle problems far in excess of any currently being investigated.

A References

[Angeline, 1996a]	An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover. Angeline, Peter J. Pp 21-29. In: Genetic Programming 1996. Proceedings of the first annual conference. Eds: Koza, John R; Goldberg, David E; Fogel, David B; Riolo, Rick. The MIT Press. 1996. ISBN 0-262-61127-9.
[Bratley, 1983]	A guide to simulation. Bratley, Paul; Fox, Bennett L; Schrage, Linus F. Springer-Verlag. 1983. ISBN 038790820X.
[Brooks, 1995]	The mythical man-month. Essays on software engineering. Ed 20th Anniversary. Brooks, Frederick P. Addison-Wesley Publishing Company. 1995. ISBN 0201835959.
[Bruhn, 2002]	Genetic programming over context-free languages with linear constraints for the knapsack problem. First results. Bruhn, Peter; Geyer-Schulz, Andreas. Evolutionary Computation. Vol 10. No 1. Pp 51-74. Massachusetts Institute of Technology.
[Bryson, 1975]	Applied optimal control. Bryson, Arthur E; Ho, Yu-Chi. Hemisphere.
[Clack, 1997]	Performance enhanced genetic programming. Clack, Chris; Yu, Tina. In: Proceedings of the sixth conference on evolutionary programming. Eds: Angeline, Peter J; Reynolds, Robert G; McDonnell, John R; Eberhart, Russ. Lecture notes in computer science. Vol 1213. Springer-Verlag.
[Cohen, 1995]	Empirical methods for artificial intelligence. Cohen, Paul R. The MIT Press. 1995. ISBN 0-262-03225-2.
[Conover, 1971]	Practical nonparametric statistics. Conover, William Jay. John Wiley & Sons. 1971. ISBN 0-471-16851-3.
[Daida, 1997]	Challenges with verification, repeatability and meaningful comparisons in genetic programming. Daida, Jason M; Ross, Steven; McClain, Jeffrey; Ampy, Derrick; Holczer, Michael. Pp 64-70. In: Genetic Programming 1997. Proceedings of the second annual conference. Eds: Koza, John R; Deb, Kalyanmoy; Dorigo, Marco; Fogel, David B; Garzon, Max H; Iba, Hitoshi; Riolo, Rick. Morgan Kaufmann Publishers. 1997. ISBN 1-55860-483-9.
[Darwin, 1859]	On the origin of species by means of natural selection. Or, the preservation of favoured races in the struggle for life. Darwin, Charles. URL http://www.infidels.org/library/historical/charles_darwin/origin_of_species/index.shtml.
[de Jong, 2001]	Reducing bloat and promoting diversity using multi-objective methods. de Jong, Edwin D; Watson, Richard A; Pollack, Jordan B. Pp 11-18. In: GECCO-2001. Proceedings of the genetic and evolutionary computation conference. Eds: Spector, Lee; Goodman,

Erik D; Wu, Annie S; Langdon, William B; Voigt, Hans-Michael; Gen, Mitsuo; Sen, Sandip; Dorigo, Marco; Pezeshk, Shahram; Garzon, Max H. Morgan Kaufmann Publishers. 2001. ISBN 1-55860-774-9.

[Deransart, 1988] Attribute grammars. Definitions, systems and bibliography. Deransart, Pierre; Jourdan, Martin; Lorho, Bernard. Springer-Verlag. 1988. ISBN 3-540-50056-1.

[Felsenstein, 1995] PHYLIP. Phylogeny inference package. Ed 3.57c. Felsenstein, Joseph. URL http://evolution.genetics.washington.edu/phylip.html. Downloaded 12-Jun-1998.

[Ferreira, 2001] Gene expression programming. A new adaptive algorithm for solving problems. Ferreira, Cândida. URL http://www.gene-expression-programming.com/webpapers/gep.pdf. Downloaded 10-Oct-2001.

[Fitch, 1967] The construction of phylogenetic trees. A generally applicable method utilizing estimates of the mutation distance obtained from cytochrome c sequences. Fitch, Walter M; Margoliash, Emanuel. Science. Vol 155. No 3760. Pp 279-284.

[Freeman, 1998] A linear representation for GP using context free grammars. Freeman, Jennifer J. Pp 72-77. In: Genetic programming 1998. Proceedings of the third annual conference. Eds: Koza, John R; Banzhaf, Wolfgang; Chellapilla, Kumar; Deb, Kalyanmoy; Dorigo, Marco; Fogel, David B; Garzon, Max H; Goldberg, David E; Iba, Hitoshi; Riolo, Rick. Morgan Kaufmann Publishers. 1998. ISBN 1-55860-548-7.

[Freund, 1979] Modern elementary statistics. Ed 5. Freund, John E. Prentice/Hall. 1979. ISBN 0-13-593376-5.

[Holland, 1992] Adaptation in natural and artifical systems. An introductory analysis with applications to biology, control and artificial intelligence. Holland, John H. The MIT Press. 1992. ISBN 0-262-58111-6.

[Hopcroft, 1969] Formal languages and their relation to automata. Hopcroft, John E; Ullman, Jeffrey D. Addison-Wesley Publishing Company. 1969.

[Hörner, 1996] A C++ class library for genetic programming. The Vienna University of Economics Genetic Programming Kernel. Ed 1. Hörner, Helmut. Vienna University of Economics. 1996.

[Keijzer, 2001] Ripple Crossover in Genetic Programming. Keijzer, Maarten; Ryan, Conor; O'Neill, Michael; Cattolico, Mike; Babovic, Vladan. Pp 74-86. In: Genetic programming. Proceedings of EuroGP'2001. Eds: Miller, Julian F; Tomassini, Marco; Lanzi, Pier Luca; Ryan, Conor; Tettamanzi, Andrea G B; Langdon, William B. Lecture notes in computer science. Vol 2038. Springer-Verlag. ISBN 3-540-41899-7.

[Keller, 1996] Genetic programming using genotype-phenotype mapping from linear genomes into linear phenotypes. Keller, Robert E; Banzhaf, Wolfgang. In: Genetic Programming 1996. Proceedings of the first annual conference. Eds: Koza, John R; Goldberg, David E; Fogel, David B; Riolo, Rick. The MIT Press. 1996. ISBN 0-262-61127-9.

[Kirby, 2000] S-algol & Napier 88. Eds: Kirby, Graham; Brown, Alfred; Connor, Richard; Cutts, Quintin; Dearle, Alan; Morrison, Ron; Munro, Dave. URL ftp://ftp.dcs.st-and.ac.uk/pub/Software/PROG_LANGS. Downloaded 18-Apr-2002. School of Computer Science, University of St Andrews.

[Koza, 1992] Genetic programming. On the programming of computers by means of natural selection. Koza, John R. The MIT Press. 1992. ISBN 0-262-11170-5.

[Koza, 1994] Genetic Programming II. Automatic discovery of reusable programs. Koza, John R. The MIT Press. 1994. ISBN 0-262-11189-6.

[Koza, 1996a] Four problems for which a computer program evolved by genetic programming is competitive with human performance. Koza, John R; Bennett III, Forrest H; Andre, David; Keane, Martin A. In: Proceedings of 1996 IEEE International Conference on Evolutionary Computation (ICEC '96). IEEE Press. 1996. ISBN 0-7803-2902-3.

[Koza, 1997a] Genetic Programming 1997. Proceedings of the second annual conference. Eds: Koza, John R; Deb, Kalyanmoy; Dorigo, Marco; Fogel, David B; Garzon, Max H; Iba, Hitoshi; Riolo, Rick. Morgan Kaufmann Publishers. 1997. ISBN 1-55860-483-9.

[Koza, 1998c] Evolutionary Design of Analog Electrical Circuits using Genetic Programming. Koza, John R; Bennett III, Forrest H; Andre, David; Keane, Martin A. In: Proceedings of Adaptive Computing in Design and Manufacture Conference. 1998.

[Lawrence, 1997] On the distribution of performance from multiple neural network trials. Lawrence, Steve; Back, Andrew D; Tsoi, Ah Chung; Giles, C Lee. IEEE Transactions on Neural Networks. Vol 8. No 6. Pp 1507-1517.

[Loehlin, 1992] Latent variable models. An introduction to factor, path and structural analysis. Ed 2. Loehlin, John C. Lawrence Erlbaum Associates. 1992.

[Luke, 1997] A comparison of crossover and mutation in genetic programming. Luke, Sean; Spector, Lee. Pp 240-248. In: Genetic Programming 1997. Proceedings of the second annual conference. Eds: Koza, John R; Deb, Kalyanmoy; Dorigo, Marco; Fogel, David B; Garzon, Max H; Iba, Hitoshi; Riolo, Rick. Morgan Kaufmann Publishers. 1997. ISBN 1-55860-483-9.

[Luke, 1998] A revised comparison of crossover and mutation in genetic programming. Luke, Sean; Spector, Lee. Pp 208-213. In: Genetic programming 1998. Proceedings of the third annual conference. Eds: Koza, John R; Banzhaf, Wolfgang; Chellapilla, Kumar; Deb, Kalyanmoy; Dorigo, Marco; Fogel, David B; Garzon, Max H; Goldberg, David E; Iba, Hitoshi; Riolo, Rick. Morgan Kaufmann Publishers. 1998. ISBN 1-55860-548-7.

[Luke, 2001] ECJ. A Java-based evolutionary computation and genetic programming research system. Ed 8. Luke, Sean. URL http://www.cs.umd.edu/projects/plus/ec/ecj/ec.tar.gz. Downloaded 7-Mar-2002.

[Macki, 1982] Introduction to optimal control. Macki, Jack; Strauss, Aaron. Springer-Verlag. 1982.

[Merelo, 1996] Genetic algorithms from Granada, Spain. Merelo, J J. URL ftp://kal-el.ugr.es/GAGS/GAGS-.95.tar.gz.

[Michalewicz, 1994] Genetic algorithms + Data structures = Evolution programming. Michalewicz, Zbigniew. Springer-Verlag. 1994. ISBN 3-540-58090-5.

[Montana, 1995] Strongly typed genetic programming. Montana, David J. Evolutionary Computation. Vol 3. No 2. Pp 199-230.

[Morrison, 1979] S-algol reference manual. Technical report CS/79/1, School of Computer Science, University of St Andrews. Morrison, Ron. 1979.

[Morrison, 2001] Society of systematic Australian biologists. Website. Ed: Morrison, David. URL http://www.science.uts.edu.au/sasb/. Downloaded 12-Aug-2002. University of Technology, Sydney.

[Naur, 1963] Revised report on the algorithmic language Algol 60. Eds: Naur, P; Woodger, M. Communications of the ACM. Vol 6. No 1. Pp 1-17.

[Nordin, 1994a] A compiling genetic programming system that directly manipulates the machine code. Nordin, Peter. Pp 311-331. In: Advances in genetic programming. Ed: Kinnear Jr, Kenneth E. The MIT Press. 1994. ISBN 0-262-11188-8.

[O'Neill, 2001b] Automatic programming in an arbitrary language: evolving programs with grammatical evolution. A thesis for the PhD degree submitted to the University of Limerick. O'Neill, Michael. 2001.

[O'Neill, 2001c] Grammatical evolution. O'Neill, Michael; Ryan, Conor. IEEE Transactions on Evolutionary Computation. Vol 5. No 4. Pp 349-358.

[Pagan, 1981] Formal specification of programming languages. A panoramic primer. Pagan, Frank G. Prentice-Hall. 1981. ISBN 0-13-329052-2.

[Paterson, 1996] Distinguishing genotype and phenotype in genetic programming. Paterson, Norman R; Livesey, Michael J. Pp 141-150. In: Genetic Programming 1996. Proceedings of the first annual conference. Eds: Koza, John R; Goldberg, David E; Fogel, David B; Riolo, Rick. The MIT Press. 1996. ISBN 0-262-61127-9.

[Paterson, 2000]	Performance comparison in genetic programming. Paterson, Norman R; Livesey, Michael J. Pp 253-260. In: GECCO-2000. Late-breaking papers at the genetic and evolutionary computation conference. Ed: Whitley, Darrell. Morgan Kaufmann Publishers. 2000. ISBN 1-55860-708-0.
[Ryan, 1998a]	Grammatical evolution. Evolving programs for an arbitrary language. Ryan, Conor; Collins, J J; O'Neill, Michael. Pp 83-95. In: Proceedings of the first European workshop on genetic programming. Eds: Banzhaf, Wolfgang; Poli, Riccardo; Schoenauer, Marc; Fogarty, Terence C. Springer-Verlag. 1998. ISBN 3-540-64360-5.
[Spector, 2002]	Genetic programming and autoconstructive evolution with the Push programming language. Spector, Lee; Robinson, Alan. Genetic programming and evolvable machines. Vol 3. Pp 7-40. Kluwer Academic Publishers.
[Whigham, 1996]	Grammatical bias for evolutionary learning. A thesis submitted to the School of Computer Science, University College, University of New South Wales, Australian Defence Force Academy, for the degree of Doctor of Philosophy. Whigham, Peter Alexander. 1996.
[Whitley, 1995]	Genetic Algorithms and Neural Networks. Whitley, Darrell. Pp 203-216. In: Genetic Algorithms in Engineering and Computer Science. Eds: Winter, G; Periaux, Jacques; Galan, M; Cuesta, P. John Wiley & Sons. 1995.
[Wolpert, 1995]	No free lunch theorems for search. Wolpert, David H; Macready, William G. URL http://www.santafe.edu/sfi/publications/Working-Papers/95-02-010.ps. Downloaded 23-Aug-2002.
[Wolpert, 1997]	No free lunch theorems for optimization. Wolpert, David H; Macready, William G. IEEE Transactions on Evolutionary Computation. Vol 1. No 1. Pp 67-82. IEEE Press.

B S-algol rag

The Java code is included here as an aid to understanding details of the implementation. However it must be stressed that this implementation is intended to be a single-use, throw-away design as suggested in [Brooks, 1995].

The rag is given in three parts:

B.1 Syntax
The context-free syntax component of the rag.

B.2 The production methods.
The class definitions for Node and S_algolNode.

B.3 Object creation
The class definition for the initialiser which sets up the run-time objects for the grammar at the start of each run.

A Perl script (not shown) translates the CFG in §B.1 *Syntax* into Java statements which are combined with a standard header and footer to produce the `RAGInitializer` class shown in §B.3 *Object creation*. This class is called by the ECJ GA at the start of each run, and sets up the objects that define the rag (ie terminals, nonterminals, serial counter for identifiers, etc).

The Node class contains the default production method. Its extension S_algolNode contains production methods specific to S-algol. Each method is identified by a number. The `RAGInitializer` class shown in §B.3 *Object creation* uses these numbers to associate a production method with the corresponding production.

B.1 Syntax

The syntax below is switched to the Fact problem, as indicated by the first production.

```
Rules

######## ########
#       Session
######## ########

#       IMPORTANT!  The structure of the following lines
#       is used by ec.experiment.nrp.rag.ecj.RAGProblem4
#       to locate the phenotype.
#       Use next line (program:) to choose a program.
#       Then arrange that the corresponding section is
#       uncommented in Phenotypes.

program:        program fact.

program monkey:
                preamble symbol,
                phenotype monkey, postamble monkey symbol,
                end of program.

program cart:
                preamble symbol,
                phenotype cart, postamble cart symbol,
                end of program.

program tile:
                preamble symbol,
                phenotype tile, postamble tile symbol,
                end of program.

program multiplexer:
                preamble symbol,
                phenotype multiplexer, postamble multiplexer symbol,
                end of program.

program power:
                preamble symbol,
                phenotype power, postamble power symbol,
                end of program.

#       Two Box is in 4 versions:
#
#               Two Box 1 (Koza style: without ADF).
#               Two Box 2 (Koza style: with ADF).
#               Two Box 3 (Paterson style: with unprescribed, non-recursive
ADF).
#               Two Box 4 (Paterson style: with unprescribed, recursive ADF).
#
#       The phenotype is problem-specific, but the rest of the wrapper
#       is common to all 4 versions.

program two box 1:
                preamble symbol,
                phenotype two box 1, postamble two box symbol,
                end of program.

program two box 2:
                preamble symbol,
                phenotype two box 2, postamble two box symbol,
                end of program.

program two box 3:
                preamble symbol,
                phenotype two box 3, postamble two box symbol,
                end of program.

program two box 4:
                preamble symbol,
                phenotype two box 4, postamble two box symbol,
                end of program.

program fact:
                preamble symbol,
                phenotype fact, postamble fact symbol,
                end of program.

program annie:
                preamble symbol,
                phenotype annie, postamble annie symbol,
                end of program.

end of program:     space symbol, question symbol.

sequence void:      clause void;

                clause void, sequence separator, sequence void;

                decl let;

                decl let, sequence separator, sequence void;

                decl proc;

                decl proc, sequence separator, sequence void.

sequence int: clause int;

                sequence void, sequence separator, clause int.

sequence real:      clause real;

                sequence void, sequence separator, clause real.

sequence bool:      clause bool;

                sequence void, sequence separator, clause bool.

sequence string:    clause string;

                sequence void, sequence separator, clause string.

sequence separator:
                space symbol, semicolon symbol, newline symbol.

appl proc type void:
                     proc type void symbol.

appl proc type int:
                     proc type int symbol,
                     clause l, exp0 int, clause r.

appl proc type real:
                     proc type real symbol,
                     clause l, exp0 real, clause r.

appl proc type bool:
                     proc type bool symbol,
                     clause l, exp0 bool, clause r.

appl proc type string:
                     proc type string symbol,
                     clause l, exp0 string, clause r.

######## ########
#      Declarations
######## ########

decl let:     decl let int;

                decl let real;

                decl let bool;

                decl let string.

decl proc:    decl proc void;
```

```
              decl proc int;

              decl proc real;

              decl proc bool;

              decl proc string.

decl let int:
              let symbol, id int new, assignment symbol, clause int.

decl let real:
              let symbol, id real new, assignment symbol, clause real.

decl let bool:
              let symbol, id bool new, assignment symbol, clause bool.

decl let string:
              let symbol, id string new, assignment symbol, clause string.

decl proc void:
              proc symbol, id proc new,
              sequence separator,
              clause void;

              proc symbol, id proc new,
              round l symbol, parameter list, round r symbol,
              sequence separator,
              clause void.

decl proc int:
              proc symbol, id proc new,
              round l symbol, arrow symbol, type int symbol, round r symbol,
              sequence separator,
              clause int;

              proc symbol, id proc new,
              round l symbol, parameter list, arrow symbol, type int symbol,
round r symbol,
              sequence separator,
              clause int.

decl proc real:
              proc symbol, id proc new,
              round l symbol, arrow symbol, type real symbol, round r
symbol,
              sequence separator,
              clause real;

              proc symbol, id proc new,
              round l symbol, parameter list, arrow symbol, type real
symbol, round r symbol,
              sequence separator,
              clause real.

decl proc bool:
              proc symbol, id proc new,
              round l symbol, arrow symbol, type bool symbol, round r
symbol,
              sequence separator,
              clause bool;

              proc symbol, id proc new,
              round l symbol, parameter list, arrow symbol, type bool
symbol, round r symbol,
              sequence separator,
              clause bool.

decl proc string:
```

```
              proc symbol, id proc new,
              round l symbol, arrow symbol, type string symbol, round r
symbol,
              sequence separator,
              clause string;

              proc symbol, id proc new,
              round l symbol, parameter list, arrow symbol, type string
symbol, round r symbol,
              sequence separator,
              clause string.

id int new:   id int new symbol.

id real new:  id real new symbol.

id bool new:  id bool new symbol.

id string new:       id string new symbol.

id proc new:  id proc new symbol.

parameter list:      parameter;

              parameter, parameter separator, parameter list.

parameter:    type int symbol, id int new;

              type real symbol, id real new;

              type bool symbol, id bool new;

              type string symbol, id string new.

parameter separator:        space symbol, semicolon symbol.

######## ########
#      Clauses
######## ########

########
#      Clauses - void
########

clause void:  exp0 void;

              if symbol, clause bool,
              do symbol, clause void;

              if symbol, clause bool,
              then symbol, clause void,
              else symbol, clause void;

              while symbol, clause bool, do symbol, clause void;

              for symbol, id int new, space symbol, equals symbol, clause
int,
              to symbol, clause int,
              do symbol, clause void;

              write;

              id int, assignment symbol, clause int;

              id real, assignment symbol, clause real;

              id bool, assignment symbol, clause bool;

              id string, assignment symbol, clause string.
```

```
write:          write symbol, write list, write newline.

write list:     clause writable;

                clause writable, clause separator, write list.

#               <write newline> is added to S-algol
#               to ensure that output actually appears.
write newline:          clause separator, quote symbol, apostrophe r symbol,
letter lower n symbol, quote symbol.

clause writable:        clause int;

                        clause real;

                        clause string;

                        clause bool.

clause separator:       space symbol, comma symbol.

########
#       Clauses - int
########

clause int:     exp0 int;

                if symbol, clause bool,
                then symbol, clause int,
                else symbol, clause int.

########
#       Clauses - real
########

clause real:    exp0 real;

                if symbol, clause bool,
                then symbol, clause real,
                else symbol, clause real.

########
#       Clauses - bool
########

clause bool:    exp0 bool;

                if symbol, clause bool,
                then symbol, clause bool,
                else symbol, clause bool.

########
#       Clauses - string
########

clause string:          exp0 string;

                if symbol, clause bool,
                then symbol, clause string,
                else symbol, clause string.

######## ########
#       Expressions
######## ########

########
#       Expressions - level 0
########
```

```
exp0 void:    exp1 void.

exp0 int:     exp1 int.

exp0 real:    exp1 real.

#             <new line symbol> is added here to prevent long lines
#             from building up - they crash the S-algol compiler.

exp0 bool:    exp1 bool;

              exp1 bool, or symbol, newline symbol, exp1 bool.

exp0 string:  exp1 string.

########
#     Expressions - level 1
########

exp1 void:    exp2 void.

exp1 int:     exp2 int.

exp1 real:    exp2 real.

#             <new line symbol> is added here to prevent long lines
#             from building up - they crash the S-algol compiler.

exp1 bool:    exp2 bool;

              exp2 bool, and symbol, newline symbol, exp2 bool.

exp1 string:  exp2 string.

########
#     Expressions - level 2
########

#     S-algol says:
#
#             <exp2> ::= [~]<exp3>[<rel_op><exp3>]
#
#     which permits
#
#             ~ 0 = 0
#
#     but this generates a type error.
#     To avoid this we add round brackets.

exp2 void:    exp3 void.

exp2 int:     exp3 int.

exp2 real:    exp3 real.

exp2 bool:    exp3 bool;

              not op, exp3 bool;

              exp3 int, eq op, exp3 int;

              exp3 real, eq op, exp3 real;

              exp3 bool, eq op, exp3 bool;

              exp3 string, eq op, exp3 string;

              exp3 int, compar op, exp3 int;
              exp3 real, compar op, exp3 real;

              exp3 string, compar op, exp3 string;

              not op, clause l, exp3 int, eq op, exp3 int, clause r;

              not op, clause l, exp3 real, eq op, exp3 real, clause r;

              not op, clause l, exp3 bool, eq op, exp3 bool, clause r;

              not op, clause l, exp3 string, eq op, exp3 string, clause r;

              not op, clause l, exp3 int, compar op, exp3 int, clause r;

              not op, clause l, exp3 real, compar op, exp3 real, clause r;

              not op, clause l, exp3 string, compar op, exp3 string, clause
r.

exp2 string:  exp3 string.

not op:              space symbol, tilde symbol.

eq op:        space symbol, equals symbol;

              tilde equals symbol.

compar op:    space symbol, angle l symbol;

              angle l equals symbol;

              space symbol, angle r symbol;

              angle r equals symbol.

########
#      Expressions - level 3
########

exp3 void:    exp4 void.

exp3 int:     exp4 int;

              exp4 int, add op, exp4 int.

exp3 real:    exp4 real;

              exp4 real, add op, exp4 real;

              exp4 int, add op, exp4 real;

              exp4 real, add op, exp4 int.

exp3 bool:    exp4 bool.

exp3 string:  exp4 string.

#             <new line symbol> is added here to prevent long lines
#             from building up - they crash the S-algol compiler.
add op:              space symbol, plus symbol, newline symbol;

              space symbol, hyphen symbol, newline symbol.

########
#      Expressions - level 4
########

#      S-algol defines
```

```
#
#             <exp4> ::= <exp5>[<mult_op><exp5>]*
#             <exp5> ::= <[add_op>]<exp6>
#
#       which produces
#
#             4 * + 6
#
#       which gives rise to a precedence error.
#
#       So I have redefined exp4 in terms of exp6.
#
#       Division (div, rem, /) uses protected procedure calls.

exp4 void:    exp5 void.

exp4 int:     exp5 int;

              proc div symbol, clause l,
                     exp5 int, clause separator,
                     exp6 int, clause r;

              proc rem symbol, clause l,
                     exp5 int, clause separator,
                     exp6 int, clause r;

              exp5 int, mult op int, exp6 int.

exp4 real:    exp5 real;

              proc slash symbol, clause l,
                     exp5 real, clause separator,
                     exp6 real, clause r;

              exp5 real, mult op real, exp6 real.

exp4 bool:    exp5 bool.

exp4 string:  exp5 string;

              exp5 string, mult op string, exp5 string.

mult op int:  space symbol, asterisk symbol.

#             div symbol;

#             rem symbol.

mult op real: space symbol, asterisk symbol.

#             space symbol, slash symbol.

mult op string:     concat symbol.

########
#      Expressions - level 5
########

exp5 void:    exp6 void.

exp5 int:     exp6 int;

              add op, exp6 int.

exp5 real:    exp6 real;

              add op, exp6 real.

exp5 bool:    exp6 bool.
```

```
exp5 string:  exp6 string.

########
#       Expressions - level 6
########

#       Substring (x|y) uses a protected procedure call.

exp6 void:    sequence l, sequence r;

              clause l, clause void, clause r;

              sequence l, sequence void, sequence r.

exp6 int:     literal int;

              clause l, clause int, clause r;

              sequence l, sequence int, sequence r;

              appl int;

              id int;

              id int c.

exp6 real:    literal real;

              clause l, clause real, clause r;

              sequence l, sequence real, sequence r;

              appl real;

              id int;

              id int c;

              id real;

              id real c.

exp6 bool:    literal bool;

              clause l, clause bool, clause r;

              sequence l, sequence bool, sequence r;

              appl bool;

              id bool.

exp6 string:  literal string;

              clause l, clause string, clause r;

              sequence l, sequence string, sequence r;

              proc substr symbol, clause l,
                     exp0 string, clause separator,
                     exp0 int, clause separator,
                     exp0 int, clause r;

              appl string;

              id string.

clause l:     space symbol, round l symbol.
clause r:     space symbol, round r symbol.

sequence l:   space symbol, curly l symbol.

sequence r:   space symbol, curly r symbol.

######## ########
#       Literals
######## ########

#       The use of space must be made explicit for literal int.
#       Allowing an <add_op> in a literal int can lead to
#
#               5 * -6
#
#       which is illegal, so I have removed this option.
#       Negative literals are already provided for by exp5 (qv).
#       This requires an explicit add op before the exponent of
#       a scientififc notation real.

#       These are minor language glitches which have come to
#       light - they are not significant for GP.

literal int:  space symbol, numeral 0 symbol;

              space symbol, digits.

literal real: space symbol, numeral 0 symbol, period symbol, numeral 0
symbol;

              literal int, period symbol;

              literal int, period symbol, digits;

              literal int, period symbol, digits, letter lower e symbol,
digits;

              literal int, period symbol, digits, letter lower e symbol, add
op, digits.

literal bool: true symbol;

              false symbol.

literal string:     space symbol, quote symbol, quote symbol;

              space symbol, quote symbol, chars, quote symbol.

chars:        character;

              character, chars.

#       char is renamed character to avoid conflict with Java char.

character:    ascii;

              special.

ascii:        letter;

              digit;

              punctuation.

letter:             letter lower a symbol; letter lower b symbol; letter
lower c symbol;
              letter lower d symbol; letter lower e symbol; letter lower f
symbol;
```

```
              letter lower g symbol; letter lower h symbol; letter lower i
symbol;
              letter lower j symbol; letter lower k symbol; letter lower l
symbol;
              letter lower m symbol; letter lower n symbol; letter lower o
symbol;
              letter lower p symbol; letter lower q symbol; letter lower r
symbol;
              letter lower s symbol; letter lower t symbol; letter lower u
symbol;
              letter lower v symbol; letter lower w symbol; letter lower x
symbol;
              letter lower y symbol; letter lower z symbol; letter upper a
symbol;
              letter upper b symbol; letter upper c symbol; letter upper d
symbol;
              letter upper e symbol; letter upper f symbol; letter upper g
symbol;
              letter upper h symbol; letter upper i symbol; letter upper j
symbol;
              letter upper k symbol; letter upper l symbol; letter upper m
symbol;
              letter upper n symbol; letter upper o symbol; letter upper p
symbol;
              letter upper q symbol; letter upper r symbol; letter upper s
symbol;
              letter upper t symbol; letter upper u symbol; letter upper v
symbol;
              letter upper w symbol; letter upper x symbol; letter upper y
symbol;
              letter upper z symbol.

digits:              digit;

              digit, digits.

digit:        numeral 0 symbol; numeral 1 symbol; numeral 2 symbol; numeral
3 symbol; numeral 4 symbol;
              numeral 5 symbol; numeral 6 symbol; numeral 7 symbol; numeral
8 symbol; numeral 9 symbol.

punctuation:  space symbol; exclamation symbol; hash symbol; dollar symbol;
              percent symbol; ampersand symbol; round l symbol; round r
symbol;
              asterisk symbol; plus symbol; comma symbol; hyphen symbol;
              period symbol; slash symbol; colon symbol; semicolon symbol;
              angle l symbol; equals symbol; angle r symbol; question
symbol;
              at symbol; square l symbol; backslash symbol; square r symbol;
              caret symbol; underscore symbol; apostrophe l symbol; curly l
symbol;
              bar symbol; curly r symbol; tilde symbol.

special:      apostrophe r symbol, special follow.

special follow:      letter lower n symbol;
              letter lower p symbol;
              letter lower o symbol;
              letter lower t symbol;
              letter lower b symbol;
              apostrophe r symbol;
              quote symbol.

######## ########
#       Standard identifiers
######## ########

#       Standard identifiers are no different to programmer-declared ones,
#       except that they are in the root grammar.
```

```
id int:             variable int symbol;

             real width symbol;

             string width symbol;

             int width symbol.

id int c:    maxint symbol.

id real:     variable real symbol.

id real c:   epsilon symbol;

             pi symbol;

             maxreal symbol.

id bool:     variable bool symbol.

id string:   variable string symbol.

######## ########
#     Standard procedures
######## ########

#     Procedures floor and ceiling are enhancements.

appl void:          appl proc type void.

appl int:           appl proc type int;

                    appl abs;

                    appl decode;

                    appl find substr;

                    appl length;

                    appl random;

                    appl truncate.

appl real:          appl proc type real;

                    appl atan;

                    appl cos;

                    appl exp;

                    appl ln;

                    appl rabs;

                    appl sin;

                    appl sqrt.

appl bool:          appl proc type bool;

                    appl digit;

                    appl letter.

appl string:        appl proc type string;
```

```
                        appl code;

                        appl iformat.

appl abs:
                proc abs symbol,
                        clause l, exp0 int, clause r.

appl atan:
                proc atan symbol,
                        clause l, exp0 real, clause r.

appl ceiling:
                proc ceiling symbol,
                        clause l, exp0 real, clause r.

appl code:
                proc code symbol,
                        clause l, exp0 int, clause r.

appl cos:
                proc cos symbol,
                        clause l, exp0 real, clause r.

appl decode:
                proc decode symbol,
                        clause l, exp0 string, clause r.

appl digit:
                proc digit symbol,
                        clause l, exp0 string, clause r.

appl exp:
                proc exp symbol,
                        clause l, exp0 real, clause r.

appl find substr:
                proc find substr symbol,
                        clause l, exp0 string, clause separator, exp0 string,
clause r.

appl floor:
                proc floor symbol,
                        clause l, exp0 real, clause r.

appl iformat:
                proc iformat symbol,
                        clause l, exp0 int, clause r.

appl length:
                proc length symbol,
                        clause l, exp0 string, clause r.

appl letter:
                proc letter symbol,
                        clause l, exp0 string, clause r.

appl ln:
                proc ln symbol,
                        clause l, exp0 real, clause r.

appl rabs:
                proc rabs symbol,
                        clause l, exp0 real, clause r.

appl random:
                proc random symbol,
                        clause l, exp0 int, clause r.
```

```
 appl sin:
              proc sin symbol,
                     clause l, exp0 real, clause r.

 appl sqrt:
              proc sqrt symbol,
                     clause l, exp0 real, clause r.

 appl truncate:
              proc truncate symbol,
                     clause l, exp0 real, clause r.

 ######## ########
 #      Phenotype
 ######## ########

 #      There is one subsection per problem.
 #
 #      In each subsection, there is a production with LHS
 #      of the form
 #
 #             phenotype _problem_:
 #
 #      followed by zero or more productions which redefine
 #      notions that were already defined in the main body of
 #      the grammar above.
 #
 #      To choose problem X, first check that the production
 #      for _program_ is _program X_ at the head of this file.
 #
 #      Then uncomment all productions in the problem X subsection
 #      below.
 #
 #      Finally, in each of the other problem subsections,
 #      comment all productions except the first (ie except the
 #      production with LHS of the form:
 #
 #             phenotype _problem_:

########
#      Monkey
########

phenotype monkey:
             phenotype monkey begin symbol,
             literal string,
             phenotype monkey end symbol.

########
#      Cart
########

phenotype cart:
             phenotype cart begin symbol,
             exp3 real,
             phenotype cart end symbol.

#@clause real:       exp0 real.
#@
#@exp3 real:
#@           exp4 real;
#@           exp4 real, add op, exp4 real.
#@
#@exp6 real:
#@           literal real;
#@           appl real;
#@           id real c;
#@           clause l, clause real, clause r.
#@
```

```
#@literal real:
#@          space symbol, round l symbol, hyphen symbol, numeral 1 symbol,
round r symbol.
#@
#@appl real:
#@          appl rabs.
#@
#@id real c:
#@          space symbol, letter lower x symbol;
#@          space symbol, letter lower v symbol.

########
#     Tile
########

phenotype tile:
          phenotype tile begin symbol,
          exp0 int,
          phenotype tile end symbol.

#     Redefine clause int and clause real to avoid sequences.

#@clause int: exp0 int.
#@
#@clause real:     exp0 real.
#@
#@exp5 int:  exp6 int.
#@
#@exp6 int:
#@          literal int;
#@
#@          clause l, clause int, clause r;
#@
#@          appl int.
#@
#@exp6 real:
#@          literal real;
#@
#@          clause l, clause real, clause r;
#@
#@          appl real;
#@
#@          id real c.
#@
#@appl int:
#@          appl abs;
#@
#@          appl floor;
#@
#@          appl ceiling.
#@
#@appl real:
#@          appl rabs;
#@
#@          appl sqrt.
#@id real c:
#@          space symbol, letter lower x symbol;
#@          space symbol, letter lower y symbol.

########
#     Multiplexer
########

phenotype multiplexer:
          phenotype multiplexer begin symbol,
          exp0 bool,
          phenotype multiplexer end symbol.

#@exp2 bool: exp3 bool;
```

```
#@
#@             not op, exp3 bool;
#@
#@             exp3 bool, eq op, exp3 bool;
#@
#@             not op, clause l, exp3 bool, eq op, exp3 bool, clause r.
#@
#@exp6 bool:   literal bool;
#@
#@             clause l, clause bool, clause r;
#@
#@             id bool c.
#@
#@id bool c:
#@             space symbol, letter lower a symbol, numeral 2 symbol;
#@             space symbol, letter lower a symbol, numeral 1 symbol;
#@             space symbol, letter lower a symbol, numeral 0 symbol;
#@             space symbol, letter lower d symbol, numeral 7 symbol;
#@             space symbol, letter lower d symbol, numeral 6 symbol;
#@             space symbol, letter lower d symbol, numeral 5 symbol;
#@             space symbol, letter lower d symbol, numeral 4 symbol;
#@             space symbol, letter lower d symbol, numeral 3 symbol;
#@             space symbol, letter lower d symbol, numeral 2 symbol;
#@             space symbol, letter lower d symbol, numeral 1 symbol;
#@             space symbol, letter lower d symbol, numeral 0 symbol.

########
#      Power
########

phenotype power:
             phenotype power begin symbol,
             sequence void, sequence separator,
             clause real,
             phenotype power end symbol.

#@sequence void:
#@             clause void;
#@
#@             clause void, sequence separator, sequence void;
#@
#@             decl let;
#@
#@             decl let, sequence separator, sequence void.
#@
#@decl let:
#@             decl let int;
#@
#@             decl let real.
#@
#@clause void:
#@             exp0 void;
#@
#@             for symbol, id int new, space symbol, equals symbol, clause
int,
#@             to symbol, clause int,
#@             do symbol, clause void;
#@
#@             id int, assignment symbol, clause int;
#@
#@             id real, assignment symbol, clause real.
#@
#@clause int:
#@             exp0 int.
#@
#@clause real:
#@             exp0 real.
#@
#@exp6 int:
```

```
#@              literal int;
#@
#@              clause l, clause int, clause r;
#@
#@              id int.
#@
#@exp6 real:
#@              literal real;
#@
#@              clause l, clause real, clause r;
#@
#@              id int;
#@
#@              id real.
#@
#@id int:
#@              space symbol, letter lower n symbol.
#@
#@id real:
#@              space symbol, letter lower x symbol.

########
#       Two Box 1 (Koza style: without ADF).
########

#       The Two Box 1 phenotype is:
#               (1) A real clause involving only:
#                       variables       L0, W0, H0, L1, W1 and H1
#                       operators       +, -, * and SLASH

phenotype two box 1:
                phenotype two box begin symbol,
                clause real,
                phenotype two box end symbol.

#@clause real:
#@              exp0 real.
#@
#@exp3 real:
#@              exp4 real;
#@              exp4 real, add op, exp4 real.
#@
#@exp5 real:
#@              exp6 real.
#@
#@exp6 real:
#@              id real c;
#@              clause l, clause real, clause r.
#@
#@id real c:
#@              space symbol, letter upper l symbol, numeral 0 symbol;
#@              space symbol, letter upper w symbol, numeral 0 symbol;
#@              space symbol, letter upper h symbol, numeral 0 symbol;
#@              space symbol, letter upper l symbol, numeral 1 symbol;
#@              space symbol, letter upper w symbol, numeral 1 symbol;
#@              space symbol, letter upper h symbol, numeral 1 symbol.

########
#       Two Box 2 (Koza style: with ADF).
########

#       The Two Box 2 phenotype is:
#               (1) A non-recursive procedure of type
#                       (real, real, real -> real)
#               (2) A real clause involving only:
#                       seed procedure       PROC.REAL
#                       variables       L0, W0, H0, L1, W1 and H1
#                       operators       +, -, * and SLASH
#                       procedure       as declared above
```

```
#
#       Assignment is not supported.
#
#       The non-recursive nature is controlled
#       by the choice of production method.

phenotype two box 2:
            phenotype two box begin symbol,
            sequence void,
            phenotype two box end symbol.

#@sequence void:
#@          decl proc, sequence separator, clause real.
#@
#@decl proc:
#@          decl proc real.
#@
#@decl proc real:
#@          proc symbol, id proc new,
#@          round l symbol, parameter list 3, arrow symbol, type real
symbol, round r symbol,
#@          sequence separator,
#@          clause real.
#@
#@#    The odd way to get exactly three parameters means
#@#    we can use the existing production methods.
#@
#@parameter list 3:
#@          parameter, parameter separator, parameter list 2.
#@
#@parameter list 2:
#@          parameter, parameter separator, parameter list 1.
#@
#@parameter list 1:
#@          parameter.
#@
#@parameter:
#@          type real symbol, id real new.
#@
#@clause real:
#@          exp0 real.
#@
#@exp3 real:
#@          exp4 real;
#@          exp4 real, add op, exp4 real.
#@
#@exp5 real:  exp6 real.
#@
#@#    <id real> is 1st alternative to avoid default loop.
#@
#@exp6 real:
#@          id real;
#@          appl real;
#@          clause l, clause real, clause r.
#@
#@#    <id real> must exist so that <parameter> can extend it
#@#    with formal arguments.  But there is no assignment.
#@
#@id real:
#@          space symbol, letter upper l symbol, numeral 0 symbol;
#@          space symbol, letter upper w symbol, numeral 0 symbol;
#@          space symbol, letter upper h symbol, numeral 0 symbol;
#@          space symbol, letter upper l symbol, numeral 1 symbol;
#@          space symbol, letter upper w symbol, numeral 1 symbol;
#@          space symbol, letter upper h symbol, numeral 1 symbol.
#@
#@#    <appl real> must exist so that <decl proc> can extend it
#@#    with the defined ADF.  It is given a harmless initial value.
#@
```

```
#@appl real:
#@              appl proc type real.

########
#       Two Box 3 (Paterson style: with unprescribed, non-recursive ADF).
########

#       The Two Box 3 phenotype is:
#               (1) A non-recursive procedure of type
#                       (real* -> real)
#               where real* means zero or more reals.  For example:
#                       ( -> real)
#                       (real, real, real -> real)
#               (2) A real clause involving only:
#                       seed procedure       PROC.REAL
#                       variables     L0, W0, H0, L1, W1 and H1
#                       operators     +, -, * and SLASH
#                       procedure     as declared above

phenotype two box 3:
                phenotype two box begin symbol,
                sequence void,
                phenotype two box end symbol.

#@sequence void:
#@              decl proc, sequence separator, clause real.
#@
#@decl proc:
#@              decl proc real.
#@
#@parameter:
#@              type real symbol, id real new.
#@
#@clause real:
#@              exp0 real.
#@
#@exp3 real:
#@              exp4 real;
#@              exp4 real, add op, exp4 real.
#@
exp5 real:      exp6 real.
#@
#@#     <id real> is 1st alternative to avoid default loop.
#@
#@exp6 real:
#@              id real;
#@              appl real;
#@              clause l, clause real, clause r.
#@
#@#     <id real> must exist so that <parameter> can extend it
#@#     with formal arguments.  But there is no assignment.
#@
#@id real:
#@              space symbol, letter upper l symbol, numeral 0 symbol;
#@              space symbol, letter upper w symbol, numeral 0 symbol;
#@              space symbol, letter upper h symbol, numeral 0 symbol;
#@              space symbol, letter upper l symbol, numeral 1 symbol;
#@              space symbol, letter upper w symbol, numeral 1 symbol;
#@              space symbol, letter upper h symbol, numeral 1 symbol.
#@
#@#     <appl real> must exist so that <decl proc> can extend it
#@#     with the defined ADF.  It is given a harmless initial value.
#@
#@appl real:
#@              appl proc type real.

########
#       Two Box 4 (Paterson style: with unprescribed, recursive ADF).
########
```

```
#       The Two Box 4 phenotype is:
#               (1) A possibly recursive procedure of type
#                       (real* -> real)
#               where real* means zero or more reals.  For example:
#                       ( -> real)
#                       (real, real, real -> real)
#               (2) A real clause involving only:
#                       seed procedure       PROC.REAL
#                       variables       L0, W0, H0, L1, W1 and H1
#                       operators       +, -, * and SLASH
#                       procedure       as declared above

phenotype two box 4:
                phenotype two box begin symbol,
                decl proc, sequence separator, clause real,
                phenotype two box end symbol.

#       The productions for Two Box 4 are the same as for Two Box 3.
#       The difference is in the RAGInitializer.suffix.

########
#       Fact.
########

phenotype fact:
                phenotype fact begin symbol,
                sequence void,
                phenotype fact end symbol.

sequence void:
                decl proc, sequence separator, clause real.

#       Procedure: (int -> real).
decl proc:
                decl proc real.

decl proc real:
                proc symbol, id proc new,
                round l symbol, parameter list, arrow symbol, type real
symbol, round r symbol,
                sequence separator,
                clause real.

#       Simple ints only.
clause int:     exp0 int.

#       Proc has just one int parameter.
parameter list:     parameter.

parameter:
                type int symbol, id int new.

#       Remove references to strings.
exp2 bool:
                exp3 bool;
                not op, exp3 bool;
                exp3 int, eq op, exp3 int;
                exp3 real, eq op, exp3 real;
                exp3 int, compar op, exp3 int;
                exp3 real, compar op, exp3 real;
                not op, clause l, exp3 int, eq op, exp3 int, clause r;
                not op, clause l, exp3 real, eq op, exp3 real, clause r;
                not op, clause l, exp3 int, compar op, exp3 int, clause r;
                not op, clause l, exp3 real, compar op, exp3 real, clause r.

exp5 real:      exp6 real.

#       <id real> is 1st alternative to avoid default loop.
```

```
exp6 int:       literal int;
                clause l, clause int, clause r;
                id int.

exp6 real:
                id int;
                id real;
                appl real;
                clause l, clause real, clause r.

#       <id int> must exist so that <parameter> can extend it
#       with formal arguments.  But there is no assignment.

id int:
                space symbol, letter lower n symbol.

#       <appl real> must exist so that <decl proc> can extend it
#       with the defined ADF.  It is given a harmless initial value.

appl real:
                appl proc type real.

########
#       Annie.
########

phenotype annie:
                phenotype annie begin symbol,
                sequence void,
                phenotype annie end symbol.

######## ######## ######## ######## ######## ######## ######## ########
#       Representation table
######## ######## ######## ######## ######## ######## ######## ########

Symbols

######## ######## ######## ########
#       Wrapper
######## ######## ######## ########

#       The wrapper consists of:
#
#       preamble symbol
#               Contains the code for implementing Rag, for
#               protected S-algol operations, and for common evaluation
#               procedures.  The preamble is common to all problems,
#               though not all of it is used by all problems.
#
#       phenotype <problem> begin symbol
#       phenotype <problem> end symbol
#               The phenotype begin and end symbols contain comments
#               to help locate the phenotype in the wrapper.
#
#               All problems except Annie are wrapped in a procedure,
#               whose prototype varies from problem to problem.  The
#               begin and end symbols contain the problem-specific
#               procedure prototype and {} to enclose the procedure body,
#               which is the actual phenotype.
#
#       postamble <problem> symbol
#               The postamble is problem-specific.  It has whatever is
#               needed to evaluate the phenotype and output its
#               functionality.  For Annie, the phenotype is responsible
#               for everything, so the postamble is just a rump.

preamble symbol:                %
let DEBUG     = false        ! Control debug output.
```

```
! Seed variables.
let INT := 0
let REAL := 0.0
let BOOL := false
let STRING := ""

! Seed procedures.
procedure PROC.VOID
       {}
procedure PROC.INT(int x -> int)
       {x}
procedure PROC.REAL(real x -> real)
       {x}
procedure PROC.BOOL(bool x -> bool)
       {x}
procedure PROC.STRING(string x -> string)
       {x}

! Protected procedures.
procedure DECODE(string s -> int)
       if s = "" then 0 else decode(s)
procedure DIV(int n, d -> int)
       if d = 0 then 0 else n div d
procedure REM(int n, d -> int)
       if d = 0 then 0 else n rem d
procedure SLASH(real n, d -> real)
       if rabs(d) < 1e-6 then 0 else n / d
procedure SUBSTR(string s; int i, j -> string) ! Java indexing semantics.
       if 0 <= i and i < j and j <= length(s) then s(i+1|j-i) else ""

! Enhancements to S-algol.
procedure FLOOR (real x -> int)
       truncate(x)

procedure CEILING (real x -> int)
       {
       let t = truncate(x)
       if x = t then t else t+1
       }

! Evaluation procedures.
let Seed := 635547864
procedure random.real (-> real)   ! Returns uniform in (0.0, 1.0).
       {
       Seed := random (Seed)
       Seed/maxint
       }

let LOG = true          ! Named constants ...
let REV = true          ! ... to make call to map ...
let SCA = true          ! ... easier to understand.
let MAX.FUNCTIONALITY = 9999       ! Functionality is in [0,
MAX.FUNCTIONALITY].
let LOG.MAXINT = ln(maxint)        ! Useful constant.
let LOG.MAXREAL = ln(maxreal)      ! Useful constant.
procedure map (bool log, rev, sca; real in.min, in.max, in.obs -> real); {

       if DEBUG do write "MAP: log: ", log, "; rev: ", rev, "; sca: ", sca,
"; in.min: ", in.min, "; in.max: ", in.max, "; in.obs: ", in.obs, ")'n"

       ! Prepare the output observation.
       let out.obs := in.obs

       ! Convert to log scale.
       ! out.obs in [log.in.min, log.in.max]
       if log do {
              if in.min = 0 do {write "***** map: FP exception: log (0):
in.min.'n"; abort}
```

```
                if in.max = 0 do {write "***** map: FP exception: log (0):
in.max.'n"; abort}
                if out.obs = 0 do {write "***** map: FP exception: log (0):
out.obs.'n"; abort}
                in.min := ln (in.min)
                in.max := ln (in.max)
                out.obs := ln (out.obs)
                if DEBUG do write "LOG: in.min: ", in.min, "; in.max:",
in.max, "; out.obs: ", out.obs, "'n"
        }

        ! Reverse the sense of the scale.
        ! out.obs in [in.min, in.max]
        if rev do {
                out.obs := in.min + in.max - out.obs
                if DEBUG do write "REV: out.obs: ", out.obs, "'n"
        }

        ! Apply a scaling factor.
        if sca do {
                out.obs := (out.obs - in.min)/(in.max-
in.min)*MAX.FUNCTIONALITY
                if DEBUG do write "SCA: out.obs: ", out.obs, "'n"
        }

        ! Trim and cap result.
        out.obs := truncate (out.obs+0.5)
        if out.obs < 0 do out.obs := 0
        if out.obs > MAX.FUNCTIONALITY do out.obs := MAX.FUNCTIONALITY

        ! Return.
        out.obs
}

! Set up IO environment.
i.w := 0
s.w := 0
r.w := 0

%

######## ########
#       Monkey
######## ########

phenotype monkey begin symbol:                  %
procedure observed (-> string)
{
!<<<<< phenotype begins
%

phenotype monkey end symbol:                    %
!>>>>> phenotype ends
}
%

postamble monkey symbol:                %
! Evaluation parameters.
let BANANA = "Hello, world!'n"

! Ideal phenotype.
procedure expected (-> string)
BANANA

! Compute RMS difference between Ascii of observed and expected.
procedure RMS (string obs, exp -> real)
        {
        if DEBUG do write "obs, exp: '"", obs, "'", '"", exp, "'"'n"
        let rms := 0.0
        let s := length(obs); if length(exp) > length(obs) do s :=
length(exp)
        for i = 0 to s-1 do
                {
                let o = SUBSTR(obs,i,i+1)
                let e = SUBSTR(exp,i,i+1)
                let d = DECODE(o)-DECODE(e)
                if DEBUG do write "o, e, d: '"", o, "'", '"", e, "'", ", d,
"'n"
                rms := rms + d*d
                }
        rms := sqrt(rms/s)
        if DEBUG do write "rms: ", rms, "'n"
        rms
}

! Write the functionality.
write map (-LOG, REV, SCA, 0, RMS("", expected), RMS(observed, expected)),
"'n"
%

######## ########
#       Cart
######## ########

phenotype cart begin symbol:                    %
procedure observed (real x, v -> real)
{
!<<<<< phenotype begins
%

phenotype cart end symbol:              %
!>>>>> phenotype ends
}
%

postamble cart symbol:                  %
! Evaluation parameters.
let CASES = 20          ! Number of test cases.
let TAU = 0.02          ! Time step.
let RAD = 0.1 ! Target radius in (x, v) space
let EOT = 10  ! Time limit in simulated seconds

! Ideal phenotype.
procedure expected (real x, v -> real)
-x - v*rabs(v)

! Simulate cart.
procedure time.to.center(real x, v; (real, real -> real) control -> real)
        {
        let t := 0.0
        while t < EOT and x*x+v*v > RAD * RAD do
                {
                let a = if control (x, v) < 0 then -0.5 else +0.5
                t := t + TAU
                x := x + TAU * v + TAU*TAU*a/2
                v := v + TAU * a
                }
        t
        }

! Compute RMS difference between observed and expected.
let rms := 0.0
if DEBUG do write "x'tv'to'te'td'n"
for i = 1 to CASES do
        {
        let x = random.real*1.5-0.75
        let v = random.real*1.5-0.75
        let o = time.to.center (x, v, observed)
        let e = time.to.center (x, v, expected)
        let d = o - e
        if DEBUG do write x, "'t", v, "'t", o, "'t", e, "'t", d, "'n"
        rms := rms + d*d
        }
rms := sqrt(rms/CASES)
if DEBUG do write "rms = ", rms, "'n"

! Write the functionality.
write map (-LOG, REV, SCA, 0, EOT, rms), "'n"
%

######## ########
#       Tile
######## ########

phenotype tile begin symbol:                    %
procedure observed (real x, y -> int)
{
!<<<<< phenotype begins
%

phenotype tile end symbol:              %
!>>>>> phenotype ends
}
%

postamble tile symbol:                  %
! Evaluation parameters.
let CASES = 20          ! Number of test cases.

! Ideal phenotype.
procedure expected (real x, y -> int)
{
        let t := 0
        for i = 0 to FLOOR(x) do {
                t := t + CEILING(y-i/x*y)
        }
        t
}

! Compute RMS of difference between observed and expected.
let rms := 0.0
if DEBUG do write "x'ty'to'te'td'n"
for i = 1 to CASES do
        {
        let x = truncate(random.real*100)+1
        let y = truncate(random.real*100)+1
        let o = 0.0+observed (x, y)        ! Use real to avoid int overflow
        let e = 0.0+expected (x, y)        ! Use real to avoid int overflow
        let d = o - e
        if DEBUG do write x, "'t", y, "'t", o, "'t", e, "'t", d, "'n"
        rms := rms + d*d
        }
rms := sqrt(rms/CASES)
if DEBUG do write "rms = ", rms, "'n"

! Write the functionality.
write map (LOG, REV, SCA, 1, maxint, rms+1), "'n"
%

######## ########
#       Multiplexer
######## ########

phenotype multiplexer begin symbol:                     %
procedure observed (bool a2, a1, a0, d7, d6, d5 ,d4 ,d3, d2, d1, d0 -> bool)
{
!<<<<< phenotype begins
```

```
%

phenotype multiplexer end symbol:                    %
!>>>>> phenotype ends
}
%

postamble multiplexer symbol:                        %
! Evaluation parameters.
let CASES = 2048     ! Number of test cases.

! Ideal phenotype.
procedure expected (bool a2, a1, a0, d7, d6, d5 ,d4 ,d3, d2, d1, d0 -> bool)
{
        if a2
        then    if a1
                then    if a0
                        then    d7
                        else    d6
                else    if a0
                        then    d5
                        else    d4
        else    if a1
                then    if a0
                        then    d3
                        else    d2
                else    if a0
                        then    d1
                        else    d0
}


! Compute raw functionality = count of correct results in all cases
let f := 0    ! Functionality = count of correct results.
let a2 := false
let a1 := false
let a0 := false
let d7 := false
let d6 := false
let d5 := false
let d4 := false
let d3 := false
let d2 := false
let d1 := false
let d0 := false
if DEBUG do write "e'to'tf'n"
for c = 1 to CASES do
        {
        if DEBUG do write "Case ", c, "'n"
        let i := c-1; a2 := i rem 2 = 0
        i := i div 2; a1 := i rem 2 = 0
        i := i div 2; a0 := i rem 2 = 0
        i := i div 2; d7 := i rem 2 = 0
        i := i div 2; d6 := i rem 2 = 0
        i := i div 2; d5 := i rem 2 = 0
        i := i div 2; d4 := i rem 2 = 0
        i := i div 2; d3 := i rem 2 = 0
        i := i div 2; d2 := i rem 2 = 0
        i := i div 2; d1 := i rem 2 = 0
        i := i div 2; d0 := i rem 2 = 0
        let o := observed (a2, a1, a0, d7, d6, d5 ,d4 ,d3, d2, d1, d0)
        let e := expected (a2, a1, a0, d7, d6, d5 ,d4 ,d3, d2, d1, d0)
        if e = o do f := f + 1
        if DEBUG do write e, "'t", o, "'t", f, "'n"
        }

! Write the functionality.
write map (~LOG, ~REV, SCA, 0, CASES, f), "'n"
%
```

```
######## ########
#       Power
######## ########

phenotype power begin symbol:                             %
procedure observed (real x; int n -> real)
{
! Re-initialise seed variables
INT := 0
REAL := 0.0
BOOL := false
STRING := ""
!<<<<< phenotype begins
%

phenotype power end symbol:                               %
!>>>>> phenotype ends
}
%

postamble power symbol:                        %
! Evaluation parameters.
let CASES = 30
let MAX.X = 10          ! x is in (0, MAX.X).
let MAX.N = 10          ! n is in [0, MAX.X].

! Ideal phenotype.
procedure expected (real x; int n -> real)
{
        let result := 1.0
        for i = 1 to n do
                result := result * x
        result
}

! Compute RMS difference beteen observed and expected.
let rms := 0.0
if DEBUG do write "c'tx'tn'to'te'td'n"
for c = 1 to CASES do
        {
        let x = random.real*MAX.X
        let n = truncate(random.real*MAX.N) + 1
        let o = observed (x, n)
        let e = expected (x, n)
        let d = o - e
        rms := rms + d * d
        if DEBUG do write c, "'t", x, "'t", n, "'t", o, "'t", e, "'t", d,
"'n"
        }
rms := sqrt(rms/CASES)
if DEBUG do write "rms ", rms, "'n"

! Write the functionality.
write map (LOG, REV, SCA, 1, maxreal, rms+1), "'n"
%

######## ########
#       Two Box
######## ########

phenotype two box begin symbol:                           %
procedure observed (real L0, W0, H0, L1, W1, H1 -> real)
{
!<<<<< phenotype begins
%

phenotype two box end symbol:                             %
!>>>>> phenotype ends
```

```
}
%

postamble two box symbol:                %
! Evaluation parameters.
let CASES = 10
let RANGE = 10        ! Each dimension is in [0, RANGE].

! Ideal phenotype.
procedure expected (real L0, W0, H0, L1, W1, H1 -> real)
{
      procedure volume (real ARG0, ARG1, ARG2 -> real)
      ARG0 * ARG1 * ARG2

      volume (L0, W0, H0) - volume (L1, W1, H1)
}

! Compute RMS difference between observed and expected
let rms := 0.0
if DEBUG do write "Case    L0     W0     H0     L1     W1     H1     Obs
      Exp    Obs-Exp'n"
for c = 1 to CASES do
      {
      ! Define fitness cases, as in [Koza, 1994].
      let L0 = truncate(random.real*RANGE) + 1
      let W0 = truncate(random.real*RANGE) + 1
      let H0 = truncate(random.real*RANGE) + 1
      let L1 = truncate(random.real*RANGE) + 1
      let W1 = truncate(random.real*RANGE) + 1
      let H1 = truncate(random.real*RANGE) + 1
      let o = observed (L0, W0, H0, L1, W1, H1)
      let e = expected (L0, W0, H0, L1, W1, H1)
      let d = o - e
      rms := rms + d * d
      if DEBUG do write c, "'t", L0, "'t", W0, "'t", H0, "'t", L1, "'t",
W1, "'t", H1, "'t", fformat(o,3,1), "'t", fformat(e,3,1), "'t",
fformat(d,3,1), "'n"
      }
rms := sqrt(rms/CASES)
if DEBUG do write "rms = ", rms, "'n"

! Write the functionality.
write map (LOG, REV, SCA, 1, maxreal, rms+1), "'n"
%

######## ########
#      Fact
######## ########

phenotype fact begin symbol:                %
procedure observed (int n -> real)
{
!<<<<< phenotype begins
%

phenotype fact end symbol:                %
!>>>>> phenotype ends
}
%

postamble fact symbol:                %
! Evaluation parameters.
let CASES = 10
let RANGE = 95        ! Largest number such that rms cannot overflow.

! Ideal phenotype.
procedure expected (int n -> real); {
      procedure factorial (int n -> real)
            if n <= 0 then 1 else factorial(n-1)*n
```

```
       factorial (n)
}

! Compute RMS difference between observed and expected
let rms := 0.0
if DEBUG do write "Case    n      o      e      d      rms'n"
for c = 1 to CASES do
       {
       let n = truncate (c / CASES * RANGE + 0.5)
       let o = ln (1+rabs(observed (n)))
       let e = ln (1+rabs(expected (n)))
       let d = o - e
       rms := rms + d * d
       if DEBUG do write c, "'t", n, "'t", o, "'t", e, "'t", d, "'t", rms,
"'n"
       }
rms := sqrt(rms/CASES)
if DEBUG do write "rms = ", rms, "'n"

! Write the functionality.
write map (~LOG, REV, SCA, 1, ln(maxreal), rms+1), "'n"
?
%

######## ########
#       Annie
######## ########

phenotype annie begin symbol:                    %
!<<<<< phenotype begins
%

phenotype annie end symbol:                      %
!>>>>> phenotype ends
%

postamble annie symbol:                   %
%

######## ######## ######## ########
#       Preamble symbols
######## ######## ######## ########

variable int symbol:                      " INT"
variable real symbol:                     " REAL"
variable bool symbol:                     " BOOL"
variable string symbol:                   " STRING"

proc type void symbol:                    " PROC.VOID"

proc type int symbol:                     " PROC.INT"

proc type real symbol:                    " PROC.REAL"

proc type bool symbol:                    " PROC.BOOL"

proc type string symbol:            " PROC.STRING"

proc decode symbol:                 " DECODE"

proc div symbol:                    " DIV"

proc rem symbol:                    " REM"

proc slash symbol:                  " SLASH"

proc substr symbol:                 " SUBSTR"
```

```
######## ######## ######## ########
#       Reserved words
######## ######## ######## ########

and symbol:                     " and"
by symbol:                      " by"
div symbol:                     " div"
do symbol:                      " do"
else symbol:                    " else"
false symbol:                   " false"
for symbol:                     " for"
if symbol:                      " if"
let symbol:                     " let"
or symbol:                      " or"
proc symbol:                    " procedure"
rem symbol:                     " rem"
then symbol:                    " then"
true symbol:                    " true"
to symbol:                      " to"
type int symbol:                " int"
type bool symbol:               " bool"
type real symbol:               " real"
type string symbol:             " string"
type void symbol:               " void"
while symbol:                   " while"
write symbol:                   " write"

######## ######## ######## ########
#       Predeclared identifiers - procedures
######## ######## ######## ########

#       Procedure decode is redefined above as a protected procedure.

proc abs symbol:                " abs"
proc atan symbol:               " atan"
proc ceiling symbol:                    " CEILING"
proc code symbol:               " code"
proc cos symbol:                " cos"
proc digit symbol:              " digit"
proc exp symbol:                " exp"
proc find substr symbol:        " find.substr"
proc floor symbol:              " FLOOR"
proc iformat symbol:                    " iformat"
proc length symbol:             " length"
proc letter symbol:             " letter"
proc ln symbol:                         " ln"
proc rabs symbol:               " rabs"
proc random symbol:             " random"
proc sin symbol:                " sin"
proc sqrt symbol:               " sqrt"
proc truncate symbol:                   " truncate"

######## ######## ######## ########
#       Predeclared identifiers - variables
######## ######## ######## ########

epsilon symbol:                         " epsilon"
int width symbol:               " i.w"
maxint symbol:                          " maxint"
maxreal symbol:                         " maxreal"
pi symbol:                      " pi"
real width symbol:              " r.w"
string width symbol:                    " s.w"

######## ######## ######## ########
#       Punctuation - composite
######## ######## ######## ########

arrow symbol:                   " ->"
```

```
assignment symbol:          " :="
concat symbol:                         " ++"
angle r equals symbol:                 " >="
angle l equals symbol:                 " <="
tilde equals symbol:                   " ~="

######## ######## ######## ########
#       Letters
######## ######## ######## ########

letter lower a symbol:                 "a"
letter lower b symbol:                 "b"
letter lower c symbol:                 "c"
letter lower d symbol:                 "d"
letter lower e symbol:                 "e"
letter lower f symbol:                 "f"
letter lower g symbol:                 "g"
letter lower h symbol:                 "h"
letter lower i symbol:                 "i"
letter lower j symbol:                 "j"
letter lower k symbol:                 "k"
letter lower l symbol:                 "l"
letter lower m symbol:                 "m"
letter lower n symbol:                 "n"
letter lower o symbol:                 "o"
letter lower p symbol:                 "p"
letter lower q symbol:                 "q"
letter lower r symbol:                 "r"
letter lower s symbol:                 "s"
letter lower t symbol:                 "t"
letter lower u symbol:                 "u"
letter lower v symbol:                 "v"
letter lower w symbol:                 "w"
letter lower x symbol:                 "x"
letter lower y symbol:                 "y"
letter lower z symbol:                 "z"
letter upper a symbol:                 "A"
letter upper b symbol:                 "B"
letter upper c symbol:                 "C"
letter upper d symbol:                 "D"
letter upper e symbol:                 "E"
letter upper f symbol:                 "F"
letter upper g symbol:                 "G"
letter upper h symbol:                 "H"
letter upper i symbol:                 "I"
letter upper j symbol:                 "J"
letter upper k symbol:                 "K"
letter upper l symbol:                 "L"
letter upper m symbol:                 "M"
letter upper n symbol:                 "N"
letter upper o symbol:                 "O"
letter upper p symbol:                 "P"
letter upper q symbol:                 "Q"
letter upper r symbol:                 "R"
letter upper s symbol:                 "S"
letter upper t symbol:                 "T"
letter upper u symbol:                 "U"
letter upper v symbol:                 "V"
letter upper w symbol:                 "W"
letter upper x symbol:                 "X"
letter upper y symbol:                 "Y"
letter upper z symbol:                 "Z"

######## ######## ######## ########
#       Numerals
######## ######## ######## ########

numeral 0 symbol:           "0"
numeral 1 symbol:           "1"
numeral 2 symbol:          "2"
numeral 3 symbol:          "3"
numeral 4 symbol:          "4"
numeral 5 symbol:          "5"
numeral 6 symbol:          "6"
numeral 7 symbol:          "7"
numeral 8 symbol:          "8"
numeral 9 symbol:          "9"

######## ######## ######## ########
#      Punctuation - simple - in Ascii order
######## ######## ######## ########

space symbol:              " "
exclamation symbol:        "!"
quote symbol:              '"'
hash symbol:               "#"
dollar symbol:                    "$"
percent symbol:                   "%"
ampersand symbol:          "&"
apostrophe r symbol:              "'"

round l symbol:                   "("
round r symbol:                   ")"
asterisk symbol:           "*"
plus symbol:               "+"
comma symbol:              ","
hyphen symbol:                    "-"
period symbol:                    "."
slash symbol:              "/"

colon symbol:              ":"
semicolon symbol:          ";"
angle l symbol:                   "<"
equals symbol:                    "="
angle r symbol:                   ">"
question symbol:           "?"

at symbol:                 "@"

square l symbol:           "["
backslash symbol:          "\"
square r symbol:           "]"
caret symbol:              "^"
underscore symbol:         "_"

apostrophe l symbol:              "`"

curly l symbol:                   "{"
bar symbol:                "|"
curly r symbol:                   "}"
tilde symbol:              "~"

newline symbol:                   "
"

######## ######## ######## ########
#      Placeholders
######## ######## ######## ########

id int new symbol:         " idIntNew"
id real new symbol:        " idRealNew"
id bool new symbol:        " idBoolNew"
id string new symbol:             " idStringNew"
id proc new symbol:        " idProcNew"
```

B.2 Production methods

The production methods are coded in the node constructor. A general Node class is presented, followed by the specialised SalgolNode.

B.2.1 Node

```
// Node - a Node in a parse tree.

// This is an abstract class.  It has to be specified for each
// language because the production methods are language-specific.
// This is not good.  It would be better to devise a virtual
// machine code for production methods in the longer term.
// Then the language-specific detail could go into the grammar,
// and a more general Node class would interpret it.

package ec.experiment.nrp.rag;
import java.io.*;
public abstract class Node {

        public Node () {
        }

        // INHERITED ATTRIBUTES

        // The Symbol at this node.
        public Symbol s = null;

        // SYNTHETIC ATTRIBUTES

        // Children of this node.
        public Node[] child = null;
        public int childCount = 0;

        // PRODUCTION METHODS

        // Apply specified production method.  This method must be
        // overridden to cater for non-default production methods.
        public void applyMethod(int m, ReflexiveAttributeGrammar RAG, Symbol
s, Symbol[] RHS, Genotype g) {
                switch (m) {
                case 0:
                        method_0 (RAG, (NonTerminal)s, RHS, g);
                        break;
                default:
                        System.out.println ("      Node: applyMethod: error:
unexpected method number: " + m);
                }
        }

        public static final int METHOD_DEFAULT = 0;
        public void method_0 (ReflexiveAttributeGrammar RAG, NonTerminal LHS,
Symbol[] RHS, Genotype g) {
                // Beginning.
                // System.out.println ("   Node: method_0: beginning.");

                // Descend into subtree.
                for (int i = 0; i < this.child.length; i++) {
                        // System.out.println ("   Node: creating child node: "
+ i);
                        // Uncomment next line with specific Node subclass.
```

```
                    // child[i] = new Node(RAG, RHS[i], g);
            }

            // Completed.
            // System.out.println ("   Node: method_0: completed.");
    }

    // CONSTRUCTORS

    public Node (ReflexiveAttributeGrammar RAG, Symbol s, Genotype g) {

            // System.out.println ("   Node: beginning symbol: " +
s.getName());

            // Note the Symbol at this node.
            this.s = s;

            // If the Symbol s is a Terminal, we are finished.
            if (s instanceof Terminal)
                    {
                    return;
                    }

            // Use Genotype g to choose a production for this NonTerminal.
            Production p = ((NonTerminal)s).getProduction(g);

            // Get the RHS of this production.
            Symbol[] RHS = p.getRHS();

            // Prepare to have children.
            // this.child = new Node[RHS.length];

            // Apply specified method.
            applyMethod(p.getMethodNumber(), RAG, s, RHS, g);

            // System.out.println ("   Node: completed symbol: " +
s.getName());
    }

    // PHENOTYPE

    public String getLeaves () {
            if (child == null) {
                    return (s.getName());
            } else {
                    String leaves = "";
                    for (int i = 0; i < this.child.length; i++) {
                            leaves = leaves + child[i].getLeaves();
                    }
                    return (leaves);
            }
    }

    public void writeLeaves (FileOutputStream fos) throws IOException {
            if (child == null) {
                    fos.write(s.getName().getBytes());
            } else {
                    for (int i = 0; i < this.child.length; i++) {
                            child[i].writeLeaves(fos);
                    }
            }
    }

    public int countLeaves () {
            if (child == null) {
                    return (1);
            } else {
                    int leaves = 0;
                    for (int i = 0; i < this.child.length; i++) {
```

```
                                leaves = leaves + child[i].countLeaves();
                        }
                        return (leaves);
                }
        }
}
```

B.2.1 S_algolNode extends Node

```
// S_algolNode - a S_algolNode in a parse tree.

package ec.experiment.nrp.rag.ecj;
import ec.experiment.nrp.rag.*;
import ec.experiment.nrp.rag.s_algol.*;
public class S_algolNode extends Node {

        public S_algolNode () {
        }

        // SYNTHETIC ATTRIBUTES

        // S_algolPrototype - synthesized by procedure declarations.
        S_algolPrototype prototype = null;

        // S_algolParameter - synthesized by _parameter_.
        S_algolParameter parameter = null;

        // S_algolParameterList - synthesized by _parameter list_.
        S_algolParameterList parameterList = null;

        // PRODUCTION METHODS

        // Apply specified production method.  This method must be
        // overridden to cater for non-default production methods.
        public void applyMethod(int m, ReflexiveAttributeGrammar RAG, Symbol
s, Symbol[] RHS, Genotype g) {
                switch (m) {
                case 0:
                        method_0 (RAG, (NonTerminal)s, RHS, g);
                        break;
                case 1:
                        method_1 (RAG, (NonTerminal)s, RHS, g);
                        break;
                case 2:
                        method_2 (RAG, (NonTerminal)s, RHS, g);
                        break;
                case 3:
                        method_3 (RAG, (NonTerminal)s, RHS, g);
                        break;
                case 4:
                        method_4 (RAG, (NonTerminal)s, RHS, g);
                        break;
                case 5:
                        method_5 (RAG, (NonTerminal)s, RHS, g);
                        break;
                case 6:
                        method_6 (RAG, (NonTerminal)s, RHS, g);
                        break;
                case 7:
                        method_7 (RAG, (NonTerminal)s, RHS, g, true);
                        break;
                case 8:
                        method_8 (RAG, (NonTerminal)s, RHS, g);
                        break;
```

```
			case 9:
				method_9 (RAG, (NonTerminal)s, RHS, g);
				break;
			case 10:
				method_7 (RAG, (NonTerminal)s, RHS, g, false);
				break;
			default:
				System.out.println ("	S_algolNode: applyMethod:
error: unexpected method number: " + m);
			}
		}

		public static final int METHOD_DEFAULT = 0;
		public void method_0 (ReflexiveAttributeGrammar RAG, NonTerminal LHS,
Symbol[] RHS, Genotype g) {
			// Beginning.
			// System.out.println ("	S_algolNode: method_0: beginning:
this.child.length = " + this.child.length);
			// System.out.println ("	S_algolNode: method_0: beginning:
RAG == null: " + (RAG==null));
			// System.out.println ("	S_algolNode: method_0: beginning:
LHS == null: " + (LHS==null));
			// System.out.println ("	S_algolNode: method_0: beginning:
RHS == null: " + (RHS==null));
			// System.out.println ("	S_algolNode: method_0: beginning: g
== null: " + (g==null));

			// Descend into subtree.
			for (int i = 0; i < this.child.length; i++) {
				// System.out.println ("	S_algolNode: creating child
node: " + i);
				child[i] = new S_algolNode(RAG, RHS[i], g);
				// System.out.println ("	S_algolNode: creating child
node: " + i + " completed.");
			}

			// Completed.
			// System.out.println ("	S_algolNode: method_0: completed.");
		}

		public static final int METHOD_ID_NEW = 1;
		public void method_1 (ReflexiveAttributeGrammar RAG, NonTerminal LHS,
Symbol[] RHS, Genotype g) {
			// This production method processes the following
NonTerminals:
			//	id int new
			//	id real new
			//	id bool new
			//	id string new
			//	id proc new
			// In each case the processing is the same:
			//	1.  Create a new identifier.
			//	2.  Declare it as a Terminal Symbol.
			//	3.  Make it this node's Symbol.

			// Beginning.
			// System.out.println ("	S_algolNode: method_1: beginning.");

			// Create a new identifier.
			// - the NonTerminal name will be one of these strings:
			//	" <idIntNew>"
			//	" <idRealNew>"
			//	" <idBoolNew>"
			//	" <idStringNew>"
			//	" <idProcNew>"
			// - decide which by getting char at 4.
			String type = s.getName();
			// System.out.println ("	S_algolNode: method_1: type: " +
type);
```

```
                char typeChar = type.charAt(4);
                String i = null;
                switch (typeChar) {
                case 'I':
                        i = Identifier.getId("int");
                        break;
                case 'R':
                        i = Identifier.getId("real");
                        break;
                case 'B':
                        i = Identifier.getId("bool");
                        break;
                case 'S':
                        i = Identifier.getId("string");
                        break;
                case 'P':
                        i = Identifier.getId("proc");
                        break;
                default:
                        System.out.println ("      S_algolNode: error:
unexpected type initial: " + typeChar);
                }

                // Declare new id as a Terminal Symbol.
                Terminal terminal = new Terminal (i);

                // Make it this S_algolNode's Symbol.
                this.s = terminal;
                this.child = null;

                // Completed.
                // System.out.println ("   S_algolNode: method_1: completed.");
        }

        public static final int METHOD_CLAUSE_FOR = 2;
        public void method_2 (ReflexiveAttributeGrammar RAG, NonTerminal LHS,
Symbol[] RHS, Genotype g) {
                // Beginning.
                // System.out.println ("   S_algolNode: method_2: beginning.");
                // System.out.println ("   S_algolNode: method_2: RAG is null:
" + (RAG==null));
                // System.out.println ("   S_algolNode: method_2: LHS is null:
" + (LHS==null));
                // System.out.println ("   S_algolNode: method_2: RHS is null:
" + (RHS==null));
                // System.out.println ("   S_algolNode: method_2: g is null: "
+ (g==null));

                // Production:
                //      0       for symbol,
                //      1       id int new,
                //      2       space symbol,
                //      3       equals symbol,
                //      4       clause int,
                //      5       to symbol,
                //      6       clause int,
                //      7       do symbol,
                //      8       clause void.
                // Expand all children except the final clause void.
                // System.out.println ("   S_algolNode: method_2: expand first
children.");
                for (int i = 0; i < 8; i++) {
                        child[i] = new S_algolNode(RAG, RHS[i], g);
                }

                // Get the <id int new> Terminal.
                // System.out.println ("   S_algolNode: method_2: new
terminal.");
                Terminal t = (Terminal)(child[1].s);
```

```
            // System.out.println ("   S_algolNode: method_2: new terminal
is null: " + (t==null));

            // Make new Production.
            //     id int c:    <id int new>.
            // System.out.println ("   S_algolNode: method_2: new
production.");
            Production p = new Production (RAGInitializer.idIntC, new
Symbol[] {t});
            // System.out.println ("   S_algolNode: method_2: new
production is null: " + (p==null));

            // Push the new Production onto the grammar.
            // System.out.println ("   S_algolNode: method_2: push.");
            // System.out.println ("   S_algolNode: method_2: RAG is null:
" + (RAG==null));
            RAG.addProduction(p);

            // Expand the clause void node.
            // System.out.println ("   S_algolNode: method_2: expand last
child.");
            child[8] = new S_algolNode (RAG, RHS[8], g);

            // Pop the Production from the grammar.
            // System.out.println ("   S_algolNode: method_2: pop.");
            RAG.removeProduction();

            // Completed.
            // System.out.println ("   S_algolNode: method_2: completed.");
      }

      public static final int METHOD_DECLARATION = 3;
      public void method_3 (ReflexiveAttributeGrammar RAG, NonTerminal LHS,
Symbol[] RHS, Genotype g) {
            // This production method processes the following Productions:
            //     0       decl,
            //     1       sequence separator,
            //     2       sequence void.
            // The declaration may be for an int, real, bool or string.
            // In each case the processing is the same:
            // 1.  Expand all children except the final sequence void.
            // 2.  Get the Terminal declared in child[0].
            // 3.  Make a Production for the Terminal.
            // 4.  Push the Production onto the grammar.
            // 5.  Use the grammar to expand child[2].
            // 6.  Pop the Production.

            // Beginning.
            // System.out.println ("   S_algolNode: method_3: beginning.");

            // 1.  Expand all children except the final sequence void.
            for (int i = 0; i < 2; i++) {
                  // System.out.println ("   S_algolNode: method_3:
expanding child " + i);
                  child[i] = new S_algolNode(RAG, RHS[i], g);
            }

            // 2.  Get the Terminal declared in child[0].
            // It's at decl -> decl _type_ -> id _type_ new
            // but by now it's been changed from id _type_ new
            // to an identifier of the form _type_._n_
            // eg "int.0" or "bool.27".
            //     where _type_ = int, real, bool or string
            Terminal t = (Terminal)(child[0].child[0].child[1].s);

            // 3.  Make a Production for the Terminal.
            // The production is of the form:
            //     id _type_:   id _type_ new.
            Production p = null;
            // Get the first letter of its type (i, r, b or s):
            char typeChar = t.getName().charAt(1);
            switch (typeChar) {
            case 'i':
                  p = new Production (RAGInitializer.idInt, new Symbol[]
{t});
                  break;
            case 'r':
                  p = new Production (RAGInitializer.idReal, new Symbol[]
{t});
                  break;
            case 'b':
                  p = new Production (RAGInitializer.idBool, new Symbol[]
{t});
                  break;
            case 's':
                  p = new Production (RAGInitializer.idString, new
Symbol[] {t});
                  break;
            default:
                  System.out.println("      S_algolNode: method_3: error:
unexpected typeChar: " + typeChar);
            }

            // 4.  Push the Production onto the grammar.
            RAG.addProduction(p);

            // 5.  Use the grammar to expand child[2].
            child[2] = new S_algolNode (RAG, RHS[2], g);

            // 6.  Pop the Production.
            RAG.removeProduction();

            // Completed.
            // System.out.println ("   S_algolNode: method_3: completed.");
      }

      public static final int METHOD_PARAMETER = 4;
      public void method_4 (ReflexiveAttributeGrammar RAG, NonTerminal LHS,
Symbol[] RHS, Genotype g) {
            // This production method processes the following Productions:
            //              parameter_0:
            //      0       int symbol,
            //      1       id int new.
            //              parameter_1:
            //      0       real symbol,
            //      1       id real new.
            //              parameter_2:
            //      0       bool symbol,
            //      1       id bool new.
            //              parameter_3:
            //      0       string symbol,
            //      1       id string new.
            // That is, 4 alternatives each comprising a type and a
            // new identifier.
            // In each case the processing is the same:
            // 1.  Expand all children in the usual way.
            // 2.  Let type = Terminal declared in child[0].
            // 3.  Let name = Terminal declared in child[1].
            // 4.  Create a S_algolParameter object.

            // Beginning.
            // System.out.println ("   S_algolNode: method_4: beginning.");

            // 1.  Expand all children in the usual way.
            method_0 (RAG, LHS, RHS, g);

            // 2.  Let type = Terminal declared in child[0].
            Terminal type = (Terminal)(child[0].s);
            // 3.  Let name = Terminal declared in child[1].
            Terminal name = (Terminal)(child[1].s);

            // 4.  Create a S_algolParameter object.
            parameter = new S_algolParameter(type, name);

            // Completed.
            // System.out.println ("   S_algolNode: method_4: completed.");
    }

    public static final int METHOD_PARAMETER_LIST_0 = 5;
    public void method_5 (ReflexiveAttributeGrammar RAG, NonTerminal LHS,
Symbol[] RHS, Genotype g) {
            // This production method processes the following Productions:
            //     parameter list: parameter.
            // The processing is as follows:
            // 1.  Expand all children in the usual way.
            // 2.  Start new S_algolParameterList.

            // Beginning.
            // System.out.println ("   S_algolNode: method_5: beginning.");

            // 1.  Expand all children in the usual way.
            method_0 (RAG, LHS, RHS, g);

            // 2.  Start new S_algolParameterList.
            parameterList = new
S_algolParameterList(((S_algolNode)child[0]).parameter);

            // Completed.
            // System.out.println ("   S_algolNode: method_5: completed.");
    }

    public static final int METHOD_PARAMETER_LIST_1 = 6;
    public void method_6 (ReflexiveAttributeGrammar RAG, NonTerminal LHS,
Symbol[] RHS, Genotype g) {
            // This production method processes the following Productions:
            //     parameter list: parameter, parameter separator,
parameter list.
            // The processing is as follows:
            // 1.  Expand all children in the usual way.
            // 2.  Add to S_algolParameterList.
            // 3.  Delete child's parameterList to avoid space leak.

            // Beginning.
            // System.out.println ("   S_algolNode: method_6: beginning.");

            // 1.  Expand all children in the usual way.
            method_0 (RAG, LHS, RHS, g);

            // 2.  Add to S_algolParameterList.
            parameterList = new
S_algolParameterList(((S_algolNode)child[0]).parameter,
((S_algolNode)child[2]).parameterList);

            // 3.  Delete child's parameterList to avoid space leak.
            ((S_algolNode)child[1]).parameterList = null;

            // Completed.
            // System.out.println ("   S_algolNode: method_6: completed.");
    }

    public static final int METHOD_DECL_PROC = 7;
    public static final int METHOD_DECL_PROC_NONREC = 10;
    public void method_7 (ReflexiveAttributeGrammar RAG, NonTerminal LHS,
Symbol[] RHS, Genotype g, boolean recursionEnabled) {
            // This production method processes the following Productions:
            //     declProcVoid_0:
```

```
//      0       procSymbol,
//      1       idProcNew,
//      2       sequenceSeparator,
//      3       clauseVoid.
//      declProcVoid_1:
//      0       procSymbol,
//      1       idProcNew,
//      2       roundLSymbol,
//      3       parameterList,
//      4       roundRSymbol,
//      5       sequenceSeparator,
//      6       clauseVoid.
//      declProcInt_0:
//      0       procSymbol,
//      1       idProcNew,
//      2       roundLSymbol,
//      3       arrowSymbol,
//      4       typeIntSymbol,
//      5       roundRSymbol,
//      6       sequenceSeparator,
//      7       clauseInt.
//      declProcInt_1:
//      0       procSymbol,
//      1       idProcNew,
//      2       roundLSymbol,
//      3       parameterList,
//      4       arrowSymbol,
//      5       typeIntSymbol,
//      6       roundRSymbol,
//      7       sequenceSeparator,
//      8       clauseInt.
//      declProcReal_0:
//      0       procSymbol,
//      1       idProcNew,
//      2       roundLSymbol,
//      3       arrowSymbol,
//      4       typeRealSymbol,
//      5       roundRSymbol,
//      6       sequenceSeparator,
//      7       clauseReal.
//      declProcReal_1:
//      0       procSymbol,
//      1       idProcNew,
//      2       roundLSymbol,
//      3       parameterList,
//      4       arrowSymbol,
//      5       typeRealSymbol,
//      6       roundRSymbol,
//      7       sequenceSeparator,
//      8       clauseReal.
//      declProcBool_0:
//      0       procSymbol,
//      1       idProcNew,
//      2       roundLSymbol,
//      3       arrowSymbol,
//      4       typeBoolSymbol,
//      5       roundRSymbol,
//      6       sequenceSeparator,
//      7       clauseBool.
//      declProcBool_1:
//      0       procSymbol,
//      1       idProcNew,
//      2       roundLSymbol,
//      3       parameterList,
//      4       arrowSymbol,
//      5       typeBoolSymbol,
//      6       roundRSymbol,
//      7       sequenceSeparator,
//      8       clauseBool.
```

```
                //      declProcString_0:
                //      0       procSymbol,
                //      1       idProcNew,
                //      2       roundLSymbol,
                //      3       arrowSymbol,
                //      4       typeStringSymbol,
                //      5       roundRSymbol,
                //      6       sequenceSeparator,
                //      7       clauseString.
                //      declProcString_1:
                //      0       procSymbol,
                //      1       idProcNew,
                //      2       roundLSymbol,
                //      3       parameterList,
                //      4       arrowSymbol,
                //      5       typeStringSymbol,
                //      6       roundRSymbol,
                //      7       sequenceSeparator,
                //      8       clauseString.
                //
                // That is, productions for procedure declaration, with or
                // without parameters, returning any type.  These productions
                // come in several forms which we must distinguish.  The
                // simplest way is by counting how many children the node has,
                // though it's a bit of a kludge.  The children are numbered
                // above for convenience.  It can be seen that the number of
                // children is always 4, 7, 8 or 9.
                //
                // The processing is as follows:
                // 1.  Expand all children in the usual way except last child.
                // 2.  Let prototype = new S_algolPrototype object.
                // 3.  Compute prototype.name.
                // 4.  Compute prototype.parameter.
                // 5.  Compute prototype.returnType.
                // 6.  Apply productions to grammar.
                // 7.  Expand last child.
                // 8.  Remove productions from grammar.
                //
                // This method is different to others in that it takes an
extra
                // boolean parameter which indicates whether to enable
recursion.
                // If recursion is enabled (ie true), the external
declarations
                // are made available inside the procedure body.  Otherwise
they
                // are not.  This is not an S_algol issue: S_algol supports
                // recursion, end of story.  The choice is provided to
investigate
                // the effect of recursion on Gads performance.
                //
                // To enable recursion, use method_7.  To disable it, use
method_10.

                // Beginning.
                // System.out.println ("   S_algolNode: method_7: beginning.");

                // 1.  Expand all children in the usual way except last child.
                for (int i = 0; i < child.length-1; i++) {
                        // System.out.println ("   S_algolNode: method_7:
expanding child " + i);
                        child[i] = new S_algolNode(RAG, RHS[i], g);
                }

                // 2.  Let prototype = new S_algolPrototype object.
                // System.out.println ("   S_algolNode: method_7: creating
prototype.");
                prototype = new S_algolPrototype();
```

```
            // 3.  Compute prototype.name.
            // System.out.println ("   S_algolNode: method_7: setting
name.");
            prototype.setName((Terminal)(child[1].s));

            // 4.  Compute prototype.parameter.
            // If the number of children is 4 or 8, there are no
parameters.
            // The prototype's parameter list is empty by default so in
this
            // case there is nothing to do.
            // If there are parameters, the parameterList is always
child[3].
            // System.out.println ("   S_algolNode: method_7: testing
child.length.");
            if (child.length == 7 || child.length == 9) {
                  // System.out.println ("   S_algolNode: method_7:
setting parameter.");

      prototype.setParameter(((S_algolNode)child[3]).parameterList);
            }

            // 5.  Compute prototype.returnType.
            // If the number of children is 7 or less, we are expanding
            // a _decl proc void_ node, so the type is void.  Otherwise,
            // the type symbol is always the third last child.
            if (child.length <= 7) {
                  // System.out.println ("   S_algolNode: method_7:
setting void type.");
                  prototype.setType(RAGInitializer.typeVoidSymbol);
            } else {
                  // System.out.println ("   S_algolNode: method_7:
setting other type.");
                  prototype.setType((Terminal)(child[child.length-4].s));
            }

            // 6.  Apply productions to grammar.
            // System.out.println ("   S_algolNode: method_7: applying
internal.");
            prototype.applyInternal(RAG);
            if (recursionEnabled) {
                  // System.out.println ("   S_algolNode: method_7:
applying external.");
                  prototype.applyExternal(RAG);
            }

            // 7.  Expand last child.
            // System.out.println ("   S_algolNode: method_7: expanding
last child.");
            child[child.length-1] = new S_algolNode(RAG, RHS[child.length-
1], g);

            // 8.  Remove productions from grammar.
            if (recursionEnabled) {
                  // System.out.println ("   S_algolNode: method_7:
removing external.");
                  prototype.removeExternal(RAG);
            }
            // System.out.println ("   S_algolNode: method_7: removing
internal.");
            prototype.removeInternal(RAG);

            // Completed.
            // System.out.println ("   S_algolNode: method_7: completed.");
      }

      public static final int METHOD_SEQUENCE_VOID_5 = 8;
      public void method_8 (ReflexiveAttributeGrammar RAG, NonTerminal LHS,
Symbol[] RHS, Genotype g) {
		// This production method processes the following production:
		//
		//	sequence void: decl proc, sequence separator, sequence
void.
		//
		// It applies the external declarations of declProc to
sequenceVoid,
		// so that the procedure is in scope in the sequence.
		//
		// A typical external declaration looks like this:
		//
		//	appl string:
		//		id proc,
		//		clausel,
		//		exp0 int,
		//		clause separator,
		//		exp0 real,
		//		clause separator,
		//		exp0 bool,
		//		clause r.
		//
		// The type of the procedure and the number and type of the
arguments
		// are variable.
		//
		// The processing is as follows.  The S_algolPrototype object
which
		// describes the procedure is obtained by navigating the parse
		// tree to the node where the object is held (as the synthetic
		// attribute "prototype").  This object is always at the same
		// relative position:
		//
		//	child[0].child[0].prototype
		//
		// whatever the procedure type or parameters.
		//
		// In detail the steps are:
		// 1.  Expand all children in the usual way except last child.
		// 2.  Let prototype = created S_algolPrototype object.
		// 3.  Apply external productions to grammar.
		// 4.  Expand last child.
		// 5.  Remove external productions from grammar.

		// Beginning.
		// System.out.println ("   S_algolNode: method_8: beginning.");

		// 1.  Expand all children in the usual way except last child.
		for (int i = 0; i < this.child.length-1; i++) {
			// System.out.println ("   S_algolNode: (1) creating
child node: " + i);
			child[i] = new S_algolNode(RAG, RHS[i], g);
		}

		// 2.  Let prototype = created S_algolPrototype object.
		// System.out.println ("   S_algolNode: method_8: (2)
prototype.");
		S_algolPrototype prototype =
((S_algolNode)(child[0].child[0])).prototype;

		// 3.  Apply external productions to grammar.
		// System.out.println ("   S_algolNode: method_8: (3)
applyExternal");
		prototype.applyExternal(RAG);

		// 4.  Expand last child.
		// System.out.println ("   S_algolNode: method_8: (4) last
child");
		child[child.length-1] = new S_algolNode(RAG, RHS[child.length-
1], g);
```

```
                // 5.  Remove external productions from grammar.
                // System.out.println ("    S_algolNode: method_8: (5) remove");
                prototype.removeExternal(RAG);

                // Completed.
                // System.out.println ("    S_algolNode: method_8: completed.");
        }

        public static final int METHOD_LITERAL = 9;
        public void method_9 (ReflexiveAttributeGrammar RAG, NonTerminal LHS,
Symbol[] RHS, Genotype g) {
                // This production method processes the following productions:
                //
                //      literal int:
                //      literal real:
                //      literal bool:
                //      literal string:
                //
                // That is, all the literal types.  The effect of this method
is
                // to dispense with the micro-syntax definition of these
literals.
                // The reason for using this method is principally to avoid
                // overflow in arithmetic literals, and partly for efficiency.
                //
                // The processing method is to generate a random literal of
the
                // required type, and convert this node from a nonterminal to
                // a terminal.
                //
                // Note that negative literals are not supported - the effect
is
                // achieved by the unary minus.  (This is a change from
S_algol.)

                // Beginning.
                // System.out.println ("    S_algolNode: method_9: beginning.");

                // Generate appropriate literal and make it a Terminal.
                Terminal terminal = null;
                if (LHS == RAGInitializer.literalInt) {
                        terminal = new Terminal (
                                " "+String.valueOf(
                                        (int)(
                                                Math.random()*Include.MAX_INT
                                        )
                                )
                        );
                } else if (LHS == RAGInitializer.literalReal) {
                        terminal = new Terminal (
                                " "+(
                                        String.valueOf(
                                                (int)(
                                                        Math.random()*Include.MAX_INT
                                                )*Math.pow(
                                                        10,
                                                        (int)(

        Math.random()*Include.MAX_REAL_EXP
                                                        )
                                                )
                                        )
                                ).toLowerCase()      // Convert E to e in
scientific notation
                        );
                } else if (LHS == RAGInitializer.literalBool) {
                        terminal = new Terminal (
                                " "+String.valueOf(Math.random() < 0.5)
                        );
                } else if (LHS == RAGInitializer.literalString) {
                        // Generate a string whose length is...
                        //
                        //      0 with probability 50%
                        //      1 with probability 25%
                        //      2 with probability 12.5%
                        //      3 with probability 6.25%
                        //      ...
                        //
                        // and whose characters are drawn uniformly from ascii
                        // and the S-algol specials.

                        // Compose ascii alphabet.
                        String lower = "abcdefghijklmnopqrstuvwxyz";
                        String upper = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
                        String digit = "0123456789";
                        String punct = " !#$%&()*+,-./:;<=>?@[\\]^_`{|}~";
                        String alphabet = lower+upper+digit+punct;

                        // Compose special alphabet.
                        String specialf = "npotb'\"";

                        // Prepare a target string.
                        String s = "";

                        // With probability 50% add another character.
                        while (Math.random() < 0.5) {

                                // Choose a char at random from alphabet +
specialf.
                                int i =
(int)(Math.random()*(alphabet.length()+specialf.length()));
                                String x = "";
                                if (i < alphabet.length()) {
                                        x = String.valueOf(alphabet.charAt(i));
                                } else {
                                        x = "'" +
String.valueOf(specialf.charAt(i-alphabet.length()));
                                }

                                // Apend it to the target string.
                                s = s + x;
                        }
                        terminal = new Terminal (" \""+s+"\"");
                } else {
                        System.out.println("      S_algolNode: method_9: error:
unexpected LHS: " + LHS.toString());
                }

                // Make it this S_algolNode's Symbol.
                this.s = terminal;
                this.child = null;

                // Completed.
                // System.out.println ("   S_algolNode: method_9: completed.");
        }

        public S_algolNode (ReflexiveAttributeGrammar RAG, Symbol s, Genotype
g) {

                // System.out.println ("   S_algolNode: beginning symbol: " +
s.getName());
                // System.out.println ("   S_algolNode: beginning: RAG == null:
" + (RAG==null));
                // System.out.println ("   S_algolNode: beginning: s == null: "
+ (s==null));
                // System.out.println ("   S_algolNode: beginning: g == null: "
+ (g==null));
```

```
            // Note the Symbol at this node.
            this.s = s;

            // If the Symbol s is a Terminal, we are finished.
            // System.out.println ("   S_algolNode: finished?");
            if (s instanceof Terminal)
                {
                return;
                }

            // Use Genotype g to choose a production for this NonTerminal.
            // System.out.println ("   S_algolNode: getProduction");
            Production p = ((NonTerminal)s).getProduction(g);

            // Get the RHS of this production.
            // System.out.println ("   S_algolNode: getRHS");
            Symbol[] RHS = p.getRHS();

            // Prepare to have children.
            // System.out.println ("   S_algolNode: new node");
            this.child = new S_algolNode[RHS.length];

            // Apply specified method.
            // System.out.println ("   S_algolNode: apply");
            applyMethod(p.getMethodNumber(), RAG, s, RHS, g);

            // System.out.println ("   S_algolNode: completed symbol: " +
s.getName());
        }
}
```

B.3 Object creation

The following class is generated from the rag definition. The part between the lines

```
// INSERT S_algol.java below this line
// INSERT S_algol.java above this line
```

is generated automatically from the rag, by means of a Perl script. To save space I have replaced large chunks of this with ellipsis. Following this comes a section which sets the production method numbers specific to this problem (ie Fact). To control multiple versions, this section is written by giving a "patch" which deviates from the "standard" production methods.

```
package       ec.experiment.nrp.rag.ecj;
import ec.experiment.nrp.rag.*;
import ec.vector.*;
import ec.*;
import ec.simple.*;
import ec.util.*;

// RAGInitializer - sets up objects for this problem
public class RAGInitializer extends SimpleInitializer {
        // Public Symbols - these are S_algol Symbols used in other places.
        // Defined here to avoid unnecessary searching by name.
        public static ReflexiveAttributeGrammar S_algol = null;
        public static Terminal roundLSymbol;
        public static Terminal roundRSymbol;
        public static Terminal typeVoidSymbol;
        public static NonTerminal program;
        public static NonTerminal idInt;
        public static NonTerminal idIntC;
        public static NonTerminal idReal;
        public static NonTerminal idBool;
        public static NonTerminal idString;
        public static NonTerminal exp0Void;
        public static NonTerminal exp0Int;
        public static NonTerminal exp0Real;
        public static NonTerminal exp0Bool;
        public static NonTerminal exp0String;
        public static NonTerminal applVoid;
        public static NonTerminal applInt;
        public static NonTerminal applReal;
        public static NonTerminal applBool;
        public static NonTerminal applString;
        public static NonTerminal clauseSeparator;
        public static NonTerminal literalInt;
        public static NonTerminal literalReal;
        public static NonTerminal literalBool;
        public static NonTerminal literalString;

        public void setup(final EvolutionState state, final Parameter base) {
               super.setup(state,base);

        // System.out.println("    RAGInitializer: beginning.");
// INSERT S_algol.java below this line
...
Terminal procTypeVoidSymbol = new Terminal(" PROC.VOID");
Terminal procTypeIntSymbol = new Terminal(" PROC.INT");
Terminal procTypeRealSymbol = new Terminal(" PROC.REAL");
Terminal procTypeBoolSymbol = new Terminal(" PROC.BOOL");
Terminal procTypeStringSymbol = new Terminal(" PROC.STRING");
Terminal letSymbol = new Terminal(" let");
Terminal assignmentSymbol = new Terminal(" :=");
Terminal procSymbol = new Terminal(" procedure");
Terminal roundLSymbol = new Terminal("(");
Terminal roundRSymbol = new Terminal(")");
Terminal arrowSymbol = new Terminal(" ->");
...
Terminal divSymbol = new Terminal(" div");
Terminal remSymbol = new Terminal(" rem");
Terminal typeVoidSymbol = new Terminal(" void");
NonTerminal program = new NonTerminal("program");
NonTerminal programFact = new NonTerminal("programFact");
NonTerminal programMonkey = new NonTerminal("programMonkey");
...
NonTerminal applCode = new NonTerminal("applCode");
NonTerminal applIformat = new NonTerminal("applIformat");
NonTerminal applCeiling = new NonTerminal("applCeiling");
NonTerminal applFloor = new NonTerminal("applFloor");
Production program_0 = new Production(program, new Symbol[] {programFact});
Production programFact_0 = new Production(programFact, new Symbol[]
{preambleSymbol, phenotypeFact, postambleFactSymbol, endOfProgram});
Production programMonkey_0 = new Production(programMonkey, new Symbol[]
{preambleSymbol, phenotypeMonkey, postambleMonkeySymbol, endOfProgram});
Production phenotypeMonkey_0 = new Production(phenotypeMonkey, new Symbol[]
{phenotypeMonkeyBeginSymbol, literalString, phenotypeMonkeyEndSymbol});
Production endOfProgram_0 = new Production(endOfProgram, new Symbol[]
{spaceSymbol, questionSymbol});
...
```

```
Production applCode_0 = new Production(applCode, new Symbol[]
{procCodeSymbol, clauseL, exp0Int, clauseR});
Production applIformat_0 = new Production(applIformat, new Symbol[]
{procIformatSymbol, clauseL, exp0Int, clauseR});
Production applCeiling_0 = new Production(applCeiling, new Symbol[]
{procCeilingSymbol, clauseL, exp0Real, clauseR});
Production applFloor_0 = new Production(applFloor, new Symbol[]
{procFloorSymbol, clauseL, exp0Real, clauseR});
ReflexiveAttributeGrammar S_algol = new ReflexiveAttributeGrammar();
S_algol.addTerminal(preambleSymbol);
S_algol.addTerminal(postambleMonkeySymbol);
S_algol.addTerminal(postambleCartSymbol);
S_algol.addTerminal(postambleTileSymbol);
S_algol.addTerminal(postambleMultiplexerSymbol);
S_algol.addTerminal(postamblePowerSymbol);
...
S_algol.addTerminal(phenotypeAnnieBeginSymbol);
S_algol.addTerminal(phenotypeAnnieEndSymbol);
S_algol.addTerminal(bySymbol);
S_algol.addTerminal(divSymbol);
S_algol.addTerminal(remSymbol);
S_algol.addTerminal(typeVoidSymbol);
S_algol.addNonTerminal(program);
S_algol.addNonTerminal(programFact);
S_algol.addNonTerminal(programMonkey);
S_algol.addNonTerminal(phenotypeMonkey);
S_algol.addNonTerminal(endOfProgram);
...
S_algol.addNonTerminal(applSin);
S_algol.addNonTerminal(applSqrt);
S_algol.addNonTerminal(applDigit);
S_algol.addNonTerminal(applLetter);
S_algol.addNonTerminal(applCode);
S_algol.addNonTerminal(applIformat);
S_algol.addNonTerminal(applCeiling);
S_algol.addNonTerminal(applFloor);
S_algol.addProduction(program_0);
S_algol.addProduction(programFact_0);
S_algol.addProduction(programMonkey_0);
S_algol.addProduction(phenotypeMonkey_0);
S_algol.addProduction(endOfProgram_0);
S_algol.addProduction(programCart_0);
S_algol.addProduction(phenotypeCart_0);
S_algol.addProduction(programTile_0);
S_algol.addProduction(phenotypeTile_0);
...
S_algol.addProduction(applLetter_0);
S_algol.addProduction(applCode_0);
S_algol.addProduction(applIformat_0);
S_algol.addProduction(applCeiling_0);
S_algol.addProduction(applFloor_0);
// INSERT S_algol.java above this line

              // Patch for Fact.
              // parameter_1              deleted
              // parameter_2              deleted
              // parameter_3              deleted
              // parameterList_1  deleted
              // declProcReal_1   deleted
              // sequenceVoid_3   deleted
              // sequenceVoid_5   renamed       sequenceVoid_0

              // Set method numbers in selected Productions.
              // This way of setting method numbers is a kludge
              // because the BNF is not up to it in the .rag file.
              // The methods are defined in S_algolNode.

              // Methods to introduce new IDs.
              idIntNew_0.setMethodNumber(S_algolNode.METHOD_ID_NEW);
```

```
            idRealNew_0.setMethodNumber(S_algolNode.METHOD_ID_NEW);
            idBoolNew_0.setMethodNumber(S_algolNode.METHOD_ID_NEW);
            idStringNew_0.setMethodNumber(S_algolNode.METHOD_ID_NEW);
            idProcNew_0.setMethodNumber(S_algolNode.METHOD_ID_NEW);

            // clause void: for symbol, id int new, ...
            clauseVoid_4.setMethodNumber(S_algolNode.METHOD_CLAUSE_FOR);

            // sequence void: decl let, sequence separator, sequence void.
            //
sequenceVoid_3.setMethodNumber(S_algolNode.METHOD_DECLARATION);

            // parameter: _type_ symbol, id _type_ new.
            // for _type_ = int, real, bool, string.
            parameter_0.setMethodNumber(S_algolNode.METHOD_PARAMETER);
            // parameter_1.setMethodNumber(S_algolNode.METHOD_PARAMETER);
            // parameter_2.setMethodNumber(S_algolNode.METHOD_PARAMETER);
            // parameter_3.setMethodNumber(S_algolNode.METHOD_PARAMETER);

            // parameter list: parameter.

      parameterList_0.setMethodNumber(S_algolNode.METHOD_PARAMETER_LIST_0);

            // parameter list: parameter, parameter list.
            //
parameterList_1.setMethodNumber(S_algolNode.METHOD_PARAMETER_LIST_1);

            // decl proc _type_: ...
            // for _type_ = int, real, bool, string.
            declProcVoid_0.setMethodNumber(S_algolNode.METHOD_DECL_PROC);
            declProcVoid_1.setMethodNumber(S_algolNode.METHOD_DECL_PROC);
            declProcInt_0.setMethodNumber(S_algolNode.METHOD_DECL_PROC);
            declProcInt_1.setMethodNumber(S_algolNode.METHOD_DECL_PROC);
            declProcReal_0.setMethodNumber(S_algolNode.METHOD_DECL_PROC);
            //
declProcReal_1.setMethodNumber(S_algolNode.METHOD_DECL_PROC);
            declProcBool_0.setMethodNumber(S_algolNode.METHOD_DECL_PROC);
            declProcBool_1.setMethodNumber(S_algolNode.METHOD_DECL_PROC);

      declProcString_0.setMethodNumber(S_algolNode.METHOD_DECL_PROC);

      declProcString_1.setMethodNumber(S_algolNode.METHOD_DECL_PROC);

            // sequence void: declProc, sequenceSeparator, sequenceVoid
            //
sequenceVoid_5.setMethodNumber(S_algolNode.METHOD_SEQUENCE_VOID_5);

      sequenceVoid_0.setMethodNumber(S_algolNode.METHOD_SEQUENCE_VOID_5);

            // Set public static Symbols.
            RAGInitializer.S_algol = S_algol;
            RAGInitializer.roundLSymbol = roundLSymbol;
            RAGInitializer.roundRSymbol = roundRSymbol;
            RAGInitializer.typeVoidSymbol = typeVoidSymbol;
            RAGInitializer.program = program;
            RAGInitializer.idInt = idInt;
            RAGInitializer.idIntC = idIntC;
            RAGInitializer.idReal = idReal;
            RAGInitializer.idBool = idBool;
            RAGInitializer.idString = idString;
            RAGInitializer.exp0Void = exp0Void;
            RAGInitializer.exp0Int = exp0Int;
            RAGInitializer.exp0Real = exp0Real;
            RAGInitializer.exp0Bool = exp0Bool;
            RAGInitializer.exp0String = exp0String;
            RAGInitializer.applVoid = applVoid;
            RAGInitializer.applInt = applInt;
            RAGInitializer.applReal = applReal;
            RAGInitializer.applBool = applBool;
            RAGInitializer.applString = applString;
            RAGInitializer.clauseSeparator = clauseSeparator;
            RAGInitializer.literalInt = literalInt;
            RAGInitializer.literalReal = literalReal;
            RAGInitializer.literalBool = literalBool;
            RAGInitializer.literalString = literalString;

            // Mark this position.
            S_algol.mark();

            // System.out.println("    RAGInitializer: completed.");
        }
}
```