Performing the Code Generation Phase of a Software Compiler Through Linear Genetic Programming

by

Mathew Carr

A DISSERTATION

Submitted to

The University of Liverpool

in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

24th September, 2010

Performing the Code Generation Phase of a Software Compiler Through Linear Genetic Programming

by

Mathew Carr

Abstract

This dissertation considers the process of code generation in a compiler: the task of transforming a program stored in some form of high level representation into a series of instructions in a low level language suitable for execution by a machine such as a computer. The problem is interpreted as one of computer program induction and optimisation.

The evolutionary computation method Linear Genetic Programming (LGP) is adapted for this task by use of a novel fitness function based upon methods a human programmer may consider to be good practice.

Two methods are considered, that of direct application of LGP and a second method which attempts to accelerate the process by subdividing the input program into smaller sections. The methods are compared and contrasted by considering the required computational effort to produce a solution to a series of sample programs, and the distribution of program lengths that result.

The dissertation concludes that LGP alone is not currently a suitable method for the task of code generation, but may act as a useful optimisation tool within some larger system.

Declaration

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

Mathew Carr

TABLE OF CONTENTS

1 Introduction

This dissertation investigates several methods for using Linear Genetic Programming (LGP) to augment the code generation phase of a software compiler, improving the ability of the compiler to transform an input program written in a high level language into the most optimal machine code form given the available functionality of the target architecture.

1.1 Software Compilers

Compilers are computer programs that translate one language to another (Louden, 1997). The purpose of a software compiler is to transform an input program in the form of some high-level, human readable language to an output program in the form of object code or machine code executable by some real processor or virtual machine.

High level programming languages are designed to allow the programmer to specify the solution to a problem in terms approximating those of the problem space (Wexelblat, 1981). For example, a high level language may allow the programmer to group multiple pieces of data into an aggregate data structure representing some object from the problem space. The programmer is then able to write programs that manipulate instances of these aggregate data structures without any manual manipulation of processor elements such as the register file or the stack. These features are provided through abstraction: it is the responsibility of the compiler to provide the transformation from high level language features into instructions executable by the target machine. Examples of high level languages include BASIC, C, C++ and Pascal.

Conversely, low level languages require the programmer to specify programs in terms of the exact operations to be performed by the target machine. A typical example of a low level language is assembly language. Assembly language acts as a wrapper above the level of machine code instructions, providing features to aid editing by a human programmer such as human-readable instruction mnemonics and sub-routine constructs.

The design of a program written in a low level language will be influenced by the constructs provided by the language, which in turn will be influenced by the properties of the target machine, such as the instruction set and makeup of the register file. The programmer may choose to take advantage of accelerated features unique to the target architecture, and the ability to address the functionality of the target machine directly can allow for more optimised code than would be automatically produced by compiler software.

Programs written in low level languages may be more difficult to maintain than those written in a high level language due to the low level program's reliance on the

properties of the target machine. Such programs are not as easily ported between architectures as programs written in a high level language.

If the program is heavily optimised, the overall objective of a given piece of code may become unclear upon review. Examples of this include specialised mathematical routines that provide approximations to common functions through unintuitive exploitation of the layout of floating point numbers within memory, such as ‘fast inverse square root’ (Lomont, 2003).

Almost all commercial software is written in a high level language, and is transformed into binary executables by means of a software compiler. It is very rare for machine code to be directly manipulated by a human programmer, except for the development of software for uncommon architectures where a compiler may not be available, or in the design of programs which meta-manipulate machine code, such as the assembler function of a compiler. As such, a considerable amount of research has been invested in improving the quality and efficiency of compiler software.

Where compilers are available, they may not be fully able to automatically exploit specialised functionality available to advanced architectures. This functionality includes complex vector mathematics, and operations on large homogeneous data sets. These functions are highly valuable in applications such as image and sound processing. For this specialised functionality to be used effectively, the programmer must manually design the software to use these functions and invoke it explicitly. Regardless of whether this invocation is triggered from code written in a high level or low level language, this places a dependency on the architecture, making it harder to reuse the same software elsewhere.

It is highly desirable to investigate methods where compilers can be augmented with the ability to automatically identify opportunities to optimise programs by using specialised functionality where appropriate. With this capability available, a programmer would be free to use to specify solution programs in a high level language and rely on the compiler to correctly transform this program into the most optimal machine code possible, given the available functionality of the target architecture.

We now consider the various stages performed by a typical software compiler:

The parsing and lexical analysis stages transform a human-readable plain-text input program (source code) written in a high level language into an intermediate representation (IR). The IR is a machine manipulable expression of the input program in terms of the constructs provided by the high level language, such as the structure of statements, variable names and control structures. A ‘symbol table’ is produced as part of the IR containing information describing the type, scope and other properties

of the variables and symbolically named entities appearing in the source code. These stages are sometimes collectively referred to as the 'front end' of a compiler.

The 'code generation' phase of compilation transforms the IR into some form of linear assembly language or machine code for the target architecture. This phase is sometimes referred to as the 'back end' of a compiler. It is during this phase that much of the high level nature of the input program such as the presence of high level control structures is dissolved, and the properties and capabilities of the target architecture become a significant factor in the selection of instructions. Many optimisations may be performed at this stage, such as the elimination of intermediate variables and common sub-expression elimination. Therefore, the optimality of the output object code is highly dependent on the sophistication of the techniques used during the code generation phase. It is the code generation phase of compilation that will be considered in detail in this dissertation.

1.2 Genetic Programming

Genetic Programming (GP) (Koza, 1992) is an evolutionary computation technique that can automatically solve problems without requiring the user to know or specify the form or structure of the solution in advance (Poli, et. al., 2008).

Evolutionary computation techniques such as GP can be used to automatically induce solutions in the form computer programs, given a high level statement of the problem to be solved and a method for calculating the suitability, or 'fitness', of a given candidate solution and a criterion indicating when a sufficiently suitable program has been found. Solution programs are produced through a search the space of possible computer programs using a model of evolution through natural selection. GP typically works upon, and produces, tree structures, such as the S-Expressions used in the LISP language or other graph-based structures.

The GA algorithm typically proceeds as follows. First, a pool of random candidate programs is generated. Then, the fitness of each of these programs is calculated. A new 'generation' of candidate programs is produced by repeatedly selecting several highly suitable candidate programs to participate in 'crossover' or 'mutation' operations. The crossover operation combines parts of two parent programs to produce a 'child' program. The mutation operation produces a new program by randomly altering some component of a parent program. The fitness of each of the new generation of candidate programs is then calculated and the process is repeated until a candidate program of sufficient fitness is evolved. The crossover and mutation operations do not perform any analysis on the parent programs to select the most productive method to produce a child program; they are applied randomly. It is the intention that, by manipulating those programs that are observed to be the most fit, candidate solutions of successively better fitness can be produced with each generation until an acceptable solution is found.

Linear Genetic Programming (LGP) is an adaptation of GP techniques for use in working on linear structures. As such, it is suitable for the direct manipulation of assembly language (Cramer, 1985) or machine code instructions (Nordin, 1994, 1997). It is proposed that LGP may provide a means to accelerate or augment the code generation phase of compiler software.

1.3 Application of Linear Genetic Programming

In this project, two methods of applying LGP are investigated: the use of LGP as a method for performing code generation given an IR; and the use of LGP as a method for optimising low level code it has generated.

In the case of using LGP to perform code generation, the objective of the LGP system is to evolve a candidate solution in a low level language that has the same semantics as an input program given in the form of a parse tree. In the case of using LGP as a method for optimising low level code, the objective of the LGP system is to evolve a candidate solution that has greater fitness (or some other desirable quality such as reduced program length) than the input program given in the form of a low level language solution. Ten simple input programs are considered; these are given as an appendix together with a discussion of how each program may be considered by the LGP system and an 'optimal' low level solution program listing.

The primary problem that must be solved to enable the application of LGP in any scenario is the definition of a 'fitness function', a function allowing the LGP system to calculate the fitness of a given candidate solution. The fitness function must have some extent of graduation for the LGP system to function effectively; the system requires the ability to identify the 'more correct' of two candidate solutions. Without a graduated fitness function, the LGP system only has the ability to identify a wholly correct solution and cannot encourage the creation of successively fitter programs. The problem of identifying a suitable fitness function for the task of code generation has not been widely studied previously.

The fitness function proposed in this project takes the form of an analysis of the actions taken by a candidate solution program. The candidate solution is run in a virtual machine and each step of its execution recorded. This record is compared against a record representing the 'ideal' execution, produced by running the parse tree representation of the input program in an interpreter. Where the two records indicate that the same (or similar) calculations are being attempted, the candidate solution is rewarded.

In addition, a candidate solution is rewarded for taking actions that are designated as being of a productive type. This category includes actions that a human programmer would employ, such as loading a value from an input variable and storing a value to

an output variable. A candidate solution is punished for taking actions that are designated as being of a non-productive type. This category includes actions that a human programmer would avoid, such as assigning a value to a register twice in succession without having read it in the interval, and reading values from registers that have not yet been initialised with a value.

These two methods are designed to provide a graduated fitness function that encourages the gradual assembly of programs in a similar manner to how a human programmer may attempt to solve the problem. However, the LGP system is not restricted to simply assembling the program as a mechanical, statement by statement translation of the input program; the system may combine candidate solutions freely using the genetic operators. As a result, a correct candidate solution may arise which has the appropriate semantics but achieves this in an unexpected way (reversed order of statements, selection of unusual instructions, etc.), a property typical of programs produced by a GP system.

In applying LGP as a method for code generation, two models are considered: 'standard' and 'incremental'. In the 'standard' method, the LGP system is directed to evolve a single solution program exhibiting the full semantics of the input program. In the 'incremental' method, the input program is disassembled into smaller programs, and the LGP system is directed to evolve a solution program to each of these in turn. The partial solutions are then combined to produce a single solution program which will exhibit the same semantics as the complete input program.

These methods are compared using two measurements. The first is the 'computational effort', the number of low level instructions that must be executed in the virtual machine in order to evolve a solution program with 99% probability. This measurement reflects the difficulty of evolving a solution program given an input program and a set of parameters to the LGP system. The second measurement considers the length of the solution programs produced by the LGP system, given that sufficient resources have been devoted to the system to produce a solution.

To investigate the ability of LGP to optimise existing programs, an additional 'refinement' stage is defined. The refinement stage uses the LGP system to evolve a solution program as before, with the change that some fraction of the random population of candidate programs is initialised as duplicates of the solution originally found. The refinement stage is terminated upon reaching a set limit of new candidate program creations, and the best candidate produced so far is returned.

The refinement stage is performed after a program has been returned by the 'standard' or 'incremental' evolution models. A non-evolutionary third method of code generation using a tree-walking algorithm is also considered. This algorithm does not perform any optimisations, which can cause it to produce sub-optimal code.

The resulting program length distributions are compared against each other, and with an ‘optimal’ solution produced by a human programmer.

As part of this project, software has been produced allowing experiments to be conducted investigating the use of LGP as a means of solution program induction, and as a means of improving already identified solution programs. A guide to the operation of this software is provided as an appendix.

Upon analysis of the results, the experiments have demonstrated that LGP is capable of automatically performing the task of code generation for the ten input programs when given sufficient time to do so. The LGP system has demonstrated the ability to automatically use the instruction set available to it in a productive manner without any additional information on which instructions from the set are a suitable translation of the given input parse tree nodes. Therefore, I believe that the LGP will be capable of automatically assimilating any additional instructions inserted into the instruction set and applying these in low level candidate programs where they are most productive.

This report concludes with the observation that Linear Genetic Programming alone is not suitable for the task of code generation for the ten input programs considered. An examination of the computational effort metric shows that the ‘standard’ model is highly unsuitable due to scaling exponentially with increasing calculation complexity, whereas the ‘incremental’ method scales only linearly. The use of evolutionary methods such as LGP is discouraged due to their stochastic nature; they are not guaranteed to be productive. In the models considered here, the LGP system terminates as soon as an acceptable program is found, which results in programs containing considerable amounts of non-productive code.

The technique of using LGP as a method for optimising existing programs appears to be more promising. The refinement stage is able to manipulate programs produced by the ‘standard’, ‘incremental’ or tree-walking algorithm into improved programs of the same quality as a human produced ‘optimal’ solution. Combining the algorithmic code generation method with the evolutionary refinement stage results in a fast, reliable method of generating code of a quality comparable to a human programmer.

In the final section, we identify some of the limitations of the work due to the choice of LGP system parameters, and discuss some further possibilities for research.

2 Study of Problem

This dissertation studies the process of code generation within a software compiler, and investigates means by which it can be performed or accelerated through the application of Linear Genetic Programming (LGP) techniques. This section discusses how code generation is performed in a traditional compiler and introduces the concepts that will be used to define the scope of the project and the nature of the experiments performed in section 4.

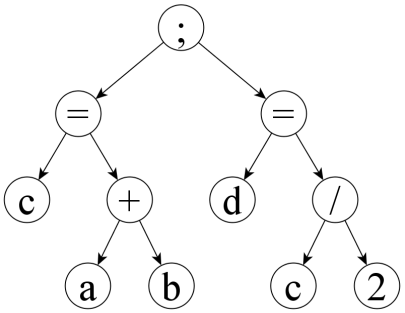
2.1 What is Code Generation?

Code generation is the process by which a compiler transforms an input program in the form of an intermediate representation (IR) into a linear form suitable for execution by some target machine (Louden, 1997). The code generator may output machine code directly, or output some form of assembly language to be converted into machine code by a separate assembly stage. The objective is to produce the most optimal output program given the properties of the target machine while ensuring that all the relevant semantics as defined by the IR are expressed.

The target machine considered in this dissertation is a register machine modelled after a simplified model of a modern Von Neumann processor. The machine is capable of executing strings of instructions supplied to it in a simple assembly-like low level language (in this manner, it is an interpreter). The machine contains two memory areas for storage of value: a register file consisting of a finite number of storage locations, addressed through a non-negative integer index, where the values of intermediate calculations may be stored; and a memory area, addressed by the plain-text name of a variable from the symbol table, for the storage of the values of variables. The instructions are executed in a linear sequence starting from line 1 until the input instruction string is exhausted or a RETURN instruction is reached. It is assumed that all instructions take the same length of time to executed and no pipelining is implemented: the sole measure of the optimality of a program is its length in instructions.

In this dissertation, an IR is considered to consist of two parts, the parse tree (or abstract syntax tree) and the symbol table.

The parse tree is a machine manipulable representation of the input program in terms of the high level language features used in the original input source code file. The parse tree will have been generated from a plain text input source code file by the parsing and lexical analysis stage. A visualisation of a possible parse tree is shown below, together with the high level language program it encapsulates:




```
c = a + b;
d = c / 2
```

Figure 2.1: Visualisation of possible parse tree part of an input IR together with high level source code

The interior nodes of the graph denote language constructs such as sequencing (represented here by a semicolon character), assignments, branches and loops; and arithmetic operators such as addition, subtraction, division and multiplication. The leaf nodes of the graph denote variables (represented here by a node showing the symbolic name of the variable) or constants (represented here by a node showing the value of the constant).

This tree has the effect of first calculating the value of the expression (`a+b`) and assigning this value to the variable c, and then calculating the value of the expression (`c/2`) and assigning this value to the variable d.

The program expressed by this parse tree can be executed in a top-down manner. Any node can be invoked to return the numeric value of the sub tree in the current state at the time of invocation. To execute the program, the value of the root node is requested in the context of some given input system state. A system state in this context refers to the values of all symbolic variables. This system state is global and shared throughout execution of the tree. As such, the state may be altered during execution (for example, as a result of the assignment operation), and this altered state subsequently inspected. For sequencing nodes, the left hand side child is evaluated first, followed by the right hand side child and the value of this child returned.

The second part of the IR is the symbol table. The symbol table contains the information about the variables referred to in the input program. This includes the symbolic name of the variable, the constant nature of the variable and the scoping of the variable in the context of the program. The parse tree may represent an isolated subroutine apart from calling program; certain variables may exist only within the scope of the subroutine. Together, the parse tree and the symbol table hold the full semantics of the input program.

The code generation phase includes the following tasks:

- Instruction selection – The compiler selects which instructions from the instruction set of the target architecture to use to construct the output program. It is important to select instructions that perform the required tasks as implied by the IR.

- Instruction scheduling – The compiler selects the most optimal ordering for the instructions it has selected. This stage can be critical in processors which implement pipelining, as correct instruction scheduling may allow multiple functions be undertaken simultaneously (such as memory access, instruction decoding and instruction execution).

- Register allocation – The compiler allocates register locations for the results of intermediate calculations and variable values. Effective use of registers reduces the need to access memory locations to store or read values, resulting in decreased execution time.

Together, the function of these tasks is to provide a translation from the high level parse tree program form into a low level machine executable language form.

2.2 Example of Code Generation

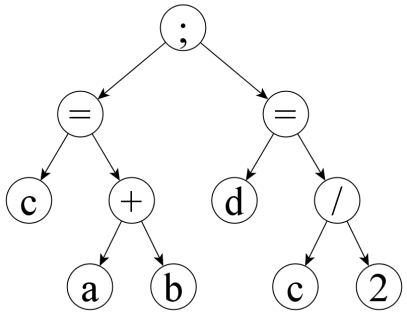
One possible output program produced by the code generation phase of a compiler is shown below, together with the input IR used to create it. The output program is shown in a simple assembler-like low level language.

Symbol Table:

Symbolic variable name	Data type	Scope	Value of constant
a	Signed Integer	Global – exists outside of subroutine	N/A
b	Signed Integer	Global – exists outside of subroutine	N/A
c	Signed Integer	Local - not accessible outside subroutine	N/A
d	Signed Integer	Global – exists outside of subroutine	N/A
2	Signed Integer	Global – exists outside of subroutine	2

Table 2.2: Symbol table component of example IR

Parse Tree:




```
c = a + b;
d = c / 2
```

Figure 2.3: Parse tree component of example IR

Output program:

```
1: LOADS 0,   a    // load the value of variable a into register 0
2: LOADS 1,   b    // load the value of variable b into register 1
3: ADD   0, 0, 1   // add the values of registers 0 and 1 and store the result in register 0
4: STORS 0,   c    // store the value of register 0 into variable c
5: LOADS 0,   c    // load the value of variable c into register 0
6: LOADV 1,   2    // load the direct value 1 into register 2
7: DIV   0, 0, 1   // divide the value of r0 by the value of r1 and store the result in r0
8: STORS 0,   d    // store the value of register 0 into variable d
9: RETURN          // end procedure
```

Listing 2.4: Low level program output of code generation phase of compiler for example IR produced by an algorithm

Integer division in this case will round toward negative infinity.

We consider a low level program to be syntactically correct with respect to an IR if and only if, after execution, for all possible values of all input variables, the values of all variables that exist outside the scope of the subroutine have the same value they have after execution of the program in its high level form. To show this briefly, some possible values of the input variables together with the expected output value (calculated by interpretation of the parse tree) are shown in the table below:

	Values of variables before execution of high level program					Values of variables after execution of high level program				
	a	b	c	d	2	a	b	c	d	2
Input case 1:	4	8	1	11	2	4	8	12	6	2
Input case 2:	8	2	6	20	2	8	2	10	5	2
Input case 3:	5	15	2	4	2	5	15	20	10	2
Input case 4:	-1	3	-5	0	2	-1	3	2	1	2

Table 2.5: List of possible input and output values after execution of a model parse tree for various input cases

If the system state is initialised with the values from the row labelled 'Input case 1', we can trace the execution of the low level program to determine if it has the same semantics as the IR. Where the value of register or variable in memory has changed, this is shown in boldface.

	Value of machine register or variable in memory after execution of instruction in left column						
	Machine Register		Variable in memory				
	0	1	a	b	c	d	2
`Initialisation`	garbage	garbage	4	8	1	11	2
`1: LOADS 0, a`	**4**	garbage	4	8	1	11	2
`2: LOADS 1, b`	4	**8**	4	8	1	11	2
`3: ADD 0, 0, 1`	**12**	8	4	8	1	11	2
`4: STORS 0, c`	12	8	4	8	**12**	11	2
`5: LOADS 0, c`	**12**	8	4	8	12	11	2
`6: LOADV 1, 2`	12	**2**	4	8	12	11	2
`7: DIV 0, 0, 1`	**6**	2	4	8	12	11	2
`8: STORS 0, d`	6	2	4	8	12	**6**	2
`9: RETURN`	6	2	4	8	12	6	2

Table 2.6: Values of machine registers and variables in memory during execution of a low level program

This low level program has shown the same semantics as defined by the IR for this input case. The values stored in variables a and b are recalled in lines 1 and 2, and the addition is performed in line 3. This value is then stored in the variable c in line 4. Lines 5 and 6 recall the values of variables 2 and the constant 2, and the division is performed in line 7. The result of this calculation is then stored in line 8. The RETURN statement on line 9 ends the subroutine. After termination of execution, the values of variables a, b, c and d are as expected. Analyses for the other input cases will show the same result.

This output program was produced by an algorithmic approach; each calculation and variable access was performed explicitly as it appeared in the parse tree. This output

program is not optimal for this input IR, however. Line 5 may be removed as it reinserts the value stored in the variable c into the register 0, which is a redundant operation. The symbol table contains additional information which may be used to further optimise the program. The variable c does not exist outside of this isolated subroutine; it may have been included by the human programmer to ease comprehension or maintenance of the program during editing. In this case, it can be seen that this program requires only that the values of variables a, b and d have the correct value upon program termination. If this addition information is considered, line 4 may be omitted as there is no requirement that the value of variable c be calculated and stored. With these two lines omitted, the program becomes:

```
1: LOADS 0,   a    // load the value of variable a into register 0
2: LOADS 1,   b    // load the value of variable b into register 1
3: ADD   0, 0, 1   // add the values of registers 0 and 1 and store the result in register 0
4: LOADV 1,   2    // load the direct value 1 into register 2
5: DIV   0, 0, 1   // divide the value of r0 by the value of r1 and store the result in r0
6: STORS 0,   d    // store the value of register 0 into variable d
7: RETURN          // end procedure
```

Listing 2.7: Optimised low level program listing produced by optimisation of code produced by an algorithm

This optimised output program contains no references to the variable c, but exhibits the same semantics with respect to the complete IR.

Optimising compilers contain code generators that are capable of performing many optimisations automatically. Instead of explicitly recalculating or recalling a value from memory each time it is needed, a code generator may place frequently used values in registers for fast reuse. ‘Peephole optimisation’ is a technique where a small section of produced output code is considered in isolation, similar to the inspection described above, and analysed to determine if its content may be manipulated or removed to increase its quality. In pipelined processors, instructions are scheduled to maximise usage of the stages of the processing pipeline. Processors with complex instruction sets provide many different approaches to performing the same task; it is the responsibility of the code generator to select the most appropriate series of instructions with respect to the requirements of the programmer, such as faster execution or reduced code size.

In all of these cases, the optimisation capability must be explicitly programmed into the code generation software by the author. The programmer must supply a series of rules or patterns describing when it is appropriate to use each instruction or attempt a particular optimisation. As a result, the implementation of an optimising compiler can become highly complicated and error prone.

In this dissertation, a method is presented whereby LGP may be used as part of an extensible code generation facility capable of automatically performing code generation using only a description of the semantics of the available instruction set, and performing many of these optimisations implicitly due to the effect of fitness pressure.

3 Background

Since the origin of machine learning research in the 1950s, the automatic induction of computer programs is a task that has been the subject of considerable research. This section provides a brief description of some of the approaches that have been developed to attempt the automatic induction of computer programs. The section concludes with a discussion of the reasons why the Linear Genetic Programming evolutionary model has been considered as a suitable algorithm for use in performing code generation.

3.1 Machine Learning as Induction of Computer Programs

The goal of the field of research known as 'machine learning' is to develop computer programs capable of automatically and autonomously solving problems. As computers primarily work within the domain of computer programs and information stored within memory, research has focused on methods for granting programs the ability to work within these domains to search for solutions. The task of automatically producing a computer program exhibiting some kind of complex behaviour based on a high-level statement of the problem is known as 'program induction'.

In 1958 Friedberg (Friedberg, 1958) (Friedberg et. al., 1959), developed a computer program model in which a software agent is required to modify its own behaviour to emulate the behaviour of another; that is, to produce a computer program through induction. In this model, a 'learner' agent is tasked with emulating a behaviour, in the form of a computer program copying the state of bits within a memory, defined by a 'teacher' agent. The two agents are not able to communicate directly except for the following: for each cycle, the 'learner' agent pseudo-randomly develops and executes a candidate solution program against an initial memory state supplied by the teacher agent. The teacher agent then examines the resulting memory state produced by the learner and returns a 'success' or 'failure' result indicating whether or not the desired behaviour was exhibited. In response, the learner agent pseudo-randomly alters its behaviour program and re-submits this to the teacher. Friedberg demonstrated that, over time, the learner would become able to exhibit the behaviour defined by the teacher's result function.

As part of this research, Friedberg gave the following observation detailing the rationale behind pursing the automatic induction of computer programs, and the insight into how such methods may be realised:

> Although modern electronic computers have relieved us of many tedious calculations, we are still faced with difficult tasks in which the slowness of our thoughts and the shortness of our memory limit us severely, but for which present machines are less adequate than we because they lack judgment. If we are ever to make a machine that

> will speak, understand or translate human languages, solve mathematical problems with imagination, practice a profession or direct an organization, either we must reduce the activities to a science so exact that we can tell a machine precisely how to go about doing them or we must develop a machine that can do things without being told precisely how.
>
> If a machine is not told how to do something, at least some indication must be given of what it is to do; otherwise we could not direct its efforts toward a particular problem. It is difficult to see a way of telling it what without telling it how, except by allowing it to try out procedures at random or according to some unintelligent system and informing it constantly whether or not it is doing what we wish. The machine might be designed to gravitate toward those procedures which most often elicit from us a favourable response. We could teach this machine to perform a task even thought we could not describe a precise method for performing it, provided only that we understood the task well enough to be able to ascertain whether or not it had been done successfully.

Investigation of the reduction of activities 'to a science so exact that we can tell a machine precisely how to go about doing them' would lead to knowledge-based systems such as expert systems (Giarratino et. al., 1998), and other knowledge-based AI reasoning techniques.

Counter to knowledge-based systems, there are several classes of techniques which consider the case of developing 'a machine that can do things without being told precisely how'. The class of techniques considered in this dissertation is known as 'evolutionary algorithms', or 'evolutionary computation'.

3.2 Evolutionary Computation

Evolutionary computation (EC) techniques are a class of techniques that can be considered to be the result of interpreting Friedberg's analysis of a 'machine that can do things without being told precisely how'. EC techniques can be used in many applications. They can be used to search for the most optimal values for a set of variables, to create classifier systems, or to create computer programs through induction. We consider the 'genetic' sub-class of EC techniques, which includes Genetic Algorithms (GA), Genetic Programming (GP) and Linear Genetic Programming (LGP).

Genetic EC techniques attempt to solve problems through use of a model of evolution through natural selection. In this model of natural selection, a population of candidate solution individuals is maintained. Each 'generation', candidate solutions from the

population are selected for reproduction based on their 'fitness', that is, their suitability to the target task. This is the selection phase. Randomly selected sections of these highly fit candidate solutions are combined or slightly modified to produce the next generation of candidate solutions. This is the recombination phase; such operations are referred to as recombination operations. Candidate solutions with poor fitness are removed from the population. This process continues until a 'success criterion' is met, such as the creation of a sufficiently fit program; or until a 'termination criterion' is met, such as the generation of some large number of candidate solutions without successfully evolving an acceptable solution.

In algorithms such as this, the selection of generally highly fit candidate solutions for recombination operations acts as a form of pressure on the population, causing it to tend towards candidate solutions of higher quality. The recombination operators act on these fit candidate solutions to create diversity in the population while attempting to maintain qualities which lead to good fitness scores. Genetic evolutionary computation algorithms require the definition of a problem-specific 'fitness function' mapping candidate solutions to fitness values. The objective of the fitness function is to provide a graduated mapping from candidate solutions to fitness values allowing the evolutionary system to distinguish between candidate solutions by how 'close' they are to the ideal solution. As EC techniques attempt to improve upon the random programs produced during the initialisation stage using semi-guided search, they can be considered to be class of metaheuristic optimisation algorithms.

In defining the representation for the structures undergoing adaptation and the primitives from which these structures will be composed, EC techniques impose two requirements: 'sufficiency' and 'closure'.

A definition of a candidate solution structure is said to be sufficient if there exists the capability for the expression of an acceptable candidate solution. For example, if the objective is to induce a computer program capable of performing addition, then instances of candidate solution structures must be capable of somehow performing this function, either directly or indirectly. Without this, the evolutionary system will continuously generate candidate solutions that perform operations that are similar to the desired operation, but never an acceptable solution.

As the recombination operations of EC attempt to produce solutions through repeated manipulation of candidate solutions, it is capable of exploring any possible candidate solution. If the candidate solutions represent computer programs, it is possible that the programs produced by EC contain incomplete or otherwise invalid structures. As a result, the user must account for all possible kinds of irregularity when evaluating the candidate solutions; this is the 'closure' requirement. Alternatively, the user may choose to impose restrictions on the structure of the random candidate solutions produced during the initialisation stage, and ensure that the recombination operations never produce an 'invalid' individual. For example, if the candidate solutions

represent instructions in an assembly-like language, the recombination operations may produce an illegal instruction such as an attempt to access a register outside the valid range. The user may instruct the interpreter to use some form of wrap-around to transform the illegal instruction into a legal instruction, instruct the interpreter to ignore illegal instructions, or take other measures to prevent such an instruction from being randomly generated or otherwise produced as a result of crossover or mutation.

The structure of the candidate solution individuals, the approaches used to combine and modify these individuals, the nature of the fitness function, and the success and termination criteria all differ according to the specific genetic technique used. The earliest, well studied of the genetic methods is the Genetic Algorithm.

3.3 Genetic Algorithm

The Genetic Algorithm (GA) methodology is founded upon an analysis performed by Holland in *Adaptation in Natural and Artificial Systems* (Holland, 1992). In this analysis, Holland considers the task of calculating an optimal set of values for one or more variables of known type. The values of the set of variables are encoded as a single string of binary digits. To search the space of possible combinations of binary digits (and therefore search the set of possible variables), a model of evolution through natural selection is used.

The algorithm begins by randomly generating a population of competing random candidate solution strings. The fitness of each of these candidate solutions is then calculated using the fitness function. There are two methods used to generate the next generation of candidate solutions: ‘crossover’ and ‘mutation’. These methods are selected at random (with user-specified rate) during the reproduction phase of the GA.

In ‘crossover’, the genetic material of two parent individuals is combined to produce a child individual containing some genetic material from each of the parents. A crossover point is selected within the encoding of the binary string, and the binary values to the left of this point from the first parent and the binary values to the right of this point from the second parent are combined to produce the child program. The length of the resulting child program is the same as that of each of the parents. This method acts as an analogue to the exchange of chromosomal material in sexual reproduction in nature, giving the ‘Genetic Algorithm’ its name.

In ‘mutation’, a random bit or series of bits within the string encoding is selected and these positions within a selected parent individual are toggled, producing a child program that is a slightly modified copy of the parent program.

New generations of candidate solutions are generated successively until a candidate solution of sufficient fitness is generated, or some other termination criterion is met,

such as a maximum number of generations. In such cases, a copy of the best-of-run candidate solution (the candidate solution with the best fitness found at any point during the execution of the algorithm) will be returned.

There are two broad models for managing the individuals in the population during the reproduction stage: 'generational' and 'steady state'. In the generational model, the entire population is replaced with an equal number of child programs simultaneously. In the steady state model, programs with low fitness in the population are replaced by new child programs in a continuous fashion. For GA, the steady state model is often used.

As an example, we will consider the use of GA in attempting to solve a symbolic regression problem. In this problem, we have access to a set of empirically obtained data points in two dimensions, and suspect that a quadratic relationship exists mapping the value of one variable to that of the other. That is, for x within the interval –5 to +5, we suspect there exists the relationship $y = A\ x^2 + B\ x + C$, where A, B and C are constants to be found (in this case, we will assume that they are signed integers).

x	y
-5	6
-4	0
-3	-4
-2	-6
-1	-6
0	-4
1	0
2	6
3	14
4	24
5	36

Table 3.1: Empirically obtained data points for symbolic regression example

To use GA to solve this problem, we need to specify the nature of the candidate solutions (the nature of what exactly we wish to find as a result of running the algorithm), how a candidate solution will be encoded as a binary string, how the recombination operations will proceed, the fitness function, and the success and termination criteria. Additional parameters to the evolutionary system, such as population size and rates of the recombination operations are often given as a tableau of parameters. (Not shown here)

In this example, for simplicity, we will attempt to find the optimal value of three signed integer constants in order to construct a function modelling the data in the table above. Our candidate solutions will consist of an encoding of three signed

integer values. We will use 4-bit, two's complement encoding to transform each candidate solution 3-tuple of signed integers into a 12-bit binary string.

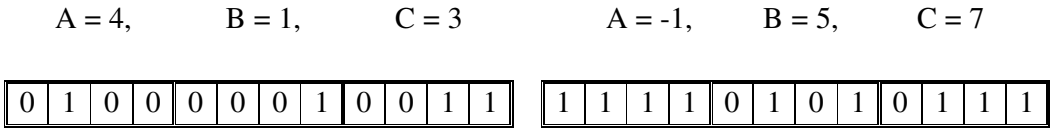


Figure 3.2: Example representations of two candidate solutions as 12-bit binary strings

Using this representation, the values of A, B and C may range from –8 to 7, giving 2^{12} possible solutions. This choice of encoding implies a number of assumptions regarding the data set and the nature of the solution. We are assuming that there exists a quadratic relationship between the two variables, and that the constants A, B and C are integers. If any of these assumptions are false, our attempt to use the GA system to evolve a solution will fail in some capacity. It may or may not be able to produce an approximated solution in which our assumptions do hold. For example, if the ideal solution is quartic, then the returned solution may be a very coarse quadratic model of the data, or if the ideal quadratic solution does not have integer coefficients, the GA system may return a solution with integer coefficients which is close to the ideal solution.

The recombination operations of crossover and mutation will act as described previously. The mutation operator may only alter one bit at a time.

In symbolic regression applications of GA, fitness is commonly calculated by means of fitness cases. Each fitness case considers one point from the input data set. The fitness contribution for each fitness case is the absolute error between the value of y calculated by the candidate solution and the value of y from the data set. All the fitness contributions are summed to produce the fitness value for the candidate solution. For this function, lower fitness indicates a candidate solution of higher quality. If the fitness contribution for a fitness case is less than 3 (an arbitrary and user-defined value), this is recorded as a 'hit'. The success criterion for this problem is met when the number of 'hits' is equal to the number of data points (11).

The following two candidate solutions have been produced as a result of random initialisation of the population.

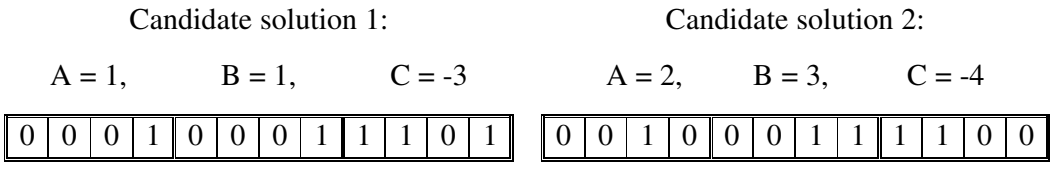


Figure 3.3: Example representations of two candidate solutions as 12-bit binary strings

The following table gives the values of f(x) for these two candidate solutions.

	Value of y			Error (fitness contribution)	
Value of x	Data set	Candidate solution 1:	Candidate solution 2:	Candidate solution 1:	Candidate solution 2:
-5	6	17	31	11	25
-4	0	9	16	9	16
-3	-4	3	5	7	9
-2	-6	-1	-2	5	4
-1	-6	-3	-5	3	1
0	-4	-3	-4	1	0
1	0	-1	1	1	1
2	6	3	10	3	4
3	14	9	23	5	9
4	24	17	40	7	16
5	36	27	61	9	25

Table 3.4: Values produced from candidate solutions 1 and 2 for each fitness case along side the fitness contribution value

Candidate solution 1 has fitness value 61 and 4 hits; candidate solution 2 has fitness value 110 and 3 hits. To put these values in context, the following table shows the fitness values for some other candidate solutions:

Candidate solution	Fitness value	Hits
(A = 1, B = 1, C = -3)	61	4
(A = 2, B = 3, C = -5)	110	3
(A = 2, B = 1, C = 3)	187	0
(A = 6, B = 1, C = 6)	660	0
(A = -2, B = 1, C = -1)	307	2
(A = -8, B = 0, C = 6)	908	1
(A = 3, B = -8, C = -4)	330	1
(A = 2, B = 7, C = -3)	135	4
(A = 1, B = 6, C = 5)	117	1

Table 3.5: Fitness values for various candidate solutions to the symbolic regression example problem

Typically, most randomly generated candidate solutions to a problem will have very large (poor) fitness values. Of course, it is possible, though unlikely, that the exact solution may be found as a result of the random initialisation of the population.

The following graph shows the values from the data set (shown as points), and the curves defined by candidate solutions 1 (as the dashed line) and 2 (as the solid line).

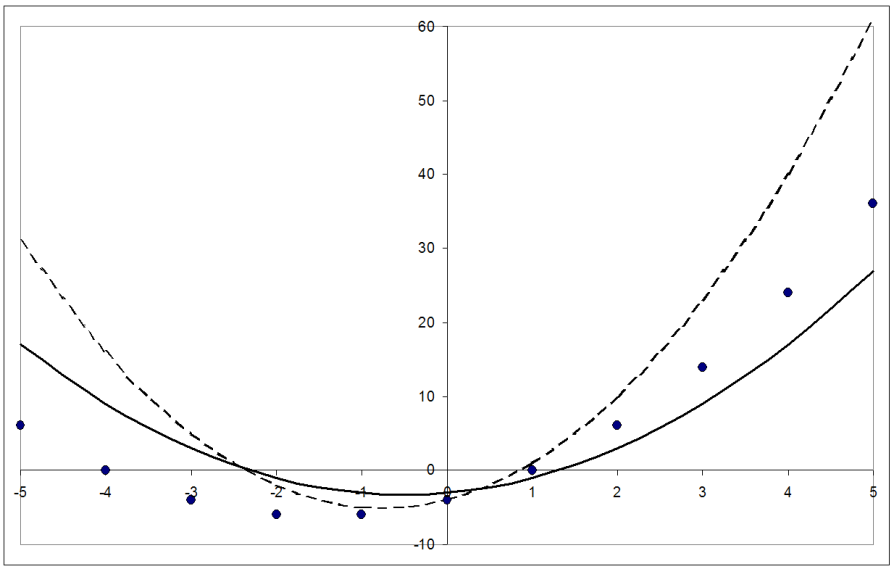


Figure 3.6: Plot of empirically obtained data set [points] together with two candidate solutions (A=1, B=1, C=-3) [dashed], and (A=2, B=3, C=-4) [solid]

From this graph, it can be seen that both candidate solutions 1 and 2 produce a very coarse model of the data set. The figure below shows how candidate solutions 1 and 2 can be used as parent individuals by the crossover recombination operation to produce a third child program.

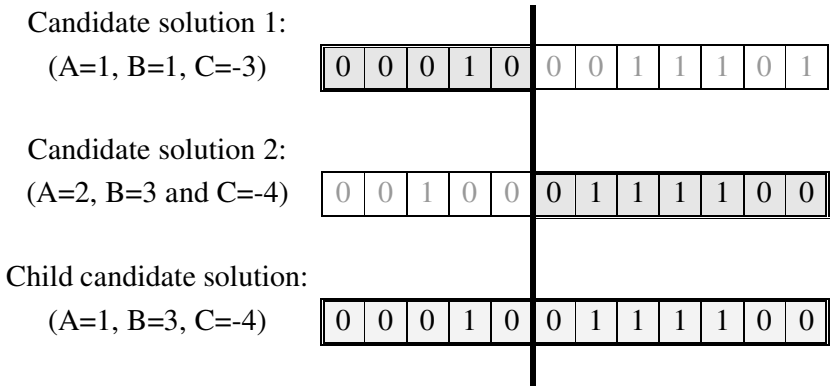


Figure 3.7: Crossover between two candidate solution programs within GA producing a third child program

The first five binary cells from candidate solution 1 and the last seven binary cells from candidate solution 2 have been combined to produce a child candidate solution containing genetic material from both parents. The user may opt to also retain the child produced as a complement to the above operation (i.e., A=2, B=1, C=-3), but this is not shown here. The following table shows the values produced by the child candidate solution for all fitness cases:

x	y	$x^2 + 3x - 4$	Error
-5	6	6	0
-4	0	0	0
-3	-4	-4	0
-2	-6	-6	0
-1	-6	-6	0
0	-4	-4	0
1	0	0	0
2	6	6	0
3	14	14	0
4	24	24	0
5	36	36	0

Table 3.8: Values of x and y from the input data set compared with the values produced by child candidate solution program

This child program scores 11 hits and is a perfect solution to the symbolic regression problem.

This example presents only 2^{12} possible solutions, and therefore may easily be examined through exhaustion of all possible solutions. However, a more realistic example may consider a dozen or more variables of 32-bit integer type. For such a program, the number of possible solutions rises to 2^{32n}, where n is the number of 32-bit integer variables. For large n, this becomes impossible to solve by exhaustion.

Holland contends that instances of binary strings encode information regarding the fitness contributions of a large number of hyperplanes within the search space, collectively referred to as schemata. The production of new individuals by combination of genetic material implicitly calculates a great deal of information of information about the search space. The reproduction stage of the evolutionary model will tend to favour those individuals with good fitness, causing successive generations to have a greater concentration of any given productive schema. When a large number of individuals possess the same productive schema, the probability that the crossover operation will be able to produce programs that do not possess this schema is reduced. As a result, the search is implicitly directed toward locating productive values for those parts of the binary string not already determined to be a likely component of the optimal solution. Note that this does not exclude the algorithm from escaping a local minimum by mutating part of a candidate solution.

Software employing systems based on the GA model are capable of solving a large number of diverse problems such as robotics, automotive design, engineering, routing, studies of encryption and financial analysis. GA has the desirable property

that it has a constant and small memory resource requirement, due to only requiring the storage of a constant number of binary strings of constant length.

To use GA as a method for the induction of computer programs, the user must define and encoding from the space of computer programs to the space of constant length binary strings, and define a fitness function mapping candidate solutions to fitness values.

For example, an instruction in a simple imperative assembly-like language may consist of two parts: an operation part and an operand part. If the instruction set of this language has four instructions, the operation part of the instruction may be encoded as a two bit binary string. Assuming that the operand part of the instruction can be encoded in four bits, each complete instruction can be encoded as a six bit binary string. Therefore, a candidate solution program in binary string form will consist of a multiple of six binary values. Using this encoding, the crossover operator will exchange series of instructions and operands between individuals to produce new candidate solution programs. If the crossover operation is not constrained to only exchange material either side of a boundary between instructions, the crossover point may be placed between instructions, causing operations and operands to be generated in the child individual that did not appear in either of the parents. This effect is usually unpredictable and hinders the evolutionary process.

As GA manipulates binary strings of a constant length, for the symbolic regression example discussed previously, the user would need to specify a different representation if they were to attempt cubic or quartic regression instead of quadratic. In the case of automatic induction of computer programs, this places a constraint on the maximum length of the candidate solution programs that will be considered. (Programs with fewer than the maximum number of permitted instructions can be encoded through the use of ‘no-operation’ instructions which are skipped)

Genetic methods such as GA have been used previously indirectly in the optimisation of code in code generation. (Cooper et. al., 1999) examines the task of selecting the most optimal ordering of a number of possible algorithmic applications which may be attempted during the code generation phase of compilation. The GA is used to select the most optimal ordering of optimisations.

In (Beaty et. al, 1996), performed an analysis of the use of GA to optimise the scheduling of machine code instructions in an attempt to exploit instruction-level parallelism – the parallelism exposed by a processor implementing pipelining. GA was used as a means of determining the optimal weight values for the configuration of a non-evolutionary scheduling algorithm.

Lorenz and Marwedel (Lorenz et al., 2004) used GA as a method for performing code generation for a number of specialised DSP architectures in the context of an audio

decoding and processing problem. The objective of this research is to produce a data flow graph from which linear assembly code can then be produced. The data flow graph must perform some required calculation, while complying with a number of constraints imposed by the target architectures. Their approach applies GA in several coupled stages of the compilation pipeline to form several stages of optimisation.

3.4 Genetic Programming

Genetic Programming (GP) is an adaptation of genetic techniques to work on more complex structures than the binary strings considered by GA. In GP, the structures undergoing adaptation are hierarchical trees composed of functions and terminals appropriate to the problem domain. Functions are interior tree nodes whose evaluation will depend in some way on their child nodes. This includes arithmetic operations, Boolean operations and control structures. Terminals are leaf nodes. This includes variables, sensor inputs and functions with no arguments which have side effects. GP was analysed in depth by Koza in *Genetic Programming* (Koza, 1992).

The use of a hierarchy of nodes makes GP more amenable to use in the induction of computer programs than GA. The structure of a GP program tree is very similar to a parse tree produced by the front end of a compiler, or a LISP S-Expression. In fact, many of the original experiments in GP were conducted using the LISP language, with the candidate solution structures stored directly as LISP S-Expressions within the LISP environment. The use of LISP structures is no longer prevalent in GP research due to the overheads involved in interpreting LISP code.

The primary difference between GA and GP is that there is very little predefined structure to the candidate solutions considered by GP. GP was designed deliberately so that the size and structure of the solution would be part of the solution, rather than something determined in advance by the user. In GP, the recombination operations (described in full later) are able to manipulate the size and structure of the candidates. As a result, genetic methods with fixed-length candidate solutions are usually considered to be derivatives of the GA approach, whereas genetic methods with the capability of manipulating the length of candidate solutions are considered to be derivates of the GP approach; this includes Linear Genetic Programming (LGP).

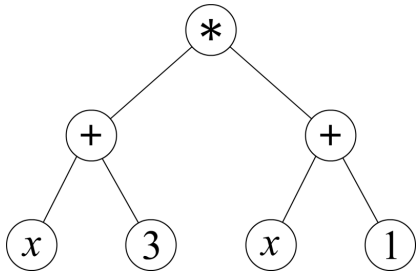


Figure 3.9: Possible candidate solution program tree individual in Genetic Programming

The above figure shows a possible candidate solution program tree that may be produced as a result of GP. The program is evaluated by evaluating the root node. This evaluation propagates down the tree to the leaf nodes. Conversely, the tree can be considered to have values which propagate up through the tree toward the root. This program calculates the value of (x+3)*(x+1) and returns the result.

Random initialisation in GP differs from that of GA as there is no predefined container of information (such as a binary string) that must be filled to construct an individual. In GP, random individuals are generated to some kind of mode defined by the user, such as requiring all leaf nodes in the tree to be at a certain depth. A commonly used mode is 'ramped half-and-half' whereby half the individuals in the population are required to have all leaf nodes at a specific depth, and the other half may have leaf nodes at any depth. Within a population, this ramping of depth values is used to create variety.

To manipulate the tree structures comprising a candidate individual, GP defines its own recombination operations of crossover and mutation.

In GP, crossover acts upon subtrees within the parent individuals. The following figure shows an example:

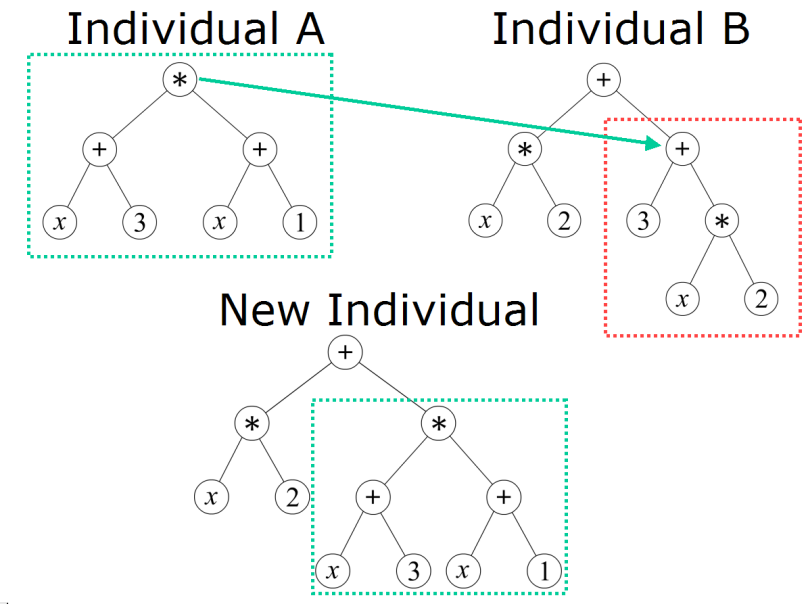


Figure 3.10: Subtree crossover in GP. The indicated subtree in Individual B is removed and replaced with the root node of Individual A to produce the New Individual

In the above example, Individual A represents the expression (x+3)*(x+1), and Individual B represents the expression (x*2)+(3+(x*2)). The root node of Individual A, representing the expression (x+3)*(x+1), and the right child node of the root of

Individual B, representing the expression 3+(x*2), have been selected for crossover. The indicated subtree in Individual B is removed and replaced with a copy of the indicated subtree (i.e. the whole tree) from Individual A to produce the new individual. This new individual, representing the expression (x*2)+((x+3)*(x+1)), contains genetic material from both parent programs. In addition, it has a size and structure different to both parent individuals.

The GP mutation operator may be defined to have any number of effects: it may replace the selected subtree with another subtree of arbitrary size and structure, it may replace the selected subtree with a terminal, or it may replace the node at the selection point with another node of the same type (an addition function node for a multiplication function node, or replace a variable terminal with a different variable terminal).

In addition to these, there are many different recombination operations which may be used, such as the 'hoist operation' (Kinnear, 1994), which produces a new child program by promoting a randomly selected subtree of a parent program to root level, and the 'shrink operation' (Angeline, 1996), a specialised case of mutation where the randomly chosen subtree is replaced by a terminal chosen at random from the terminal set.

Closure and sufficiency must be satisfied by a GP specification as before. For programs of the form described above, the function nodes must be able to gracefully accept any values produced by their child nodes. For example, the division operator may appear as a function node in a program, where it is defined to return the result of dividing the result of its left child by the result of its right child. This description of the semantics does not allow for the case where the result of the right child has the value zero. A human programmer may actively take steps to avoid such an eventuality, but the GP system is free to construct such a program as a result of random application of the recombination operators. To handle this case in GP, the semantics of the division operator are often augmented to return zero, one or the value of the left child in the case if the value of the right child is zero, and the normal value of division otherwise. The resulting operator is often referred to as the 'protected division' operator.

Alternatively, the Strongly Typed Genetic Programming STGP (Montana, 1995) may be considered. STGP introduces the concept of strict data typing into the tree genome. As is the case in many high level languages, in STGP, each terminal has an associated return type, and each function has a type for each of its arguments (children) and a return type. The process which generates the initial, random candidate solutions and the recombination operations are implemented so as to ensure they do not violate the constraints imposed by this type system.

The sufficiency condition is satisfied by ensuring that the set of functions and terminals available to the GP system is sufficient to express a solution program. For complex or novel problems, it can be difficult to determine this manually beforehand. The user can maximise the chance that the sufficiency condition is satisfied by saturating the function and terminal sets with a wide array of operations and variables. Due to the ability of GP to manipulate the size and shape of the candidate solutions, it implicitly has the ability to manipulate the concentration of instances of the functions and terminals available to it. As a result, the GP system will act to automatically remove those elements which do not contribute towards the creation of a useful solution, due to the poor fitness scores of those programs in which they appear.

As the fitness values of the population improve, only those elements which actively contribute to a useful solution will be retained. For example, in the symbolic regression example considered in section 3.3, the function set may contain addition, multiplication, subtraction, protected division as well as trigonometric operations such as sine and cosine. As the curve can be sufficiently expressed with just multiplication and addition (though would not be known in advance), the GP system may tend to remove candidate programs which contain the trigonometric functions. Alternatively, as there is no restriction on the format of the solutions returned by GP, it may find a way to express the curve using the available functions in an unexpected manner. For example, it may be possible to construct an acceptable model of the data set within the interval $-5 < x < 5$ by a sine curve.

Kinnear (Kinnear, 1993) has used GP to evolve a LISP program capable of performing sorting, requiring the definition of a novel fitness function. In *Genetic Programming and Data Structures* (Langdon, 1998), Langdon demonstrated that tree-based GP is capable of constructing and using several common aggregate data structures used in computer programs, such as the list and the stack, from typical programming primitives such as increment, load/store and conditional instructions. In the *Genetic Programming* series (Koza, 1992, 1994) (Koza et. al., 1999, 2003), Koza augments the basic GP approach with 'automatically defined functions', enhancing the ability of GP to evolve solutions to problems by granting it the ability to automatically and autonomously identify and reuse useful sections of code. Koza also demonstrates additional applications of GP including circuit design and invention. He does not attempt to provide formal proof of the ability of GP as a problem solving technique, but provides a large amount of promising experimental data. In this series, it is argued that GP techniques provide the means for human-competitive intelligence, due to the power of GP in program induction.

Although GP is capable of inducing solutions to many problems in the form of programs, it is poorly suited for dealing with linear instruction strings. It is possible to constrain the various structure altering operations of GP to produce linear or line-like structures, but these constraints reduce the effectiveness of the GP method. A

more suitable approach for the manipulation of linear structures such as strings of instructions in a low level language is Linear Genetic Programming (LGP). LGP is an approach which combines the ability of GP to automatically manipulate the size and shape of candidate solutions with the linear structure of GA.

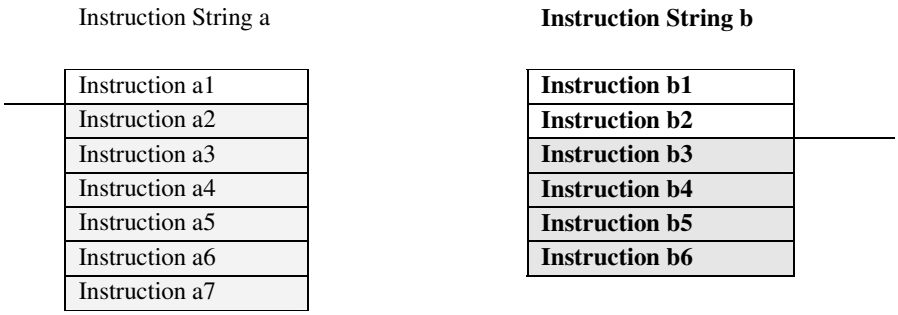
3.5 Linear Genetic Programming

Linear Genetic Programming (LGP) is an approach which combines the ability of GP to automatically manipulate the size and shape of candidate solutions within the population with the linear structure and recombination operations of GA (Banzhaf et. al., 1998).

In LGP, the candidate solution structures are linear strings of user-defined structures. Although LGP lacks the freely manipulable structure offered by GP, the linear nature of LGP structures offers a number of advantages. The linear structure of LGP programs allows it to be directly used in the production of programs written in assembly-like languages and machine code.

The nature of LGP crossover depends heavily on the nature of the structures used. LGP may be used to manipulate strings of indivisible instructions, or to manipulate strings of digits which are then decoded in a similar method to GA. The interpretation of a free-form string of digits as a computer program is sometimes referred to as a 'codon' approach.

A typical LGP crossover considering strings of atomic instructions combines the contents of two parents in a manner similar to GA. A transition point is randomly placed at an instruction boundary within each of the parent individuals. Instructions are then copied from one parent up until its transition point, then copying resumes from the position of the transition point in the second parent, and vice versa. This is shown below:



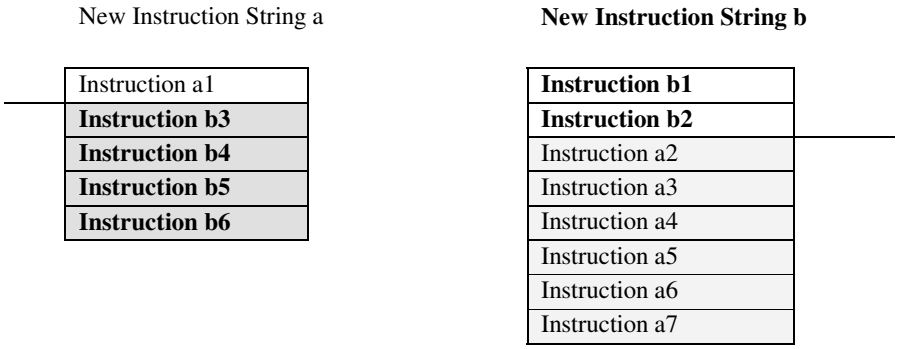


Figure 3.11: LGP crossover between two strings of indivisible instructions

This crossover is very simple to implement. If the instructions are of a uniform length in memory, the crossover can be implemented in a small number of fast memory block copy operations. Note that the two child instruction strings have a different length than either of the parent strings, unlike GA.

For the codon approach, a similar method is used except digits or groups of digits are copied instead of instructions. To ensure closure, the interpreter transforming codons into program fragments must be able to interpret any incoming codon string; this may be achieved by silently ignoring unexpected codon groupings that may be produced as a result of the crossover or any remaining incomplete codons appearing at the end of a string.

One of the earliest approaches to evolution of computer programs using a variable length linear genome is the JB language and system (Cramer, 1985). This method was formulated as a general approach for the evolution of programs. In this system, an instruction consists of a string of three consecutive digits. These digits are interpreted by the JB system in a manner similar to the decoding of a machine code instruction by processor; the first digit indicates the operation part of the instruction and the remaining two digits provide the arguments. This system uses a crossover operation capable of altering the length of the instruction string in a manner similar to the GP crossover. This approach was abandoned due to the high risk of the production of programs containing infinite loops.

Similarly, Perkis (Perkis, 1994) investigated methods where a variable length genome consisting of LISP S-Expression structures may be used to develop programs ‘designed’ around manipulating a stack of values. The use of S-Expressions in this work is typical of a GP system, yet the use of a two-point crossover operator is more similar to LGP. This work investigated a series of regression problems, and concluded allowing the developed programs use of a stack greatly increased the efficiency of the resulting programs when compared with tree based GP.

The GEMS system produced by Crepeau (Crepeau, 1995) is an extensive LGP-like system for the evolution of machine code for the Z80 processor. The GEMS system includes a reimplementation of the Z80 CPU in the form of an interpreter. This approach was used to evolve a program capable of writing a 'hello world' string to output ports on the processor. In this system, the LGP crossover acts only on the boundaries between instructions, enforcing the manipulation of complete instructions only.

VRM-M (Huelsbergen, 1996) is a similar approach using formal methods. It uses a linear representation of programs using strings of indivisible instructions and a crossover operation working upon such strings. It has been used to develop a program performing multiplication through iterations. The system is capable of creating this non-liner program from a selection of low level instruction primitives such as branches and jumps, and is shown to perform significantly better than random search.

The CGPS (later known as AIM-GP) approach developed by Nordin (Nordin, 1997) directly manipulates and executes machine code for the SPARC processor with no discrete interpretation or compilation stages; this approach is designed to proceed as efficiently as possible. As CGPS functions consist of machine code in memory, they may be invoked by a C function call by an appropriately typecast pointer. This is made possible by wrapping the evolved 'body' of a candidate solution program in a shell consisting of a boilerplate header and footer. The function of this header and footer is to ensure that the state of the stack and processor registers are in a consistent state to allow the body of the CGPS function to proceed and return successfully. The inclusion of these functions would be considered standard practice by a human programmer, but they must be added explicitly by the CGPS system to ensure the working of the system.

The SPARC architecture is a RISC architecture; the subset of instructions considered by CGPS are all of a fixed, shared length, therefore the crossover operation is fast to execute. In addition to evolving single linear programs, the CGPS is able to automatically develop a small number of reusable subroutines in a manner similar to the automatically defined functions of GP. This is achieved by limiting the method by which subroutine calling instructions may be generated; the only subroutine invocation instructions that may appear in a candidate solution are those which are targeted at a valid subroutine at an address stored in a predefined SPARC-specific register dedicated to that purpose. Jumps to arbitrary memory locations are not allowed. It is possible for CGPS programs to contain control flow statements which jump to positions within the 'body' of an evolved program, and therefore the potential for infinite loops is present. Execution of CGPS programs is bounded by imposing a maximum limit on the number of executed instructions before execution is forcefully aborted by an external monitoring process. The fitness measure is application specific and based upon considering the results of program execution on

the environment. This approach is suitable for many applications, including image processing, sound processing, robot control and plant control.

Kühling (Kühling, et. al. 2002) studied methods whereby Nordin's LGP-based AIM-GP method could be applied to CISC processors such as the Intel 386. This research attempted to use the LGP system to evolve a machine code program capable of performing a classification task. The focus of this research was to delegate the majority of the error-checking work to the physical CPU, reducing the need to dedicate processing time to the task of ensuring that the candidate programs were valid before execution. This research has been extended by performing hardware-tied LGP in parallel on GPU (Harding et. al., 2007) and embedded GPU (Wilson et. al., 2010).

(Orlov et. al., 2009) examined the use of LGP to evolve bytecode programs for the Java virtual machine. Direct measures were taken in the implementation of the recombination operators to ensure the consistency in the state of the stack. For example, a crossover operation may only replace a given code section with another code section of the same data type and with the same difference in stack frame depth. A thorough mathematical treatment of these conditions is given.

Further adaptations of GA and GP are possible, such as Cartesian Genetic Programming (Wilson et. al., 2008), where the LGP system is augmented by considering a directed acyclic graph of structures, rather than a strict line of structures.

The approach and fitness function described in this dissertation is primarily inspired by the approach taken by Jackson in *Evolution of Processor Microcode* (Jackson, 2005). This paper considers the use of a LGP system to evolve the microcode implementation of several machine code instructions for a processor. Jackson introduces the concept of supplying a varied set of 'hints' to the LGP system to provide a graduated fitness function capable of guiding the genetic system to assemble complex microprograms with the required semantics.

As far as I am aware, my dissertation is the first piece of research to consider the use of LGP towards performing the task of code generation in a compiler directly. That is, the use of LGP where the structures undergoing adaptation are a direct representation of the low level program output form, and the task to be performed is the evolution of a program with the same semantics as expressed by an IR, without manual reconfiguration of the fitness function between cases.

3.6 Applicability of Linear Genetic Programming

It has been observed that evolutionary computation techniques are routinely successful where in situations where a number of properties are evident (Poli et. al, 2008, page 111). The problem of code generation identified in this dissertation exhibits some of these traits:

Finding the size and shape of the ultimate solution is a major part of the problem: In this problem, the length of the (optimal) solution program is not specified in advance. We do not want to restrict the space of programs available for consideration by the LGP system by imposing any restrictions on the size or structure of the solution program.

Significant amounts of test data are available in computer readable form: In this problem, all input and output data is already in computer readable form. Additional fitness cases may be synthesized as required to increase the amount of input data available to the system.

There are good simulators to test the performance of tentative solutions to a problem, but poor methods to directly obtain good solutions: To test the performance of a candidate solution program, it is necessary to either produce an interpreter within which the programs may be run, or develop some kind of specialised hardware for the task. In this problem, developing an interpreter for an assembly-like language is not a difficult task. If the target architecture were a real, physical processor, then the processor itself may be used to execute the programs directly, quickly and exactly. However, executing candidate programs in an interpreter allows their execution to be monitored carefully, an approach which is used in this dissertation to guide the evolution of solution programs. There already exist very good algorithmic solutions for the task of code generation. However, this project aims to determine whether LGP may be used to develop programs of a higher quality than a certain class of naïve algorithms.

Small improvements in performance are routinely measured (or easily measurable) and highly prized: All factors affecting the suitability of a program, including program length, memory length and required processor time are all easily measurable. It is the intention that where a correct solution program of possibly poor quality may be produced initially, the LGP system may act to improve this program over time as a result of fitness pressure, i.e. the improvements in program quality, for measurements such as program length, are easily measurable and highly prized.

The problem of code generation does not exhibit the remaining traits as identified by this work, but they are worthy of comment:

The interrelationships among the relevant variables is unknown or poorly understood (or where it is suspected that the current understanding may possibly be wrong): In this problem, the semantics of the input parse tree are specified exactly, and the full semantics of all available instructions in the instruction set are known. However, highly complex programs may exhibit certain emergent behaviours to which purely algorithmic methods may be blind without significant analysis. An example of this may be the appearance of patterns in specific calculations, with the consequence that a specific alternate sequence of instructions may be used to achieve the same effect at a lower cost. Therefore, I consider it appropriate to attempt to perform or expedite the task of code generation with an unpredictable, stochastic evolutionary computation method.

Conventional mathematic analysis does not, or cannot, provide analytic solutions: In addition to the reasoning given above, analytic examination of the semantics of a program through formal methods or exhaustion is a highly involved process.

An approximate solution is acceptable (or is the only result that is ever likely to be obtained): In this task, an approximate solution is absolutely *not* acceptable. It is possible that many runs of the LGP system will result in the population becoming trapped in local minima that have somewhat similar, but not exactly the same, semantics as the input parse tree. If this were a symbolic regression problem, if the solutions were extracted at this point, they may be considered to be acceptable models of the input data. This is not the case with code generation: the only acceptable solution is one that exhibits all of the required semantics and does not otherwise damage the state of the system.

4 Solution Methodology

This section describes the methods devised to apply LGP in the task of performing the task of code generation in a compiler. These include the specification of the scope of the project and the associated experiments, the definition of a suitable fitness function to guide the evolutionary process, the methods used to detect when the task has been completed satisfactorily and the different models of evolution considered during experiment.

4.1 Scope of Experiments

The code generation device must be able to successfully transform input parse trees containing any valid combination of the following nodes. A valid combination is one where each node is saturated and has the maximum number of possible child nodes. A program is executed by evaluating its root node.

Readable name	Parse tree representation symbol	Number of children	Semantics
Semicolon – sequencing operator	;	2	Evaluate the program in the left child position, then evaluate the program in the right child position. Return the value returned by the right child.
Addition operator	+	2	Evaluate the program in the left child position, then evaluate the program in the right child position. Return the sum of these two values.
Subtraction operator	−	2	Evaluate the program in the left child position then evaluate the program in the right child position. Return the value returned by the left child subtract the value returned by the right child.
Multiplication operator	*	2	Evaluate the program in the left child position, then evaluate the program in the right child position. Return the product of these two values.
Protected division operator	/	2	Evaluate the program in the left child position, then evaluate the program in the right child position. If the value of the right child is zero, return 1. Else return (the value of the left child / the value of the right child).
Assignment operator	=	2	Evaluate the program in the right child position. Consider the left child node as a reference to a variable and assign the value of the right child to this variable.
Variable	<string>	0	When evaluated, return the value of the variable from memory.

Table 4.1: Table of parse tree nodes that may appear in an input program IR

The assignment operator can only have a variable node as its left child, and this variable node cannot refer to a constant.

The symbol table part of the IR contains the following information for each variable:

Name	Function
Symbolic name	The symbolic name of the variable as given in the high level language source file
Nature	Describes the scope of the variable. Can take one of the following values: INPUT: The variable exists outside the scope of the subroutine. This variable has a starting value that should be placed into memory during the initialisation stage. Operations on this variable should be reflected in the low level translation of the parse tree. OUTPUT: This variable exists outside the scope of the subroutine. This variable contains garbage after the initialisation stage. Operations on this variable should be reflected in the low level translation of the parse tree. INTERMEDIATE: This variable exists only within the scope of the subroutine. This variable contains garbage after the initialisation stage. Operations on this variable do not need to be reflected in the low level translation of the parse tree. SYMBOLIC_CONSTANT: This variable is initialised with a constant value during initialisation. The low level translation of the parse tree should not attempt to the change the value of this variable during execution. This category of variables includes integer literals (i.e. the integer literal '2' is a variable named '2' of type SYMBOLIC_CONSTANT with the constant value 2).

Table 4.2: Description of possible symbol table entries in an input program IR

The target architecture for the evolution of programs is a simple register machine. The machine contains two memory areas for storage of values: a register file and a symbolically addressable memory.

The register file consists of four of general purpose registers indexed from zero. The program counter is not available to the program for read or write access. The symbolically addressable memory contains an arbitrary number of cells that may be addressed by the symbolic name of a variable. Both the register file and the memory are available to programs for both read and write access at any time.

The following simplified, RISC-like low level instruction set is specified for the purposes of this project. An instruction in the low level language consists of an operation and a number of operands whose quantity and nature are defined by the choice of operation. This instruction set is orthogonal; any register may be used where a register argument is expected. The following instructions are defined:

<r> indicates that the argument is a register index
<s> indicates that the argument is a symbolic variable name
<v> indicates that the argument is an integer

ADD <r>a, <r>b, <r>c	**Addition** Calculates the sum of the values stored in registers **<r>a** and **<r>b** and stores the result in **<r>c**.
SUB <r>a, <r>b, <r>c	**Subtraction** Calculates **<r>a** - **<r>b** and stores the result in **<r>c**.
MUL <r>a, <r>b, <r>c	**Multiplication** Calculates **<r>a** * **<r>b** and stores the result in **<r>c**.
DIVP <r>a, <r>b, <r>c	**Protected division** If **<r>b** is non-zero, calculates **<r>a** divided by **<r>b** and stores the result in **<r>c**. Else, store **1** in **<r>c**.
LOADS <r>a, <s>b	**Symbolic load** Retrieves the value associated with the symbol **<s>b** and stores it in **<r>a**.
STORS <r>a, <s>b	**Symbolic store** Stores the current value of **<r>a** in the memory associated with the symbol **<s>b**.
LOADV <r>a, <v>b	**Direct value load** Stores the value **<v>b** in the register **<r>a**.

Table 4.3: Allowed instruction set of low level language instructions for code generator

The only data type used is the 64-bit signed integer. All variables in memory, symbolic constants, direct values and register locations are of this data type.

A set of ten input program IR of increasing complexity have been designed as target programs for the various code generation methods to attempt. They have the following properties:

Program A01: Assignment of a single constant to a variable.

```
a = 3
```

Program A02: Assignment of two constants to two variables.

```
a = 234; b = 1056
```

Program B01: Simple calculation. Addition of two input variables to be stored in one output variable.

```
a = b + c
```

Program B02: Two calculations with a non-commutative arithmetic operation.

```
a = b - (c + d)
```

Program B03: Complex multiple stage calculation involving a constant.

```
a = (18 * (c - d)) + b
```

Program B04: Complex multiple stage calculation with division.

```
a = ((c - 90) * (b + d)) / e
```

Program C01: Simple multiple stage calculation involving intermediate variable

```
b = c + d; a = b * e
```

Program D01: Calculation involving intermediate variables with great possible optimisation

```
d = b + c; e = b - c; a = d + e
```

Program D02: Calculation involving intermediate variables; difference of two squares

```
d = b + c; e = b - c; a = d * e
```

Program D03: Complex calculation involving intermediate variables

```
i1 = ((b + c) - d); i2 = ((b - c) + d); a = i1 * i2
```

A tree walking algorithm (described later) will be applied to each of the defined programs to produce a predictable program length for comparison with those produced by the evolutionary methods. A fully optimised 'perfect' solution as produced by a skilled human programmer has also been produced. A full list of these input program IRs with parse tree visualisation and human-developed 'perfect' solution are given as an appendix.

4.2 Application of Linear Genetic Programming

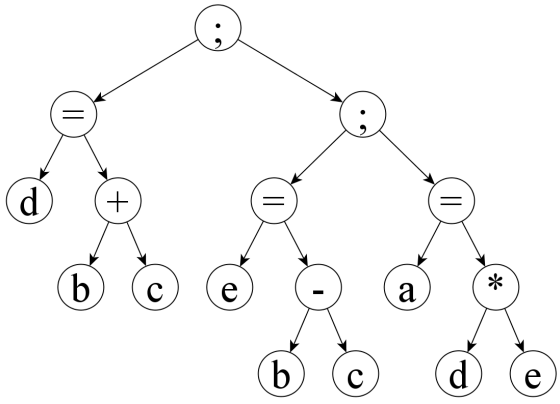
Two different methods of applying LGP are considered, which will be referred to as 'standard' and 'incremental'.

In the ‘standard’ method, the evolution system is used to attempt to evolve a single, complete solution program expressing the same semantics as the input program. This process is guided by the LGP heuristic and a fitness metric described later in the dissertation.

Candidate solutions will take the form of strings of atomic instructions written in the low level language. The population model used in this system is a steady state model where the number of candidate programs in the population will remain constant throughout. For each new program produced by a genetic operation, a tournament is used to choose a random program with low fitness (i.e. a tournament is held, selecting the lowest fitness from the participants) to be removed from the population. The tableau of genetic system parameters will be given as part of the design of experiments section.

In the ‘incremental’ method, the input parse tree from the IR is mechanically transformed into a series of smaller programs that, when executed sequentially, have the same semantics as the complete program. A solution program for each of these subprograms is evolved in turn, and these are concatenated to produce a solution to the input program.

For example, consider the complex program shown below:




```
d = b + c;
e = b - c;
a = d * e
```

Figure 4.4: Visualisation of complex program to be handled by incremental method

This program is transformed by the ‘incremental’ method by taking each interior node in turn and transforming these into isolated, smaller programs. The interior nodes furthest down the graph are considered first (i.e. the high level program fragments with the highest precedence).

Here, the transformation begins with the + node in the lower left of the graph. This is extracted into a separate program with an assignment node at its root, a new temporary variable as its left child, and the extracted program as its right child. The extracted subtree from the original program is replaced with a node referencing the same temporary variable. It can be seen that executing the new subprogram followed by the altered program does not result in a change in semantics if the temporary variable is not considered to be a critical part of the program; this variable is added to the symbol table (all program fragments share a global symbol table) as an intermediate variable.

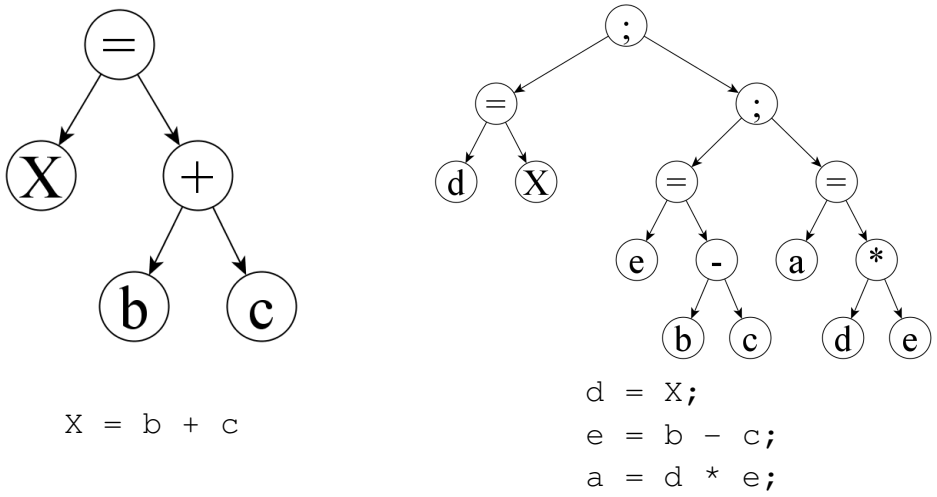


Figure 4.5: Produced program fragment and the resulting modified program tree after the first extraction by incremental method

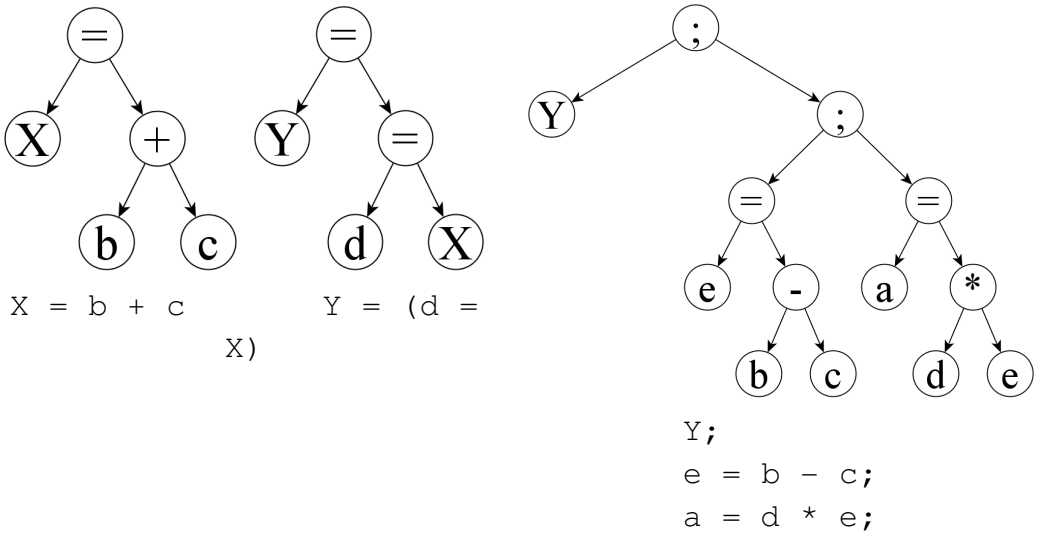


Figure 4.6: Produced program fragments and the resulting modified main program tree after the second extraction by incremental method

This process continues until a number of subprograms equal to the number of interior nodes in the original program have been created.

The evolutionary system is then tasked with evolving solution programs to each of these subprograms in turn, and then concatenating the resulting solutions into a composite solution program which will have the same semantics as the original input program.

I hypothesise that, with increasing input program complexity, evolving a large number of smaller programs using the 'incremental' method will result in a considerably lower processor time requirement would be required using the 'standard' method.

4.3 Comparison to Naïve Algorithm

A simple tree walking algorithm has been developed capable of mechanically generating low level code given an input program IR. The following pseudocode algorithm is used to construct instruction strings:

```
TreeWalkingCompiler(node, register)
        IF node is interior node
                IF node is semicolon
                        TreeWalkingCompiler(left_child, register)
                        TreeWalkingCompiler(right_child, register)
                ELSE IF node is assignment
                        TreeWalkingCompiler(right_child, register)
                        output [STORS register left_child]
                ELSE IF node is calculation (one of +,-,*,÷)
                        TreeWalkingCompiler(left_child, register)
                        TreeWalkingCompiler(right_child, register + 1)
                        output [calc register register+1]
                ENDIF
        ELSE
                output [LOADS register left_child]
        ENDIF
END
```

Listing 4.7: Pseudocode for tree walking computer algorithm

This code generator is capable of generating code for all of the defined input programs; it requires a virtual machine register file of the same length as the depth of the most complex calculation (three registers for programs B02, B03 and B04).

This code generator does not perform any kind of analysis or optimisation on the input program. For example, when processing programs C01, D01, D02 and D03 it will generate code that accesses the intermediate variables, and for programs D01, D02 and D03 it will generate code that explicitly performs every calculation as specified.

This tree walking algorithm will be applied to each of the defined programs to produce a predictable program length for comparison with those produced by the evolutionary methods. A fully optimised 'perfect' solution as produced by a skilled human programmer has also been produced.

4.4 Refinement Stage

An additional stage of processing is proposed to investigate the ability of LGP to improve solution programs that have already been found.

The 'refinement' stage occurs after a solution program has been developed by the evolutionary system. A new population of random instruction strings is produced, with a predefined fraction of the population initialised as copies of the previously identified satisfactory solution program. The termination criterion of this new system is set to return the best-of-run program after a predefined number of new candidate solution creations. The remaining parameters of the evolutionary system, such as evolutionary system parameters and instruction set, remain unchanged.

The evolutionary system will attempt to breed fitter programs and, given that solution programs are already present in the genetic population, these fitter programs will most likely be modified copies of the solution programs improved by application of the genetic operations. As a result, it is believed that the refinement stage will result in the development of programs of successively greater quality (in these experiments, shorter length).

4.5 Design of Fitness Function

In order to apply evolutionary methods such as LGP in solving a problem, it is necessary to provide a 'fitness function'. The fitness function provides the mapping from candidate solutions to fitness values, allowing the evolutionary system to determine the suitability of a candidate solution. With this mapping, the system is able to identify the most fit candidate solutions from the population and bias the reproduction stages so successive generations will tend to contain a high concentration of modified copies of these most fit candidate solutions.

For the LGP system to function effectively, the fitness function must have the following properties:

The fitness function should be of a generally continuous nature, with successively decreasing output values as a candidate solution becomes 'closer' to an acceptable solution. Without the ability to sort the population of candidate programs by their perceived correctness, the LGP will not be able to introduce bias in the selection of programs for the reproduction stage. As a result, the model of gradual program improvement as useful genetic material is identified and propagated will not hold.

The fitness function should be sufficiently general to allow its use in the generation of code for any possible input IR, and not require significant human intervention or configuration to adapt to a different input IR. This requirement ensures that the LGP-based code generator is suitable replacement for an algorithmic code generator.

The fitness function should contain as little *a priori* information about the instruction set as possible, which will ensure that the LGP-based code generator can be extended automatically by inserting additional instruction semantics.

The task of the fitness function in this project is to measure the degree of semantic correlation between the input program IR and an output low level candidate solution instruction string. Considering only the final values of the variables after execution has terminated is not sufficient to meet the above requirements, as this does not allow for the case where a candidate solution performs all the calculations required of it, but does not store these values to memory after they are calculated (or they are subsequently overwritten with garbage values). Such a program is clearly 'close' to the semantics of the input program, but an examination of the output values would dismiss it as of very little suitability.

Although it may be possible to devise a common language capable of expressing the exact semantics of programs in both the input and output program forms, it would be difficult to quantify the degree of similarity from descriptions in this language, and then transfer this into the form of a computer program.

Instead, a sampling of the possible values of the input variables is considered. Each sample is a possible instantiation of the input state of the program before it is executed. This is analogous to compiler testing. It is believed that through a representative sampling of the input values, enough data will be available to construct a sufficient expression of the state mapping defined by the program. The number of samples will affect the accuracy of the expression, and therefore the accuracy of the output program. If there are too few samples, then the resulting program may exploit properties specific to the sample set. Increasing the number of samples will increase the time required to calculate the fitness of a candidate solution program.

For each sample set of input values, a fitness case is built by evaluating the input parse tree program. Each fitness case contains the values of the symbolic variables before execution, after execution, and a full record of all the intermediate evaluations

that took place during evaluation. In addition, the number of calculations that were required, in total, to produce the result value is recorded as a heuristic measure of the complexity of calculation.

To calculate the degree of semantic correlation, each candidate program is tested against each fitness case in turn. A virtual machine instance is created and reset; the input symbolic variable value set is copied from the fitness case into the variable machine symbolic variable memory, and the execution started. When execution terminates, the final values of the target variables in the memory of the virtual machine are compared against those from the fitness case. If the values of all target variables are exactly the same, no penalty is applied. If the value of a variable differs from its target value, a constant penalty is given together with a variable amount of penalty as a function of the error. In addition, a fitness penalty will be given proportional to the length of the candidate program, causing the evolutionary system to favour programs of a smaller length.

During the execution of the candidate program, a line by line record of the execution is produced. This record contains, for each executed arithmetic instruction, the operation that was performed, the register locations and values of the operands used and the register location and value produced as a result. For symbolic loads, only the destination register location and value is stored. It is possible to perform some analysis of the program without this record, but this may become complicated if conditional or jumping instructions are applied. With the introduction of the fitness cases as fixed points in the input space, it makes sense to continue in this vein by analysing the exact actions taken as a result of these input sets.

The complete record allows for the construction of a timeline showing the values of the registers after each instruction execution. If the registers are not reset to a known value before execution begins, this record shows which registers hold determinate values (which may be assumed to be of some use), or indeterminate garbage values (which will not be the same between executions, and therefore should not be relied upon in the output program).

Fitness bonuses and penalties are activated during analysis of the behaviour of the candidate program. These bonuses are designed to ‘coax’ the evolution of candidate programs towards those which a human programmer would consider productive.

The following productive behaviour is rewarded:

- Reading the value of a symbolic variable.
- Writing to the value of a variable that may change during execution.
- Writing to the value of a variable that must change during execution.
- Reading from a register whose value is determinate at the time of reading.
- Performing a calculation that results in a value that was encountered during construction of the corresponding fitness case. An added bonus is applied if

the low level instruction correlates to the construct in the parse tree that was used.

The following counterproductive behaviour is penalised:

- Writing to a register and never subsequently reading it.
- Writing to a symbolic variable (other than one designated as an output) and never subsequently reading it.
- Writing to a register twice in succession without reading it in the interval.
- Reading from a register containing an indeterminate value.
- Performing a calculation upon indeterminate values.
- Writing an indeterminate value to any register.
- Writing an indeterminate value to any symbolic variable.

It is hypothesised that the crossover operation will combine programs that are correct 'up to a point' with genetic material from elsewhere to produce child programs that provide further functionality than either of their parents. It is also hypothesised that the mutation operations will act to 'repair' programs by removing or rewriting counterproductive instructions in programs, hence increasing their suitability.

The specification of a large number of fitness modifiers is intended to provide a more gradual fitness landscape. If only the error in the values of target variables is considered, the mapping from input candidate program space will be discontinuous. In such a space, many programs will share the same fitness value, and the evolutionary system will be able to offer little improvement.

With the above modifiers, there is the risk that the system may produce a program that calculates a useful value at some point during execution, and then attempt to improve such a program by repeating the segment that triggers the reward, resulting in a program that does nothing more than calculate the same (albeit useful, or even necessary) value multiple times. Given time, such programs may dominate the candidate program population. To prevent this, the system can be configured to allow these rewards to be awarded only finitely many times per action, or per expected appearance of a result value. The complexity heuristic is designed so that a candidate program that correctly implements a multiple stage calculation is deemed to be more suitable than a candidate program that performs a simple calculation multiple times.

A 'hit' is recorded for a given fitness case if the resulting value for each variable is equal between the resulting state of the virtual machine memory after execution has terminated and the final state of the parse tree evaluation. If a 'hit' is recorded for each fitness case in the training set, the candidate program is tested against each fitness case in the test set. If a 'hit' is recorded for all fitness cases in the test set, the candidate program is judged to be a satisfactory solution: the evolutionary system terminates and returns the candidate program.

4.6 Genetic Operations

There are three operations available to the evolutionary system to produce new candidate solution programs through manipulation of existing candidates. These are typical operators used in many LGP applications. During the reproduction stage, these actions are selected randomly according to rate parameters given to the evolutionary system.

- Rate of crossover – The probability that the reproduction stage will result in new programs being produced by the crossover operation.
- Rate of mutation – The probability that the reproduction stage will result in new programs being produced by the mutation operation.
- Rate of reproduction – The probability that the reproduction stage will result in new programs being produced by the reproduction operation.

Tournament selection will be used to select programs from the candidate solution population for application to the available genetic operations. In tournament selection, a set number of programs are randomly selected from the population to participate in the tournament. From these programs, the M programs with the highest fitness values are passed to the genetic operation.

The population model used in this system is a steady state model where the number of candidate programs in the population will remain constant throughout. For each new program produced by a genetic operation, a tournament is used to choose a random program with low fitness (i.e. a tournament is held, selecting the lowest fitness from the participants) to be removed from the population

The crossover operation combines the contents of two source instruction strings to produce two new instruction strings, which are then inserted into the population as two new individuals. Transition points are randomly placed within the two instruction strings. Then, new strings are constructed by copying instructions from the first string until the first transition point, then copying instructions from the second string starting at the first transition point until the second transition point, then copying the remaining instructions from the first string starting from the second transition point until the end.

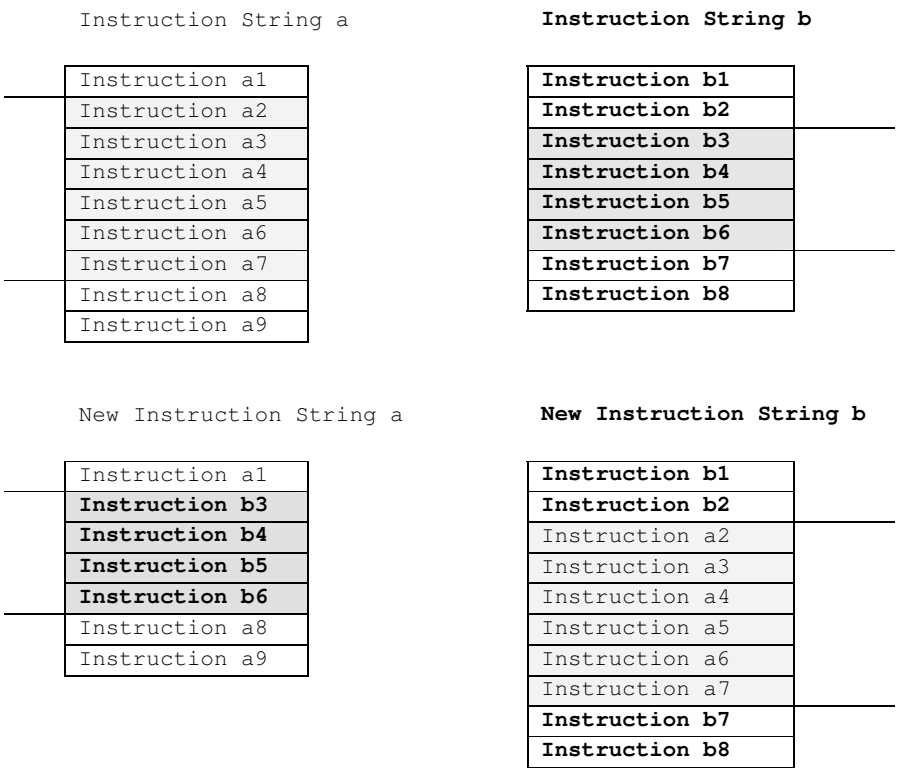


Figure 4.8: Results of the crossover operation used in the experiment on two instruction strings

The effect of the mutation operation is select one operation at random from 'insertion', 'deletion' and 'alteration'. Insertion inserts a new random instruction at a random position within the existing instruction string. Deletion selects a random instruction from the existing instruction string and removes it (deletion is not available if the instruction string only consists of a single instruction). The function of alteration is to randomly permute some part of an existing instruction chosen at random from the instruction string. One component of the instruction is chosen at random from the available components (as described in the instruction set listing previously). If an operand component is chosen, it is replaced with an operand of compatible type. If the operation component is chosen, then the all components of the instruction are reinitialised.

4.7 Experimental Measurements

The complexity of the required calculations increases with each program. With this increasing complexity, additional opportunities for optimisation become available, such as the omission of unnecessary calculations, or the 'folding' of intermediate calculations into the program (hence obviating the need to store and load from the intermediate variables). For each of these input programs, the software has been used

to calculate two primary metrics: 'Computational Effort' and the distribution of solution program lengths.

The 'Computational Effort' (E_I), for a given program, evolutionary system parameter set and instruction set, is the minimum number of low level language instructions that must be considered (executed in the low level virtual machine) to be able to evolve a solution program with 99% probability of success. When a new candidate instruction string is created by any method, its length in instructions is tallied as 'considered'. It is an adaptation of the Computational Effort (E) measure used by Koza (Koza, 1992), where the number of complete candidate solutions is used. A measure of required program quanta (LISP program nodes in the case of the S-Expressions used by Koza) was proposed by Koza, but was not implemented due to insufficient processing capacity and other factors. This different measure is required to compare the difficulty of evolving solutions using the 'standard' method and the 'incremental' method, where the number of candidate solutions produced cannot be used as a measurement.

E_I is used as an empirical measure of the difficulty of evolving a solution program using the given parameters: a higher E_I indicates that more processing time is required to evolve a solution. For a series of programs of increasing complexity, E_I can be used to identify trends in processing requirements. E_I is calculated as follows:

Koza suggests that multiple independent runs of the evolutionary system should be attempted to minimize the effect of premature population convergence to a sub-optimal solution. Over a large number of runs (200 in this project), we measure the number of instruction considered to evolve each solution program. If a run does not produce a solution program within the maximum number of allowed creations, that run is aborted.

These measurements are then collected to compute the cumulative probability $P(i)$ of a solution program being produced as a function any given number of instruction considerations i. The probability of producing a solution program at least once in R runs can be calculated as $1 - (1 - P(i))^R$. If the desired probability of success z is fixed at a high value, here 99%, then the number of required runs can be calculated by: (where the brackets denote the ceiling function)

$$R(z,i) = \left\lceil \frac{\log(1-z)}{\log(1-P(i))} \right\rceil$$

This function defines thresholds where with increasing $P(i)$, the number of required runs decreases. For example, if $P(i)$ is 0.68 then four independent runs are required; if $P(i)$ is 0.78 then three independent runs are required and if $P(i)$ is 0.90 then two independent runs are required. Multiplying $R(z, i)$ by i gives the total number of

instructions that must be considered if each run is aborted after considering i instructions. EI is the minimum value of i . $R(z, i)$ over all i for z = 99%.

This can be visualised in a performance curve as shown below:

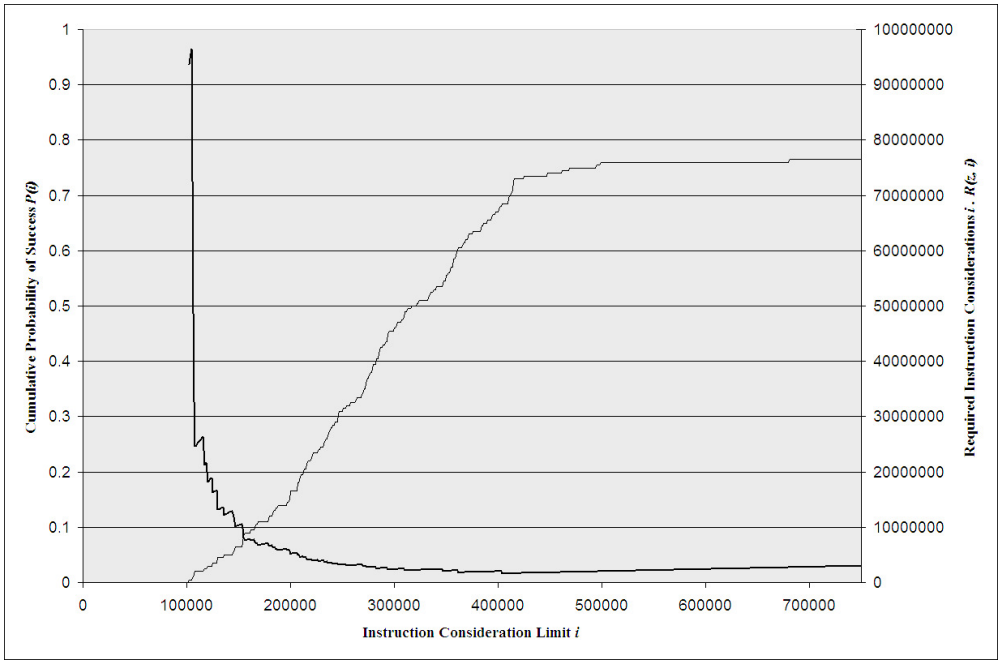


Figure 4.9: Example of performance curve showing required number of instruction considerations for a given cumulative probability

In this graph, the cumulative probability P(i) is shown as the curve rising from left to right. The required number of instruction considerations i . R(z, i) is shown as the heavier curve generally falling from left to right. As the cumulative probability increases beyond the thresholds given by the R(z, i) function, the number of independent runs necessary to produce a solution program decreases, giving the sawtooth nature to the i . R(z, i) curve. This curve hits a minimum at i = 404000, where four runs are necessary, giving an E_I value of 1616000.

For the 'standard' method, E_I is calculated as above over 200 runs with z = 99%. For the 'incremental' method, 200 attempts at evolving the input program are used, with the sum of the E_I values for each subprogram taken as the E_I value for that attempt. It is not likely that E_I values for a given input program are directly comparable between the two methods, but the trends detectable when considering a series of successively more complex programs are of value. Computational effort is not considered where the refinement operation is applied.

The second metric used to evaluate the evolutionary methods is the distribution of solution program lengths.

For both the ‘standard’ and the ‘incremental’ methods, the evolutionary system has been used to develop 200 solution programs for each input program. For each of these 200 solution programs, the refinement operation has been applied 5 times to produce a collection of 1000 solution programs. The distribution of solution program lengths within these sets can be used as a point of comparison, together with the program lengths produced by the tree walking algorithm and the ‘optimal’ solution produced by a human programmer.

4.8 Tableaux of Evolutionary System Parameters

For the measurement of computational effort by ‘standard’, the following parameters are used:

Population size:	1000
Population initialisation modus:	Ramp of program lengths
Initial maximum low level program size:	50
Initial minimum low level program size:	5
Maximum allowed program recreations before abort:	25000
Re-attempt crossover if program length over:	500
Tournament size for crossover:	5
Tournament size for reproduction:	5
Tournament size for mutation:	5
Tournament size for deletion:	5
Maximum number of niche pre-emption repeat runs:	1 (one run only: no pre-emption)
Rate of crossover:	0.49
Rate of mutation:	0.49
Rate of reproduction:	0.02
Fitness case training set count:	20
Fitness case test set count:	10
Number of registers in virtual machine:	4
Low level instruction set:	`ADD`, `SUB`, `MUL`, `DIVP`, `LOADS`, `LOADV`, `STORS`
Low level instruction random selection rates:	Equal distribution among all instructions.
Hits measurement:	Number of training fitness cases successfully emulated by low level candidate program.
Success criterion:	Hits equal to number of training fitness cases.
Failure criterion:	Number of allowed program recreations exceeded.

Table 4.10: Tableau of parameters for computational effort experiments using ‘standard’

For the measurement of computational effort by 'incremental', the following parameters are used for each subprogram fragment:

Population size:	1000
Population initialisation modus:	Ramp of program lengths
Initial maximum low level program size:	20
Initial minimum low level program size:	2
Maximum allowed program recreations before abort:	25000
Re-attempt crossover if program length over:	50
Tournament size for crossover:	5
Tournament size for reproduction:	5
Tournament size for mutation:	5
Tournament size for deletion:	5
Maximum number of niche pre-emption repeat runs:	1 (one run only: no pre-emption)
Rate of crossover:	0.49
Rate of mutation:	0.49
Rate of reproduction:	0.02
Fitness case training set count:	20
Fitness case test set count:	10
Number of registers in virtual machine:	4
Low level instruction set:	`ADD`, `SUB`, `MUL`, `DIVP`, `LOADS`, `LOADV`, `STORS`
Low level instruction random selection rates:	Equal distribution among all instructions.
Hits measurement:	Number of training fitness cases successfully emulated by low level candidate program.
Success criterion:	Hits equal to number of training fitness cases.
Failure criterion:	Number of allowed program recreations exceeded.

Table 4.11: Tableau of parameters for computational effort experiments using 'incremental'

Although, different program length ramps are used for the standard and incremental methods, I believe that this has limited effect on the number of instruction considerations necessary.

For the measurement of program length by 'standard', the following parameters are used when evolving the initial length of a program:

Population size:	1000
Population initialisation modus:	Ramp of program lengths
Initial maximum low level program size:	50
Initial minimum low level program size:	5

Maximum allowed program recreations before abort:	25000
Re-attempt crossover if program length over:	500
Tournament size for crossover:	5
Tournament size for reproduction:	5
Tournament size for mutation:	5
Tournament size for deletion:	5
Maximum number of niche pre-emption repeat runs:	1000 (niche pre-emption is used to ensure the eventual creation of an acceptable candidate)
Rate of crossover:	0.49
Rate of mutation:	0.49
Rate of reproduction:	0.02
Fitness case training set count:	20
Fitness case test set count:	10
Number of registers in virtual machine:	4
Low level instruction set:	`ADD`, `SUB`, `MUL`, `DIVP`, `LOADS`, `LOADV`, `STORS`
Low level instruction random selection rates:	Equal distribution among all instructions.
Hits measurement:	Number of training fitness cases successfully emulated by low level candidate program.
Success criterion:	Hits equal to number of training fitness cases.
Failure criterion:	None – continue until success criterion is met.

Table 4.12: Tableau of parameters for program length experiments using 'standard'

For the measurement of program length by 'incremental', the following parameters are used when evolving the initial length of a program:

Population size:	1000
Population initialisation modus:	Ramp of program lengths
Initial maximum low level program size:	20
Initial minimum low level program size:	2
Maximum allowed program recreations before abort:	25000
Re-attempt crossover if program length over:	50
Tournament size for crossover:	5
Tournament size for reproduction:	5
Tournament size for mutation:	5
Tournament size for deletion:	5
Maximum number of niche pre-emption repeat runs:	1000 (niche pre-emption is used to ensure the eventual creation of an acceptable candidate)
Rate of crossover:	0.49

Rate of mutation:	0.49
Rate of reproduction:	0.02
Fitness case training set count:	20
Fitness case test set count:	10
Number of registers in virtual machine:	4
Low level instruction set:	`ADD`, `SUB`, `MUL`, `DIVP`, `LOADS`, `LOADV`, `STORS`
Low level instruction random selection rates:	Equal distribution among all instructions.
Hits measurement:	Number of training fitness cases successfully emulated by low level candidate program.
Success criterion:	Hits equal to number of training fitness cases.
Failure criterion:	None – continue until success criterion is met.

Table 4.13: Tableau of parameters for program length experiments using 'incremental'

For the refinement stage, the programs produced by either incremental or standard are treated by an LGP system using the following parameters:

Population size:	1000
Population initialisation modus:	X * population size of copies of previously found solution program. ((1 – X) * population size) ramp of program lengths.
Fraction of population initialised with copies of previous solution:	0.025 (25 programs)
Initial maximum low level program size:	50
Initial minimum low level program size:	5
Maximum allowed program recreations before abort:	25000
Re-attempt crossover if program length over:	50
Tournament size for crossover:	5
Tournament size for reproduction:	5
Tournament size for mutation:	5
Tournament size for deletion:	5
Maximum number of niche pre-emption repeat runs:	1 (one run only: no pre-emption)
Rate of crossover:	0.49
Rate of mutation:	0.49
Rate of reproduction:	0.02
Fitness case training set count:	20
Fitness case test set count:	10
Number of registers in virtual machine:	4
Low level instruction set:	`ADD`, `SUB`, `MUL`, `DIVP`, `LOADS`, `LOADV`, `STORS`

Low level instruction random selection rates:	Equal distribution among all instructions.
Hits measurement:	Number of training fitness cases successfully emulated by low level candidate program.
Success criterion:	Hits equal to number of training fitness cases. Continue until number of allowed recreations is exceeded.
Failure criterion:	None – continue until success criterion is met.

Table 4.14: Tableau of parameters for refinement stage population length experiments

5 Design and Implementation of Software

This section describes the software that has been designed and created to perform the experiments described in the previous section. An outline of the design of the software is given and each phase of the software is then examined in detail.

5.1 Outline of Software

The software used to perform the experiments consists of a single command-line executable written in C++. The software requires the user to supply command-line arguments instructing it which type of experiment to perform, and which input program on which to attempt code generation. The parameters of the LGP system, such as population size, recombination operation rates, and the number of repeat experiments to perform, are stored in parameter files.

Execution of the software uses the following phases.

- Parse the command-line arguments passed to the program.
- Read an input program in plain text from a text file into memory.
- Construct the in-memory representation of the parse tree and symbol table.
- Read and parse the parameters to the LGP system from the parameters file.
- Prepare the instruction set that will be made available to the LGP system.
- Perform the experiment as defined by the command-line arguments.
- Perform refinement stage using the program generated by the above experiment.
- Repeat evolution experiment and/or refinement stage as defined by the parameter file.

The program performs the required experiments and writes its. It can optionally output information regarding the current best-of-run program to the console window during execution, or it can be set to act in a fully interactive mode, pausing whenever a ‘breakthrough’ has been made by the program. This mode allows the user to step through a simulation of the low level virtual system running the resulting low level program. This mode is suitable for demonstrations.

Throughout the program, the following `typedef` are used:

`BigInteger`
: Signed integer type used throughout the program. An alias of `long long int` (64-bit signed integer). The large range of values is

necessary to handle the large random numbers produced as a result of repeated attempted multiplication of numbers. It appeared to be very difficult to evolve programs without the availability of 64-bit values. It is possible to `typedef BigInteger` to a well-defined C++ arbitrary precision signed integer data type such as those provided by GMP, however `long long int` was chosen for fast execution.

`VM_TYPE`
: Signed integer type used in the low level virtual machine for all values. An alias of `BigInteger`.

`PT_VALUE_TYPE`
: Signed integer type used in the high level parse tree interpreter for all values. An alias of `BigInteger`.

`PT_COMPLEXITY_TYPE`
: Unsigned integer type used in the high level parse tree interpreter for the values of the complexity heuristic. An alias of `unsigned int` (32-bit unsigned integer).

5.2 Command-Line Arguments

The software requires the user to supply command-line arguments instructing it which type of experiment to perform, the location of the input program on which to perform code generation, and the location of the parameter files.

The program has the following invocation syntax:

```
evolve.exe [source_program_file] [analysis_type] [parameters_file]
           [parameters_file2] [output_file]
```

`[source_program_file]` is a string containing the location of the input program on the hard drive. Input programs are stored in a plain text format for easy modification by the user. The full syntax of this file format is described later in this dissertation.

`[analysis_type]` is a string indicating what form of experiment is to be performed. It can take one of the following values:

`attempt_standard`
: Perform evolution using the 'standard' model using the parameters in `[parameters_file]`.

`attempt_standard_with_refine`
: Perform evolution using the 'standard' model using the parameters in `[parameters_file]`, followed by the refinement stage using the parameters in `[parameters_file2]`.

`attempt_incremental`
: Perform evolution using the 'incremental' model using the parameters in `[parameters_file]`.

`attempt_incremental_with_refine`
: Perform evolution using the 'incremental' model using the parameters in `[parameters_file]`, followed by the refinement stage using the parameters in `[parameters_file2]`.

`tree_walking_compiler`
: Perform code generation of the input program using the tree walking algorithmic compiler, using the parameters in `[parameters_file]`.

`tree_walking_compiler_with_refine`
: Perform code generation of the input program using the tree walking algorithmic compiler, using the parameters in `[parameters_file]`, followed by the refinement stage using the parameters in `[parameters_file2]`.

`[parameters_file]` is a string containing the location of the parameters file containing the parameters to use during the primary stage of evolution. The full file format of this file is described later in this dissertation.

`[parameters_file2]` is a string containing the location of the parameters file containing the parameters to use during the refinement stage of evolution. The full file format of this file is described later in this dissertation.

`[output_file]` is a string containing the location of the file to which the experiment results will be saved. If this file already exists, the results are appended to the end of the existing file.

Example:

```
evolve.exe program_a01.txt attempt_standard_with_refine
             parameters_attempt_standard_for_program_length.txt
             parameters_attempt_refinement.txt
             attempt_standard_with_refine_for_program_length_program_a01.txt
```

This example instructs the software to attempt to evolve a solution to the input program stored in the file `program_a01.txt` using the 'standard' model of evolution followed by the refinement stage. The parameters for the 'standard' phase of evolution are stored in the file `parameters_attempt_standard_for_program_length.txt` and the parameters for the refinement stage are stored in the file `parameters_attempt_refinement.txt`. The output results are stored in the file `attempt_standard_with_refine_for_program_length_program_a01.txt`.

5.3 Plain-text Input Program Parser

The software contains a very basic parser and lexical analyser similar to that found in a regular compiler. This system is has been included to facilitate the easy specification of input IRs in a familiar high level C-like language, instead of error-prone manual construction of a parse tree.

The parsing of the input file is performed by the function `ParseTreeSourceProgram_ConstructSourceProgramFromFile()`. This function takes as its argument the location of the input program file as a string.

The input file format is a simple, human readable plain-text format:

```
--------------
Calculation involving intermediate variables.

INPUT             c d e
OUTPUT            a
INTERMEDIATE      b
SYMBOLIC_CONSTANT 2

PROGRAM
  b = (c + d);
  a = ((b * e) + 2)
ENDPROGRAM
--------------
```

Listing 5.1: Sample plain text input program file

The `INPUT`, `OUTPUT`, `INTERMEDIATE` and `SYMBOLIC_CONSTANT` lines define the symbol table used in the main program. Each of the lines contains a space-separated list of variable names. Variable names may consist of character strings (a-z, A-Z, 0-9) of any length, but must not start with a number. Additional `INPUT`, `OUTPUT`, etc. lines may be used to construct long lists of variables. These categories map exactly to those symbol table variable kinds defined in the Scope subsection of the Solution Methodology. Numeric literals used in the program body (such as 2) must be declared in advance in the `SYMBOLIC_CONSTANT` section.

The input program is given between the `PROGRAM` and `ENDPROGRAM` statements; the file parser concatenates all the non-whitespace characters between these lines to form the input program in its high level form. This input language recognises variable names, the assignment (=), sequencing (;), addition (+), subtraction (-), multiplication (*) and protected division (/) operators, and parentheses. The sequencing operator enforces the order of evaluation between statements, but the program must not end on a semicolon. Space characters and newlines are ignored. There exists operator precedence mirroring that of C, though this can be overridden with parentheses to explicitly lay out the parse tree.

Any characters outside the `PROGRAM` and `ENDPROGRAM` pair that are not part of a symbol table definition line are ignored as comments (such as the dashed lines and the program description).

After the parser has exhausted the input file, the input program is stripped of white space and converted into Polish notation. The program defined by the above file is represented as the string `; = b + c d = a + * b e 2` . This string is then used to build the in-memory representation of the parse tree part of the input program IR. This is a straightforward operation, as the Polish notion form of the input program is a preorder depth-first traversal of the expected parse tree. Each language symbol has a direct mapping to one of the parse tree nodes defined previously. Variable nodes (including numeric literals) are created as necessary. This input file will result in the following parse tree:

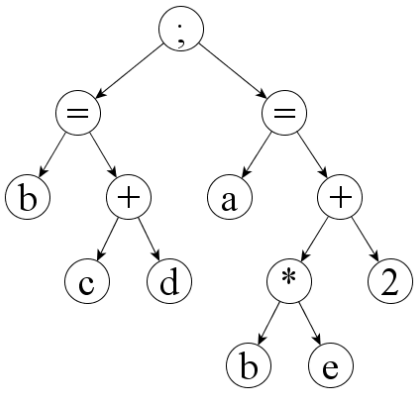


Figure 5.2: Parse tree visualisation for sample IR

5.4 In-memory Representation of IR

An IR is stored in memory as an instance of the `ParseTreeSourceProgram` structure. This structure has four members:

```
std::map<std::string,  ParseTreeInteriorNodeSetMember *> interior_node_set;
std::map<SYMBOL_TABLE_KEY, ParseTreeLeafNodeSetMember *>     leaf_node_set;

ParseTreeNode *target_program;

SymbolTable *symbol_table;
```

The parse tree component is stored in the `ParseTreeNode` pointed to by `target_program`, together with the maps `interior_node_set` and `leaf_node_set`.

These maps act as storage for lists of instances of `ParseTreeInteriorNodeSetMember` and `ParseTreeLeafNodeSetMember`. There exists one instance of `ParseTreeInteriorNodeSetMember` for each possible parse tree language feature. Each

ParseTreeInteriorNodeSetMember acts as the 'archetype' for the language feature, holding data that is common to all appearances of the language feature. This information includes data describing how the language feature should be interpreted by the parse tree high level interpreter and how the language feature should be represented as a string when the interactive mode is enabled. Similarly, there exists one instance of ParseTreeLeafNodeSetMember for each possible terminal node in the parse tree. Here, the only possible terminal nodes are the variables and constants defined in the symbol table. Each ParseTreeLeafNodeSetMember acts as the 'archetype' for the terminal node, holding data that is common to all appearances of the terminal node. This information primarily links the node type to variables in the SymbolTable.

These ParseTree…NodeSetMember structures collectively define the full 'catalogue' of possible nodes of which a parse tree may be constructed. They are used to prevent duplication of information in the parse tree: each instance of a node representing assignment is registered as an instance of the 'archetype' ParseTreeInteriorNodeSetMember representing the assignment operation.

The parse tree itself is held within the ParseTreeNode pointed to by target_program. The structure ParseTreeNode is used to represent a node in the parse tree in memory. target_pointer points to the root node of the program; in the previous example, this is the semicolon node. An instance of ParseTreeNode has members indicating whether it represents an interior or leaf node, and a pointer to an 'archetype' ParseTree…NodeSetMember describing which one of the possible nodes from the catalogue it is. Interior nodes contain references to their child nodes as pointers to other instances of ParseTreeNode. Functions acting on programs stored in ParseTreeNode trees are implemented as recursive functions.

The symbol table is stored in the instance of the structure SymbolTable pointed to by symbol_table. The symbol table is implemented as a map mapping instances of SYMBOL_TABLE_KEY to instances of the structure SymbolicVariableData. SYMBOL_TABLE_KEY is a typedef for unsigned integer type, and acts a primary key to the symbol table. SYMBOL_TABLE_KEY is used throughout the program wherever a reference to a variable is needed. A SymbolicVariableData instance holds a row of the symbol table; it contains the following information:

- The symbolic name of the variable as a string.
- A flag indicating if the variable may be written to.
- A flag indicating if the variable may be read from.
- A flag indicating if the semantics of the IR expect the value of this variable to assume some new value during execution of the program.
- A flag indicating if the semantics of the IR expect the value of this variable to remain constant during execution of the program.

- A flag indicating if the contents of the variable are known at the time of program initialisation.
- A flag indicating if the variable is symbolic constant (numeric literal) with a constant, known value implied its string name.
- The value of the variable, if it is a constant.

The symbol table can be queried by `SYMBOL_TABLE_KEY` or variable symbolic name.

5.5 Parameters Available to the LGP System

The parameters to the LGP system are stored externally as plain text files. The user must indicate the location of two parameter files, one for the main evolution stage and one for the refinement stage, when invoking the software. An input file consists of a series of string – value pairs. The full listing of possible parameters is given as an appendix.

A set of parameters to the LGP system is stored in memory as an instance of the `EvolutionSystem_Parameters` structure.

5.6 Instruction and Instruction Set Representation

In instruction in the low level language is stored as instance of the `Instruction` structure. The Instruction structure contains a `VM_INSTRUCTION_OPERATION` indicating which operation from the available set of operations the instruction represents, and an array of `InstructionOperand` instances holding the operands of the instruction.

`VM_INSTRUCTION_OPERAND` is an unsigned integer key to an enumeration of all the possible instructions in the low level language.

As not all instructions in the low level language have operands of the same type, `InstructionOperand` contains an instance of each of the possible types of operand value: `unsigned int` register index, `SYMBOL_TABLE_KEY` variable reference and `VM_TYPE` direct integer value. The low level interpreter simply chooses the instance of the appropriate type when the instruction is interpreted.

Candidate programs (instruction strings) are represented as instances of `InstructionString`, a `typedef` of `std::list<Instruction>`.

`InstructionSetProbabilistic` is a structure holding a list of all the possible instructions from which the LGP may select. It also holds the probability that the LGP system may select any of the possible instructions (the operation selection rate). Currently, the system is configured internally to select from `ADD`, `SUB`, `MUL`, `DIVP`, `LOADV`, `LOADS` and `STORS` with equal probability.

5.7 Overview of Experiment Process

The base function for performing evolution of low level programs by means of the LGP heuristic is `EvolutionSystem_EvolveInstructionStringFromParseTree()`. The 'standard', 'incremental', 'standard_with_refine' and 'incremental_with_refine' models all use this function as the basis for experiments.

An evolution attempt consists of the following phases.

- Prepare the configuration of the low level virtual machine.
- Prepare the fitness cases (training set and test set).
- Initialise bookkeeping variables.
- For the number of repeat runs defined in the parameter file:
  - Create a random population of candidate programs.
  - Calculate the fitness and hits of all candidate programs.
  - Until the number of allowed program recreations is exceeded:
    - If in reporting mode, display in-progress report.
    - If a candidate program has maximum hits, return it, end experiment.
    - Select recombination operation based on operation rates.
    - Use tournament selection to select unfit programs from population.
    - Create new programs through recombination operations.
    - Calculate the fitness of the new programs.
    - Replace unfit programs in population with new programs.
  - If the number of allowed program recreations is exceeded, proceed to next repeat run.
- If the number of allowed repeat runs has been exceeded, the LGP returns a value indicating that it was unable to evolve a suitable candidate program.

The success criterion for a typical evolution run is the generation of candidate program with the maximum amount of hits (an amount of hits equal to the amount of fitness cases in the training set).

The termination criterion for a typical evolution run is the exhaustion of all allowed candidate program recreations in all allowed runs.

5.8 Candidate Solution Program Initialisation

Within the software, a candidate solution in the population is stored in an instance of `GeneticLinearProgram`, associating each `InstructionString` with a `float` fitness value and an `unsigned int` hits value.

The size of the population is given by the user in the parameter file. The population model used in this system is a steady state model where the number of candidate programs in the population will remain constant throughout; as new programs are created, old programs are removed.

In the parameter file, the user must specify both a minimum and maximum size for the instruction strings in the original population. The initial population is generated containing programs forming a ramp of instruction lengths between these limits.

When a new Instruction is created, the operation component is created first, followed by the operand components. The operation component is randomly selected using the instruction inclusion rates given in the `InstructionSetProbabilistic`. The operands of the instruction are randomly selected from the range of possible values of that type: register index operands are selected from the range of valid indices into the register file, variable reference operands are selected at random from the list of variables described in the symbol table while observing the restrictions imposed by the symbol table: a variable that is read only cannot be the subject of a `STOR` instruction.

5.9 Fitness Case Construction

Fitness cases are stored as instances of the `FitnessCase` structure. This structure contains the starting and ending values of each variable in the symbol table before and after the program IR is evaluated by an interpreter (as a `std::map<SYMBOL_TABLE_KEY, VM_TYPE>`, together with a log of the operations performed during interpretation. Fitness cases are constructed by the function `FitnessCase_InitialiseCases()` at the start of an evolutionary experiment and persist for the duration of the experiment.

Fitness cases are constructed in a number of stages. First, the starting value of each of the variables is established. For input, output and intermediate variables, this is a random variable within some predefined bounds. For a numeric literal symbolic constant, this is simply the value it represents. Second, entries representing this initialisation of the initial variable state are entered into the evaluation log. These entries allow the fitness system to reward programs for recalling the values of variables from memory. Third, the input parse tree is evaluated in a high level

interpretation environment (through `ParseTreeNode_Evaluate()`). Finally, the values of the variables in the resulting state are copied into the fitness case as the target values for the candidate program.

`ParseTreeNode_Evaluate()` performs high level interpretation by requesting the value of the root node in the input variable memory state constructed previously. The function recursively evaluates all the nodes in the parse tree by requesting the value of or the reference of the child nodes, as required by the defined semantics of each parse tree language feature (these are referred to as 'evaluation' and 'ereferation' in the source code). As the interpreter performs evaluation or ereferation on a node, it records the values the node obtained from its children, and the value that the node itself produced as a result of its defined operation. For example, an addition node evaluation is performed by evaluating its left child node and right child node, adding the resulting values, and returning the result. A variable node is evaluated by returning the value of the variable in the current memory state. As a result, a complete, chronological list of all the operations performed will be generated. This is recorded as the 'model' execution of this parse tree.

To demonstrate this interpretation and logging method, we recall the example parse tree previously considered:

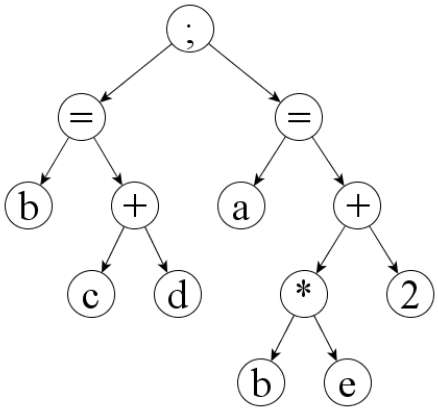


Figure 5.3: Parse tree visualisation for sample IR

The following debugging output listing shows the evaluation log for this parse tree.

```
1 INITIALISATION    :          0 <0> to      0 <0> giving  23996     <1>.
2 INITIALISATION    :          0 <0> to      0 <0> giving  17517     <1>.
3 INITIALISATION    :          0 <0> to      0 <0> giving  44798     <1>.
4 INITIALISATION    :          0 <0> to      0 <0> giving  41142     <1>.
5 INITIALISATION    :          0 <0> to      0 <0> giving  37331     <1>.
6 INITIALISATION    :          0 <0> to      0 <0> giving      2     <1>.
7 ADDITION          :      23996 <0> to  17517 <0> giving  41513     <1>.
8 MULTIPLICATION    :      41513 <2> to  44798 <0> giving 1859699374 <3>.
9 ADDITION          : 1859699374 <3> to      2 <0> giving 1859699376 <4>.
```

Listing 5.4: Debug output of high level evaluation log for sample parse tree

All values handled by the interpreter are combined with a second value indicating its heuristic 'complexity of calculation' value. This value represents the 'difficulty' of performing this calculation randomly and is used in the weighting of fitness rewards. The numbers in the angled brackets show the complexity heuristic value associated with each result value.

The first six lines show the random initialisation of variables a, b, c, d and e, and the constant initialisation of 2. Line 7 shows the value of variable c (recalled from an initialised variable so has complexity 0) being added to the value of variable d (also recalled from an initialised variable) resulting in the value 41513 with complexity 1. With each successive calculation or assignment of a value, the complexity increases by 1. Line 8 shows the value of 41513 with complexity 2 (having been calculated through addition, and then assigned to the intermediate variable b) multiplied with the value of the variable d (recalled from an initialised variable) to produce the value 18596993754 with complexity 3. The final line shows this value being added to the constant 2 to produce 1859699376 with complexity 4.

The log of the evaluations is stored in an instance of `ParseTreeEvaluationLog`, a `typedef` of `std::vector<ParseTreeEvaluationLogEntry>`. Each `ParseTreeEvaluationLogEntry` corresponds to a row of the log shown above, containing two input value-complexity pairs, an output value-complexity pair and a record of the operation performed.

5.10 Fitness and Hits Calculation

As described in the Solution Methodology section, the software calculates the fitness of a candidate solution instruction string by considering an instruction-by-instruction record of the actions taken when the candidate is interpreted in a virtual machine within the context of a given fitness case. The objective of the fitness calculation is to provide a graduated mapping from candidate solutions to fitness values that allows the LGP system to distinguish between candidate solutions by how 'close' they are to the semantics of the input IR.

Within the software, a candidate solution in the population is stored in an instance of `GeneticLinearProgram`, associating each `InstructionString` with a `float` fitness value and an `unsigned int` hits value. The function `GeneticLinearProgram_FitnessEvaluationOnTrainingSet()` is used to calculate the fitness and hits values of a candidate program on the training set of fitness cases.

The fitness value is calculated by setting the fitness of the candidate solution to a large high value, then considering each `FitnessCase` in the training set in turn to produce a modifier for each case, then summing these values to produce the final value. Low fitness values are considered to be productive, 'reward' modifiers have a

negative value. High fitness values are considered to be unproductive, 'penalty' modifiers have a positive value.

Evaluation of the fitness modifier for a given `FitnessCase` is performed in a number of stages. First, a `VirtualMachine` instance is generated and its memory initialised with the initial values from the `FitnessCase`. Then, the candidate program instruction string is executed in the `VirtualMachine` to produce an output memory state and a `VirtualMachineExecutionLog` detailing the exact operations performed by the virtual machine during execution. A log of execution is used instead of direct analysis of the candidate low level program because the low level program may contain jump instructions, causing it to be executed in a non-linear way, which may in turn be affected by some property of the values in the input state. (However, in these experiments, the instruction set does not contain jump instructions.)

The `VirtualMachineExecutionLog` instance is then translated into a `VirtualMachine ExecutionRegisterStatusTimeline` instance detailing the exact value of each virtual machine register before and every instruction execution, together with a flag indicating whether the value of the register is determinate at that point in time. At the start of low level execution, the values of all registers are indeterminate. A register is no longer flagged as indeterminate if its value is overwritten with that of a variable or a constant. If an indeterminate value is used as the input to a calculation, then the result is indeterminate. The timeline also stores a flag indicating when the last write to each register occurred; this is used to identify when a register has been written to and not subsequently read at any point. This is recorded as a 'dangling write'.

For each variable in the `SymbolTable` which has the `expected_match_old_state` or `expected_match_new_state` flag set, the value of the variable in the resulting memory state is compared with the desired value as defined by the fitness case. If these values do not match, a constant value penalty modifier is applied together with a variable modifier based on the error between the values.

A fitness penalty is applied if the low level virtual machine execution ends on an unsafe termination condition.

For each entry in the `ParseTreeEvaluationLog`, a number of 'tickets' are initialised, indicating the remaining number of times each reward can be given for creating that value.

The two log formats are then analysed to effect the following fitness modifiers:

If the contents of a register are indeterminate before the execution of an instruction and this register is read as a result of the instruction, this triggers `FITNESS_PENALTY_FOR_REGISTER_INDETERMINATE_READ`. This penalty can be triggered any number of times. The triggering of this penalty precludes the awarding of a 'hit'.

If the contents of a register are determinate before the execution of an instruction and this register is read as a result of the instruction, this triggers FITNESS_BONUS_FOR_REGISTER_DETERMINATE_READ. This reward can be triggered any number of times.

If the contents of a register are determinate before the execution of an instruction and this register is written to as a result of the instruction and the register has not been read from in the interval from when it was last written to this point, (i.e. a 'dangling write' has been overwritten), this triggers FITNESS_PENALTY_FOR_REGISTER_REWRITING. This penalty can be triggered any number of times.

If the contents of a register are altered as the result of the execution of an instruction and the resulting value appears as the result of a calculation performed during the high level interpretation of the IR and there is a remaining 'ticket' associated with that row of the ParseTreeEvaluationLog, then FITNESS_BONUS_FOR_PRODUCING_USEFUL_VALUE is triggered and the amount of remaining 'tickets' is decremented. (The software has the ability to identify if the high level parse tree node and the low level language instruction correlate (i.e. when an addition node creates a specific value in the parse tree, and the low level language candidate instruction string mirrors this exact operation with its ADD instruction) and apply the FITNESS_BONUS_FOR_PRODUCING_USEFUL_VALUE_THEN_CORRECT_OPERATION reward, though this was not used in the experiments in this dissertation as it grants the LGP system information about the two language forms that it may not have.)

If a LOAD instruction is used to recall a value that appears as the result of a calculation performed during the high level interpretation of the IR and there is a remaining 'ticket' associated with that row of the ParseTreeEvaluationLog, then both FITNESS_BONUS_FOR_PRODUCING_USEFUL_VALUE and FITNESS_BONUS_FOR_READING_VALUE_FROM_MEMORY are triggered and the amount of remaining 'tickets' is decremented.

If a STOR instruction is used to store the value of a register to a variable as the result of an instruction and the value of the register is indeterminate at that point in time, then FITNESS_PENALTY_FOR_STORING_INDETERMINATE_VALUE is triggered. This penalty can be triggered any number of times. The triggering of this penalty precludes the awarding of a 'hit'.

If a STOR instruction is used to store the value of a register to a variable as the result of an instruction and the variable is defined as expected_match_new_state by the symbol table (that is, its value is expected to change to a new value as a result of program execution and this value is of importance), then FITNESS_BONUS_FOR_WRITING_TO_VARYING_VARIABLE_EXPECTED_CHANGE is triggered. This reward can be triggered any number of times.

If a STOR instruction is used to store the value of a register to a variable as the result of an instruction and the variable is not defined as expected_match_new_state by the symbol table (that is, its value may change as a result of the program, but this is not required), then FITNESS_BONUS_FOR_WRITING_TO_VARYING_VARIABLE is triggered. This reward can be triggered any number of times.

If a register has 'dangling write' status at the time of program termination, the penalty FITNESS_PENALTY_FOR_REGISTER_DANGLING_WRITE is triggered. This reward can be triggered once for each register.

There exists a final fitness bonus which is not applied in these experiments: if the contents of a register are altered as the result of the execution of a LOADV (load constant) instruction and the resulting value appears as the result of a constant load during the high level interpretation of the IR and there is a remaining 'ticket' associated with that row of the ParseTreeEvaluationLog, then FITNESS_BONUS_FOR_PRODUCING_USEFUL_VALUE and FITNESS_BONUS_FOR_PRODUCING_USEFUL_CONSTANT are triggered and the amount of remaining 'tickets' is decremented. This was not used in the experiments in this dissertation as it grants the LGP system information about the two language forms that it may not have.

If a 'hit' is not awarded for a given fitness case, a large penalty FITNESS_PENALTY_FOR_NON_HIT is triggered. If a 'hit' is triggered, then a penalty proportional to the length of the candidate program is triggered, but this penalty is always less than the penalty for not scoring a hit. These penalties are designed to reward programs which score hits, and reward programs of shorter length where a hit is scored.

The hits value is the number of fitness cases for which the candidate solution produces the appropriate value for every variable in the SymbolTable which has the expected_match_old_state or expected_match_new_state flag set, unless the hit has been negated by the triggering of a penalty. The ability of a penalty to preclude the awarding of a hit is used to prevent the evolutionary system from returning candidate solution programs which have the correct effects, but rely on indeterminate values or related effects during their execution. As such programs will still have good fitness scores due to the triggering of other rewards, it is intended that the LGP system be able to identify programs that have the correct semantics only under specific circumstances and use the available recombination operations to modify the program to remove the instructions which manipulate the indeterminate values.

5.11 Implementation of Low Level Virtual Machine

The software uses a low level virtual machine to execute candidate programs to determine the results of execution given an input state, and to create an instruction-by-instruction log of execution.

Execution of candidate instruction strings is performed by calling `VirtualMachine InstructionString_ExecuteInstructionString()` with a target `InstructionString`. This function has the capability to pause before each instruction execution and dump the current contents of the variable memory and register file to the screen. This is used if the fully interactive mode is enabled in the parameters file.

The virtual machine has a finite register file of `VM_REGISTER_GENERAL_PURPOSE_COUNT` values of `VM_TYPE` type, a memory associating each variable in the symbol table (through `SYMBOL_TABLE_KEY`) with a VM_TYPE value. The virtual machine also has an error code variable used for reporting if and why the virtual machine was abnormally terminated. The virtual machine also has a program counter which is not accessible to low level programs directly.

Instructions in the low level language are implemented as a series of functions accessible as an array of function pointers, indexed by a `VM_INSTRUCTION_OPERATION` value (the instruction operation enumeration type stored within `Instruction`).

Every instruction step, the virtual machine considers the instruction at the position of the program counter and executes the associated implementation function for that operation. Each instruction records the values considered and produced during execution in an instance of `VirtualMachineInstructionExecutionRecord` and appends this to a shared `VirtualMachineExecutionLog`. The values of all registers before and after the instruction execution are also recorded.

The execution loop terminates on the following conditions:

- The end of the instruction string is encountered. This is considered a safe termination.
- Division by zero. This is considered an unsafe termination. (This is not possible with the low level instructions specified in this dissertation as protected division is used.)
- The maximum limit of instruction executions is reached. This is considered an unsafe termination.
- A terminate instruction is encountered. (There are no terminate instructions currently used in these experiments.)

If one of these conditions is met, the execution loop returns. The error state value is available to the calling function for analysis.

5.12 Implementation of Recombination Operations

There are three recombination operations available to the LGP system: crossover, mutation and reproduction. One of these operations is selected as random during the recombination stage of the LGP system according to the rates supplied by the user in the parameter file. The number of child programs produced depends on the type of recombination selected: mutation and reproduction create a single child program, crossover creates two child programs.

Instructions are copied individually within these operations instead of via a memory block copy as the instructions may contain complex elements, such as instances of classes, which cannot be copied in this manner (currently, this depends on the chosen base type of `BigInteger`).

The crossover operation creates two new `GeneticLinearProgram` instances simultaneously by performing the crossover operation on the two selected parent `GeneticLinearProgram` instances. In this system, both products of crossover are retained.

Transition points are randomly placed within the two instruction strings. Then, new `InstructionStrings` are constructed by copying instructions from the first string until the first transition point, then copying instructions from the second string starting at the first transition point until the second transition point, then copying the remaining instructions from the first string starting from the second transition point until the end.

As the parent programs and transition points are chosen at random, the resulting programs can be of any length. To prevent the explosive growth of instructions in the population, there is a limit to the length of programs which may be produced as a result of crossover. This limit is specified in the parameter file. If, after selection, crossover would result in a program that violates this limit, the crossover is aborted and the transition points are re-chosen until a valid combination is selected.

The mutation operation first produces an exact copy of the selected parent `GeneticLinearProgram`, then selects a random instruction from the `InstructionString` of this new program. Then one component of the instruction is chosen at random from the available components (as described in the instruction set listing previously). If an operand component is chosen, it is replaced with an operand of compatible type, chosen at random, while observing the restrictions imposed by the symbol table: a variable that is read only cannot be the subject of a `STOR` instruction. If the operation component is chosen, then the all components of the instruction are reinitialised as if the `Instruction` were recreated from scratch.

The reproduction operation creates a new instance of `GeneticLinearProgram` and inserts a copy of the `InstructionString` component of the selected parent `GeneticLinearProgram`.

After the child programs have been created through the application of one of these operations, an equal number of existing programs are selected by tournament selection for removal based on their unfitness.

5.13 Tournament Selection

Tournament selection is used to select programs from the candidate solution population for application to the available genetic recombination operations and the elimination operation.

In tournament selection, a set number of programs are randomly selected from the population to participate in the tournament. From these programs, a number of programs, equal to the number of required parent programs for the operation, with the highest fitness values are passed to the genetic recombination operation.

The population model used in this system is a steady state model where the number of candidate programs in the population will remain constant throughout. For each new program produced by a genetic operation, a tournament is used to choose a random program with low fitness (i.e. a tournament is held, selecting the lowest fitness from the participants) to be removed from the population.

The sizes of the tournaments used in selection for each of the recombination operation and elimination are specified separately in the parameter file.

In this implementation, the population is stored in memory as an array of pointers to `GeneticLinearProgram` instances. During the elimination selection stage, pointers to the positions in this array are obtained. The indices into this array of the programs participating in the tournament can be found by randomly selecting the required number of random numbers from `0` to `population_size - 1`. The `IndexSeries` functions are designed for this purpose. When the program occupying a slot in this array of pointers is destroyed, the new programs created through the recombination operation take their place.

5.14 Experiment Models

There are number of different experiment evolutionary models available to the user in this implementation: ‘standard’, ‘standard with refine’, ‘incremental’ and ‘incremental with refine’. Depending on the required measurement, the use of niche pre-emption will differ also. Each of these methods wraps the base implementation of LGP as described previously in a different way.

For experiments measuring computational effort, we are measuring how many low level language instructions must be evaluated in a typical run before a solution is found. To do this, we examine the capability of the system to evolve an acceptable candidate solution within a single run, and how many instruction evaluations it takes to achieve this. We use a single run of the experiment with a limited number of candidate program recreations. If the LGP system successfully produces an acceptable solution within the limited number of recreations, the number of instruction evaluations that was necessary is recorded. If the LGP exhausted the number of candidate program recreations without finding a solution, a 'high' value is recorded, signifying that the number of evaluations necessary is beyond the range we consider in this experiment.

For experiments measuring program length, we are measuring the length of an acceptable solution program returned by the system, given that the system has been given sufficient resources to do so. To do this, we allow the system to use a large number of independent runs and a large number of candidate program recreations. The use of a large number of independent runs allows the system to reattempt the problem with a number of different initial populations until a solution is found. When the LGP successfully produces an acceptable solution, the length of this solution in low level language instructions is recorded.

For experiments using the 'standard' evolution model, the input IR is exposed to the base `EvolutionSystem_EvolveInstructionStringFromParseTree()` function directly, and the appropriate results recorded. The implementation of experiments using the 'incremental' model is discussed in the following section.

After an acceptable solution has been found during the primary evolution phase, this solution may be re-inserted into the LGP system to attempt to improve the quality of the solution. This is the 'refinement' stage. For experiments which include the refinement stage, after each complete acceptable low level language translation of the input IR is evolved, this solution is then reintroduced to the LGP system for a second time using the same input IR. For this second stage of evolution, a set proportion of the initial population is initialised as copies of the previously obtained solution. The value of this proportion is defined in the parameters file. In this second phase, the parameters from the second parameters file are used. The remainder of the population is initialised as a ramp as before. The experiment is terminated upon reaching a maximum number of candidate solution recreations. The length of the best-of-run candidate solution in instructions is returned as the result. There is no attempt to determine if the optimal solution has been reached.

The measurement of computational effort in combination with the refinement stage is not considered in this dissertation.

5.15 Implementation of Incremental Model

In the incremental model, the parse tree component of the input IR is mechanically segmented into a number of smaller sub-programs.

Each interior node of the parse tree is assigned a string name indicating the position of the node in the tree. The following figure shows the naming used for the example program discussed previously.

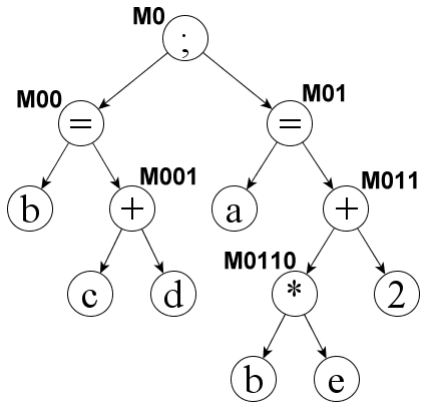


Figure 5.5: Naming of interior nodes for sample program under incremental model

Intermediate variables with these names are added to the symbol table as intermediate variables, and associated entries in the `ParseTreeLeafNodeSetMember` catalogue are generated.

A depth-first traversal of the tree is performed and each interior is removed and replaced with a variable node of the same name. The detached subtree is then used to create a new program with an assignment node as its root and an instance of the interior node variable as the subject of the assignment. This results in a number of subprograms equal to the number of interior nodes each consisting of five nodes: the assignment, the interior node intermediate variable and the three nodes of the original subtree.

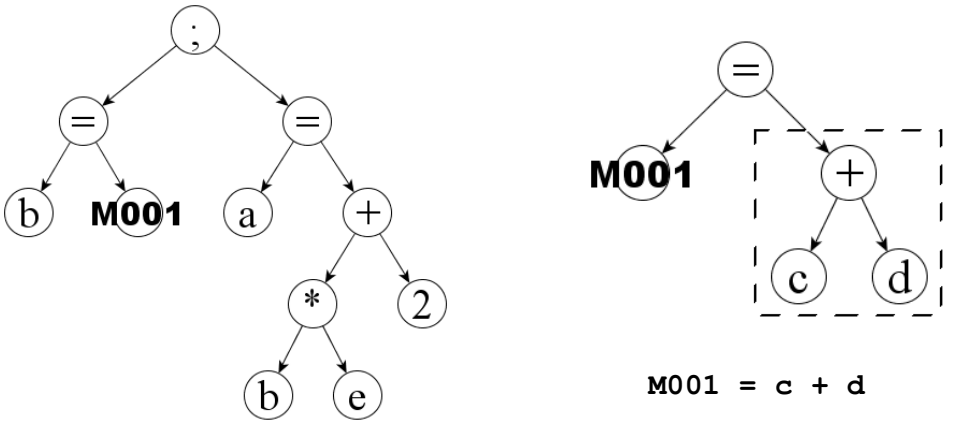


Figure 5.6: Modified parse tree and detached program segment for sample program under incremental model

The full decomposition of the input IR using this method results in the following subprograms:

```
1: M001  = (c + d)
2: M00   = (b = M001)
3: M0110 = (b + e)
4: M011  = (M0110 + 2)
5: M01   = (a = M011)
6: M0    = (M00; M01)
```

The decompositions on lines 2 and 5 are possible because the assignment operator returns the value of its right hand side. Line 6 has no effect on the system whatsoever.

Solutions for each of these subprograms are evolved as if by the standard model, using the parameters from the first parameters file, to produce a series of subsolutions. These subsolutions are then concatenated to produce the complete solution.

The value of computational effort for an assembled solution program produced under the incremental model is the sum of the computational effort values for each of its component parts, as calculated by logarithms.

The value of program length for an assembled solution program produced under the incremental model is the sum of the program lengths of the component parts.

6 Analysis of Results

This section examines the results obtained from performing the experiments described in this dissertation and explores some of the traits apparent in the data. An aggregation of the results obtained from the experiment is given as an appendix. The full raw data logs are included on the accompanying CD-ROM.

6.1 Overall

The LGP system was capable of evolving appropriate low level language solution programs for all ten of the specified programs at least once. The system was able to automatically determine which instructions from the provided instruction set were of value, as shown by the evolution of optimal programs; in these programs, all unproductive instructions were removed by the recombination operations.

6.2 Computational Effort

Across all programs, for the 'incremental' method, the required computational effort appears to scale linearly with the number of parse tree nodes in the input program. Increasing the 'depth' of the calculation, considering programs B01 to B04, does not seem to have a pronounced additional effect on the computational effort. This may be because all of the program fragments are of the same shape, and of similar difficulty; commutative operators such as addition or multiplication would be easier to evolve than non-commutative operations such as subtraction, division or assignment due to there being fewer possible programs within the program space that have the desired effect.

For the 'standard' method, required computational effort appears to scale exponentially with increasing calculation depth and increasing numbers of statements (semicolons).

Program B04 has exceptionally high values for computational effort for both the 'standard' and 'incremental' methods. Program B04 contains the greatest depth of calculation. The primary stumbling block appears to be the evolution of the division operation. The following graph shows the performance curves for the evolution of the various subprograms created when Program B04 is treated by the 'incremental' method.

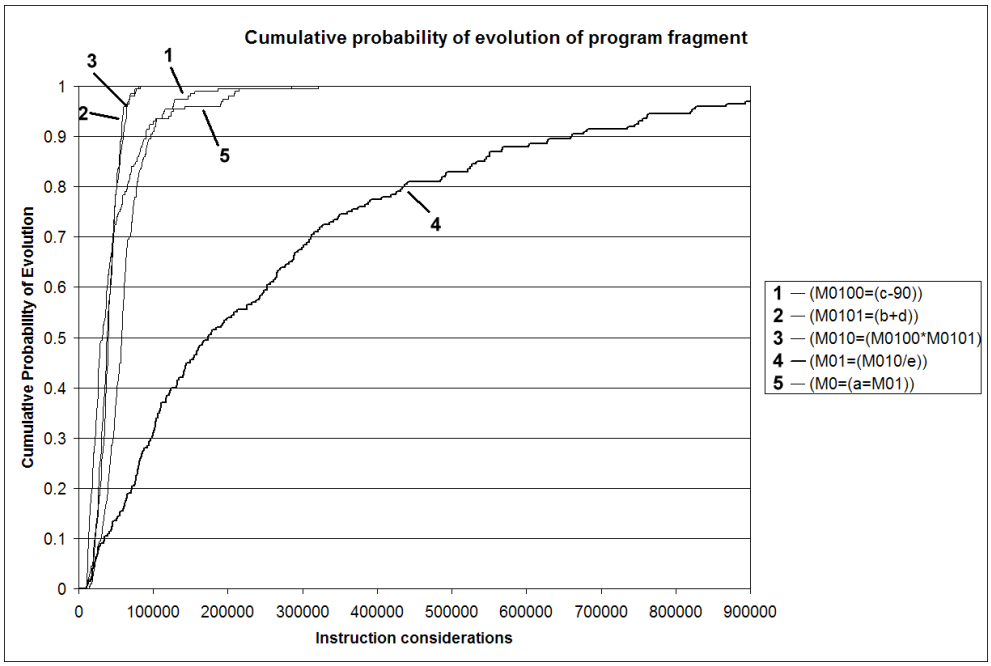


Figure 6.1: Cumulative probability of evolution of each subprogram fragment for program B04

Fragments 2 and 3 appear to be the easiest to evolve a solution for: they consist of a single commutative arithmetic operation followed by an assignment. Fragments 1 and 5 appear to be the second easiest to solve; they consist of a single non-commutative operation followed by an assignment. Fragment 4 is the hardest fragment to solve, and contributes 1110000 of the 1595500 instructions of the E_I value. This may be because the inclusion of the integer protected division operation in a program will often result in very small output values being produced due to the input values all lying within a small input range. Such programs may be difficult to improve using the genetic operations, as many of the possible operations will not have a noticeable productive effect.

6.3 Program Length

Where both the 'standard' and 'incremental' methods are able to evolve a solution program within a reasonable amount of time, it appears that the solution produced by the 'standard' method will be of shorter length by approximately 30% - 60%.

When refinement is applied, the evolutionary system is able to reduce the mean solution program length by 60% - 80% for programs produced by the 'standard' method, and approximately 20% for programs produced by the 'incremental' method.

For all programs except B04 and D03, 'standard' with the refinement operation was able to produce at least one solution that is 'perfect' given the available instructions.

For program D03, ‘standard’ with the refinement operation was able to produce at least one solution that was better than the unoptimised tree walking algorithm.

For programs A01, A02, B01 and D01, ‘incremental’ with the refinement operation was able to produce at least one solution that is ‘perfect’ given the available instructions. For all other programs, the minimum solution program length was greater than that of the unoptimised tree walking algorithm. These long programs may be the result of the evolutionary system being unable to remove intermediate variables introduced during the fragmentation process.

From these results, it appears that the ‘standard’ method is capable of producing optimal programs in many circumstances, but only if significant amounts of processor time are dedicated to the problem. If ‘any solution’ is acceptable, then the ‘incremental’ method is capable of producing such a program quickly and rapidly. For these programs, the tree walking compiler performs considerably faster and produces programs of near optimal quality. The only advantage that the ‘incremental’ method has over the tree walking algorithm is that the tree walking algorithm does not take into account the finite register file in the virtual machine; it may simply exhaust the register file and crash.

7 Evaluation of Project

This section provides a reflection on the conduct of the project, and the applicability of the results and conclusions drawn from the data. This section concludes with a discussion of the limitations of the project due to its design and offers some insight into how the work may be extended or adapted in future.

7.1 Evaluation of Experiment Conduct

I have successfully implemented the program as described in the Design section of this document. The evolutionary system prototype software is capable of evolving low level instruction string programs that are semantically equivalent to short sequences of statements in the high level source code language. In some cases, the system is capable of evolving instruction strings of optimal quality given the instruction set. However, this process is time consuming and processor intensive due to the evaluation of many thousands of candidate programs against hundreds of fitness cases. No guarantee may be made that the process will succeed at all, due to the probabilistic nature of the Linear Genetic Programming system.

The incremental approach described in the previous report has been implemented and shown to be superior to the standard approach in terms of processor requirements, but inferior when the lengths of the output programs are considered.

Due to limited available processor time, the E_I values for programs B03, B04 and D03 using the 'standard' method may be artificially high due to the extreme unlikelihood of finding solution programs.

It was necessary to alter the method for calculating the distribution of solution program lengths for Programs B04, C01, D01, D02 and D03 due to the extreme unlikelihood of finding a solution program. It was decided to perform 20 refinements on each solution program rather than 5 as normal. This was chosen as a reasonable compromise to dedicating significant time to finding multiple unrefined programs, as it is the *distribution* of program lengths after refinement that is under examination, so additional refinements of the same raw program will suffice, given that the raw program was produced in the correct manner.

8 Professional Issues

This section examines the conduct of this dissertation in the context of the British Computer Society Code of Conduct and Code of Good Practice.

8.1 British Computer Society Code of Conduct

The British Computer Society Code of Conduct 'sets out the professional standards required by BCS as a condition of membership' (BCS, 2006). This section discusses how the Code of Conduct has been observed during the preparation of this dissertation.

The Public Interest

'In your professional role you shall have regard for the public health, safety and environment.'

Throughout the preparation of this dissertation, I have conducted myself in a manner observing this requirement. I have used all necessary electronic equipment in the proper manner and according to their safety guidelines, and have taken the appropriate measures to safeguard my own health while using the equipment.

'You shall have regard to the legitimate rights of third parties.'

I have conducted myself in the proper manner with regards to the guideline document *Plagiarism, Collusion and the Fabrication of Data: Guidelines for Staff and Students* (UoL, a) produced by the University. As directed by Part B of these guidelines, I have done the following:

- Where I have referenced the work or concepts of others, I have correctly identified all sources and included all relevant bibliographic information in the Bibliography section of this dissertation.
- I have not represented another's work or concepts as my own.
- I have not allowed any student access to my work at any time.
- I have not attempted to assist any other student in any way.

'You shall ensure that within your professional field/s you have knowledge and understand of relevant legislation, regulations and standards, and that you comply with such requirements'

I have familiarised myself with all legislation relevant to my work. Where I have requested documents, software or data from outside the University, I have observed the relevant regulations with respect to the import or export of technology.

'You shall conduct your professional activities without discrimination against clients or colleagues.'

Throughout the preparation of this dissertation, I have conducted myself in a professional manner when dealing with University staff.

'You shall reject and shall not make any offer of bribery or inducement.'

I have not made any offer of bribery or inducement.

Duty to Relevant Authority

'You shall carry out work or study with due care and diligence in accordance with the relevant authority's requirements, and the interests of system users. If your professional judgment is overruled, you shall indicate the likely risks and consequences.'

Throughout the preparation of this dissertation, I have conducted myself in a manner observing the requirements set forth by the University's codes of conduct.

'You shall avoid any situation that may give rise to a conflict of interest between you and your relevant authority. You shall make full and immediate disclosure to them if any conflict is likely to occur or be seen by a third party as likely to occur. You shall endeavour to complete work undertaken on time to budget and shall advise the relevant authority and soon as practicable if any overrun is foreseen.'

At the time of preparation of this dissertation, my only active professional affiliation is that to the University of Liverpool. I have not allowed any situation to develop which may be construed as forming a conflict of interest.

At the outset of the preparation of this dissertation, a schedule of work was agreed with my primary supervisor. I have been in contact with my supervisors at the University on a regular basis, and during these meetings have reported the progress of my work. As required by the COMP702 guidelines (UoL, b), I have submitted regular written progress reports to the University detailing the work undertaken to date. This dissertation will be submitted before the required deadline given by the University.

'You shall not disclose or authorise to be disclosed, or use for personal gain or to benefit a third party, confidential information except with the permission of your relevant authority, or at the direction of a court of law.'

No information of a personal or confidential nature was handled at any point during the preparation of this dissertation.

'You shall not misrepresent or withhold information on the performance of products, systems or services, or take advantage of the lack of relevant knowledge or inexperience of others.'

I have not misrepresented or withheld any information on the performance of the system described in this dissertation. The complete software used to generate the results detailed in this dissertation is included in full on the accompanying CD-ROM in both compiled executable binary and human-readable source code forms. This software may be used to repeat the experiments described in this dissertation.

'You shall uphold the reputation and good standing of the BCS in particular, and the profession in general, and shall seek to improve professional standards through participation in their development, use and enforcement.'

Throughout the preparation of this dissertation, I have conducted myself in a professional manner, and in full compliance of all relevant guidelines, standards and codes of conduct.

'You shall act with integrity in your relationships with all members of the BCS and with members of other professions which whom you work in a professional capacity.'

Throughout the preparation of this dissertation, I have conducted myself in a professional manner when dealing with University staff.

'You shall have due regard for the possible consequences of your statement on others. You shall not make any public statement in your professional capacity unless you are properly qualified and, where appropriate, authorised to do so. You shall not purport to represent the BCS unless authorised to do so.'

I have not made any public statement during the preparation of this dissertation. Insofar as the dissemination of this dissertation is considered as a statement in my capacity as a postgraduate student, I have conducted my studies in a professional manner and with integrity.

'You shall notify the Society if convicted of a criminal offence or upon becoming bankrupt or disqualified as Company Director.'

This section is not applicable in this instance.

Professional Competence and Integrity

'You shall seek to upgrade your professional knowledge and skill, and shall maintain awareness of technological developments, procedures and standard which are relevant to your field, and encourage your subordinates to do likewise.'

The preparation and execution of this dissertation is the result of my intent to upgrade my professional knowledge and skill in the field of Computer Science. In preparing this dissertation, I have performed a significant amount of research into the technological developments surrounding the subjects studied in this dissertation.

'You shall not claim any level of competence that you do not possess. You shall only offer to do work or provide a service that is within my professional competence.'

The undertaking of research, development of computer software, and analysis of results necessary for the preparation of this dissertation are within my level of competence.

'In addition to this Code of Conduct, you shall observe whatever clauses you regard as relevant from the BCS Code of Good Practice and any other relevant standards, and you shall encourage your colleagues to do likewise.'

Throughout the preparation of this dissertation, I have conducted myself in a professional manner, and in full compliance of all relevant guidelines, standards and codes of conduct. A discussion regarding how the preparation of this dissertation has observed the BCS Code of Good Practice appears in the following section.

'You shall accept professional responsibility for your work and for the work of colleagues who are defined in a given context as working under your supervision.'

I accept professional responsibility for my work, and have made relevant declarations to the University regarding plagiarism and academic conduct to this effect. No colleagues have worked under supervision at any point during the preparation of this dissertation.

8.2 British Computer Society Code of Good Practice

The British Computer Society Code of Good Practice 'describes standards of practice relating to the contemporary multifaceted demands found in IT' (BCS, 2004). This section discusses how the Code of Good Practice has been observed during the preparation of this dissertation. Only those subsections deemed relevant to the preparation of this dissertation and my role as a student are included in this section. There is some overlap between this section and the previous section; sections which exactly duplicate language from the Code of Conduct are not considered.

Practices Common to all Disciplines

Maintain Your Technical Competence

Throughout the preparation of this dissertation, I have sought to improve my IT skills in the areas of software development and document preparation through the use of online resources and other resources provided by the University of Liverpool. In

performing the research stages of this dissertation, I have myself aware of all relevant technological advances in the fields of software development and the development and use of evolutionary computation metaheuristic algorithms.

I have sought to obtain an appropriate qualification as a result of the submission of this dissertation; this dissertation has been submitted in partial fulfilment of the requirements of the degree of Master of Science at the University of Liverpool for the degree entitled Advanced Computer Science.

Adhere to Regulations

Throughout the preparation of this dissertation, I have conducted myself in a professional manner, and in full compliance of all relevant guidelines, standards and codes of conduct. I have performed and presented the research described in this dissertation according to the commonly observed conventions of academic research, such as the full description of all experiments performed and the collation of a bibliography detailing all references consulted.

Use Appropriate Methods and Tools

The software has been written using the C++ language, a language highly appropriate for the development of applications manipulating the processing of large amounts of data at a rapid rate, as required for the function of a genetic evolutionary computation system. The purpose of this dissertation is to explore the applicability of the Linear Genetic Programming evolutionary computation model for the task of performing code generation in a general-purpose software compiler; little prior information exists on this subject, to my knowledge.

Practices Specific to Education and Research Functions

When Performing Research

This dissertation has been prepared in accordance with academic convention, as mentioned previously. The purpose and consequences of this research have been fully explored; the research performed as part of this dissertation may lead toward the development of enhanced software compilers in the future. This, in turn, would allow for expedited development of new software at a lower cost.

It is my intention that this dissertation be made freely available to all interested parties after the conclusion of all University assessment procedures.

Practices Specific to Business Functions

When Conducting Systems and Business Analysis

This dissertation has been targeted toward the reader with some knowledge of the terms and techniques used within the field of Computer Science and has been worded as such. The dissertation contains a full description of the techniques used; the source

code is included on the associated CD-ROM. The technical constraints of the software produced have been fully explored and detailed.

When Designing Software / When Programming

The software produced as part of this dissertation is of high quality, extensible, well-documented and modular in nature. Coding conventions, such as the consistent naming of variables and functions, and the use of indentation are used throughout the source code.

The software does not contain any platform specific code limiting its use to any one operating system. The programming language used throughout the project, C++, is available for use on almost all platforms.

When Writing Technical / User Documentation

The design and use of the software is described extensively in this dissertation. Comments are used throughout the source code. A full function and data type reference is included as an appendix.

9 Conclusion

9.1 Conclusions

In this dissertation, we have formulated, specified, implemented and evaluated several new models for using Linear Genetic Programming (LGP) to transform an input program written in a high level language into the most optimal machine code form given the available functionality of the target architecture. The LGP system has been shown to implicitly perform the tasks of instruction selection and register allocation as would be performed by a non-evolution code generator.

It is believed that the system described in this dissertation is capable of automatically assimilating additional instructions provided to it and automatically applying them in its determination of low level output programs without being guided as to their best use by a human programmer. The system is guided by a graduated fitness function which has been shown to be sufficiently general to be directly reused between programs without configuration by a human programmer.

The refinement stage has shown the most promise of the two stages examined in this dissertation. The experiments described previously have shown that LGP has the capability of reducing complex programs written in a low level assembly-like language to simpler programs automatically without the loss of semantic information. In some cases, the system has shown the capability of producing what is believed to be the most optimal form of a given program.

9.2 Summary of Contributions

This dissertation has demonstrated the first attempted use of LGP, to my knowledge, to perform the task of performing the translation from an input IR to low level assembly-like code through the formulation of a single fitness function applicable to all cases.

9.3 Limitations of Project and Further Work

The experiments undertaken as part of this project have all used the same set of evolutionary system parameters. A more complete analysis of the problem would consider the effects of altering the various parameters to the evolutionary system, such as the maximum number of new creations, the maximum length of a candidate program produced through crossover, the maximum number of new creations available to the refinement stage and the fraction of raw solution copies in the genetic population during the refinement stage.

The input programs considered by this project have been simple lists of statements with no control structures. Additional programs may be investigated containing IF statements, WHILE and FOR loops, and other constructs manipulating program flow. In order to investigate these, the low level instruction set must be extended by providing the means for non linear control flow. This may include call-and-return

instruction pairs for the automatic development of subroutines, or compare-and-jump instructions for conditional control flow similar to those found in a RISC architecture such as ARM. To test the ability of the system to automatically assimilate new instructions, it is suggested to repeat the experiments with additional instructions, such as combined arithmetic instructions, and observe how the system responds.

The memory model used in this project is a simple symbolic associative memory. It is possible to extend the work attempted in this project to low level machines with indexed memory. Alternatively, the virtual machine definition can be extended by introducing the concept of a stack. The methods by which the evolutionary system attempts to use the stack can then be investigated.

Alternatively, methods other than evolutionary computation could be used to provide a more controllable approach to attempting code generation, while still retaining desirable qualities such the ability to automatically and autonomously determine the most appropriate use of each instruction from the available set.

Bibliography

(Angeline, 1996) Angeline, P. J. (1996) *An investigation into the sensitivity of genetic programming to the frequency of leaf selection during subtree crossover*. In J. R. Koza, et al., editors, Genetic Programming 1996: Proceedings of the First Annual Conference, pages 21-29, Stanford University, CA, USA, MIT Press.

(Banzhaf et. al., 1998) Banzhaf, W., Nordin, P., Keller, R. and Francone, F. (1998) *Genetic Programming – An Introduction. On the automatic Evolution of Computer Programs and its Application*. Dpunkt/Morgan Kaufmann, Heidelberg/San Francisco.

(BCS, 2004) BCS Trustee Board (2004) *The British Computer Society Code of Good Practice*. British Computing Society. **http://www.bcs.org**

(BCS, 2006) BCS Trustee Board (2006) *The British Computer Society Code of Conduct for BCS Members*. British Computing Society. **http://www.bcs.org**

(Beaty et. al., 1996) Beaty, S. J., Colcord, S. and Sweany, P. H. (1996) *Using Genetic Algorithms to Fine-Tune Instruction-Scheduling Heuristics*. In Proceeding of the Second International Conference on Massively Parallel Computing Systems, pages 6-9, IEEE Computer Society.

(Cooper et. al, 1999) Cooper, K. D., Schielke, P. J. and Subramanian, D. (1999) *Optimizing for Reduced Code Space using Genetic Algorithms*. Rice University, Houston, TX, USA.

(Cramer, 1985) Cramer, N. L. (1985) *A Representation for the Adaptive Generation of Simple Sequential Programs*. In Proceedings of An International Conference on Genetic Algorithms and their Applications, pages 183 – 187, Carnegie-Mellon University, Pittsburge, PA, USA.

(Crepeau, 1995) Crepeau, R. L. (1995) *Genetic evolution of machine language software*. In J. P. Rosca, editor, Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications, pages 121 – 134, Tahoe City, California, USA.

(Friedberg, 1958) Friedberg, R. M. (1958) *A Learning Machine: Part I*. IBM Journal of Research and Development, volume 2, pages 2 - 12.

(Friedberg et. al., 1959) Friedberg, R. M., Dunham, B. and North, J. H. (1959) *A Learning Machine: Part II*. IBM Journal of Research and Development, volume 3, page 282.

(Giarratino et. al., 1998) Giarratino, J. C. and Riley, G. D. (1998) *Expert Systems: Principles and Programming*. PWS.

(Harding et. al., 2007) Harding, S. and Banzhaf, W. (2007) *Fast Genetic Programming on GPUs*. Computer Science Department, Memorial University, Newfoundland.

(Holland, 1992) Holland, J. H. (1992) *Adaptation in Natural and Artificial Systems*. MIT Press, Cambridge, MA, USA.

(Huelsbergen, 1996) Huelsbergen, L. (1996) *Toward Simulated Evolution of Machine-Language Iteration.* In Conference on Genetic Programming 1996, pages 315-320. Bell Labs.

(Jackson, 2005) Jackson, D. (2005) *Evolution of Processor Microcode*. In IEEE Transactions on Evolutionary Computation, Volume 9, Issue 1, pages 44 – 54. University of Liverpool.

(Kinnear, 1993) Kinnear, K. E. (1993) *Generality and Difficulty in Genetic Programming: Evolving a Sort*. In Proceedings of the Fifth International Conference on Genetic Algorithms.

(Kinnear, 1994) Kinnear, K. E. (1994) *Fitness landscapes and difficulty in genetic programming*. In Proceedings of the 1994 IEEE World Conference on Computational Intelligence, volume 1, pages 142-147, Orlando, FL, USA. IEEE Press.

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

(Koza, 1994) Koza, J. R. (1994). *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA.

(Koza et al., 1999) Koza, J. R., Andre, B., Bennett III, F. H., and Keane M. (1999). *Genetic Programming 3: Darwinian Invention and Problem Solving*. Morgan Kaufman.

(Koza et al., 2003) Koza, J. R., Keane M. A., Streeter, M. J. Mydlowec, W., Yu, J., and Lanza, G (2003). *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers.

(Kühling, et. al. 2002) Kühling, F., Wolff, K. and Nordin, P (2002) *A Brute-Force Approach to Automatic Induction of Machine Code on CISC Architectures.* Chalmers Technical University, Physical Resource Theory, Sweden.

(Langdon, 1998) Langdon, W. B. (1998) *Genetic Programming and Data Structures*, Kluwer Academic Publishers.

(Lomont, 2003) Lomont, C. (2003) *Fast Inverse Square Root*. Purdue University, IN, USA.

(Lorenz et. al. 2004) Lorenz, M. and Marwedel, P. (2004) *Phase Coupled Code Generation for DSPs Using a Genetic Algorithm*. In Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, pages 1530-1591 / 04

(Louden, 1997) Louden, K. C. (1997) *Compiler Construction: Principles and Practice*. PWS Publishing Company.

(Montana, 1995) Montana, D. J. (1995) *Strongly Typed Genetic Programming*. In Evolutionary Computation Journal, Volume 3, pages 199 – 230.

(Nordin, 1994) Nordin, P. (1994). *A compiling genetic programming system that directly manipulates the machine code*. In Kinnear, Jr., K. E. editor, Advances in Genetic Programming, chapter 13, pages 311-331. MIT Press.

(Nordin, 1997) Nordin, P. (1997) *Evolutionary Program Induction of Binary Machine Code and its Applications*. Ph.D thesis, der Universitat Dortmund am Facherich Informatik.

(Orlov et. al., 2009) Orlov, M. and Sipper, M. (2009) *Genetic Programming in the Wild: Evolving Unrestricted Bytecode*. In Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, pages 1043 – 1050.

(Perkis, 1994) Perkis, T. (1994) *Stack Based Genetic Programming*. In Proceedings of the First IEEE Conference on Evolutionary Computation, volume 1, pages 148 – 153, Albany, CA, USA.

(Poli et. al., 2008) Poli, R., Langdon, W. B., and McPhee, N. F. (2008) *A Field Guide to Genetic Programming*. http://www.gp-field-guide.org.uk

(UoL, a) *Declaration on Plagiarism and Collusion*. Department of Computer Science, University of Liverpool.

(UoL, b) *MSc PROJECTS COMP702 – SUMMER 2010*. Department of Computer Science, University of Liverpool.
http://www.csc.liv.ac.uk/~leszek/COMP702/

(Wexelblat, 1981) Wexelblat, R. L. (ed.) (1981) *History of Programming Languages*. Academic Press.

(Wilson et. al., 2008) Wilson, G. and Banzhaf, W. (2008) *A Comparison of Cartesian Genetic Programming and Linear Genetic Programming*. In Proceedings of the 11th European Conference on Genetic Programming, pages 182 – 193, Naples, Italy

(Wilson et. al., 2010) Wilson, G. and Banzhaf, W. (2010) *Deployment of parallel linear genetic programming using GPUs on PC and video game console platforms*. Department of Computer Science, Memorial University of Newfoundland, Canada.

(Woodward et. al., 2009) Woodward, J. R. and Bai, R. (2009) *Why Evolution is Not a Good Paradigm For Program Induction; A Critique of Genetic Programming*. In Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation, pages 593 – 600, China.

List of Figures

List of Tables

List of Program Listings

Appendix A Work Log

May 4th

Initial meeting with primary supervisor David Jackson to discuss available projects based on those described on the COMP702 web site.

May 14th

Second meeting with primary supervisor to discuss content of project. In this meeting, the subject of the dissertation and the scope of the project was finalised.

May 14th onwards

Development of original prototypes of genetic system. These prototypes use a different structure for the candidate solutions than the one used in this dissertation. In this software, instructions may be divisible. This was found to be an unproductive method, and was abandoned.

June 29th

Meeting with supervisor to discuss progress so far; experimental methodologies finalised.

July 1st

Presentation of research undertaken so far presented to Ph. D selection committee: scope of research, software produced, experimental methodologies and hypotheses.

July 2nd

Submission of 'Specification Document': details of research to be undertaken, examination of program to be researched and initial design of software.

July 15th

Meeting with supervisor to discuss progress so far, implementation and acceleration of evaluation of fitness function.

August 1st onwards

Design and implementation of the experimental testbed used to generate the results.

August 4th

Submission of 'Design Report': details of research undertaken, problem to be solved, experimental methodology, software design and changes to software from initial specification document.

August 6th

Design presentation to primary and secondary assessor: summary of experimental methodology and software design, elaboration of research undertaken and hypotheses produced.

August 10th

First experiments using LGP-based system are performed.

September 6th

Submission of 'Final Presentation Report': details of software design and implementation, primary analysis of results and preliminary conclusions.

September 9th

Final presentation to primary and secondary assessor: summary and discussion of results; software demonstration.

September 16th

Final meeting with secondary supervisor to discuss final content of dissertation.

September 24th

Submission of Dissertation

Appendix B Structure of Accompanying CD-ROM

/readme.txt

Guide to the files stored on CD-ROM. (This listing)

/Documents/1ProjectSpecification.doc
/Documents/1ProjectSpecification.pdf

Project Specification document submitted July 2nd.

/Documents/2DesignDocument.doc
/Documents/2DesignDocument.pdf

Design Document submitted August 4th.

/Documents/3DesignPresentation.ppt
/Documents/3DesignPresentation.pdf

Design Presentation presented August 6th.

/Documents/4FinalPresentationReport.doc
/Documents/4FinalPresentationReport.pdf

Final Presentation Report submitted September 6th.

/Documents/5FinalPresentation.ppt
/Documents/5FinalPresentation.pdf

Final Presentation Report submitted September 9th.

/Documents/6Dissertation.doc
/Documents/6Dissertation.pdf

Dissertation to be submitted September 24th.

/Software/Sourcecode/

Contains full source code for software described in this dissertation. Includes layout and environment files for use with Code::Blocks C++ IDE.

/Software/Executable/

Contains executable software (for Windows XP), and all files used in the experiments described in this dissertation.

evolve.exe

Executable software (for Windows XP).

program_---.txt

Plain text input file for programs considered in this dissertation.

perform_standard.bat

Batch file performing all experiments considering the ‘standard’ model of applying LGP.

perform_standard_for_computational_effort.bat
Batch file performing all experiments for computational effort considering the ‘standard’ model.

perform_standard_for_program_length.bat
Batch file performing all experiments for program length (no refinement) considering the ‘standard’ model.

perform_standard_for_program_length_refinement.bat
Batch file performing all experiments for program length (with refinement) considering the ‘standard’ model. This batch file should be run to perform all standard program length experiments as it performs 200 standard evolution attempts, and 5 refinement attempts on each of these.

perform_incremental.bat
Batch file performing all experiments considering the ‘incremental’ model of applying LGP.

perform_incremental_for_computational_effort.bat
Batch file performing all experiments for computational effort considering the ‘incremental’ model.

perform_incremental_for_program_length.bat
Batch file performing all experiments for program length (no refinement) considering the ‘incremental’ model.

perform_incremental_for_program_length_refinement.bat
Batch file performing all experiments for program length (with refinement) considering the ‘incremental’ model. This batch file should be run to perform all incremental program length experiments as it performs 200 complete incremental evolution attempts, and 5 refinement attempts on each of these.

tree_walking_compiler.bat
Batch file performing all experiments for program length when using the tree walking compiler.

tree_walking_compiler_with_refine.bat
Batch file performing all experiments for program length with refinement when using the tree walking compiler.

parameters_attempt_standard_for_computational_effort.txt

Parameter file containing the parameters for performing the standard evolution experiments for computational effort.

parameters_attempt_standard_for_program_length.txt
Parameter file containing the parameters for performing the standard evolution experiments for program length.

parameters_attempt_incremental_for_computational_effort.txt
Parameter file containing the parameters for performing the incremental evolution experiments for computational effort.

parameters_attempt_incremental_for_program_length.txt
Parameter file containing the parameters for performing the incremental evolution experiments for program length.

parameters_attempt_refinement.txt
Parameter file containing the parameters for performing the refinement stage.

parameters_attempt_refinement_200.txt
Parameter file containing the parameters for performing the refinement stage with 200 refinements for each found candidate solution.

/Data/RawData/

Contains all output files produced as a result of performing experiments. Files are labelled according to the experiment used to generate them and the input program they are associated with.

/Data/Analyses/

Contains Microsoft Excel 2000 .xls files analysing the results obtained as a result of the experiment. Analyses of computational effort values are accompanied by performance curves. Analyses of program length distributions are accompanied by histograms showing the relative frequency distribution of program lengths before and after the refinement stage.

Appendix C Specification of Input Programs

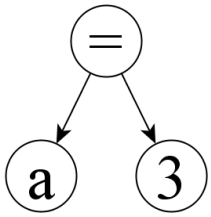
The following ten input programs are specified. These can be reproduced by executing the software using the following command-line arguments within a command prompt window and monitoring the console output.

```
evolve program_a01.txt tree_walking_compiler empty.txt empty.txt
          tree_walking_compiler_for_program_length_program_a01.txt
```

Program A01:

Assignment of a single constant to a variable

```
a = 3

Output: a
Symbolic Constant: 3

Number of nodes: 3
Number of interior nodes: 1
```



Tree walking compiler output:

```
LOADS  0,  3
STORS  0,  a

Length: 2
```

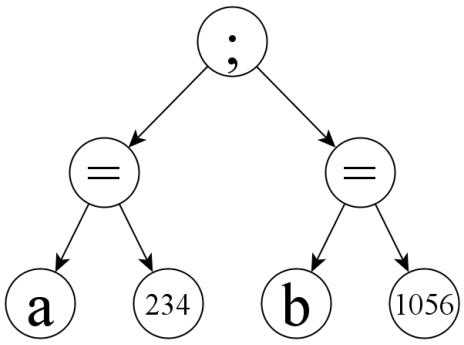
Optimal solution program:

```
LOADS  0,  3
STORS  0,  a

Length: 2
```

Program A02:

Assignment of two constants to two variables

```
a = 234;
b = 1056

Output: a, b
Symbolic Constant: 234, 1056

Number of nodes: 7
Number of interior nodes: 3
```



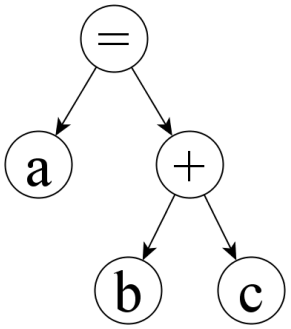
Tree walking compiler output:

```
LOADS  0,  234
STORS  0,  a
LOADS  0,  1056
STORS  0,  b

Length: 4
```

Optimal solution program:

```
LOADS  0,  234
STORS  0,  a
LOADS  0,  1056
STORS  0,  b

Length: 4
```

Program B01:
Simple calculation; two input variables
one output variable




```
a = b + c

Input: b, c
Output: a

Number of nodes: 5
Number of interior nodes: 2
```

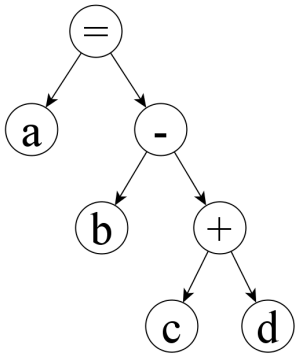
Tree walking compiler output:

```
LOADS  0,  b
LOADS  1,  c
ADD    0,   0,   1
STORS  0,  a

Length: 4
```

Optimal solution program:

```
LOADS  0,  b
LOADS  1,  c
ADD    0,   0,   1
STORS  0,  a

Length: 4
```

Program B02:
Progressively more complex calculation




```
a = b - (c + d)

Input: b, c, d
Output: a

Number of nodes: 7
Number of interior nodes: 3
```

Tree walking compiler output:

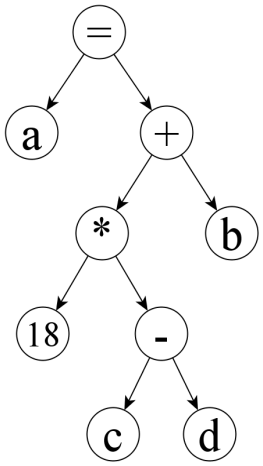
```
LOADS  0,  b
LOADS  1,  c
LOADS  2,  d
ADD    1,   1,   2
SUB    0,   0,   1
STORS  0,  a

Length: 6
```

Optimal solution program:

```
LOADS  0,  b
LOADS  1,  c
LOADS  2,  d
ADD    1,   1,   2
SUB    0,   0,   1
STORS  0,  a

Length: 6
```

Program B03:
Complex calculation involving a constant

```
a = (18 * (c - d)) + b

Input: b, c, d
Output: a
Constant: 18

Number of nodes: 9
Number of interior nodes: 4
```



Tree walking compiler output:

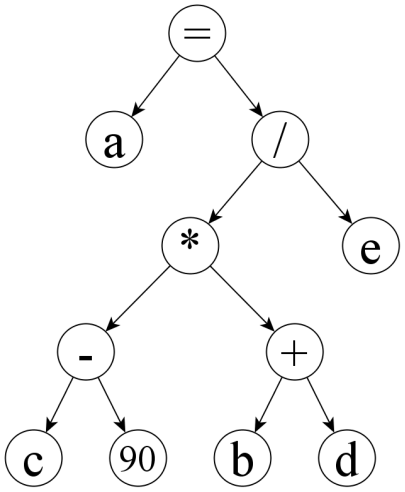
```
LOADS  0,  18
LOADS  1,  c
LOADS  2,  d
SUB    1,   1,   2
MUL    0,   0,   1
LOADS  1,  b
ADD    0,   0,   1
STORS  0,  a

Length: 8
```

Optimal solution program:

```
LOADS  0,  18
LOADS  1,  c
LOADS  2,  d
SUB    1,   1,   2
MUL    0,   0,   1
LOADS  1,  b
ADD    0,   0,   1
STORS  0,  a

Length: 8
```

Program B04:
Complex calculation with division

```
a = ((c - 90) * (b + d)) / e

Input: b, c, d, e
Output: a
Constant: 90

Number of nodes: 11
Number of interior nodes: 5
```



Tree walking compiler output:

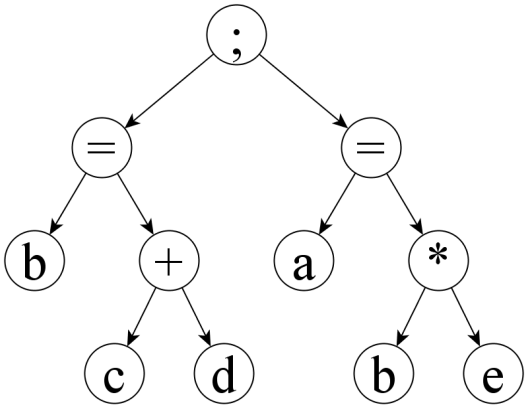
```
LOADS  0,  c
LOADS  1,  90
SUB    0,   0,   1
LOADS  1,  b
LOADS  2,  d
ADD    1,   1,   2
MUL    0,   0,   1
LOADS  1,  e
DIVP   0,   0,   1
STORS  0,  a

Length: 10
```

Optimal solution program:

```
LOADS  0,  c
LOADS  1,  90
SUB    0,   0,   1
LOADS  1,  b
LOADS  2,  d
ADD    1,   1,   2
MUL    0,   0,   1
LOADS  1,  e
DIVP   0,   0,   1
STORS  0,  a

Length: 10
```

Program C01:
Calculation involving intermediate variable

```
b = c + d;
a = b * e

Input: c, d, e
Output: a
Intermediate: b

Number of nodes: 11
Number of interior nodes: 5
```



Tree walking compiler output:

```
LOADS  0,  c
LOADS  1,  d
ADD    0,   0,   1
STORS  0,  b
LOADS  0,  b
LOADS  1,  e
MUL    0,   0,  1
STORS  0,  a

Length: 9
```

Optimal solution program:

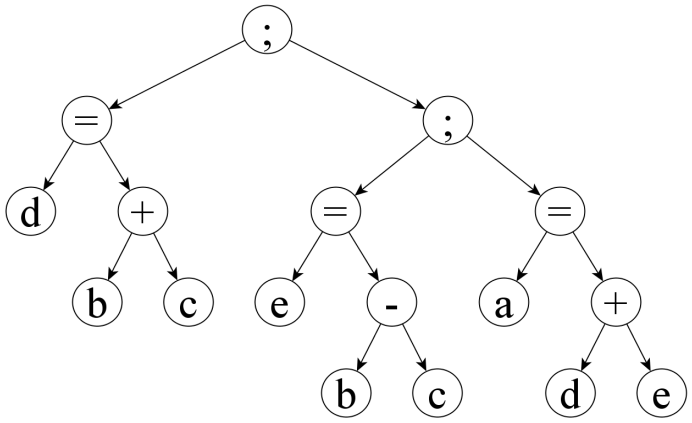
```
LOADS  0,  c
LOADS  1,  d
ADD    0,   0,   1
LOADS  1,  e
MUL    0,   0,  1
STORS  0,  a

Length: 7
```

Program D01:

Calculation involving intermediate variables; optimisations possible
(a = 2*b)

```
d = b + c;
e = b - c;
a = d + e
```

```
Input: b, c
Output: a
Intermediate: d, e

Number of nodes: 17
Number of interior nodes: 8
```



Tree walking compiler output:

```
LOADS  0,  b
LOADS  1,  c
ADD    0,   0,   1
STORS  0,  d

LOADS  0,  b
LOADS  1,  c
SUB    0,   0,   1
STORS  0,  e

LOADS  0,  d
LOADS  1,  e
ADD    0,   0,   1
STORS  0,  a

Length: 12
```

Optimal solution program:

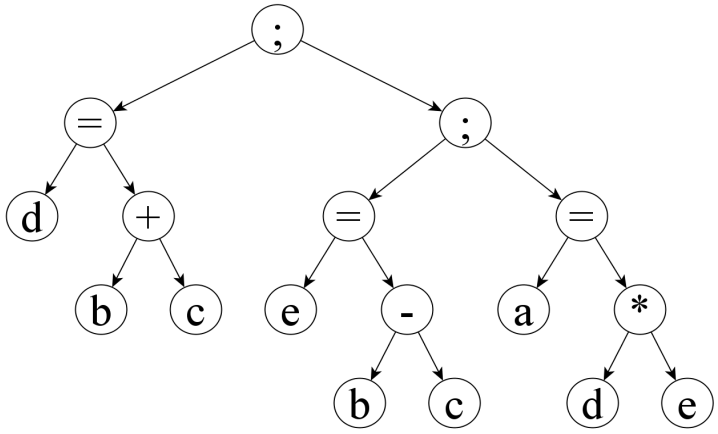
```
LOADS  0,  b
ADD    0,  0,  0
STORS  0,  a

Length: 3
```

Program D02:

Calculation involving intermediate variables; difference of two squares (a = b*b – c*c)

```
d = b + c;
e = b - c;
a = d * e
```

```
Input: b, c
Output: a
Intermediate: d, e

Number of nodes: 17
Number of interior nodes: 8
```



Tree walking compiler output:

```
LOADS  0,  b
LOADS  1,  c
ADD    0,   0,   1
STORS  0,  d

LOADS  0,  b
LOADS  1,  c
SUB    0,   0,   1
STORS  0,  e

LOADS  0,  d
LOADS  1,  e
MUL    0,   0,   1
STORS  0,  a

Length: 12
```

Optimal solution program:

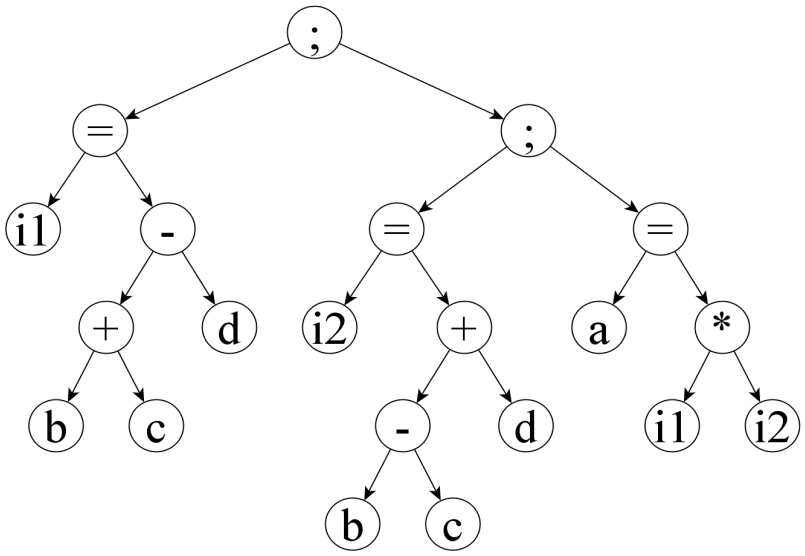
```
LOADS  0,  b
MUL    0,   0,   0
LOADS  1,  c
MUL    1,   1,   1
SUB    0,   0,   1
STORS  0,  a

Length: 6
```

Program D03:

Complex calculation involving intermediate variables

```
i1 = ((b + c) - d);
i2 = ((b - c) + d);
a = i1 * i2
```

```
Input: b, c, d
Output: a
Intermediate: i1, i2

Number of nodes: 21
Number of interior nodes: 10
```



Tree walking compiler output:

```
LOADS  0,  b
LOADS  1,  c
ADD    0,   0,   1
LOADS  1,  d
SUB    0,   0,   1
STORS  0,  i1

LOADS  0,  b
LOADS  1,  c
SUB    0,   0,   1
LOADS  1,  d
ADD    0,   0,   1
STORS  0,  i2

LOADS  0,  i1
LOADS  1,  i1
MUL    0,   0,   1
STORS  0,  a

Length: 16
```

Optimal solution program:

```
LOADS  0,  c
LOADS  1,  d
SUB    0,   0,   1
MUL    0,   0,   0
LOADS  1,  a
MUL    1,   1,   1
SUB    0,   1,   0
STORS  0,  a

Length: 8
```

Appendix C Summary of Data

The full raw output of the evolve.exe program can be found on the accompanying CD-ROM in the directory RAW_DATA, together with the files used to collate this data in its aggregate form.

C.1 Aggregated Result Data – Computational Effort

Computational effort E_I values:

	'Standard'	'Incremental'	No. internal nodes	No. nodes
PROGRAM A01	52000	180000	1	3
PROGRAM A02	432000	359000	3	7
PROGRAM B01	203000	250500	2	5
PROGRAM B02	1616000	404500	3	7
PROGRAM B03	16590000	454500	4	9
PROGRAM B04	----------	1595500	5	11
PROGRAM C01	1816000	538000	5	11
PROGRAM D01	450000	787000	8	17
PROGRAM D02	4842000	845500	8	17
PROGRAM D03	619406000	1031000	10	21

The 'standard' value for Program B04 was incalculable due to exceptionally low probability of success for any number of instruction considerations.

C.2 Aggregated Result Data – Program Length

Distribution of solution program lengths:

PROGRAM A01	Minimum	Maximum	Mean	Median	Mode
'Standard'	3	95	22.42	21	6
'Incremental'	3	44	15.235	15	16
'Standard with refinement'	2	2	2	2	2
'Incremental with refinement'	2	2	2	2	2

PROGRAM A02	Minimum	Maximum	Mean	Median	Mode
'Standard'	5	63	21.105	19	12
'Incremental'	14	89	39.09	37.5	33
'Standard with refinement'	4	60	6.218	4	4
'Incremental with refinement'	4	9	4.008	4	4

PROGRAM B01	Minimum	Maximum	Mean	Median	Mode
'Standard'	4	46	14.09	13	11
'Incremental'	11	39	22.375	22	19
'Standard with refinement'	4	26	5.566	4	4
'Incremental with refinement'	4	30	6.991	6	4

PROGRAM B02	Minimum	Maximum	Mean	Median	Mode
'Standard'	9	46	21.545	21	17
'Incremental'	17	60	33.305	33	35
'Standard with refinement'	6	37	10.864	10	7
'Incremental with refinement'	9	44	21.902	21	20

PROGRAM B03	Minimum	Maximum	Mean	Median	Mode
'Standard'	15	47	28.51	28	26
'Incremental'	26	63	44.165	44	43
'Standard with refinement'	8	37	15.069	14	12
'Incremental with refinement'	15	56	35.436	35	37

PROGRAM B04	Minimum	Maximum	Mean	Median	Mode
'Standard'	29	62	41	38	38
'Incremental'	36	85	56.65	56	60
'Standard with refinement'	12	27	18.7	16.5	14
'Incremental with refinement'	28	81	48.968	48	46

Program B04 was only run for 5 independent runs (each with 20 refinement attempts) using the 'standard' (with refinement) model due to the unlikelihood of finding 200 solution programs within a reasonable amount of time.

PROGRAM C01	Minimum	Maximum	Mean	Median	Mode
'Standard'	19	27	22.6	23	N/A
'Incremental'	30	83	51.51	50	49
'Standard with refinement'	6	21	9.27	7	6
'Incremental with refinement'	10	68	34.654	34	32

Program C01 was only run for 5 independent runs (each with 20 refinement attempts) using the 'standard' (with refinement) model due to time constraints.

PROGRAM D01	Minimum	Maximum	Mean	Median	Mode
'Standard'	17	30	21.2	19	N/A
'Incremental'	54	119	82.84	82.5	78
'Standard with refinement'	3	18	6.62	3	3
'Incremental with refinement'	3	95	58.73	60	63

Program D01 was only run for 5 independent runs (each with 20 refinement attempts) using the 'standard' (with refinement) model due to time constraints.

PROGRAM D02	Minimum	Maximum	Mean	Median	Mode
'Standard'	20	32	27.6	30	32
'Incremental'	54	111	81.91	80	79
'Standard with refinement'	6	21	11.41	12	6
'Incremental with refinement'	25	91	59.604	59	61

Program D02 was only run for 5 independent runs (each with 20 refinement attempts) using the 'standard' (with refinement) model due to time constraints.

ROGRAM D03	Minimum	Maximum	Mean	Median	Mode
'Standard'	29	44	37.6	42	N/A
'Incremental'	78	147	103.935	103	103
'Standard with refinement'	10	39	21.93	22	14
'Incremental with refinement'	54	129	83.68	83	79

Program D03 was only run for 5 independent runs (each with 20 refinement attempts) using the 'standard' (with refinement) model due to time constraints.

C.3 Non-Evolutionary Data – Program Length

Solution program lengths in instructions when using non genetic methods:

	Human Programmer (optimised)	Tree Walking Compiler (unoptimised)
PROGRAM A01	2	2
PROGRAM A02	4	4
PROGRAM B01	4	4
PROGRAM B02	6	6
PROGRAM B03	8	8
PROGRAM B04	10	10
PROGRAM C01	6	8
PROGRAM D01	3	12
PROGRAM D02	6	12
PROGRAM D03	8	16

Appendix D User Guide to Software

D.1 General

The software requires the user to supply command-line arguments instructing it which type of experiment to perform, the location of the input program on which to perform code generation, and the location of the parameter files.

The program has the following invocation syntax:

```
evolve.exe [source_program_file] [analysis_type] [parameters_file]
               [parameters_file2] [output_file]
```

`[source_program_file]` is a string containing the location of the input program on the hard drive. Input programs are stored in a plain text format for easy modification by the user.

`[analysis_type]` is a string indicating what form of experiment is to be performed. It can take one of the following values:

`attempt_standard`
: Perform evolution using the 'standard' model using the parameters in `[parameters_file]`.

`attempt_standard_with_refine`
: Perform evolution using the 'standard' model using the parameters in `[parameters_file]`, followed by the refinement stage using the parameters in `[parameters_file2]`.

`attempt_incremental`
: Perform evolution using the 'incremental' model using the parameters in `[parameters_file]`.

`attempt_incremental_with_refine`
: Perform evolution using the 'incremental' model using the parameters in `[parameters_file]`, followed by the refinement stage using the parameters in `[parameters_file2]`.

`tree_walking_compiler`
: Perform code generation of the input program using the tree walking algorithmic compiler, using the parameters in `[parameters_file]`.

`tree_walking_compiler_with_refine`
: Perform code generation of the input program using the tree walking algorithmic compiler, using the parameters in `[parameters_file]`, followed by the refinement stage using the parameters in `[parameters_file2]`.

[parameters_file] is a string containing the location of the parameters file containing the parameters to use during the primary stage of evolution. [parameters_file2] is a string containing the location of the parameters file containing the parameters to use during the refinement stage of evolution.

[output_file] is a string containing the location of the file to which the experiment results will be saved. If this file already exists, the results are appended to the end of the existing file.

Example:

```
evolve.exe program_a01.txt attempt_standard_with_refine
             parameters_attempt_standard_for_program_length.txt
             parameters_attempt_refinement.txt
             attempt_standard_with_refine_for_program_length_program_a01.txt
```

This example instructs the software to attempt to evolve a solution to the input program stored in the file program_a01.txt using the 'standard' model of evolution followed by the refinement stage. The parameters for the 'standard' phase of evolution are stored in the file parameters_attempt_standard_for_program_length.txt and the parameters for the refinement stage are stored in the file parameters_attempt_refinement.txt. The output results are stored in the file attempt_standard_with_refine_for_program_length_program_a01.txt.

D.2 Parameter Files

A parameter file is a plain text file containing a list of values for parameters in the form <parameter> <value>. A parameter file may contain values for the following parameters.

population_size <integer>
Sets the population size used in the LGP system to <integer> programs.

initial_maximum_program_size_in_instructions <integer>
initial_minimum_program_size_in_instructions <integer>
Sets the values of the depth ramp used to initialise the population.

discard_crossover_programs_of_size_above <integer>
Instructs the LGP system to reselect the transition points for the crossover operation if the size of the resulting program is over <integer>. To disable this functionality, set this value to a high value such as 1000000.

max_creations_per_run <integer>
Sets the maximum number of candidate program recreations allowed before the current run is aborted.

`tournament_size_for_crossover <integer>`
`tournament_size_for_reproduction <integer>`
`tournament_size_for_mutation <integer>`
`tournament_size_for_deletion <integer>`
Sets the sizes used in the tournaments for selection of programs for crossover, reproduction (exact duplication), mutation and deletion (removal of unfit programs).

`max_runs <integer>`
Sets the number of independent runs used as part of niche-preemption to ensure the eventual evolution of an acceptable candidate solution.

`repeat_experiment <integer>`
Sets the number of times that independent experiments will be performed. For evolution of new programs, this is the number of complete experiments to find a solution that will be attempted. For refinement evolution, this is the number of refinements that will be attempted for each acceptable solution previously found by other methods.

`rate_of_crossover <float>`
`rate_of_mutation <float>`
`rate_of_reproduction <float>`
Sets the rates of application of the crossover, mutation and reproduction recombination operations (as values between 0 and 1).

`fitness_case_training_set_count <integer>`
Sets the number of fitness cases used during training of the population.

`fitness_case_training_set_count <integer>`
Sets the number of additional fitness cases applied after a candidate solution has passed the training set to ensure that the candidate solution is sufficiently general.

`maximum_acceptable_fitness <float>`
Imposes an addition condition on the maximum acceptable fitness of a candidate solution. This is set to a high value for evolution of new programs to ensure that they terminate as soon as a candidate solution which passes the training set is found. For refinement evolution, this is set to a low value so that evolution will continue even though the population will already contain acceptable solution programs (the termination criterion then becomes exhaustion of available candidate recreations).

`instruction_executions_max_count <integer>`
Sets the maximum number of instructions that may be executed in the low level virtual machine before execution is forcefully terminated with a bad error state.

`vm_register_general_purpose_count <integer>`

Sets the number of general purpose registers present in the register file of the low level architecture. May be up to 8. (This may be altered by changing the constant `VM_REGISTER_COUNT` and recompiling the software.

`adaptive_instruction_set <boolean>`
Enables a mode where the software is allowed to analyse the structure of the parse tree before evolution and adapt the instruction set available to the LGP system based on this knowledge. This mode is not used in the experiments described in this dissertation as it allows the system a priori information about relationships between the instruction set and the parse tree.

`adaptive_instruction_set <float>`
Sets the ratio `1/<float>` of the adaptive addition to the instruction set. This mode is not used in the experiments described in this dissertation as it allows the system a priori information about relationships between the instruction set and the parse tree.

`reporting <boolean>`
If set to true, the software will output status reports to the console window at regular intervals.

`reporting_granularity_in_new_creations <integer>`
Where reporting is enabled, a report is shown during evolution no sooner than every `<integer>` recreations.

`reporting_delay_in_seconds <float>`
Where reporting is enabled, a report is shown during evolution no sooner than every `<float>` seconds.

`interactive <boolean>`
If set to true, the software will wait for the user to press the Return key after each stage of reporting.

Any other directives (such as 'end') appearing in the parameter file are ignored as comments.

D.3 Console Screen Reporting

If reporting is enabled, the following output reports are printed to the console window at the various stages of evolution:

First, the input program is shown in the form it appears in the input source file, with all white space removes, and in Polish notation.

```
Read:
  d = b + c;  e = b - c;  a = d + e
Nospaced:
```

```
d=b+c;e=b-c;a=d+e
Prefixed:
; = d + b c ; = e - b c = a + d e
```

Then, the overall symbol table is printed.

```
Variable          b is an INPUT.
Variable          c is an INPUT.
Variable          a is an OUTPUT.
Variable          d is an INTERMEDIATE.
Variable          e is an INTERMEDIATE.
Symbol table key ' 1': b        Flags: WY RY NN ON DY SN
Symbol table key ' 2': c        Flags: WY RY NN ON DY SN
Symbol table key ' 3': a        Flags: WY RY NY ON DN SN
Symbol table key ' 4': d        Flags: WY RY NN ON DN SN
Symbol table key ' 5': e        Flags: WY RY NN ON DN SN
```

The flags correspond to the following variable properties: (Y is Yes, N is No)

W: (Writable) The variable can be written to.

R: (Readable) The variable can be read from.

N: (New value) The semantics of the IR require that the value of this variable be consistent with the value it takes after high level interpretation of the parse tree.

O: (Old value) The semantics of the IR require that the value of this variable remain unchanged through execution of the low level program.

D: (Determinate) The contents of this variable are determinate at program start.

S: (Symbolic constant) This variable is a symbolic constant with a value defined by its symbolic name (a numeric literal).

Then, the program will report the construction of the parse tree in memory, followed by the results of loading the two parameter files.

```
Loaded 'population_size = 1000'
Loaded 'initial_maximum_program_size_in_instructions = 20'
Loaded 'initial_minimum_program_size_in_instructions = 2'
Loaded...
```

For each evolution attempt, the target program is printed, together with the full symbol table and the instruction set (with selection rate).

```
BEGIN EVOLUTION:
Trying to evolve program:

    (M001 = (b + c))

Symbol table key ' 1': b        Flags: WN RY NN OY DY SN
Symbol table key ' 2': c        Flags: WN RY NN OY DY SN
```

```
Symbol table key ' 3': a        Flags: WN RN NN OY DN SN
Symbol table key ' 4': d        Flags: WN RN NN OY DN SN
Symbol table key ' 5': e        Flags: WN RN NN OY DN SN
Symbol table key ' 6': M0       Flags: WN RN NN OY DN SN
Symbol table key ' 7': M00      Flags: WN RN NN OY DN SN
Symbol table key ' 8': M001     Flags: WY RY NY ON DN SN
Symbol table key ' 9': M01      Flags: WN RN NN OY DN SN
Symbol table key '10': M010     Flags: WN RN NN OY DN SN
Symbol table key '11': M0101    Flags: WN RN NN OY DN SN
Symbol table key '12': M011     Flags: WN RN NN OY DN SN
Symbol table key '13': M0111    Flags: WN RN NN OY DN SN

Instruction Set:
ADD       , 0.143 (<= 0.143)
SUB       , 0.143 (<= 0.286)
MUL       , 0.143 (<= 0.429)
DIVP      , 0.143 (<= 0.571)
LOADV     , 0.143 (<= 0.714)
LOADS     , 0.143 (<= 0.857)
STORS     , 0.143 (<= 1.000)
```

During evolution, the following report screen is displayed.

```
-[ TIME     33.0s - NEW_CREATION   12720 - EFF GEN   12 - BEST FITNESS 10132200.
000 - RUN    0 ]-
 10132200.00, 20 hits:
Instruction    0: LOADS  0,  M001          => r[0] = M001;
Instruction    1: LOADS  3,  b             => r[3] = b;
Instruction    2: STORS  3,  M001          => M001 = r[3];
Instruction    3: LOADS  3,  b             => r[3] = b;
Instruction    4: LOADS  2,  c             => r[2] = c;
Instruction    5:   ADD  3,   3,   2       => r[3] = r[3] + r[2];
Instruction    6: LOADS  0,  c             => r[0] = c;
Instruction    7: LOADS  2,  b             => r[2] = b;
Instruction    8: LOADS  2,  M001          => r[2] = M001;
Instruction    9: LOADS  1,  M001          => r[1] = M001;
Instruction   10: LOADS  3,  b             => r[3] = b;
Instruction   11: LOADS  2,  c             => r[2] = c;
Instruction   12:   ADD  3,   3,   2       => r[3] = r[3] + r[2];
Instruction   13: LOADS  3,  c             => r[3] = c;
Instruction   14:   ADD  3,   2,   1       => r[3] = r[2] + r[1];
Instruction   15: STORS  3,  M001          => M001 = r[3];
Instruction   16: LOADS  3,  b             => r[3] = b;
Instruction   17: LOADS  0,  d             => r[0] = d;
Instruction   18: STORS  0,  d             => d = r[0];
Instruction   19: LOADV  1,    1           => r[1] = 1;
Instruction   20: LOADS  2,  M00           => r[2] = M00;
Instruction   21: LOADS  1,  M00           => r[1] = M00;
Instruction   22: LOADS  1,  M001          => r[1] = M001;
Instruction   23: STORS  1,  d             => d = r[1];
Instruction   24: LOADS  2,  M001          => r[2] = M001;
Instruction   25: STORS  2,  M00           => M00 = r[2];
Instruction   26: LOADS  2,  b             => r[2] = b;
Instruction   27: STORS  2,  M0101         => M0101 = r[2];
Instruction   28: LOADS  3,  b             => r[3] = b;
Instruction   29: LOADS  3,  c             => r[3] = c;
Instruction   30: LOADS  1,  M0101         => r[1] = M0101;
Instruction   31: LOADS  0,  M0101         => r[0] = M0101;
Instruction   32: LOADS  0,  c             => r[0] = c;
Instruction   33: LOADV  2,    8           => r[2] = 8;
Instruction   34:   ADD  2,   0,   0       => r[2] = r[0] + r[0];
Instruction   35:   SUB  0,   1,   0       => r[0] = r[1] - r[0];
Instruction   36: STORS  0,  M0101         => M0101 = r[0];
Instruction   37: LOADS  0,  b             => r[0] = b;
Instruction   38:   MUL  0,   1,   2       => r[0] = r[1] * r[2];
Instruction   39: LOADV  1,    2           => r[1] = 2;
Instruction   40:   SUB  0,   3,   1       => r[0] = r[3] - r[1];
```

The top row indicates the time spent so far in the current run, the number of new candidate programs produced through recombination so far, the effective generation

(the number of candidates generated divided by the population size), the fitness value of the best candidate program in the population and the niche-preemption run number.

The remainder of the screen shows the fitness value, hits count, and the full listing of the best candidate program in the population. The left column shows an assembly-like listing as used throughout this dissertation. The right column shows the semantic interpretation of the assembly-like instructions in a C-like language. If this listing is copied into a C source file and the string '`Instruction`' replaced with '`/*`' and the string '`=>`' replaced with '`*/`', this listing can be executed with little effort.

D.4 Plain Text Program Input File Format

The input file format is a simple, human readable plain-text format:

```
--------------
Calculation involving intermediate variables.

INPUT             c d e
OUTPUT            a
INTERMEDIATE      b
SYMBOLIC_CONSTANT 2

PROGRAM
  b = (c + d);
  a = ((b * e) + 2)
ENDPROGRAM
--------------
```

The `INPUT`, `OUTPUT`, `INTERMEDIATE` and `SYMBOLIC_CONSTANT` lines define the symbol table used in the main program. Each of the lines contains a space-separated list of variable names. Variable names may consist of character strings (a-z, A-Z, 0-9) of any length, but must not start with a number. Additional `INPUT`, `OUTPUT`, etc. lines may be used to construct long lists of variables. These categories map exactly to those symbol table variable kinds defined in the Scope subsection of the Solution Methodology. Numeric literals used in the program body (such as 2) must be declared in advance in the `SYMBOLIC_CONSTANT` section.

The input program is given between the `PROGRAM` and `ENDPROGRAM` statements; the file parser concatenates all the non-whitespace characters between these lines to form the input program in its high level form. This input language recognises variable names, the assignment (=), sequencing (;), addition (+), subtraction (-), multiplication (*) and protected division (/) operators, and parentheses. The sequencing operator enforces the order of evaluation between statements, but the program must not end on a semicolon. Space characters and newlines are ignored. There exists operator precedence mirroring that of C, though this can be overridden with parentheses to explicitly lay out the parse tree.

Any characters outside the PROGRAM and ENDPROGRAM pair that are not part of a symbol table definition line are ignored as comments (such as the dashed lines and the program description).

D.5 Output File Formats

The format of the output file is different depending on the type of evolution used:

Standard uses the following format:

```
experiment 0
evolve (a=((18*(c-d))+b))
26000 -1 536439 1 aborted
experiment_over 0
experiment 1
evolve (a=((18*(c-d))+b))
22088 22 347685 1 verified
experiment_over 1
…
```

Each experiment is bounded by an 'experiment' 'experiment_over' pair. An evolution attempt begins with 'evolve' followed by the target program. The five entries in the next row indicate the number of candidate solutions considered before the run ended, the length of the candidate solution in instructions (negative if no solution was found), the number of instruction considerations used, the number of niche-preemption runs used (this is always 1 for computational effort), and whether run ended with the creation of an acceptable solution ('verified') or ended due to abort ('aborted').

Incremental uses a more complicated format:

```
experiment 0
evolve (M0101=(c-d))
7542 15 60677 1 verified
constructed (c-d)
length 15
evolve (M010=(18*M0101))
4066 15 33139 1 verified
constructed (18*(c-d))
length 30
evolve (M01=(M010+b))
3223 8 21818 1 verified
constructed ((18*(c-d))+b)
length 38
evolve (M0=(a=M01))
2365 7 19544 1 verified
constructed (a=((18*(c-d))+b))
length 45
experiment_over 0
…
```

Each experiment is bounded by an 'experiment' 'experiment_over' pair. A fragment evolution attempt begins with 'evolve' followed by the target program fragment. The five entries in the next row indicate the number of candidate solutions considered before the run ended, the length of the candidate solution in instructions (negative if no solution was found), the number of instruction considerations used, the number of niche-preemption runs used (this is always 1 for computational effort), and whether run ended with the creation of an acceptable solution ('verified') or ended due to abort ('aborted'). As the program is assembled in memory from the subprograms, the length of the composite program is reported as 'length x'. The final 'length' value is the length of the complete program.

Where refinement is used, additional lines are inserted:

```
experiment 0
evolve (M01=(b+c))
2900 15 23654 1 verified
constructed (b+c)
length 15
evolve (M0=(a=M01))
3058 19 23729 1 verified
constructed (a=(b+c))
length 34
refinement 0
evolve (a=(b+c))
26000 6 484011 1 verified
refinement_over 0
refinement 1
evolve (a=(b+c))
26000 14 610821 1 verified
refinement_over 1
refinement 2
evolve (a=(b+c))
26001 13 595384 1 verified
refinement_over 2
...
experiment_over 0
...
```

After the complete program has been evolved, the refinement attempts are contained within 'refinement' 'refinement_over' pairs. The content of these pairs is the same as for the evolution of programs: number of candidate solutions considered (for refinement, this is always the maximum permitted by the parameter file), the length of the final program, the number of instruction considerations, the number of pre-emption runs (refinement does not use this, however), and whether run ended with the creation of an acceptable solution or ended due to abort.

The analyses performed in this dissertation were performed by interpreting these output files as space-separated-value files.

Appendix E Function and Data Type Reference

E.1 Data Type and Constant Reference

`typedef long long BigInteger`

`BigInteger` is the base signed integer data type used throughout the project for the high level interpretation of the parse tree and the low level interpretation of candidate programs. Any data type with a full implementation of the arithmetic operators such as GNU GMP's `mpz_class` can be used in place of long long.

`template <typename T> struct BoundsPair`

`BoundsPair` is a composite type used to represent a pair of bounds for a variable, such as when input variables are given random values at the time of fitness case initialisation. It contains methods allowing a variable to be compared with the bounds values.

`struct SymbolicVariableData`

`SymbolicVariableData` represents a row within the symbol table and contains all the information relating to a single variable such as its readable symbolic name, whether or not it can be written to, read from, and whether its value is important with respect to establishing the degree of semantic correlation.

`enum SYMBOL_TABLE_VARIABLE_ARCHETYPE`

`SYMBOL_TABLE_VARIABLE_ARCHETYPE` defines a number of constants used for convenience when adding variables to the symbol table. When a symbol is placed into the symbol table, an `ARCHETYPE` is used to indicate what type of variable is being inserted.

- `SYMBOL_TABLE_VARIABLE_ARCHETYPE_PARAMETER,` Incoming value as function parameter. Writable, Readable, Value irrelevant, Determinate.
- `SYMBOL_TABLE_VARIABLE_ARCHETYPE_VARIABLE,` Incidental variable in calculation Writable, Readable, Value irrelevant, Indeterminate.
- `SYMBOL_TABLE_VARIABLE_ARCHETYPE_RESULT,` Outgoing value from function parameter. Writable, Readable, Value should match end state. Indeterminate.
- `SYMBOL_TABLE_VARIABLE_ARCHETYPE_PARAMETER_RESULT,` Outgoing value from function parameter. Writable, Readable, Value should match end state. Determinate.
- `SYMBOL_TABLE_VARIABLE_ARCHETYPE_CONSTANT,` Constant. Not writable, readable, value should match start state. Determinate.
- `SYMBOL_TABLE_VARIABLE_ARCHETYPE_SYMBOLIC_CONSTANT,` Symbolic constant. Not writable, readable, value should match start state. Determinate.
- `SYMBOL_TABLE_VARIABLE_ARCHETYPE_BYSTANDER,` Variable that is not used in calculation Not writable, not readable, value should match start state. Indeterminate.

typedef unsigned int SYMBOL_TABLE_KEY

SYMBOL_TABLE_KEY is used as the primary key into the SymbolTable to refer to a variable. SYMBOL_TABLE_KEY is used wherever a reference to a variable is necessary.

typedef std::map<SYMBOL_TABLE_KEY,SymbolicVariableData> SymbolTableData

SymbolTableData is the main storage structure within a SymbolTable mapping SYMBOL_TABLE_KEYs to SymbolicVariableData records.

const SYMBOL_TABLE_KEY SYMBOL_TABLE_KEY_BAD = 0

The value zero is a reserved SYMBOL_TABLE_KEY to signify an illegal access attempt upon a SymbolTable.

enum RANDOM_SYMBOL_TABLE_KEY_NATURE

Enumeration allowing for the retrieval of a random SYMBOL_TABLE_KEY relating to a variable that is readable, writable or both.

- RANDOM_SYMBOL_TABLE_KEY_ANY – Return any variable randomly.
- RANDOM_SYMBOL_TABLE_KEY_WRITABLE – Return a writable variable randomly.
- RANDOM_SYMBOL_TABLE_KEY_READABLE – Return a readable variable randomly.

typedef BigInteger PT_VALUE_TYPE

PT_VALUE_TYPE is the single 'value' data type shared throughout high level interpretation of a parse tree.

typedef SYMBOL_TABLE_KEY PT_REFERENCE_TYPE

PT_REFERENCE_TYPE is the type used to manipulate references during high level interpretation of a parse tree. It is used to store the reference produced by the ereferation of a variable node as the left child of the assignment operator node.

typedef unsigned int PT_COMPLEXITY_TYPE

PT_COMPLEXITY_TYPE is the type used to store complexity heuristic values during high level interpretation of a parse tree.

const PT_COMPLEXITY TYPE PARSE_TREE_COMPLEXITY_EVALUATING_CONSTANT

This constant indicates the complexity heuristic contribution of the evaluation of a constant (loading an integer constant from a node).

const PT_COMPLEXITY TYPE PARSE_TREE_COMPLEXITY_INITIALISING_CONSTANT

This constant indicates the complexity heuristic contribution of the placing of a constant into memory during fitness case creation.

struct ParseTreeEvaluationResult

This structure combines a PT_VALUE_TYPE value type result with a PT_COMPLEXITY_TYPE complexity heuristic value to form a composite type that is the result of any 'evaluation' operation performed during high level interpretation of a parse tree.

struct ParseTreeEreferationResult

This structure combines a `PT_REFERENCE_TYPE` reference type result with a `PT_COMPLEXITY_TYPE` complexity heuristic value to form a composite type that is the result of any 'ereferation' operation performed during high level interpretation of a parse tree.

struct ParseTreeEvaluationEnvironment

`ParseTreeEvaluationEnvironment` stores the full state of the interpreter during high level interpretation of a parse tree. It holds a handle to the associated `SymbolTable` part of the input IR, the full variable memory, the `ParseTreeEvaluationResult` produced by the root node of the parse tree, the number of evaluation calls; this can used to implement an upper bound of calls during interpretation using `EVALUATIONS_LIMIT`.

const unsigned int MAXIMUM_PARSE_TREE_OPERATOR_LIKE_SYMBOL_ARITY = 4

To avoid costly memory management operations, the `ParseTreeNode` structure representing a node is set to hold a static array of `ParseTreeNode` structure pointers pointing to its children. This constant defines the size of this static array. If the high level language is modified to include structures with more than this number of children, the user should increase this number.

enum LOGGED_OPERATION_PERFORMED

This enumeration defines a number of constants used to designate which type of high level operation was performed within each `ParseTreeEvaluationLogEntry`.

- `LOGGED_OPERATION_INITIALISATION,` – initialisation at program start
- `LOGGED_OPERATION_ADDITION,` – addition parse tree node
- `LOGGED_OPERATION_SUBTRACTION,` – subtraction parse tree node
- `LOGGED_OPERATION_MULTIPLICATION,` – multiplication parse tree node
- `LOGGED_OPERATION_DIVISION,` – division parse tree node
- `LOGGED_OPERATION_CONSTANT_LOAD,` – load of constant from variable
- `LOGGED_OPERATION_RETURN,` – return statement (not used)
- `LOGGED_OPERATION__END_OF_LIST,` – signifies highest value of enumeration

const char *LOGGED_OPERATION_PERFORMED_STRINGS[LOGGED_OPERATION__END_OF_LIST]

Maps `enum LOGGED_OPERATION_PERFORMED` values into printable strings for debugging output.

struct ParseTreeEvaluationLogEntry

Contains a record of a single evaluation performed during high level interpretation of a parse tree. Contains a `enum LOGGED_OPERATION_PERFORMED` value indicating the operation performed, the `ParseTreeEvaluationResult` values of the operands and the `ParseTreeEvaluationResult` value of the result.

typedef std::vector<ParseTreeEvaluationLogEntry> ParseTreeEvaluationLog
Combines a linear series of `ParseTreeEvaluationLogEntry` records into a continuous `ParseTreeEvaluationLog` data type suitable for storing the complete log of a high level interpretation of a parse tree.

struct ParseTreeNode
A `ParseTreeNode` instance represents a single parse tree node within the parse tree component of an IR.

> **enum Nature**
> A value indicating whether the node is a leaf node (`NODE_NATURE_LEAF_NODE`) or an interior node (`NODE_NATURE_INTERIOR_NODE`).
>
> **struct {} as_interior_node**
> This structure contains the properties which are active if the node is an interior node. This includes the `ParseTreeInteriorNodeSetMember` reference indicating what kind of interior node this is, and an array of pointers to all child `ParseTreeNode`.
>
> **struct {} as_leaf_node**
> This structure contains the properties which are active if the node is a leaf node. This includes the `ParseTreeLeafNodeSetMember` reference indicating what kind of leaf node this is.

#define EVALUATION_OF_PARSE_TREE_OPERATOR_LIKE_SYMBOL_SIGNATURE
This define provides a simple name to the shared signature used within the functions defining the functionality of the evaluation operation used during high level interpretation of a parse tree.

#define EREFERATION_OF_PARSE_TREE_OPERATOR_LIKE_SYMBOL_SIGNATURE
This define provides a simple name to the shared signature used within the functions defining the functionality of the ereferation operation used during high level interpretation of a parse tree.

enum PARSE_TREE_OPERATOR_LIKE_SYMBOL_SOURCE_SYNTAX_FORM
This enumeration defines a number of bitmask flags defining how a given parse tree node should be displayed when it is to be printed in the form of a human readable string.

> `PARSE_TREE_OPERATOR_LIKE_SYMBOL_SOURCE_SYNTAX_FORM_INVALID`,
> Error value.
> `PARSE_TREE_OPERATOR_LIKE_SYMBOL_SOURCE_SYNTAX_FORM_FLAG_NEWLINE`,
> Symbol should be followed by a newline.

PARSE_TREE_OPERATOR_LIKE_SYMBOL_SOURCE_SYNTAX_FORM_FLAG_NO_BRACKETS,

Symbol should not be surrounded by brackets.

PARSE_TREE_OPERATOR_LIKE_SYMBOL_SOURCE_SYNTAX_FORM_FLAG_NO_SPACES,

Symbol should not be surrounded by spaces.

PARSE_TREE_OPERATOR_LIKE_SYMBOL_SOURCE_SYNTAX_FORM_FIX_MASK,

Mask value used to isolate the following flags.

PARSE_TREE_OPERATOR_LIKE_SYMBOL_SOURCE_SYNTAX_FORM_PREFIX,

Symbol should precede the values upon which it acts.

PARSE_TREE_OPERATOR_LIKE_SYMBOL_SOURCE_SYNTAX_FORM_INFIX,

Symbol should be interposed between the values upon which it acts.

enum ParseTreeOperatorLikeSymbolHandle

This enumeration assigns a unique value to each of the constructs available in the high level language considered in this dissertation.

PARSE_TREE_OPERATOR_LIKE_SYMBOL_PLUS,
PARSE_TREE_OPERATOR_LIKE_SYMBOL_MINUS,
PARSE_TREE_OPERATOR_LIKE_SYMBOL_MULTIPLY,
PARSE_TREE_OPERATOR_LIKE_SYMBOL_DIVIDE,
PARSE_TREE_OPERATOR_LIKE_SYMBOL_SEMICOLON,
PARSE_TREE_OPERATOR_LIKE_SYMBOL_ASSIGNMENT,
PARSE_TREE_OPERATOR_LIKE_SYMBOL_RETURN,
PARSE_TREE_OPERATOR_LIKE_SYMBOL__END_OF_LIST, - marker value

struct ParseTreeOperatorLikeSymbolMetadata

Within the software, language features encoded as parse tree nodes are often referred as ‘operator-like symbols’. This structure holds the shared metadata associated with a given type of operator-like symbol. This includes the arity of the symbol (how many children it may have), the evaluation and ereferation functions it uses, the complexity heuristic contribution due to invocation, the symbol strings and syntax data used to display as it as a readable string.

const unsigned int PARSE_TREE_OPERATOR_LIKE_SYMBOL_COUNT

This constant uses the ParseTreeOperatorLikeSymbolHandle enumeration as a marker value for specifying static arrays in C++.

const ParseTreeOperatorLikeSymbolMetadata parse_tree_operator_like_symbol_metadata[PARSE_TREE_OPERATOR_LIKE_SYMBOL_COUNT]

This array contains all of the ParseTreeOperatorLikeSymbolMetadata data for the operator-like symbols defined.

enum ParseTreeLeafNodeNature

This enumeration defines the exact nature of a given parse tree leaf node. There is only one nature currently defined: access of a variable PARSE_TREE_LEAF_NODE_NATURE_VARNAME.

struct ParseTreeLeafNodeSetMember

This structure defines a 'prototype' leaf node containing the properties common to all instances of a given leaf node. For example, all nodes referencing the variable 'a' are linked to a single `ParseTreeLeafNodeSetMember` defining this variable.

enum ParseTreeInteriorNodeNature

This enumeration defines the exact nature of a given parse tree interior node. There is only one nature currently defined: the operator-like symbol `PARSE_TREE_INTERIOR_NODE_NATURE_OPERATOR_LIKE_SYMBOL`.

struct ParseTreeInteriorNodeSetMember

This structure defines a 'prototype' interior node containing the properties common to all instances of a given interior node. For example, all addition nodes are linked to a single `ParseTreeInteriorNodeSetMember` defining addition.

struct ParseTreeSourceProgram

This structure represents the complete definition of an input IR. This structure consists of maps containing the `ParseTreeInteriorNodeSetMember` and `ParseTreeLeafNodeSetMember` lists defining the available nodes from which the parse tree may be constructed, a pointer to the `ParseTreeNode` at the root of the parse tree and a pointer to the `SymbolTable` holding the symbol table part of the IR.

enum INPUT_SOURCE_PROGRAM_SYMBOLIC_VARIABLE_NATURE

This enum defines values used to designate the symbol table type of variable read in from a plain text input source program file.

`INPUT_SOURCE_PROGRAM_SYMBOLIC_VARIABLE_NATURE_NO_NATURE`,
Invalid.

`INPUT_SOURCE_PROGRAM_SYMBOLIC_VARIABLE_NATURE_INPUT`,
This variable is only used as input to the program.

`INPUT_SOURCE_PROGRAM_SYMBOLIC_VARIABLE_NATURE_OUTPUT`,
This variable is only used as output.

`INPUT_SOURCE_PROGRAM_SYMBOLIC_VARIABLE_NATURE_INPUT_OUTPUT`,
This variable is both input and output.

`INPUT_SOURCE_PROGRAM_SYMBOLIC_VARIABLE_NATURE_CONSTANT`,
The value of this variable is constant during execution.

`INPUT_SOURCE_PROGRAM_SYMBOLIC_VARIABLE_NATURE_SYMBOLIC_CONSTANT`,
This variable is a symbolic constant with the same value as its symbolic name.

`INPUT_SOURCE_PROGRAM_SYMBOLIC_VARIABLE_NATURE_INTERMEDIATE`,
This variable is used to hold intermediate values.

struct InputSourceProgramSymbolicVariableData

This structure holds a record of the symbol table defined while the plain text input program is being read. It contains the symbolic name of the variable and the enum `INPUT_SOURCE_PROGRAM_SYMBOLIC_VARIABLE_NATURE` of the variable.

struct ParseTreeCreationStatus

This structure is used to hold the status of the parse tree as it is constructed during parsing of the plain text input program source file.

unsigned int INSTRUCTION_EXECUTIONS_MAX_COUNT

This variable defines the maximum number of low level instructions which may be executed during evaluation of a fitness case. It is set when the parameters files are read.

struct IndexSeries

An `IndexSeries` instance holds a series of unique unsigned integers within a defined range. It is used to select candidate programs from the population during tournament selection. A number of helper functions are defined to allow the user to generate `IndexSeries` instances of any size from any population size.

typedef std::vector<VirtualMachineInstructionExecutionRecord> VirtualMachineExecutionLog

A `VirtualMachineExecutionLog` holds a complete record of the low level execution of a candidate program instruction string. It is composed of a linear series of `VirtualMachineInstructionExecutionRecord` instances.

#define VM_INSTRUCTION_EXECUTION_SIGNATURE

This define provides a simple name to the shared signature used within the functions defining the functionality of the low level instructions used in the low level virtual machine.

const unsigned int VM_REGISTER_COUNT

This constant defines the size of the register file, including program counter and stack pointer, which is defined as a static array in many places throughout the code.

const unsigned int VM_REGISTER_PROGRAM_COUNTER

The program counter is a register with an index two greater than that of the maximum general purpose register. This constant defines the index of the program counter register.

const unsigned int VM_REGISTER_STACK_POINTER

The stack pointer is a register with an index one greater than that of the maximum general purpose register. This constant defines the index of the stack pointer register. The stack pointer is not used in any of the experiments in this dissertation.

unsigned int vm_register_general_purpose_count
unsigned int vm_register_final_general_purpose
These variables hold the number of general purpose registers available to low level programs, and the index of the final general purpose register in the register file. These variables are initialised at the time of reading the parameter files.

const unsigned int VM_INDEXED_MEMORY_TOTAL_SIZE
const unsigned int VM_INDEXED_MEMORY_STACK_SECTION_SIZE
const int VM_INDEXED_MEMORY_STACK_SECTION_INDEX_HIGHEST
const int VM_INDEXED_MEMORY_STACK_SECTION_INDEX_LOWEST
const unsigned int VM_INDEXED_MEMORY_FREE_SECTION_SIZE
These variables define the size and parameters of the stack section when indexed memory is used. These variables are not used in the experiments described in this dissertation.

std::vector<std::string> VM_REGISTER_MNEMONIC
This vector holds the readable mnemonic names of each register in the file (such as the 'PC – program counter').

typedef BigInteger VM_TYPE
`VM_TYPE` is the value type used throughout the low level virtual machine.

struct VirtualMachine
A `VirtualMachine` instance holds the persistent state of a low level virtual machine during execution of a candidate program. It holds the state of the register file, the state of the variable memory, the error state and the total number of instructions executed so far.

enum VM_STATE_BAD_REASON
This enumeration assigns a constant to each of the possible error states of the virtual machine.

```
VM_STATE_BAD_REASON_ZERO,           - no bad state
VM_STATE_BAD_REASON_HALT_INSTRUCTION_ENCOUNTERED,
VM_STATE_BAD_REASON_INSTRUCTION_OUT_OF_RANGE,
VM_STATE_BAD_REASON_INSTRUCTION_UNUSUAL_INPUT,
VM_STATE_BAD_REASON_STACK_OVERFLOW,
VM_STATE_BAD_REASON_MEMORY_ACCESS_OUT_OF_BOUNDS,
VM_STATE_BAD_REASON_PROGRAM_COUNTER_OUT_OF_RANGE,
VM_STATE_BAD_REASON_DIVISION_BY_ZERO,
VM_STATE_BAD_REASON_INSTRUCTION_LIMIT_REACHED,
VM_STATE_BAD_REASON_UNIDENTIFIED_ERROR_STATE,
VM_STATE_BAD_REASON__END_OF_LIST,   - marker value for arrays
```

const unsigned int VM_STATE_BAD_REASON_COUNT
This constant uses the marker value from `enum VM_STATE_BAD_REASON` to specify the size of static arrays.

const char *VM_STATE_BAD_REASONS_STRINGS[VM_STATE_BAD_REASON_COUNT]
This array contains a printable human-readable string version of each bad state indicated by `enum VM_STATE_BAD_REASON`.

enum VM_INSTRUCTION_OPERATION
This enumeration assigns a unique constant to each low level language operation implemented in the virtual machine interpreter. These values are used whenever a reference to a low level operation is needed:

```
VMI_A_ADD,
VMI_A_SUB,
VMI_A_MUL,
VMI_A_DIVP, (virtual machine instruction, arithmetic)

VMI_M_LOADS,(virtual machine instruction, memory)
VMI_M_STORS,

VMI_V_LOADV,(virtual machine instruction, value)
```

const unsigned int VM_INSTRUCTION_OPERATION_COUNT
This constant uses the marker value from `enum VM_INSTRUCTION_OPERATION` to specify the size of static arrays.

const unsigned int INSTRUCTION_OPERANDS_MAX_COUNT
This constant defines the maximum number of operands an instruction in the low level language can take. This value is used to specify the value of static arrays.

struct InstructionOperand
An `InstructionOperand` instance holds one operand component of a complete low level `Instruction`. It has the ability to represent a register index, symbol table key or direct signed integer value depending on the containing `Instruction`.

struct Instruction
An `Instruction` instance is a complete assembly language instruction expressed in the low level language. It consists of an operation part defined by a `VM_INSTRUCTION_OPERATION` and up to `INSTRUCTION_OPERANDS_MAX_COUNT` operands of type `InstructionOperand`.

typedef std::list<Instruction> InstructionString;
A string of instructions in the low level language is stored as a `std::list` of `Instruction` instances.

struct VMInstructionOperationMetadata
This structure holds the shared metadata associated with a given defined operation in the low level language. This includes the readable mnemonic name of the operation, a pointer to the evaluation function it uses, and the number of operands it takes.

struct VirtualMachineInstructionExecutionRecordLocation
Within a `VirtualMachineInstructionExecutionRecord` instance, a value may be read from a register file location or a variable in memory. This structure may hold information on either, and its interpretation is dependent on the operation used to perform the operation.

struct VirtualMachineInstructionExecutionRecord
An instance of this structure holds a full record of the execution of a single low level instruction in the low level virtual machine. It contains the operation performed, the result produced and its location, the values of the operands used to calculate the result and their location, and the values contained in the register file both before and after the execution of the instruction.

#define DIVISION_BY_ZERO_IS_ERROR
This define may be used to configure the low level interpreter to either use protected division when evaluating the `DIV` instruction, or to abort execution with an error state. This dissertation considers the case where protected division is used.

enum JUMP_PREDICATE
This enumeration is not used.

VMInstructionOperationMetadata vm_instruction_operation_metadata [VM_INSTRUCTION_OPERATION_COUNT]
This array holds all the metadata for the operations implemented by the low level language interpreter.

enum VM_EXECUTE_INTERACTIVE_FLAG
This flag is used to indicate to the virtual machine interpreter whether or not it should active. It can take the values `VM_EXECUTE_NONINTERACTIVE` or `VM_EXECUTE_INTERACTIVE`.

struct InstructionProbability
This structure combines a `VM_INSTRUCTION_OPERATION` with a rate of inclusion. It also contains a member indicating the cumulative probability upper bound as a convenience (used in the selection algorithm).

typedef std::vector<InstructionProbability> InstructionSetProbabilistic
Multiple `InstructionSetProbability` are combined in a `std::vector` to produce a single `InstructionSetProbabilistic` containing a list of all the operations available to the system to select from together with their rate of selection (as a probability).

struct GeneticLinearProgram
Within the LGP system, a candidate program represented by a `GeneticLinearProgram` instance is composed of an `InstructionString` holding

the candidate program itself, a floating point value indicating the fitness of the program and an unsigned integer holding the number of hits.

struct FitnessCase

A fitness is represented in the system by an instance of the `FitnessCase` structure. An instance of `FitnessCase` contains the starting and ending values of all the variables in the symbol table, the return value of the root node of the parse tree (not used) and a `ParseTreeEvaluationLog` indicating the model evaluation produced by high level interpretation of the parse tree.

struct VirtualMachineExecutionRegisterStatus

An instance of this structure holds the state of a single register at some point within a log of virtual machine execution.

struct VirtualMachineExecutionRegisterStatusRecord

This structure combines a number of `VirtualMachineExecutionRegisterStatus` instances to produce a full record of the state of the register file at some point within a log of virtual machine execution.

typedef std::vector<VirtualMachineExecutionRegisterStatusRecord> VirtualMachineExecutionRegisterStatusTimeline

This data type holds a complete record of the state of the registers between the execution of every low level instruction by combining a series of `VirtualMachineExecutionRegisterStatusRecord` in a `std::vector`.

struct GeneticLinearProgram_Crossover_Pair

This structure is used as a convenient method for transferring the two `GeneticLinearProgram` instances produced as a result of the `GeneticLinearProgram_Crossover2` crossover implementation function.

typedef std::list<GeneticLinearProgram **> SortedGeneticLinearProgramPopulation

An additional layer of pointers to the `GeneticLinearProgram` instances forming the population candidate programs is stored in this `std::list` based data type. The use of this list structure allows for fast insertion and deletion of programs from the list: it is assumed that the majority of programs produced as a result of the recombination operators will be significantly worse fitness than those already present. Therefore, this structure is biased to insert new programs from the 'worse' side, requiring only a small number of fitness comparisons to determine the position of this new program in the sorted population. This comes at the cost of many fitness value comparisons if the fitness of the new program is 'good' rather than 'bad'.

struct EvolutionSystem_Parameters

This structure is used to hold all of the evolutionary system parameters. The layout of this structure is designed to mirror the plain text file format used to enter the parameters.

struct EvolutionSystem_Report

This structure is used to hold the result of a single invocation of the evolutionary system to evolve a program or program fragment. It holds statistics such as the total number of candidate programs considered, the total number of niche pre-emption runs required and the total number of low level candidate program instructions considered.

E.2 Function and Macro Reference

```
 std::string bigIntegerToString(const BigInteger &b)
         int bigIntegerToInt   (const BigInteger &b)
unsigned int bigIntegerToUInt  (const BigInteger &b)
```

Converts the given `BigInteger` into a `std::string`, signed integer or unsigned integer respectively. The implementation of these functions differs depending on the underlying data type of `BigInteger`. If the conversion results in truncation, only the least significant bits are returned.

#define wait_for_return()

Prints a prompt to the console window and waits for the user to press Return.

static inline float random_float()

Returns a random float value from 0 to <1.

static inline float random_float(float lower, float upper)

Returns a random float value `lower` <= x <= `upper`.

static inline unsigned int random_int_0_to_(unsigned int max)

Returns a random unsigned integer in the range 0 <= x <= `max`.

static inline unsigned int random_int_1_to_(unsigned int max)

Returns a random unsigned integer in the range 1 <= x <= `max`.

static inline int random_int(int lower, int upper)

Returns a random signed integer in the range `lower` <= x <= `upper`.

template <typename T>
static inline const T &max(const T &a, const T &b)

Returns a reference (const) to the greater of a and b by `operator>()`.

template <typename T>
static inline const T &max(const T &a, const T &b)

Returns a reference (const) to the lesser of a and b by `operator<()`.

bool BoundsPair::Within(const T &value)

Returns `true` if `value` is within the bounds given by this instance of `BoundsPair`, else `false`.

```
template <typename T>
static inline std::string String_ToString(const T& t)
```
Returns `t` in the form of a `std::string`.

```
static inline bool char_in_str(const char *s, char c)
```
Returns `true` if the character `c` is present in the null terminated string `s`.

```
std::string String_LoseTrailing(const std::string &string,
                                const char *characters)
```
Return a duplicate of `string` with all characters from null terminated string `characters` removed from the end of the string. E.g. If `characters = "\r\n "`, then this function returns a copy of `string` with carriage returns, newlines and space characters removed from the end.

```
static inline std::string String_LoseTrailingNewlines
                                    (const std::string &string)
static inline std::string String_LoseTrailingSpaces
                                    (const std::string &string)
```
Return a copy of `string` with trailing newlines and spaces removed respectively.

```
std::string String_ReturnWord(const std::string &s,
                              unsigned int word,
                              const char *delimiters = " ")
```
Return the word in zero-indexed position `word` from the left of string `s` using the characters within the null-terminated string `delimiters` as word delimiters. For example `String_ReturnWord(std::string("Jones the cat"), 1, " ")` returns a `std::string` containing `"the"`.

```
std::string String_ReturnWordRemainder(const std::string &s,
                                       unsigned int word,
                                       const char *delimiters = " ")
```
Return the remainder of the string `s` after removing all characters before the word in zero-indexed position `word` from the left of string `s` using the characters within the null-terminated string `delimiters` as word delimiters. For example `String_ReturnWordRemainder(std::string("Jones the cat"), 1, " ")` returns a `std::string` containing `" cat"`.

```
void SymbolicVariableData_ApplyArchetype
      (SymbolicVariableData *symbolic_variable_data,
       SYMBOL_TABLE_VARIABLE_ARCHETYPE archetype)
```
Applies the `SYMBOL_TABLE_VARIABLE_ARCHETYPE archetype` to the `SymbolicVariableData` pointed to by `symbolic_variable_data` to establish its properties, such as read-only status, constant nature, etc.

```
SYMBOL_TABLE_KEY SymbolTable_InsertSymbolTableEntry(
                        SymbolTable *symbol_table,
                    const std::string &variable_symbolic_name,
     SYMBOL_TABLE_VARIABLE_ARCHETYPE  archetype,
                         BigInteger  value = 0)
```

Creates a new symbol in the `SymbolTable` instance `symbol_table` with symbolic name `variable_symbolic_name` with the archetype `archetype`. If this is symbolic constant, it has value `value`.

```
SymbolicVariableData *SymbolTable_QuerySymbolTableKey
    (SymbolTable *_symbol_table,
     const SYMBOL_TABLE_KEY _key)
```

Returns a pointer to the `SymbolicVariableData` containing information about the symbol indicated by the `SYMBOL_TABLE_KEY _key`.

```
SYMBOL_TABLE_KEY SymbolTable_ReverseLookupKeyFromVarname
      (SymbolTable *_symbol_table,
       const char *_varname,
       bool  _allow_return_bad)
```

Return the `SYMBOL_TABLE_KEY` that can be used to access the `SymbolicVariableData` relating to the symbol with the symbolic name `_varname`.

```
SymbolicVariableData *SymbolTable_QuerySymbolTableVarname
            (SymbolTable *_symbol_table,
             const char *_varname)
```

Return a pointer to the `SymbolicVariableData` containing information about the symbol with symbolic name `_varname`.

```
void SymbolTable_ReaffirmContainingTypes(SymbolTable *symbol_table)
```

Updates the `contains_any_readable_variables` and `contains_any_writable_variables` members of the `SymbolTable` after alteration or insertion of variables.

```
SYMBOL_TABLE_KEY SymbolTable_RandomSymbolTableKey
       (SymbolTable *symbol_table, RANDOM_SYMBOL_TABLE_KEY_NATURE nature)
```

Return a random `SYMBOL_TABLE_KEY` from `symbol_table` of nature `nature` as defined by `RANDOM_SYMBOL_TABLE_KEY_NATURE`.

```
void SymbolTable_Print(SymbolTable *symbol_table)
```

Print a human readable version of the symbol table `symbol_table` to standard output.

```
void ParseTreeEvaluationLogEntry_Print
        (const ParseTreeEvaluationLogEntry *entry)
```

Print a `ParseTreeEvaluationLogEntry` in human readable form to standard output.

```
ParseTreeEvaluationResult parse_tree_node_evaluate__plus
         (EVALUATION_OF_PARSE_TREE_OPERATOR_LIKE_SYMBOL_SIGNATURE)
```

Performs high level interpreter evaluation of the given node interpreting it as an addition node. In general `parse_tree_node_evaluate__*` functions supply the

implementation of the evaluation capability of the interpreter for each possible type of input node.

ParseTreeEvaluationResult parse_tree_node_evaluate__return (EREFERATION_OF_PARSE_TREE_OPERATOR_LIKE_SYMBOL_SIGNATURE)

Performs high level interpreter ereferation of the given node interpreting it as a variable node. The variable node is the only parse tree node which can be ereferated currently, and it returns the `SYMBOL_TABLE_KEY` associated with the variable it represents.

ParseTreeEreferationResult parse_tree_node_ereferate__invalid (EREFERATION_OF_PARSE_TREE_OPERATOR_LIKE_SYMBOL_SIGNATURE)

Returns an error indicating that the given type of node cannot be the subject of ereferation. This is used to prevent the user from supplying a program in the form `(2+3) = a`, where the attempt to ereferate (interpret as a reference) the addition node is meaningless in the high level language as defined.

ParseTreeLeafNodeSetMember *ParseTreeLeafNodeSetMember_NewParseTreeLeafNode SetMember_Internal_NewParseTreeLeafNodeSetMember()

Allocates memory for a new `ParseTreeLeafNodeSetMember` and returns a pointer to the new instance. A `ParseTreeLeafNodeSet` is a set of `ParseTreeLeafNodeSet Members`. These are the primitives that one can build a parse tree out of. You have to define all these PTLNSMs first, and then use an 'instantiation' function upon them to get the actual nodes for inclusion in the parse tree.

ParseTreeLeafNodeSetMember *ParseTreeLeafNodeSetMember_NewParseTreeLeaf NodeSetMember_VariableFromSymbolTableKey(SYMBOL_TABLE_KEY varkey)

Allocate a new `ParseTreeLeafNodeSetMember` that, when invoked, can create a `ParseTreeNode` of the ‘variable’ variety using the variable associated with `varkey`.

ParseTreeInteriorNodeSetMember *ParseTreeInteriorNodeSetMember_NewParseTree InteriorNodeSetMember_Internal_NewParseTreeInteriorNodeSetMember()

Allocates memory for a new `ParseTreeInteriorNodeSetMember` and returns a pointer to the new instance. A `ParseTreeInteriorNodeSet` is a set of `ParseTreeInteriorNodeSetMembers`. These are the primitives that one can build a parse tree out of. You have to define all these PTINSMs first, and then use an 'instantiation' function upon them to get the actual nodes for inclusion in the parse tree.

ParseTreeInteriorNodeSetMember *ParseTreeInteriorNodeSetMember_NewParseTree InteriorNodeSetMember_OperatorLikeSymbol (ParseTreeOperatorLikeSymbolHandle operator_like_symbol)

Allocate a new `ParseTreeInteriorNodeSetMember` that, when invoked can create a `ParseTreeNode` of the operator like symbol variety designated by the `ParseTreeOperatorLikeSymbolHandle operator_like_symbol`.

unsigned int ParseTreeNode_OperatorLikeSymbol_ReturnArity
(const ParseTreeNode *node)

Return the arity of the language feature reflected by the `ParseTreeNode` `node`.

void ParseTreeEvaluationEnvironment_Initialise
(ParseTreeEvaluationEnvironment *environment, SymbolTable *symbol_table)

Set a `ParseTreeEvaluationEnvironment` to default values before execution. Copy the values of the symbolic constants from the symbol table to their respective places in the map.

ParseTreeEvaluationResult ParseTreeNode_Evaluate
(EVALUATION_OF_PARSE_TREE_OPERATOR_LIKE_SYMBOL_SIGNATURE)

Evaluates the program given by a given `ParseTreeNode` in the specified `ParseTreeEvaluationEnvironment`. Returns a `ParseTreeEvaluationResult`. This function is designed to operate recursively; if the indicated node is an interior node, the child nodes are evaluated by means of this function and then used to calculate the value of this node. Throughout execution of this function, a log of all evaluations performed is kept.

ParseTreeEreferationResult ParseTreeNode_Ereferate
(EREFERATION_OF_PARSE_TREE_OPERATOR_LIKE_SYMBOL_SIGNATURE)

Ereferates the program fragment given by a given `ParseTreeNode` in the specified `ParseTreeEvaluationEnvironment`. Returns a `ParseTreeEreferationResult`. This function is designed to operate recursively; if the indicated node is an interior node, it is possible for some kind of indirect referencing to occur (i.e. a calculation is performed, which results in the value returned by this function). This is not attempted in this dissertation.

void ParseTreeNode_Print_AsReadableExpression
(const ParseTreeNode *node, SymbolTable *symbol_table)

Prints a human readable expression of the input `ParseTreeNode` tree to standard output using the syntax rules defined in the operator-like symbol metadata. The output is highly similar to high level source code and may contain newlines.

std::string ParseTreeNode_Print_ToString
(const ParseTreeNode *node, SymbolTable *symbol_table)

Return a `std::string` human readable expression of the input `ParseTreeNode` tree using the syntax rules defined in the operator-like symbol metadata. The output is highly similar to high level source code and will not contain newlines.

void SwitchNodes(ParseTreeNode **a, ParseTreeNode **b)

Switches the `ParseTreeNode` instances pointed to by the pointers pointed to by a and b in place.

ParseTreeNode *ParseTreeNode_Internal_NewNode()

Allocates and returns a new `ParseTreeNode`. Used internally during the construction and initialisation of `ParseTreeNode` instances.

void ParseTreeNode_Internal_FreeParseTreeNode(ParseTreeNode *node)
Free the memory used to store a `ParseTreeNode`. Used internally to manage `ParseTreeNode` instances.

void ParseTreeNode_DestroyTree(ParseTreeNode *n)
Free all the memory allocated to the full `ParseTreeNode` tree contained within node `n`. This only works if the tree doesn't contain any undefined children. If the tree contains pointers that point to bad nodes the program will crash.

unsigned int ParseTreeNode_GetHighestDepth(const ParseTreeNode *node)
Returns the greatest depth of a `ParseTreeNode` tree. A leaf node alone counts as 1.

unsigned int ParseTreeNode_GetTotalNumberOfNodesInTree
(const ParseTreeNode *node)
Return the total number of nodes in the given `ParseTreeNode` tree, including the root.

ParseTreeNode *ParseTreeNode_DuplicateNode(const ParseTreeNode *node)
Returns a shallow copy of the `ParseTreeNode` pointed to by `node`. This duplicate will have invalid child pointers if it is an interior node and it is used in its default state, as the original and the duplicate will point to the same children.

ParseTreeNode *ParseTreeNode_DuplicateTree(const ParseTreeNode *node)
Returns a deep copy of the `ParseTreeNode` tree pointed to by node.

ParseTreeNode *ParseTreeNode_NewLeafNodeFromParseTreeLeafNodeSetMember
(const ParseTreeLeafNodeSetMember *leaf_node_master)
Instantiate a new `ParseTreeNode` given the 'master node' `ParseTreeLeafNodeSetMember leaf_node_master`. This is the method used to create new leaf tree nodes from the ‘prototype’ set.

ParseTreeNode *ParseTreeNode_NewIncompleteInteriorNodeFromParseTreeInterior
NodeSetMember(const ParseTreeInteriorNodeSetMember *interior_node_master)
Instantiate a new `ParseTreeNode` given the 'master node' `ParseTreeInterior NodeSetMember interior_node_master`. This is the method used to create new interior tree nodes from the ‘prototype’ set. The child pointers will be invalid and will require attention.

void ParseTreeSourceProgram_DestroySourceProgram
(ParseTreeSourceProgram *parse_tree_source_program)
Destroys an entire `ParseTreeSourceProgram` together with its symbol table, parse tree program, and all of the ‘prototype’ master node sets.

std::string String_RemoveOuterBracketsFromString(const std::string &input)
Returns a copy of the `std::string input` with all of the matched curved bracket pairs appearing on the start and end of the string removed.

```
std::string String_PrefixifySourceCodeString(const std::string &input)
```
Transforms a high level program string in infix form as read from a plain text input file into a string in prefix form using the priorities map to define precedence.

```
std::string String_RemoveWhitespacesFromString(const std::string &input)
```
Return a copy of the `std::string` input with all whitespace, carriage return, newline and tab characters removed.

```
ParseTreeSourceProgram *ParseTreeSourceProgram_
      ConstructSourceProgramFromFile(const std::string &input_filename)
```
Creates a `ParseTreeSourceProgram` containing both symbol table and parse tree components of the input IR based on the contents of the given plain text high level source file. Returns `NULL` on error.

```
ParseTreeNode *ParseTreeCreation_CreateParseTreeNode
        (ParseTreeCreationStatus *status)
```
Create a `ParseTreeNode` from the next node in the Polish notation program string held within the `ParseTreeCreationStatus` structure. This function is designed to work recursively, performing a depth first initialisation of the parse tree by considering the Polish notation string as a stack.

```
void ParseTreeSourceProgram_CalculateOperatorLikeSymbolPopulations
(ParseTreeNode *program,
 std::map<const ParseTreeInteriorNodeSetMember *, unsigned int> &population)
```
Calculate the total number of occurrences of each `ParseTreeInteriorNode SetMember`-based interior node in the given `ParseTreeNode` program and store these values in the map population.

```
IndexSeries *IndexSeries_NewIndexSeries(unsigned int _length)
```
Allocate and return a new, blank `IndexSeries` of length `_length`.

```
void IndexSeries_DestroyIndexSeries(IndexSeries *index_series)
```
Free the `IndexSeries` pointed to by `index_series`.

```
IndexSeries *IndexSeries_ReturnTournament(unsigned int population_size,
                                          unsigned int tournament_size)
```
Given a population size and a tournament size, return `tournament_size` unique indices in the range `0 <= x < population size`. This function is suitable for generating a list of indices of programs from a population of known size for participation in tournament selection.

```
static inline unsigned int VirtualMachine_RandomRegister()
```
Returns a random register index from the full range of the register file. This function is not used, as the stack pointer and program counter are not available for manipulation by the low level programs.

static inline unsigned int VirtualMachine_RandomGeneralPurposeRegister()
Returns a random register index from the range of general purpose registers in the register file.

void VirtualMachine_InitialiseStaticMnemonics()
Populate the array containing the human readable mnemonics for the registers in the register file of the virtual machine.

void VirtualMachine_Initialise
(VirtualMachine *vm, SymbolTable *symbol_table)
Fill the `VirtualMachine` with default values; copy the starting values from the symbol table into the virtual machine.

void VirtualMachine_InitialiseFromExisting
(VirtualMachine *vm, VirtualMachine *vm_existing)
Copy the state of one `VirtualMachine` into another.

int VirtualMachine_IsValidMemoryLocation(int memory_location)
Return an `int` indicating if the given index can refer to some part of indexed memory. This function is not used in this dissertation as indexed memory is not investigated.

void VirtualMachine_Abort(VirtualMachine *vm, VM_STATE_BAD_REASON reason)
Abort execution within the given `VirtualMachine` with the error state `reason`.

const char *VirtualMachine_ExplainCurrentBadState(VirtualMachine *vm)
Return the human readable string associated with the current error state of the given `VirtualMachine` instance

Instruction *InstructionString_RetrieveInstruction
(InstructionString *is, unsigned int instruction_cell_index)
Return a pointer to the instruction in zero-indexed position `instruction_cell_index` in the `InstructionString is`.

int InstructionString_IsInstructionIndexWithinRange
(InstructionString *is, int program_counter)
Return an `int` indicating if it is safe to access the instruction in zero-indexed position `instruction_cell_index` in `InstructionString is`.

unsigned int InstructionString_ReturnHighestRegisterUsed(InstructionString *is)
Return the index of the highest virtual machine register referred into the program pointed to by `is`.

```
InstructionString *InstructionString_ConstructNewCompositeString(
```

Construct a new `InstructionString` by taking the first `start1` instructions of `str1`, then `count2` instructions from `str2` starting at `start2`, then the remaining instructions from `str1` that appear `count1` instructions after `start1`.

```
InstructionString *InstructionString_DuplicateInstructionString
       (InstructionString *is)
```

Return a duplicate of the `InstructionString is`.

```
void InstructionString_ConcatenateInstructionStringsInPlace
       (InstructionString *is1, InstructionString *is2)
```

Append a copy of the `Instruction` instances within `is2` to the end of `is1`.

```
InstructionString *InstructionString_ConcatenateInstructionStrings
       (InstructionString *is1, InstructionString *is2)
```

Return a new `InstructionString` containing the instructions from `is1` followed by the instructions from `is2`.

```
void InstructionString_AppendInstruction
       (InstructionString *str, Instruction *i)
```

Append the `Instruction i` to the end of `InstructionString str`.

```
void InstructionString_InsertInstruction
   (InstructionString *str, unsigned int insertion_position, Instruction *i)
```

Insert the `Instruction i` before the instruction at index `insertion_position` in `InstructionString str`.

```
void TreeWalkingCompiler_BasicCompileParseTreeSourceProgram
       (ParseTreeSourceProgram *program,
                 ParseTreeNode *node,
                  unsigned int  register_destination,
              InstructionString *output_string)
```

Use the tree-walking compiler algorithm to compile the program expressed by the `ParseTreeSourceProgram program` and `ParseTreeNode node`, using the register index `register_destination` as the basis for calculation, appending the produced output program to the `InstructionString` pointed to `output_string`. This function is designed to act recursively as described in the body of the dissertation.

```
void VirtualMachineInstructionExecutionRecord_Print
    (VirtualMachineInstructionExecutionRecord *r, SymbolTable *symbol_table)
```

Print a `VirtualMachineInstructionExecutionRecord` in human readable form to standard output.

```
#define assert_register_good(_r_)
```

Insert an assert ensuring that register index `_r_` is in the range of valid register file indices.

#define assert_symbol_good(_s_)

Insert an assert ensuring that the SYMBOL_TABLE_KEY _s_ is in the range of valid symbol table keys.

void vm_instruction_implementation_vmi_a_add
(VM_INSTRUCTION_EXECUTION_SIGNATURE)

Interpret the given Instruction as an ADD instruction and perform it within the given VirtualMachine environment, logging the calculation to the VirtualMachine InstructionExecutionRecord. In general vm_instruction_implementation_* functions supply the base implementation of each low level operation for each instruction defined in the low level language.

int VirtualMachineInstruction_IsValidInstruction(Instruction *i)

Return an int indicating if the operation part of the given Instruction is valid; it may have been tampered with and invalidated by a recombination operation (or program bug).

void VirtualMachineInstruction_ExecuteInstruction
(VM_INSTRUCTION_EXECUTION_SIGNATURE)

Execute the given instruction in the given VirtualMachine. This function extracts the VM_INSTRUCTION_OPERATION for the given instruction and executes the appropriate base implementation function (described above) for that operation.

void VirtualMachine_PrintRegisters(VirtualMachine *vm)

Print the contents of all registers (with mnemonic) in the given VirtualMachine state to standard output.

```
void VirtualMachineInstructionString_Advance(          VirtualMachine *vm,
                                                    InstructionString *is,
                                             VirtualMachineExecutionLog *log)
```

Advance the program counter register in the given VirtualMachine and execute the next instruction in the InstructionString, recording the execution in the log log.

```
void VirtualMachineInstructionString_ExecuteInstructionString
       (              VirtualMachine *vm,
                   InstructionString *is,
          VirtualMachineExecutionLog *log,
                        unsigned int  instruction_limit,
         VM_EXECUTE_INTERACTIVE_FLAG  interactive)
```

Executes the given InstructionString in the given VirtualMachine state, recording all executions in the log log. A maximum of instruction_limit instructions will be executed before execution is terminated with error state VM_BAD_REASON_INSTRUCTION_LIMIT_REACHED. If interactive is VM_EXECUTE_INTERACTIVE, then the execution will pause after every instruction execution.

```
void VirtualMachineInstruction_PrintSemantics
       (Instruction *i, SymbolTable *symbol_table)
```

Print a human readable version of the semantics of the given `Instruction` to standard output.

```
void Instruction_Print
       (Instruction *i, SymbolTable *symbol_table, bool with_semantics)
```

Print an assembly-like representation of the given `Instruction` to standard output.

```
void InstructionString_Print (InstructionString *is,
                                    SymbolTable *symbol_table,
                                           bool with_semantics = false)
```

Print an assembly like representation of a complete `InstructionString` to standard output. If `with_semantics` is `true` then a second column of human readable semantics are printed alongside.

```
void InstructionSetProbabilistic_NormaliseAndCalculateCumulatives
       (InstructionSetProbabilistic *isp)
```

This function normalises and calculates the cumulative probabilities of all the `VM_INSTRUCTION_OPERATION`s in the `InstructionSetProbabilistic`. After this, a random number in the range `0.0f-1.0f` can be used to select a `VM_INSTRUCTION_OPERATION` from the set by comparing against cumulative probability value in turn.

After the execution of this function, the `InstructionSetProbabilistic` will have the following values. The second column of values show the cumulative probability of each instruction. The cumulative probability of each instruction and the previous instruction provide a set of bounds for selecting this instruction with a random value in the range `0.0f-1.0f`.

```
Instruction Set:
ADD       , 0.143 (<= 0.143)
SUB       , 0.143 (<= 0.286)
MUL       , 0.143 (<= 0.429)
DIVP      , 0.143 (<= 0.571)
LOADV     , 0.143 (<= 0.714)
LOADS     , 0.143 (<= 0.857)
STORS     , 0.143 (<= 1.000)
```

```
void Instruction_FurnishInstruction(Instruction *instruction,
                                    SymbolTable *symbol_table,
                                          int operand_no = -1)
```

This function initialises some or all of the given `Instruction` based on the value of `operand_no`. If `operand_no` is less than zero, we fill in the instruction completely. If `operand_no` is non zero, we fill in the appropriate operand. If `operand_no` is out of range of the number of operands for the current operation an assertion failure occurs.

Where a component of the instruction is selected for initialisation, that part of the instruction is selected at random from the range of possible appropriate values. A STORS instruction contains a single variable reference operand referencing a variable that can be written to, it is not possible to write to a read-only variable so this instruction will never be produced as a result of this function, etc.

```
VM_INSTRUCTION_OPERATION InstructionSetProbabilistic_RandomOperation
          (InstructionSetProbabilistic *i)
```

Returns a random VM_INSTRUCTION_OPERATION from a InstructionSetProbabilistic normalised by InstructionSetProbabilistic_NormaliseAndCalculateCumulatives.

```
InstructionString *InstructionString_NewRandomInstructionString
       FromInstructionSetProbabilistic(InstructionSetProbabilistic *i,
                                       SymbolTable *symbol_table,
                                       unsigned int length_in_instructions)
```

Return an InstructionString consisting of length_in_instructions random instructions using the probabilities stored in the InstructionSetProbabilistic and the given SymbolTable.

```
void ParseTreeEvaluationLog_InsertInitialisationSteps
    (std::map<SYMBOL_TABLE_KEY, VM_TYPE> *starting_values,
     SymbolTable *symbol_table,
     ParseTreeEvaluationLog *log)
```

Insert records into the ParseTreeEvaluationLog reflecting the automatic initialisation of SymbolTable entries with known values such as constants and input variables.

```
void FitnessCase_InitialiseCases(
      ParseTreeNode *target_program,
        SymbolTable *symbol_table,
        FitnessCase *fitness_case_array,
       unsigned int  fitness_case_count)
```

Construct fitness_case_count fitness cases reflecting the behaviour of the target program specified by target_program and symbol_table and place these into fitness_case_array.

```
void VirtualMachineExecutionRegisterStatusTimeline_
                  PopulateFromVirtualMachineExecutionLog(
        VirtualMachineExecutionRegisterStatusTimeline *timeline,
                        VirtualMachineExecutionLog *log)
```

Interpret a VirtualMachineExecutionLog and populate a VirtualMachine ExecutionRegisterStatusLine containing the status of each register before and after each instruction.

```
void VirtualMachine_FitnessCase_PopulateStartingState
       (VirtualMachine *v, SymbolTable *symbol_table, FitnessCase *c)
```

Initialise a VirtualMachine by copying the starting state of the given FitnessCase.

```
bool ParseTreeEvaluationLogEntry_InstructionCorrelates
    (ParseTreeEvaluationLogEntry *le,
     VirtualMachineInstructionExecutionRecord *vi)
```

Inspect a `ParseTreeEvaluationLogEntry` / `VirtualMachineInstruction ExecutionRecord` pair and return true if they reflect the same (or similar) instructions. This function is not used in this dissertation for reasons described in the main body.

```
int qsort_GeneticLinearProgramByFitness(const void *a, const void *b)
```

C language `qsort` compatible function for sorting an array of `GeneticLinearProgram`.

```
GeneticLinearProgram *GeneticLinearProgram_Internal_NewProgram()
```

Allocates and returns a new `GeneticLinearProgram`. This function is used internally during program construction.

```
void GeneticLinearProgram_Internal_FreeProgram(GeneticLinearProgram *node)
```

Free the memory used for storing a `GeneticLinearProgram`. This function is used internally during program construction.

```
GeneticLinearProgram *GeneticLinearProgram_NewRandomProgram
  (InstructionSetProbabilistic *i,
   SymbolTable *symbol_table,
   unsigned int  length_in_instructions)
```

Create a new random `GeneticLinearProgram` containing an `InstructionString` consisting of `length_in_instructions` random instructions using the probabilities stored in the `InstructionSetProbabilistic` and the given `SymbolTable`.

```
GeneticLinearProgram *GeneticLinearProgram_DuplicateProgram
            (GeneticLinearProgram *program)
```

Returns a duplicate of the given `GeneticLinearProgram`.

```
GeneticLinearProgram_Crossover_Pair GeneticLinearProgram_Crossover2
          (GeneticLinearProgram *p0,
           GeneticLinearProgram *p1,
           unsigned int maximum_length_in_instructions)
```

Produce two child programs simultaneously by executing the crossover recombination operation on the two given parent programs. If the length of either program produced by crossover exceeds `maximum_length_in_instructions`, the transition points are reselected until this is no longer the case.

```
void GeneticLinearProgram_DestroyProgram(GeneticLinearProgram *p)
```

Free all the memory allocated to the `GeneticLinearProgram` and its attributes.

```
GeneticLinearProgram *GeneticLinearProgram_MutateProgram
          _AlterRandomInstruction(GeneticLinearProgram *program,
                                  InstructionSetProbabilistic *is,
                                  SymbolTable *symbol_table)
```

Mutate the given `GeneticLinearProgram` program by altering a random instruction. New VM_`INSTRUCTION_OPERATION` components will be selected according to the given `InstructionSetProbabilistic` and `SymbolTable`.

```
GeneticLinearProgram *GeneticLinearProgram_MutateProgram
         _DeleteRandomInstruction(GeneticLinearProgram *program)
```

Mutate the given `GeneticLinearProgram` program by deleting a random instruction.

```
GeneticLinearProgram *GeneticLinearProgram_MutateProgram
      _InsertRandomInstruction(GeneticLinearProgram *program,
                               SymbolTable *symbol_table,
                               InstructionSetProbabilistic *instruction_set)
```

Mutate the given `GeneticLinearProgram` program by inserting a random instruction.

```
GeneticLinearProgram *GeneticLinearProgram_MutateProgram
                    (GeneticLinearProgram *program,
                     InstructionSetProbabilistic *is,
                     SymbolTable *symbol_table)
```

Mutate a `GeneticLinearProgram` program by performing an operation chosen from the previously defined operations of deletion, insertion and alteration randomly.

```
unsigned int SortedGeneticLinearProgramPopulation_Insert
     (SortedGeneticLinearProgramPopulation &a, GeneticLinearProgram **p)
```

Insert a new program into the `SortedGeneticLinearProgramPopulation`, retaining the sorted property. This has linear complexity, but assumes that most programs have very high fitness, so the actual cost will be minimal as their position will be near the start of the insertion position search.

```
SortedGeneticLinearProgramPopulation::iterator
SortedGeneticLinearProgramPopulation_At
         (SortedGeneticLinearProgramPopulation &a, unsigned int l)
```

Return an `iterator` to the program at index `l` in the `SortedGeneticLinearProgramPopulation`. Linear complexity. Searches iteratively from the closest edge.

```
EvolutionSystem_Parameters EvolutionSystem_Parameters_LoadFromFile
         (const std::string &file)
```

Load the plain text parameter file from the filename `file` into memory and return an instance of `EvolutionSystem_Parameters` populated with its values.

```
void GeneticLinearProgram_FitnessEvaluationOnTrainingSet(
            GeneticLinearProgram *p,
      EvolutionSystem_Parameters *param,
                      FitnessCase *fitness_case_training_set,
                      SymbolTable *symbol_table,
                             bool  alert = false)
```

Populate the fitness field of a `GeneticLinearProgram` by calculating its fitness against the full fitness case training set using the fitness function described in this dissertation. The `param` argument holds values such as the size of the training set.

```
bool GeneticLinearProgram_AcceptanceTestOnTestSet(
            GeneticLinearProgram *p,
      EvolutionSystem_Parameters *param,
                      FitnessCase *fitness_case_test_set,
                      SymbolTable *symbol_table)
```

Returns a `bool` indicating if the given `GeneticLinearProgram` passes all of the fitness cases in the test set.

```
void EvolutionSystem_EvolveInstructionStringFromParseTree(
                   ParseTreeNode  *target_program,
          EvolutionSystem_Report  *output_report,
               InstructionString **output_string,
      EvolutionSystem_Parameters  *param,
                            bool   reporting,
     InstructionSetProbabilistic  *instruction_set,
                      SymbolTable  *symbol_table,
                InstructionString  *input_string_embryo)
```

This function is the base LGP implementation function which tries to evolve a low level instruction string with the same semantics as expressed by the input IR given by the `ParseTreeNode` pointed to by `target_program` and the `SymbolTable` pointed to by `symbol_table`. The instruction set available to the system is given by `instruction_set`. An instance of `EvolutionSystem_Report` must be supplied to hold statistical data produced during evolution. The evolutionary system is controlled by the parameters given in the `EvolutionSystem_Parameters` instance `param`. If reporting is `true`, continuous status reports are printed to the standard output console window at intervals defined by `param`. If an acceptable candidate instruction string is found, a copy of this program will be returned by `output_string`. If not, `output_string` will not be altered. If the refinement model is used, the previously obtained `InstructionString` is supplied by `input_string_embryo`.

```
void EvolutionSystem_Master_PrepareSymbolTableEntriesForBranchingPoints(
       const std::string &prefix_of_current_node,
           ParseTreeNode *target_program,
             SymbolTable *symbol_table)
```

This function augments the `SymbolTable symbol_table` with new symbols representing the intermediate values produced by evaluating the interior nodes in the given tree, in preparation for subprogram evolution by the 'incremental' method. This function acts recursively. The `prefix_of_current_node` argument supplies the string fragment which should appear at the start of all variables produced as a result

of considering this node; this parameter is used to assemble the new names of variables based on their position in the tree.

```
void EvolutionSystem_EvolveInstructionStringFromParseTree_PerformIncrement(
               const std::string  &prefix_of_current_node,
                   ParseTreeNode  *target_program,
          EvolutionSystem_Report  *output_report,
               InstructionString **output_string,
      EvolutionSystem_Parameters  *param,
     InstructionSetProbabilistic  *instruction_set,
                     SymbolTable  *symbol_table)
```

The final result of this function will be the creation of an `InstructionString` containing the appended programs necessary to create the full program by considering partial programs and combining them. This function works by constructing a new parse tree program encapsulating the immediate actions of the root node of the given parse tree, then evolving a program that performs that action. It recursively evolves solutions to child programs which are then combined at the end of the function.

```
void EvolutionSystem_EvolveInstructionStringFromParseTree_Incremental(
                   ParseTreeNode  *target_program,
          EvolutionSystem_Report  *output_report,
               InstructionString **output_string,
      EvolutionSystem_Parameters  *param,
     InstructionSetProbabilistic  *instruction_set,
                     SymbolTable  *symbol_table)
```

This function is the main LGP wrapper function which tries to evolve a low level instruction string by the 'incremental' model with the same semantics as expressed by the input IR given by the `ParseTreeNode` pointed to by `target_program` and the `SymbolTable` pointed to by `symbol_table`. The instruction set available to the system is given by `instruction_set`. An instance of `EvolutionSystem_Report` must be supplied to hold statistical data produced during evolution. The evolutionary system is controlled by the parameters given in the `EvolutionSystem_Parameters` instance `param`. If reporting is `true`, continuous status reports are printed to the standard output console window at intervals defined by `param`. If an acceptable candidate instruction string is found, a copy of this program will be returned by `output_string`. If not, `output_string` will not be altered.

```
void EvolutionSystem_VerifyInstructionStringFromParseTree(
                   ParseTreeNode  *target_program,
          EvolutionSystem_Report  *output_report,
               InstructionString  *input_string,
      EvolutionSystem_Parameters  *param,
                     SymbolTable  *symbol_table)
```

This function performs training and test set analysis on an `InstructionString` and determines if it passes both sets of fitness cases. This is used to verify the viability of composite programs produced by incremental evolution.