# MATHEW CARR – MSc. Project ‘Final Presentation’ Report
# Compiler Code Generation Through Genetic Programming

## Aims

The aims of this project are twofold:

To investigate methods whereby Genetic Programming (GP) techniques (and in particular Linear Genetic Programming (LGP) techniques) can be applied to achieve or expedite the code generation phase of software compilation.

To produce a complete, functional and well-documented LGP environment to allow experiments to be conducted. A series of test cases, in the form of input programs expressed in a high level language, have been devised to explore the behaviour of the evolutionary system. These test cases can be applied to the LGP system to repeat the experiments and observe evolutionary behaviour comparable to that identified in the report.

## Software Design

### Low Level Target Architecture

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 accessible memory. All storage areas and intermediate values are of the 64-bit signed integer data type (referred to as **`VM_TYPE`** by typedef).

The register file consists of a configurable number of general purpose registers. The symbolically accessible memory contains an arbitrary number of cells that may be addressed by a **`SYMBOL_TABLE_KEY`**. Both the register file and the memory are available to programs for both read and write access at any time.

A simplified, RISC-like low level language 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 **`SYMBOL_TABLE_KEY`**
<v> indicates that the argument is a **`VM_TYPE`**

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

For example, an instruction performing addition may appear as ADD 2, 0, 1. This instruction consists of the operation (represented by the readable mnemonic ‘ADD’), the number of the register into which the value of the result of the addition should be placed (the ‘destination register’), and the numbers of the registers from which the values to be used in the addition should be read (the ‘source registers’). This instruction stores into register 2 the value produced by adding the values currently stored in registers 0 and 1.

### Symbols

Within the program, symbols are always referred to in the context of a `SymbolTable` structure. The `SymbolTable` structure associates each symbol with a `SYMBOL_TABLE_KEY` (unsigned 32-bit integer). The `SYMBOL_TABLE_KEY` acts as a key into the `SymbolTable` to allow the properties of the symbol to be accessed quickly.

### Instructions

Within the program, low level language instructions are stored as instances of the structure **`Instruction`**. An **`Instruction`** consists of a **`VM_INSTRUCTION_OPERATION`** indicating which operation the **`Instruction`** holds, and an array of **`InstructionOperand`** holding the operands of the instruction. An **`InstructionOperand`** can hold the index of a register, a **`SYMBOL_TABLE_KEY`** or a direct value (**`VM_TYPE`**).

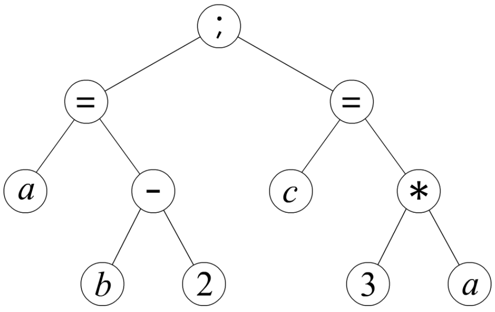
An instruction string is stored as a linked list of **`Instruction`** instances. The **`InstructionString`** type is defined as **`std::list<Instruction>`** by typedef.

### Parse Tree and Symbol Table

The source language considered in these experiments is a simple high level imperative language, similar to C. To begin, this language will only feature sequences of statements containing arithmetic operations between integer constants and named variables. The only data type used within the language is the 64-bit signed integer. The exact syntax of this language is not important, however, as this project will manipulate the parse tree intermediate representation produced during compilation. Where a parse tree is shown in this document, high level language source code will be shown to ease reading.

It is assumed that the parsing and lexical analysis stages of compilation have been completed previously, and the complete results of these operations are available for use.

For the purpose of these experiments, an input program is given as a parse tree and symbol table pair. A parse tree is a directed, pointed acyclic tree. A visualisation of a possible input program parse tree is shown below, together with the high level language program it encapsulates.



```
a = b-2;
c = 3*a
```

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. Constants (such as 2 and 3 in the above program) are treated no differently than variables, other than the setting of a flag in the symbol table.

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 this value returned. These semantics are similar to the comma operator in C.

This tree has the effect of first calculating the value of the expression `b-2` and assigning this value to the variable a, and then calculating the value of the expression `3*a` and assigning this value to c.

We may evaluate the program considering one instantiation of an input system state to produce a single output system state. That is, numerically. We may alternatively evaluate the program symbolically, across all input states, to provide an algebraic representation of the system state to system state mapping given by the program.

A symbol table contains the information about the variables referred to in the input program. The symbol table is stored internally as a mapping from unique keys (unsigned integers) to structures containing data defining the nature of the variable.

This data includes:

- The symbolic name of the variable.
- The scoping of the variable in the context of the program. The parse tree may represent an isolated subroutine apart from calling program. In this case, certain variables may exist only within the scope of the subroutine.
- The constant nature of the variable.

For the purposes of this project, we consider only variables of signed 64-bit integer type.

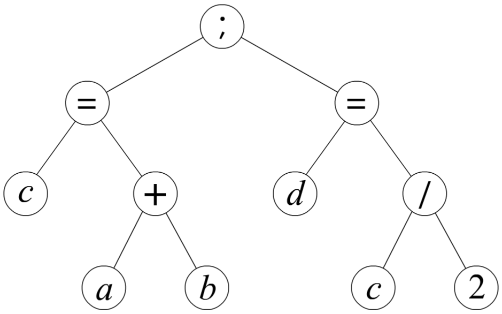
**Target Architecture**

The target architecture for the evolution of programs is a simple register machine consisting of a number of general purpose registers, capable of storing 64-bit signed integers and available to the inserted program, and a symbolically accessible memory: storage for 64-bit signed integer values that may be addressed by a symbolic name for read or write access. Software has been created that allows for non-interactive or interactive (user-stepped) interpretation of programs written in the low level language for this architecture.

A simplified, RISC-like low level language 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.

For example, an instruction performing addition may appear as ADD 2, 0, 1. This instruction consists of the operation (represented by the readable mnemonic ‘ADD’), the number of the register into which the value of the result of the addition should be placed (the ‘destination register’), and the numbers of the registers from which the values to be used in the addition should be read (the ‘source registers’). This instruction stores into register 2 the value produced by adding up the values currently stored in registers 0 and 1.

Possible programs performing the operation ‘*Calculate the integer mean of the values of the variables a and b, and store the result in variable d*’ are shown below in high level source language form, parse tree form and as an instruction string written in the low level language.



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

```
LOADS 0,    a    // load the value of variable a into register 0
LOADS 1,    b    // load the value of variable a into register 0
ADD   0, 0, 1    // add the values of registers 0 and 1 and store the result in register 0
LOADV 1,    2    // load the direct value 2 into register 1
DIVP  0, 0, 1    // divide (protected division) the value of r0 by the value of r1 and store the result in r0
STORS 0,    d    // store the value of register 0 into variable d
HALT             // end program
```

Note that the version written in the low level language does not refer to variable c at any time during its execution: the semantics of the program did not specify its inclusion. An optimising compiler may produce machine code that removes the need to directly store or consider the intermediate calculation; certain architectures provide a ‘barrel shift’ that can augment a standard arithmetic instruction with a bitshift operation. This may only occur if it is determined that the variable c is not required subsequently. The required information may be obtained by inspecting the scope of variable c, defined in the symbol table.

In this project, Linear Genetic Programming (LGP) will be used to attempt to induce programs that have the same semantics as an input program in the form of a parse tree and symbol table pair. It will do this by first producing a random population of candidate programs in the low level language, calculating a value representing the suitability of each program, then repeatedly combining and modifying programs with high suitability with the aim of iteratively producing successively more suitable programs until a program that is sufficiently suitable is generated. The calculation of the suitability value, referred to henceforth as 'fitness' to mirror genetic programming literature, will include the degree of correlation to the semantics of the input parse tree program, and several other measures reflecting the 'sanity' present in the candidate programs.

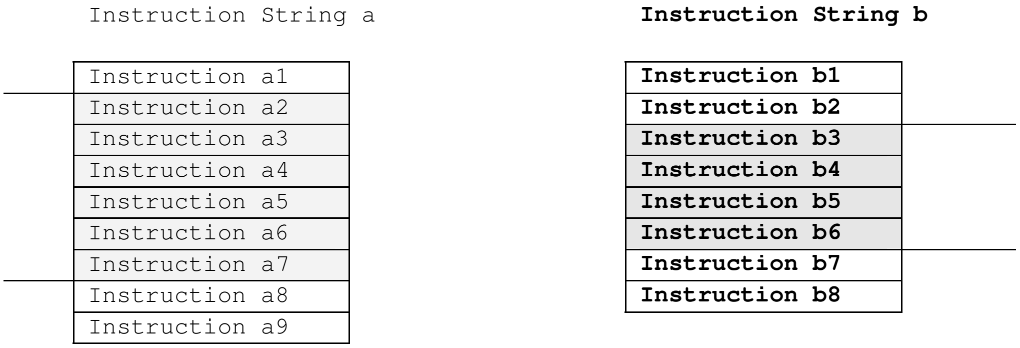
### Genetic Operations

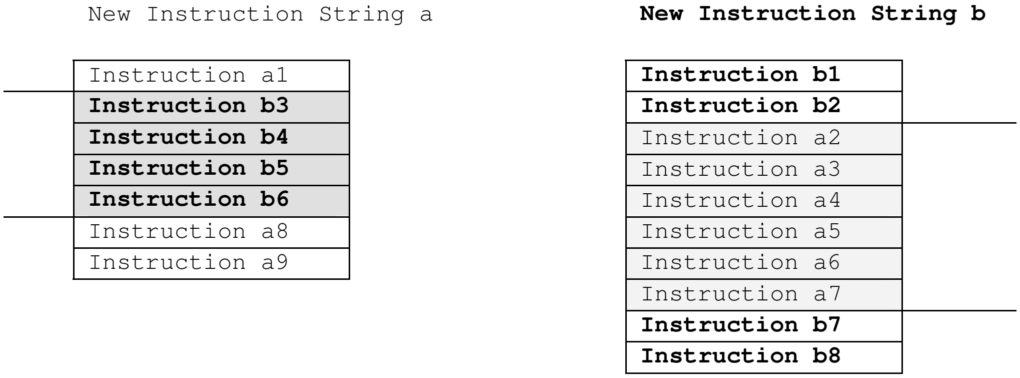
Tournament selection will be used to select programs from the population for application to the available operations. In tournament selection, N programs are randomly selected from the population to participate in the tournament, then from this set the M programs with the highest fitness values are passed to the operator.

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 to be removed from the population.

There are two operations available to the evolutionary system to manipulate candidate programs. These actions are selected randomly according to rate parameters given to the evolutionary system.

The crossover operation combines the contents of two source instruction strings to produce two new instruction strings, which are then inserted into the population. Transition points are randomly placed within the two instructions. The new strings are constructed by copying instructions from 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.





New Instruction String a
Instruction a1
**Instruction b3**
**Instruction b4**
**Instruction b5**
**Instruction b6**
Instruction a8
Instruction a9

**New Instruction String b**
**Instruction b1**
**Instruction b2**
Instruction a2
Instruction a3
Instruction a4
Instruction a5
Instruction a6
Instruction a7
**Instruction b7**
**Instruction b8**

The effect of the mutation operation is to alter an existing instruction at random from the chosen instruction string. One component of the instruction is chosen at random from the available components (the number and nature of which will depend on the current operation). If an operand component is chosen, an operand of compatible type is chosen in its place. If the operation component is chosen, then the all components of the instruction are reinitialised.

In addition, the mutation operation may insert a new random instruction or remove an existing instruction chosen at random from the string.

**Fitness Case Construction**

The first stage is to establish the semantics of the input program, as these will be used to determine the degree of semantic correlation between the parse tree and a candidate low level language instruction string.

As stated previously, it is assumed that the parsing and lexical analysis stages of compilation have been fully completed. Therefore, we can assume that the symbol table contains sufficient information to determine which variables referred to within the input program are of importance.

We can draw from the symbol table the following attributes of any given variable:

- If the variable exists purely within some limited scope, solely as a holder of intermediate calculations, then candidate programs are permitted to freely read and write to this variable at any time.
- If the variable exists in a scope higher than that of the parse tree, then candidate programs are permitted to read the value of this variable, but they are not permitted to change it.
- If the variable has a value of use associated with it before execution begins.

The symbol table contains flags determining whether the value of each variable must change (as with d in the previous example), must not change (as with a, b and, implicitly, all variables other than c), or if it doesn't matter (as with c).

Intuitively, it appears that it may be possible to execute the low level instruction string in a symbolic fashion to produce an exact statement of the state to state mapping that the program performs. This statement could then be compared with a statement describing the semantics of the parse tree to determine the degree of semantic correlation. However, I believe that this approach would become unworkable for all but the simplest of cases.

Instead, a sampling of the possible values of the input variables is considered. 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 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.

### Fitness Evaluation

The fitness value for a candidate is calculated as the total of the degree of semantic correlation and additional values representing the 'sanity' of the candidate program, over all fitness cases.

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, a constant penalty is given together with a variable amount of penalty as a function of the error.

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.

## Experiment Design

### Summary

The aim of this project is to investigate methods whereby Linear Genetic Programming techniques can be applied to achieve or expedite code generation.

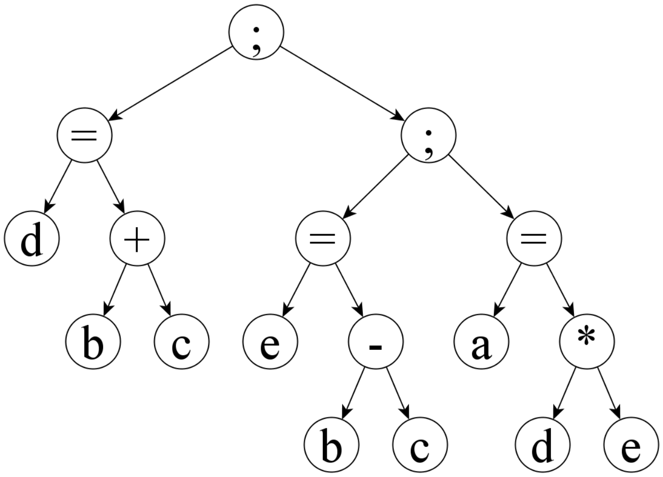
Two different methods of applying LGP are considered in this project, 'standard' and 'incremental'.

A 'solution program' is a program (a sequence of instructions in the low level language) which has the same semantics as an input program given in the form of a parse tree.

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 fitness metric described previously.

In the 'incremental' method, the input program 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.

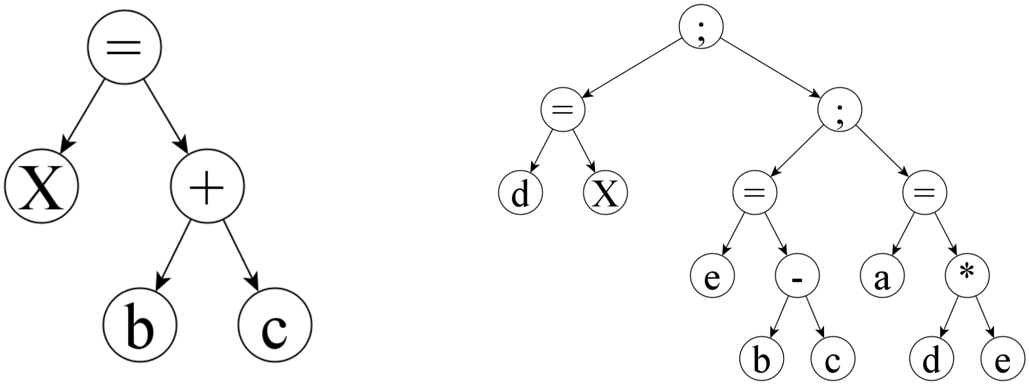
For example, consider the complex program shown below:



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

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 (ie. the program fragments with the highest precedence).

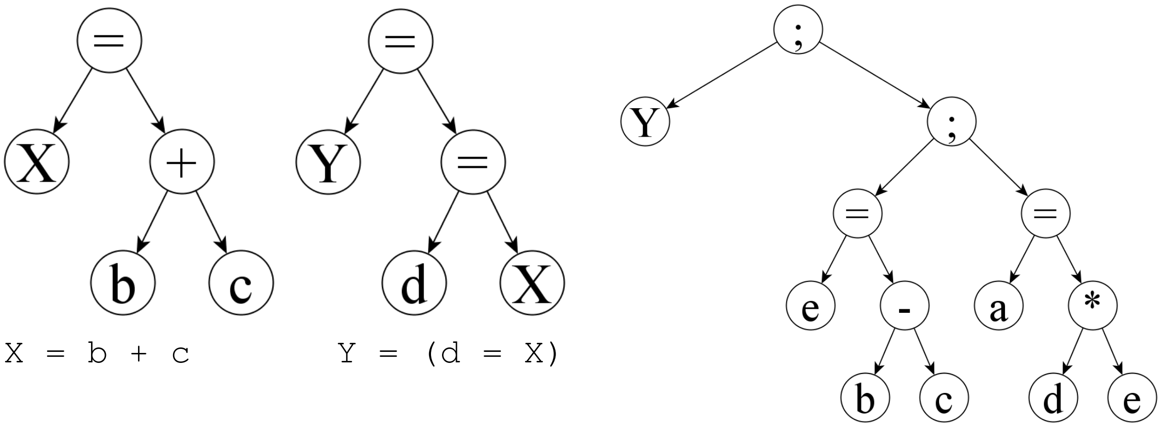
Here, the transformation begins with the + node in the lower left of the graph. This is extracted into a separate program with an assignment node at its root, a new temporary variable as its left child, and the extracted program as its right child. The extracted subtree from the original program is replaced 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 as an intermediate variable.



```
X = b + c
```

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

*The produced program fragment and the resulting modified program after the first extraction*



```
X = b + c
```

```
Y = (d = X)
```

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

*The produced program fragments and the resulting modified program after the second extraction*

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.

An additional stage of processing has been 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 found 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 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.

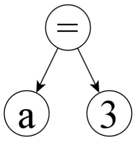
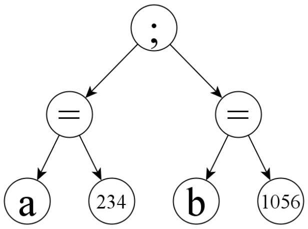
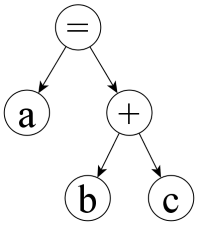
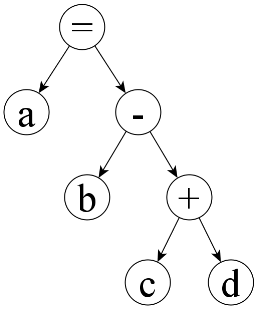
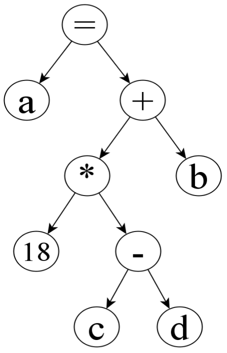
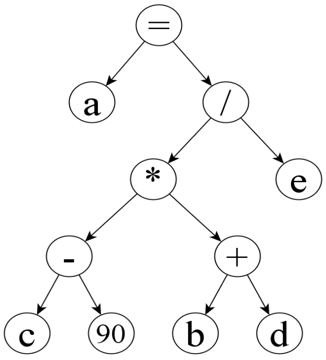
The evolutionary system will attempt to breed fitter programs and, given that solution programs are already present in the genetic population, these fitter programs will most likely be modified copies of the solution programs improved by application of the genetic operations.

**Measurements**
Two measures are used in this project to evaluate the result of the evolutionary methods.

For a given input program and set of evolution parameters, Computational Effort ($E_I$) is an empirical measure of the difficulty of evolving an appropriate solution program using those parameters. This value is the expected minimum number of instructions that must be considered to evolve a solution with 99% probability of success. It is similar to the Computational Effort (E) defined by Koza [koza], except we consider the number of instructions rather than the number of candidate solutions.

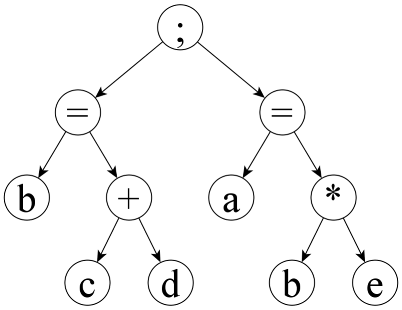
This different measure is required to compare the difficulty of evolving solutions using the ‘standard’ method and the ‘incremental’ method.

The following ten input programs are specified:

| | |
|---|---|
| **Program A01:**<br>Assignment of a single constant to a variable<br>`a = 3`<br>`Output: a`<br>`Constant: 3`<br><br> | **Program A02:**<br>Assignment of two constants to two variables<br>`a = 234;`<br>`b = 1056`<br>`Output: a, b`<br>`Constant: 234, 1056`<br><br> |
| **Program B01:**<br>Simple calculation; two input variables one output variable<br>`a = b + c`<br>`Input: b, c`<br>`Output: a`<br><br> | **Program B02:**<br>Progressively more complex calculation<br>`a = b – (c + d)`<br>`Input: b, c, d`<br>`Output: a`<br><br> |
| **Program B03:**<br>Complex calculation involving a constant<br>`a = (18 * (c – d)) + b`<br>`Input: b, c, d`<br>`Output: a`<br>`Constant: 18`<br><br> | **Program B04:**<br>Complex calculation with division<br>`a = ((c – 90) * (b + d)) / e`<br>`Input: b, c, d, e`<br>`Output: a`<br>`Constant: 90`<br><br> |

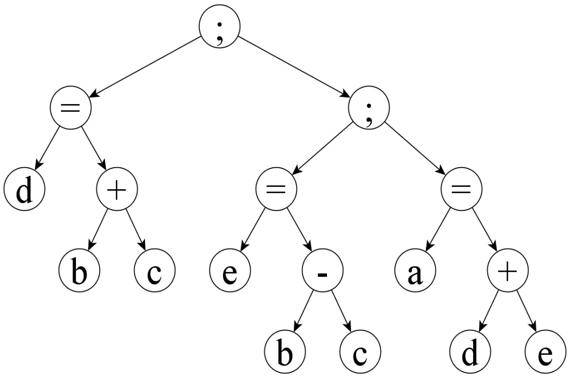
**Program C01:**
Calculation involving intermediate variable

```
b = c + d;
a = b * e
Input: c, d, e
Output: a
Intermediate: b
```



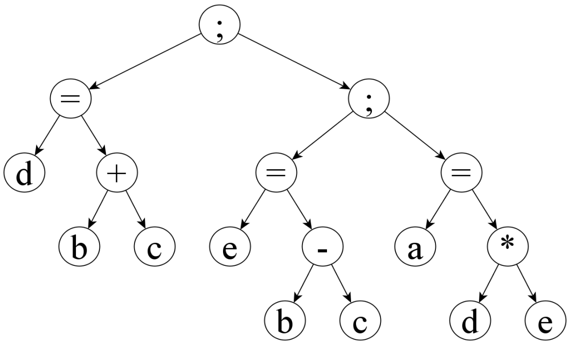
**Program D01:**
Calculation involving intermediate variables; optimisations possible (a = 2*b)

```
d = b + c;
e = b - c;
a = d + e
Input: b, c
Output: a
Intermediate: d, e
```



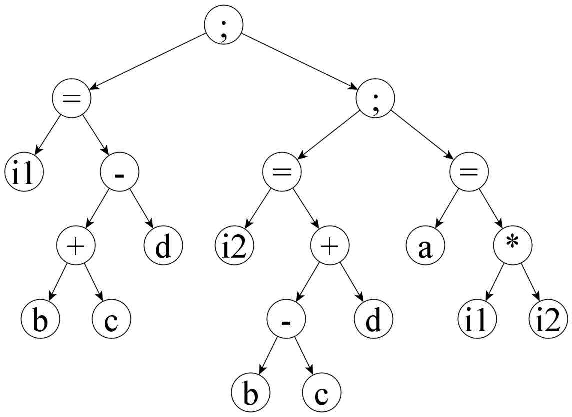
**Program D02:**
Calculation involving intermediate variables; difference of two squares (a = b*b – c*c)

```
d = b + c;
e = b - c;
a = d * e
Input: b, c
Output: a
Intermediate: d, e
```



**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
```



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 variables).

For each of these input programs, evolve.exe 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 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 in (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.

$E_I$ is used as an empirical measure of the difficulty of evolving a solution program: 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:

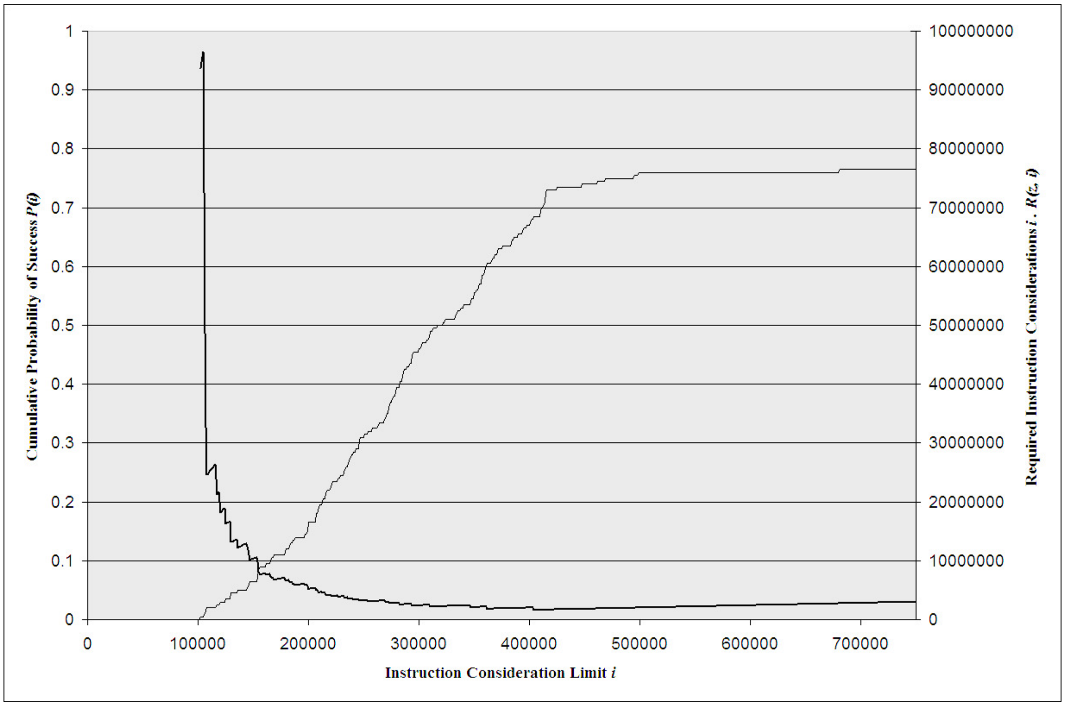
Koza suggests that multiple independent runs of the evolutionary system should be attempted to minimize the effect of premature population convergence to a sub-optimal solution. Over a large number of runs (200 in this project), we measure the number of instruction considered to evolve each solution 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. $E_I$ 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:



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 to gauge.

A simple tree walking algorithm has been developed capable of mechanically generating low level code given an input program in the form of a parse tree. The following psuedocode 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 (+,-,*,÷)
            TreeWalkingCompiler(left_child, register)
            TreeWalkingCompiler(right_child, register + 1)
            output [calc register register+1]
        ENDIF
    ELSE
        output [LOADS register left_child]
    ENDIF
END
```

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.

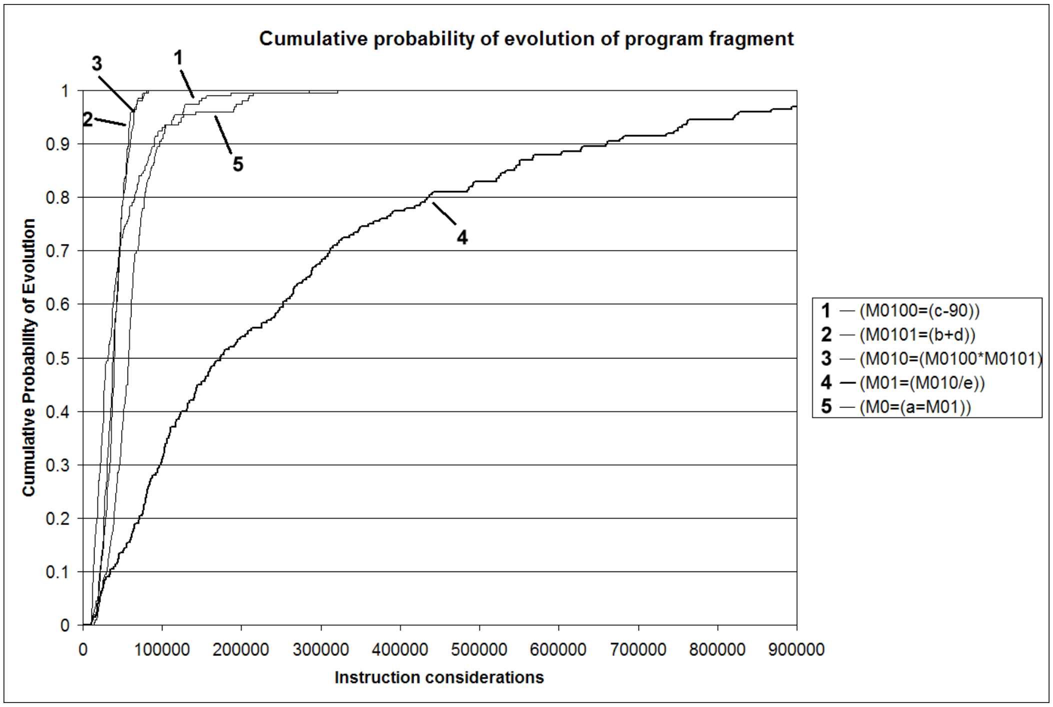
This tree walking algorithm will be applied to each of the defined programs to produce a predictable program length for comparison with those produced by the evolutionary methods. A fully optimised ‘perfect’ solution as produced by a skilled human programmer has also been produced.

## Analysis of Results

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.



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.

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. However, 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.

# Evaluation

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 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.

# Future 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.

This project has only focused on 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.

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.

## Result Data

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.

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.

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 |