Specifying the Behaviour of Python Programs: Language and Basic Examples

Maurice HT Ling
School of Chemical and Life Sciences, Singapore Polytechnic
Department of Zoology, The University of Melbourne, Australia
mauriceling@acm.org

Abstract

This manuscript describe BeSSY, a function-centric language for formal behavioural specification that requires no more than high-school mathematics on arithmetic, functions, Boolean algebra and sets theory. An object can be modelled as a union of data sets and functions whereas inherited object can be modelled as a union of supersets and a set of object-specific functions. Python list and dictionary operations will be specified in BeSSY for illustration.

1. Introduction

Formal methods are mathematically-based methods to define the actions of a system. The result of formal methods is a formal specification of a system which can then be verified for consistency using the rules of mathematics. A specification of a system is a description of what a system should do without going into details of how the system does it (Spivey, 1992; Woodcock et al., 2009). For example, the specification of all sort algorithms is the same with the end result being an ascending or descending sequence although the implementation and performance vary greatly. This suggests that a specification deals with the destination, not the journey, and certainly, no optimisation involved. Putting in layman's terms, a specification is to say “I am not interested in how you do it but this is what it must do”.

The formal aspect of formal specification is synonymous with un-ambiguity. That is to say, a formal specification cannot be ambiguous or implying more than one interpretation. This suggest that natural language cannot be the basis of formal specification as a natural language statement can have multiple interpretations based on context (such as situation, cultural context and body language), otherwise we will not have mis-communication or mis-interpretation of another person's intentions. The lack of the “un-writable” aspects of natural language communication such as body language is one of the main reasons that written communication such as emails and letters are easily mis-quoted or mis-interpreted.

In contrast, the language of mathematics is much more precise and less ambiguous than natural language. It is the precision and un-ambiguity of mathematical statements that underpins the formal aspect of formal methods. Ironically, it is also this nature of precision

in mathematics that makes the mastery of mathematics difficult – there is no veil of ambiguity and benefit of doubt to cushion against any errors in expression.

This is made worse by the general sense of "unfriendliness" and difficult type-setting of mathematical notations (Clarke et al., 1996; Sobel, 1996). These difficulties are emphatically stated in Bowen and Hinchey (1995) when they exclaimed that "*many nonformalists seems to believe that formal methods are merely an academic exercise – a form of mental masturbation that has no relation to real-world problems.*"

A number of student/instructor surveys on formal methods in the 1990s revealed that students understood the importance of mathematical formalism in software engineering but were concerned about its adoption and utility in the industry (Palmer and Pleasant, 1995; Sobel, 1996).

Despite the inherent difficulties, formal methods of specification are slowly gaining acceptance (Sharpe, 2004) over the last decade through a number of success stories (Hinchey and Bowen, 1999) in industrial context (Berry, 2008; Ciapessoni et al., 2002). For example, formal methods have been used in the following situations: to verify compliance of floating-point operations in hardware (Russinoff, 1998; O'Leary et al., 1999), real-time CORBA-based application (Rossi and Mandrioli, 2004), and memory allocation in C language compilation (Leroy and Blazy, 2008). Sobel and Clarkson (2002) did a case-controlled experiment comparing software implementations and found that the group of students using formal methods achieved more correct implementation than the group that did not use formal methods. A survey by Woodcock et al. (2009) reveals an improvement of software quality with no cases reporting a decrease in quality through the use of formal methods.

These successes do not imply that formal methods are easy to learn. The learning curve of formal methods has always been steep and much of it is to achieve competency with the notations (Chiang, 2004). This is supported by Bowen and Hinchey (2005) in their First Commandment, "*Thou shalt choose an appropriate notation.*" They propose that any forms of mathematical and symbolic obfuscation should be avoided.

This manuscript describes the language for a behavioural specification system (abbreviated to BeSSY) which uses no more than high-school mathematics (arithmetic, sets theory, functions, and Boolean algebra) as a basis to describe the behaviour of software systems. Functions and logic (in the form of Boolean algebra) is an important aspect of computer programming; hence, is a bridge to convert a specification to skeletal source codes.

The rest of this manuscript are organized as follow: Section 2 describes the mathematical aspect of BeSSY language while Section 3 illustrates the use of BeSSY language through specifying the operations of Python list and dictionary. Section 4 concludes this manuscript with a general discussion of BeSSY and future work.

2. The BeSSY Language

BeSSY is a function-centric language – every operation is a function. Other terms synonymous with "function" includes operator, transformation, mapping etc. The concept of function is central and fundamental in mathematics and is also a crucial concept in programming. The main difference between a programmatic function and a mathematical function is that a programmatic function is always pre-fixed (such as <operator> <operand(s)>) whereas a mathematical function can be pre-fixed (such as Σx), in-fixed (such as 8+4), or post-fixed (such as 4!).

Strictly, BeSSY only allows pre-fixed function but since it is more natural to write addition as an in-fixed operator without any detriment to the mathematical rigor, a list of arithmetic equivalence rules (Section 2.2) relaxes the requirement of pre-fixing and allows for casting specific arithmetic operators from in-fixed form to pre-fixed form as required by BeSSY.

The rest of this section contains the definition of BeSSY language.

2.1. Logical True and Holding Variable

Definition 1.1: The term 'true' is used to denote logical true or not numeric zero.

Definition 1.2: The term 'false' is used to denote logical false or numeric zero.

Definition 1.3: A variable, hV, is used to denote a holding variable for the calling function.

For example, given the function $y = z^2$ and z is equal to 5, then the right-hand side (RHS) of the equation is evaluated to 25 ($5^2 = 25$). The variable, hV, is used to hold the value of 25 to transit to the calling equation.

2.2. Arithmetic Equivalence Rules

The following arithmetic equivalence rules are defined to enable the use of natural arithmetic expressions, which are in-fixed operators. These natural arithmetic expressions are re-composed into pre-fixed function forms, summarized in Table 2.1.

Definition 2.1: Arithmetic equal is defined as a function, =(x, y), where x = y.

Definition 2.2: Material equivalence (usually denoted by the symbol '⇔' or '<=>') is defined as a function, ==(x, y), where x and y are equivalent to each other.

```
D2.2.1:    x <=> y <=> ==(x, y)
```

Hence, we can re-write Definition 2.1 above as

```
D2.1.1:    =(x, y) <=> x = y
D2.1.2:    ==(=(x,y), x=y)
```

Definition 2.3: Material implication (usually denoted by the symbol '⇒' or '=>') is defined as a function, =>(x, y), where x implies y.

```
D2.3.1:    x => y <=> =>(x, y)
```

Definition 2.4: Arithmetic addition is defined as a function, +(x, y), and returns the evaluated value to hV,

```
D2.4.1:    x + y <=> +(x, y)
D2.4.2:    +(x, y) => =(hV, +(x,y))
```

Definition 2.5: Arithmetic subtraction is defined as a function, -(x, y), and returns the evaluated value to hV,

```
D2.5.1:    x - y <=> -(x, y)
D2.5.2:    -(x, y) => =(hV, -(x,y))
```

Definition 2.6: Arithmetic multiplication is defined as a function, *(x, y), and returns the evaluated value to hV,

```
D2.6.1:    x * y <=> *(x, y)
D2.6.2:    *(x, y) => =(hV, *(x,y))
```

Definition 2.7: Arithmetic division is defined as a function, /(x, y), and returns the evaluated value to hV,

```
D2.7.1:    x / y <=> /(x, y)
D2.7.2:    /(x, y) => =(hV, /(x,y))
```

Definition 2.8: Arithmetic exponential is defined as a function, ^(x, y), and returns the evaluated value to hV,

D2.8.1: x^y <=> x ^ y

```
D2.8.2:    x ^ y <=> ^(x, y)
D2.8.3:    ^(x, y) => =(hV, ^(x,y))
```

Definition 2.9: Arithmetic modulus is defined as a function, %(x, y), and returns the evaluated value to hV,

```
D2.9.1:    x % y <=> %(x, y)
D2.9.2:    %(x, y) => =(hV, %(x,y))
```

Definition 2.10: Arithmetic more than is defined as a function, >(x, y), and returns the 'true' to hV if the expression is arithmetically correct or else returns the value 'false' to hV if the expression is arithmetically incorrect,

```
D2.10.1:  x ≻ y <=> >(x, y)
D2.10.2:  x ≻ y => 'true' = >(x, y)
D2.10.3:  x ≻ y => =(hV, =('true', >(x, y)))
D2.10.4:  x ⊁ y => =(hV, =('false', >(x, y)))
```

Definition 2.11: Arithmetic less than is defined as a function, <(x, y), and returns the 'true' to hV if the expression is arithmetically correct or else returns the value 'false' to hV if the expression is arithmetically incorrect,

```
D2.11.1:  x ≺ y <=> <(x, y)
D2.11.2:  x ≺ y => 'true' = <(x, y)
D2.11.3:  x ≺ y => =(hV, =('true', <(x, y)))
D2.11.4:  x ⊀ y => =(hV, =('false', <(x, y)))
```

Definition 2.12: Arithmetic less than or equals (symbolically denoted as ≤ or ≦) to is defined as a function, <=(x, y), and returns the 'true' to hV if the expression is arithmetically correct or else returns the value 'false' to hV if the expression is arithmetically incorrect,

```
D2.12.1:  x ≤ y <=> x <= y
D2.12.2:  x ≤ y <=> <=(x, y)
D2.12.3:  x ≤ y => 'true' = <=(x, y)
D2.12.4:  x ≤ y => =(hV, =('true', <=(x, y)))
D2.12.5:  x ≰ y => =(hV, =('false', <=(x, y)))
```

Definition 2.13: Arithmetic more than or equals to (symbolically denoted as ≥ or ≧) is defined as a function, >=(x, y), and returns the 'true' to hV if the expression is arithmetically correct or else returns the value 'false' to hV if the expression is arithmetically incorrect,

```
D2.13.1:  x ≥ y <=> x >= y
D2.13.2:  x ≥ y <=> >=(x, y)
D2.13.3:  x ≥ y => 'true' = >=(x, y)
D2.13.4:  x ≥ y => =(hV, =('true', >=(x, y)))
D2.13.5:  x ≱ y => =(hV, =('false', >=(x, y)))
```

Definition 2.14: Arithmetic not equals to is defined as a function, !=(x, y), where x ≠ y. The typographical arithmetic equivalent form is x != y. This function returns 'true' to hV if x is not equals to y, or else this function will return 'false' to hV if x is equals to y.

```
D2.14.1:  x ≠ y <=> x != y
D2.14.2:  x ≠ y <=> !=(x, y)
D2.14.3:  x ≠ y => 'true' = !=(x, y)
D2.14.4:  x ≠ y => =(hV, =('true', !=(x, y)))
D2.14.5:  x = y => =(hV, =('false', !=(x, y)))
```

Definition	Mathematical Meaning	Function Form	Arithmetic Form
2.1	Equals	=(x, y)	x = y
2.2	Material equivalence	==(x, y)	x <=> y
2.3	Material implication	=>(x, y)	x => y
2.4	Addition	+(x, y)	x + y
2.5	Subtraction	-(x, y)	x - y
2.6	Multiplication	*(x, y)	x * y
2.7	Division	/(x, y)	x / y
2.8	Modulus	%(x, y)	x % y
2.9	Exponential	^(x, y)	x ^ y
2.10	More than	<(x, y)	x < y
2.11	Less than	>(x, y)	x > y
2.12	More than or equal	<=(x, y)	x <= y
2.13	Less than or equal	>=(x, y)	x >= y
2.14	Not equals to	!=(x, y)	x != y

Table 2.1: Syntactically equivalence rules between arithmetic expression and functional form.

2.3. Set Construction

This section defines the syntactic rules for set construction.

Definition 3.1: A function for set construction, set, is defined as

```
D3.1.1:   A = { x | conditionA | ... | conditionZ} <=>
               A = set(x, conditionA, ..., conditionZ)
D3.1.2:   A = { x | conditionA | ... | conditionZ} <=>
               =(A, set(x, conditionA, ..., conditionZ))
D3.1.3:   A = { x | conditionA | ... | conditionZ} =>
               =(A, =(hV, set(x,
                              conditionA, ..., conditionZ)))
```

where conditions will be evaluated in the order of appearance and 'x' will be evaluated to lowest form.

Definition 3.2: A sequence is a multiset. A multiset is a generalization of set whereby each member need not be unique.

Definition 3.3: A function for sequence construction, seq, is defined as

```
D3.3.1:   A = (x, ..., y) <=>
               A = seq(enum(x, ..., y))
D3.3.2:   A = (x, ..., y) <=>
               =(A, seq(enum(x, ..., y)))
D3.3.3:   A = (x, ..., y) =>
               =(A, =(hV, seq(enum(x, ..., y))))
```

where

```
D3.3.4:   seq(x, y) != seq(y, x)
D3.3.5:   !=(seq(x, y), seq(y, x))
```

Sequence construction function, seq, can also take conditions as per normal set construction and each condition must be fulfilled for each element in the order of enumeration. Each element will also be evaluated to lowest form.

```
D3.3.6:   A = (y | memb(y, X) | conditionA | … |
               conditionZ) <=>
               A => seq(y, memb(y, X), conditionA, …,
                        conditionZ)
D3.3.7:   A = (y | memb(y, X) | conditionA | … |
               conditionZ) <=>
               =(A, seq(y, memb(y, X), conditionA, …,
                        conditionZ))
D3.3.8:   A = (y | memb(y, X) | conditionA | … |
               conditionZ) =>
               =(A, =(hV, seq(y, memb(y, X),
                         conditionA, …,conditionZ)))
```

Definition 3.3: A function for enumeration, enum, is defined as

```
D3.3.1: enum(x, y, ..., z) <=> x, y, ..., z
          (if x, y, ..., z are not sequences)
D3.3.2: enum(x, y, ..., z) <=> x1, ..., xn, y, z
          (if x is a sequence, y and z are not)
D3.3.3: enum(x, y, ..., z) <=> x1, ..., xn, y1, ..., ym, z
          (if x and y are sequence, z is not)
D3.3.4: enum(x, y, ..., z) <=> x1, ..., xn, y1, ..., ym,
                               z1, ..., zi
          (if x, y, ..., z are sequences)
```

Hence, enumeration can be seen as a flattening of nested set or sequence.

2.4. Basic Logical and Set Operators

This section defines a list of fundamental logical and set operations.

Definition 4.1: The logical AND operator (symbolically denoted as ∧) and equivalent set intersect or global set intersect (symbolically denoted as ∩) is defined as a function, &. The evaluated value is returned to hV.

```
D4.1.1:   x₁ ∩ ... ∩ xₙ <=> &(x₁, ..., xₙ)
D4.1.2:   x₁ ∩ ... ∩ xₙ => =(hV, &(x₁, ..., xₙ))
```

Definition 4.2: The logical OR operator (symbolically denoted as ∨) and equivalent set union or global set union (symbolically denoted as ∪) is defined as a function, $. The evaluated value is returned to hV.

```
D4.2.1:   x₁ ∪ ... ∪ xₙ <=> $(x₁, ..., xₙ)
D4.2.2:   x₁ ∪ ... ∪ xₙ => =(hV, $(x₁, ..., xₙ))
```

Definition 4.3: The logical NOT operator (symbolically denoted as ~) is defined as a function, ~. This operator is also used to describe complement set. The evaluated value is returned to hV.

```
D4.3.1:   x' <=> ~(x)
D4.3.2:   x' => =(hV, ~(x))
```

Definition 4.4: The set operator, member of (symbolically denoted as ∈), is defined as a function, memb(x, A). This function returns 'true' to hV if x is a member of set A, otherwise it returns 'false' denoting that x is not a member of set A.

```
D4.4.1:   x ∈ A <=> memb(x, A)
D4.4.2:   x ∈ A => =(hV, =('true', memb(x, A)))
D4.4.3:   x ∉ A => =(hV, =('false', memb(x, A)))
```

Definition 4.5: The set operator, not a member of (symbolically denoted as ∉), is defined as a function, nmemb(x, A). This function returns 'true' to hV if x is not a member of set A, otherwise it returns 'false' denoting that x is a member of set A.

```
D4.5.1:   x ∉ A <=> nmemb(x, A)
D4.5.2:   x ∉ A => =(hV, =('true', nmemb(x, A)))
D4.5.3:   x ∈ A => =(hV, =('false', nmemb(x, A)))
```

Definition 4.6: The set operator, subset (symbolically denoted as ⊂), is defined as a function, ss(A, B). This function returns 'true' to hV if A is a subset of B, otherwise it returns 'false' denoting that A is not a subset of B.

```
D4.6.1:   A ⊂ B <=> ss(A, B)
D4.6.2:   A ⊂ B => =(hV, =('true', ss(A, B)))
D4.6.3:   A ⊄ B => =(hV, =('false', ss(A, B)))
```

Definition 4.7: The set operator, proper subset (symbolically denoted as ⊆), is defined as a function, pss(A, B). This function returns 'true' to hV if A is a proper subset of B, otherwise it returns 'false' denoting that A is not a proper subset of B.

```
D4.7.1:   A ⊆ B <=> pss(A, B)
D4.7.2:   A ⊆ B => =(hV, =('true', pss(A, B)))
D4.7.3:   A ⊈ B => =(hV, =('false', pss(A, B)))
```

Definition 4.8: The set difference operator is defined as a function, sdiff(A, B) as

```
D4.8.1:   sdiff(A, B) <=> A\B
D4.8.2:   A\B <=> {x | memb(x, A) | nmemb(x, B)}
D4.8.3:   sdiff(A, B) =>
               =(hV, {x | memb(x, A) | nmemb(x, B)})
```

Definition 4.9: The cardinal operator, #, is defined as a function to return the measure of the number of elements in a set.

```
D4.9.1:   #(X) <=> |X|
D4.9.2:   #(X) => =(hV, #(X))
```

Definition 4.10: The random operator, rand, is defined as a function to return a random element in a set.

```
D4.10.1:  rand(X) => =(hV, rand(X))
```

2.5. Special Set Constructs

This section defines a list of sets with special meanings within BeSSY.

Definition 5.1: A set of real numbers ($R^{\#}$) is defined as real().

D5.1.1: $R^{\#}$ <=> real()
```
D5.1.2:   real() => =(hV, real())
```

Definition 5.2: A set of complex numbers is defined as

```
D5.2.1:   complex() <=> { x | nmemb(x, real()) }
D5.2.2:   complex() => =(hV, { x | nmemb(x, real()) })
```

Definition 5.3: A set of rational numbers is defined as ratn().

Definition 5.4: A set of irrational numbers is defined as

```
D5.4.1:   irratn() <=> { x | nmemb(x, ratn()) }
D5.4.2:   irratn() => =(hV, { x | nmemb(x, ratn()) })
```

Definition 5.5: A set of integer numbers is defined as

```
D5.5.1:   int() <=> { x | memb(x, real()) |
                         memb(x/2, {1, 0}) }
D5.5.2:   int() => =(hV, { x | memb(x, real()) |
                           memb(x/2, {1, 0}) })
```

Definition 5.6: A power set of A (2^A) is defined as

```
D5.6.1:   2^A <=> ps(A)
D5.6.2:   ps() => =(hV, ps())
```

Definition 5.7: A null set is defined as

```
D5.7.1:   ns() <=> {}
D5.7.2:   ns() => =(hV, {})
```

Definition 5.8: An universal set is defined as

```
D5.8.1:   us() <=> ~({})
D5.8.2:   us() => =(hV, ~({}))
```

Definition 5.9: A set of odd numbers is defined as

```
D5.9.1:   oddn() <=> { x | memb(x, int()) | x % 2 = 1 }
```

Definition 5.10: A set of odd numbers is defined as

```
D5.10.1:  evenn() <=> { x | memb(x, int()) | x % 2 = 0 }
```

2.6. Construction and Properties of Functions

Functions are the executable aspect of BeSSY.

Definition 6.1: A function is a composition of a list of functions from a set of pre-defined or previously-defined functions (G) that takes in a set of input variables (X) and output a set of result objects (Y). The notation for declaring a function named 'test' is given as follows,

```
D6.1.1:   test(X) = Y
D6.1.2:   test(X) <=> seq(z | memb(z, G))
```

Definition 6.2: The set of input variables (X) of a function is known as the domain of a function. A function, domain, is used to generate the domain of a function. The domain of itself is the universal set.

```
D6.2.1:   domain(function) <=> X
D6.2.2:   domain(function) => =(hV, X)
```

```
D6.2.3:   domain(domain) <=> us()
```

Definition 6.3: The set of output objects (Y) of a function is known as the range of a function. A function, range, is used to generate the range of a function. The range of itself is the null set.

```
D6.3.1:   range(function) <=> Y
D6.3.2:   range(function) => =(hV, Y)
D6.3.3:   range(range) <=> ns()
```

Definition 6.4: A coordinate of a function 'f' is defined as a 2-element ordered set where the first element is the input variable and the second element is the output object.

```
D6.4.1:   f(x) = y => (x, y)
```

Definition 6.5: The graph of a function 'f' can be defined by the set of all possible coordinates of the function f.

```
D6.5.1:   f(x) = y <=> {(x, y) | memb(x, domain(f)) |
                          memb(y, range(f)) }
```

Definition 6.6: Two functions, 'f' and 'g', are equivalent if function f and function g have the same graph.

```
D6.6.1:   f(X) = Y <=> g(X) = Y
D6.6.2:   (f(X) <=> g(X)) <=> =(hV, 'true')
```

Definition 6.7: A function 'f' is an identity function (1A) if the domain of f is equivalent to the range of f.

```
D6.7.1:   f(X) = X => 1A()
```

Definition 6.8: The inverse of function f, 'f-1', is defined when the domain of f is equivalent to the range of f-1, and the range of f is equivalent to the domain of f-1.

```
D6.8.1:   f(X) = Y => f-1(Y) = X
```

Definition 6.9: Every function has an inverse function of itself.

2.7. Construction and Properties of Objects

This section details the construction of objects and the rules related to the evaluation of sets and functions within objects.

Definition 7.1: An object (obj) is the union of a set of sets (S) and/or a set of pre-defined or previously defined functions (preF).

```
D7.1.1:   obj = $(S, preF)
```

Definition 7.2: A set of objects is denoted as 'Oset'.

```
D7.2.1:   Oset = { obj }
```

Definition 7.3: An object (obj) can be the union of other objects.

```
D7.3.1:   obj = { obj | memb(obj, Oset) }
```

Definition 7.4: Assuming that an object 'obj' is a union of set B and function F, set B and function F can be accessed external to object 'obj' using a location function '@', defined as

```
          Given obj = $(B, F)
D7.4.1:   @(B, obj) => =(hV, B)          (data access)
D7.4.2:   @(F(X), obj) => =(hV, F(X))    (function access)
```

Definition 7.5: Assuming that an object 'objS' is a union of object 'objA', and set C. Object 'objA' is a union of set B and function F, set B and function F can be accessed external to object 'objS' using a location function '@', defined as

```
          Given objA = $(B, F) and objS = $(objA, C)
D7.5.1:   @(@(B, objA), objS) => =(hV, B)
               (data access)
D7.5.2:   @(@(F(X), objA), objS) => =(hV, F(X))
          (function access)
```

Definition 7.6: It is understandable that this syntax will get very long and messy when there is multiple layers of object unions. Combining Definition 7.5 with Definition 2.2 (Material equivalence), the following short-hand can be defined,

```
          Given objA = $(B, F) and
               objS = $(objA, C,
                        ==(objAsB, @(B, objA),
                        ==(objAfF(X), @(F(X), objA)))
```

where set B and function F are made equivalence as objAsB and objAfF in object 'objS' respectively.

```
From D7.5.1:   @(objAsB, objS) => =(hV, objAsB)
From D7.5.2:   @(objAfF(X), objS) => =(hV, objAfF(X))
```

2.8. Build-in Functions

This section details a list of common build-in functions that are needed for logical flow, such as selection functions and iteration functions, and other utility functions.

Definition 8.1: An if-then-else selection function is defined as

```
D8.1.1:   IF(condF, X, Y) =>
               $(seq(k | memb(k, X) | condF ),
                 seq(h | memb(h, Y) | ~(condF) ))
```

The sequence X is evaluated if condF is true; the ordered set Y is evaluated if condF is false.

Definition 8.2: An iteration function, FOR, can be defined as

```
D8.2.1:   FOR(X, y, z) =>
               $(seq(k | memb(k, X) | &(y, z)))
```

The sequence X is evaluated if conditions y and z are true. In addition, an endless loop can be defined if conditions y and z are tautology (always true). For example,

```
D8.2.2:   FOR(X, 'true', 'true') => X is permanent
```

Definition 8.3: A generic mathematical expression wrapper function, MATH, is defined to wrap and evaluate any syntactically correct mathematical expression.

```
D8.3.1:   MATH(x) => =(hV, MATH(x))
```

where x is any syntactically correct mathematical expression.

Definition 8.4: A generic formal expression wrapper function, FORMAL, is defined to wrap and evaluate any syntactically correct formal expression 'exp' specified in a formal language 'x'.

```
D8.4.1:   FORMAL(x, exp) => =(hV, FORMAL(x, exp))
```

where x is any syntactically correct mathematical expression.

Definition 8.5: INDEX, declared in the form of INDEX(x, Y), is defined a function to give the position of element x in set Y.

```
D8.5.1:   INDEX(x, Y) => =(hV, INDEX(x, Y))
```

Definition 8.6: eINDEX, declared in the form of eINDEX(x, Y), is defined as a function to give the element in index x of set Y.

```
D8.6.1:   eINDEX(x, Y) => =(hV, eINDEX(x, Y))
```

3. Examples of Specification Construction with BeSSY

In this section, I will attempt to re-construct the specifications of Python list and dictionary operations using BeSSY.

3.1. Python List Operations

Given that 'data' is an ordered sequence, an empty 'data' can be defined as

```
(1)  data = seq(ns())
```

Hence, a Python list data type (L) can be defined as

```
(2)  L = $(data,
          __add__, __contains__, __delattr__,
          __delitem__, __delslice__,__eq__, __ge__,
          __getattribute__, __getitem__, __getslice__,
          __gt__, __iadd__, __imul__, __iter__, __le__,
          __len__,__lt__, __mul__, __ne__, __reversed__,
          __rmul__,__setattr__,__setitem__,__setslice__,
          append, count, extend, index, insert, pop,
          remove, reverse, sort)
```

Given that 'y' is to be added to L and returns the result as a new list, the concatenation function (__add__) can be defined as

```
(3)  __add__(y) = seq(enum(@(data, L), y))
```

Given that L is to be checked for the presence of 'y', the membership function (__contains__) can be defined as

```
(4)  __contains__(y) = memb(y, @(data, L))
```

Given that 'y' is the name of the attribute to be deleted from L, the delete attribute function (__delattr__) can be defined as

```
(5)  __delattr__(y) =>
          L = set(x | memb(x, L) | x != y)
```

Given that 'posn' is the integer position of the data to delete in L, the delete by index function (__delitem__) can be defined as

```
(6)  __delitem__(position)  =>
          @(data, L) = seq(x |
                           memb(x, @(data, L)) |
                           posn != INDEX(x, @(data, L)) )
```

Given that 'start' is the starting integer position and 'end' is the ending integer position to delete a segment of L, the delete item by slice function (__delslice__) can be defined as

```
(7)  __delslice__(start, end) =>
          IF(&(start ≤ end, 0 < start, 0 < end),
             @(data, L) =
                 enum(x, 0 ≤ index ≤ start,
                      eINDEX(index, @(data, L))) +
                 enum(x, end ≤ index ≤ #(@(data, L)),
                      eINDEX(index, @(data, L)))),
             ns())
```

Given that L is to be checked for equality to 'y', the equality function (__eq__) can be defined as

```
(8)  __eq__(y) = &(ss(y, @(data, L)),
                   ss(@(data, L), y))
```

Given that 'y' is to be checked for greater or equal to L, the greater or equal than function (__ge__) can be defined as

```
(9)  __ge__(y) =
          &(set(eINDEX(index, y) <=
                    eINDEX(index, @(data, L) |
                0 ≤ index ≤ #(y)),
            #(@(data, L)) = #(y))
```

Given that 'y' is the name of the attribute to access in L, the get attribute function (__getattribute__) can be defined as

```
(10) __getattribute__(y) = @(y, L)
```

Given that 'y' is the integer position of the data to access in L, the get item by index function (__getitem__) can be defined as

```
(11) __getitem__(y) = eINDEX(y, @(data, L))
```

Given that 'start' is the starting integer position and 'end' is the ending integer position to extract a segment of L, the get item by slice function (__getslice__) can be defined as

```
(12) __getslice__(start, end) =
          seq(x | start ≤ position < end |
                  eINDEX(position, @(data, L)))
```

Given that 'y' is to be checked for greater to L, the greater than function (__gt__) can be defined as

```
(13) __gt__(y) =
          &(set(eINDEX(index, y) <
                    eINDEX(index, @(data, L)) |
                0 ≤ index ≤ #(y)),
            #(@(data, L)) = #(y))
```

Given that 'y' is the data to be added into L (does not return a new list), the in-place concatenation function (__iadd__) can be defined as

```
(14) __iadd__(y) =>
          @(data, L) = enum(@(data, L), y)
```

Given that 'n' is the number of times to repeat L and present the results in a new list, the multiplication function (__mul__) can be defined as

```
(15) __mul__(n) = enum(FOR(__iadd__(@(data, L),
                           0 < x ≤n,
                           x = x + 1))
```

Given that 'n' is the number of times to repeat L and store the results in L (does not return a new list), the in-place multiplication function (__imul__) can be defined as

```
(16) __imul__(n) => @(data, L) = __mul__(n)
```

The iterator function (__iter__) presents an enumeration of the data in L; hence, can be defined as

```
(17) __iter__() = enum(@(data, L))
```

Given that 'y' is to be checked for lesser or equal to L, the lesser or equal than function (__le__) can be defined as

```
(18) __le__(y) =
       &(set(eINDEX(index, y) >=
               eINDEX(index, @(data, L)) |
             0 ≤ index ≤ #(y)),
         #(@(data, L)) = #(y))
```

The length function (__len__) which gives the number of data elements in L can be defined as

```
(19) __len__() = #(@(data, L))
```

Given that 'y' is to be checked for lesser to L, the lesser than function (__lt__) can be defined as

```
(20) __lt__(y) =
```

```
        &(set(eINDEX(index, y) >
                eINDEX(index, @(data, L)) |
              0 ≤ index ≤ #(y)),
          #(@(data, L)) = #(y))
```

Given that L is to be checked for non-equality to 'y', the nonequality function (__ne__) can be defined as

```
(21) __ne__(y) = ~(@(__eq__(y), @(data, L)))
```

The reverse iterator function (__reversed__) presents the data in L in the reverse order; hence, can be defined as

```
     index = #(@(data, L))
(22) __reversed__() = enum(x,
                           FOR(eINDEX(index, @(data, L)),
                               0 ≤index, index = index - 1))
```

The reverse multiplication function (__rmul__) in a Python list (L) has the same behaviour as multiplication function (__mul__),

```
(23) __rmul__(n) => __mul__(n)
```

Given that 'name' is the attribute to set as 'value', the set attribute function (__setattr__) can be defined as

```
(24) __setattr__(name, value) => @(name, L) = value
```

Given that 'y' is the position in L to set to 'value', the set item function (__setitem__) can be defined as

```
(25) __setitem__(y, value) =>
          @(INDEX(y, @(data, L)), L) = value
```

Given that a sequence (y) is to replace L from the 'start' position to 'end' position, the set a slice of items function (__setslice__) can be defined as

```
(26) __setslice__(start, end, y) =>
          IF(&(#(y) = end-start, 0 < start, 0 < end,
               start ≤ #(@(data, L)),
               end ≤ #(@(data, L)),
             seq(__setitem__(index ,
                 eINDEX(index, @(data, y))),
                 start ≤ index < end),
             ns())
```

The 'append' function concatenates the value 'y' into L; hence, can be defined as

```
(27) append(y) == __iadd__(y)
```

The 'count' function counts the number of occurrences of 'y' in L; hence, can be defined as

```
(28) count(y) = #(enum(x,
                       0 ≤ index ≤ #(@(data, L)),
                       eINDEX(index, @(data, L)) = y))
```

The 'extend' function concatenates a list 'y' into L; hence, can be defined as

```
(29) extend(y) == __iadd__(y)
```

The 'index' function gives the position of 'y' in L; hence, can be defined as

```
(30) index(y) = IF(INDEX(y, @(data, L)),
                   INDEX(y, @(data, L)),
                   ValueError())
```

Given a value 'y' to be inserted before a 'position' in L, the 'insert' function can be defined as

```
(31) insert(position, y) =>
          @(data, L) =
               enum(x, 0 ≤ index < position,
                    eINDEX(index, @(data, L))) +
               enum(y) +
               enum(x, position ≤ index ≤ #(@(data, L)),
                    eINDEX(index, @(data, L)))
```

Given a 'value' to remove from L, the 'remove' function can be defined as

```
(32) remove(value) =>
          IF(INDEX(value, @(data, L),
             __delslice__(INDEX(value, @(data, L)),
                          INDEX(value, @(data, L))),
             ValueError())
```

The 'pop' function which removes the last element of L can be defined as

```
(33) pop() =
          IF(#(@(data, L)) > 0,
             $(eINDEX(#(@(data, L)), @(data, L)),
               remove(eINDEX(#(@(data, L)),@(data, L))),
             ValueError())
```

The 'pop' function may also remove the n-th element of L can be defined as

```
(34) pop(n) =
          IF(&(0 < #(@(data, L)), n ≤ #(@(data, L))),
             $(eINDEX(n, @(data, L)),
               remove(n, @(data, L))),
             ValueError())
```

The in-place 'reverse' function can be defined as

```
(35) reverse() =>
          @(data, L) = __reversed__()
```

The in-place 'sort' function can be defined as

```
(36) sort() =>
          @(data, L) =
               seq(x | memb(x, @(data, L)) |
                       INDEX(x, @(data, L)) <
                         INDEX(x+1, @(data, L)) )
```

3.2. Python Dictionary Operations

A hash table, known as dictionary in Python, can be considered as a bundle of 2 lists (defined in Section 3.1), a key list and a value list, with 2 constraints. Firstly, the number of elements in key list must match that of value list. Secondly, the elements in key list must not be duplicated. Therefore, 'key' and 'value' lists can be defined as

```
(1)  key = seq(ns())

(2)  value = seq(ns())

(3)  kv_constraint = &(#(key) = #(value))
```

A Python dictionary data type (D) can be defined as

```
(4)  D = $({(key, value) | kv_constraint},
          __cmp__, __contains__, __delattr__,
          __delitem__, __eq__, __ge__, __getattribute__,
          __getitem__, __gt__, __iter__, __le__,
          __len__,__lt__, __ne__, __setattr__,
          __setitem__,
          clear, copy, fromkeys, get, has_key, items,
          iteritems, iterkeys, itervalues, keys, pop,
          popitem, setdefault, update, values)
```

Given that 'y' is to be compared for equality to D, the comparator function (__cmp__) which checks for both the key and value lists can be defined as

```
(5)  __cmp__(y) =
          IF(&(@(__eq__(@(key, y)), @(key, D)),
               @(__eq__(@(value, y)), @(value, D))),
             0,
             IF(&(@(__gt__(@(key, y)), @(key, D)),
                  @(__gt__(@(value, y)), @(value, D)))
                1,
                -1)
            )
```

Given that 'y' is to be checked if it is found in the key list of D, the membership function (__contains__) can be defined as

```
(6)  __contains__(y) = @(__contains__(y), @(key, D))
```

Given that 'y' is the name of the attribute to be deleted from D, the delete attribute function (__delattr__) can be defined as

```
(7)  __delattr__(y) =>
          D = {x | memb(x, D) | x != y}
```

Given that 'y' is the key of the data to delete in D, the delete by index function (__delitem__) can be defined as

```
(8)  __delitem__(y)  =>
          seq(value_index = INDEX(y, @(key, D)),
              @(__delitem__(value_index), @(value, D)),
              @(key, D) = seq(x |
                              memb(x, @(key, D)) |
                              y != INDEX(x, @(data, L))))
```

Given the 'y' is to be checked for equality with D, the equality function (__eq__) which checks for both key and value lists can be defined as

```
(9)  __eq__(y) = IF(__cmp__(y) == 0, 'true', 'false')
```

Given the 'y' is to be checked for greater than with D, the greater than function (__gt__) which only checks key list can be defined as

```
(10) __gt__(y) = @(__gt__(@(key, y)), @(key, D))
```

Given the 'y' is to be checked for greater or equal than with D, the greater or equal than function (__ge__) which only checks key list can be defined as

```
(11) __ge__(y) = $(__eq__(y), __gt__(y))
```

Given that 'y' is the name of the attribute to access in D, the get attribute function (__getattribute__) can be defined as

```
(12) __getattribute__(y) = @(y, D)
```

Given that 'y' is the key of the value to access in D, the get item by index function (__getitem__) can be defined as

```
(13) __getitem__(y) = eINDEX(INDEX(y, @(key, D)),
                             @(value, D))
```

The iterator function (__iter__) presents an enumeration of the key list in D; hence, can be defined as

```
(14) __iter__() = enum(@(key, D))
```

Given the 'y' is to be checked for less than with D, the less than function (__lt__) which only checks key list can be defined as

```
(15) __lt__(y) = @(__lt__(@(key, y)), @(key, D))
```

Given the 'y' is to be checked for less or equal than with D, the less or equal than function (__le__) which only checks key list can be defined as

```
(16) __le__(y) = $(__eq__(y), __lt__(y))
```

The length function (__len__) which gives the number of data elements in D can be defined as

```
(17) __len__() = #(@(key, D))
```

Given that D is to be checked for non-equality to 'y', the nonequality function (__ne__) can be defined as

```
(18) __ne__(y) = ~(@(__eq__(y), @(key, D)))
```

Given that 'name' is the attribute to set as 'value', the set attribute function (__setattr__) can be defined as

```
(29) __setattr__(name, value) => @(name, L) = value
```

Given that 'y' is the position in L to set to 'value', the set item function (__setitem__) can be defined as

```
(30) __setitem__(y, value) =>
                __setattr__(eINDEX(y, @(key, D)), value)
```

The clear function removes all items from the dictionary.

```
(31) clear() => __delitem__(seq(@(key, D)))
```

The copy function is a shallow copy of the called dictionary.

```
(32) copy() => =(hV, D)
```

The items function returns a set of (key, value) pairs in the dictionary.

```
(33) items() => set((key, value),
                    @(key, D), @(value, D))
```

The fromkeys function creates another dictionary and copies each item in the called dictionary into the created dictionary. This is also known as deep copy.

```
(34) fromkeys = @(__setattr__(items(), D')
```

The get function returns the value of 'k' key. If 'k' key is not found, the function will return a default None value.

```
(35) get(k, default=ns()) => IF(memb(k, @(key, D)),
                                __getitem__(k),
                                default)
```

Given that 'y' is to be checked if it is found in the key list of D, the membership function (has_key) is equivalent to __contains__ function and can be defined as

```
(36) has_key(y) => __contains__(y)
```

The iterkeys function provides an iterative output over the keys in the dictionary.

```
(37) iterkeys() => seq(@(key, D))
```

The itervalues function provides an iterative output over the values in the dictionary.

```
(38) itervalues() => seq(@(value, D))
```

The iteritems function provides an iterative output over the (key, value) pairs in the dictionary.

```
(39) iteritems() => items()
```

The keys function returns a list of keys in the dictionary.

```
(40) keys() => seq(@(key, D))
```

Given a 'k' key, the pop function returns the corresponding value and deletes the key-value pair from the dictionary.

```
(41) pop(k) => seq(=(hV, get(k)),
                   __delitem__(k))
```

The pop item function returns the random key-value pair and deletes the key-value pair from the dictionary.

```
(42) popitem() => seq(rand(=(k, @(key, D))),
                      =(hV, (k, get(k)
                      __delitem__(k))
```

The get function returns the value of 'k' key. If 'k' key is not found, the function will return a default None value and set the 'k' key to the given default value or None.

```
(43) setdefault(k, default=ns()) =>
                          seq(get(k, default),
                              IF(~(has_key(k)),
                                 __setattr__(k, default),)
```

The dictionary can be updated with E and F using the update function.

```
(44) update(E, F) => seq(__setitem__(G), seq(E), seq(F))
```

The values function returns a list of values in the dictionary.

```
(45) values() => seq(@(value, D))
```

4. Concluding Remarks and Future Work

The beauty of mathematics lies in its precision. However, it is this very precision that makes mathematics fearful as there is no veil of ambiguity, yet it is required to define the behaviour of a system. To make things worse, mathematical notations are usually terse; thus, tend to be tougher to read than prose. Despite the necessity of mathematics, an appropriate choice of notations may go a long way in reducing the learning curve (Bowen et al., 2005). This study describes a formal behavioural specification language, BeSSY, which uses only 4 areas of mathematics, namely, sets theory, functions, arithmetic and Boolean operations. We believe that these 4 areas should be familiar to most, if not all, software engineers and developers as they lie at the fundamentals of computer science. Arithmetic and Boolean operations and functions are both fundamental concepts in high-school mathematics as well as procedural programming languages. Set theory is fundamental for relational database management systems; hence, should be versed by the time one completes first year undergraduate computer science curriculum.

The Python programming language defined 4 primary data structures (van Rossum, 2008), namely, list, dictionary, sets and tuple. Of these, list and dictionary are more commonly used than sets and tuple. Tuple is an immutable list while the functions of sets can be simulated using dictionary keys. Hence, only list and dictionary were chosen for specification in BeSSY. As a litmus test, this suggests that BeSSY is likely to be sufficiently rich to describe most Python programs.

Although mathematical simplicity is a major design criterion for BeSSY, are there certain things that are impossible to express in BeSSY despite demonstration of Python list and dictionary operations in this study? Expressiveness is an important criteria in evaluating formal languages (Ruiz et al., 1994, van Harmelen et al., 1993, van Harmelen et al., 1996). We used Turing completeness as a test for expressiveness. Brainfuck is an esoteric programming language invented by Urban Muller aiming to create a Turing-complete language. There have been a number of Brainfuck interpreter implemented using Python programming language and Python list as the storage array (McGugan, 2005, Lindstrom, 2007). In addition, Frans Fasse had provided several proofs of Brainfuck's Turing-completeness, including implementation of each Turing operations in Brainfuck (http://www.iwriteiam.nl/Ha_bf_Turing.html). As Python list operations was used to implement an interpreter for a Turing-complete language; thus, Python list operations are Turing-complete by extension. Since Python list operations can be fully expressed in BeSSY (as shown in Section 3.1), suggesting that BeSSY is Turing-complete; thus, sufficiently expressive to express all computable operations.

BeSSY has a few significant limitations. Although BeSSY is Turing-complete, it does not imply ease of expression. At the same time, there is no tool support for BeSSY as it is a newly created formal language. However, the specification of Python list and dictionary data types in BeSSY suggests future possibility in developing tools that may assist in the conversion of BeSSY specifications into Python codes which is likely to improve the employment of formal languages in Python software development (Bowen et al., 2005).

5. References

Berry, G. 2008. Synchronous design and verification of critical embedded systems using SCADE and Esterel. Proceedings of the Formal Methods for Industrial Critical] Systems. Lecture Notes in Computer Science 4916. Springer-Verlag.

Bowen, Jonathan P. and Hinchey, Michael G. (1995) Seven more myths of formal methods. IEEE Software 12(4):56.

Bowen, Jonathan P. and Hinchey, Michael G. (2005) Ten commandments revisited: a ten-year perspective on the industrial application of formal methods. Proceedings of the 10th international workshop on formal methods for industrial critical systems.

Chiang, Chia-Chu. (2004) Teaching a formal method in a software engineering course. Proceedings of the 2nd annual conference on Mid-south college computing.

Ciapesson, Emanuele, Coen-Porisini, Alberto, Crivelli, Ernani, Mandrioli, Dino, Mirandola, Piergiorgia, Morzenti, Angelo. (2002) From formal models to formally-based methods: an industrial experience. ACM Transaction on Software Engineering and Methodolgy 8:79

Clarke, Edmund M., Wing, Jeannette et al. (1996) Formal methods: state of the art and future directions. ACM Computing Surveys 28:4.

Hinchey, Michael G. and Bowen, Jonathan P. (ed). (1999) Industrial-strength formal methods in practice. Springer-Verlag FACIT Series, London.

Leroy, Xavier and Blazy, Sandrine. (2008) Formal verification of a C-like memory model and its use for verifying program transformations. Journal of Automated Reasoning 41:1

Lindstrom, E. 2007. Yet another brainfuck interpreter. [*http://www.uselesspython.com/download.php?script_id=202]*

McGugan, W. 2005. Brainf***. [*http://www.uselesspython.com/download.php?script_id=20]*

O'Leary, John, Zhao, Xudong, Gerth, Rob, Geger, Carl-Johan H. (1999) Formally verifying IEEE compliance of floating-point hardware. Intel Technology Journal Q1'99

Palmer, Thomas V. and Pleasant, James C. (1995) Attitudes towards the teaching of formal methods of software development in the undergraduate computer science curriculum: a survey. ACM SIGCSE Bulletin 27:3.

Rossi, Matteo and Mandrioli, Dino. (2004) A formal approach for modeling and verification of RTCORBA-based applications. ACM SIGSOFT Software Engineering Notes 29: 263.

Ruiz, F., van Harmelen, F., Aben, M., van de Plassche, J. 1994. Evaluating a formal modelling language. In: Steels, Schreiber and van de Welde (eds). Proceedings of the 8th European Knowledge Acquisition Workshop (EKAW'94). Lecture Notes in Artificial Intelligence 867: 26-45. Springer Verlag.

Russinoff, David M. (1998) A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7(tm) processor. LMS Journal of Computation and Mathematics 1: 148

Sharpe, R. (2004) Formal methods start to add up again. Computing, 301. [http://www.computing.co.uk/computing/features/2072361/formal-methods-start-add-again]

Sobel, Ann E. Kelly. (1996) Experience integrating a formal method into a software engineering course. ACM SIGCSE Bulletin 28:271

Sobel, Ann E. Kelly and Clarkson, Michael R. 2002. Formal methods application: an empirical tale of software development. IEEE Transactions on software engineering 28:3

Spivey, J. Michael. 1992. The Z notation: a reference manual. Prentice Hall International Series in Computer Science.

van Harmelen, F., de Mantaras, RL., Malec, J., Treur, J. 1993. Comparing formal specification languages for complex reasoning systems. In Treur, J., Wetter, T. (eds). Formal specification of complex reasoning systems. Ellis Horwood.

van Harmelen, F., Aben, M., Ruiz, F., van de Plassche, J. 1996. Evaluating a formal KBS specification language. IEEE Expert 11(1): 56-62.

van Rossum, G. 2008. Python tutorial, release 2.5.4. [http://www.python.org/doc/2.5.4/tut/tut.html]

Woodcock, J., Larson, PG., Bicarregui, J., Fitzgerald, J. 2009. Formal methods: Practice and experience. ACM Computing Surveys 41(4): Article 19.

Appendix I: Brainfuck interpreter by Edvin Lindstrm

```
#! /usr/bin/env python
# -*- coding: utf-8 -*-
#brainfuck.py
#Brainfuck interpreter
#By Edvin Lindström
#Started 2007-02-26
#Latest revision 2007-02-26

import sys

def main():
    mem = [0] * 30000       #Memory "array"
    c = 0                   #Counter ("pointer")
    source = raw_input("Complete path of brainfuck source file: ")
    try:
        f = open(source, 'r')
    except IOError:
        print "Error while trying to open file %s" % source
        return False        #Quit program execution
    else:
        print "File opened."
    debug = ''
    while debug != 'y' and debug != 'n':
        debug = raw_input("Debug mode? (y/n) ")
    if debug == 'y':
        debug = True
    elif debug == 'n':
        debug = False

    try:
        program = f.read()
    except MemoryError:
        print "Memory error occurred while reading file %s" % source
        return False        #Quit program execution
    else:
        f.close()
        print "File data read."
        print "Executing program.\n-----"

    loops = 0               #The current byte in the file

    #Main loop
    while True:
        byte = program[loops]   #Read the current byte
        if debug:
            print "BF byte %d" % loops

        #Fix out-of-range problems
        if c < 0:
            c = 0
        elif c > 29999:
            c = 29999
```

```
#Do what's supposed to be done
if byte == '>':
    c += 1
elif byte == '<':
    c -= 1
elif byte == '+':
    mem[c] += 1
elif byte == '-':
    mem[c] -= 1
elif byte == '.':
    if mem[c] in range(256):
        sys.stdout.write(chr(mem[c]))
elif byte == ',':
    while True:
        inp = raw_input()
        #If more than one byte is entered only the first is
        #accepted
        if len(inp) > 1:
            inp = inp[0]
        #This catches input shortened by the previous if too
        if len(inp) == 1:
            mem[c] = ord(inp)
            break
        #If nothing was entered the while loop starts over
elif byte == '[':
    if mem[c] == 0:
        leftps = 0      #Left parentheses found
        rghtps = 0      #Right parentheses found
        bt = loops
        while True:
            bt += 1     #Move to the next byte
            if program[bt] == '[':
                leftps += 1
            elif program[bt] == ']':
                if leftps == rghtps:
                    loops = bt      #Move to the right bracket
                    break   #Incrementing loops is carried out
                            #later
                else:
                    rghtps += 1
elif byte == ']':
    leftps = 0          #Left parentheses found
    rghtps = 0          #Right parentheses found
    bt = loops
    while True:
        bt -= 1         #Move to the previous byte
        if program[bt] == ']':
            rghtps += 1
        elif program[bt] == '[':
            if leftps == rghtps:
                loops = bt - 1  #Because later loops will be
                                #incremented again
                break
            else:
                leftps += 1

if debug:
```

```
            print "   mem[%d] = %d" % (c, mem[c])

        if loops < len(program) - 1:
            loops += 1
        else:
            return True         #Quit program execution

if __name__ == '__main__':
    a = main()
    if a == True:
        print "\n-----"
        print "Execution terminated properly."
    elif a == False:
        print "\n-----"
        print "Execution unexpectedly terminated."
```