The Symlah language

Designed and developed by Mohamed Shaafiee
For additional help please email shaafiee@gmail.com

Although Symlah is meant to be a visual development tool, the underlying language which is to be utilized by this tool is a lexical one. This is to conform with the data abstraction lineage of software layers in a computer system as well as to ensure development capability in non-graphical systems.

A program written in Symlah is not compiled but interpreted by a virtual machine (VM), similar to the Java VM (Flanagan 2005). A user wishing to run a Symlah program will have to connect to the Symlah VM via a TCP socket (the default being 10115) and use the Symlah VM Protocol (SVMP) to execute the program.

The lexical implementation of Symlah has lexical symbols which is not be used in any form of permutation. A simple example of a lexical permutation in Perl is 'my $somevariable'. Here, the keyword 'my' could have been replaced with 'our' or omitted completely (depending on the desired effect). Symlah does not use such permutations to define clarifications, classifications and/or specifications. Instead, where such clarifications, classifications and/or specifications are required, a strictly functional approach is sought.

The control flow of a Symlah program is sequential and single-threaded, unless a function call is made to a another function. Each program consists of a sequence of functions, with the function at the head of the source code being the entry function and the one at the end being the exit function. The data passed to the entry function flows from it to the subsequent functions until or unless this data stream is modified by any of the functions.

The Symlah program's interaction with the environment/user is strictly through the Symlah VM. The way this works is the user connects to the Symlah VM via a TCP socket (using telnet for instance) and executes the program by passing the appropriate commands and the relevant data.

The most unique feature of the Symlah language is that it does not have any functions to store and/or retrieve data. That is, the programmer need not be concerned with data storage operations. Instead, depending on the operations on data symbols the Symlah VM determines data table creation prior to running the program. Data storage is automatically done for all data elements in the data stream.

Symlah VM (SVM)

The Symlah VM (SVM) interprets a Symlah program by reading the source code, creating a symbol table, preparing data table and sequentially executing the functions in the program. Since the Symlah language does not have permutable lexemes and follows functional programming rules, the SVM does not need to do any prior lexical analysis.

The SVM's source code has been included in appendix Error: Reference source not found. The main program for SVM is 'svm.pl'. This program is currently meant to be run on only Unix systems with Perl. When executed, the 'svm' program will look for the configuration file '/etc/svm.conf'. The program will exit with a warning if this configuration file is not found. If the '/etc/svm.conf' configuration file is found, then the SVM environment is initialized in accordance with the configuration settings. The configuration settings which can be specified in the configuration file are:

- svm_dir <directory>: where <directory> can be replaced to indicate the service directory in which all the Symlah program source codes are located (for instance, '/symlah/programs').
- svm_port <port>: where <port> can be replaced to indicate the desired TCP port on which the SVM should be listening (if this directive is not declared in the configuration file, the default port 10115 is used).
- svm_data <directory>: where <directory> can be replaced with the location for data storage (for instance, '/symlah/data').

It is important to know the file naming convention defined in Symlah for naming its programs. Although Symlah is developed on a Unix platform it does not use all the possible permutations of characters used for Unix file names. The following BNF notations define Symlah's file naming convention:

⟨filename⟩ ::= ⟨letter⟩ | ⟨filename⟩ ⟨letter⟩ | ⟨filename⟩ ⟨digit⟩

⟨letter⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | Y | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Symlah VM Protocol (SVMP)

The protocol for communicating with the Symlah VM has been given the acronym SVMP. SVMP is a set of rules which specify the way a user communicates with the SVM in order to run a program. The following are the keywords used in the protocol and their rules:

- run <program> <input_data>: where <program> can be replaced with a reference to the program needed to be run. This reference is relative to the SVM service directory (ie. the reference '/somedir/someprogram' is a reference to '/symlah/programs/somedir/someprogram' if the service directory has been defined as '/symlah/programs'). The format of the text which should replace <input_data> is explained in the following subsection.
- quit: disconnect from the SVM.

The format of <input_data>

The <input_data> in the 'run' keyword is replaced with data to be passed to the program being run. The format of <input_data> is described by the following BNF notations:

⟨input_data⟩ ::= ⟨input_data⟩ “|” ⟨data_group⟩ | ⟨input_data⟩ “|” ⟨data_element⟩ “(“ ⟨data_group⟩ “)” | ⟨input_data⟩ “|” ⟨data_assoc⟩ “(“ ⟨data_group⟩ “)”

⟨data_group⟩ ::= ⟨data_assoc⟩ | ⟨data_group⟩ “|” ⟨data_assoc⟩

⟨data_assoc⟩ ::= ⟨data_element⟩ “=” ⟨data_element⟩

⟨data_element⟩ ::= ⟨data_element⟩ ⟨letter⟩ | ⟨data_element⟩ ⟨digit⟩ | ⟨data_element⟩ ⟨letter⟩ ⟨symbol⟩ | ⟨data_element⟩ ⟨digit⟩ ⟨symbol⟩

⟨letter⟩ ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | Y | W | X | Y | Z | a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u | v | w | x | y | z

⟨digit⟩ ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

⟨symbol⟩ ::= “ “ | , | “ | ' | - | + | * | / | = | (|) | { | } | [|]

The above BNF dictates that the following text for <input_data> are valid:

- var1=hello|var2=my world|var1=love me
- divine=so(sym=do|sym=do not)|world=chapatti(life=live|death=live not)|true=something|simple=rotti(say=hello|say(farewell=goodbye))

The format of <input_data> also holds true for all data output by a program. In the above statements, where key-value pairs have been grouped using parentheses, the key-value pair immediately preceding the opening parenthesis dictates that the grouping is associated to that key, when the indicated value occurs (the grouping will not be associated to other values in the same key).

Grouping key-value pairs can also tell Symlah the relationship between the keys. For instance, in order to tell Symlah that all values of 'wheel' is associated to each and every value of 'motorcycle' the following input data can be used:

- motorcycle(wheel=front|wheel=back)

In the above case an empty 'motorcycle' variable will be created with the variable linked to another variable called 'wheel', which has two values, 'front' and 'back'. It is possible to define associations with multiple variables as thus:

- motorcycle(wheel=front|wheel=back|engine=125cc)

Associations are dissected further later in section .

Syntax and semantics of Symlah

Functions and program flow

Given the precursors established thus far, Symlah's design stems from a linear data flow sequence with an entry function and an exit function. Additionally, due to the way in which the <input_data> format was defined in the previous section, there is no need to specify any variable names for the data passed to the program. This is because all data passed to the program has to be formatted as key-value pairs. Where two key-value pairs have the same key name, Symlah creates an array (see section) with a corresponding variable name.

Function declaration is simple too. Symlah uses the symbols '<' and '>' (angular brackets) to demarcate a function. Within these angular brackets any number of space-delimited literals can be written, the first one being the keyword indicating the operation required to be performed and the subsequent ones being the arguments. Hence three consecutive functions will look like this:

```
< <name_function_1> <arguments> >
< <name_function_2> <arguments> >
< <name_function_3> <arguments> >
```

Listing 1

Although the program flow is primarily sequential, the Symlah design accedes to calls to functions. When such a call is made, then the control flow is altered from the next function to the specified one. This means that a function can optionally be given a name. Function names are declared by prefixing an alphanumeric literal before the opening angular bracket of a function. A sequence of five functions in Symlah can look like this:

```
adder< <name_function_1> <arguments> >
< <name_function_2> <arguments> >
divider< <name_function_3> <arguments> >
< <name_function_4> <arguments> >
result[3]< <name_function_5> <arguments> >
```

Listing 2

In the above code, only functions 1, 3 and 5 are given names. Implicitly, it is impossible to alter the flow by calling functions 2, or 4 because neither of these functions have been given a unique name. Hence, if the program flow is altered at function 4 by calling 'adder', the program executes function 1 and sequentially executes functions 2 through 5 thereafter. Though function 5's name is strangely augmented with a number within brackets, it too can be called via the label 'result'. The number within brackets only specify an index location of a variable (see section).

Although Symlah uses the phrase 'function' to describe each statement demarcated by angular brackets, none of these functions perform more than one operation. That is, each function has a keyword that specifies what its task is, along with the specification of the behaviour of that task. For instance, the keyword 'add' is used to add the content of two or more variables (as are keywords 'multiply', 'divide', 'subtract' and 'modulus' for their respective implied operations). An example of the use of the addition function is:

```
adder<add number1 number2 number3>
```

Listing 3

The 'add' function accepts any number of arguments as evident from the above statement (which holds true for 'multiply', 'divide', 'subtract' and 'modulus'). The result yielded from the operation in Listing 3 is stored in a variable created by Symlah. This variable is given the same name as the function's. Hence, in order to add the values of the three variables along with the present value of 'adder' the following function can be used:

```
adder<add number1 number2 number3 adder>
```

Listing 4

If the above 'adder' function is called more than once then these results are stored in the 'adder' variable in the respective index locations. That is, the result from the first call already having been stored in the first index location, the next two will be stored in locations 2 and 3, respectively.

The functional style of Symlah makes the building of mathematical equations more tedious (at least in its lexical implementation) than in other programming languages. An example follows:

```
add<add number1 number2 number3>
multiply<multiply add number4>
divide<divide multiply number5>
```

Listing 5

In Perl, this same equation can be represented as thus:

```
$add = ( ( $number1 + $number2 + $number3 ) * $number4 ) / $number5
```

Listing 6

The tedium of coding in Symlah need not be regarded as a flaw as avoiding mutable functions (hence, each function becomes symbolic of a single actual process) ensures proper set associations when developing the visual application development tool.

Non-mutable functions imply that writing highly iterative code with several independent calls to a single group of functions would mean repeatedly writing these functions at the desired locations. This is avoided in Symlah by allowing programs to be able to call other programs. That is, a function group which is repeatedly accessed by a program can be placed in a second program with a valid file name, which is accessed from the first. The function that facilitates such an inter-program call is 'linkif', exemplified by the following statement:

```
<linkif '1' 'factorial'>
```

Listing 7

In the above line of code, the 'linkif' function passes the control flow from the current program to a program saved with the file name 'factorial'. The literal '1' is passed to the 'linkif' function to specify the current condition: with '1' denoting a logical true (see section). When the 'factorial' program's exit function is completed then Symlah takes the data flowing out of the 'factorial' program and adds it to the data flow of the current program.

It has been mentioned earlier that within a program the flow control can be altered by specifically calling a function by its name. As with all other processes in Symlah, this operation is also facilitated by a single function (see section for the clarification of '1' in the 'callif' function):

```
adder<add number1 number2>
adder_returned<callif '1' adder>
```

Listing 8

When a call is made to another function by a 'callif' or 'linkif' function, then when the program flow comes back to the caller the program flow moves on to the function immediately following the function that initiated the call. It should be noted that there is no data scoping in Symlah, and as such all data variables which have been defined thus far in the program are available to all subsequent functions in the program flow. This is regardless of whether the function is one which has already been surpassed in the sequence of program flow (and is now being explicitly called by the current function).

How Symlah handles calls to programs

Symlah handles calls to other programs by opening a socket to the Symlah VM and then using SVMP to incur the requested program. In this process, whatever data has been in the data flow of the calling program is also passed to the program being called. When the latter program completes its processing and returns the program flow to the caller, any additional or altered data returned is added or adjusted in the caller's data flow.

See section for more on conditional calls.

Variables (and arrays)

Variable names in Symlah are case-insensitive alphanumeric (no other characters are allowed) literals which cannot exceed a total of 30 characters. The reason for limiting the length of variable names is in part to save disk space when storing them. Symlah's variables are not typed and do not require exclusive casting. All necessary type-casting is performed automatically by the Symlah VM.

Arrays in Symlah are variables with multiple values. Each value is given an index number and can be referred to by specifying this number within brackets appended to the variable name. The first value of an array variable has the index number 1 (this is to avoid ambiguity when Symlah is being used by people who are not very familiar with programming). The following specifies the 3rd value of a multi-valued variable:

```
somevariable[3]
```

Listing 9.

A specific set of values from within a multi-valued variable can also be specified in Symlah. A set can be defined as a sequential range or non-sequential comma-delimited individual indices. Examples of this are:

```
somevariable[3 to 5]
somevariable[4, 8, 22]
```

Listing 10

The range description for a list of values can be augmented further with a 'step' postfix (which is valid only for a ranged array description). This tells Symlah which of the numbers in the range are relevant. Consider the following examples:

```
somevariable[3 to 10 step 2]
somevariable[10 to 5 step -1]
```

Listing 11

In the first of the two statements above, the range description tells Symlah that the relevant numbers from the range are 3, 5, 7 and 9 (because, after each number Symlah is told to take a step up of 2). In the second range description Symlah is told to take a step of -1 (a step down) after each number (the relevant numbers being 10 through 5, in that order).

Developers can also use variables in place of static numbers in their range descriptions. Consider the following statement:

```
somevariable[anothervariable]
```

Listing 12

If the content of 'anothervariable' at the juncture of the above statement contained the values 3, 5, 7, and 9, the statement is exactly equivalent to the first statement in Listing 11. Now consider the following statement:

```
somevariable[first to second step third]
```

Listing 13

If at the juncture of the above statement the first value of 'first', 'second' and 'third' held the values 10, 5 and -1, respectively, then the statement enacts the same functionality as the second statement in Listing 11.

Symlah provides a function which can find the total number of values in a given variable (in essence, the index of the last value). This function is given the keyword 'count' and is exemplified below:

```
lastElement<count somevariable>
```

Listing 14

The above code finds the total number of values in the variable 'somevariable' and stores it in a variable named 'lastElement' (corresponding to the name given to the function).

As evident thus far, Symlah makes a departure from the conventional implementation of variables in programming languages by combining scalar variables and arrays into one data type. That is, any variable can be specified with an index number regardless of whether prior specifications did not use indexing. Hence, a hitherto undefined variable 'somevariable' can be assigned a value without specifying an index. Such an assignment operation will make Symlah store the value in the first (and, at the time, the only) index location of the variable. Later the value can be accessed by using either of the following:

```
1. somevariable
2. somevariable[1]
```

Listing 15

The unification of scalar variables and arrays are further enhanced by enabling programmers to implicitly specify all values in an array by omitting the index. Hence, if 'somevariable' has 4 values,

then all 4 values (or, depending on the modus operandi of a given function, the first value only) are specified by the following:

```
somevariable
```

Listing 16

Data assignment

Instead of an operator, Symlah uses a function to assign values to variables. This function has the keyword 'assign' and is exemplified below:

```
<assign somevariable 'hello world'>
```

Listing 17

The single-quotes in the above statement signifies that the phrase within is to be interpreted as a literal (i.e. it is not a keyword or a name of a variable). There is no distinction between strings and numbers in Symlah, either of which will need to be written within single-quotes to be regarded as literals.

The values of other variables can also be stored in a given variable using the 'assign' function. For instance:

```
<assign somevariable anothervariable hisvariable hervariable>
```

Listing 18

In the above statement the values of variables 'anothervariable', 'hisvariable' and 'hervariable' are assigned to 'somevariable'. Symlah accomplishes this by appending the values in the three variables at the end of 'somevariable' in order, much like a queue. Specific locations for storage can also be indicated as follows:

```
<assign somevariable[4 to 6] anothervariable[1] hisvariable[3] hervariable[7]>
<assign somevariable[4, 5, 6] anothervariable[2] hisvariable[4] hervariable[13]>
```

Listing 19

It should be noted that the cardinality of indices to the number of variables should reciprocate, as in the above two lines. Otherwise, Symlah will stop the program and return an error.

Symlah provides the 'discard' function for clearing all or some of the values in a variable. The 'discard' function is demonstrated below:

```
<discard somevariable>
<discard somevariable[5]>
<discard somevariable[2 to 4]>
<discard somevariable[1,3,7,20]>
```

Listing 20

Determining output scope

Unless an output scope is defined SVM will not output any data. Definition of an output scope is done using the 'output' keyword. To specify that the variables 'somevariable' and 'anothervariable' is to be output, the 'output' keyword can be used at any point of the Symlah program with the following syntax:

```
<output somevariable anothervariable>
```

Listing 21

Dereferencing a variable

In Symlah it is possible to acquire the address of a variable from the value of another variable. The function keyword assigned for this is 'deref'. Similar to C or Perl, this type of referencing in Symlah creates a pointer to a variable. Consider the following example:

```
<assign aloha 'hello'>
<assign somevariable 'aloha'>
anothervariable<deref somevariable>
```

Listing 22

In the above example, the address of the variable 'aloha' is copied into 'anothervariable'. This is done by first looking for the variable with the name corresponding to the first value of 'somevariable' and then getting its address. If 'somevariable' had more than one value then the 'deref' function will require specifying an index after the variable name, or it will use the value in the first index location by default. The following statement shows index-specific dereferencing:

```
anothervariable<deref somevariable[23]>
```

Listing 23

Referencing works for literals which define referrals to data associations too (section). The literal 'somevariable.another' can be referenced using:

```
anothervariable<deref 'somevariable.another'>
```

Listing 24

Once a variable has been assigned the address of another variable using the 'deref' function then manipulation of the former only manipulates the data in the latter. Hence, in the case of Listing 22, if a value were assigned to 'anothervariable' after the 'deref' statement, then the value will actually be assigned to 'somevariable'.

Data associations

Associations are the only forms of data structures in Symlah. An association can be made between two or more variables with 3 different types of relationships. The number of relationship types were not determined from any surveys but from currently established norms in leading programming languages. Specifically, the three basic Ruby on Rails relationship types were implemented (Fernandez 2007). This design decision is based on the wide acceptance of Ruby on Rails' CRUD (create/read/update/delete) application principle (Fernandez 2007).

MANY-MANY association

A parent variable is associated with the child variable without indicating any indices on either. This means that any given value in the parent variable is associated to all the values of the child variable. This is a MANY-MANY relationship.

The basic syntax for the function call to make any type of association is:

```
<associate <parent_variable> <children> >
```

Listing 25

In the above line, <parent_variable> will need to be replaced with the name of the parent variable and <children> with one or more variable names (separated by spaces). Hence, to create a relationship between the variables 'motorbike' and 'wheel':

```
<associate motorbike wheel>
```

Listing 26

The following example illustrates how to create a relationship between the variable 'motorbike' and more than two children:

```
<associate motorbike wheel gasTank>
```

Listing 27

ONE-MANY association

A ONE-MANY relationship is made when a single value of a parent variable is associated with all the values of a child variable by indicating an index location in the variable specification for the former and not the latter.

Examples of function calls for ONE-MANY associations are:

```
<associate somevariable[3] variable1 variable2>
<associate car[4] wheel frame engine gearbox body>
```

Listing 28

It is possible to create a ONE-MANY relationship between a parent value and a limited range of or list of values in a child. This is illustrated in the following statement:

```
<associate car[4] wheel frame[1,3,7] engine[3 to 5] gearbox body>
```

Listing 29

ONE-ONE association

A single value of a parent variable can be associated with a single value of a child variable by indicating indices in both cases, in reciprocating cardinality. This is a ONE-ONE relationship. The function call that creates ONE-ONE relationships can associate a single parent index to a single child index or a number of indices (a range or set) of the parent variable to the same number of indices in the child variable (although not the exact same index locations). An example of this function call is:

```
<associate somevariable[1 to 3] anothervariable[5,9,20]>
```

Listing 30

Overlapping relationships

Symlah allows some overlapping associations. If a MANY-MANY relationship has been made between a parent variable and a set of child variables, it is still possible to create ONE-ONE relationships between the same variables.

Why ONE-ONE relationships are allowed over an existing MANY-MANY relationship is in order to associate a specific value of a parent variable with a specific value of a child variable. In such a case, a reference to the relationship via the parent's relevant value will yield the related value from the child, and not the set of all values in the child (the latter being the case for references via all other values in the parent variable). Where variables share MANY-MANY as well as ONE-ONE relationships, if a value involved in the latter relationship is referred then the MANY-MANY relationship will be superseded by the ONE-ONE relationship.

It is also possible to create many ONE-ONE relationships between the same value in a parent variable and different values in a child variable. In such a case, when the second ONE-ONE association is made, then Symlah takes the previous ONE-ONE relationship, and consolidates both into a ONE-MANY relationship. If a third ONE-ONE relationship is made between the same value of the parent and a third value of the child, then that one also is added to the ONE-MANY relationship.

The final overlapping relationship Symlah allows is the range-limited ONE-MANY relationship over a MANY-MANY relationship. That is:

```
<associate college lecturer student>
<associate college[1] lecturer[3,5,7] student[5 to 30]>
```

Listing 31

However, Symlah does not allow creating ONE-MANY relationships, encompassing all values in a child variable, over existing MANY-MANY relationships. Hence, the following association will yield an error:

```
<associate college lecturer student>
<associate college[1] lecturer student[5 to 30]>
```

Listing 32

Dissociating relationships

Another major difference between Symlah and other languages (apart from Ruby on Rails) is that once an association type is made between two variables, the association is permanent until the relevant dissociation functions are called. This means that the association will not be lost even after the program has exited.

The following statements exemplify Symlah's dissociation functions and the relevant syntax:

1. MANY-MANY: <dissociate somevariable anothervariable>
2. ONE-MANY: <dissociate somevariable[4] anothervariable>
3. ONE-ONE: <dissociate somevariable[5,9,11] anothervariable[1 to 3]>

An important rule to keep in mind is that when the dissociation function is called to annul a MANY-MANY relationship, Symlah destroys all other relationships between the two variables. Similarly, if a ONE-MANY relationship is annulled then all ONE-ONE relationships that fall within its umbra are also destroyed.

Listing associations

Symlah provides the 'relations' function to discover all the associations made from a given variable to others. The following statement illustrates the use of the 'relations' function:

```
relations<relations somevariable>
```

Listing 33

The above statement will cause Symlah to list all the associations made with 'somevariable' and store it in the 'relations' variable (as a list of values). Consider the following code:

```
<assign somevariable 'hello'>
<assign anothervariable 'one' 'two'>
<associate somevariable anothervariable>
somevariableRelations<relations somevariable>
```

Listing 34

When the above code completes execution, at the end of the last function a variable called 'somevariableRelations' will be added to the data stream. Stored in index 1 of 'somevariableRelations' will be the value 'somevariable.anothervariable' in the first index location (as 'anothervariable' is the only variable to which 'somevariable' is associated to).

If two variables share more than one type of relationship between them, then each association which creates these are returned by the 'relations' function. Consider the following code:

```
<assign somevariable 'alpha' 'beta' 'gamma'>
<assign anothervariable 'one' 'two' 'three'>
<associate somevariable anothervariable>
<associate somevariable[2] anothervariable[3]>
allrelations<relations somevariable>
```

Listing 35

The last function in the above code will populate the variable 'allrelations' with the values 'somevariable.anothervariable' and 'somevariable[2].anothervariable[3]'.

Referring associated variables

Consider the following code:

```
<assign college 'APIIT' 'UPM' 'HELP'>
<assign student 'Joe' 'James' Javid' 'John' 'Johan'>
<assign lecturer 'GEETHA' 'SAMANTHA' 'KWAN' 'JAGATHA' 'JAYANTHA'>
<associate college[1] student[3] lecturer[1,3]>
<associate college[2] college[1].student lecturer[2,5]>
```

Listing 36

The last line in the above code illustrates how references can be used as variables in functions. In this particular line the 2nd value of 'college' is associated with all values of 'student' which have been associated with the 1st value of college. Thus the resulting association will be 'college[2].college[1].student'. Hence, references to associated variables in Symlah are similar to references to abstract objects in Java (Flanagan 2005).

Decision-making

The cornerstone of Symlah's decision-making is the axiom that any variable that does not contain a value constitutes a logical false state. Unlike in Perl, the non-existence of a variable does not make the array index '-1'. Instead, the emptiness of a variable (or specific index locations of one) is indicative of a boolean false. Hence, the test to see non-existence requires the following function call:

```
emptiness<not <some_variable> >
```

Listing 37

Since the emptiness of <some_variable> indicates a logical false, the 'not' function inverts this

boolean flag and stores a '1' as the first value of a variable 'emptiness'. If the given variable has more than one value, then all values are inverted and stored in the variable 'emptiness' at the reciprocating indices. If any value in a given variable is not empty, then the respective index location of 'emptiness' will be stored with nothing.

If only a single variable is passed to either the 'and' or 'or' function, then the respective logical assessment is made between all the values in that variable. However, if two or more variables, each with more than one index, are passed then all values in the corresponding indices will be operated on and all the results of these logical operations will then be stored in the respective index locations of a variable corresponding to the function name. For instance, assume that the variables being passed to the 'and' function in the following statement have multiple indices:

```
opResult<and somevariable anothervariable>
```

Listing 38

The above 'and' function will perform a logical AND between all the values in the first index locations of the two variables. Then it will perform a logical AND between the values in the second index locations, and so on. Given that Symlah considers emptiness as a boolean false, if any given index location does not have any value, then logical AND should yield a boolean false even if the value in the reciprocating index location of the other variable contained a value. All the results of these logical operations will be stored in the reciprocating index locations of 'opResult'.

Symlah allows passing index-specific variable constructs to the logical functions. This allows the programmer to match a value in a specific location of one variable with a specific value stored in another. This also means that if a specific index location for one variable is specified then the same should be true for all other variables passed to the logical operation function. The following exemplifies this:

```
opResult<and somevariable[2] anothervariable[10] yetanothervariable[13]>
```

Listing 39

Similar to arithmetic operations, creating complicated logical evaluation statements in Symlah is tedious. This tedium is exemplified in the following statement:

```
allVisitors<and visitor1 visitor2 visitor3>
visitors2and3<and visitor2 visitor3>
notVisitors2and3<not visitors2and3>
onlyVisitor1<and visitor1 notVisitors2and3>
<callif noMoreVisitors allVisitors>
```

Listing 40

Comparative operators

Logical operators can evaluate only the truth or false conditions depicted by variables. They do not provide any means of comparing two different variables. The functions which provide this capability in Symlah are the 'equal', 'greater', 'less', 'lessorequal' or 'greaterorequal' functions.

These comparative operators do not perform just a comparison of two values in any two variables. Instead, they perform a comparison between the first value of the first variable with each of the values in the second variable. Consider the following statement:

```
results<equal first second>
```

Listing 41

What the above code will accomplish is cycle through the values of the 'second' variable, comparing each of these values with the value in the first index location of the 'first' variable (even if the latter had more than one value). If any of these comparisons find an equal match then the name of the second variable and the index location of the matching value is stored in 'results'. For instance, if the value in the fifth index location of the 'second' variable matched the first value of the 'first' variable, then 'results' will be appended with the value 'second[5]'.

Conditional calls to functions and other programs

The last statement in Listing 40 exemplifies the conditional call function, which uses the keyword 'callif'. The 'callif' function accepts two arguments, the first being a variable name (or a literal) and the second the name of the function to be called. The 'callif' function will make the call to the specified function only if the value of the variable defined by the first argument is not empty (which translates to a logical true).

When a specific function is called using 'callif' then this function call is stored in an internal stack which keeps track of the last caller (the last call always overwrites this stack in order to avoid

potential stack overflow issues). The program will commence from the function being called moving on to the one after that in the program flow. The only way to return to the original caller is through the use of the 'return' function. Hence, the following code assigns only the value 'Symlah' to the variable 'name' (see section for an explanation on the 'exit' function):

```
<callif '1' 'assignment'>
<exit '1'>
<assign name 'Java'>
assignment<assign name 'Symlah'>
<return>
```

Listing 42

Conditional calls to programs is also made using a unique function, which is signified by the keyword 'linkif'. An example of a conditional call to a program is:

```
allVisitors<and visitor1 visitor2 visitor3>
<linkif allVisitors 'program2'>
```

Listing 43

In the above code listing, the 'linkif' function will call the program specified by the second literal passed to the function. In this case the literal is 'program2'. This means that a Symlah program by the name 'program2' has to be stored in the the directory that the above code is saved in. This also means that Symlah programs can call other programs in different directories (relative to the Symlah resource directory). For instance:

```
allVisitors<and visitor1 visitor2 visitor3>
<linkif allVisitors '/dir1/program2'>
```

Listing 44

The above code depicts that the conditional link will be made to the program in the physical location '/symlah/programs/dir1/program2', if the Symlah service directory has been defined as '/symlah/programs' in '/etc/svm.conf' (see section on service directory).

It should be noted that 'linkif' can make a call to the caller program itself. In such a case the data flow passed to the second instance of the program is that which exists at the juncture of the 'linkif' call.

Exceptions for conditional calls

Both the 'callif' and 'linkif' functions can be augmented with an additional argument. For the 'callif' function this argument defines the function to call if the given condition is false. For the 'linkif' function the argument defines the alternate program to call if the condition is false. Consider the following 'callif' function:

```
<callif allVisitors function2 fallback>
```

Listing 45

The above statement tells Symlah to jump to function 'function2' if the first value of 'allVisitors' was not empty. If the first value of 'allVisitors' is empty then Symlah will jump to the function 'fallback'.

The 'linkif' function also handles exceptions similar to the 'callif' function. Consider the following statement:

```
<linkif allVisitors '/dir1/program2' '/dir2/program3'>
```

Listing 46

The above statement tells Symlah to run the program '/dir1/program2' if the first value of 'allVisitors' is not found to be empty. Otherwise, if the first value of 'allVisitors' is indeed found to be empty, then Symlah should run the program '/dir2/program3'.

Conditional exit

Symlah provides a conditional exit function which terminates the current program and returns with the data flow existing up to that point in the program. The keyword for this function is 'exit'. The 'exit' function takes one argument, which is a variable name. If the first value of the variable is not empty then the function will terminate the program. The following statement demonstrates the use of the function:

```
<exit variable1>
```

Listing 47

Data manipulation

Arithmetic functions

The following list depicts Symlah's arithmetic functions:

- <add <variables> >
- <subtract <variables> >
- <multiply <variables> >
- <divide <variables> >
- <mod <variables> >
- <squareroot <variable> >

Listing 48

The <variables> notation in the above list of functions, except for the 'mod' and 'squareroot' functions, will have to be replaced with a space-delimited list of variables when writing the actual code. In the case of the 'mod' function only two variables can be passed whilst for the 'squareroot' only one variable can be passed.

If a programmer wishes to add the values of two variables together, the following code will have to be used:

```
result<add somevariable anothervariable>
```

Listing 49

In the above case, if the value in the first index location of 'somevariable' were 2 and the value of the first index location of 'anothervariable' were 7, then the value stored in 'result' will be 9. If the first index locations of 'somevariable' and 'anothervariable' held the same values and both variables had the values 5 and 3 in the second index location, then the 'result' will end up having two values: the value 9 in the first index location and the value 8 in the second.

If the variables passed to the 'add' function in Listing 49 contained non-numeric variables in corresponding index locations, then these variables will be concatenated in order. Hence, if the first value in 'somevariable' was the literal 'hello ' and that of 'anothervariable' was 'world' then the first index location of 'result' will end up having 'hello world' assigned to it.

If only one variable is passed to the 'add' function it will perform the task of adding up all the numeric values stored in all the index locations of the variable. This is useful to perform tasks such as the following:

```
result[1]<add '2' '3'>
result[2]<add '3' '17'>
result<add result>
```

Listing 50

The above code takes two direct literals (instead of literals stored in variables) and adds them together in the first two functions. The first one stores its result in the first index location of 'result', whilst the second stores its result in the second index location. The third function then adds up all the values in 'result' and stores it back in 'result' (the first index location of it, as no specific index location has been declared).

The remaining arithmetic functions perform their tasks in the same way as the 'add' function. That is, if more than two variables are passed, each with multiple indices, then the respective arithmetic function is performed iteratively on all the subsequent index values.

Just as with logical operation functions, Symlah allows passing index-specific variable constructs to arithmetic functions. The following statement shows how this is done:

```
result<add somevariable[7] anothervariable[1] yetanothervariable[32]>
```

Listing 51

Manipulating non-numeric data

Symlah's design philosophy subscribes to Chomsky's linguistic theories and consequently implements regular expressions as the only means of manipulating string literals (numeric and non-numeric alike). The regular expression semantics of Symlah has been designed to mirror Perl's implementation (Wall 2006). However, the syntax for writing the code for respective functions is different.

Symlah implements three functions for applying regular expressions to values in variables. The first function performs a regular expression match against a string literal. The second function is a substitution function to replace content in a string variable, which is similar to the 's///' pseudo-function in Perl. The third function splits a value in a variable at the points where a defined regular expression occur.

Regular expression matching

The syntax for the function is demonstrated in the following code:

```
<assign somevariable 'hello world'>
testResult<match somevariable '/world$/'>
```

Listing 52

In the above code the regular expression is 'world$' (the forward-slashes being mandatory). As per Larry Wall's implementation, the '$' sign in this expression indicates the end of the current line (each newline character indicating the end-of-line in the string value of any variable) being matched against the given string (Wall 2006). In the case of Listing 52 the given regular expression will be matched with all values of 'somevariable'. The results of these matches will be stored in 'testResult' at the corresponding index locations.

Regular expression can be made to match with values in specific index locations (or index range) by specifically declaring an index, as follows:

```
<assign somevariable[3] 'hello world'>
testResult<match somevariable[3] '/world$/'>
```

Listing 53

Since the second argument passed to the 'match' function is a literal, Symlah allows the substitution of this literal with a variable name, which has a literal in its first index location (or any user-defined single index location) that corresponds to a valid Perl-style regular expression. Hence the functionality of the above statement can also be implemented using the following code:

```
<assign somevariable[3] 'hello world'>
<assign anothervariable '/world$/'>
testResult<match somevariable[3] anothervariable>
```

Listing 54

The programmer may opt to define an index location for 'anothervariable' in the above code. At present there is no index location defined, which makes Symlah use the first index location of 'anothervariable'.

If any values in a given variable are matched then the name of the variable and the respective index location are stored in 'testResult'. To clarify this further consider the following code:

```
<assign somevariable[1] 'hello world'>
<assign somevariable[2] 'hello world'>
<assign anothervariable '/world$/'>
testResult<match somevariable[3] anothervariable>
```

Listing 55

Since both index 1 and 2 of 'somevariable' will be matched by the 'match' function then 'testResult'

will be assigned 'somevariable[1]' and 'somevariable[2]' in index locations 1 and 2, respectively.

Substitution using regular expressions

Consider the following Perl code:

```
$somevariable = 'Man is mortal. Socrates is a man. Therefore Socrates is mortal.';
$somevariable =~ s/man/woman/gi;
```

Listing 56

The tasks performed by the above code can be duplicated in Symlah using the following code:

```
<assign somevariable 'Man is mortal. Socrates is a man. Therefore Socrates is mortal.'>
<substitute somevariable 's/man/woman/gi'>
```

Listing 57

Once the above code is interpreted by Symlah, the resulting literal stored in '$somevariable' will be:

'woman is mortal. Socrates is a woman. Therefore Socrates is mortal.'

Symlah uses the exact same format of Perl-style regular expressions in the 'substitute' function. Hence, all the switches which can be used in a Perl regular expression can also be used in Symlah: switches such as 'g' for global and 'i' for case-insensitive matching (Wall 2006).

When a 'substitute' function finds a match the name of the variable and the index location is stored in the variable corresponding to the function name. If multiple matches are found then their respective index locations along with the variable name are stored, as in the case with the 'match' function (see the preceding section on 'Regular expression matching'). The 'substitute' function can be made to operate on a single value in a variable by specifying an index location for the variable passed to the function, as follows:

```
<substitute somevariable[1] 's/man/woman/gi'>
```

Listing 58

A quirk that the 'substitute' function purposefully adopts is a departure from the rules governing variables in Symlah within the regular expression literals it accepts. This quirk is the inclusion of temporary buffers used in Perl in the form of $1, $2, etc., within the regular expression literals. The following example demonstrates this:

```
<assign somevariable 'man woman'>
<substitute somevariable 's/(man).*(woman)/$2 $1/gi'>
```

Listing 59

What the above code does is swap the literals 'man' and 'woman' resulting in the first value of 'somevariable' being changed to 'woman man'. This is possible because Perl (the language in which the current incarnation of Symlah is implemented) assigns regular expression groupings (those enclosed in pairs of parentheses) to temporary buffers in the form of $1, $2, etc., in the respective order they occur. In the regular expression passed to the 'substitute' function above, the replacement literal is '$2 $1', which means that the first literal grouping matched by the regular expression will be appended to a space preceded by the second literal grouping.

Splitting using regular expressions

The 'split' function in Symlah splits a variable or literal at the points where a defined regular expression occur. The following statement is an example:

```
results<split somevariable '/ /'>
```

Listing 60.

What the above statement will do is split the first value in 'somevariable' at the points where a single space occur (not including the spaces), and store the resulting literals in the respective index locations of the variable 'result'. The first argument passed to the 'split' function can also be a literal.

Last matched literal groupings

Symlah provides a function to acquire a list of the literal groupings in a regular expression which were successfully matched against a variable last. This function is given the keyword 'last' and requires a function name to be associated with it. If a function name is not associated with it, then the function is ignored. The following code demonstrates the use of the 'last' function:

```
<assign somevariable 'man woman'>
<substitute somevariable 's/(man).*(woman)/$2 $1/gi'>
listofmatches<last>
```

Listing 61

In the above code, there are two literal groupings in the regular expression. Hence, the variable 'listofmatches' will be populated with the values 'man' and 'woman' in the index locations 1 and 2,

respectively.

Loops

Cycling through a list of values with 'foreach'

The 'foreach' function can be utilized to cycle through a list of values in a variable, calling another program at each cycle. The following code demonstrates the use of this function:

```
<assign somevariable 'hello' 'goodbye' 'good morning' 'goodnight'>
<foreach tempvariable somevariable 'program2'>
```

Listing 62

The second statement in the above code can be read as 'for each tempvariable in somevariable call the program “program2”'. This means that at each cycle the next value in the variable 'somevariable' is assigned to 'tempvariable', which in turn is passed to the program 'program2'. As implied by this functionality, 'program2' is passed only the variable 'tempvariable' and not the whole data flow of the current program (as opposed to the behaviour of Symlah with normal program calls, which is described in section).

Cycling through a numerical range with 'for'

The 'for' function is very similar to the 'foreach' function except for the source of data which is used in the iteration. Whereas 'foreach' uses the values in a variable as the data source, the 'for' function uses a set of values contrived from a range description. The following code demonstrates this:

```
<for tempvariable [1 to 10] 'program2'>
```

Listing 63

As evident in the above statement, the range description is written within brackets. In this case the range is the numbers 1 through 10, and the function cycles through these numbers. At each cycle the current number in the range is assigned to 'tempvariable' before passing it to the program 'program2'.

The range description used by the 'for' function follows the same rules as range descriptions for variables with multiple values. This means that the 'step' postfix can be used in the 'for' function as follows:

```
<for tempvariable [1 to 10 step 2] 'program2'>
```

Listing 64

Date and time

Symlah implements two temporal functions, one of which returns the current date and time and the other the seconds elapsed since the system epoch. These functions are given the keywords 'now' and 'whence', respectively.

The 'now' function returns a literal composed of the current time and date attained from the system epoch. The format for this literal is defined by the following BNF:

```
<now> ::= <hour> <colon> <minute> <colon> <second> <space> <dayOfMonth>
<slash> <month> <slash> <year>

<second> ::= <digit> <digit>

<minute> ::= <digit> <digit>

<hour> ::= <digit> <digit>

<dayOfMonth> ::= <digit> <digit>

<month> ::= <digit> <digit>

<year> ::= <digit> <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<slash> ::= “/”

<colon> ::= “:”

<space> ::= “ “
```

Data storage

The principles governing Symlah's data storage mechanism is where it departs drastically from the prevailing paradigms of programming language design. As established in the author's previous literature review (Shaafiee 2008), almost all of these paradigms pay homage to the von Neumann architecture (Tucker & Noonan 2007). Symlah breaks from this tradition by not implementing any functions for data storage.

One of Symlah's design principles dictate doing away with any functions for the purpose of storing and retrieving data. Thus, as far as the Symlah programmer is concerned, there are no data storage devices within the Symlah VM (which marks the departure from von Neumann architecture). However, this does not mean that Symlah does not keep persistent data within its VM.

Data modification points

There are distinct points at which data is modified in Symlah. These are:

1. When data is explicitly assigned to a variable using the 'assign' function
2. When a variable corresponding to a defined function name is created and data returned by the function is assigned to it
3. When a 'substitute' function is called and it finds a value in the variable passed to it that matches with the defined regular expression
4. When 'discard' function is called to clear the contents of a variable
5. When a 'unique' function call is made (delved into later in section)
6. When a 'deref' function assigns the address of one variable to another

Of the above data modification points, data persistency is synchronized only when modifications 1., 3. and 4. occur. When data in a variable corresponding to a function name is modified then that data is not made persistent. In order to force persistency of data in such a variable, the developer will have to exclusively assign the contents of it to another variable using the 'assign' function or the 'persist' function (see section).

The Symlah VM is designed to compartmentalize all tasks relating to variables into a single subsystem (referred to as the 'variables subsystem' in the Symlah design doctrine). The variables subsystem will be invoked to either assign, discard or modify (in the case of the 'substitute'

function) one or more values in a variable. Whenever this happens the variables subsystem invokes the data cloud subsystem to update the persistent state of the variable.

Exclusive persistence

A variable can be made to be persistent by forcing the issue with the 'persist' function. For instance, if a variable were assigned some content via the 'deref' function (see section), then the pointer variable (which holds the address of the target variable) is not persistent as per the rules of data persistency in Symlah (see section). In order to make such a variable persistent, the developer can call the function 'persist' as exemplified below:

```
returnedVariable<deref someVariable>
<persist returnedVariable>
```

Listing 65

Unique variables

It is possible for a situation to arise where a temporary variable, which is unique to the present program only, is needed. Consider the following scenario, where 'program1' has made a call to 'program2', which in turn has made a call to 'program3':

Illustration 1

In the above case, if a variable 'somevariable' were assigned a value in 'program1' then the variable will be passed to 'program2' and 'program3' in the general data flow. However, if within 'program1' the variable 'somevariable' were made unique (using the statement in Listing 66) then the variable will not be passed to 'program2'.

```
<unique somevariable>
```

Listing 66

Additionally, if 'program2' were to invoke a variable named 'somevariable' then this one will be separate from that which exists in 'program1'. In order to ensure this capability, any variable which was modified using the 'unique' function becomes non-persistent. The only way to make the data within such a variable persistent is to exclusively assign it to another, or call the 'persist' function (see section). It should be noted that even with enforced persistency unique variables will not be

forwarded to daughter programs, even if these programs can acquire the variable from the data cloud.