SLAWscript 1.0 Documentation

Draft Release 1

Abe Skolnik
Steve Henderson
Levi Lister

Table of Contents

Section Starting Page Number

Section 1: Introduction

1.1 Executive Summary

- Simplified Python.
- Runs in Java.
- Requires Java 1.5 or higher.

1.2 Introduction

The name of our language is "SLAWscript" (Steve, Levi, Abe, and the World's scripting language). SLAWscript is a general-purpose (yet simple) scripting language, designed to enable the easy production of text (i.e. command-line environment) applications. Amongst other possible uses, it will allow for quickly programming and deploying interactive training, tutorial, and survey applications.

SLAWscript is modeled on Python, but on a smaller scale. SLAWscript has no arrays, classes, or objects. At this time, only the three standard UNIX-like channels (stderr, stdin, and stdout) are accessible; that is to say, files cannot be opened and used. Also, SLAWscript is not strict about the use of leading spacing.

1.3 Key Features

• Conventional

SLAWscript attempts to use conventional notation where possible, as limited by the expressive abilities of ASCII. For example, the bar symbol ('|') is used to both begin and end an operator which returns either the absolute value (for numeric operands) or the string length (for string operands).

• Dynamic

In SLAWscript, variables don't need to be declared, and they are allowed to contain different data types at different points in time.

• Flexible

In SLAWscript, the addition operator can take either a number or string as either of its parameters, and intelligently decides whether to perform arithmetic addition or string concatenation. The multiplication operator is similarly flexible, and intelligently decides whether to perform arithmetic multiplication or string multiplication (i.e. multiplying 3 by "Hi" produces "HiHiHi"). In general, wherever a number is required, a variable containing a string containing an appropriate number may be used instead. (The primary exception is "assert" statements.) This allows for easier use of user-entered numbers in SLAWscript programs. For example, if the user entered "3" in response to a prompt, and that string is stored in a variable called "input", then the expression `10-input` yields the number seven.

• Interpreted

Our implementation of SLAWscript is an interpreter, which facilitates rapid development.

• Intuitive

SLAWscript is designed to use the English language as a basis whenever it is helpful to do so; for example, to copy the data from a variable named 'a' to a variable named 'b', simply use the command: "`copy a to b`".

• Portable and architecture-neutral

Our implementation of SLAWscript is based on Java, which gets us "for free" the advantages that it should be able to run correctly on many different operating systems and CPU types.

• Bug-preventing

In SLAWscript, the equals sign means only one thing: test for equality. (Contrast this with the fact that some other programming languages allow their '=' operator to both perform assignment and return a value, thus leading to confusion between using '=' for assignment and using it for comparison.) SLAWscript also encourages the writing of "safe" code by including an "assert" keyword.

1.4 Representative program

One representative program is an exam preparation assistant for a course in the humanities, such as a history course. Many of these tests require memorization of large amounts of information. SLAWscript can easily be used to create a program to act as an interactive practice exam. This practice exam would involve a series of text prompts that display practice questions, prompting for student input after each question.
SLAWscript's control logic allows the test designer to then branch and evolve the exam based on the student's input. For example, if the student answers incorrectly, hints can be presented to aid in memorization. Or, if the student is mastering the questions corresponding to a certain level of difficulty, the test can provide more difficult questions, thus adapting to the individual student.

1.5 Examples of Syntax

```
set a to 9              # this is how we “load” a literal value
set a to a+1            # this is how we increment a variable
copy a to b             # this is how we copy from one variable to another
put b to stdout         # this is how we “print”
put b+"\n" to stdout # this is how we “println”
```

1.6 Relevant Terminology

SLAWscript is…

- Imperative

SLAWscript's initial implementation is…

- Interpreted

SLAWscript variables are…

- Dynamically scoped
- Dynamically typed

SLAWscript subroutines are…

- Statically scoped (file scope)

SLAWscript subroutines' parameters are…

- evaluated by using applicative order
- fixed in quantity once the subroutine has been defined

SLAWscript functions' return values are…

- Dynamically typed

Section 2: Language Tutorial

In this section, we will lead you through the main features as well as techniques step by step in SLAWscript by writing a simple interactive program.

2.1 Getting Started

In SLAWscript, you have great freedom in creating your program by taking advantage of 34 keywords, but you may also successfully build a lightweight program using two or three of them. Now, we will build an introduction to our interactive program. First, create an empty file and name it "tutorial.SLAW", open it, and write the following line of code, and save it.

```
put "Welcome to the SLAWscript Tutorial.\n" to stdout
```

In command prompt mode, type "`slaw tutorial.SLAW`", and press return/enter. You will find that "Welcome to the SLAWscript Tutorial" is printed on the screen.

2.2 Using Variables

After the warm-up, we will now add a couple of variables to our program. Note that if at any time you want to add a comment to your code, just write your comment beginning with a '#' character.

```
set question1 to "sample question level 1: ..."
set question2 to "sample question level 2: ..."
set question3 to "sample question level 3: ..."
set question4 to "sample question level 4: ..."
set answer1 to "history"
set answer2 to "literature"
set answer3 to "architecture"
set answer4 to "economics"
```

Here we used the `set ... to ...` grammar to created eight variables: four questions and four answers.

2.3 Procedures and Functions

Like in other popular programming languages, it is always a good idea to write code in subroutines and to access functionality by calling them. SLAWscript supports two types of subroutines: functions and procedures; the major difference is that a procedure does not return a value to the program.

```
# main program starts
do welcome

set playing to true
while playing
  set cur_Question to getQuestion[level]
  put cur_Question to stdout
  put "Please type your answer below: \n" to stdout
  get cur_answer
  set result to checkAnswer[cur_answer]
  set level to grade[result]
  ignore nextStep[level]
end while

do exit
```

The previous code defines our main procedure which contains a "while" loop for running a couple of functions to implement the user interaction in our program. Every procedure and function begins with either `define procedure` or `define function` respectively, and ends with `end procedure` or `end function`. The invocation of a procedure and a function is a little different. We call our getQuestion function by `set cur_Question to getQuestion[level]`, where we use `level` in the brackets ("[ ]") as the argument passed into the function, and `cur_Question` as the variable for storing the value returned from the function. Note that we introduced a keyword here: "`ignore`", which means that we do want to invoke a function, but we do not care about the return value. Also, we used the keyword `get` to receive user input.
Next we will examine some of the functions used in the program to explore more of the features of SLAWscript.

2.4 Control Flow

We've seen the use of a `while` loop in our previous code; we also need conditional branching logic in our program. SLAWscript uses the `if ... else if ... else ... end if` structure, so we could write our function like this:

```
define function getQuestion[level]
  if level == 1
    copy answer1 to answer
    return question1
  else if level == 2
    copy answer2 to answer
    return question2
  else
    put "error in question level\n" to stderr
    stop
 end if
end function
```

Here we used a new keyword `copy`, which is used to copy the content of a variable. In our program we copied `answer1` or `answer2` to the `answer` variable based on which question is in use. If we accidentally come up with an error when running the program, we would like to stop it, so the else section gives an error message using `stderr` and then uses the `stop` keyword to terminate the program.

In the previous code, we used the equals-to operator ("`==`") in the way it is commonly used. Similarly, we used many operators and keywords such as `true` and `false` as their conventional way of use; also, we adopted usage of the keywords such as `and` and `or` in the logic judgment.

This tutorial covers only a limited subset of the features of SLAWscript, with the intention to give users a brief overview of how SLAWscript works, and to introduce users to writing programs by themselves.
For comprehensive instructions, please refer to Section 3: Language Reference Manual.

Section 3: Language Reference Manual

3.1 Introduction

The name of this language is derived from both the initials of the first names of the project members and the fact that SLAWscript is a scripting language. SLAWscript is a general-purpose (yet simple) scripting language, designed to enable the easy production of text-based (i.e. command-line) applications. SLAWscript is modeled on Python, but is on a smaller scale; SLAWscript has no arrays, classes, or objects. As of this writing, only the three standard UNIX-like channels (stderr, stdin, and stdout) are accessible within SLAWscript; that is to say, files cannot be opened and used without external assistance, e.g. shell redirection. Also, unlike Python, SLAWscript is not strict about the use of leading spacing.

3.1.1 Hello World

```
put "Hello World\n" to stdout
```

3.2 Lexical Conventions

3.2.1 Comments

The '#' character starts a comment from any place which is not inside a literal string. The comment begins at the '#' character and continues until the end of the line. SLAWscript does not support multi-line comments as a separate comment class; multi-line comments may be emulated by writing a sequence of contiguous lines each of which starts with a '#' as its first non-white-space character. Examples follow.

```
# this entire line is a comment
copy i to k # the part of this line including and after the first '#' symbol is also a comment
put "# <- this does not start a comment since it is in double quotes" to stdout
```

3.2.2 Constants

The following constants are always available in SLAWscript, and may not be changed:

`e`	2.718281828459045…
`escape`	ASCII/Unicode codepoint number 27
`pi`	3.141592653589793… (i.e. π)
`false`	0
`true`	1

For definitions of programmer-provided constants, please see “Numeric Literals” and “String Literals.”

3.2.3 Identifiers

The term “Identifiers” refers to all user-defined names (both the names of variables and the names of subroutines). The following rules define which characters can be used in which positions.

first character:	'A'...'Z', 'a'...'z'
all other characters:	'A'...'Z', 'a'...'z', '0'...'9', '_' (underscore)

Keywords are not allowed as identifiers. Otherwise-usable strings that have keywords as part, but not all, of their strings are allowed as identifiers (e.g. “if_I_were_a_rich_man”, “while_I_am_still_poor”).

Duplication across the names of subroutines and the names of variables is not allowed. This prohibition ignores letter case, so in a program with a subroutine named “hello”, a variable named “Hello” is illegal. A subroutine or variable may be referenced equivalently with any case variation; the identifiers “hello”, “Hello”, and “HELLO” all refer to the same subroutine or variable.

3.2.4 Keywords

The following strings are reserved for use as keywords, and therefore may not be used as identifiers. They may, however, appear within identifiers, e.g. “do_if_true”.

and	end	ignore	put	stdout
assert	escape	is	randomize	step
copy	false	localize	repeat	stop
define	from	not	return	times
do	function	or	true	to
e	get	pi	set	while
else	if	procedure	stderr	with

Keywords must be typed in all-lower-case, with the exception of “`pi`”, which may be typed in any case combination; choosing either “`pi`” or “`Pi`” is recommended, although “`pI`” and “`PI`” will also be recognized and considered equivalent.

Any attempt at redefining any of these words, either as a variable or as a subroutine, will fail regardless of case; therefore “While” and “IF” are also reserved words, even though they are not recognized as equivalent to “`while`” and “`if`”, respectively.

3.2.5 Numeric Literals

Numeric literals must be in the form of an optional minus sign followed by either an integer, i.e. one or more digits, or a floating-point number, i.e. zero or more digits followed by a period followed by one or more digits. Scientific notation is not allowed in direct numeric literals, but may be encoded in a string literal and converted to a number at run-time. In this case, the rules for the Java standard library's “Math.Double.parseDouble(String)” method apply. Examples follow.

```
3        -9        3.0       0.3       .3        0+"5e4"
```

3.2.6 String Literals

String literals must be enclosed in ASCII double-quote marks, e.g. `"Hello"`. The character sequence “`\n`” (backslash+'n') (when not immediately preceded by a single backslash) is converted to a newline. This sequence may appear more than once in the same string. If a string with “`\`” literally enclosed is desired, then the backslash should be doubled, like so: “`\\`”. Thus, if a string with “`\n`” literally enclosed is desired, then the backslash must be doubled, like so: “`\\n`”. If a string with “`"`” literally enclosed is desired, then the double-quote must be preceded by a backslash, like so: “`\"`”.

3.2.7 White Space

Leading spacing, that is to say any number of space or tab characters from the beginning of each line to the first (if any) non-white-space character, is ignored.

Ending spacing, that is to say any number of space or tab characters from the last non-white-space character to the end of the line, is also ignored.

Unnecessary separational spacing, that is to say any number of space or tab characters above and beyond the one required space or tab between two adjacent but distinct parts of a line, is also ignored.

Line endings are meaningful in SLAWscript; each complete sentence must end in a newline character or character sequence (any of CR, LF, and CR+LF is acceptable). The same rule applies to paragraph headers and footers, even though they are not complete sentences; for example, the “if” line that starts an “if” paragraph and the “end if” line which ends it.

3.3 Subroutines

Nested subroutines are not supported in SLAWscript. Subroutine definitions may only appear inside main-body code, e.g. not inside an "if" paragraph. Subroutine definitions may appear before, after, or in between sentences and other paragraphs in the main body of a program.

In SLAWscript, there are two distinct types of subroutines: functions and procedures. A SLAWscript program may not have a function and a procedure with the same name (ignoring case differences).
Subroutine invocations must have the same exact number of parameters (including the possibility of zero) as the respective subroutine definitions.

"if" paragraphs are the only places inside a subroutine other than in the main body of the subroutine itself where the "localize" verb may be used. "if" paragraphs are also the only places inside of a function other than in the main body of the function itself where the "return" verb may be used. In both cases, the special verbs may be used inside "if" paragraphs inside other "if" paragraphs if and only if there are no non-"if" paragraphs intervening depth-wise. An "if" paragraph anywhere inside a loop inside a subroutine does not have the special privilege of being allowed to contain a "localize" or a "return".

3.3.1 Subroutine Scope (summarily: static)

All subroutines in SLAWscript are statically scoped; the subroutines may be invoked from anywhere in the program (including inside themselves, i.e. recursive subroutines are allowed).

3.3.2 Procedures

Procedures do not return any values, and cannot be invoked from inside expressions; thus, the "`return`" keyword may not be used inside of a procedure. Parameters are optional, and listed in brackets if desired. Invoking a procedure is done by using the "`do`" keyword followed either by the procedure's identifier alone for a procedure that takes zero parameters, or by the identifier followed by the bracketed list of actual parameters for a procedure that takes a positive number of parameters. An example follows.

```
define procedure say_hello
  put "Hello World.\n" to stdout
end procedure

do say_hello  #  example procedure invocation
```

3.3.3 Functions

Functions return exactly one value; parameters are optional, and listed in brackets when desired.
Formal parameters are automatically local variables; global variables with the same names as any of the formal parameters are hidden for the duration (see "Variable Scope"). Invoking a function is done simply by using its identifier alone inside an expression (including the possibility of just the identifier itself) for a function that takes zero parameters, or by using the identifier followed by the bracketed list of actual parameters for a function that takes a positive number of parameters. An example:

```
define function square[x]
  if x?  #   the '?' operator here returns 0 if 'x' is not usable as a number
    return (0+x)*x
    # "(0+x)" in case 'x' is e.g. "3"; otherwise x*x for x="3" would return "333"
  else
    put "Error: this is not a number: '"+x+"'.\n" to stderr
    stop  #  this causes the whole program to stop, not just the subroutine
  end if
end function

set a to square[3]   #  example function invocation
```

If you wish to invoke a function for the sake of its side-effect(s) but do not care about its return value, then use the keyword "ignore". For example...

```
ignore square[b]
```

… which will "throw away" b-squared if it exists, and will exit with an explanatory error message if it does not exist because 'b' contains a non-numeric string, e.g. "Hello".

A function may perform a "return" in its main body, inside an "if" paragraph (including its possible "else if" and "else" dependent clauses) in its main body, and inside "if" paragraphs inside those "if" paragraphs.
The keyword "return" may not appear inside a loop, including inside an "if" paragraph inside a loop.
Each execution of a function must end in a "return" sentence; arriving at the "end function" line without returning any value is an error.

3.4 Variables

In SLAWscript, variables do not require declaration. However, a variable must be set to some value before an attempt to read from it is made.

3.4.1 Data Types

A SLAWscript variable can hold either string data (16-bit Unicode) or numeric data (IEEE double-precision).

Variables holding strings containing only a number (after the removal of optional surrounding white space) are given special privilege not accorded to other string-holding variables: they are permitted wherever a variable containing a number is permitted. For example, a variable containing the string "-1.2e3" is permitted as a parameter to the subtraction operator. The rules for the Java standard library's "Math.Double.parseDouble(String)" method control what is allowed as a number.

For the purpose of concision, throughout the rest of this manual the following terms will be used to denote the type of data to which a phrase is referring:

- number: a datum which is stored in IEEE double-precision format
- numeric string: a string which can be converted to a number
- non-numeric string: a string which cannot be converted to a number
- arbitrary string: a string without regard to its numeric convertibility

3.4.2 Assignment

Assignment of the result of evaluating any valid expression may be done with the "set" keyword.
For convenience and clarity, a "copy" keyword also exists for the copying of the data in one variable into another variable without change. Examples follow.

```
set hello to "world"
set a to a+1  # This will increment 'a' if it is a number, and append '1' to 'a' if 'a' is an arbitrary string
set a to b    # This is not illegal, but it is both confusing and inefficient; please use "copy" instead.
copy a to b   # Please note the opposite direction of data flow relative to "set".
```

3.4.3 Variable Scope (summarily: dynamic)

A SLAWscript variable is global by default, even if it is first set inside a subroutine, with the exception of formal parameters and explicitly localized variables.

The formal parameters of subroutines are implicitly local variables. This cannot be overridden; during the execution of a subroutine containing a formal parameter 'x', the global variable 'x' (if one exists) is hidden for the duration. This duration includes the execution of subroutines called from the subroutine containing the formal parameter 'x', subroutines called from those subroutines, and so on.

Variables may be explicitly localized inside of subroutines by using the keyword "`localize`" followed by an identifier. Localizing the same identifier again within the same subroutine as another localization with the same identifier (or a formal parameter with the same identifier) is not an error, but it has no effect. Localizing an identifier for which there is no global variable is not an error; it allows that variable to exist for the duration (see the preceding paragraph for the definition of "duration"), and causes the variable to cease to exist after the subroutine in which it was localized ends.

Localization may occur in the main body of a subroutine, inside an "if" paragraph in the main body of a subroutine (including its possible "else if" and "else" dependent clauses), and inside "if" paragraphs inside those "if" paragraphs. Localization may not occur inside a loop, including inside an "if" paragraph inside a loop. The scope of a localization that occurs inside an "if" paragraph is the same as if it had occurred in the main body of the subroutine.

Once a variable has been localized, it remains localized for the duration, i.e. until the same execution of the same subroutine ends. Thus, subroutines called from a (potentially the same, i.e. recursive) subroutine receive their "parent" subroutine's local variables, if any, rather than global variables with the same identifiers.

An example follows.

```
define procedure localization_example
  localize a

# At this point, 'a' is a local variable until this procedure ends; any procedures or functions called by
# this procedure inherit this version of 'a' unless they have an 'a' in their formal parameters.

# An example of what is not valid here: "put a to stdout"; reason: 'a' is undefined as of now and
# may not be used except to set it (using "copy", "get", "set", "randomize", or "repeat with").

  set a to 10
  put a to stdout     # this is now OK: it will put "10" to stdout
end procedure         # after this line, not only is control returned to the caller, but 'a' is also
                      # automatically delocalized.  If "localization_example" was called from a
                      # context where 'a' was 9, then 'a' shall now be 9 again.
```

3.4.4 Randomization

A variable may be explicitly randomized, which sets it to a random number between 0 (inclusive) and 1 (exclusive) when the "`randomize`" sentence is executed. The variable is not continually re-randomized; it must be re-randomized if a new random number is required. This is the only random number support in SLAWscript. An example: "`randomize r`".

3.5 Operators

3.5.1 Unary Operators

`( )` order-of-precedence overrides.

`| |` absolute value, string length (surround the operand as if with parentheses).

`!` factorial (postfix).

`–` unary negative (e.g. "–a").

`?` variable content type (postfix): returns 0 if the variable holds a non-numeric string, 1 if it holds a numeric string, and 2 if it holds a (non-string) number; illegal to use after anything but a variable.

`~` prefix: returns the rounded number if followed by a numeric expression or a numerically convertible string expression; invalid if followed by a non-numeric string.

`%` postfix: divides the preceding number by 100; may only appear after a literal number, e.g. not after a variable.

`not` boolean NOT.

3.5.2 Binary and Tertiary Operators

`^` exponentiation.

`/` division.

`*` multiplication (both numeric and string: `3*4` yields 12, and `3*"Hi"` yields “HiHiHi”).

`-` subtraction (e.g. *a*`-`*b*).

`+` addition, string concatenation.

`<` is less than.

`<=` is less than or equal to.

`>` is greater than.

`>=` is greater than or equal to.

`=` relaxed equality (see “Binary Operator Auto-conversion”).

`==` strict equality (see “Binary Operator Auto-conversion”).

`<>` relaxed inequality (see “Binary Operator Auto-conversion”).

`<<>>` strict inequality (see “Binary Operator Auto-conversion”).

`and` boolean AND (short-circuited: left operand is always evaluated, right operand is not always evaluated).

`or` boolean OR (short-circuited: left operand is always evaluated, right operand is not always evaluated).

`@` substring (postfix): must be followed by either one or two number or numeric string operands (separated by a semicolon if two are present). *a*`@9` returns the string in 'a' from the 9th char. onwards; *a*`@9;2` returns a string of length of at most 2, starting from the 9th char. of 'a'. Returns an empty string if the first operand is not long enough for the second operand, or if the third operand (if present) is non-positive after rounding; in both of those cases, a warning is sent to standard error.

`:` substring position: `"Hello":"el"` returns 2; `"Hello":"x"` returns 0; `"x":"Hello"` returns 0. `"":`*a* returns 0 for any non-empty-string 'a' (including numbers). *a*`:""` returns -1 for any non-empty-string 'a' (including numbers), since the empty string is implicitly contained within every string, yet its position within the enclosing string cannot be defined. `"":""` returns 1, since the empty string is exactly equal to itself (i.e. for the same reason as the reason why `"hi":"hi"` returns 1). Note: when the right parameter matches the left one in more than one place, the first match determines the index that is returned; for example, `"Hello":"l"` returns 3. Also please note: the values returned by this operator have been chosen so that the result of the operator may be used in a boolean context with the meaning that the result is true (non-zero) if the strings are equal or the right one is contained in the left one, and false (zero) if the right string is provably (i.e. the right is not empty) not contained in the left string.

3.5.3 Operator Precedence

The following list of groups of operators is in the order of highest-precedence-first. Operators within the same group have the same precedence level, and are evaluated left-to-right.

1. `( )` `| |`
2. `!` `not` `~` `%` `?` unary `-`
3. `@` `:`
4. `^`
5. `/`
6. `*`
7. `+`
8. binary `-`
9. `<` `>` `<=` `>=` `=` `==` `<>` `<<>>`
10. `and` `or`

The order of precedence has been carefully chosen so as to make it as likely as possible that what the programmer intended is what is "understood" by the computer, even if parentheses were not used, especially with respect to equality/inequality and chains of all-and/all-or operations. Therefore, for example, "`a>b+c and b<c/d and f=0 and g>=h!`" is equivalent to "`(a>(b+c)) and (b<(c/d)) and (f=0) and (g>=(h!))`". Additionally, the '+' operator has been given higher precedence than the binary minus operator so that expressions such as `1-""+0+"."+5` result in the complete addition chain being evaluated first, so that in the case of the preceding example, `""+0+"."+5` is first evaluated to the string "`0.5`", followed by the subtraction. Had this not been the case, the `1-""` part would have resulted in an error message and program halt.

Be aware, however, that chains of boolean operations which mix "and" with "or" must use parentheses if they are to be evaluated in an order other than left-boolean-operation-first. Also be aware that "not" (like other group-2 operators) binds tightly, and therefore requires its operand to be grouped using "( )" or "| |" if the operand is not either indivisible (i.e. a constant, a literal, or a function invocation) or an expression of group-1 or group-2 precedence level.

3.5.4 Operator Chaining

Most of the binary operators allow chaining; for example, "`set a to b+c+d`" is perfectly valid. However, the relational operators (precedence group 9) do not allow chaining. This is to prevent the writing of code which does not mean what the author thinks it means. For example: in math courses, one is typically taught that "a<b<c" means that 'a' is less than 'b' and 'b' is less than 'c'. However, in many programming languages, although "`a<b<c`" is a valid expression, it does not mean what it would mean in a math course. To get the mathematical meaning of "a<b<c" in SLAWscript, you must use something equivalent to "`a<b and b<c`".

The substring operators ('@' and ':') also do not allow chaining. If one wishes to write a SLAWscript expression that takes, for example, a substring of a substring, one must use parentheses, e.g. "(a@b)@c". This was designed in this way mainly to avoid potential ambiguity in case of tertiary parameters to '@' operators; in the case of "a@b@c;d", had such a thing been allowed, of which '@' is 'd' the third parameter? Also, taking a substring position of the result of a substring position operator (which returns a number) is fairly useless.

Please note that the proscription against the chaining of certain operators does not prevent a SLAWscript programmer from intentionally simulating the same chaining by using parentheses. For example, "a<(b<c)" is valid, and means the same as "a<1" if 'b' is less than 'c', and the same as "a<0" otherwise.

3.6 Auto-conversion

The SLAWscript implementation shall, when needed, convert data from its current type to another type, possibly with multiple conversion steps, in order to use the operands that are given to operators and verbs. These conversions do not affect the data or the type of data stored in a variable; the conversions are temporary, and may include converting the data type of the result of an expression to the needed data type.

The SLAWscript implementation shall output an error message and halt the program due to an incompatible data type only when it cannot convert the supplied (or computed) data to the needed type; for example, when a number is needed, and a non-numeric string (e.g. "Hello") is supplied instead.

3.6.1 Unary Operator Auto-conversion

Since the following unary operators in SLAWscript expect a number, they all attempt to convert a string to a number when they find a string as their operand: '!', "not", unary '–'. Examples follow.

```
set a to not "0"      # this should set 'a' to 1
set c to "3"
set d to c!           # this should set 'd' to 6 (numeric)
set f to -c           # this should set 'f' to -3 (numeric)
```

The following unary operators never perform auto-conversion: "( )", "| |", '%', '?'.

3.6.2 Binary Operator Auto-conversion

For binary operations, the implementation is to make its best effort at making sense of the expression, and only abort with an error if absolutely necessary. If at least one of the operands must be converted, and the choice of which operand to convert is ambiguous because the expressions resulting from either expression would be valid, then the left operand "wins", i.e. it gets to keep its current type. Therefore, `"3"*4` yields the string "3333", whereas `3*"4"` yields the number 12.

Auto-conversion of a data type does not implicitly change the data type in a variable. For example...

```
set g to 4*c # reminder: 'c' is the string "3"
# g is now 12
set h to c*4
 # 'h' is now "3333" because 'c' is still a string
```

If you wish to permanently change the data type of a variable by using auto-conversion, you must use "copy" or "set". For example...

```
set i to 9
set i to ""+i
# 'i' now contains the string "9" because the left-hand empty string "won" the data-type conversion "contest"
copy i to j
set j to 0+j
# 'j' now holds the number 9 because the left-hand zero "won"
copy i to k
set k to 1*k # this is another way to force a numeric without a change of value
# 'k' is now 9 (numeric)
```

As a result of the autoconversion rules, the '+' operator and the '*' operator are not always commutative. In other words, `a+b` is not always the same as `b+a` and `a*b` is not always the same as `b*a`. When 'a' and 'b' are both truly numbers, i.e. not merely numeric strings, commutativity is preserved.

The following binary operators attempt to perform autoconversion to a number on both of their parameters: '^', '–', '<', "<=", ">=", '>', '/', "and", "not", "or".

The relaxed equality ("=") and relaxed inequality ("<>") operators only check for either string equality or numeric equality, and don't care which one they find; if they find <u>any</u> equality, even if by auto-conversion, then they consider their operands to be equal. In other words, `"3"=3` and `9="9"` are both true, and `"3"<>3` and `9<>"9"` are both false. `"Hello"="Hello"` is also true.

The strict equality ("==") and strict inequality ("<<>>") operators <u>never</u> perform auto-conversion. Therefore, `"3"==3` and `9=="9"` are both false, and `"3"<<>>3` and `9<<>>"9"` are both true. In all cases where the (in)equality holds without any auto-conversion, strict (in)equality works the same as relaxed (in)equality.

The '@' operator performs conversion to a string if it finds a number as its first operand, and attempts to perform auto-conversion to a number on either or both of its second or third operands, as needed. For example, `12345@"2";"3"` produces "234".

The '+' and '*' operators perform autoconversion of their parameters, if needed. Where ambiguity comes into play due to the types and content of the parameters, the left-hand parameter (with its original type) determines the result of the operation. For example, `1+"2"` yields the number 3 whereas `"1"+2` yields the string "12". For the '*' operator, `"1"*2` yields "11", whereas `1*"2"` yields the number 2. For non-numeric strings, there is no autoconversion; therefore, `"a"+"b"` always yields "ab", and `"a"*"b"` is always an error.

The ':' operator performs conversion to strings on any operand it finds as a number. Therefore, even a substring position between an enclosing number and an enclosed number can be found; for example, `Pi:1` yields 3.

For forcing a value to be in either number form or string form, the recommended idioms are `0+…` and `""+…`, respectively. Other techniques may also produce the desired result. For example, "`1*…`" is mathematically equivalent to "`0+…`", but may not produce exactly the same result due to the inherent issues with IEEE 754 math.

3.6.3 Boolean Context

While there is no boolean data type in SLAWscript, there is a concept of boolean context. This context occurs whenever a boolean value is needed from the program. For example, an "if" statement, a "not" operator, and a "while" loop each require one boolean value, whereas "and" and "or" require two of them.

In this context, any numeric expression that evaluates to 0 is considered false. All other numeric expressions are considered true. Any string that can be converted to the number zero (e.g. "0", "0.0", "-0") is considered false. Any string that can be converted to a non-zero number is considered true. A non-numeric string in this context is an error, and causes an error message to be output and the program to be halted.

The inequality and equality operators (both relaxed and strict), as well the "and", "not", and "or" operators, all produce a boolean number, i.e. either 0 or 1, as their output.

The '?' operator, while it does not produce only strictly boolean values, has been designed in such a way as to make it usable by itself, as if it were strictly boolean. The '?' operator can be used by itself to ensure that a variable is currently usable wherever a number is needed, as long as having a numeric string will also cause the desired result to be produced. If a number is required, and a numeric string will not necessarily cause the desired result to be produced, then use e.g. "if x?=2" ("if 2=x?" is equivalent, but misleading to a naïve reader).

3.6.4 Integer Context

While there is no integer data type in SLAWscript, there is a concept of integer context. This context occurs whenever an integer value is needed from the program.

Whenever an integer is required, the SLAWscript implementation shall convert the datum first to a number, if it is a numeric string, and then from a number to an integer by rounding. Therefore, –0.1 and 0.1 both convert to 0, 0.5 converts to 1, –0.9 converts to –1, etc. The rules for the Java standard library's "Math.round(double)" method apply. (In particular, please note that, for example, -0.5 rounds to zero.)

The '@ operator must take either one or two integers as its second and optional third operands. The second operand must be positive after rounding.

Also, the "repeat…times" loop takes one integer, which must be non-negative after rounding.

3.7 Conditionals

SLAWscript supports the usual "if…else if…else…end if" structure. The "else if" section may appear any non-negative integer number of times per "if". The "else" section may appear zero times or one time per "if". The "end if" marker must appear exactly once per "if", regardless of the presence or lack thereof of "else if" sections and an "else" section. Any valid expression may appear after "if" and "else if"; see "Boolean Context" for the details of the interpretation. At least one space must be present between the "else" and the "if" of "else if" and between the "end" and the "if" of "end if". The code (if any) inside all sections must be valid, even if it will never be executed. Empty sections are allowed, including comment-only sections, blank-line-only sections, and truly empty sections with no lines at all.

An example follows.

```
if 0 and 0
                    # Putting code here won't help - it will never execute since (0 and 0) is false.
else if 0 or 0      # This "else if" is a silly exercise in futility.

else
end if
```

3.8 Loops

SLAWscript supports three kinds of loops: “repeat … times” loops, “repeat with” loops, and “while” loops.

3.8.1 “repeat … times” Loops

This type of loop is useful for code that needs to be executed any zero-or-more integer number of times, and the code inside the loop does not need to keep track of the number of times it has been executed.

The loop is started with a line containing the word “repeat”, followed by at least one space or tab, followed by an expression, followed by at least one space or tab, followed by the word “times”. The loop must be ended with a line containing “end repeat”, where the number of spaces or tabs between “end” and “repeat” must be at least one.

The existence of this type of loop frees SLAWscript programmers from having to worry about index variables, index incrementation, and loop termination. Furthermore, it prevents unnecessary “pollution” of the variable namespace with a variable that is only going to be used for “housekeeping”. In the case of this loop type, SLAWscript performs the housekeeping automatically.

The expression between “repeat” and “times” is evaluated in integer context, and is therefore rounded. If this expression (taken as a number) rounds to zero, the loop is not executed at all. A positive number (after rounding) causes the appropriate number of loop executions (provided the program does not halt before the loop ends). Negative numbers (after rounding) and non-numeric strings as the expression result are both errors.

An example follows.

```
repeat square_root[81] times
  put "Number 9... " to stdout
end repeat
```

3.8.2 “repeat with” Loops

This type of loop is useful for code that needs to be executed any zero-or-more integer number of times, and the code inside the loop does need to keep track of the number of times it has been executed.

The loop is started with a line containing the word “repeat”, followed by at least one space or tab, followed by the word “with”, followed by an identifier that is not in use as the name of a subroutine (ignoring letter case), followed by at least one space or tab, followed by the word “from”, followed by at least one space or tab, followed by an expression, followed by at least one space or tab, followed by the word “to”, followed by at least one space or tab, followed by an expression, optionally followed by [at least one space or tab, followed by the word “step”, followed by at least one space or tab, followed by an expression].

The loop must be ended with “end repeat”, where the number of spaces or tabs between “end” and “repeat” must be at least one.

The identifier that comes after the word “with” is used as the loop index, which is still accessible after the loop ends. The usual scope rules for variables apply, so if the identifier was not previously localized (for a loop within a subroutine), then it identifies a global variable. If the identifier was previously localized, then the higher-level local variables (if any) and global variables (if any) with the same name are not affected. In this paragraph, the term “localized” refers to both explicit localization using the “localize” keyword and to the implicit localization that comes with formal parameters.

The expression that comes immediately after the word “from” is used as the loop's starting index. This expression must yield either a number or a numeric string (which will be auto-converted to a number). This can be a real number, so long as the loop makes sense. (Please see 'Note on “repeat with” Precision'.)

The expression that comes immediately after the word “to” is used as the loop's ending index. This expression must yield either a number or a numeric string (which will be auto-converted to a number). This can be a real number, so long as the loop makes sense. (Please see 'Note on “repeat with” Precision'.)

If the two indices are equal, then the loop is not executed at all, regardless of the optional “step” section. In this case, the loop's index variable is set, the same as if the loop had been executed; it is set to the value to which both of the indices are equal. If the optional “step” section is omitted, then SLAWscript automatically sets the loop increment either to one, .in the case of the starting index being less than the ending index, or to negative one, in the case of the starting index being greater than the ending index.

If the optional “step” section is not omitted, then SLAWscript sets the loop increment to the value yielded by the expression that comes after the word “step”, converting it to a number if it was a numeric string. In this case, if the value of the “from” expression is less than the value of the “to” expression, then the value of the “step” expression must be positive, and if the value of the “from” expression is greater than the value of the “to” expression, then the value of the “step” expression must be negative. (This is the “makes sense” which was referred to earlier in this section.) In the case of the loop's starting and ending indices being equal, the value of the “step” expression is irrelevant, since the loop will not be executed. The “step” expression is still evaluated; therefor, the start and end indices being equal does not exclude the “step” expression from its usual requirement of being required to execute correctly and return either a number or a numeric string.

Non-numeric strings as the result of evaluating the “from” expression, the “to” expression, or the “step” expression (if it is present) are all errors.

The results of evaluating the “from” expression, the “to” expression, and the “step” expression (if it is present) are not rounded; therefore, repeating from 0.1 to 0.5 with a step of 0.001 is valid and will behave as expected.

An example follows.

```
repeat with a from b+1 to c-1 step d/2
  put a+"\n" to stdout
end repeat
```

3.8.2.1 Note on “repeat with” Precision

Due to the inherent imprecision of IEEE 754 mathematical operations, certain limitations must be put on the indices, step values, and counter variable values of “repeat with” loops. In order to prevent incorrect numbers of loop executions when using a fractional part that is not exactly representable with a binary fraction (e.g. one tenth), SLAWscript implementations are only required to correctly handle four decimal digits on the right side of the decimal separator. Additional accuracy may be present; eight-digit precision(on either side of the decimal separator) should be possible within the limitations of 64-bit floating point and 64-bit signed integer data types.

3.8.3 “while” Loops

This type of loop is useful for code that needs to be executed a non-predetermined number of times. It is the same as the “while” loop the reader is likely to be familiar with from at least one other programming language. For the details on the interpretation of the expression following the word “while”, see “Boolean Context”.

The loop must be ended with “end while”, where the number of spaces or tabs between “end” and “while” must be at least one.

An example follows.

```
while a<b
  set a to a+1
end while
```

3.9 Input and Output

Only the three standard UNIX-like channels (stderr, stdin, and stdout) are accessible within SLAWscript; that is to say, files cannot be opened and used without external assistance, e.g. shell redirection.

3.9.1 Input

In SLAWscript, there is only one input technique: the “get” verb, which fetches one line from the “Standard Input” channel. Each such input must end with the system's appropriate newline, which is not included in the value which is stored into the variable indicated by the “get” sentence. The value stored by “get” is always a string. If a number is desired, and a numeric string was retrieved by “get”, then a numeric conversion may be performed by e.g. “0+input”. See “Auto-conversion” for more on this topic.

Please note that the input is not guaranteed to be non-empty; in particular, if the user presses “return” or “enter” on her/his keyboard immediately following the input request, the variable will be set to an empty string.

An example follows.

```
get foo
# At this point, “foo” should contain a string representing one line of input, minus the ending newline.
```

3.9.2 Output

In SLAWscript, there is only one output technique: the “put” verb, which sends data to either the “Standard Output” channel or the “Standard Error” channel, as determined by the SLAWscript code.

A “put” sentence consists of a line containing the word “put”, followed by at least one space or tab, followed by an expression, followed by at least one space or tab, followed by the word “to”, followed by at least one space or tab, followed by either the keyword “stdout” or the keyword “stderr” (all lower-case for both).

The expression included in a “put” sentence may evaluate to either a string or a number. If it evaluates to a number, that number is auto-converted to its string representation.

The implementation of the “put” sentence shall not output any characters that were not specified by the evaluation of the expression. In particular, an explicit “\n” is required in order to output an end-of-line.

Examples follow.

```
put "'a' is currently <"+a+">\n" to stdout # an explicit “\n” is required to output an end-of-line
put "Oops!\nI did it again!\n" to stderr   # more than one “\n” in a single string is OK
```

3.10 Program Termination

SLAWscript programs normally terminate only when they either arrive at their normal conclusion after the execution of the last line of main body (i.e. non-subroutine) code, or when an error occurs.
However, additional methods to cause program termination to occur are available.

3.10.1 “stop”

SLAWscript includes a verb called “stop” that causes immediate unconditional program termination. It is valid anywhere, including inside loops, subroutines, and conditionals.

3.10.2 Assertions

As an aid to programmers, the language defines a verb “assert” that is similar to verbs with the same name in other programming languages. An “assert” sentence compares the current contents (if any) of a variable whose identifier immediately follows the word “assert” (followed by at least one space or tab) to a literal or constant value which follows the word “is” (surrounded on both sides by at least one space or tab) which, in turn, follows the identifier.

This is mainly a convenience mechanism, as otherwise the programmer could write something like this:

```
if a<<>>"test"
  stop
end if
```

However, in the name of increasing the likelihood of programmers using this kind of assertion liberally, the following is equivalent to the preceding:

```
assert a is "test"
```

In the case of “assert” statements, auto-conversion does not apply; that is to say, `assert a is 3` and `assert a is "3"` are different. In any place that one would succeed, the other would fail. If the programmer is certain that a variable contains either a number or a numeric string, is not sure which is the case, and wants either one to succeed, then (s)he may write something like this:

```
if 3<>a #  This is intentionally not “a<>3”, which would auto-convert 3 to a string if 'a' were a string,
  stop  #        which would then lead to a possibly-erroneous result, since "3"="03.0" is false.
end if
```

... which will allow program execution to continue only if 'a' was either 3 or “3” or “3.0” or “03.00” etc.

3.11 User-defined Constants

SLAWscript does not explicitly include the possibility of user-defined constants; the only supported constants are 'e', "escape", "false", "pi", and "true". The only exception to the usual SLAWscript rule of lower-case-only for built-in words is for the constant "pi", of which each letter may appear in any case.

A side-benefit of the fact that SLAWscript invocations of zero-parameters functions do not use brackets is the fact that such a function is invoked in a way that is visually indistinguishable from using a variable. Not only that, but since functions and variables are not allowed to share a name in a program, you can be assured that a correctly-written SLAWscript program containing a function named "foo" does not contain a variable named "foo" (or "Foo", or "FOO", etc.) anywhere in that program (due to the fact that SLAWscript subroutines are statically scoped, with case-insensitive clash-checking). This means that, in a program containing a subroutine named "foo", you are guaranteed that the sentence "`set foo to 9`", for example, is in error. This, coupled with the following example, makes "foo" effectively a constant.

```
define function foo
  return 42
end function
```

The preceding code fragment effectively assigns the number 42 to the (invariant) return value of "foo", thus effectively making "foo" a constant. The same technique may be used for strings; for example...

```
define function Professor
  return "Edwards"
end function
```

... effectively defines a constant named "Professor" whose invariant value is the string "Edwards".

Given both of the following preceding definitions, I can then write e.g. "`set a to 3+foo+Professor`", which will set 'a' to "45Edwards", providing that there is no subroutine named either 'a' or 'A' in the program.

3.12 Formal Grammar (not yet finished)

The following formal grammar is intended to be in the Extended Backus-Naur Form, as can be read about on-line at this address: http://en.wikipedia.org/wiki/Extended_Backus-Naur_form

```
SLAWscript program = { effectively empty line
                     | normal sentence
                     | main body paragraph
                     }; (* zero or more times *)

effectively empty line = optional spacing, [comment], newline;
                           (* the comment is optional *)

comment = “#”, printable-(CR | LF); (* '-' here means "except" *)

spacing = “ ” | “\t”;

optional spacing = {spacing};

required spacing = spacing, {spacing};

printable = ? all printable characters, including space ?;

CR = “\r”;

LF = “\n”;

CRLF = “\r\n”;

newline = (CRLF | CR | LF); (* PCDOS-style, Mac pre-[OS X] style, and Unix-style *)

normal sentence = assert sentence
                | copy sentence
                | do sentence
                | get sentence
                | ignore sentence
                | put sentence
                | randomize sentence
                | set sentence
                | stop sentence
                ;

main body paragraph = normal if paragraph
                    | subroutine definition paragraph
                    | repeat paragraph
                    | while paragraph
                    ;

assert sentence = optional spacing, “assert”, required spacing, identifier,
                  required spacing, “is”, required spacing, (constant | number | string),
                  optional spacing, [comment], newline;

copy sentence = optional spacing, “copy”, required spacing, identifier, required spacing,
                “to”, required spacing, identifier, optional spacing, [comment], newline;

do sentence = optional spacing, “do”, required spacing, identifier,
              [ “[”, optional spacing, expression, optional spacing,
                { “,”, optional spacing, expression, optional spacing }, “]”
              ], optional spacing, [comment], newline;

get sentence = optional spacing, “get”, required spacing, identifier, optional spacing,
               [comment], newline;
```

```
ignore sentence = optional spacing, “ignore”, required spacing, identifier,
                  [ “[”, optional spacing, expression, optional spacing,
                    { “,”, optional spacing, expression, optional spacing }, “]”
                  ], optional spacing, [comment], newline;

put sentence = optional spacing, “put”, required spacing, expression, required spacing,
               “to”, required spacing, (“stdout” | “stderr”), optional spacing,
               [comment], newline;

randomize sentence = optional spacing, “randomize”, required spacing, identifier,
                     optional spacing, [comment], newline;

set sentence = optional spacing, “set”, required spacing, identifier, required spacing,
               “to”, required spacing, expression, optional spacing, [comment], newline;

stop sentence = optional spacing, “stop”, optional spacing, [comment], newline;

normal if paragraph =
      optional spacing, “if”, required spacing, expression, optional spacing, [comment], newline,
            { normal if paragraph
            | repeat paragraph
            | while paragraph
            | normal sentence
            | effectively empty line
            },
      { optional spacing, “else”, required spacing, “if”, required spacing, expression,
        optional spacing, [comment], newline,
            { normal if paragraph
            | repeat paragraph
            | while paragraph
            | normal sentence
            | effectively empty line
            }
      },
      [ optional spacing, “else”, optional spacing, [comment], newline,
            { normal if paragraph
            | repeat paragraph
            | while paragraph
            | normal sentence
            | effectively empty line
            }
      ],
      optional spacing, “end”, required spacing, “if”, optional spacing, [comment], newline
;

while paragraph =
  optional spacing, “while”, required spacing, expression, optional spacing, [comment], newline,
       { normal if paragraph
       | repeat paragraph
       | while paragraph
       | normal sentence
       | effectively empty line
       },
  optional spacing, “end”, required spacing, “while”, optional spacing, [comment], newline
;

digit = “0”|“1”|“2”|“3”|“4”|“5”|“6”|“7”|“8”|“9”;

number = ( {digit} “.” (digit, {digit}) )
       | (digit, {digit})
       ;
```

```
letter = “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”|
         “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”;

identifier = letter, { letter | “_” | digit };

string = “"”, { ( printable-(CR | LF | “\” | “"”) ) | “\\”| “\n” | “\t” | “\"” } , “"”;

constant = “false” | “true” | “escape” | “e” | “pi” | “Pi” | “pI” | “PI”;

repeat paragraph = optional spacing, “repeat”, required spacing,
      ( “with”, required spacing, identifier, required spacing, “from”, required spacing,
        expression, required spacing, “to”, required spacing, expression,
          ( required spacing, “step”, required spacing, expression )
      )
      |
      ( expression, required spacing, “times”
      ),
      optional spacing, [comment], newline,
      { normal if paragraph
      | repeat paragraph
      | while paragraph
      | normal sentence
      | effectively empty line
      },
  optional spacing, “end”, required spacing, “while”, optional spacing, [comment], newline
;

expression = relExpr, optional spacing, ( (“and”, optional spacing, expression)
                                        | (“or”, optional spacing, expression) );

relExpr = addExpr, optional spacing,
    ( (“<” | “>” | “<=” | “>=” | “=” | “==” | “<>” | “<<>>”), optional spacing,  addExpr );

addExpr = {addExpr, optional spacing, “+”, optional spacing}, subExpr;

subExpr = {subExpr, optional spacing, “-”, optional spacing}, mulExpr;

mulExpr = {mulExpr, optional spacing, “*”, optional spacing}, divExpr;

divExpr = {divExpr, optional spacing, “/”, optional spacing}, powExpr;

powExpr = {powExpr, optional spacing, “^”, optional spacing}, strExpr;

strExpr = ( atomicExpr, optional spacing, “:”, optional spacing, atomicExpr )
          |
          ( atomicExpr, optional spacing, “@”, optional spacing, atomicExpr,
            (optional spacing, “;”, optional spacing, atomicExpr),
          )
          | atomicExpr;

localize sentence = optional spacing, “localize”, required spacing, identifier,
                    optional spacing, [comment], newline;

return sentence = optional spacing, “return”, required spacing, expression,
                  optional spacing, [comment], newline;
```

```
atomicExpr = ( "|", optional spacing, expression, optional spacing, "|", ("!") )
           | ( "(", optional spacing, expression, optional spacing, ")", ("!") )
           | constant
           | identifier ("!" | "?")
           | number ("!" | "%")
           | string
           | "not" atomicExpr
           | "~" atomicExpr
           | "-" atomicExpr
           | identifier, "[", optional spacing, expression, optional spacing,
                      { ",", optional spacing, expression, optional spacing }, "]"
           ;

subroutine definition paragraph = function definition | procedure definition;

function definition =
      optional spacing, "define", required spacing, "function", required spacing, identifier,
      (   "[", optional spacing, expression, optional spacing,
        { ",", optional spacing, expression, optional spacing }, "]" ),
      optional spacing, [comment], newline,
      { function if paragraph
      | repeat paragraph
      | while paragraph
      | normal sentence
      | effectively empty line
      | localize sentence
      | return sentence
      },
  optional spacing, "end", required spacing, "function", optional spacing, [comment], newline
;
```

3.13 Summary

A SLAWscript variable's data and/or data type can only be changed or initialized by subroutine calls with parameters and by the following sentence types: "copy", "get", "randomize", "repeat with", "set".

Auto-conversion produces temporary converted copies of the original values which are not stored for later use unless the auto-conversion occurs in the context of a "repeat with" or "set" sentence or a subroutine invocation with parameters.

Within the context of a subroutine, formal parameters are automatically localized and therefore "hide" global or higher-level local (i.e. the caller is another subroutine) variables with the same names until the end of the relevant execution cycle (i.e. recursion included) of the subroutine that did the hiding. Variables may be explicitly localized by using the "localize" verb. Subroutines called by another subroutine initially "inherit" its immediate caller's local variables (regardless of whether they were localized by formal parameters or explicitly) unless they were "hidden" by the callee's formal parameters. Subroutines more than two calls "deep" are not guaranteed initial access to all of the entire call chain's local variables, even in the absence of formal parameters in the most-recent subroutine, since a "parent" subroutine may have hidden a "grandparent" or "great-grandparent" etc. local variable either through the use of formal parameters or through the use of the "localize" verb.

Section 4: Project Plan

4.1 Project Overview

4.1.1 Purpose, Scope, and Objectives

The purpose of this project is to design, implement and test the SLAWscript language. Please refer to "Section 1: Introduction" for a complete description and overview of SLAWscript.

The scope of the project will be confined to the requirements and functionality described in the SLAWscript Language Reference Manual (Section 3). Every effort will be made to ensure the scope does not exceed the limits prescribed in this document. The objectives of this project are as follows.

4.1.2 Assumptions, Constraints and Risks

The SLAWscript project operated under the following assumptions:

1. A functional, well-designed scripting language is both interesting and appropriate for Columbia University's Programming Languages and Translators course.
2. Our project should touch on the various aspects of the course, so as to complement and extend what we learned in class.

The SLAWscript project faced the following constraints:

1. The project must be implemented, tested, and documented within one semester (10 weeks).
2. The project team is comprised of four students.
3. The project must be led by a single person.

The SLAWscript team faced the following risks, and adopted the corresponding mitigating measures:

1. Scope expansion. As the project was implemented, it was tempting to expand the functionality of the language. We mitigated this temptation by carefully adhering to the scope prescribed in our submitted Language Reference Manual.
2. Failure to Implement Language Features on Time. Throughout this project, we faced the possibility that our design team would not be able to complete all the features outlined in the Language Reference Manual on time. We overcame this risk by carefully selecting project priorities and allowing ourselves several fall back points. This increased our focus and allowed us to finish most of the language features we desired on schedule.
3. Architecture Risks. We faced inherent risks posed by the particular development tools we opted to use. For example, our team decided early on to use a beta version of ANTLR (3.0) and ANTLRWorks for front-end development. We managed this risk by adopting our language implementation to fit the capabilities of the applications we used, and on several occasions, by working directly with the ANTLR and ANTLRWorks authors to fix bugs.
4. Team Dynamics. As with all group projects, our team faced the risk of not normalizing relationships and roles as the project moved forward. Although we did experience natural friction, we used increased communication, face-to-face meetings, and informal team activities (such as dinner and deep sea fishing) to speed up normalization of our team.
5. Version Control. Because we developed and implemented the language in a collaborative manner, we faced risks posed by multiple developers accessing shared code and documentation. These risks included accidentally overwriting source code, comments, and documentation. Moreover, the inherent coordination involved with synchronizing team efforts might detract from meaningful work. We overcame this risk by standing up a Subversion (SVN) repository and rigorously enforcing its use.
6. Data Loss. We faced a minor risk of losing actual source code or documentation. Our use of the SVN repository mitigated this risk as individual copies of the repository served as backup.

4.1.3 Project Deliverables

• Project Proposal. The project proposal is a one-page document describing the SLAWscript language. The purpose of the proposal is to identify the initial concept for the language and set an initial project scope. This documents was due 7 February 2007 via electronic submission to the course instructor.

• SLAWscript Language Reference Manual. The SLAWscript Language Reference Manual is a complete and concise description of the language, its features, and syntax. This deliverable is due 5 March 2007 via electronic submission to the course instructor.

• SLAWscript Report and Source Code. The SLAWscript Report and source code are due at 11:59pm, 7 May 2007 via electronic submission to the course instructor. The purpose of the deliverable is to communicate to the instructor our efforts for the project. The submission will include a specific formatted report and all team generated source code for SLAWscript.

• SLAWscript Presentation. The SLAWscript Presentation is an oral presentation involving the course instructor and the team participants. The purpose of this deliverable is to demonstrate the working language, discuss its design and implementation, and answer questions from the course instructor. The presentation must be made on 7 May 2007.

4.1.4 Schedule Summary

Mandated Milestones:

Deliverable	Date
Project Proposal	7 February 2007
Language Reference Manual	5 March 2007
Project Report and Source Code	7 May 2007
Project Presentation	7 May 2007

4.2 Project Processes

4.2.1 Planning

Our primary planning process revolved around a collaborative web resource called BaseCamp (http://www.basecamphq.com/). BaseCamp provides management of a project calendar, deliverables, and individual tasks. It also includes mechanisms for posting messages and files related to various aspects of the project.

Using BaseCamp, we first entered the deliverables defined on the course website. BaseCamp automatically adds these deliverables to the project calendar which provides a graphical representation of the various project milestones, complete with countdown to the next deliverable. Over the course of the project, we augmented these deliverables with specific tasks to individuals, such as “Levi – establish SVN repository NLT 8 FEB.” These tasks are automatically color coded by BaseCamp according to their due dates (e.g. red indicating overdue tasks). The team leader and team members can then use BaseCamp’s intuitive Graphical User Interface to monitor overall deliverables as well as individual task assignments. Additionally, BaseCamp’s messaging system provides a way to comment on individual tasks, and quickly disseminate this comments to all team members electronically.

We completed our use of BaseCamp with an ad hoc and informal set of internal milestones and goals established during team meetings and via email. These are roughly captured, along with the major project deliverables, in the Gantt Chart shown in Paragraph 4.4 (Project Time Line).

4.2.2 Specification

We carefully used our Language Reference Manual (LRM) as the main specification process for the project. We solidified the LRM early in the project and referenced it daily, both in individual development activities and team meetings. *Taking the time early on to ensure this document was complete and detailed paid huge dividends in our project.* We added the LRM to our SVN repository where it could be easily maintained and referenced from any web browser.

Occasionally, we used code comments to capture ongoing or pending changed to the LRM. For example, if the front end team identified a change in the LRM, they would write an in-line Java comment of the form "`// TO DO: Change specification for ??`". We routinely scanned and reviewed these comments and integrated them into the LRM at various points in the project.

4.2.3 Development

A centralized SVN repository formed the backbone of our development process. The repository was populated with several directories – one for front end and back-end development, one for testing, one for documentation, and several for early prototyping. When developing application code, test scripts, or documentation, team members first updated their local copies of the repository, then added and committed any changes with self explaining comments. The team leader routinely scanned these directories for modifications, and discussed coordination and discrepancies via email or in team meetings.

Each team member used slightly different development processes for code development, testing, and document management. The overall code development was done by editing individual text files on various platforms: GNU/Linux, Mac OS X, Microsoft Windows, Solaris. Some team members did use the Eclipse Integrated Development Environment, but all project development was executed using individual text files.

Please refer to Paragraph 4.8 (Software Environment) for additional details on the development process.

4.2.3 Testing

Our testing process consisted of two main activities: unit testing and integrated testing. Unit testing was provided by a series of raw SLAWscript files, each designed to test a specific aspect of the SLAWscript interpreter. These files were individually run against the interpreter, and the results were analyzed for expected output. Usually, we executed such tests immediately following, or during, development of specific functionality in the interpreter, e.g. after developing the "repeatWhile" functionality. However, we did execute individual tests many times during the course of development as new pieces of code were added to the project.

Integrated testing involved a series of chained SLAWscript programs that thoroughly tested all aspects of the SLAWscript language. These were designed to run in batch mode, producing output that we could analyze for correctness. This test output provided invaluable feedback to the front end and back-end teams about particular language features that needed improvement.

Please refer to the section 6 ("Test Plan") of this document for a complete discussion of the testing process.

4.3 Programming Style Guide

The SLAWscript team used the following guidelines in order to smooth the development process.

- We do not follow the Java convention of starting a class' name with a capital letter if and only if the class is to represent a reserved word in our language, e.g. "setSentence.java", "andExpr.java".
- We prevent default constructors from being callable where the object does not make sense without data (as most of them don't), in order to prevent bugs.
- We use the "final" keyword on variables that won't change, also in order to prevent bugs.
- We put a space between "if" and the '(' that follows it (since "if" is not a method that is being invoked).
- We put at least one space after "//" and before the first non-white-space character following it on the same line.
- We do not use tab characters in our files. We use two spaces for each level of indentation.
- We do not submit text (including Java and SLAW) files (including updates to existing files) to the repository with DOS or old-Mac newlines; we submit files (including updates) with Unix newlines.
- We use "member___" (triple-underscore at the end) as a prefix for class data members.
- We do not use "this." to access class members.
- We write "TO DO" (space included, all-upper-case) in a comment indicating something that needs to be fixed, added, or improved, but we don't know how or have time to do it as of the writing of the "TO DO".
- We check for NaNs and infinity and negative-infinity after each math operation on "double" data (i.e. all SLAW numbers); if one of those conditions was found, then print out an error message and halt the program. Hints: "java.Double.isNaN()", "java.Double.isInfinite()".
- We send all of our error messages to "System.err", not to "System.out".
- We use "long" instead of "int" for integers that come from SLAWscript numbers, e.g. the integer in "repeat … times", since a double can have a number that rounds to an integer that is greater than MAX_INT or less than MIN_INT.
- We convert from "double" to "long" using rounding; we do not just cast it over, which causes truncation (e.g. 1.6 → 1).
- We document our authorship, thusly: the first person to write a file puts…
  `// this file was written by (author)`
  ... near the top; the second person to edit it changes it to…
  `// this file was written by (author1) and (author2)`
  ... and the third or fourth person changes it to…
  `// this file was written by (author1), (author2), and (author3)`
  ... or…
  `// this file was written by (author1), (author2), (author3), and (author4)`

- We use JavaDoc class description blocks at the top of every class. These start and end with a line of asterisks. These should be as descriptive as possible, and include links to the language reference manual. This should include TO DO comments as well as author information. Example:

```
/**************************************************************
 *
 * The repeatTimesParagraph class repeats a block of code
 * a particular number of times (pg 12, LRM): <br><br>
 *
 *
 * An example follows.
 * <code><pre>
 * repeat 999999999 times
 *   put 'Number 9... ' to stdout
 * end repeat
 * </pre></code>
 *
 * @see <a href='../SLAWscript.html#Repeat_Times'>Repeat Times in Language  Reference Manual</a>
 *
 * @author Abe, Steve
 *
 **************************************************************/
```

- We use JavaDoc comment blocks at the beginning of each attribute, method, and constructor. These should included pertinent JavaDoc tags where applicable. These are formatted according to the following examples:

```
/**
 * The code inside the repeat block...
 */
private Vector<NormalParagraphOrNormalSentence> member___code;

/**************************************************************
 *
 * Attempt to return the Variable as a double.  *
 * @return The Variable as a double; produce an error
 * if variable is a non-numeric string that can't be converted
 * to a double
 *
 **************************************************************/
public double get_as_a_number() { …

 /*********************************************************
 *
 * Creates a new repeatTimesParagraph object which will
 * repeat the supplied code inTimes.
 *
 * @param code
 * @param inTimes
 *********************************************************/
public repeatTimesParagraph(Vector<NormalParagraphOrNormalSentence> code, UsableInExpressions timesExpr)
{ …
```

4.4 Roles and Responsibilities

4.4.1 Internal Structure

The SLAWscript team's work was broken down into four major activities: *Front-end development, Back-end Development, Testing,* and *Documentation.* With the exception of documentation, specific team members were assigned to each activity. The team member assigned to each activity had primary responsibility for completion of that activity. The interface between the activity sub-teams consisted of three tiers of communication, as described below:

• Tier 1: Conventional E-mail. Conventional e-mail between team members was used for regular and trivial coordination among the team. This included progress reports to the rest of the team or e-mails asking for support or answering questions. This tier was not used for distributing important and persistent information (e.g. updated source code) to the team.

• Tier 2: Project Website (BaseCamp). A project team website was established (described in Paragraph 4.2.1) and used for formal and persistent communication among the team.

• Tier 3. Weekly Project Meetings.

4.4.2 Roles and Responsibilities

Front-end Development Activity

1. Responsibilities: The Front-end Development Activity sub-team was responsible for the following tasks:
    - Development of the ANTLR code required to build a SLAWscript lexer and parser.
    - Analysis of the Language Reference Manual for overall feasibility of language features.
    - Identification of language features that would not be easily supported by the front-end within the time constraints of the project.
    - Identification of the overall objects, interfaces, and constructs required by the back-end to manage the output of the ANTLR generated parser.
    - Identification of testing requirements for individual aspects and elements of the front-end.
2. Members: The Front-end Development Activity was done primarily by Abe Skolnik, with Steve Henderson providing support.

Back-end Development Activity

1. Responsibilities. The Back-end Development Activity sub-team was responsible for the following tasks:
    - Design and implementation of all classes invoked by the front-end-generated parser.
    - Design of the “main” class for the SLAWscript interpreter. This class creates the executable SLAWscript parser and handles such tasks as instantiating the front end lexer and parser, loading SLAWscript source files, and handling the results of the front-end parsed SLAWscript file.
    - Identification of individual testing requirements for specific aspects of the back end code.
2. Members. The Back-end Development Activity was done by Steve Henderson and Abe Skolnik.

Testing Activity

1. Responsibilities. The Testing Activity sub-team was responsible for the following tasks:
    - Conduct unit testing of all major classes.
    - Conduct integrated testing using more complex SLAWscript examples.
    - Provide feedback to the front-end and back-end sub-teams about problems that were uncovered.
2. Members. The Testing Activity was done mostly by Levi Lister.

Documentation Activity

1. Responsibilities. The Documentation Activity sub-team was responsible for the following tasks:
    - Oversee code documentation.
    - Publish regular JavaDoc updates to a shared location accessible by the team.
    - Draft the Project Proposal.
    - Draft and update the Language Reference Manual.
    - Draft the final project report.
2. Members. The Documentation Activity sub-team consisted of all the overall team members, with the following general responsibilities:
    - Proposal: Abe (lead); Levi and Steve supporting
    - Reference Manual: Abe (lead); Levi and Steve supporting
    - Code Documentation: Steve (lead), Abe supporting
    - Final Project Report: All; Abe (editor and publisher)

4.5 Software Development Environment

4.5.1 Overview

The software designed in this project consists mainly of a SLAWscript language interpreter that runs as a console-based Java application (implemented with the Java version 1.5 SDK). This application leverages Java classes provided by the ANTLR library (version 3.0b6). Specifically, the ANTLR library provides constructs that allow for an ANTLR grammar file (front-end) to produce an automatically generated lexer and parser. The lexer and parser operate in concert with custom-design Java classes (back-end) to parse and, if parsable, execute a SLAWscript file. This entire process is described in detail below.

4.5.2 Front-end Software Development Environment

The main component of the front-end development environment is the ANTLRWorks Graphical User Interface tool. This tool allows for the efficient and rapid development of the code required to build the ANTLR-generated lexer and parser for SLAWscript. The ANTLRWorks application runs as an executable Java program, and provides a powerful text editor for developing, analyzing, and testing an ANTLR grammar (SLAWscript's grammar, in our case). ANTLRWorks includes such features as ANTLR grammar checking, syntax highlighting, the display of syntax diagrams for lexer and parser rules, nondeterminism warnings, discrete finite automaton generation, and automatic code generation for the SLAWscript lexer and parser. The front end's primary component consists of a grammar-checked ANTLR ".g" file, named "SLAWscript.g" in our case. This file is used to produce the SLAWscript lexer and parser. This grammar file is kept in the team Subversion repository where team members can use their individual copies of ANTLRWorks to generate the lexer and parser.

4.5.3 Back-end Software Development Environment

The main components of the back end are multiple Java classes that together support the SLAWscript parser. These are described in detail in Section 5 (Architecture Design) of this report. We primarily used text editors (such as jEdit) to edit back-end class files, then tested them using individual SLAWscript files on the command line. These files were then updated and committed to the team's Subversion repository.

As the project grew, an overall build script that executes from the Unix command line was written. This script uses symbolic links to the source files in the back-end to produce a single packaged, executable "jar" file (with all byte-code based binaries) for the interpreter. This script greatly facilitated routine and rapid updates of the interpreter. It also includes the needed parts of ANTLR 3.0b6 into the generated "jar" file, so that users of SLAWscript do not need to install that separately, and so that the resulting "jar" file will not conflict with other versions of ANTLR that may already be installed on the user's machine. Care is taken to only include the parts of ANTLR that are needed at run-time, thereby saving several hundreds of kilobytes in the size of the "jar" file.

Section 5: Architectural Design

5.1 Architecture Overview

Fundamentally, our design consists of an interpreter that reads, parses, and, if possible, executes a SLAWscript file. A SLAWscript file is a text file that contains a SLAWscript program, and is conventionally named with a ".SLAW" extension. The interpreter processes this file and interacts with the user.

The SLAWscript interpreter runs as a console Java application, and can be described using the following abstract design layers: front-end and back-end. The front-end layer performs analysis of the SLAWscript file and creates an intermediate representation for use by the interpreter. The back-end layer uses this intermediate representation to execute the code in the SLAWscript file, if possible.

5.2 Front End Architecture

5.2.1 Front End Architecture Overview

The front end architecture consists of two principle components:

• SLAWscript Lexer (SLAWscriptLexer.java): A custom-designed lexer that performs the lexical analysis of the SLAWscript file.

• SLAWscript Parser (SLAWscriptParser.java): A custom-designed parser that uses the output from the lexer to perform syntactical analysis and creation of the intermediate representation of the SLAWscript source file.

The SLAWscript lexer and parser are relatively complex components each warranting robust and efficient designs. To facilitate their creation, our architecture leverages the ANTLR framework to generate the lexer and and parser. ANTLR allows developers to describe a language such as SLAWscript using a separate language designed for concise language specification. This allows for the entire front-end to be described with a single ANTLR grammar file. This file is then processed by ANTLR, which automates the creation of Java source code for the target language (here, SLAWscript).

5.2.2 Front-end Components

The SLAWscript front-end design defined in the ANLTR grammar file consists of the following principle components:

• Lexer Rules. The front-end design uses lexer rules to specifies the tokens in the SLAWscript language. The following examples demonstrate a few lexer rules:

```
Colon: ':';

Spacing: (' ' | '\t')* { $channel=HIDDEN; };

End_if: 'end' (' ' | '\t')+ 'if';

End_repeat: 'end' (' ' | '\t')+ 'repeat';
```

• Parser Rules. A series of powerful parser rules form the backbone of the front-end design. Theserules match the string literals in the SLAWscript file against the language constructs defined by the SLAWscript grammar. The matching rules use ANTLR rewrite syntax to create, on the fly, a set of instantiated SLAWscript Java Objects (hereinafter "SJO"s) that form the intermediate representation of the SLAWscript program. The following example demonstrates a rule used to match a
"while <expr> ... end while" block in SLAW.

```
whileParagraph returns [whileParagraph wp]
  @init {
    Vector<NormalParagraphOrNormalSentence> code = new
                          Vector<NormalParagraphOrNormalSentence>();
  }
  :
  'while' ex=expr EOL
    (normalParagraphOrNormalSentence
{ code.add($normalParagraphOrNormalSentence.npns); }
  | EOL)*
  End_while (EOF|EOL) { $wp = new whileParagraph($ex.uie,code); } ;
```

As shown in this example, this rule first matches the “while” literal and the expression in the SLAWscript. The rule then uses ANTLR's rewrite syntax to create a new whileParagraph SJO that contains the conditional expression for executing the “while” loop, as well as the code the execute in the “while” block. Please note that the whileParagraph and other SJOs are described in detail in paragraph 5.3.2.

5.2.3 Intermediate Representation

The parser (SLAWscriptParser.java) creates the intermediate representation as a single Java class (SLAWscriptReturnType.java). This class contains three collection attributes, each consisting of zero or more instantiated SJOs such as the one described in the previous paragraph.

• Main Body Code Collection. The main body code collection consists of an array of normal paragraphs or normal sentences (modeled as the SJO superclass NormalParagraphOrNormalSentence.java). These objects represent the various sentences and paragraphs (minus functions and procedure blocks) in the main body of the SLAWscript file and are ordered according to how they appear in the SLAWscript file.

• Functions Collection. The functions collection consists of a Java hash table that defines the function identifiers and corresponding code for each function in the SLAWscript file.

• Procedures Collection. The procedures collection consists of a Java hash table that defines the procedures identifiers and corresponding code for each procedure in the SLAWscript file.

These three collections embody all the actual code in the parsed SLAWscript file, and as such do not contain any comments or whitespace.

5.3. Back-end Architecture

5.3.1 Back-end Architecture Overview

The interpreter's main class (which is contained in “SLAWscript.java”) is the heart of the SLAWscript back end. This single class reads the SLAWscript file and then instantiates and uses the front-end lexer and parser to analyze its contents. The interpreter then instantiates a variable stack (as defined in “VariableStack.java” and described in paragraph 5.7) that serves as the symbol table for all variables in the program. Finally, the interpreter iterates over the main body collection contained in the SLAWscriptReturnType. This iteration corresponds to the evaluation that occurs at each node of an Abstract Syntax Tree. The following paragraph describes this process in further detail.

5.3.2 Iteration of Main Body SLAWscript Java Objects (SJOs)

Iteration over the main body code is trivial, as the main functionality for SLAWscript constructs is modeled inside the individual SJOs (described in subsequent paragraphs). The interpreter simply locates the next element in the main body array, that element being a NormalParagraphOrNormalSentence object (which was already instantiated by the front end). The NormalParagraphOrNormalSentence class is an abstract superclass for all non-subroutine-definition-related SLAWscript constructs, and contains a single method – doYourThing(). The implementing subclass (e.g. whileParagraph) implements the doYourThing() method to perform a particular aspect of SLAWscript (e.g. the execution of a "while" loop).

Subroutines (including both procedures and functions) are executed, when needed, inside this iteration. Subroutines can only be invoked by other SLAWscript constructs, and are thus triggered within a NormalParagraphOrNormalSentence doYourThing() method (e.g. within the execution of "do myProcedure" and of "set x to resultOfFunction[42]"). The modeling and functionality of subroutines is described below.

5.3.3. Design of SLAWscript Java Objects (SJOs)

5.3.3.1 Top-level Abstract Classes

Figure 1 denotes three top-level abstract classes that are used to derive all SJOs in the design.

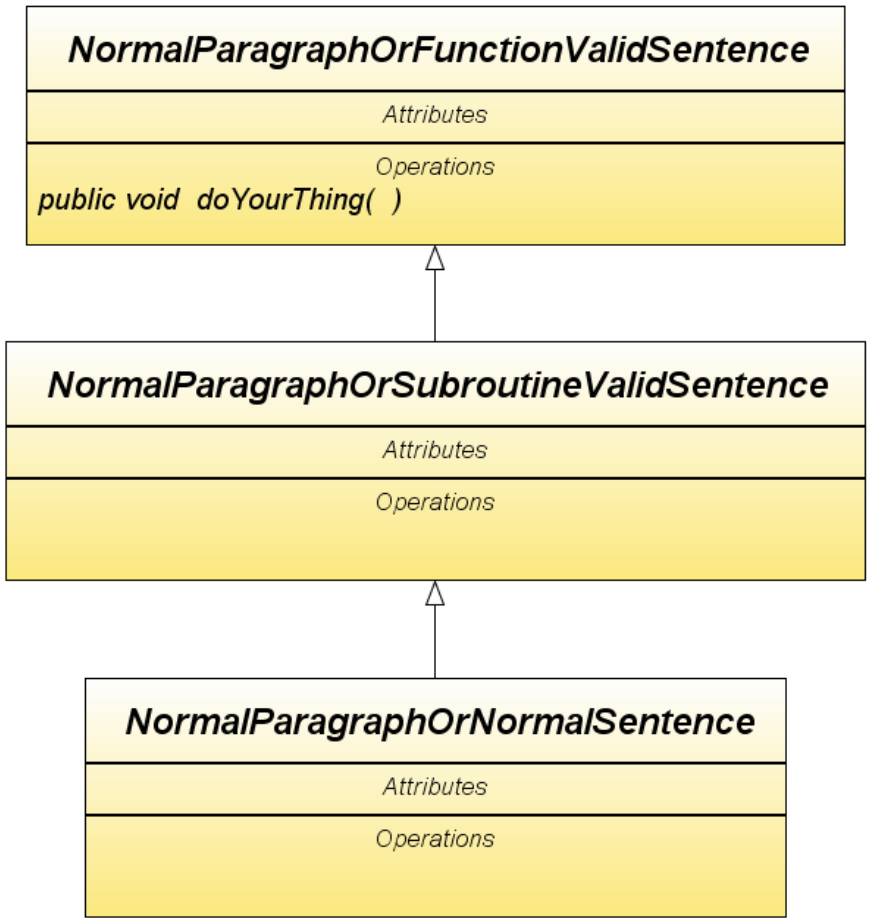

Figure 1: Top Level Abstract Classes and Interfaces

These top-level classes are described on the following pages.

• The **NormalParagraphOrFunctionValidSentence** object is the top superclass of most of our non-expression-related classes in the software design. The NormalParagraphOrFunctionValidSentence contains one abstract method: doYourThing(). As described in previous paragraphs, the concrete doYourThing() methods are called by the interpreter during iteration of the main body code, and represent the program functionality prescribed for a particular SLAWscript construct. Only two SJOs directly extend the NormalParagraphOrFunctionValidSentence superclass: returnSentence and FunctionIfParagraph (please see paragraph 5.6.2).

• The **NormalParagraphOrSubroutineValidSentence** subclass serves as a "parent" to non-expression-related SJOs that may appear inside of subroutines. Only two classes directly extend this abstract class: NormalParagraphOrNormalSentence and localizeSentence.

• The **NormalParagraphOrNormalSentence** subclass serves as a "parent" to non-expression-related SJOs that aren't related to subroutines (either function or procedures). For example, the "whileParagraph" object models a "while <expr>... end while" loop in SLAWscript.

5.4 The "Constant" Class

The **Constant** class is used to model a constant in SLAWscript, e.g. 5.0, -5.5, "Hello", etc. The Constant class implements the UsableInExpressions interface, and can therefor be used in any SLAWscript expression. Note: this class implements the evaluate() method by simply returning the Constant object, as it's already a constant. This class supports any constant in SLAWscript, i.e. both strings and numbers. Several methods exist so as to allow the use of those methods to check for the return type (based on the contents of the Constant), which can be either a string or a number. The class also offers appropriate accessors to retrieve the Constant's value as either a string or (if possible) a double-precision floating-point number.

5.5 The UsableInExpressions Interface and its Implementations

The **UsableInExpressions** interface is an important interface that is implemented by all SJOs that can be evaluated inside of expressions: constants, logical expressions, mathematical/string expressions, identifiers, etc. The UsableInExpressions interface contains only one method to be implemented: "evaluate()", which returns a Constant object. This method ensures that any implementing class can be properly evaluated in an expression.

Classes that implement the UsableInExpressions class can be divided into three main categories: logic expressions, mathematical/string expressions, and utility expressions.

5.5.1 Logic Expressions

andExpr: This class models boolean AND. It requires two member UsableInExpressions objects: one for the left side and one for the right side. When the implemented method evaluate() is called for andExpr(), each of these is evaluated to a constant, and compared accordingly (using short-circuited evaluation). The class then returns the result as a new boolean Constant (a number in the set {0,1}).

notExpr: This class models boolean NOT. It is a unary expression that requires a single operand. When the implemented method evaluate() is called for notExpr, this operand is checked for its boolean value (by comparing it to 0). The class then returns the result as a new boolean Constant.

orExpr: This class models boolean OR. It requires two member UsableInExpressions objects: one for the left side and one for the right side. When the implemented method evaluate() is called for orExpr(), each of these is evaluated to a constant (using their own implemented "evaluate()" methods), and compared accordingly (using short-circuited evaluation). The class then returns the result as a new boolean Constant.

5.5.2 Mathematical and String Expressions

DivExpr: This class models mathematical division. It contains a single attribute: a Java vector of UsableInExpressions objects, which allows for chained division operations. When the implemented evaluate() method is called, the class iterates over this collection, evaluating each UsableInExpression (via its "evaluate()" method), and computing an overall result. The class then returns this result as a new Constant (numeric).

FactorialExpr: This class models the factorial operator ('!') that is popular in mathematics. It requires a single member containing the operand to undergo factorial multiplication. When the implemented evaluate() method is called, the class first evaluates the operand (by calling its evaluate() function) which returns a Constant. The class then attempts to retrieve the numeric value of this constant using the procedure described in paragraph 5.4. If successful, the class then uses a helper class (Factorial.java) to evaluate the result. The class then returns this result as a new Constant (numeric).

GreaterThanExpr: This class models the greater-than expression ('>'). It requires two member UsableInExpressions objects: one for the left side and one for the right side. When the implemented method evaluate() is called for GreaterThanExpr, each of these are evaluated to a constant, and compared accordingly. The class then returns the result as a new boolean Constant.

GreaterThanOrEqualExpr: This class models the greater-than-or-equal-to expression (“>=”). It requires two member UsableInExpressions objects: one for the left side and one for the right side. When the implemented method evaluate() is called for GreaterThanOrEqualExpr(), each of these are evaluated to a constant, and they are then compared accordingly. The class then returns the result as a new boolean Constant.

LessThanExpr: This class models the less-than expression ('<'). It requires two member UsableInExpressions objects: one for the left side and one for the right side. When the implemented method evaluate() is called for LessThanExpr(), each of these are evaluated to a constant (using their respective “evaluate()” methods) and compared accordingly. The class then returns the result as a new boolean Constant.

LessThanOrEqualExpr: This class models the less-than-or-equal-to expression (“<=”). It requires two member UsableInExpressions objects: one for the left side and one for the right side. When the implemented method evaluate() is called for LessThanOrEqualExpr(), each of these are evaluated to a Constant, and compared accordingly. The class then returns the result as a new boolean Constant.

MinusExpr: This class models mathematical subtraction. It contains a Java vector of UsableInExpressions objects, thus allowing chained subtraction operations despite our innovative parsing technique combined with ANTLR forbidding left recursion. When the implemented evaluate() method is called, the class iterates over this collection, subtracting elements as it goes. The class then returns the result as a new Constant (numeric).

MulExpr: This class models mathematical multiplication, but includes functionality to multiply a string by a number or a numeric string (please see section 3: “Language Reference Manual”). It contains a Java vector of UsableInExpressions objects, thus allowing chained multiplication operations. When the implemented evaluate() method is called, the class iterates over this collection, evaluating each UsableInExpressions contained in its Java vector, and tracking an overall product (if numeric multiplication) or an expanded string (if string multiplication). Because of the potential for mixed data types, the class checks to ensure the overall expression is valid. If successful, the class then returns the result as a new Constant (either numeric or string depending on the supplied operands in the vector collection).

PipeExpr: This class models both the absolute value function in mathematics and the string length function. It requires a single member containing the operand to undergo the function. When the implemented evaluate() method is called, the class first evaluates the operand (by calling its “evaluate()” function) which returns a Constant. The class then either returns the length of the string, or returns the absolute value of the evaluated constant. Either way, it returns the result as a new numeric Constant.

PlusExpr: This class models both mathematical addition and string concatenation (please see section 3: “Language Reference Manual”). It contains a Java vector of UsableInExpressions objects, thus allowing us to perform chained addition/concatenation operations. When the implemented evaluate() method is called, the class iterates over this collection, adding or concatenating elements as it goes. This is accomplished by evaluating the individual UsableInExpressions objects within its Java vector (via their implemented “evaluate()” methods). Because of the potential for mixed data types, the class ensures the overall expression is valid. If successful, the class returns the result as a new constant (either numeric or string data depending on the supplied operands in its collection).

PowerExpr: This class models the mathematical exponent/power symbol ('^'). It requires two member UsableInExpressions objects: one for the base and one for the exponent. When the implemented method evaluate() is called, each of these are evaluated to a constant (using their respective “evaluate()” methods). The class then attempts to retrieve the results of these individual evaluations as numbers, and if successful, uses Java's Math.pow() to compute the result. The class then returns the result as a new numeric constant.

RelaxedDoesNotEqualExpr: This class models the relaxed inequality (“<>”) expression. It requires two member UsableInExpressions objects: one for the left side and one for the right side. When the implemented evaluate() method is called, each of the operands are evaluated to a constant (using their respective “evaluate()” methods) and compared accordingly. The class then returns the result as a new boolean constant.

RelaxedEqualsExpr: This class models the relaxed equals symbol ('='). It requires two member UsableInExpressions objects: one for the left side and one for the right side. When the implemented evaluate() method is called, each of these are evaluated to a constant (using their respective “evaluate()” methods) and compared accordingly. The class then returns the result as a new boolean constant.

StrictlyDoesNotEqualExpr: This class models the strict inequality expression (“<<>>”). It requires two member UsableInExpressions objects: one for the left side and one for the right side. When the implemented evaluate() method is called, each of these are evaluated to a constant (using their respective “evaluate()” methods) and compared accordingly. The class then returns the result as a new boolean constant.

StrictlyEqualsExpr: This class models the strict equality expression (“==”). It requires two member UsableInExpressions objects: one for the left side and one for the right side. When the implemented evaluate() method is called, each of these are evaluated to a constant (using their respective “evaluate()” methods) and compared accordingly. The class then returns the result as a new boolean constant.

5.5.3 Utility expressions

Identifier: The Identifier class models a variable name in SLAWscript, e.g. “cats”, “do_not”, “eat_slaw”. The evaluate() method of this class' UsableInExpressions implementation does the following process: determine if the identifier represents a zero-parameters function, and if so invoke that function; otherwise, query the variable stack, find the target variable, and fetch its current value as a constant.

InstrExpr: The InstrExpr class implements the ':' operator, which attempts to find the right operand's string in the left operand's string. The class contains two member UsableInExpressions objects for the right and left operands. Upon evaluation, the class evaluates the individual operands (using their “evaluate()” methods) and uses the Java String.indexOf() function to perform the lookup. It then returns a new numeric constant.

RoundExpr: The RoundExpr class implements the mathematical rounding of a UsableInExpressions object's return value, which must be either a number or a numeric string. When evaluated, this class first evaluates its single UsableInExpressions member to a numeric constant, and then executes Java's Math.round() function and returns the result as a new numeric constant.

SingleQuestionMarkExpr: This class implements the variable content operator ('?') for a given identifier. During evaluation, the class queries the interpreter's variable stack (“VariableStack.java”) for the target variable. The class then uses the variable's data-type determination methods (“is_a_number()”, “is_usable_as_number()”, etc.) to determine its type. The class then returns a new numeric constant in the set {0, 1, 2}.

SubstrExpr: The SubstrExpr implements the '@' operator, which returns a substring of the original string. It has three UsableInExpressions members: one for the original string, one for the position, and an optional (i.e. may validly be “null”) member for the limit of the substring. During evaluation of the '@' operator, these UsableInExpressions objects are evaluated, and bounds checking is performed to ensure the supplied ranges are valid. If there are no errors, the classes uses its members' evaluated constants and the Java String.substring() method to create the result. This result is returned as a new string constant.

5.6 Sentences and Paragraphs

Paragraph 5.5 and its subparagraphs detailed the SJOs that implement the UsableInExpressions interface. The SLAWscript architecture also has additional SJOs that leverage these expression classes for more complex functionality. These can be roughly divided into two categories: non-subroutine sentences and paragraphs, and subroutines' sentences and paragraphs.

5.6.1 Non-Subroutine Sentences and Paragraphs

SLAWscript's non-subroutine sentences and paragraphs each extend the NormalParagraphOrNormalSentence superclass. These SJOs can be grouped into the following categories: program execution sentences and paragraphs, loop constructs, input and output sentences, and utility sentences.

5.6.1.1 Program Execution Sentences and Paragraphs

ignoreSentence: The ignoreSentence class implements the "ignore" keyword. It contains a single member, which captures the UsableInExpression class to evaluate and then ignore the result. Upon execution of its doYourThing() method, the ignoreSentence class evaluates its single member (using the member's "evaluate()" method), but does not capture the result.

NormalIfParagraph: This class models a SLAWscript "if" paragraph which is not able to contain either a "localize" or a "return". It contains four members:

- A vector of UsableInExpressions objects as the conditions,
- A vector of NormalParagraphOrNormalSentence objects as the body of the "if" block,
- A double vector of NormalParagraphOrNormalSentence objects as the bodies of the "else if" blocks,
- A vector of NormalParagraphOrNormalSentence objects as the body of the terminating "else" block.

During execution of its doYourThing() method, the class first evaluates its first condition (using the UsableInExpressions object's "evaluate()" method). If the conditions hold, the class then iterates over the vector of objects that represent the "if" body code (using their respective "doYourThing()" methods). If the "if" condition is not met, the class iterates over its "else if" conditions (if any exist), and, if evaluated as true, executes where appropriate. If these also do not match, the class will iterate over the SJOs in the vector representing the terminating "else" code, if that exists.

stopSentence: This class models the 'stop' keyword. Upon invocation of its doYourThing() method, the class reports the line number of the "stop" sentence and then terminates the interpreter.

5.6.1.2 Loop Constructs

repeatParagraph: The repeatParagraph class is an abstract class that acts as a superclass to the repeatTimes and repeatWith subclasses. It contains no methods or members.

repeatTimesParagraph: The repeatTimesParagraph class models a "repeat … times" SLAWscript block. It contains one member for the repeat code block (A vector of NormalParagraphOrNormalSentence objects) and another member (of UsableInExpressions type) for the loop counter. Upon invocation of its doYourThing() method, the class evaluates its UsableInExpressions counter. It then sets up a simple Java "for" loop. For each iteration of this loop, the class performs a complete iteration of its code vector, executing the doYourThing() method for each NormalParagraphOrNormalSentence in its vector.

repeatWithParagraph: The **repeatWithParagraph** class models a "repeat with…" SLAWscript block. It contains a vector of NormalParagraphOrNormalSentence objects for the loop's code body, and UsableInExpressions members for the from, to, and step variables. Members are also provided to indicate if a default step is in use (please refer to Section 3 for details) and for identifying the counter. Execution of this class' doYourThing() method functions similar to that of the repeatTimesParagraph class, described above.

whileParagraph: The whileParagraph class models a "while…" SLAWscript block. The class contains one member for the conditional (a UsableInExpressions object) and a vector of NormalParagraphOrNormalSentence objects for the body. Upon execution of its doYourThing() method, the class sets up a Java "while loop", with the "while" loop's conditional continually evaluated against the SJO's conditional (via its "evaluate()" method). If the conditional holds, the class will perform a complete iteration of its code vector, executing the doYourThing() method for each NormalParagraphOrNormalSentence in its vector.

5.6.1.3 Input and Output Sentences

getSentence: The getSentence class models SLAWscript input. It contains a single member, which indicates the target variable for the input. Upon invocation of its doYourThing() method, the class uses the Java BufferedReader and InputStreamReader classes to gather the input. It then passes this as a new Variable to the target identifier in the variable stack using the VariableStack.put(…) method.

putSentence: The putSentence class models SLAWscript output. The class contains a member for the source expression, which is evaluated during execution of the class doYourThing() method, as well as a boolean member for keeping track of whether the output is to go to Standard Error or Standard Out.

5.6.1.4 Utility Sentences

copySentence: The copySentence copies the value of one variable to another. It accomplishes this by retrieving a datum from the variable stack, then sending that same datum to the variable stack with a different identifier.

randomizeSentence: The randomizeSentence class performs the numerical randomization of a variable by utilizing the Java Math.random() method.

assertSentence: The assertSentence models the SLAWscript "assert" sentence type. This class contains two members, corresponding to the checked identifier and the compared constant. Upon execution of its doYourThing() method, the class looks up the identifier in the variable stack, then uses the variable's accessors to compare it with the constant. The class will exit via the Java System.exit method if the assertion fails.

setSentence: The setSentence class models the SLAWscript "set" sentence type. Upon execution of its doYourThing() method, the class evaluates the expression, then uses the variable stack's put(…) method to update the variable's value.

5.6.2 Subroutine Sentences and Paragraphs

The SLAWscript architecture includes several classes that model the functionality required to execute subroutines (procedures and functions). These are described in the following paragraphs.

doSentence: The doSentence class represents a SLAWscript command to execute a procedure. This class extends the NormalParagraphOrNormalSentence class. The class has a member representing the name of the target procedure, and an array of UsableInExpressions objects that represent the parameters passed to the procedure. Upon execution of its doYourThing() method, the class first iterates over its array of parameters, and evaluates each of them (using the implemented "evaluate()" method for each UsableInExpressions object). During this process, the class builds a second array with the evaluated parameters (now all Constants). It then uses the name of the procedure to look up the procedure in the interpreter's procedure collection (described in paragraph 5.2.3). If the procedure is located, the class executes the procedure's doYourThing(Constant[]) method, passing it the array of evaluated constants. If the procedure is not located, the interpreter is aborted.

Function: The Function class models a function in SLAWscript. The function is a special class that does not extend any of the high-level superclasses described in Paragraph 5.3.3.1. The class contains three members: an array of NormalParagraphOrFunctionValidSentence objects for the function code, an array of strings for the formal parameter names, and a member for the function name itself. The class contains a single method, "doFunction(Constant[])", which is used to invoke the SLAWscript function (from a FunctionCallWithParams or Identifier object's "evaluate()" method) and returns a constant containing the function's derived value. When the doFunction method is called, the class first establishes a new context using the variable stack's new_context() method. This allows the class to impose a new scope on its contents. The class then copies the incoming parameters into this new scope. The class then iterates over its code array (using the "doYourThing()" methods of its main body code array members). Special attention is paid to the "return" keyword, which is handled via introspection rather than via the usual "doYourThing()" method. If the function does not execute a "return" statement before ending, the class informs the user of the failure and aborts the interpreter. Otherwise, the return value is then passed back to the calling class as a Constant.

FunctionCallWithParams: This class serves to model a SLAWscript invocation of a function with parameters. It contains two members, which are both supplied in its construction: a string representing the function's name, and an array of UsableInExpressions objects representing the function's actual parameters. During class construction, the supplied function name is compared (via the Validator helper class) against function names in the interpreter. If the function name is valid, the FunctionCallWithParams object is created and the supplied array of parameters is mapped to the member "member___ actual_parameters". When the class' evaluate() method is called, the class attempts to locate the function name in the interpreter's hash table of Function objects (see paragraph 5.2.3). If the function is found, the class then evaluates its actual parameters and passes them to the target function's doFunction method. The result of this invocation (a Constant) is then returned for the evaluation of the FunctionCallWithParams. If the function is not found, then the interpreter is aborted.

FunctionIfParagraph: This class models a SLAWscript "if" paragraph which is able to contain either a "localize" or a "return" in addition to all the sentence and paragraph types that are valid in an "if" paragraph which occurs in main-body code. It contains four members:

- A vector of UsableInExpressions objects as the conditions,
- A vector of NormalParagraphsOrFunctionValidSentence objects as the body of the "if" block,
- A double vector of NormalParagraphsOrFunctionValidSentence objects as the bodies of the "else if" blocks,
- A vector of NormalParagraphsOrFunctionValidSentence objects as the body of the terminating "else" block.

During execution of its doYourThing() method, the class first evaluates its first condition (using the UsableInExpressions object's "evaluate()" method). If the conditions hold, the class then iterates over the vector of objects that represent the "if" body code (using their respective "doYourThing()" methods). If the "if" condition is not met, the class iterates over its "else if" conditions (if any exist), and, if evaluated as true, executes where appropriate. If these also do not match, the class will iterate over the SJOs in the vector representing the terminating "else" code, if that exists. Special attention is paid to the "return" sentence type, which is handled via introspection rather than via the usual "doYourThing()" method. The same is true of the instance of an "if" paragraph inside another function-specific (i.e. ["localize"/"return"]-enabled) "if" paragraph.

localizeSentence: The localizeSentence class is used to implement the "localize" keyword in SLAWscript. The class extends the NormalParagraphOrSubroutineValidSentence superclass. It contains a single member, which corresponds to the name of the variable to be localized. It accomplishes localization by invoking the variable stack's "reserve(String)" method.

Procedure: The Procedure class models a SLAWscript procedure. Like "Function", the procedure class is a special class that does not extend any of the high-level superclasses described in Paragraph 5.3.3.1. The class contains three members: an array of objects for the procedure code, an array of strings for the formal parameter names, and a member for the procedure name itself. The class contains a single method, "doProcedure(Constant[])", that is used to invoke the procedure (from the "doSentence" class, described above). When the doProcedure method is called, the class first establishes a new context using the variable stack's new_context() method. This allows the class to impose a new scope on its contents. The class then copies the incoming parameters into this new scope. The class then iterates over its code array (using the "doYourThing()" methods of its main body code array members).

ProcedureIfParagraph: This class models a SLAWscript "if" paragraph which is able to contain a "localize" in addition to all the sentence and paragraph types that are valid in an "if" paragraph which occurs in main-body code. It contains four members:

- A vector of UsableInExpressions objects as the conditions,
- A vector of NormalParagraphOrSubroutineValidSentence objects as the body of the "if" block,
- A double vector of NormalParagraphOrSubroutineValidSentence objects as the bodies of the "else if" blocks,
- A vector of NormalParagraphOrSubroutineValidSentence objects as the body of the terminating "else" block.

During execution of its doYourThing() method, the class first evaluates its first condition (using the UsableInExpressions object's "evaluate()" method). If the conditions hold, the class then iterates over the vector of objects that represent the "if" body code (using their respective "doYourThing()" methods). If the "if" condition is not met, the class iterates over its "else if" conditions (if any exist), and, if evaluated as true, executes where appropriate. If these also do not match, the class will iterate over the SJOs in the vector representing the terminating "else" code, if that exists.

returnSentence: The returnSentence class extends the NormalParagraphOrFunctionValidSentence superclass only for the purpose of fitting in with the rest of the object-oriented design of the SLAWscript back-end. In this case, the "doYourThing" method, which only exists because it must exist in order for this class to have the superclass that it must, is actually a forbidden function, since return sentences (in the SLAWscript back-end) must be detected via introspection, and then have their "getReturnValue()" methods called, in order to retrieve the constant that corresponds to the function's return value. This is primarily due to the fact that the "doYourThing" interface method was intentionally designed to not return anything, since sentences and paragraphs normally do not return any data. (The exceptions are "return" sentences, for the obvious reasons, and "if" paragraphs of the type modeled by the FunctionIfParagraph class, which may return a datum.)

5.7 Helper Classes

The SLAWscript architecture also contains several important helper classes. These don't provide the functionality of a sentence or paragraph type or of an operator, but instead provide functionality which is needed in order to implement the functionality of the classes described above.

Constant: This class models a runtime constant in SLAWscipt. This class contains three members that are used to model the universal data type concept in SLAWscript:

- a double that captures the numeric value of the constant (if the constant is a number),
- a string that captures the string value of the constant (if the constant is a string),
- a boolean to indicate whether or not the constant is a string.

The get_as_number() method is provided in order to get the constant numeric value, if possible.
The get_as_string() method returns the constant as a string. In the event that the constant is, in fact, a number, this method will cause the number to be converted to a string.

The methods is_a_numeric_string() and is_a_string() are provided to ascertain the true nature of the constant and can be used to prevent the automatic type conversion described above (if required). A third method, is_usable_as_a_number(), can be used to determine if the datum is of a numeric type: either a bona-fide number or a numeric string.

SLAWmisc: This class is a static construct that provides important string processing which is required in order to correctly handle string literals. Its single method, "StringLiteralParser(String)", returns a back-end-friendly string, in the process accounting for such string nuances as backslashes with formatting characters (e.g. "\n").

Validator: The Validator class contains methods to check the usability of SLAWscript variable names, subroutine names, and numbers. For a given candidate variable or subroutine name, the class checks for invalid conditions (such as a null Java String reference) as well as for possible conflicts with the set consisting of both SLAWscript reserved words and the names of already-defined subroutines. For numbers, the class audits an IEEE 754 double-precision datum to ensure that it is a valid number (i.e. not a NaN and not an infinity).

Variable: The Variable class is virtually identical to the Constant class, with two important additions: the methods “set_to(double)” and “set_to(String)”, which allow the class to model a flexible data type (like the Constant class) which, unlike objects of the Constant class, are able to change at run-time. Note: this class was not derived from Constant, nor Constant from this, due to the limited precision of the Java protection scheme.

VariableStack: The VariableStack class models the symbol table in SLAWscript. This powerful class provides provisions for multiple contexts, which is a needed ability in order to support SLAWscript's dynamic scoping of variables. This class is used throughout the processing of the SLAWscript file.. The new_context() method creates a new context for the environment, and is used for both subroutines with parameters and for the “localize” keyword. The previous_context() method rolls back the variable stack to its previous context following the completion of a subroutine. The current_context_number() method returns the current context number, so that the context may be rolled back to the proper level later on. .The method put(String, Variable) places a variable (connected to its supplied name) onto the variable stack. This method also performs error checking and name-collision detection. The reserve(String) method reserves a place for a variable in the current context, but does not create a Variable object for this place; this method is needed for the “localize” keyword. The get(String name) method is used to retrieve a variable.

Section 6: Test Plan

6.1 Representative Programs

6.1.1 Hello World

```
put "Hello World.\n" to stdout
```

This program is the canonical first source program for most programming languages. In SLAWscript, sending text output to the screen (i.e. the default output device) is very simple. Using the "put" keyword, this simple program is accomplished in a single line of source code.

6.1.2 Test of Logical OR

```
if false or false
  put "'or' fails.\n" to stdout
else
  put "'or' works.\n" to stdout
end if

if false or true
  put "'or' works.\n" to stdout
else
  put "'or' fails.\n" to stdout
end if

if true or false
  put "'or' works.\n" to stdout
else
  put "'or' fails.\n" to stdout
end if

if true or true
  put "'or' works.\n" to stdout
else
  put "'or' fails.\n" to stdout
end if
```

This program is a typical test program to verify the logical operators built into SLAWscript. In this case, we're testing the logical "or" operation. Assuming the operation has been implemented properly along with the necessary reserved words for performing the conditional test and printing to the screen, then a series of positive statements will print to the screen. This will illustrate immediately whether or not there is a bug in the language implementation.

6.1.3 GCD

```
do test_GCD[3,5,1]
do test_GCD[5,3,1]

do test_GCD[4,8,4]
do test_GCD[8,4,4]

do test_GCD[6,9,3]
do test_GCD[9,6,3]

define procedure test_GCD[a,b,expected]
  put "The expected value for the GCD of "+a+" and "+b+" is "+expected to stdout
  put "; the result for the GCD of "+a+" and "+b+" is "+GCD[a,b]+".\n" to stdout
end procedure

define function GCD[a,b]
  if a=b
    return a
  else if a>b
    return GCD[a-b, b]
  else
    return GCD[a, b-a]
  end if
end function
```

This program contains an implementation of the GCD (Greatest Common Divisor) algorithm. At the bottom, the recursive GCD function is defined which takes two arguments and returns a number. Immediately above the function definition is a test procedure to easily invoke the GCD function and print out its value along with an expected value. At the top is a series of procedure calls to invoke the testing and output to the screen.

6.2 Test Methods

6.2.1 Unit Testing

For unit testing of the SLAWscript language, we created and ran a collection of SLAWscript files. Each file was created specifically to test a small set of functionality. In some cases, we needed to test our error-handling, so we devised test programs to specifically trigger an error (e.g. divide by zero). These unit test programs were typically run immediately after the implementation of the specific language functionality. In many cases, these test programs were also run in response to a change in implementation or addition of functionality that would potentially affect the item that is being tested.

What follows is a listing of those unit test programs.

GCD.SLAW
HelloWorld.SLAW
OneOfEverything.SLAW
chaining.SLAW
empty_function.SLAW
empty_procedure.SLAW
flexible.SLAW
number_guessing_game.SLAW
numbers.SLAW
power_NaN_test.SLAW
regression.SLAW
subroutines.SLAW
substring.SLAW
test.SLAW
test_absolute_value.SLAW
test_addition.SLAW
test_and.SLAW
test_assert.SLAW
test_constants.SLAW
test_copy.SLAW
test_division.SLAW
test_division_by_zero.SLAW
test_empty_string_output.SLAW
test_exponent.SLAW

test_factorial.SLAW
test_greatThan.SLAW
test_greatThanOrEqualTo.SLAW
test_if.SLAW
test_if_and_formal_parameters_locality_and_localize_in_a_procedure.SLAW
test_instring.SLAW
test_lessThanOrEqualTo.SLAW
test_lessthan.SLAW
test_multiplication.SLAW
test_multiplication_cases.SLAW
test_negative.SLAW
test_not.SLAW
test_or.SLAW
test_postfix.SLAW
test_precedence.SLAW
test_prefix.SLAW
test_procedure_not_enough_params.SLAW
test_procedure_one_param.SLAW
test_procedure_too_many_params.SLAW
test_procedure_zero_params.SLAW
test_recursion.SLAW
test_relaxed_equality.SLAW
test_relaxed_inequality.SLAW
test_repeat_negstring_times.SLAW
test_repeat_times.SLAW
test_repeat_with.SLAW
test_stop.SLAW
test_strict_equality.SLAW
test_strict_inequality.SLAW
test_string_length.SLAW
test_substring_postfix.SLAW
test_subtraction.SLAW
test_variableContentType.SLAW
test_variableValidity.SLAW
test_while.SLAW

Our testing process consisted of two main activities: unit testing and integrated testing. Unit testing was provided by a series of SLAWscript files, each designed to test a specific aspect of the SLAWscript interpreter. These files were individually run with the interpreter, and the results were analyzed for the expected output. Usually, we executed such tests either immediately following, or during, development of specific functionality in the interpreter, e.g. after developing the “repeatWhile” functionality. However, we did execute individual tests many times during the course of development as new pieces of code were added to the project.

Integrated testing involved a series of SLAWscript programs that thoroughly tested all aspects of the SLAWscript language. These were designed to run in batch mode, producing output that we could analyze for correctness. This test output provided valuable feedback to the front-end and back-end teams about particular language features that needed improvement.

Please refer to the Testing Section of the document for a complete discussion of the testing process.

6.2.2 Integrated Testing

Integrated testing consisted of a shell script that traversed through the entire listing of SLAWscript files within the directory of unit test programs and invoked each one-by-one. The output from each program was printed to the screen in such a way as to easily distinguish between expected and unexpected output. Since most of these programs output the word 'fail' if there was a problem with the expected output, this could be queried by redirecting the output from 'stdout' to the 'grep' command. In the case of an error from unexpected output, only the failure items were printed to the screen along with a small description of the source of the error.