Columbia University
Programming Languages and Translators
Spring 2007
Professor Edwards

SLAWscript Final Report

Steve Henderson
Levi Lister
Abe Skolnik
Wei Teng

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 Wei’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 main
define procedure main
  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
end procedure
```

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 main procedure with `do main`, while 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 invoked 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

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
                     | main body 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 *)

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

main body paragraph = main body 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;

main body if paragraph =
      optional spacing, “if”, required spacing, expression, optional spacing, [comment], newline,
            { main body if paragraph
            | repeat paragraph
            | while paragraph
            | main body sentence
            | effectively empty line
            },
      { optional spacing, “else”, required spacing, “if”, required spacing, expression,
        optional spacing, [comment], newline,
            { main body if paragraph
            | repeat paragraph
            | while paragraph
            | main body sentence
            | effectively empty line
            }
      },
      [ optional spacing, “else”, optional spacing, [comment], newline,
            { main body if paragraph
            | repeat paragraph
            | while paragraph
            | main body 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,
       { main body if paragraph
       | repeat paragraph
       | while paragraph
       | main body 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,
       { main body if paragraph
       | repeat paragraph
       | while paragraph
       | main body 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
      | main body 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 Project Time-line

The Gantt Charts on this page and the following pages detail our project time-line.

TASK	DATE (WEEK) 1/5	2/5	2/12	2/19	2/26	3/5	3/12	3/19	3/26	4/2	4/9	4/16	4/23	4/30	5/7
FRONTEND															
Initial Prototyping with ANTLR/ANTLR Works															
Establish Development Environment															
Initial Front End Syntax and Tokens															
Develop Rules for Group 1 (Architecture)															
assertSentence															
Constant															
Identifier															
InstrExpr															
NormalIfParagraph															
NormalParagraphOrFunctionValidSentence															
NormalParagraphOrNormalSentence															
NormalParagraphOrSubroutineValidSentence															
UsableInExpressions															
Variable															
Develop Rules for Group 2 (Basic Functionality)															
andExpr															
copySentence															
DivExpr															
doSentence															
Factorial															
FactorialExpr															
getSentence															
GreaterThanExpr															
GreaterThanOrEqualExpr															
ignoreSentence															
LessThanExpr															
LessThanOrEqualExpr															
MinusExpr															
MulExpr															
notExpr															
orExpr															
ParserReturnType															
PipeExpr															
PlusExpr															
PowerExpr															
putSentence															
RelaxedDoesNotEqualExpr															
RelaxedEqualsExpr															
setSentence															
stopSentence															
StrictlyDoesNotEqualExpr															
StrictlyEqualsExpr															
SubstrExpr															
DoubleQuestionMarkExpr															
Develop Rules for Group 3 (IF and Loops)															
FunctionIfParagraph															
ProcedureIfParagraph															
randomizeSentence															
repeatParagraph															
repeatTimesParagraph															
repeatWithParagraph															
RoundExpr															
whileParagraph															
Develop Rules for Group 4 (Procedures)															
localizeSentence															
Procedure															
SLAWmisc															
Develop Rules for Group 5 (Functions)															
assertSentence															
Function															
FunctionCallWithParams															
returnSentence															
SingleQuestionMarkExpr															
Respond to Testing Results															

TASK	DATE (WEEK)														
	1/5	2/5	2/12	2/19	2/26	3/5	3/12	3/19	3/26	4/2	4/9	4/16	4/23	4/30	5/7
BACKEND															
Establish Development Environment															
Develop Group 1 (Architecture)															
Constant															
Identifier															
InstrExpr															
NormalIfParagraph															
NormalParagraphOrFunctionValidSentence															
NormalParagraphOrNormalSentence															
NormalParagraphOrSubroutineValidSentence															
UsableInExpressions															
Validator															
Variable															
VariableStack															
Develop Group 2 (Basic Functionality)															
andExpr															
copySentence															
DivExpr															
doSentence															
Factorial															
FactorialExpr															
getSentence															
GreaterThanExpr															
GreaterThanOrEqualExpr															
ignoreSentence															
LessThanExpr															
LessThanOrEqualExpr															
MinusExpr															
MulExpr															
notExpr															
orExpr															
ParserReturnType															
PipeExpr															
PlusExpr															
PowerExpr															
putSentence															
RelaxedDoesNotEqualExpr															
RelaxedEqualsExpr															
setSentence															
stopSentence															
StrictlyDoesNotEqualExpr															
StrictlyEqualsExpr															
SubstrExpr															
DoubleQuestionMarkExpr															
"Hello World" Example															
Develop Group 3 (IF and Loops)															
FunctionIfParagraph															
ProcedureIfParagraph															
randomizeSentence															
repeatParagraph															
repeatTimesParagraph															
repeatWithParagraph															
RoundExpr															
whileParagraph															
Develop Group 4 (Procedures)															
localizeSentence															
Procedure															
SLAWmisc															
Develop Group 5 (Functions)															
assertSentence															
Function															
FunctionCallWithParams															
returnSentence															
SingleQuestionMarkExpr															
Respond to Testing Results															

TASK	DATE (WEEK)														
	1/5	2/5	2/12	2/19	2/26	3/5	3/12	3/19	3/26	4/2	4/9	4/16	4/23	4/30	5/7
TESTING															
Front End Testing (in ANTLR Works)															
Hello World Test															
Language Construct Testing															
comment test															
power_NaN_test test															
absolute_value test															
addition test															
and test															
constants test															
copy test															
division test															
division_by_zero test															
exponent test															
factorial test															
greatThan test															
greatThanOrEqualTo test															
lessThanOrEqualTo test															
lessthan test															
multiplication test															
multiplication_cases test															
negative test															
not test															
or test															
precedence test															
prefix test															
stop test															
ring test															
relaxed_equality test															
relaxed_inequality test															
repeat_negstring_times test															
strict_equality test															
strict_inequality test															
string_length test															
substring_position test															
substring_postfix test															
subtraction test															
undefined test															
variableContentType test															
variableValidity test															
if test															
repeat_times test															
repeat_with test															
while test															
empty_procedure test															
procedure_not_enough_params test															
procedure_one_param test															
procedure_too_many_params test															
procedure_zero_params test															
empty_function test															
assert test															
Integration Testing															
DOCUMENTATION															
Proposal															
Language Reference Manual															
Final Documentation															
Project Report															

4.5 Roles and Responsibilities

4.5.1 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. The team met regularly every Friday from 3:30pm-5:30pm in CEPSR Room 620, with a few exceptions (e.g. Spring Break). Due to two team members being ill at the time, one meeting was executed remotely using Skype (this turned out to be one of our most productive meetings). Our team used these meeting to check progress, identify priorities, assign responsibility, and coordinate development. Toward the final stages of the project, our team met multiple times a week.

4.5.2 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 and Wei Teng.

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, Wei, and Steve supporting
    - Reference Manual: Abe (lead); Levi, Wei, and Steve supporting
    - Code Documentation: Steve (lead), Abe supporting
    - Final Project Report: All; Abe (editor and publisher)

4.5.3 External Interfaces

Our team's primary external interface is with the course instructor, Dr. Stephen Edwards. The course instructor serves as our project mentor and source of guidance.

4.6 Software Development Environment

4.6.1 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.

4.6.2 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.

4.6.3 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.

4.7 Project Log

The following list significant events recorded on our team's collaboration website. This is not a comprehensive list of team communication or efforts.

April 2007

- 22 April Abe wrote the shell script "comment_dump"
- 22 April: Abe reports an obscure error in input handling code
- 21 April: Abe reports '@' operator working
- 20 April: Levi completes test_division_by_zero.SLAW
- 13 April: Levi post weekly meeting notes
- 13 April: Weekly team meeting
- 13 April: Steve updates JavaDoc (web version)
- 13 April: Steve reports procedures working
- 12 April: Abe provides jEdit config file for SLAWscript
- 7 April: Abe reports error handling working
- 7 April: Steve and Abe complete and test whileParagraph, repeatTimes, repeatWith
- 6 April: Steve posts weekly meeting notes
- 6 April: Weekly team meeting (via Skype)
- 5 April: Steve updates JavaDoc with links to on-line LRM
- 4 April: Steve completes initial repeatTimes
- 4 April: Abe adds color output to SLAWscript

March 2007

- 31 March: Abe brings up HelloWorld (using non-AST build)
- 31 March: Abe devises new build system in repository
- 20 March: Weekly team meeting
- 29 March: Steve post first JavaDoc to web
- 23 March: Weekly team meeting
- 16 March: Weekly team meeting
- 9 March: Team begins major implementation design overhaul – not using AST and TreeWalker
- 2 March: Weekly meeting

February 2007

- 27 February: Abe finished PlusExpr
- 26 February: Levi reviews draft LRM – comments to Abe
- 25 February: Abe revises ".g" code for "localize" and "return"
- 23 February: Weekly meeting
- 23 February: Steve and Abe get first Hello World working (Tree Walker)
- 22 February: Steve codes initial back-end (with Tree Walker)
- 21 February: Steve generates the first AST (with Tree Walker)
- 9 February: Levi stands up OpenSvn repository
- 9 February: Weekly meeting
- 4 February: Levi stands up web-based collaboration site
- 2 February: First weekly meeting

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.

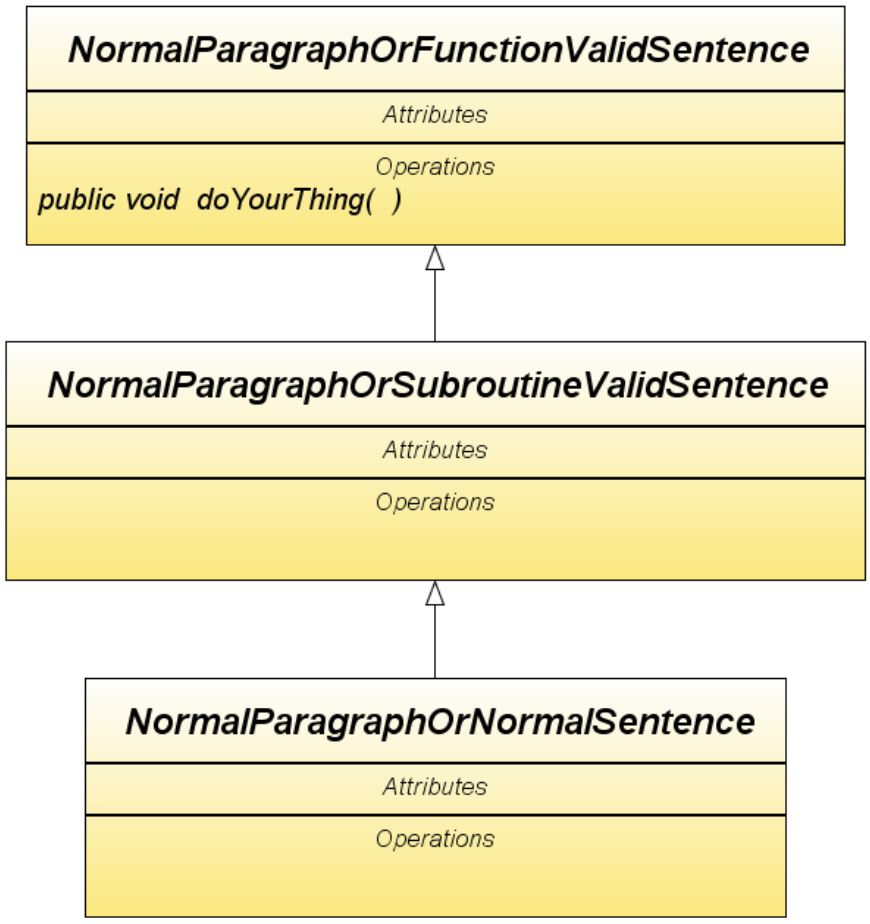

Figure 1: Top Level Abstract Classes and Interfaces

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.

6.3 Roles and Responsibilities

The testing activity was mainly conducted by Levi Lister and Wei Teng. In many cases, there was direct interaction with front-end and back-end sub-teams. In these interactions, the front-end and back-end teams would make requests for test programs to validate specific functionality. Many attempts were made to work in parallel by having the test sub-team develop programs in anticipation of certain functionality.

However, there were also some occasions when new test programs were created on-the-fly in response to unanticipated issues with functionality. The front-end and back-end teams also developed a few test programs on those occasions when it was faster or simpler to implement a quick test to verify functionality as opposed to waiting for time to allow someone else to code up a small test program.

During our weekly meetings, we co-ordinated closely to allow the exchange of feedback about problems that were uncovered during testing as well as problems with the test programs and scripts themselves.

Section 7: Lessons Learned

7.1 Steve's Conclusions

My biggest recommendations about things to sustain are related to specification and tools for collaboration. With regard to specification, our team took the time to write a 95% complete and detailed Language Reference Manual (LRM) when it was first due (early February). This was crucial to our success in countless ways. First, given the collaborative nature of our design and development process, the LRM served as an important and constant beacon that unified our efforts. Second, strictly adhering to the LRM helped prevent scope creep in our project. Third, the LRM is a very handy reference when it comes time to implement the underlying code for the design. Integrating the LRM with the source code JavaDoc was extremely helpful.

With regards to tools for collaboration, an effective file sharing mechanism is essential for this kind of project. A file-management repository (based on e.g. Subversion) that contains carefully organized and structured files is a must. Future teams should invest the time to set up and use Subversion at the very early stages of the project, and use it for everything (even the most mundane of files). Another take-away related to collaboration is having each member working on the same developmental platform. Most of us used a UNIX-like system (e.g. GNU/Linux, Mac OS X). This helped with the standardization of things like end-of-line characters. It also helped to keep our Subversion repository clean and free of extraneous files.

On the improvements side, I wish each of us could have been more involved in every design activity: front-end development, back-end development, and testing. Because this project is such a large one, we were forced to compartmentalize our efforts to meet the deadline. I think this is suboptimal from a learning standpoint. Choosing a smaller and less-robust language may help reduce the amount of work such that each team member can take more time to participate in all of the design activities. I think compartmentalizing is still a must (from a project management standpoint), but future teams should try to provide time and opportunity for each member to carefully study, understand, and learn what is happening within each activity.

7.2 Levi's Conclusions

Besides the obvious lesson of starting as early as possible to make efficient use of the time allocated for the project, I think taking advantage of organizational tools and services proved to be extremely useful for us. As mentioned in our report, we used a few tools to achieve a sense of organization, but it still felt like we could have been a little more organized (as usual with any group project).

Early in the semester we set up and started using Subversion to maintain a synchronized repository for our all of the project-related files. The OpenSvn website provides free Subversion hosting along with “http” and “https” access to all the files. We also set up and used a project collaboration website, using a service called “BaseCamp”. This site allowed us to maintain shared “writeboards”, upload files, post messages, maintain to-do lists, and keep track of deadlines. Although we did not fully exhaust the site's features, it provided a centralized place for important things. Had we not used these two tools/services, we could have easily lost our sanity by relying entirely on [unorganized] emails and time-wasting code synchronization and integration.

7.3 Abe's Conclusions

Towards the end of the semester, it became clear to me that our decision to operate in parallel, rather than sequentially, had been essential to our completing the project on time. Using the strategy that we did, the parser was not perfected until almost the end of the semester, but having a partially-working parser enabled us to check our back-end work as it progressed, while work on the parser proceeded in parallel. I think that if we had worked in the old-fashioned "waterfall" method instead, the same project would probably have taken a year to complete (given the same manpower) rather than four or five months.

Suggestion to future teams: start early! I started four weeks early, had to "scratch" my first three weeks or so worth of work, and my team (including myself) still needed each and every week, even with scaled-down objectives. You will not have more time than you need; if anything, you will not have enough time to meet your original objectives. Try to set realistic goals for your project and start as early as you possibly can.

Also, don't rule out dropping planned features; quite to the contrary, I suggest that you design your project so that you can drop some parts of it later (if and when you figure out that you won't have time for everything) and still have a pretty good project left.

7.4 Wei's Conclusions

Among the several things that I learned while working on this project, I feel the most important experience is the advantage of working on a well-defined schedule to achieve goals consecutively and meet deadlines on time, and this is true especially when the time and the resources are limited. During the initial state we had only a general idea about what the project would be like, therefore, we made a great effort to discuss the scope and the concept of the project as well as the goals we wanted to achieve. Once we reached a common agreement and a clear definition, it was important to stick with the schedule and meet every single deadline. Besides the required deliverables deadlines, we also set a variety of desired dates for separate tasks to finish. This type of collaboration greatly maintained and increased our progress without getting us distracted by later new ideas that were trivial or unnecessary but might have held back the whole project.

Also, by working on this project, I gained practical experience on how to write efficient testing programs to give feedback to the development process. For example, I learned how to write testing code for a certain operator or keyword, as well as the integrated testing code that is helpful to find any potential bugs for a specific language.

Lastly, I learned the underlying knowledge of a working programming language, in terms of its design structure and how it processes. Together with lessons I learned in class, this project reinforced in me a solid knowledge of the theory of a modern programming language translator.

Section 8: Appendix

8.1 ANTLR (v3) Code

```
// this file was written by Abe

grammar SLAWscript;

options {
  k=2;
  // output=AST;
  // ASTLabelType=CommonTree;
}

@header {
  import java.util.Vector;
  import java.util.Hashtable;
}

@members {
  Vector<NormalParagraphOrNormalSentence> mainBody   =
    new Vector<NormalParagraphOrNormalSentence>();
  Hashtable<String,Function>              functions  = new Hashtable<String,Function>();
  Hashtable<String,Procedure>             procedures = new Hashtable<String,Procedure>();
}

@rulecatch {
  catch (RecognitionException re) {
    System.err.println("A syntax error was found on line "+re.line);
    System.err.println("Aborting interpreter.");
    System.exit(-1);
  }

  catch (Throwable t) {
    System.err.print("An error occurred in the parser: ");
    t.printStackTrace();
    System.err.println("Aborting interpreter.");
    System.exit(-1);
  }
}


startRule returns [ParserReturnType prt]:
          ( verbSentence      { mainBody.add($verbSentence.sentence); } (EOF|EOL)
          | normalIfParagraph { mainBody.add($normalIfParagraph.nip); }
          | whileParagraph    { mainBody.add($whileParagraph.wp); }
          | repeatParagraph   { mainBody.add($repeatParagraph.rp); }
          | Comment EOF // "Comment EOF" is intentional
          | defineParagraph
          | EOL
          )*
          { $prt = new ParserReturnType(mainBody,functions,procedures); }
          ;

verbSentence returns [NormalParagraphOrNormalSentence sentence]:
             ( assertSentence    { $sentence = $assertSentence.as; }
             | copySentence      { $sentence = $copySentence.cs; }
             | doSentence        { $sentence = $doSentence.ds; }
             | ignoreSentence    { $sentence = $ignoreSentence.is; }
             | setSentence       { $sentence = $setSentence.set; }
             | getSentence       { $sentence = $getSentence.gs; }
             | putSentence       { $sentence = $putSentence.ps; }
             | randomizeSentence { $sentence = $randomizeSentence.rs; }
             | stopSentence      { $sentence = $stopSentence.stop_ret; }
             );
// reminder: "localizeSentence" and "returnSentence" must _not_ be included here
```

```
normalParagraphOrNormalSentence returns [NormalParagraphOrNormalSentence npns]:
                                ( (verbSentence EOL) { $npns = $verbSentence.sentence; }
                                | normalIfParagraph  { $npns = $normalIfParagraph.nip; }
                                | whileParagraph     { $npns = $whileParagraph.wp; }
                                | repeatParagraph    { $npns = $repeatParagraph.rp; }
                                );


// this is only valid within a function definition
returnSentence returns [returnSentence rs]: 'return' expr EOL { $rs =
  new returnSentence($expr.uie); };


// note: "define" paragraphs are unique in that they may _not_ be inside other paragraphs

defineParagraph: 'define' (defineFunction | defineProcedure);

// the word 'define' is intentionally not repeated in the following two rules; keeping it
// in the 'defineParagraph' rule lets the spacing after the word 'define' be flexible

defineFunction
  @init {
    Vector<NormalParagraphOrFunctionValidSentence> code =
      new Vector<NormalParagraphOrFunctionValidSentence>();
    Vector<String> args = new Vector<String>();
  }
  @finally {
    functions.put( $name.text.toLowerCase(), new Function(code, args, $name.text) );
  }
  :
  'function' name=Identifier ('[' ( a=Identifier ',' { args.add($a.text); } )*
                                  b=Identifier { args.add($b.text); } ']')? EOL
    ( functionSentenceOrParagraph { code.add($functionSentenceOrParagraph.npfvs); }
    | EOL
    )*
  End_function (EOF|EOL);

functionSentenceOrParagraph returns [NormalParagraphOrFunctionValidSentence npfvs] :
                            ( (verbSentence EOL)  { $npfvs = $verbSentence.sentence; }
                            | functionIfParagraph { $npfvs = $functionIfParagraph.fip; }
                            | whileParagraph      { $npfvs = $whileParagraph.wp; }
                            | repeatParagraph     { $npfvs = $repeatParagraph.rp; }
                            | localizeSentence    { $npfvs = $localizeSentence.ls; }
                            | returnSentence      { $npfvs = $returnSentence.rs; }
                            );

defineProcedure
  @init {
    Vector<NormalParagraphOrSubroutineValidSentence> code =
      new Vector<NormalParagraphOrSubroutineValidSentence>();
    Vector<String> args = new Vector<String>();
  }
  @finally {
    procedures.put( $name.text.toLowerCase(), new Procedure(code, args, $name.text) );
  }
  :
  'procedure' name=Identifier ('[' ( a=Identifier ',' { args.add($a.text); } )*
                                   b=Identifier { args.add($b.text); } ']')? EOL
    ( procedureSentenceOrParagraph  { code.add($procedureSentenceOrParagraph.npsvs); }
    | EOL
    )*
  End_procedure (EOF|EOL);
```

```
procedureSentenceOrParagraph returns [NormalParagraphOrSubroutineValidSentence npsvs] :
                              ( (verbSentence EOL)   { $npsvs = $verbSentence.sentence; }
                              | procedureIfParagraph { $npsvs = $procedureIfParagraph.pip; }
                              | whileParagraph       { $npsvs = $whileParagraph.wp; }
                              | repeatParagraph      { $npsvs = $repeatParagraph.rp; }
                              | localizeSentence     { $npsvs = $localizeSentence.ls; }
                              );


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); } ;


normalIfParagraph returns [NormalIfParagraph nip]
  @init {
    Vector<UsableInExpressions>                          conditions =
      new Vector<UsableInExpressions>(); // this must have at least one element

    Vector<NormalParagraphOrNormalSentence>              ifCode     =
      new Vector<NormalParagraphOrNormalSentence>();

    Vector< Vector<NormalParagraphOrNormalSentence> > elseIfCode =
      new Vector< Vector<NormalParagraphOrNormalSentence> >();

    Vector<NormalParagraphOrNormalSentence>              elseCode   =
      new Vector<NormalParagraphOrNormalSentence>();
  }
  @finally {
    $nip = new NormalIfParagraph(conditions, ifCode, elseIfCode, elseCode);
  }
  :
  'if' ifex=expr EOL { conditions.add($ifex.uie); }
     (ifcode=normalParagraphOrNormalSentence { ifCode.add($ifcode.npns); } | EOL)*
   ( 'else' 'if' elifex=expr EOL
     { conditions.add($elifex.uie);
       Vector<NormalParagraphOrNormalSentence> temp =
         new Vector<NormalParagraphOrNormalSentence>(); }
     (elifcode=normalParagraphOrNormalSentence { temp.add($elifcode.npns); } | EOL)*
     { elseIfCode.add(temp); }
   )*
   ( 'else' EOL
     (elsecode=normalParagraphOrNormalSentence { elseCode.add($elsecode.npns); } | EOL)*
   )?
    End_if (EOF|EOL);

functionIfParagraph returns [FunctionIfParagraph fip]
  @init {
    Vector<UsableInExpressions>                                  conditions =
      new Vector<UsableInExpressions>(); // this must have at least one element

    Vector<NormalParagraphOrFunctionValidSentence>               ifCode     =
      new Vector<NormalParagraphOrFunctionValidSentence>();

    Vector< Vector<NormalParagraphOrFunctionValidSentence> > elseIfCode =
      new Vector< Vector<NormalParagraphOrFunctionValidSentence> >();

    Vector<NormalParagraphOrFunctionValidSentence>               elseCode   =
      new Vector<NormalParagraphOrFunctionValidSentence>();
  }
```

```
  @finally {
    $fip = new FunctionIfParagraph(conditions, ifCode, elseIfCode, elseCode);
  }
  :
  'if' ifex=expr EOL { conditions.add($ifex.uie); }
     (ifcode=functionSentenceOrParagraph { ifCode.add($ifcode.npfvs); } | EOL)*
   ( 'else' 'if' elifex=expr EOL
     { conditions.add($elifex.uie);
       Vector<NormalParagraphOrFunctionValidSentence> temp =
         new Vector<NormalParagraphOrFunctionValidSentence>(); }
     (elifcode=functionSentenceOrParagraph { temp.add($elifcode.npfvs); } | EOL)*
     { elseIfCode.add(temp); }
   )*
   ( 'else' EOL
     (elsecode=functionSentenceOrParagraph { elseCode.add($elsecode.npfvs); } | EOL)*
   )?
    End_if (EOF|EOL);

procedureIfParagraph returns [ProcedureIfParagraph pip]
  @init {
    Vector<UsableInExpressions>                                  conditions =
      new Vector<UsableInExpressions>(); // this must have at least one element

    Vector<NormalParagraphOrSubroutineValidSentence>             ifCode     =
      new Vector<NormalParagraphOrSubroutineValidSentence>();

    Vector< Vector<NormalParagraphOrSubroutineValidSentence> > elseIfCode =
      new Vector< Vector<NormalParagraphOrSubroutineValidSentence> >();

    Vector<NormalParagraphOrSubroutineValidSentence>             elseCode   =
      new Vector<NormalParagraphOrSubroutineValidSentence>();
  }
  @finally {
    $pip = new ProcedureIfParagraph(conditions, ifCode, elseIfCode, elseCode);
  }
  :
  'if' ifex=expr EOL { conditions.add($ifex.uie); }
     (ifcode=procedureSentenceOrParagraph { ifCode.add($ifcode.npsvs); } | EOL)*
   ( 'else' 'if' elifex=expr EOL
     { conditions.add($elifex.uie);
       Vector<NormalParagraphOrSubroutineValidSentence> temp =
         new Vector<NormalParagraphOrSubroutineValidSentence>(); }
     (elifcode=procedureSentenceOrParagraph { temp.add($elifcode.npsvs); } | EOL)*
     { elseIfCode.add(temp); }
   )*
   ( 'else' EOL
     (elsecode=procedureSentenceOrParagraph { elseCode.add($elsecode.npsvs); } | EOL)*
   )?
    End_if (EOF|EOL);


repeatParagraph returns [repeatParagraph rp]
  @init {
    boolean is_with=false; // otherwise, is "times"
    Vector<NormalParagraphOrNormalSentence> code =
      new Vector<NormalParagraphOrNormalSentence>();
  }
  @finally {
    if (is_with) {
      $rp = new repeatWithParagraph(code,$from.uie,$to.uie,$step.uie,$withID.text);
      // "step" shall be null if the step was not specified
    } else { // is "times"
      $rp = new repeatTimesParagraph(code,$times.uie);
    }
  }
  :
```

```
  'repeat' ('with' withID=Identifier 'from' from=expr 'to' to=expr ('step' step=expr)?
             { is_with=true; }
           | times=expr 'times'
           ) EOL
             (normalParagraphOrNormalSentence
               { code.add($normalParagraphOrNormalSentence.npns); } | EOL)*
           End_repeat (EOF|EOL);


assertSentence returns [assertSentence as]:
               'assert' Identifier 'is' ( constant      { $as =
                  new assertSentence($Identifier.text, $constant.c); }

                                        | number        { $as =
                  new assertSentence($Identifier.text, new Constant($number.n)); }

                                        | stringLiteral { $as =
                  new assertSentence($Identifier.text, $stringLiteral.s); }
                                        );

copySentence returns [copySentence cs]:
              'copy' from=Identifier 'to' to=Identifier { $cs =
               new copySentence($from.text,$to.text); };

setSentence returns [setSentence set]:
             'set' Identifier 'to' ex=expr { $set =
              new setSentence($Identifier.text,$ex.uie); };

getSentence returns [getSentence gs]: 'get' Identifier { $gs =
              new getSentence($Identifier.text); };

putSentence returns [putSentence ps]:
             'put' ex=expr 'to' ( 'stderr' { $ps = new putSentence($ex.uie,false); }
                                | 'stdout' { $ps = new putSentence($ex.uie,true); }
                                );

randomizeSentence returns [randomizeSentence rs]:
                'randomize' Identifier { $rs = new randomizeSentence($Identifier.text); };

doSentence returns [doSentence ds]
  @init    { Vector<UsableInExpressions> params = new Vector<UsableInExpressions>(); }
  @finally { $ds = new doSentence(params,$name.text); }
  :
  'do' name=Identifier
  ('[' ap1=expr { params.add($ap1.uie); } (',' apn=expr { params.add($apn.uie); } )* ']')?
  ;

localizeSentence returns [localizeSentence ls]:
                 'localize' Identifier EOL { $ls =
                  new localizeSentence($Identifier.text); };

ignoreSentence returns [ignoreSentence is]:
               'ignore' fn=Identifier ( '[' { Vector<UsableInExpressions> params =
                  new Vector<UsableInExpressions>(); }
                  ap1=expr { params.add($ap1.uie); }
                  ( ',' apn=expr { params.add($apn.uie); } )*
                  ']' { $is = new ignoreSentence(
                                new FunctionCallWithParams(params,$fn.text) ); }
                      | /* intentionally empty rule part */ { $is = new ignoreSentence(
                                                               new Identifier($fn.text) ); }
                      );

stopSentence returns [stopSentence stop_ret]: st='stop' { $stop_ret =
                                                    new stopSentence($st.line); };
```

```
expr returns [UsableInExpressions uie]: l=relExpr ( 'and' r=expr { $uie =
  new andExpr($l.uie,$r.uie); }
                                                  | 'or'  r=expr { $uie =
  new  orExpr($l.uie,$r.uie); }
                                                  |              { $uie = $l.uie; }
  // intentionally-blank rule part
                                                  );

relExpr returns [UsableInExpressions uie]:
       l=addExpr ( '<'    r=addExpr { $uie = new LessThanExpr($l.uie,$r.uie); }
                 | '>'    r=addExpr { $uie = new GreaterThanExpr($l.uie,$r.uie); }
                 | '<='   r=addExpr { $uie = new LessThanOrEqualExpr($l.uie,$r.uie); }
                 | '>='   r=addExpr { $uie = new GreaterThanOrEqualExpr($l.uie,$r.uie); }
                 | '='    r=addExpr { $uie = new RelaxedEqualsExpr($l.uie,$r.uie); }
                 | '=='   r=addExpr { $uie = new StrictlyEqualsExpr($l.uie,$r.uie); }
                 | '<>'   r=addExpr { $uie = new RelaxedDoesNotEqualExpr($l.uie,$r.uie); }
                 | '<<>>' r=addExpr { $uie =
                                       new StrictlyDoesNotEqualExpr($l.uie,$r.uie); }
                 |                  { $uie = $l.uie; } // intentionally blank rule part
                );
// NOTE: not allowing chained comparisons w/o paren.s, e.g. "a<b<c", "a=b=c", "a>b>c"

// Examples of why '-' MUST NOT have lower precedence than '+':
//   a+b-c+d            : '+' higher: (a+b)-(c+d) === a+b-c-d   WRONG
//                      : '-' higher: a+(b-c)+d   === a+b-c+d   RIGHT
//   e-f+g-h            : '+' higher: e-(f+g)-h   === e-f-g-h   WRONG
//                      : '-' higher: (e-f)+(g-h) === e-f+g-h   RIGHT

addExpr returns [UsableInExpressions uie]
  @init { Vector<UsableInExpressions> args = new Vector<UsableInExpressions>(); }
  :
  l=subtractExpr { args.add($l.uie); } ( ( '+' r=subtractExpr { args.add($r.uie); } )+
{ $uie = new PlusExpr(args); }
                                     |                      { $uie = $l.uie; }
  // intentionally blank rule part
                                     );

subtractExpr returns [UsableInExpressions uie]
  @init { Vector<UsableInExpressions> args = new Vector<UsableInExpressions>(); }
  :
  l=mulExpr { args.add($l.uie); } ( ( '-' r=mulExpr { args.add($r.uie); } )+ { $uie =
    new MinusExpr(args); }
                                  |                 { $uie = $l.uie; }
  // intentionally blank rule part
                                  );


mulExpr returns [UsableInExpressions uie]
  @init { Vector<UsableInExpressions> args = new Vector<UsableInExpressions>(); }
  :
  l=divExpr { args.add($l.uie); } ( ( '*' r=divExpr { args.add($r.uie); } )+ { $uie =
    new MulExpr(args); }
                                  |                 { $uie = $l.uie; }
  // intentionally blank rule part
                                  );

divExpr returns [UsableInExpressions uie]
  @init { Vector<UsableInExpressions> args = new Vector<UsableInExpressions>(); }
  :
  l=powerExpr { args.add($l.uie); } ( ( '/' r=powerExpr { args.add($r.uie); } )+ { $uie =
    new DivExpr(args); }
                                    |                   { $uie = $l.uie; } //
intentionally blank rule part
                                    );
// this has higher precedence than '*' so that e.g. 3/8*5
// will mean (3/8)*5, not 3/(8*5) [good for fractions]
```

```
powerExpr returns [UsableInExpressions uie]:
       l=substrExpr ( '^' r=powerExpr { $uie = new PowerExpr($l.uie,$r.uie); }
                    |                 { $uie = $l.uie; } // intentionally blank rule part
                    ); // reminder: _intentionally_ not using the Vector technique for '^'

substrExpr returns [UsableInExpressions uie]: // NOTE: no chaining w/o paren.s
           l=atomicExpr ( ':' r=atomicExpr                          { $uie =
  new InstrExpr($l.uie,$r.uie); }
                        | '@' r1=atomicExpr (';' r2=atomicExpr)? { $uie =
  new SubstrExpr($l.uie,$r1.uie,$r2.uie); }
                        |                                          { $uie = $l.uie; }
  // intentionally-blank rule part
                        );


atomicExpr returns [UsableInExpressions uie]:
           ( '|' ex=expr '|' ( '!' { $uie = new FactorialExpr(new PipeExpr($ex.uie)); }
                             |     { $uie = new PipeExpr($ex.uie); }
  // intentionally-empty rule part
                             )
           | '(' ex=expr ')' ( '!' { $uie = new FactorialExpr($ex.uie); }
                             |     { $uie = $ex.uie; } // intentionally-empty rule part
                             )
           | constant { $uie=$constant.c; }
           | ID=Identifier ( '??' { $uie = new DoubleQuestionMarkExpr($ID.text); }
                           | '?'  { $uie = new SingleQuestionMarkExpr($ID.text); }
                           | '!'  { $uie = new FactorialExpr( new Identifier($ID.text) );
}
                           |      { $uie = new Identifier($ID.text); }
  // intentionally-blank rule part
                           )
           | number ( '%' { $uie = new Constant($number.n * 0.01); }
                    | '!' { $uie = new Constant(Factorial.factorial($number.n)); }
                    |     { $uie = new Constant($number.n); }
  // no suffix - just a straight number
                    )
           | stringLiteral       { $uie = $stringLiteral.s; }
           | 'not' ae=atomicExpr { $uie = new notExpr($ae.uie); }
           | '~' ae=atomicExpr   { $uie = new RoundExpr($ae.uie); }
           | '-' ae=atomicExpr   { $uie = new MinusExpr(new Constant(0.0), $ae.uie); }
           | fn=Identifier '[' { Vector<UsableInExpressions> params =
                                 new Vector<UsableInExpressions>(); }
  ap1=expr { params.add($ap1.uie); } ( ',' apn=expr { params.add($apn.uie); } )*
  ']' ( '!' { $uie = new FactorialExpr( new FunctionCallWithParams(params,$fn.text) ); }
  { $uie = new FunctionCallWithParams(params,$fn.text); } // intentionally empty rule part
                               )
           );


constant returns [Constant c]: ( 'e'                          { $c =
  new Constant(2.7182818284590451); }

                               | ('pi' | 'PI' | 'Pi' | 'pI') { $c =
  new Constant(3.1415926535897931); }

                               | 'true'                       { $c = new Constant(true); }
                               | 'false'                      { $c = new Constant(false); }
                               | 'escape'                     { $c = new Constant("\033"); }
                              );


// Hacks to get around the fact that lexer rules are not allowed to return custom types

stringLiteral returns [Constant s]: StringLiteral { $s =
  new Constant(SLAWmisc.StringLiteralParser($StringLiteral.text)); };

number returns [double n]: Number { $n = Double.parseDouble($Number.text); } ;
```

```
// ---------------------- Lexer -----------------------

Comment: ('#' (~('\n'|'\r'))*) { $channel=HIDDEN; };
  // this definition stops at the first possible new-line character

Number: ( ( ('0'..'9')* '.' ('0'..'9')+ ) // real number (optional leading zero)
        | ('0'..'9')+                     // integer
        );

RelaxedEq: '=';
StrictEq: '==';
GtEq: '>=';
LtEq: '<=';
RelaxedNotEq: '<>';
StrictNotEq: '<<>>';
Plus: '+';
Div: '/';
Minus: '-';
Mul: '*';
Factorial_symbol: '!';
Exponent: '^';
Percent: '%';
Pipe: '|';
Type: '?';
Substr: '@';
Open_Parens: '(';
Close_Parens: ')';
Colon: ':';


Identifier: ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'_'|'0'..'9')*;

EOL: ('\n' | '\r\n' | '\r') ;  //  EOL must _not_ be "channel HIDDEN"

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

// this is for string literals; a double-quote mark may be embedded by using '\"'

StringLiteral: '"' ( (~('"'|'\n'|'\t'|'\\')) | '\\n' | '\\t' | '\\\\' | '\\"' )* '"';
// The funny-looking '\\\\' is for in-string usages of '\\', to produce one "real" '\',
// like in C et al.  '\\"' is for literal quote marks in the destination language.

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

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

End_procedure: 'end' (' ' | '\t')+ 'procedure';

End_while: 'end' (' ' | '\t')+ 'while';

End_function: 'end' (' ' | '\t')+ 'function';
```

8.2 Java Code

```
// === andExpr.java === //

// this file was written by Abe


public class andExpr implements UsableInExpressions {

  private UsableInExpressions member___left, member___right;

  private andExpr() { } // disallow the default constructor

  andExpr(UsableInExpressions left, UsableInExpressions right) {
    if (null==left || null==right) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___left  = left;
    member___right = right;
  }


  public Constant evaluate() {

    // note from Abe: for "and" and "or", we do _not_ evaluate both expr.s first!
    //                By doing these as I have done, we inherit Java's short-circuiting.

    return new Constant( (member___left.evaluate().get_as_a_number()!=0.0)
                         &&
                         (member___right.evaluate().get_as_a_number()!=0.0) );

  }

} // end of class
```

```
// === assertSentence.java === //

// this file was written by Abe

public class assertSentence extends NormalParagraphOrNormalSentence {

  private String var_name;
  private Constant constant;

  assertSentence(String var_name_in, Constant constant_in ) { // the only constructor
    if (null==var_name_in || null==constant_in) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if (! Validator.identifier_is_usable_as_a_variable_name(var_name_in)) {
      System.err.println("A name ('"+var_name_in+"') that was not usable for a variable
was attempted to be used in an 'assert' sentence.\nAborting interpreter.");
        System.exit(-1);
    }
```

```
    var_name = var_name_in;
    constant = constant_in;
  }

  public void doYourThing() {
    Variable v = VariableStack.get(var_name);

    if (v.is_a_string() && constant.is_a_string()) { // if they are both strings...

      if ( ! v.get_as_a_string().equals(constant.get_as_a_string()) ) { // if they don't
contain the same string...
        System.err.println("Assertion failure: the variable '"+var_name+"' does not
contain '"+constant.get_as_a_string()+"'.\nAborting interpreter.");
        System.exit(-1);
      }

    } else if ( (!v.is_a_string()) && (!constant.is_a_string()) ) { // if they are both
numbers...

      if ( v.get_as_a_number() != constant.get_as_a_number() ) { // if they don't contain
the same number...
        System.err.println("Assertion failure: the variable '"+var_name+"' does not
contain ("+constant.get_as_a_number()+").\nAborting interpreter.");
        System.exit(-1);
      }

    } else { // the data types are different
        System.err.println("Assertion failure: the data type of the variable
'"+var_name+"' is different from the data type of the constant " +
( constant.is_a_string() ? "'"+constant.get_as_a_string()+"'" :
"("+constant.get_as_a_number()+")" )  + ".\nAborting interpreter.");
        System.exit(-1);
    }

  }

} // end of class
```

```
// === Constant.java === //

/*******************************************************
 *
 * The Constant class is used to model a constant in SLAW
 * (e.g. 5.0, -5.5, "Hello" etc.).<br>
 * It is also used for function return data.<br>
 *
 * Note: I tried really hard to make inheritance "work" to
 * make either Constant extend Variable or vice-versa, but
 * the best I could get out of Java was a situation in which
 * Variable was based on Constant, and Constant's properties
 * (i.e. num, str, is_a_string) were "protected" instead
 * of "private", which was bad.  It's a shame Java doesn't
 * have a richer set of protection types; I could have made
 * it work properly with inheritance if Java had something
 * in-between "private" and "protected".
 * Oh well; back to copy-paste it is!
 *
 * <br><br>
 * @see <a href='../SLAWscript.html#Constant'>See "Constant" in the Language Reference
Manual</a>
 * @author Abe and Steve
 *
 *******************************************************/
```

```
public class Constant implements UsableInExpressions {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////
  /**
   * The constant's value if it's a number
   */
  private double num;

  /**
   * The constant's value if it's a string
   */
  private String str;

  /**
   * An boolean to indicate that the constant is a string value
   * (default assumed numeric)
   * This is important, as we might have a string of numbers, so
   * can't rely on the type alone..
   */
  private boolean is_a_string;

  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////

  /*******************************************************
   * Return the constant as a String.  If the constant
   * is numeric, it will be converted to a String
   *
   *******************************************************/
  public String get_as_a_string() {
    if (is_a_string) {
      return str;
    } else { // this will cause the number to be converted to a string
      // this was: return ""+num;

      // the following more-complicated version is so as to produce e.g. "42", not "42.0"
      String temp = ""+num;
      if ( temp.substring( temp.length()-2, temp.length() ).equals(".0") ) {
        temp = temp.substring(0, temp.length()-2);
      }
      return temp;
    }
  }

  /*******************************************************
   * Return the constant as a number (double).  If the constant
   * is a string, type conversion is attempted.
   *
   *******************************************************/
  public double get_as_a_number() {
    if (is_a_string) {
      try {
        return Double.parseDouble(str);
      } catch (NumberFormatException nfe) {
        System.err.println("An error occurred while attempting to convert the string
'"+str+"' to a number.  Exception output follows...");
        System.err.println(nfe);
        System.err.println("Aborting interpreter.");
        System.exit(-1);
      }
    }
    return num;
```

```
    // This is intentionally not inside an "else" to the above "if",
    // both because otherwise "javac" complains that there's a
    // "return" missing here, and also because it doesn't need to
    //be inside an "else"; the "if" part either returns or exits.
  }

  /*******************************************************
   * Returns true _only_ if the data in this Constant is a numeric _string_.
   * Returns false for either a (true number) or a (non-numeric string).
   *******************************************************/
  public boolean is_a_numeric_string() { // this returns true _only_ for numeric _strings_
    if (is_a_string) {
      try {
        Double.parseDouble(str); // intentionally ignoring the result
        return true;
      } catch (NumberFormatException nfe) {
        return false;
      }
    } else { // the following is for "honest-to-goodness" numbers
        return false;
    }
  }

  /*******************************************************
   * Returns true if the constant is a string.  This is
   * important, as we might have a string of numbers, so
   * can't rely on the type alone..
   *
   *******************************************************/
  public boolean is_a_string() {
    return is_a_string;
  }

  /*******************************************************
   * Return the constant's value
   *
   *******************************************************/
  public Constant evaluate() {
    return this; // it's easy to convert a Constant to a Constant - just don't convert
anything!
  }

  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////
   /*****************************************************
   *
   * Disallow the default constructor.
   *
   *****************************************************/
  private Constant() {} // disallow the default constructor

  /*****************************************************
  *
  * Create a constant from a double.
  *
  *****************************************************/
  Constant(double in) {
    is_a_string=false;
    num=in;
  }
```

```
  /**************************************************
  *
  * Create a constant from a boolean.
  *
  **************************************************/
  Constant(boolean in) {
    is_a_string=false;
    num = (in ? 1.0 : 0.0);
  }

  /**************************************************
  *
  * Create a constant from a string.  Flag Constant
  * as string.
  *
  **************************************************/
  Constant(String in) {
    is_a_string=true;
    str=in;
  }


  /**************************************************
  *
  * return true if the datum is usable as a number,
  *        false if it is e.g. "Hello".
  *
  **************************************************/
  public boolean is_usable_as_a_number() {
    return (! is_a_string) || is_a_numeric_string();
  }

} // end of class
```

```
// === copySentence.java === //

// this file was written by Abe

public class copySentence extends NormalParagraphOrNormalSentence {

  private String from, to;

  private copySentence() { }; // disallow the default constructor

  copySentence(String from_in, String to_in) { // the only constructor
    if (null==from_in || null==to_in) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if (! Validator.identifier_is_usable_as_a_variable_name(from_in)) {
      System.err.println("A name ('"+from_in+"') that was not usable for a variable was
attempted to be used as the source in a 'copy' sentence.\nAborting interpreter.");
        System.exit(-1);
    }

    if (! Validator.identifier_is_usable_as_a_variable_name(to_in)) {
      System.err.println("A name ('"+to_in+"') that was not usable for a variable was
attempted to be used as the destination in a 'copy' sentence.\nAborting interpreter.");
        System.exit(-1);
    }
```

```
    from = from_in;
    to = to_in;
  }

  public void doYourThing() {
    VariableStack.put( to, VariableStack.get(from) );
  }

}
```

```
// === DivExpr.java === //

// This file was written by Steve and Abe.

import java.util.Vector;

/*********************************************************
 *
 * The DivExpr represents a division expression
 *
 *
 *
 * <br><br>
 * @see <a href='../SLAWscript.html#Binary_and_Tertiary_Operators'>Binary and Tertiary
Operators</a>
 * @author Steve and Abe
 *
 *********************************************************/
public class DivExpr implements UsableInExpressions {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////
  /**
   * A vector of expressions in a DivExpr
   */
  private Vector<UsableInExpressions> member___expressions;

  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  /*******************************************
   *
   * Evaluates the DivExpr object by iterating
   * over its sub expressions and dividing them.
   *
   *******************************************/
  public Constant evaluate() {

    // note: we do _not_ deal with "no expressions in vector" types of errors in
evaluate(),
    //       not only for this class but in general, because those kinds of errors should
be
    //       caught and dealt with at parse-time (i.e. in the constructors) instead of at
    //       run-time (i.e. "doYourThing" or "evaluate")


    final int exprCount = member___expressions.size();

    // Grab the first one...
    double exprDouble = member___expressions.elementAt(0).evaluate().get_as_a_number();

    double nextDouble; // for the "next" item in the division chain
```

```
    for (int i=1; i < exprCount; i++) {
      nextDouble = member___expressions.elementAt(i).evaluate().get_as_a_number();

      if (0.0 == nextDouble) {
        System.err.println("Error: a division by zero was attempted.\nAborting
interpreter.");
        System.exit(-1);
        // FUTURE: provide the line number of the problem
      } else {
        exprDouble /= nextDouble;
        Validator.validateDouble(exprDouble);
      }
    }

    return new Constant(exprDouble);

  }

  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////
  /************************************************
   *
   * Create a DivExpr with the arguments contained
   * in the supplied vector.
   *
   ************************************************/
  DivExpr(Vector<UsableInExpressions> expressions) {
    if (null==expressions || expressions.size()<2) { // We don't have enough expressions
      System.err.println("Not enough sub-expressions were provided to the DivExpr
constructor.  Number of sub-expressions provided: "+expressions.size()+".\nThis should
never happen.  Aborting interpreter.");
      System.exit(-1);
    }
    this.member___expressions = expressions;
  }

  /************************************************
   *
   * Create a DivExpr with the arguments contained
   * in the two supplied UsableInExpression objects.
   *
   ************************************************/
  DivExpr(UsableInExpressions leftExpr, UsableInExpressions rightExpr) {
    if (null==leftExpr || null==rightExpr) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___expressions = new Vector<UsableInExpressions>();
    member___expressions.add(leftExpr);
    member___expressions.add(rightExpr);
  }

  private DivExpr() {} // Disallow the default constructor
}
```

```
// === doSentence.java === //

import java.util.Vector;
/**************************************************************
 *
 * The doSentence class represents a SLAWscript command
 * to execute a procedure.  General usagae of this command
 * is as follows: <br><br>
 *
 *  <code><pre>
 * do say_hello[10]  #executes the say_hello procedure w/ param 10
 * </pre></code>
 *
 * @see <a href='../SLAWscript.html#Procedures'>Procedures in Language Reference
Manual</a>
 *
 * @author Abe, Steve
 *
 **************************************************************/
public class doSentence extends NormalParagraphOrNormalSentence {

  ////////////////////////
  // ATTRIBUTES
  ////////////////////////
  /**
   * The name of the procedure - e.g. "say_hello"
   */
  private String member___procedure_name;

  /**
   * The actual parameters.  Parameters are of type
   * UsableInExpression.  These are stored here and
   * evaulated in the doYourThing() method.
   */
  // was: private Vector<UsableInExpressions> member___actual_parameters;
  private UsableInExpressions[] member___actual_parameters;

  ////////////////////////
  // METHODS
  ////////////////////////
  /**************************************************************
   *
   * The doYourThing method represents execution of the do sentence
   * by the SLAWscript parser.  It evaluates the members of the
   * member___actual_parameters Vector, determining Contants for each.
   * It places these inside an array.  It then locates the
   * Procedure class inside the SLAWscript procedure Hashtable, and
   * passes the array (and execution) to the appropriate Procedure object.
   *
   **************************************************************/
  public void doYourThing() {

    // Make and populate an array of Constants...
    // was: Constant[] evaluatedParameters = new
Constant[member___actual_parameters.size()];
    Constant[] evaluatedParameters = new Constant[member___actual_parameters.length];
    // was: for (int j = 0; j<member___actual_parameters.size(); ++j) {
    for (int j = 0; j<member___actual_parameters.length; ++j) {
      // was: evaluatedParameters[j] = member___actual_parameters.elementAt(j).evaluate();
      evaluatedParameters[j] = member___actual_parameters[j].evaluate();
    }

    // Pass this puppy the appropriate Procedure...
    final Procedure targetProcedure = SLAWscript.member___procedures.get(
member___procedure_name.toLowerCase() );
    if (targetProcedure != null) {
      // System.err.println("DEBUG:doSentence:Found the procedure " +
this.member___procedure_name);
```

```
      targetProcedure.doProcedure(evaluatedParameters);
    } else {
      System.err.println("The interpreter could not locate the procedure '" +
member___procedure_name + "'.\nThis should never happen.\nAborting interpreter.");
      System.exit(-1);
    }
  } // end of "doYourThing"

  ////////////////////////
  // CONSTRUCTORS
  ////////////////////////
  private doSentence() { } // disallow the default constructor

  /**********************************************************
   *
   * Creates a doSentence object.  Sets the procedure names
   * and populates the UsableInExpression Vector.
   *
   * @param actual_parameters
   * @param procedure_name
   **********************************************************/
  doSentence(Vector<UsableInExpressions> actual_parameters, String procedure_name) {
    if (null==actual_parameters || null==procedure_name) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if (! Validator.identifier_is_usable_as_a_subroutine_name(procedure_name)) {
      System.err.println("Found an error in the name of a procedure in the parser phase:
name='"+procedure_name+"'.  This should never happen.  Bailing out.");
      System.exit(-1);
    }

    member___actual_parameters = actual_parameters.toArray(new UsableInExpressions[0]);
    member___procedure_name    = procedure_name;
  }
} // end of class
```

```
// === DoubleQuestionMarkExpr.java === //

public class DoubleQuestionMarkExpr implements UsableInExpressions {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////

  private String member___name;


  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////

  public Constant evaluate() { // this is the real reason for this class's existence
    final Variable theVarIfItExists = VariableStack.get_if_it_exists(member___name);

    if (null==theVarIfItExists) {
      System.err.println("DEBUG POINT 1 for '"+member___name+"'");
      return new Constant(0.0);
    } else if ( theVarIfItExists.is_a_string() ) {
      System.err.println("DEBUG POINT 2 for '"+member___name+"'");
      return new Constant(1.0);
    } else {
```

```
      System.err.println("DEBUG POINT 3 for '"+member___name+"'");
      return new Constant(2.0);
    }

  } // end of "evaluate"


  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////

   /**************************************************
   *
   * Disallow the default constructor.
   *
   **************************************************/
  private DoubleQuestionMarkExpr() { }

  public DoubleQuestionMarkExpr(String name) {
    if (null==name) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

     if (Validator.identifier_is_usable_as_a_variable_name(name)) {
       member___name = name;
     } else {
       System.err.println("An invalid variable identifier was attempted to be used before
a single question mark in an expression.");
       System.err.println("Aborting interpreter.");
       System.exit(-1);
     }
  }
}
```

```
// === Factorial.java === //

// this file was written by Abe

public class Factorial {

  public static long factorial(long input) {
    if ( input<0 ) {
      System.err.println("Sorry, the factorial operator received a number it doesn't
understand: "+input+"; aborting.");
      System.exit(-1);
    }

    long temp1=input, temp2=1; // sorry about the variable names; I couldn't think of
better ones at 3:08am

    while (temp1 > 1.0) {
      temp2 *= temp1;
      --temp1;
    }

    return temp2;
  }


  public static double factorial(double input) {
    return (double) factorial(Math.round(input));
  }
```

```
  public static void main(String[] args) { // for testing
    System.out.println("Computed factorial of 0 (expecting 1): "+factorial(0));
    System.out.println("Computed factorial of 1 (expecting 1): "+factorial(1));
    System.out.println("Computed factorial of 2 (expecting 2): "+factorial(2));
    System.out.println("Computed factorial of 3 (expecting 6): "+factorial(3));
    System.out.println("Computed factorial of 3.01 (expecting 6): "+factorial(3.01));
    System.out.println("Computed factorial of 9 (expecting 362880): "+factorial(9));
  }

} // end of class
```

```
// === FactorialExpr.java === //

// this file was written by Abe

public class FactorialExpr implements UsableInExpressions {

  private UsableInExpressions member___operand;

  private FactorialExpr() { } // disallow the default constructor

  FactorialExpr(UsableInExpressions in) {
    if (null==in) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___operand = in;
  }


  public Constant evaluate() {

    return new Constant( Factorial.factorial(
      member___operand.evaluate().get_as_a_number() ) );

  }

} // end of class
```

```
// === Function.java === //

// this file was written by Abe and Steve

/**********************************************************
 *
 * The Function class represents a function in SLAWscript
 * <br><br>
 * From paragraph 3.2, SLAWscript Language Reference Manual:<br><br>
 *
 * "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."
 * <br><br>
 * Example:<br><br>
 *
 * </pre>
 * 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 "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
 * </pre>
 *
 * @author Abe and Steve
 *
 ******************************************************/

import java.util.Vector;

/*****************************************************
*
* The Function class represents a function in SLAWscript.
*
*
 *@see <a href="../SLAWscript.html#Functions">Functions Defined in Language Reference
Manual</a>
* @author Abe and Steve
*****************************************************/
public class Function { // intentionally not "implements UsableInExpressions"

  ////////////////////
  // ATTRIBUTES
  ////////////////////
  private NormalParagraphOrFunctionValidSentence[] member___code;
  private String[]                                 member___formal_parameters;
  private String                                   member___name;

  ////////////////////
  // METHODS
  ////////////////////
  /*****************************************************
   *
   * Execute the function.  This involves:<br><br>
   *
   * <ol>
   *  <li>Assigning values to the function's parameters<li>
   *  <li>Iterating over the vector of member body code and finding the return value.</li>
   * </ol>
   *
   *****************************************************/
  public Constant doFunction(Constant[] actual_params) { // intentionally not "evaluate"

    if ( actual_params.length != member___formal_parameters.length ) {
      System.err.println("An error occurred while invoking the function
'"+member___name+"': the number of parameters expected was
"+member___formal_parameters.length+", but the number received was
"+actual_params.length+".\nAborting interpreter.");
      System.exit(-1);
    }

    final int previous_context = VariableStack.current_context_number(); // Preserve the
previous context.
```

```
    VariableStack.new_context();

    // Add parameters to the new context...
    for (int j = 0; j<member___formal_parameters.length; ++j) {
      if ( actual_params[j].is_a_string() ) {
        VariableStack.put_at_top(member___formal_parameters[j], new
Variable(actual_params[j].get_as_a_string()));
      } else {
        VariableStack.put_at_top(member___formal_parameters[j], new
Variable(actual_params[j].get_as_a_number()));
      }
    }

    // Iterate over the code...
    NormalParagraphOrFunctionValidSentence theCode;
    for (int j = 0; j<member___code.length; ++j) {
      try {
        theCode = member___code[j];
        if (theCode instanceof returnSentence) {
          // Note: this must be done carefully, in three steps, in case the return
expression relies on variables that are local to this function (which is very likely)
          final Constant result = ((returnSentence) theCode).getReturnValue();
          while (VariableStack.current_context_number()>previous_context)
VariableStack.previous_context(); // Restore the previous context.
          return result;
        } else if (theCode instanceof FunctionIfParagraph) {
          final Constant result = ((FunctionIfParagraph) theCode).getReturnValue();
          if (null != result) {
            while (VariableStack.current_context_number()>previous_context)
VariableStack.previous_context(); // Restore the previous context.
            return result;
          }
        } else {
          theCode.doYourThing(); // this is where all parts of the code except for
"return" happen
        }
      } catch (java.io.IOException e) {
        System.err.println("There was an IO exception while executing the function '" +
member___name +"'.\nAborting interpreter.\n");
        System.exit(-1);
      }
    }
    System.err.println("An error occurred while invoking the function
'"+member___name+"': the function ended before executing a 'return' statement!  This is
an error.\nAborting interpreter.");
    System.exit(-1);

    return null; // this is here so the code will compile
  }

  //////////////////
  // CONSTRUCTORS
  //////////////////
  /*****************************************************
   *
   * Default constructor not allowed.
   *
   *****************************************************/
  private Function() { } // disallow the default constructor

  /*****************************************************
   *
   * Creates a new SLAWscript Function
   *
   * @param code  A Vecotor of NormalParagraphOrFunctionValidSentences
   *              that represent the body of the code.
   *
   * @param params  A Vector of parameter names.
```

```
   *
   * @param name The name of the function.
   *
   ************************************************/
  Function(Vector<NormalParagraphOrFunctionValidSentence> code, Vector<String> params,
    String name) {

    if (null==code || null==params || null==name) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    Validator.add_subroutine_name(name); // do this first to save time in case of an error

    member___code = code.toArray(new NormalParagraphOrFunctionValidSentence[0]);
      // zero is correct here: it is a base case for array length
    member___formal_parameters = params.toArray(new String[0]);
      // zero is correct here: it is a base case for array length
    member___name = name;

    // The reason for converting from vectors to arrays: once the parsing is finished,
    // the number of sentences/paragraphs in the code
    // and the number of parameters will not change, so let's make an entire class of bugs
    // impossible by not allowing those numbers to change after parsing has finished.
  }
}
```

```
// === FunctionCallWithParams.java === //

// this file was written by Abe and Steve

import java.util.Vector;

public class FunctionCallWithParams implements UsableInExpressions {

  private String member___function_name;
  private UsableInExpressions[] member___actual_parameters;

  private FunctionCallWithParams() { } // disallow the default constructor

  FunctionCallWithParams(Vector<UsableInExpressions> actual_parameters,
    String function_name) {
    if (null==actual_parameters || null==function_name) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if (actual_parameters.size()<1) {
      System.err.println("Found an error in the number of parameters to a function in the
parser phase: claimed to be at least one, found "+actual_parameters.size()+".\nThis
should never happen.  Bailing out.");
      System.exit(-1);
    }

    if (! Validator.identifier_is_usable_as_a_subroutine_name(function_name)) {
      System.err.println("Found an error in the name of a function in the parser phase:
name='"+function_name+"'.\nThis should never happen.  Aborting interpreter.");
      System.exit(-1);
    }
```

```
    member___actual_parameters = actual_parameters.toArray(new UsableInExpressions[0]);
      // zero is the array-length base case
    member___function_name     = function_name;
  }


  public Constant evaluate() {
    final Function targetFunction = SLAWscript.member___functions.get(
      member___function_name.toLowerCase() );
    if (null == targetFunction) {
      System.err.println("The interpreter could not locate the function '" +
        member___function_name + "'.\nThis should never happen.  Aborting interpreter.");
      System.exit(-1);
    } else { // we found the function; let's try to invoke it...
      Constant[] evaluatedParameters = new Constant[member___actual_parameters.length];
      for (int j = 0; j<member___actual_parameters.length; ++j) {
        evaluatedParameters[j] = member___actual_parameters[j].evaluate();
      }
      return targetFunction.doFunction(evaluatedParameters);
    }

    return null; // to get the Java compiler to shut up
  }

} // end of class
```

```
// === FunctionIfParagraph.java === //

// this file was written by Abe

import java.util.Vector;
import java.io.IOException;

/*************************************************************
*
*  FunctionIfParagraph embodies SLAWscript if constructs
*  that use a function as part of the if condition.
*
*
*************************************************************/
public class FunctionIfParagraph extends NormalParagraphOrFunctionValidSentence {

  private Vector<UsableInExpressions>                                member___conditions;
  private Vector<NormalParagraphOrFunctionValidSentence>             member___ifCode;
  private Vector< Vector<NormalParagraphOrFunctionValidSentence> > member___elseIfCode;
  private Vector<NormalParagraphOrFunctionValidSentence>             member___elseCode;

  public void doYourThing() {
    System.err.println("An 'if' paragraph object inside a SLAWscript function had its
doYourThing() called.  Although this member must exist, it should never be
used.\nAborting interpreter.");
    System.exit(-1);
  }

  private FunctionIfParagraph() { } // disallow the default constructor

  FunctionIfParagraph( Vector<UsableInExpressions>                                conditions,
                       Vector<NormalParagraphOrFunctionValidSentence>             ifCode,
                   Vector< Vector<NormalParagraphOrFunctionValidSentence> > elseIfCode,
                       Vector<NormalParagraphOrFunctionValidSentence>             elseCode)
  {

    if (null==conditions || null==ifCode || null==elseIfCode || null==elseCode) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
```

```
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___conditions = conditions;
    member___ifCode     = ifCode;
    member___elseIfCode = elseIfCode;
    member___elseCode   = elseCode;

    if ( conditions.size() != ( 1+elseIfCode.size() ) ) {
      System.err.println("A wierd condition occurred during parsing: the number of
conditions passed in to an 'if' constructor was not as expected.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

  }

  // note: the following should return null if there was no SLAWscript "return" sentence
executed during the "if"
  public Constant getReturnValue() {

    try {

      if (member___conditions.elementAt(0).evaluate().get_as_a_number() != 0.0) {
        return doThisCode(member___ifCode);
      } else { // don't let this fool you: this section has to handle SLAWscript "else
if" subparagraphs as well as a possible "else"
        for (int i = 0; i<member___elseIfCode.size(); ++i) {
          if (member___conditions.elementAt(1+i).evaluate().get_as_a_number() != 0.0) {
            return doThisCode( member___elseIfCode.elementAt(i) );
          }
        }
        return doThisCode(member___elseCode); // invariant here because "if" and "else
if" are protected from this code path
      }

    } catch (IOException ioe) {
      System.err.println("There was an IO error while trying to use a function in an if
condition.\nAborting interpretter.\n");
    }

    return null; // the default, which means that the "if" (incl. "else if" & "else") did
not execute a "return"

  }


  // note: the following should return null if there was no SLAWscript "return" sentence
executed during the "if"
  private Constant doThisCode(Vector<NormalParagraphOrFunctionValidSentence> theCodeToDo)
throws IOException { // an internal service method
    if (null==theCodeToDo) {
      System.err.println("Funky Error in 'if' code.\nAborting interpreter.");
      System.exit(-1);
    }

    NormalParagraphOrFunctionValidSentence theCode;

    // note: given the design of the following loop, it should be OK to pass in a no-op
subparagraph (e.g. "if blah \n end if")
    for (int i = 0; i<theCodeToDo.size(); ++i) {
      theCode = theCodeToDo.elementAt(i);
      if (theCode instanceof returnSentence) {
        return ((returnSentence) theCode).getReturnValue();
      } else if (theCode instanceof FunctionIfParagraph) {
```

```
        final Constant theSubIfSaid = ((FunctionIfParagraph) theCode).getReturnValue();

        if (null != theSubIfSaid)  return theSubIfSaid;
        // otherwise, keep on iterating over _this_ if's code
      } else {
        theCode.doYourThing();
      }
    }
    return null; // the default, which means that the "if" (incl. "else if" & "else") did
not execute a "return"
  }

}
```

```
// === getSentence.java === //

// this file was written by Abe

import java.io.*;

public class getSentence extends NormalParagraphOrNormalSentence {

  private String variable_name;

  private getSentence() {} // disallow the default constructor

  getSentence(String in) { // the only constructor
    if (null==in) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if (! Validator.identifier_is_usable_as_a_variable_name(in)) {
      System.err.println("A name ('"+in+"') that was not usable for a variable was
attempted to be used in a 'get' sentence.\nAborting interpreter.");
       System.exit(-1);
    }

    variable_name = in;
  }

  public void doYourThing() throws IOException {
    VariableStack.put( variable_name, new Variable( (new BufferedReader(new
InputStreamReader(System.in))).readLine() ) ); // God, how I hate Java I/O
  }

}
```

```
// === GreaterThanExpr.java === //

/*********************************************************
 *
 * The GreaterThanExpr models an expression of
 * the following form:<br><br>
 *
 * UsableInExpressions1  > UsableInExpressions2
 *
 * Note 1: IAW the LRM, this operator attempts to perform
 *         auto conversion (to a number) on both sides of
 *         the > operator. <br><br>
```

```
 *
 * Examples:  <br><br>
 *
 * <pre><code>
 *  6 > 4         *OK*
 *  6 > "5.0"     *OK*
 *  a > b         *OK*
 *  a > 5.0       *OK *
 *  a > "cat"     *NOT OK*
 * </code></pre>
 * <br><br>
 *
 * @author Abe and Steve
 *
 ******************************************************/
public class GreaterThanExpr implements UsableInExpressions {

  ////////////////////
  // ATTRIBUTES
  ////////////////////
  /**
   *  The left side of the expression
   */
  private UsableInExpressions member___left;

  /**
   *  The right side of the expression
   */
  private UsableInExpressions member___right;

  ////////////////////
  // METHODS
  ////////////////////

  /*********************************************************
  *
  * Evaluates the GreaterThanExpr.  This is accomplished
  * by executing the evaluate() method for the LHS and RHS
  * UsableInExpression methods.
  *
  * @return Constant The result of evaluating the
  *                  GreaterThanExpr.
  *********************************************************/
  public Constant evaluate() {

    final Constant left  = member___left.evaluate();
    final Constant right = member___right.evaluate();

    return new Constant( left.get_as_a_number() > right.get_as_a_number() );

  }

  ////////////////////
  // CONSTRUCTORS
  ////////////////////
  /*********************************************************
   *
   * Disallow the default constructor.
   *
   *********************************************************/
  private GreaterThanExpr() { } // disallow the default constructor

  /*********************************************************
  *
  * Creates a new GreaterThanExpr with the supplied
  * left and right UsableInExpressions.
  *
  * @param left  The UsableInExpression class for the LHS
```

```
  * @param right The UsableInExpression class for the RHS
  *
  *******************************************************/
  GreaterThanExpr(UsableInExpressions left, UsableInExpressions right) {
    if (null==left || null==right) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___left  = left;
    member___right = right;
  }
} // end of class
```

```
// === GreaterThanOrEqualExpr.java === //

/*******************************************************
 *
 * The GreaterThanOrEqualExpr models an expression of
 * the following form:<br><br>
 *
 * UsableInExpressions1  &gt;= UsableInExpressions2
 *
 * Note: In accordance with our LRM, this operator attempts to perform
 *       auto-conversion (to a number) on both sides of
 *       the &gt;= operator. <br><br>
 *
 * Examples:  <br><br>
 *
 * <pre><code>
 *  6 &gt;= 4         *OK*
 *  6 &gt;= "5.0"     *OK*
 *  a &gt;= b         *OK*
 *  a &gt;= 5.0       *OK *
 *  a &gt;= "cat"     *NOT OK*
 * </code></pre>
 * <br><br>
 *
 * @author Abe and Steve
 *
 *******************************************************/
public class GreaterThanOrEqualExpr implements UsableInExpressions {

  ////////////////////
  // ATTRIBUTES
  ////////////////////
  /**
   *  The left side of the expression
   */
  private UsableInExpressions member___left;

  /**
   *  The right side of the expression
   */
  private UsableInExpressions member___right;

  ////////////////////
  // METHODS
  ////////////////////
  /*******************************************************
   *
   * Evaluates the GreaterThanOrEqualExpr.  This is accomplished
   * by executing the evaluate() method for the LHS and RHS
```

```
   * UsableInExpression methods.
   *
   * @return Constant The result of evaluating the
   *                  GreaterThanOrEqualExpr.
   ******************************************************/
  public Constant evaluate() {

    final Constant left  = member___left.evaluate();
    final Constant right = member___right.evaluate();

    // Note: We don't need to check for the possibility
    //       of either the LHS or RHS being a non-numeric
    //       string this is handled in the paricular
    //       implementations of UsableInExpressions.
    return new Constant( left.get_as_a_number() >= right.get_as_a_number() );

  }
  ////////////////////
  // CONSTRUCTORS
  ////////////////////
  /*********************************************************
   *
   * Disallow the default constructor
   *
   *********************************************************/
  private GreaterThanOrEqualExpr() { }

  /*********************************************************
   *
   * Creates a new GreaterThanOrEqualExpr with the supplied
   * left and right UsableInExpressions.
   *
   * @param left  The UsableInExpression class for the LHS
   * @param right The UsableInExpression class for the RHS
   *
   *********************************************************/
  GreaterThanOrEqualExpr(UsableInExpressions left, UsableInExpressions right) {
    if (null==left || null==right) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___left  = left;
    member___right = right;
  }

} // end of class
```

```
// === Identifier.java === //

/*********************************************************
 *
 * The Identifier class is used to model a usage of a variable
 * or of a zero-parameters function call within an expression.<br><br>
 *
 * It's important not to confuse Identifier with Variable.
 * Identifier is used to capture the concept of the
 * variable's identifer in SLAW, e.g. "cat", 'i', "dog", "myVar", etc.
 * <br><br>
 * The Variable class is used internally to track the
 * possibly-evolving value of a variable.
 *
 * <br><br>
```

```
 * @see <a href='SLAWscript.html#Identifiers'>See Identifiers in the Language Reference
Manual</a>
 * @author Abe
 *
 ******************************************************/
//
public class Identifier implements UsableInExpressions {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////

  private String member___name;

  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  /*************************************************
   *
   * Evaluates a VariableName.  Simple as going to
   * the VariableStack and grabbing the current value.
   *
   *************************************************/
  public Constant evaluate() { // this is the real reason for this class's existence
    if ( Validator.identifier_is_usable_as_a_variable_name(member___name) ) {

      return VariableStack.get(member___name).convert_to_Constant(); // reminder: no need
to canonicalize case here; "get" does it already

    } else { // evaluate the zero-parameters function call

      final Function targetFunction = SLAWscript.member___functions.get(
member___name.toLowerCase() );
      if (null == targetFunction) {
        System.err.println("The interpreter could not locate the function '" +
member___name + "'.\nThis should never happen.\nAborting interpreter.");
        System.exit(-1);
      } else {
        return targetFunction.doFunction(new Constant[0]); // 0 as in 0 param.s
      }

    }

    System.err.println("In 'Identifier.java': this code should never execute.\nAborting
interpreter.");
    System.exit(-1);
    return null; // just to satisfy the compiler

  } // end of "evaluate"

  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////
  /*************************************************
   *
   * Disallow the default constructor.
   *
   *************************************************/
  private Identifier() { }

  /*************************************************
   * Creates a new VariableName from and equal to
   * the supplied string.
   *
   * @param name
   *************************************************/
  public Identifier(String name) {
    if (null==name) {
```

```
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if (Validator.identifier_is_usable_as_a_subroutine_name(name)) {

      // intentionally not "if (Validator.identifier_is_usable_as_a_variable_name(name))
{"
      // because the identifier might represent a function, and "...subroutine..." is
less
      // restrictive than "...variable..."

      member___name = name;
    } else {
      System.err.println("An invalid identifier was attempted to be used in an
expression.");
      System.err.println("This should never happen.  Aborting interpreter.");
      System.exit(-1);
    }
  }
}
```

```
// === ignoreSentence.java === //

// this file was written by Abe based on a file written by Steve

public class ignoreSentence extends NormalParagraphOrNormalSentence {

  private UsableInExpressions member___expression;

  public void doYourThing() {
    member___expression.evaluate(); // intentionally ignoring the result
  }

  ignoreSentence(UsableInExpressions anExpression) {
    if (null==anExpression) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___expression = anExpression;
  }

  private ignoreSentence() {} // disallow the default constructor

} // end of class
```

```
// === InstrExpr.java === //

// this file was written by Abe

// this class implements the ':' operator, which attempts to find the right-hand-side
operand's string in the left operand's string

public class InstrExpr implements UsableInExpressions {

  private UsableInExpressions member___left, member___right;
```

```
  private InstrExpr() { } // disallow the default constructor

  InstrExpr(UsableInExpressions left, UsableInExpressions right) {
    if (null==left || null==right) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___left  = left;
    member___right = right;
  }


  public Constant evaluate() {

    final String left  = member___left.evaluate().get_as_a_string();
    final String right = member___right.evaluate().get_as_a_string();

    if ( left.equals("") ) {

      if ( right.equals("") ) { // they are identically empty
         return new Constant(1.0); // TO DO: document the new behavior specification that
"":"" yields 1
      } // no need for an "else" here due to the invariant "return"
      return new Constant(0.0); // the answer of zero means "no, the right operand is
_provably_ not contained within the left operand"

    } else if ( right.equals("") ) { // the empty string is implicitly contained in every
string, but its position is usually undefined

      return new Constant(-1.0); // TO DO: document the new behavior specification that
x:"" yields -1 _only_ for x<>""

    } else { // here's the non-trivial case

      return new Constant( left.indexOf(right)+1.0 ); // plus one so as to account for
(Java: starts at zero) vs. (SLAWscript: starts at one)
                                                       // Side benefit: +1 automatically
maps Java's "not found" (-1) to our "not found" (0).

    }

  }

} // end of class
```

```
// === LessThanExpr.java === //

// this file was written by Abe
/*********************************************************
 *
 * The LessThanExpr models an expression of
 * the following form:<br><br>
 *
 * UsableInExpressions1 &lt; UsableInExpressions2
 *
 * Note: In accordance with our LRM, this operator attempts to perform
 *       auto-conversion (to a number) on both sides of
 *       the &lt; operator. <br><br>
 *
 * Examples:  <br><br>
 *
 * <pre><code>
```

```
 *  6 &lt; 4        *OK*
 *  6 &lt; "5.0"    *OK*
 *  a &lt; b        *OK*
 *  a &lt; 5.0      *OK*
 *  a &lt; "cat"    *Not OK*
 * </code></pre>
 * <br><br>
 *
 * @author Abe and Steve
 *
 ******************************************************/
public class LessThanExpr implements UsableInExpressions {

  ////////////////////
  // ATTRIBUTES
  ////////////////////
  /**
   *  The left side of the expression
   */
  private UsableInExpressions member___left;
  /**
   *  The right side of the expression
   */
  private UsableInExpressions member___right;

  ////////////////////
  // METHODS
  ////////////////////
  /*********************************************************
  *
  * Evaluates the LessThanExpr.  This is accomplished
  * by executing the evaluate() method for the LHS and RHS
  * UsableInExpression methods.
  *
  * @return Constant The result of evaluating the
  *                  LessThanExpr.
  *********************************************************/
  public Constant evaluate() {

    final Constant left  = member___left.evaluate();
    final Constant right = member___right.evaluate();

    return new Constant( left.get_as_a_number() < right.get_as_a_number() );

  }
  ////////////////////
  // CONSTRUCTORS
  ////////////////////
  /************************************************
   *
   * Disallow default constructor.
   *
   ************************************************/
  private LessThanExpr() { } // disallow the default constructor

  /*******************************************************
  *
  * Creates a new GreaterThanExpr with the supplied
  * left and right UsableInExpressions.
  *
  * @param left  The UsableInExpression class for the LHS
  * @param right The UsableInExpression class for the RHS
  *
  *******************************************************/
  LessThanExpr(UsableInExpressions left, UsableInExpressions right) {
    if (null==left || null==right) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
```

```
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___left  = left;
    member___right = right;
  }


} // end of class
```

```
// === LessThanOrEqualExpr.java === //

// this file was written by Abe


public class LessThanOrEqualExpr implements UsableInExpressions {

  private UsableInExpressions member___left, member___right;

  private LessThanOrEqualExpr() { } // disallow the default constructor

  LessThanOrEqualExpr(UsableInExpressions left, UsableInExpressions right) {
    if (null==left || null==right) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___left  = left;
    member___right = right;
  }


  public Constant evaluate() {

    final Constant left = member___left.evaluate();
    final Constant right = member___right.evaluate();

    return new Constant( left.get_as_a_number() <= right.get_as_a_number() );

  }

} // end of class
```

```
// === localizeSentence.java === //

// this file was written by Abe

public class localizeSentence extends NormalParagraphOrSubroutineValidSentence {

  private String variable_name;

  private localizeSentence() {} // disallow the default constructor

  localizeSentence(String in) { // the only constructor
    if (null==in) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
```

```
    }

    if (! Validator.identifier_is_usable_as_a_variable_name(in)) {
      System.err.println("A name ('"+in+"') that was not usable for a variable was
attempted to be used in a 'localize' sentence.\nAborting interpreter.");
        System.exit(-1);
    }

    variable_name = in;
  }

  public void doYourThing() { // the assumption is that the subroutine's context is
already set up
    VariableStack.reserve(variable_name);
  }

}
```

```
// === MinusExpr.java === //

// this file was written by Abe, modified by Steve

import java.util.Vector;

/*********************************************************
 *
 * @author Abe, Steve
 *
 *********************************************************/
public class MinusExpr implements UsableInExpressions {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////
  /**
   * A vector of expressions in a MinusExpr
   */
  private Vector<UsableInExpressions> member___expressions;

  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  /*******************************************
   *
   * Evaluates the MinusExpr object by iterating
   * over its sub expressions and subtracting them.
   *
   *******************************************/
  public Constant evaluate() {

    final int exprCount = member___expressions.size();

    // Grab the first one...
    double exprDouble = member___expressions.elementAt(0).evaluate().get_as_a_number();

    for (int i=1; i < exprCount; i++) {
      exprDouble -= member___expressions.elementAt(i).evaluate().get_as_a_number();

      Validator.validateDouble(exprDouble);
    }

    return new Constant(exprDouble);
  }
```

```
  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////

  /************************************************
   *
   * Create a MinusExpr with the arguments contained
   * in the supplied vector.
   *
   ************************************************/
  MinusExpr(Vector<UsableInExpressions> expressions) {
    if (null==expressions) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if (expressions.size()<2) { // We don't have enough expressions (minimum=2)
      System.err.println("Not enough sub-expressions were provided to the MinusExpr
constructor.  Number of sub-expressions provided: "+expressions.size()+".\nThis should
never happen.  Aborting interpreter.");
      System.exit(-1);
    }
    member___expressions = expressions;
  }

  /************************************************
   *
   * Create a MinusExpr with the arguments contained
   * in the two supplied UsableInExpression objects.
   *
   ************************************************/
  MinusExpr(UsableInExpressions leftExpr, UsableInExpressions rightExpr) {
    if (null==leftExpr || null==rightExpr) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___expressions = new Vector<UsableInExpressions>();
    member___expressions.add(leftExpr);
    member___expressions.add(rightExpr);
  }

  private MinusExpr() { } // disallow the default constructor

} // end of class
```

```
// === MulExpr.java === //

import java.util.Iterator;
import java.util.Vector;

/*******************************************************
 *
 * The MulExpr represents a multiplication expression.
 * <br><br>
 * General form:  <UsableInExpressions_1> * <UsableInExpressions_2>
 * <br><br>
 * Examples:
 *
```

```
 * <pre><code>
 * 1 * 2
 * a * 2
 * f * g
 * f * somefunction[6]
 * "Bye" * 2
 * "Bye" * "2"
 * 2 * "Bye"
 * "2" * "Bye"
 * </code></pre>
 *
 *
 * <br><br>
 *
 * @author Abe
 *
 ******************************************************/
public class MulExpr implements UsableInExpressions {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////
  /**
   * A vector of expressions in a MulExpr
   */
  private Vector<UsableInExpressions> member___expressions;

  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  /*********************************************
  *
  * Evaluates the MulExpr object by iterating
  * over its sub-expressions
  *
  *********************************************/
  public Constant evaluate() {

    // note: we do _not_ deal with "no expressions in vector" types of errors in
    // evaluate(), not only for this class but in general, because those kinds of errors
    // should be caught and dealt with at parse-time (i.e. in the constructors) instead
    // of at run-time (i.e. "doYourThing" or "evaluate")

    final int exprCount = member___expressions.size();

    // Grab the first one...
    Constant left = member___expressions.elementAt(0).evaluate(); // intentionally not
"final"
    // System.err.println(this + ":my vector size= " + exprCount); // testing code

    Constant right;

    loop:
    for (int i=1; i<exprCount; i++) {
      right = member___expressions.elementAt(i).evaluate();

      if ( left.is_a_string() ) {

        if ( right.is_usable_as_a_number() ) {
          final long right_num = Math.round( right.get_as_a_number() );
          if (right_num<0) {
            if ( left.is_usable_as_a_number() ) {
              final double left_dbl = left.get_as_a_number() * right.get_as_a_number();
              Validator.validateDouble(left_dbl);
              left = new Constant(left_dbl);
              continue loop; // to make sure we don't accidentally do anything else
before the next loop iteration or the end of the loop
            } else {
```

```
              System.err.println("A multipication expression is trying to multiply a non-
numeric string by a number (or a numeric string) which is still negative after
rounding.\nAborting interpreter.");
              System.exit(-1);
            }
          } else { // right_num must now be >= 0
            left = new Constant( multiplyString(left.get_as_a_string(), right_num) );
            continue loop; // to make sure we don't accidentally do anything else before
the next loop iteration or the end of the loop
              }
            } else if ( left.is_usable_as_a_number() ) {
          final long left_num = Math.round( left.get_as_a_number() );
          if (left_num<0) {
            System.err.println("A multipication expression is trying to multiply a number
(or a numeric string) which is still negative after rounding by a non-numeric
string.\nAborting interpreter.");
            System.exit(-1);
          } else { // left_num must now be >= 0
            left = new Constant( multiplyString(right.get_as_a_string(), left_num) );
            continue loop; // to make sure we don't accidentally do anything else before
the next loop iteration or the end of the loop
          }
        } else { // case of non-numeric-string * non-numeric string, e.g. "Hi"*"Hi"
          System.err.println("A multipication expression is trying to multiply a non-
numeric string by a non-numeric string.\nAborting interpreter.");
          System.exit(-1);
        }

      } else { // "left" is _not_ a string, i.e. it is a number

        // FUTURE: redo the case-folding for this part to make it more efficient
        // FUTURE: put the "final long..." later in the section after redo-ing the case-
folding

        final long left_num = Math.round( left.get_as_a_number() );
        if (left_num<0) {
          if (right.is_usable_as_a_number() ) {
            final double left_dbl = left.get_as_a_number() * right.get_as_a_number();
            Validator.validateDouble(left_dbl);
            left = new Constant(left_dbl);
            continue loop; // to make sure we don't accidentally do anything else before
the next loop iteration or the end of the loop
          } else {
            System.err.println("A multipication expression is trying to multiply a number
which is still negative after rounding by a non-numeric string.\nAborting interpreter.");
            System.exit(-1);
          }
        } else // by now "left_num" must be >= 0  (note: intentionally no '{' after the
"else")
        /* else */ if (right.is_usable_as_a_number() ) {
          final double left_dbl = left.get_as_a_number() * right.get_as_a_number();
          Validator.validateDouble(left_dbl);
          left = new Constant(left_dbl);
        } else {
          left = new Constant( multiplyString(right.get_as_a_string(), left_num) );
        }

      } // end of the big "if left is a string ... else ..."

    } // end of the "for" loop

    return left;

  }
```

```
  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////
  /**********************************************
   *
   * Create a MulExpr with the arguments contained
   * in the supplied vector.
   *
   **********************************************/
  MulExpr(Vector<UsableInExpressions> expressions) {
    if (null==expressions) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if (expressions.size()<2) { // We don't have enough expressions
      System.err.println("Not enough sub-expressions were provided to the MulExpr
constructor.  Number of sub-expressions provided: "+expressions.size()+".\nThis should
never happen.  Aborting interpreter.");
      System.exit(-1);
    }
    member___expressions = expressions;
  }

  /**********************************************
   *
   * Create a MulExpr with the arguments contained
   * in the two supplied UsableInExpression objects.
   *
   **********************************************/
  MulExpr(UsableInExpressions leftExpr, UsableInExpressions rightExpr) {
    if (null==leftExpr || null==rightExpr) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___expressions = new Vector<UsableInExpressions>();
    member___expressions.add(leftExpr);
    member___expressions.add(rightExpr);
  }

  private MulExpr() {} // Disallow the default constructor.

  public static String multiplyString(String str, long num) {
    if (num<0) {
      System.err.println("A string was attempted to be multiplied a negative number of
times.  Aborting.");
      System.exit(-1);
    }

    String newString ="";

    for (long j=0; j < num; j++) {
      newString = newString.concat(str);
    }

    return newString;
  }
```

```
  // an executable "main", for testing the "multiplyString" subroutine

  public static void main(String[] a) {
    System.out.print("'Hi'*0: '");
    System.out.println( multiplyString("Hi",0)+"'" );
    System.out.print("'Hi'*3: '");
    System.out.println( multiplyString("Hi",3)+"'" );
    System.out.print("'Hi'*-3: ");
    System.out.println( multiplyString("Hi",-3)+"'" );
  }

} // end of class
```

```
// === NormalIfParagraph.java === //

// this file was written by Abe and Steve

import java.util.Vector;
import java.io.IOException;

public class NormalIfParagraph extends NormalParagraphOrNormalSentence {

  private Vector<UsableInExpressions>                        member___conditions;
  private Vector<NormalParagraphOrNormalSentence>            member___ifCode;
  private Vector< Vector<NormalParagraphOrNormalSentence> > member___elseIfCode;
  private Vector<NormalParagraphOrNormalSentence>            member___elseCode;

  private NormalIfParagraph() { } // disallow the default constructor

  NormalIfParagraph( Vector<UsableInExpressions>                        conditions,
                     Vector<NormalParagraphOrNormalSentence>            ifCode,
                     Vector< Vector<NormalParagraphOrNormalSentence> > elseIfCode,
                     Vector<NormalParagraphOrNormalSentence>            elseCode    )
  {

    if (null==conditions || null==ifCode || null==elseIfCode || null==elseCode) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___conditions = conditions;
    member___ifCode     = ifCode;
    member___elseIfCode = elseIfCode;
    member___elseCode   = elseCode;

    if ( conditions.size() != ( 1+elseIfCode.size() ) ) {
      System.err.println("A wierd condition occurred during parsing: the number of
conditions passed in to an 'if' constructor was not as expected.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

  }

  public void doYourThing() {

    try {

      if (member___conditions.elementAt(0).evaluate().get_as_a_number() != 0.0) {
        doThisCode(member___ifCode);
      } else { // don't let this fool you: this section has to handle SLAWscript "else
if" subparagraphs as well as a possible "else"
```

```
        for (int i = 0; i<member___elseIfCode.size(); ++i) {
          if (member___conditions.elementAt(1+i).evaluate().get_as_a_number() != 0.0) {
            doThisCode( member___elseIfCode.elementAt(i) );
            return; // to avoid having to put a boolean variable in here and a whole
bunch of nested "if"s just to avoid also doing the "else"
          }
        }
        doThisCode(member___elseCode); // invariant here because "if" and "else if" are
protected from this code path
      }

    } catch (IOException ioe) {
      System.err.println("An I/O error occurred inside an 'if' paragraph.\nAborting
interpreter.\n");
    }

  }


  private void doThisCode(Vector<NormalParagraphOrNormalSentence> theCodeToDo) throws
IOException { // an internal service method
    if (null==theCodeToDo) {
      System.err.println("Funky Error in 'if' code.\nAborting interpreter.");
      System.exit(-1);
    }

    // note: given the design of the following loop, it should be OK to pass in a no-op
subparagraph (e.g. "if blah \n end if")
    for (int i = 0; i<theCodeToDo.size(); ++i) {
      theCodeToDo.elementAt(i).doYourThing();
    }
  }

}
```

```
// === NormalParagraphOrFunctionValidSentence.java === //

// this file was written by Abe

public abstract class NormalParagraphOrFunctionValidSentence {
  // This is here just for the class heirarchy and mandating "doYourThing".

  // The only classes that should inherit from this are returnSentence
  // and NormalParagraphOrSubroutineValidSentence.

  public abstract void doYourThing() throws java.io.IOException; // I added the exception
here so that "getSentence" would compile

}
```

```
// === NormalParagraphOrNormalSentence.java === //

// this file was written by Abe

public abstract class NormalParagraphOrNormalSentence extends
NormalParagraphOrSubroutineValidSentence {

  // this class exists strictly for the purpose of having a common parent for its
subclasses
```

```
  // notes from Abe: The "NormalParagraphs" in this class`s name indicates that "define"
  //   paragraphs are not included.  The "NormalSentence" in this class`s name indicates
that
  //   "localize" and "return" sentences are not included.

}
```

```
// === NormalParagraphOrSubroutineValidSentence.java === //

// this file was written by Abe

public abstract class NormalParagraphOrSubroutineValidSentence extends
NormalParagraphOrFunctionValidSentence {
  // This is here just for the class heirarchy.

  // The only classes that should inherit from this are localizeSentence
  // and NormalParagraphOrNormalSentence.

}
```

```
// === notExpr.java === //

// this file was written by Abe


public class notExpr implements UsableInExpressions {

  private UsableInExpressions member___operand;

  private notExpr() { } // disallow the default constructor

  notExpr(UsableInExpressions in) {
    if (null==in) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___operand = in;
  }


  public Constant evaluate() {

    return new Constant( 0.0 == member___operand.evaluate().get_as_a_number() );

  }

} // end of class
```

```
// === orExpr.java === //

// this file was written by Abe


public class orExpr implements UsableInExpressions {

  private UsableInExpressions member___left, member___right;

  private orExpr() { } // disallow the default constructor

  orExpr(UsableInExpressions left, UsableInExpressions right) {
    if (null==left || null==right) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___left  = left;
    member___right = right;
  }


  public Constant evaluate() {

    // note from Abe: for "and" and "or", we do _not_ evaluate both expr.s first!
    //                By doing these as I have done, we inherit Java's short-circuiting.

    return new Constant( (member___left.evaluate().get_as_a_number()!=0.0)
                         ||
                         (member___right.evaluate().get_as_a_number()!=0.0) );

  }

} // end of class
```

```
// === ParserReturnType.java === //

// this file was written by Abe and Steve

// This file exists purely to overcome Java's one-return-value-per-method limit.

import java.util.Hashtable;
import java.util.Vector;


public class ParserReturnType {

  // The following are "public" instead of the usual "private" since this class only
  // exists for the purpose of returning multiple data from the parser.  Also, this way
  // we don't need to write getter methods.

  public NormalParagraphOrNormalSentence[] member___mainBody;
  public Hashtable<String,Function>        member___functions;
  public Hashtable<String,Procedure>       member___procedures;

  private ParserReturnType() { } // disallow the default constructor

  ParserReturnType( Vector<NormalParagraphOrNormalSentence> mainBody,
                    Hashtable<String,Function>              functions,
                    Hashtable<String,Procedure>             procedures)
  {
```

```
    if (null==mainBody || null==functions || null==procedures) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }


    if (mainBody.size() > 0) {
      member___mainBody = (NormalParagraphOrNormalSentence[]) mainBody.toArray(new
NormalParagraphOrNormalSentence[0]);

      // The reason for converting from a vector to an array: once the parsing is
finished,
      // the number of sentences/paragraphs in the main body code
      // will not change, so let's make an entire class of bugs
      // impossible by not allowing that number to change after parsing has finished.
    } else { // reminder: an empty main body is _not_ an error
      member___mainBody = new NormalParagraphOrNormalSentence[0]; // zero elements
    }

    member___functions  = functions;
    member___procedures = procedures;
  }

}
```

```
// === PipeExpr.java === //

// this file was written by Abe


public class PipeExpr implements UsableInExpressions {

  private UsableInExpressions member___operand;

  private PipeExpr() { } // disallow the default constructor

  PipeExpr(UsableInExpressions in) {
    if (null==in) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___operand = in;
  }


  public Constant evaluate() {

    Constant temp = member___operand.evaluate();

    if ( temp.is_a_string() ) {
      return new Constant( temp.get_as_a_string().length() );
    } else {
      return new Constant( Math.abs( temp.get_as_a_number() ) );
    }

  }

} // end of class
```

```
// === PlusExpr.java === //

// this file was written by Abe and Steve
import java.util.Vector;

/*******************************************************
*
* The PlusExpr represents a plus expression:
* <pre>
* 1+2
* a+b
* 3+4+5
* 4+a
* </pre>
* <br><br>
*
* @see <a href='../SLAWscript.html#Binary_and_Tertiary_Operators'>Binary and Tertiary
Operators</a>
* @author Steve and Abe
*
*******************************************************/

// Note to team: this class _should_ be used as a template for implementing '*',
//               since '*' is also overloaded vis-a-vis string vs. number,
//               but it should _not_ be used as a template for any other binary operator;
//               it might be useful (albeit in a cut-down form) as a template for '| |'
//               (overloaded: number->absolute value, string->string length).


public class PlusExpr implements UsableInExpressions {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////
  /**
   * A vector of expressions in a PlusExpr
   */
  private Vector<UsableInExpressions> member___expressions;

  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  /*********************************************
  *
  * Evaluates the PlusExpr object by iterating
  * over its sub expressions and adding them.
  *
  *********************************************/
  public Constant evaluate() {

     // note: we do _not_ deal with "no expressions in vector" types of errors in
evaluate(),
     //       not only for this class but in general, because those kinds of errors
should be
     //       caught and dealt with at parse-time (i.e. in the constructors) instead of at
     //       run-time (i.e. "doYourThing" or "evaluate")

     final int exprCount = member___expressions.size();

     // Grab the first one...
     Constant running_total = member___expressions.elementAt(0).evaluate();

     for (int i=1; i < exprCount; i++) {
       Constant next_element = member___expressions.elementAt(i).evaluate();
```

```
        if ( running_total.is_a_string() || ( ! next_element.is_usable_as_a_number() ) ) {
          // string addition

          running_total = new Constant( running_total.get_as_a_string() +
next_element.get_as_a_string() );

        } else { // the running total is a number, and the new element is usable as a
number

          final double running_total_dbl = running_total.get_as_a_number() +
next_element.get_as_a_number();
          Validator.validateDouble(running_total_dbl);
          running_total = new Constant(running_total_dbl);

        }

      }

      return running_total;

    }
    ////////////////////////////////////////
    // CONSTRUCTORS
    ////////////////////////////////////////

    /***********************************************
    *
    * Prevent the use of the default constructor.
    *
    ***********************************************/
    private PlusExpr() { }

    /***********************************************
    *
    * Create a PlusExpr with the arguments contained
    * in the supplied vector.
    *
    ***********************************************/
    PlusExpr(Vector<UsableInExpressions> expressions) {
      if (null==expressions) {
        System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
        System.err.println("Aborting interpreter.");
        System.exit(-1);
      }

      if (expressions.size()<2) { // We don't have enough expressions
        System.err.println("Not enough sub-expressions were provided to the DivExpr
constructor.  Number of sub-expressions provided: "+expressions.size()+".\nThis should
never happen.  Aborting interpreter.");
        System.exit(-1);
      }
      member___expressions = expressions;
    }

} // end of class
```

```
// === PowerExpr.java === //

// note: in this case, we are intentionally _not_ using the Vector technique because '^'
makes more sense as a
//   right-associative operator (since 2^3^4 as left-associative is equivalent to 2^12,
which makes 2^3^4 silly)
```

```
/*********************************************************
 *
 * The PowerExpr represents a power (exponent) expression
 *
 * <br><br>
 * @see <a href='../SLAWscript.html#Binary_and_Tertiary_Operators'>Binary and Tertiary
Operators</a>
 * @author Steve, Abe
 *
 *********************************************************/
public class PowerExpr implements UsableInExpressions {


  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////
  /**
   * The base of the PowerExpr
   */
  private UsableInExpressions member___base;

  /**
   * The exponent of the PowerExpr
   */
  private UsableInExpressions member___exponent;

  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  /*********************************************
   *
   * Evaluates the PowerExpr object.
   *
   *********************************************/
  public Constant evaluate() {

    double result = Math.pow(member___base.evaluate().get_as_a_number(),
member___exponent.evaluate().get_as_a_number());
    Validator.validateDouble(result);
    return new Constant(result);
  }

  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////
  /*********************************************
   *
   * Create a PowerExpr with the base and exponent contained
   * in the two supplied UsableInExpression objects.
   *
   *********************************************/
  PowerExpr(UsableInExpressions inBase, UsableInExpressions inExponent) {
    if (null==inBase || null==inExponent) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___base = inBase;
    member___exponent = inExponent;
  }

  // Disallow the default constructor.
  private PowerExpr() {}
}
```

```
// === Procedure.java === //

// this file was written by Abe and Steve

import java.util.Vector;


/**************************************************************
 *
 * The Procedure class implements a SLAWscript procedure.
 *
 *
 * @see <a href="../SLAWscript.html#Procedures">Procedures Defined in Language Reference
Manual</a>
 *
 **************************************************************/
public class Procedure {

  ////////////////////////
  // ATTRIBUTES
  ////////////////////////

  /**
   * The code body of the procedure
   */
  private NormalParagraphOrSubroutineValidSentence[] member___code;

  /**
   * A String array of the identifiers used as formal parameters
   */
  private String[]                                   member___formal_parameters;

  /**
   * The name of the procedure
   */
  private String                                     member___name;

  ////////////////////////
  // CONSTRUCTORS
  ////////////////////////
  /**********************************************************
   *
   * Default constructor disallowed.
   *
   **********************************************************/
  private Procedure() { } // disallow the default constructor

  /**********************************************************
   *
   * Creates a new Procedure object.
   *
   * @param code    A vector of NormalParagraphOrSubroutineValiedSenences
   *                thatrepresents the main body of the procedure.
   *
   * @param params  A vector of strings that represents the identifiers
   *                of the procedure's parameters.
   *
   * @param name    The name of the procedure.
   *
   **********************************************************/
  Procedure(Vector<NormalParagraphOrSubroutineValidSentence> code, Vector<String> params,
String name) {
    if (null==code || null==params || null==name) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }
```

```
    Validator.add_subroutine_name(name); // do this first to save time in case of an error

    // was: member___code              = (NormalParagraphOrSubroutineValidSentence[])
code.toArray(new NormalParagraphOrSubroutineValidSentence[code.size()]);
    member___code               = code.toArray(new
NormalParagraphOrSubroutineValidSentence[0]);
    // was: member___formal_parameters = (String[]) params.toArray(new
String[params.size()]);
    member___formal_parameters = params.toArray(new String[0]);
    member___name               = name;

    // The reason for converting from vectors to arrays: once the parsing is finished,
    // the number of sentences/paragraphs in the code
    // and the number of parameters will not change, so let's make an entire class of bugs
    // impossible by not allowing those numbers to change after parsing has finished.

    // System.err.println("DEBUG:Procedure:Created the procedure constructor.");
  }

  public void doProcedure(Constant[] actual_params) {

    if ( actual_params.length != member___formal_parameters.length ) {
      System.err.println("An error occurred while invoking the procedure '"
        +member___name+"': the number of parameters expected was "
        +member___formal_parameters.length+", but the number received was "
        +actual_params.length+".\nAborting interpreter.");
      System.exit(-1);
    }

    final int previous_context = VariableStack.current_context_number(); // Preserve the
previous context.

    // System.err.println("DEBUG:doProcedure:stack context was: " + previous_context);
    VariableStack.new_context();

    // Add parameters to the new context...
    for (int j = 0; j<member___formal_parameters.length; ++j) {
      if ( actual_params[j].is_a_string() ) {
        VariableStack.put_at_top(member___formal_parameters[j],
          new Variable(actual_params[j].get_as_a_string()));
      } else {
        VariableStack.put_at_top(member___formal_parameters[j],
          new Variable(actual_params[j].get_as_a_number()));
      }
    }

    // Iterate over the code...
    for (int j = 0; j<member___code.length; ++j) {
      try {
        member___code[j].doYourThing(); // this is where the pedal hits the metal
      } catch (java.io.IOException e) {
        System.err.println("There was an IO exception while executing the procedure '" +
member___name +"'.\nAborting interpreter.\n");
        System.exit(-1);
      }
    }

    while (VariableStack.current_context_number()>previous_context)
VariableStack.previous_context(); // Restore the previous context.

    // System.err.println("DEBUG:doProcedure:stack context after rollback: " +
VariableStack.current_context_number());

  }

}
```

```
// === ProcedureIfParagraph.java === //

// this file was written by Abe and Steve

import java.util.Vector;
import java.io.IOException;

public class ProcedureIfParagraph extends NormalParagraphOrNormalSentence {

  private Vector<UsableInExpressions>                                    member___conditions;
  private Vector<NormalParagraphOrSubroutineValidSentence>               member___ifCode;
  private Vector< Vector<NormalParagraphOrSubroutineValidSentence> > member___elseIfCode;
  private Vector<NormalParagraphOrSubroutineValidSentence>               member___elseCode;

  private ProcedureIfParagraph() { } // disallow the default constructor

  ProcedureIfParagraph( Vector<UsableInExpressions>
conditions,
                        Vector<NormalParagraphOrSubroutineValidSentence>           ifCode,
                        Vector< Vector<NormalParagraphOrSubroutineValidSentence> >
elseIfCode,
                        Vector<NormalParagraphOrSubroutineValidSentence>
elseCode    )
  {

    if (null==conditions || null==ifCode || null==elseIfCode || null==elseCode) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___conditions = conditions;
    member___ifCode     = ifCode;
    member___elseIfCode = elseIfCode;
    member___elseCode   = elseCode;

    if ( conditions.size() != ( 1+elseIfCode.size() ) ) {
      System.err.println("A wierd condition occurred during parsing: the number of
conditions passed in to an 'if' constructor was not as expected.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

  }

  public void doYourThing() {

    try {

      if (member___conditions.elementAt(0).evaluate().get_as_a_number() != 0.0) {
        doThisCode(member___ifCode);
      } else { // don't let this fool you: this section has to handle SLAWscript "else
if" subparagraphs as well as a possible "else"
        for (int i = 0; i<member___elseIfCode.size(); ++i) {
          if (member___conditions.elementAt(1+i).evaluate().get_as_a_number() != 0.0) {
            doThisCode( member___elseIfCode.elementAt(i) );
            return; // to avoid having to put a boolean variable in here and a whole
bunch of nested "if"s just to avoid also doing the "else"
          }
        }
        doThisCode(member___elseCode); // invariant here because "if" and "else if" are
protected from this code path
      }

    } catch (IOException ioe) {
        System.err.println("A procedure if paragraph generated an IO error.\nAborting
interpreter.\n");
```

```
    }

  }

  private void doThisCode(Vector<NormalParagraphOrSubroutineValidSentence> theCodeToDo)
throws IOException { // an internal service method
    if (null==theCodeToDo) {
      System.err.println("Funky Error in 'if' code.\nAborting interpreter.");
      System.exit(-1);
    }

    // note: given the design of the following loop, it should be OK to pass in a no-op
subparagraph (e.g. "if blah \n end if")
    for (int i = 0; i<theCodeToDo.size(); ++i) {
      theCodeToDo.elementAt(i).doYourThing();
    }
  }
}
```

```
// === putSentence.java === //

import java.io.*;

/*********************************************************
 *
 * The putSentence class models grammar of the
 * following form:<br><br>
 *
 * "put" expr "to" ("stdout"|"stderr")
 *
 * <br><br>
 *
 * @author Steve
 *
 *********************************************************/
public class putSentence extends NormalParagraphOrNormalSentence {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////
  /**
   * A boolean indicating destination is stdout (default)
   */
  private boolean member___destination_stdout = true;

  /**
   * The expression to assign to the variable
   */
  private UsableInExpressions member___expression;

  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  /*********************************************************
   *
   * The doYourThing method is called during "runtime"
   * evaluation of the SLAWscript.  It represents actual
   * execution of the "put expr to blah" SLAW
   *
   *********************************************************/
  public void doYourThing() throws IOException {

    // Grab the Constant from the RHS
    final Constant exprValue = member___expression.evaluate();
```

```
    String outText = exprValue.get_as_a_string();


    if (member___destination_stdout) {
      System.out.print(outText); // _not_ "println"; users must include "\n" if they want
it
      System.out.flush(); // make sure the output is done right away, not whenever the
JVM feels that the buffer is full
    } else {
      System.err.print(outText); // ditto
      System.err.flush(); // make sure the output is done right away, not whenever the
JVM feels that the buffer is full
    }
  }

  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////
  /*********************************************************
   * Creates a putSentence object with destination set
   * to stdout (if d_stdout boolean argument is not false)
   * else set destination to stderr.
   *********************************************************/
  public putSentence(UsableInExpressions expr, boolean d_stdout) {
    if (null==expr) {
      System.err.println("A put sentence object constructor got an expression which is
null!\nAborting interpreter.");
      System.exit(-1);
    }
    member___expression         = expr;
    member___destination_stdout = d_stdout;
  }

  private putSentence() {} // disallow the default constructor

}
```

```
// === randomizeSentence.java === //

// this file was written by Abe

import java.io.*;

public class randomizeSentence extends NormalParagraphOrNormalSentence {

  private String variable_name;

  private randomizeSentence() {} // disallow the default constructor

  randomizeSentence(String in) { // the only constructor
    if (null==in) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if (! Validator.identifier_is_usable_as_a_variable_name(in)) {
      System.err.println("A name ('"+in+"') that was not usable for a variable was
attempted to be used in a 'randomize' sentence.\nAborting interpreter.");
        System.exit(-1);
    }

    variable_name = in;
  }
```

```
  public void doYourThing() {
    VariableStack.put( variable_name, new Variable( Math.random() ) );
  }

}
```

```
// === RelaxedDoesNotEqualExpr.java === //

// this file was written by Abe


public class RelaxedDoesNotEqualExpr implements UsableInExpressions {

  private UsableInExpressions member___left, member___right;

  private RelaxedDoesNotEqualExpr() { } // disallow the default constructor

  RelaxedDoesNotEqualExpr(UsableInExpressions left, UsableInExpressions right) {
    if (null==left || null==right) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___left  = left;
    member___right = right;
  }


  public Constant evaluate() {

    final Constant left  = member___left.evaluate();
    final Constant right = member___right.evaluate();

    if ( left.is_a_string() ) { // reminder: "left-hand dominance"
      return new Constant( ! left.get_as_a_string().equals(right.get_as_a_string()) );
    } else if ( ! right.is_usable_as_a_number() ) {
      return new Constant(true); // if the left is a number, and the right is unusable as
a number, then they cannot possibly be equal
    } else { // we have already ascertained that "left" _is_ a number and that "right" is
at least _usable_ as a number
      return new Constant( left.get_as_a_number() != right.get_as_a_number() );
    }

  }

} // end of class
```

```
// === RelaxedEqualsExpr.java === //

// this file was written by Abe


public class RelaxedEqualsExpr implements UsableInExpressions {

  private UsableInExpressions member___left, member___right;

  private RelaxedEqualsExpr() { } // disallow the default constructor
```

```
  RelaxedEqualsExpr(UsableInExpressions left, UsableInExpressions right) {
    if (null==left || null==right) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___left  = left;
    member___right = right;
  }


  public Constant evaluate() {

    final Constant left  = member___left.evaluate();
    final Constant right = member___right.evaluate();

    if ( left.is_a_string() ) { // reminder: "left-hand dominance"
      return new Constant( left.get_as_a_string().equals(right.get_as_a_string()) );
    } else if ( ! right.is_usable_as_a_number() ) {
      return new Constant(false); // if the left is a number, and the right is unusable
as a number, then they cannot possibly be equal
    } else { // we have already ascertained that "left" _is_ a number and that "right" is
at least _usable_ as a number
      return new Constant( left.get_as_a_number() == right.get_as_a_number() );
    }

  }

} // end of class
```

```
// === repeatParagraph.java === //

// this is here just for the class hierarchy

public abstract class repeatParagraph extends NormalParagraphOrNormalSentence {

  // note from Abe: we do _not_ put "doYourThing()" here, or anything else, for that
matter

}
```

```
// === repeatTimesParagraph.java === //

// This class was written by Abe and Steve.
// import java.util.Iterator;
import java.util.Vector;

/**************************************************************
 *
 * The repeatTimesParagraph class repeats a block of code
 * a particular number of times (pg 12, LRM): <br><br>
 *
 * From paragraph 8.1 of the SLAWscript Language Reference Manual:<br><br>
 *
 * This type of loop is useful for code that needs to be executed
 * a zero-or-more predetermined number of times, and the code inside
 * the loop does not need to keep track of the number of times it has
 * been executed.<br><br>
 *
 * 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. <br><br>
 *
 * 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.<br><br>
 *
 * 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.
 * <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
 *
 ************************************************************************/
public class repeatTimesParagraph extends repeatParagraph {

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

  /**
   * The expression giving the number of times to execute the repeat code block...
   */
  private UsableInExpressions member__timesExpr;

  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  /*************************************************
  *
  * The doYourThing method is called by the SLAWscriptParser
  * when it decides to execute the repeatTimes construct.
  *
  *************************************************/
  public void doYourThing() {

    // reminders: _round_ the double, do not truncate; use a _long_, not an int, for
counting

    // caveat - not using dot chaining here in order to execute evaluate once
    final long repeatTimes = Math.round( member__timesExpr.evaluate().get_as_a_number() );
```

```
    if (repeatTimes<0) {
      System.out.println("The program attempted to execute a repeat ... times block a
negative number of times (after rounding: "+repeatTimes+").  This is a fatal
error.\nAborting interpreter.");
      System.exit(-1);
    }

    for (long i = 0; i<repeatTimes; ++i) {
      for (int j = 0; j<member___code.size(); ++j) {
        try {
          member___code.elementAt(j).doYourThing();
        } catch (java.io.IOException e) {
          System.err.println("An I/O error occurred inside a 'repeat ... times'
block.\nAborting interpreter.");
          System.exit(-1);
        }
      }
    }
  }

  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////
  /*********************************************************
   *
   * Creates a new repeatTimesParagraph object which will
   * repeat the supplied code inTimes.
   *
   * @param code
   * @param inTimes
   *********************************************************/
  public repeatTimesParagraph(Vector<NormalParagraphOrNormalSentence> code,
UsableInExpressions timesExpr) {
    if (null==code || null==timesExpr) {
      System.out.println("At least one of the object references passed in to a
repeatTimesParagraph constructor was null.  This should never happen.\nAborting
interpreter.");
      System.exit(-1);
    }

    member___code = code;
    member__timesExpr = timesExpr;

  }
  /*********************************************************
  *
  * The default constructor is disallowed.
  *
  *********************************************************/
  private repeatTimesParagraph() {}; // Disallow the default constructor
}
```

```
// === repeatWithParagraph.java === //

// This file was written by Abe and Steve.

import java.util.Vector;

/*****************************************************************
 *
 * The repeatWith class repeats a block of code
 * with an expression.<br><br>
 *
 * @see <a href='../SLAWscript.html#Repeat_With'>Repeat With in Language Reference
Manual</a>
```

```
 *
 * @author Abe, Steve
 *
 ****************************************************************/
public class repeatWithParagraph extends repeatParagraph {

  private Vector<NormalParagraphOrNormalSentence> member___code;
  private UsableInExpressions                     member___from,
                                                  member___to,
                                                  member___step;

  private boolean                                 member___default_step_is_in_use;

  private String                                  member___counter_identifier;

  public void doYourThing() {

    final double from_dbl = member___from.evaluate().get_as_a_number() * 100000000.0;
    final double to_dbl   = member___to.evaluate().get_as_a_number()   * 100000000.0;
    if ( Double.isNaN(from_dbl) ) {
      System.err.println("A 'repeat with' block is attempting to loop starting at a NaN
(not a number).\nAborting interpreter.");
      System.exit(-1);
    }

    if ( Double.isNaN(to_dbl) ) {
       System.err.println("A 'repeat with' block is attempting to loop ending at a NaN
(not a number).\nAborting interpreter.");
       System.exit(-1);
    }
    if ( Double.isInfinite(from_dbl) ) {
      System.err.println("A 'repeat with' block is attempting to loop starting at an
infinite number.\nAborting interpreter.");
      System.exit(-1);
    }
    if ( Double.isInfinite(to_dbl) ) {
      System.err.println("A 'repeat with' block is attempting to loop ending at an
infinite number.\nAborting interpreter.");
      System.exit(-1);
    }


    final long from = Math.round(from_dbl);
    final long to   = Math.round(to_dbl);
    long step;

    if (member___default_step_is_in_use) {
      if (from<to) step = 100000000;
      else         step = -100000000; // not worrying about e.g. "else if (to<from)",
because the third case (from=to) doesn't care about "step"
    } else {
      final double step_dbl = member___step.evaluate().get_as_a_number() * 100000000.0;

      if ( Double.isNaN(step_dbl) ) {
          System.err.println("A 'repeat with' block is attempting to step with a NaN (not
a number).\nAborting interpreter.");
          System.exit(-1);
      }
      if ( Double.isInfinite(step_dbl) ) {
          System.err.println("A 'repeat with' block is attempting to step with an
infinite number.\nAborting interpreter.");
          System.exit(-1);
      }

      step = Math.round(step_dbl);
    }
```

```
    if (from<to) {

      if (step<=0) {
        System.err.println("An error occurred while starting the execution of a 'repeat
with' block:");
        System.err.println("  the 'from' expr. was less than the 'to' expr., but the
'step' expr. ("+step+") was less than or equal to zero.");
        System.err.println("Aborting interpreter.");
        System.exit(-1);
      }

      Variable counter = new Variable(from/100000000.0);
      VariableStack.put(member___counter_identifier,counter);
      for (long i = from; i<=to; i+=step) {
        counter.set_to(i/100000000.0); // this line of code is redundant the first time
through the loop, but it doesn't hurt, it just wastes a little time
        for (int j = 0; j<member___code.size(); ++j) {
          try {
            member___code.elementAt(j).doYourThing();
          } catch (java.io.IOException e) {
            System.err.println("An I/O error occurred inside a 'repeat with'
block.\nAborting interpreter.");
            System.exit(-1);
          }
        }
      }

    } else if (from>to) {

      if (step>=0) {
        System.err.println("An error occurred while starting the execution of a 'repeat
with' block:");
        System.err.println("  the 'from' expr. was greater than the 'to' expr., but the
'step' expr. was greater than or equal to zero.");
        System.err.println("Aborting interpreter.");
        System.exit(-1);
      }

      Variable counter = new Variable(from/100000000.0);
      VariableStack.put(member___counter_identifier,counter);
      for (long i = from; i>=to; i+=step) {
        counter.set_to(i/100000000.0); // this line of code is redundant the first time
through the loop, but it doesn't hurt, it just wastes a little time
        // note: the increment in the preceding loop header is _intentionally_ "i+=step",
_not_ "i-=step", because step must be less than 0
        for (int j = 0; j<member___code.size(); ++j) {
          try {
            member___code.elementAt(j).doYourThing();
          } catch (java.io.IOException e) {
            System.err.println("An I/O error occurred inside a 'repeat with ...'
block.\nAborting interpreter.");
            System.exit(-1);
          }
        }
      }

    } // intentionally no "else": e.g. from 0 to 0 is a no-op, and there's no need to
check "step" for validity in this case

  }

  // the valid constructor...
  public repeatWithParagraph(Vector<NormalParagraphOrNormalSentence> code,
UsableInExpressions from, UsableInExpressions to, UsableInExpressions step, String
counter_identifier) {
  // reminder: "step" may (validly) be null, which means "assume the default stepping"
```

```
    if (null==code || null==from || null==to || null==counter_identifier) {
      System.out.println("At least one of the object references (other than 'step')
passed in to a repeatWhileParagraph constructor was null.  This should never
happen.\nAborting interpreter.");
      System.exit(-1);
    }

    member___code               = code;
    member___from               = from;
    member___to                 = to;
    member___counter_identifier = counter_identifier;

    if (null==step) {
      member___default_step_is_in_use = true;
    } else {
      member___default_step_is_in_use = false;
      member___step = step;
    }

  }


  private repeatWithParagraph() {} // Disallow the default constructor.

}
```

```
// === returnSentence.java === //

// this file was written by Abe

public class returnSentence extends NormalParagraphOrFunctionValidSentence {

  private UsableInExpressions member___expression;

  public void doYourThing() {
    System.err.println("A return sentence had its doYourThing() called.  Although this
member must exist, it should never be used.\nAborting interpreter.");
    System.exit(-1);
  }

  public Constant getReturnValue() {
    return member___expression.evaluate();
  }

  returnSentence(UsableInExpressions anExpression) {
    if (null==anExpression) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___expression = anExpression;
  }

  private returnSentence() {} // disallow the default constructor

} // end of class
```

```
// === RoundExpr.java === //

// this file was written by Abe


public class RoundExpr implements UsableInExpressions {

  private UsableInExpressions member___operand;

  private RoundExpr() { } // disallow the default constructor

  RoundExpr(UsableInExpressions in) {
    if (null==in) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___operand = in;
  }


  public Constant evaluate() {

    return new Constant( Math.round( member___operand.evaluate().get_as_a_number() ) );

  }

} // end of class
```

```
// === setSentence.java === //

import java.io.*;

/*********************************************************
 *
 * The setSentence class models grammar of the
 * following form:<br><br>
 *
 * "set" identifier "to" expression
 *
 * <br><br>
 *
 * @author Steve
 *
 *********************************************************/
public class setSentence extends NormalParagraphOrNormalSentence {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////
  /**
   * The name of the variable to set
   */
  private String member___variable_name;

  /**
   * The expression to assign to the variable
   */
  private UsableInExpressions member___expression;
```

```
  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  /*********************************************************
   *
   * The doYourThing method is called during "runtime"
   * evaluation of the SLAWscript.  It represents actual
   * execution of the "set identifier to blah" SLAW
   *
   *********************************************************/
  public void doYourThing() throws IOException {

    // Grab the Constant from the expression
    Constant exprValue = member___expression.evaluate();

    // Is it a string?
    if ( exprValue.is_a_string() ) {
      VariableStack.put( member___variable_name,
        new Variable(exprValue.get_as_a_string()) );
    } else {
      VariableStack.put( member___variable_name,
        new Variable(exprValue.get_as_a_number()) );
    }

  }

  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////
  /*********************************************************
   * Creates a setSentence object.  Sets the variable
   * name of the "Left Hand Side", and stores the expression
   * from the "Right Hand Side (RHS)."  Note:  We don't know at
   * this point what the RHS is - we just know that it is
   * a UsableInExpression subclass that we will evaluate later
   * using its doYourThing() method...
   *
   *********************************************************/
  setSentence(String in, UsableInExpressions anExpression) {
    if (null==in || null==anExpression) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if (! Validator.identifier_is_usable_as_a_variable_name(in)) {
      System.err.println("A name ('"+in+"') that was not usable for a variable was
attempted to be used in a 'set' sentence.\nAborting interpreter.");
      System.exit(-1);
    }

    // Save the variable name
    member___variable_name = in;

    // Save the RHS expression
    member___expression = anExpression;
  }

  private setSentence() {} // disallow the default constructor


} // end of class
```

```
// === SingleQuestionMarkExpr.java === //

public class SingleQuestionMarkExpr implements UsableInExpressions {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////

  private String member___name;


  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////

  public Constant evaluate() { // this is the real reason for this class's existence

    final Variable theVar = VariableStack.get(member___name);

    if (null==theVar) {
      System.err.println("A null was received from the VariableStack while computing the
single-question-mark operator in an expression.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if ( ! theVar.is_a_string() ) { // i.e. is_a_number()
      return new Constant(2.0);
    } else if ( theVar.is_usable_as_a_number() ) {
      return new Constant(1.0);
    } else {
      return new Constant(0.0);
    }

  } // end of "evaluate"


  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////

   /**************************************************
   *
   * Disallow the default constructor.
   *
   **************************************************/
  private SingleQuestionMarkExpr() { }

  public SingleQuestionMarkExpr(String name) {
    if (null==name) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

     if (Validator.identifier_is_usable_as_a_variable_name(name)) {
       member___name = name;
     } else {
       System.err.println("An invalid variable identifier was attempted to be used before
a single question mark in an expression.");
       System.err.println("Aborting interpreter.");
       System.exit(-1);
     }
  }
}
```

```
// === SLAWmisc.java === //

/*************************************************************
 *
 * SLAWmisc is a service class that is used for a variety
 * of tasks such a string preprocessing.
 *
 * @author Steve
 *
 *************************************************************/
public class SLAWmisc {

  /*************************************************************
   *
   * Returns a 'SLAW-friendly' literal string.<br><br>
   *
   * Tasks:<br><br>
   * <ul>
   *  <li>Check for beginning and ending quotes; abort if not</li>
   *  <li>Convert \\t, \\n, \t, \n</li>
   * </ul>
   * <br><br>
   *
   * @param inString
   * @return A
   *************************************************************/
  public static String StringLiteralParser(String inText) {
    String outText = inText;

    // FUTURE: We need to address double-backslash contingencies.
    // This needs to be done in a single pass, possibly with a special function,
    // so as not to convert "\\n" and "\\t" all the way to a return and a tab, but only
to "\n" and "\t".

    // Get rid of double quotes at start and end...
    if (!((outText.charAt(0) == '"') && (outText.charAt(outText.length()-1)== '"') )) {
      System.err.println("The string literal " + inText+ " does not have beginning and
ending quotes, and cannot be processed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    } else {
      outText = outText.substring(1, outText.length()-1);
    }

    // Do some special character replacements...
    outText = outText.replace("\\n", "\n");
    outText = outText.replace("\\t", "\t");
    outText = outText.replace("\\\"", "\""); // escaped double-quote symbol
    outText = outText.replace("\\\\", "\\"); // escaped backslash symbol (NOT perfect;
                                             FUTURE: fix this!)
    // FUTURE: redo the preceding in such a way as to process "\n", "\t", and "\\"
    //         all in one pass

    return outText;
  }
}
```

```
// === SLAWscript.java === //

// this file was written by Abe and Steve

import java.io.*;
import java.util.Hashtable;

import org.antlr.runtime.*;
```

```
/**************************************************************
*
* The SLAWscript class represents the SLAWscript interpreter, and
* is the main class for the application.
* <PRE>
* Usage: (assuming slaw is a shell script executing this class)
*
*   slaw [-h | --help | file | file -s | file --showcomments]");
*
*   slaw -h : this help message.
*   slaw --help : this help message.
*
*   slaw file : attempt to parse and execute the file specified by 'file'.
*
*   slaw file --showcomments : attempt to parse the file specified by 'file'
*   and output its comments to standard-out (without attempting to execute
*   the code).  JavaDoc for us poor people.");
*
*   slaw file -s : the same as for 'SLAWscript file --showcomments'
* </PRE>
**************************************************************/
public class SLAWscript {

  /**
   * The SLAWscript lexer: created by ANTLR.   *
   */
  private static SLAWscriptLexer lexer = null;

  /**
   * The SLAWscript parser: created by ANTLR.   *
   */
  private static SLAWscriptParser parser = null;

  /**
   * The return type from the SLAWscript parser.
   */
  private static ParserReturnType parserReturn = null;

  /**
   * The procedures contained in the SLAWscript file.
   */
  public static Hashtable<String,Procedure> member___procedures;

  /**
   * The functions contained in the SLAWscript file.
   */
  public static Hashtable<String,Function>  member___functions;

  /**
   * The SLAW script file under parse.
   */
  private static File theFile;


  public static void main(String[] args) throws IOException {

    int i = 1;
    switch (args.length) {

      case 0:
        usage(); // there's no need for a "break;" here - "usage()" will exit
      case 1:
        if (args[0].equals("-h")|args[0].equals("--help"))  help();
        // "else" is unneeded here because "help()" always exits

        theFile = new File(args[0]);
        checkFileOK(args[0]);
```

```
        // here is where the parser and interpreter should be invoked, if non-empty
        String fileName = args[0];

        CharStream cs = new ANTLRFileStream(fileName);
        try {
          lexer = new SLAWscriptLexer(cs);
          CommonTokenStream tokens = new CommonTokenStream(lexer);
          parser = new SLAWscriptParser(tokens);
        } catch (Exception e) {
          System.err.print("Exception in SLAW interpreter while invoking lexer and
parser: ");
          System.err.println(e.toString());
          e.printStackTrace();
          System.err.println("Aborting interpreter.");
          System.exit(-1);
        }

        // if we made it here, then we must have a good parser
        try {
          parserReturn = parser.startRule();
        } catch (Exception e) {
          System.err.print("Exception in SLAW interpreter while invoking start rule: ");
          System.err.println(e.toString());
          e.printStackTrace();
          System.err.println("Aborting interpreter.");
          System.exit(-1);
        }
        VariableStack.new_context(); // initialize the VariableStack

        // Pull up the subroutines from the parser into the execution context...
        member___procedures = parserReturn.member___procedures;
        member___functions  = parserReturn.member___functions;

        // Get the main body...
        NormalParagraphOrNormalSentence[] mainBody = parserReturn.member___mainBody;

        // Traverse the main body...
        for (int j=0; j < mainBody.length; ++j ) {
          if (null != mainBody[j])  mainBody[j].doYourThing();
          // "if" added by Abe (to avoid NullPointerException)
        }

        break;
      case 2:
        if ( !( args[1].equals("-s") | args[1].equals("--showcomments") ) )  usage();
        theFile = new File(args[0]);
        checkFileOK(args[0]);
        show_comments();
        // there's no need for a "break;" here - "show_comments()" will exit
      default:
        usage();
    }

  } // end of "main"

  static void usage() {
    System.err.println("usage: slaw [-h | --help | file | file -s | file --
showcomments]");
    System.exit(-1);
  } // end of "usage"
```

```
  static void help() {
    System.err.println("slaw -h : this help message.");
    System.err.println("slaw --help : this help message.");
    System.err.println("slaw file : attempt to parse and execute the file specified by
'file'.");
    System.err.println("slaw file --showcomments : attempt to parse the file specified by
'file' and output its comments to standard-out (without attempting to execute the code).
JavaDoc for us poor people.");
    System.err.println("slaw file -s : the same as for 'SLAWscript file --
showcomments'.");
    System.exit(-1);
  } // end of "help"

  static void checkFileOK(String file) {
    if (! theFile.exists()) {
      System.err.println("The file '"+file+"' does not seem to exist.");
      usage(); // in case the user entered something silly like "-q" hoping it's an option
    } else if (! theFile.isFile()) {
      System.err.println("The file '"+file+"' does not seem to be a normal file.");
      usage(); // in case the user entered something silly like "/var" hoping it's an
option
    } else if (theFile.length()<1) { // not just "==0" because longs are signed
      System.err.println("The file '"+file+"' seems to be an empty file.");
      System.exit(-1);
    } else if (! theFile.canRead()) {
      System.err.println("The file '"+file+"' cannot be read by this process.");
      System.err.println("Please check the file's permissions and try again.");
      System.exit(-1);
    }
  } // end of "checkFileOK"

  static void show_comments() throws IOException {

    BufferedReader theReader = new BufferedReader(new FileReader(theFile));

    String theLine;
    while (theReader.ready()) {

      theLine = theReader.readLine();
      // System.out.println(theLine); // test code

      if (theLine.trim().length()>0 && '#'==theLine.trim().charAt(0)) {
        // for comment-only lines
        System.out.println(theLine.substring(theLine.indexOf('#')));
      } else {

        // The non-entire-line comment is a little trickier to parse; we don't want to
        // mistakenly count a '#' character in the middle of a literal string as being a
        // comment starter!

        boolean in_a_string = false;
        while (theLine.length()>0) {
          // intentionally while and not do...while, for empty lines
          if ((!in_a_string) && '#'==theLine.charAt(0)) {
            System.out.println(theLine);
            break; // not the greatest of coding style, I admit
          }

          if ('"'==theLine.charAt(0))  in_a_string = ! in_a_string;
            // invert the condition

     // if the current starting character is a '\', and we are inside a string, then "eat"
     // the '\' as well as (by default) the character right after it; this is important so
     // we don't incorrectly exit literal string mode upon hitting a \" sequence
          if (in_a_string && '\\'==theLine.charAt(0))  theLine = theLine.substring(1);

          theLine = theLine.substring(1); // "eat" the starting character
        }
```

```
      }
    }

    System.exit(0); // intentionally zero exit code; if this executes, it's a sign that
"showcomments" completed normally.
  } // end of "show_comments"

} // end of class
```

```
// === stopSentence.java === //

// this file was written by Abe

public class stopSentence extends NormalParagraphOrNormalSentence {

  private int member___line;

  private stopSentence() { // disallow the default constructor
  }

  stopSentence(int in) { // expected input: line number of the "stop" sentence
    member___line = in;
  }

  public void doYourThing() { // the assumption is that the subroutine's context is
already set up
    System.err.println("Program execution was stopped by the 'stop' sentence on line
"+member___line+".");
    System.exit(-1);
  }

}
```

```
// === StrictlyDoesNotEqualExpr.java === //

// this file was written by Abe


public class StrictlyDoesNotEqualExpr implements UsableInExpressions {

  private UsableInExpressions member___left, member___right;

  private StrictlyDoesNotEqualExpr() { } // disallow the default constructor

  StrictlyDoesNotEqualExpr(UsableInExpressions left, UsableInExpressions right) {
    if (null==left || null==right) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___left  = left;
    member___right = right;
  }


  public Constant evaluate() {

    final Constant left  = member___left.evaluate();
    final Constant right = member___right.evaluate();
```

```
    if ( left.is_a_string() ) {
      if ( right.is_a_string() ) {
        return new Constant( ! left.get_as_a_string().equals(right.get_as_a_string()) );
      } else {
        return new Constant(true);
      }
    } else if ( right.is_a_string() ) {
      return new Constant(true);
    } else {
      return new Constant( left.get_as_a_number() != right.get_as_a_number() );
    }

  }

} // end of class
```

```
// === StrictlyEqualsExpr.java === //

// this file was written by Abe


public class StrictlyEqualsExpr implements UsableInExpressions {

  private UsableInExpressions member___left, member___right;

  private StrictlyEqualsExpr() { } // disallow the default constructor

  StrictlyEqualsExpr(UsableInExpressions left, UsableInExpressions right) {
    if (null==left || null==right) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___left  = left;
    member___right = right;
  }


  public Constant evaluate() {

    final Constant left  = member___left.evaluate();
    final Constant right = member___right.evaluate();

    if ( left.is_a_string() ) {
      if ( right.is_a_string() ) {
        return new Constant( left.get_as_a_string().equals(right.get_as_a_string()) );
      } else {
        return new Constant(false);
      }
    } else if ( right.is_a_string() ) {
      return new Constant(false);
    } else {
      return new Constant( left.get_as_a_number() == right.get_as_a_number() );
    }

  }

} // end of class
```

```
// === SubstrExpr.java === //

// this file was written by Abe

// this class implements the '@' operator, which returns a substring of the original
string

public class SubstrExpr implements UsableInExpressions {

  private UsableInExpressions member___original, member___position, member___limit; //
the limit is optional; null -> [no limit]

  private SubstrExpr() { } // disallow the default constructor

  SubstrExpr(UsableInExpressions original, UsableInExpressions position,
UsableInExpressions limit) {
    if (null==original || null==position) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___original  = original;
    member___position  = position;
    member___limit     = limit;
  }


  public Constant evaluate() {

    final String original = member___original.evaluate().get_as_a_string();
    final int position = (int) Math.round( member___position.evaluate().get_as_a_number()
);

    if (position<1) { // TO DO: document that this errors out
      System.err.println("An '@' expression requested a starting position [after
rounding] that was less than one.");
      System.err.println("Reminder: in SLAWscript, string character indices start at
one.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    if ( (position)>original.length() ) {
      System.err.println("Warning: An '@' expression requested a starting position [after
rounding: "+position+"] that was too big for the string ['"+original+"'].  Returning an
empty string as per the SLAWscript LRM.");
      return new Constant("");
    }

    if (null==member___limit) { // no limit

      return new Constant( original.substring(position-1) );

    } else { // there _is_ a limit

      final int limit = (int) Math.round( member___limit.evaluate().get_as_a_number() );

      if (limit<1) {
        if (limit<0) { // only warn if the limit is negative; if it's zero, that's OK:
the program "wants" an empty string
          System.err.println("Warning: An '@' expression requested a negative length
limit [after rounding: "+limit+"].  Returning an empty string as per the SLAWscript
LRM.");
        }
        return new Constant("");
      }
```

```
      if ( (position-1+limit)>original.length() ) {
        return new Constant( original.substring(position-1) );
      } else {
        return new Constant( original.substring(position-1, position-1+limit) );
      }
    }

  }

} // end of class
```

```
// === UsableInExpressions.java === //

// this file was written by Abe

public interface UsableInExpressions {

  public Constant evaluate();

}
```

```
// === Validator.java === //

// this file was written by Abe and Steve

import java.util.Vector;

public class Validator {
  // note: due to the use of a binary search (below), the following list _must_ be
correctly sorted
  final static String[] member___reserved_words = {
"and","assert","copy","define","do","e","else","end","escape","false","from","function","
get","if","ignore","is","localize","not","or","pi","procedure","put","randomize","repeat",
"return","true","set","stderr","stdout","step","stop","times","to","while","with" };
  // FUTURE: run a once-at-startup sort on the preceding list in case a human goofs up
the order
  //        this is needed due to the use of a binary sort later on

  private static Vector member___used_words = new
Vector(java.util.Arrays.asList(member___reserved_words));

  public static void main(String[] args) { // this is here for testing
    System.out.println("Is 'if' reserved?: "+member___used_words.contains("if"));
    System.out.println("Is 'foo' reserved?: "+member___used_words.contains("foo"));
  }

  public static void add_subroutine_name(String word) {
    validate_identifier(word);
    word = word.toLowerCase(); // convert to canonical form
    if (member___used_words.contains(word)) {
      System.err.println("Error: an attempt was made at redefining the word
'"+word+"'.\nAborting interpreter.");
      System.exit(-1);
    }
    member___used_words.add(word);
  }
```

```
  public static boolean identifier_is_usable_as_a_variable_name(String word) {
    validate_identifier(word);
    word = word.toLowerCase(); // convert to canonical form
    return ! member___used_words.contains(word); // return the opposite of the already-
exists status
  }

  public static boolean identifier_is_usable_as_a_subroutine_name(String word) {
    validate_identifier(word);
    word = word.toLowerCase(); // convert to canonical form

    // return the opposite of the reserved status
    return (java.util.Arrays.binarySearch(member___reserved_words, word) < 0);
  }

  private static void validate_identifier(String name) { // private on purpose, since it
doesn't really perform a full validation (i.e. it accepts "if"), only a character-stream
validation
    if (null==name) {
      System.err.println("Implementation error: a null string reference was attempted to
be used as a variable or subroutine name.\nAborting interpreter.");
      System.exit(-1);
    }

    if (name.length()<1) {
      System.err.println("Implementation error: an empty string was attempted to be used
as a variable or subroutine name.\nAborting interpreter.");
      System.exit(-1);
    }

    final char first = name.charAt(0);
    if (! ( (first>='A' && first<='Z') || (first>='a' && first<='z') ) ) {
      System.err.println("Implementation error: a non-letter (ASCII) was attempted to be
used as the first character of a variable or subroutine name.\nAborting interpreter.");
      System.exit(-1);
    }

    char c;
    for (int count=1; count< name.length(); ++count) {
      c = name.charAt(count);

      if (! ( (c>='A' && c<='Z') || (c>='a' && c<='z') || ('_'==c) || (c>='0' &&
c<='9') ) ) {
        System.err.println("Implementation error: an invalid character was attempted to
be used as a non-first character in a variable or subroutine name.\nAborting
interpreter.");
        System.exit(-1);
      }

    }

  } // end of "validate_identifier"

  public static void validateDouble(double in) {
    if ( Double.isNaN(in) ) {
      System.err.println("Implementation error:  A numeric value is attempting to be set
to a NaN (not a number).\nAborting interpreter.");
      System.exit(-1);
    }
    if ( Double.isInfinite(in) )  {
      System.err.println("Implementation error:  A numeric value is attempting to be set
to an infinity.\nAborting interpreter.");
      System.exit(-1);
    }
  }

} // end of class
```

```
// === Variable.java === //

/*******************************************************
 *
 * The Variable class is used to model a "runtime" variable
 * in SLAW.
 * <br><br>
 *
 * It's important not to confuse Variable with VariableName.
 * <br><br>
 * VariableName is used to capture the concept of the
 * variable identifer in SLAW - e.g. cat, i, dog, myVar, etc.
 * <br><br>
 * The Variable class is used internally to track the
 * possibly-evolving value of this VariableName.
 *
 * <br><br>
 *
 * @author Abe
 *
 *******************************************************/
//
public class Variable {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////

  /**
   * The variables numeric value (if applicable)
   */
  private double num;

  /**
   * The variables string value (if applicable)
   */
  private String str;

  /**
  * An boolean to indicate that the variable is a string value
  * (default assumed numeric)
  * This is importantant, as we might have a numeric string, so we
  * can't rely on the type alone..
  */
  private boolean is_a_string;

  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  public Constant convert_to_Constant() { // this is here so VariableName can implement
UsableInExpressions
    if (is_a_string) {
      return new Constant(str);
    } else {
      return new Constant(num);
    }
  }

  /*****************************************************************
   *
   * 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() {
    if (is_a_string) {
      try {
        return Double.parseDouble(str);
      } catch (NumberFormatException nfe) {
        System.err.println("An error occurred while attempting to convert the string
'"+str+"' to a number.  Exception output follows...");
        System.err.println(nfe);
        System.err.println("Aborting interpreter.");
        System.exit(-1);
      }
    }
    return num; // This is intentionally not inside an "else" to the above "if",
                // both because otherwise "javac" complains that there's a "return"
                // missing here, and also because it doesn't need to be inside an "else";
                // the "if" part either returns or exits.
  }

  /*****************************************************************
   *
   * Return the Variable as a string.
   *
   * @return The Variable as a string.
   *
   *****************************************************************/
  public String get_as_a_string() {
      if (is_a_string) {
        return str;
      } else { // this will cause the number to be converted to a string
        // this was: return ""+num;

        // the following more-complicated version is so as to produce e.g. "42", not
"42.0"
        String temp = ""+num;
        if ( temp.substring( temp.length()-2, temp.length() ).equals(".0") ) {
          temp = temp.substring(0, temp.length()-2);
        }
        return temp;
      }
    }

  /*****************************************************************
   *
   * Check if the Variable is a numeric string.
   *
   * @return true if the Variable is a numeric string.
   *
   *****************************************************************/
  public boolean is_a_numeric_string() { // this returns true _only_ for numeric _strings_
    if (is_a_string) {
      try {
        Double.parseDouble(str); // intentionally ignoring the result
        return true;
      } catch (NumberFormatException nfe) {
        return false;
      }
    } else { // the following is for "honest-to-goodness" numbers
      return false;
    }
  }


  /*****************************************************************
   *
   * Sets the Variable to the supplied double.
   *
   * @param the double value to assign to the Variable.
   *
```

```
   ************************************************************/
  public void set_to(double in) {
    is_a_string=false;
    num=in;
  }

  /*****************************************************************
   *
   * Sets the Variable to the supplied string.
   *
   * @param the string value to assign to the Variable.
   *
   *****************************************************************/
  public void set_to(String in) {
    is_a_string=true;
    str=in;
  }

  /*****************************************************************
   *
   * Check if the Variable is a string.
   *
   * @return true if the Variable is a string.
   *
   *****************************************************************/
  public boolean is_a_string() {
    return is_a_string;
  }

  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////
  private Variable() { } // disallow the default constructor

  /*****************************************************************
  *
  * Create a Variable from the supplied double.
  * Set is_a_string flag to false.
  *
  *****************************************************************/
  Variable(double in) {
    is_a_string=false;
    num=in;
  }

  /*****************************************************************
  *
  * Create a Variable from the supplied string.
  * Set is_a_string flag to true.
  *
  *****************************************************************/
  Variable(String in) {
    if (null==in) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    is_a_string=true;
    str=in;
  }

  /*************************************************
  *
  * Create a Variable from the supplied boolean.
  *
  *************************************************/
```

```
  Variable(boolean in) {
    is_a_string = false;
    num = (in ? 1.0 : 0.0);
  }

  // === end of constructors section ===

  /**************************************************
  *
  * return true if the datum is usable as a number,
  *        false if it is e.g. "Hello".
  *
  **************************************************/
  public boolean is_usable_as_a_number() {
    return (! is_a_string) || is_a_numeric_string();
  }

} // end of class
```

```
// === VariableStack.java === //

// this file was written by Abe

import java.util.HashMap;
import java.util.Vector;
/**********************************************************************
 *
 * The Variable Stack class is used by the SLAWscript
 * to manage all variables (declaration, scope).
 * <br><br>
 * Note - this class is called "VariableStack" on purpose,
 * even though it uses a Vector internally.
 * <br><br>
 *
 * <b>Documentation for the implementation team:</b>  According to the
 * way I (Abe) coded this, there should be two ways to use this class:
 * <br><br>
 * <ul>
 *    <li>
 *      Either instantiate an object of this class (e.g. <code>
 *      "VariableStack theVariableStack = new VariableStack();"</code>) , or
 *    </li>
 *    <li>
 *      Just initialize it manually:<br><br>
 *      <code>("VariableStack.new_context();" once)</code><br><br>
 *      when the interpreter starts up, and from then on use it directly
 *      (e.g. <code>''VariableStack.put("foo",new Variable(9));").</code><br><br>
 *    </li>
 *   </ul>
 *   The two should be equivalent, even if you mix them in the
 *   same program, since the data and methods are all static.
 *
 **********************************************************************/
public class VariableStack { // at least most of this should be static - one per program
only!

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////
  private static Vector< HashMap<String,Variable> > the_stack = new Vector<
HashMap<String,Variable> >();
```

```
  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  /*****************************************************************
   *
   * This is for subroutines with parameters and "localize"
   *
   *****************************************************************/
  public static void new_context() {
    the_stack.add(new HashMap<String,Variable>());
  }

  /*****************************************************************
   *
   * This is for when a subroutine with either parameters or "localize" or both ends
   *
   *****************************************************************/
  public static void previous_context() {
    if (the_stack.size()<2) {
      System.err.println("Error in implementation: attempt to destroy the global variable
context.\nAborting interpreter.");
      System.exit(-1);
    }

    // the following code block both destroys the last context and checks if it was null
    if ( null == the_stack.remove(the_stack.size()-1) ) { // size()-1 so as to get a 0-
based index
      System.err.println("Warning: a null variable context was destroyed.  Continuing
program.");
    }
  }
  /*****************************************************************
   *
   * This to help starting and ending subroutines
   *
   *****************************************************************/
  public static int current_context_number() {
    return the_stack.size()-1;
  }

  /*****************************************************************
   *
   * Put a variable on the stack.
   *
   * IMPORTANT: we have to make sure the variable gets created or overwritten in the
"newest" possible context where a variable with the same name exists (if any), or global
context (if none)
   * Reminder: context #0 is the global context
   *
   *****************************************************************/
  public static void put(String name, Variable var) {
    // the following used to be: Variable.validate_name(name);
    if (! Validator.identifier_is_usable_as_a_variable_name(name)) {
      System.err.println("A name ('"+name+"') that was not usable for a variable was
attempted to be 'put' to the VariableStack.\nAborting interpreter.");
      System.exit(-1);
    }

    name = name.toUpperCase();



    // IMPORTANT: we have to make sure the variable gets created or overwritten in the
    // "newest" possible context where a variable with the same name exists (if any),
    // or global context (if none)

    // reminder: context #0 is the global context
```

```
    int context;

    for (context=the_stack.size()-1; context>0; --context) {
      if ( the_stack.get(context).containsKey(name) )  break; // quick and dirty
    }

    // by now, if I've done it right, then "context" should be between 0 and size()-1,
    // inclusive and should contain either the context of the pre-existing variable, or 0
    // if it's a new one

    the_stack.get(context).put(name,var); // "get" here gets the last HashMap
  }

  public static void put_at_top(String name, Variable var) {
    // this is for the parameters of subroutines
    if (! Validator.identifier_is_usable_as_a_variable_name(name)) {
      System.err.println("A name ('"+name+"') that was not usable for a variable was
attempted to be 'put' to the VariableStack.\nAborting interpreter.");
      System.exit(-1);
    }

    name = name.toUpperCase();

    the_stack.get(the_stack.size()-1).put(name,var);
  }

  /*****************************************************************

  /*****************************************************************
   *
   *  This is for "localize", which creates a new context but does
   *  not set the value of the variable
   *
   *****************************************************************/
  public static void reserve(String name) {

    // the following used to be: Variable.validate_name(name);
    if (! Validator.identifier_is_usable_as_a_variable_name(name)) {
      System.err.println("A name ('"+name+"') that was not usable for a variable was
attempted to be 'reserve'd in the VariableStack.\nAborting interpreter.");
      System.exit(-1);
    }

    name = name.toUpperCase();

    // System.err.println("TEST POINT #1: x="+get("x")); // DEBUG CODE
   // System.err.println("TEST POINT #1: fubar="+get("fubar")); // DEBUG CODE
    // System.err.println("TEST POINT #1: fubar="+the_stack.get(the_stack.size()-
1).get("fubar")); // DEBUG CODE
    // System.err.println("TEST POINT #1: fubar containsKey:
"+the_stack.get(the_stack.size()-1).containsKey("fubar")); // DEBUG CODE
    if ( ! the_stack.get(the_stack.size()-1).containsKey(name) ) {    // make sure we
don't overwrite the variable with null if it already exists in the current context
      the_stack.get(the_stack.size()-1).put(name,null); // this variable name is only
reserved,    i.e. it has no value yet
      // System.err.println("TEST POINT #2: var. name="+name); // DEBUG CODE
    } // this "if" should prevent bugs due to e.g. "localize a" followed by "localize a"
or
      // (formal parameter 'x') followed by (in the same subroutine) "localize x" in the
program
  }
```

```
  /*****************************************************************
   *
   * Gets the variable with supplied name of the stack.
   *
   *****************************************************************/
  public static Variable get(String name) {
    if ( ! Validator.identifier_is_usable_as_a_variable_name(name) ) {
      System.err.println("A name ('"+name+"') that was not usable for a variable was
attempted to be retrieved from the VariableStack using
'VariableStack.get(String)'.\nAborting interpreter.");
      System.exit(-1);
    }

    // IMPORTANT: we have to make sure the variable gets read from the "newest" possible
context where a variable with the same name exists (if any), including the possibility of
the global context (if that's the only context where a variable with this name exists)
    // reminder: context #0 is the global context

    name = name.toUpperCase();

    int context;
    for (context=the_stack.size()-1; context>0; --context) {
      if ( the_stack.get(context).containsKey(name) )  break; // quick and dirty
    }
    // by now, if I've done it right, then "context" should be between 0 and size()-1,
inclusive
    // and should contain either the context of the pre-existing variable, or 0 if it's a
new one

    Variable temp = the_stack.get(context).get(name);
    if (null == temp) {
      System.err.println("Implementation error: an unset variable ('"+name+"') was
attempted to be read.\nAborting interpreter.");
      System.exit(-1);
    }
    return temp;
  }

  // The following is for the (future) "??" operator, which is allowed to come after the
name of an undefined variable (or of a subroutine) w/o program abort.
  public static Variable get_if_it_exists(String name) {
    if ( ! Validator.identifier_is_usable_as_a_variable_name(name) ) {
      System.err.println("A name ('"+name+"') that was not usable for a variable was
attempted to be retrieved from the VariableStack using
'VariableStack.get_if_it_exists(String)'.\nAborting interpreter.");
      System.exit(-1);
    }

    name = name.toUpperCase();

    System.err.println("DEBUG POINT #4: '"+name+"'");

    int context;
    for (context=the_stack.size()-1; context>0; --context) {
      if ( the_stack.get(context).containsKey(name) ) {
        final Variable temp = the_stack.get(context).get(name);
        System.err.println("DEBUG POINT #5: '"+temp+"'");
        return temp;
      }
    }

    return null; // this serves as a "not found" indicator
  }
```

```
  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////

  /*************************************************************
   * The only constructor - creates the Variable Stack
   *
   *************************************************************/
  VariableStack() {
    // the_stack = new Vector();
    // I'm pretty sure we don't need this if we already did it, above
    the_stack.add(new HashMap<String,Variable>());
    // there must be at least one variable context at all times
  }


} // end of class
```

```
// === whileParagraph.java === //

import java.util.Vector;

/*****************************************************************
*
* The whileParagraph implements a loop 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.
*
* @see <a href='../SLAWscript.html#Procedures'>Procedures Defined in Language Reference
Manual</a>
*
* @author Abe and Steve
*
*****************************************************************/
public class whileParagraph extends NormalParagraphOrNormalSentence {

  ////////////////////////////////////////
  // ATTRIBUTES
  ////////////////////////////////////////
  /**
   * The expression following the "while" keyword.
   * Note:  Boolean context
   */
  private UsableInExpressions member___expr;

  /**
   * The code to execute inside the while block.
   */
  private Vector<NormalParagraphOrNormalSentence> member___code;

  ////////////////////////////////////////
  // METHODS
  ////////////////////////////////////////
  /*************************************************************
   *
   * The doYourThing() method represents the code to be executed
   * when the whileParagraph is executed at runtime.<br><br>
   *
   * <pre>
   * Intended strategy: evaluate the expr., see if it (as a number) is not 0.0; if not,
   *                    then iterate over "code", calling "doYourThing()" on each element,
   *                    then repeat from the beginning (i.e. evaluate the expr., ...)
   * </pre>
   *
```

```
   ************************************************************/
  public void doYourThing() {
    while (member___expr.evaluate().get_as_a_number() != 0.0) {
      for (int j = 0; j<member___code.size(); ++j) {
        try {
          member___code.elementAt(j).doYourThing();
        } catch (java.io.IOException e) {
          System.err.println("An I/O error occurred inside a 'while' block.\nAborting
interpreter.");
          System.exit(-1);
        }
      }
    }
  }

  ////////////////////////////////////////
  // CONSTRUCTORS
  ////////////////////////////////////////
  /************************************************************
   *
   * Disallow the default constructor.
   *
   ************************************************************/
  private whileParagraph() { } // disallow the default constructor

  /************************************************************
  *
  * Create a new whileParagraph with the supplied expression
  * on the right hand side of "while" and the code block.
  *
  * @param expr The expression on the right hand side of while
  * @param code A vector containing the NormalParagraphOrNormalSentences make up the code
  *
  ************************************************************/
  whileParagraph(UsableInExpressions expr, Vector<NormalParagraphOrNormalSentence> code) {
    if (null==expr || null==code) {
      System.err.println("A fatal condition occurred during parsing: an object passed in
to a constructor was null where null is not allowed.");
      System.err.println("Aborting interpreter.");
      System.exit(-1);
    }

    member___expr = expr;
    member___code = code;
  }
}
```

8.3 SLAWscript Test Code

```
### === chaining.SLAW === ###

put "Expecting   6: "+(1+2+3)+"\n" to stdout # the "( )" around 1+2+3 are needed in order
to get mathematical '+'
put "Expecting 123: "+1+2+3+"\n" to stdout   # otherwise the string on the left
dominates, and you get string '+'
put "Expecting  -4: "+1-2-3+"\n" to stdout
put "Expecting  24: "+2*3*4+"\n" to stdout
put "Expecting an approximation of one-sixth: "+2/3/4+"\n" to stdout
put "Expecting 'Hi Hi Hi Hi Hi Hi ': '"+"Hi "*2*3+"'\n" to stdout
# put "Expecting an approximation of 2 to the power of 81: "+2^3^4+"\n" to stdout
put "Expecting 25.62890625: "+1.5^2^3+"\n" to stdout
```

```
### === empty_function.SLAW === ###

define function empty # this is silly, and invalid due to the lack of a return
end function

ignore empty
```

```
### === empty_procedure.SLAW === ###

define procedure empty # this is silly, but valid
end procedure

do empty # this is also silly, but valid
```

```
### === flexible.SLAW === ###

define function weird
  if false
    return a_nonexistent_variable_or_the_result_of_evaluating_a_parameterless_function
  else if false
    return true
  else
    localize r
    randomize r
    return r
  end if
end function

do stupid
ignore weird

define procedure stupid
  if true
    localize x
    set x to Pi
    put x+"\n" to stdout
  end if
end procedure
```

```
### === GCD.SLAW === ###

# This file was written by Abe, Levi, and Wei

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 computed value for the GCD of "+a+" and "+b+" is "+GCD[a,b]+".\n" to stdout
end procedure
```

```
define function GCD[a,b]

  # We don't have modulus in SLAWscript, so we are using the slow (recursive) version of
the GCD algorithm.

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

end function
```

```
### === HelloWorld.SLAW === ###

# put 42 to stdout
put "Hello World\n" to stdout
# put "\tGood Bye World!" to stdout
```

```
### === numbers.SLAW === ###

put 0.1 to stdout
put "\n" to stdout
put -0.1 to stdout
put "\n" to stdout


put 0 to stdout
put "\n" to stdout
put 1 to stdout
put "\n" to stdout
put -1 to stdout
put "\n" to stdout

# put false to stdout
# put true to stdout
# put e to stdout
# put pi to stdout
```

```
### === number_guessing_game.SLAW === ###

# Written by Levi

set stillPlaying to true

while stillPlaying

  # Grab a random number
  randomize numRandom
  set numRandom to ~(20*numRandom)

  set stillGoing to true

  # Loop until correct guess
  while stillGoing
```

```
    # Prompt user for guess
    put "Guess a number between 0 and 20\n" to stdout
    get numGuess

    # Test if higher, lower, or finished
    if numGuess = numRandom
       put escape+"[32m"+"YOU GUESSED CORRECT!\n" to stdout
       set stillGoing to false
    else if numGuess > numRandom
       # put "Too high\n" to stdout
       put escape+"[2;31m"+"Too high\n" to stdout
    else
       put escape+"[2;31m"+"Too low\n" to stdout
    end if

    put escape+"[0m" to stdout # restore defaults
  end while

  put "Play again? (Y/N)\n" to stdout
  get response

  if (response = "n") or (response = "N")
    set stillPlaying to 0
  end if

end while
```

```
### === OneOfEverything.SLAW === ###

# This file was written by Levi and Wei

put "We're about to test everything, here we go...\n" to stdout

set five to addOne[4]
put "We expect to see a 5 here: " + five to stdout

# run testRandomize procedure
do testRandomize

# test square function with numeric
set fiveSquaredNumber to square[5]
put "We expect square[5] to produce 25: " + fiveSquaredNumber to stdout

# test square function with numeric string
set fiveSquaredNumericString to square["5"]
put "We expect square[\"5\"] to produce 25: " + fiveSquaredNumericString to stdout

# test square function with non-numeric string
set badSquare to square["5a"]  # this should print an error

# test true and false keywords with e and pi
do testTrueAndFalse

# test the ignore keyword (should only see side-effect print statement)
ignore square[10]

# test the assert keyword (1 assert passes, 1 assert fails)
do testAssert
```

```
##############################################################################
#  SUBROUTINE DEFINITIONS
##############################################################################

define function addOne[x]
  return x+1
end function

define procedure testTrueAndFalse
  set trueFlag to true
  set falseFlag to false
  if trueFlag
    put "true keyword works fine!  Give it a treat: e=" + e +"\n" to stdout
  else
    put "true keyword does NOT work!!!" to stdout
  end if

  if falseFlag
    put "false keyword does NOT work!!! " + pi to stdout
  else
    put "false keyword works fine!  Give it a treat: pi=" + pi +"\n" to stdout
  end if
end procedure

define procedure testRandomize
  randomize r
  set rScaled to r*10    # Scale random r to be [0, 10)

  # Loop until random r is >5
  while rScaled < 5
    randomize r
    set rScaled to r*10
  end while

  put "Done testing randomize since rScaled=" + rScaled + " is > 5.\n" to stdout
end procedure

define function square[x]
  if x?>0      # Checks type of x to make sure x is a numeric string or a number
    put "Testing the 'square' function...\n" to stdout
    return (0+x)*x
  else
    put "Error in square: '"+x+"' is not a number.\n" to stderr
  end if
end function

define procedure testAssert
  set x to "5"
  assert x is "5"
  assert x is "4"
end procedure
```

```
### === power_NaN_test.SLAW === ###

put square_root[16]+"\n" to stdout # this should print 4
put square_root[64]+"\n" to stdout # this should print 8
put "Expecting an error due to asking for the square root of (-1)...\n" to stderr
put square_root[-1]+"\n" to stdout # this should fail with a nice error message

define function square_root[in]
  return in^0.5  # math reminder: X to the power of one-half is the square-root of X
end function
```

```
### === regression.SLAW === ###

# testing "put"...
put "Hello World\n" to stdout
put 42 to stdout
put "\n" to stdout

# testing '+'...
put 40+2 to stdout
put "\n" to stdout
put 42+"\n" to stdout
put "The answer is: "+42+".\n" to stdout
put "4"+"2"+"\n" to stdout
put 40+"2"+"\n" to stdout

# testing '-'...
put 44-2+"\n" to stdout
put 44-"2"+"\n" to stdout
put "44"-2+"\n" to stdout
put "44"-"2"+"\n" to stdout
put -42+"\n" to stdout
put -(-42)+"\n" to stdout

# testing ':'...
put "Hello":"ello"+"\n" to stdout # expect 2
put "ello":"Hello"+"\n" to stdout # expect 0
put "Hello":""+"\n" to stdout # expect -1
put "":"Hello"+"\n" to stdout # expect 0
put "":""+"\n" to stdout # expect 1
```

```
### === subroutines.SLAW === ###

define procedure say_hello
  put "hello\n" to stdout
end procedure

define procedure say_something[x]
  put x+"\n" to stdout
end procedure

put "Testing zero-parameters procedure...\n" to stdout
do say_hello

put "Testing one-parameter procedure...\n" to stdout
do say_something["Hello World."]

put "Testing zero-parameters function...\n" to stdout
put 10*random_number+"\n" to stdout

put "Testing one-parameter function......\n" to stdout
put scaled_random_number[100]+"\n" to stdout

define function random_number
  localize r
  randomize r
  return r
end function

define function scaled_random_number[scale]
  localize r
  randomize r
  return r*scale
end function
```

```
### === substring.SLAW === ###

set test_string to "Hello; I love you; won't you tell me your name?\n"

put "Complete string (via '@1'):     " to stdout
put test_string@1 to stdout

put "Partial string  (via '@8'):     " to stdout
put test_string@8 to stdout

put "Partial string  (via '@1;5'):   " to stdout
put test_string@1;5 to stdout
put "\n" to stdout

put "Partial string  (via '@15;17'): " to stdout
put test_string@15;17 to stdout
put "\n" to stdout
```

```
### === test.SLAW === ###

set nine to 9
set a to -9
set a to (-9)
set a to -(9)
set b to not -9
set c to 1+2+3+4+5+6+7+8+9
set d to 1-2-3-4-5-6-7-8-9
set p to 1+2-(3+4-5)/|6-7+8-9|

if a or (b and c)
  set y to x+3/9 # this should fail - 'x' is not set yet
end if

set q to 9!
set w to e
set t to pi
set y to (1+a)!
set p to |a-b|!

set a to pi*9
set b to pi/a
set c to d/e
set f to 1+2/3-4/5+1
set g to "Hi! "*3
set h to 3*"Bye! "

set a to 1-2+3/4*5-pi+e
set b to 5*4/3
set c to 1/2*3
set d to 2/3*4
set q to 1-1/3+2
set n to 3^2-4+5
```

```
### === test_absolute_value.SLAW === ###

# expect printed values: 2, 3, 0

set a to -2
set b to |a|
put b to stdout
```

```
set a to 3
set b to |a|
put b to stdout

set a to 0
set b to |a|
put b to stdout
```

```
### === test_addition.SLAW === ###

# expect 14 to be printed out twice

set a to 9 + 5
set b to 9+5
put a to stdout
put b to stdout
```

```
### === test_and.SLAW === ###

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

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

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

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

```
### === test_assert.SLAW === ###

set a to 9
assert a is 9 # this should succeed
put "The first assertion succeeded.\n" to stdout
assert a is "9" # this should fail
```

```
### === test_constants.SLAW === ###

# this file was written by Abe

if true == 1
  put "Success." to stdout
else
  put "Failure." to stdout
end if

if false == 0
  put "Success." to stdout
else
  put "Failure." to stdout
end if

if e == 2.7182818284590451
  put "Success." to stdout
else
  put "Failure." to stdout
end if

if pi == 3.1415926535897931
  put "Success." to stdout
else
  put "Failure." to stdout
end if

if Pi == 3.1415926535897931
  put "Success." to stdout
else
  put "Failure." to stdout
end if

if pI == 3.1415926535897931
  put "Success." to stdout
else
  put "Failure." to stdout
end if

if PI == 3.1415926535897931
  put "Success." to stdout
else
  put "Failure." to stdout
end if
```

```
### === test_copy.SLAW === ###

set a to 9
copy a to b
put b+"\n" to stdout    # expect 9 (numeric) to print to stdout

set nine to "9"
copy nine to c
put c+"\n" to stdout  # expect "9" (string) to print to stdout
```

```
### === test_division.SLAW === ###

# expect 9 and 3 to be printed to stdout

set nine to 45 / 5
set three to 6/2
put nine+"\n" to stdout
put three+"\n" to stdout
```

```
### === test_division_by_zero.SLAW === ###

put 9/0 to stdout
```

```
### === test_empty_string_output.SLAW === ###

put "hello" to stdout
put "" to stdout
put " world\n" to stdout
```

```
### === test_exponent.SLAW === ###

# expect 25 and 9 to be printed to stdout

set a to 5 ^ 2
set b to 3^2
put a to stdout
put b to stdout
```

```
### === test_factorial.SLAW === ###

# expect printed values: 6, 1 (reminder: zero factorial is one)

set a to 3
set b to a!
put b+"\n" to stdout

set a to 0
set b to a!
put b+"\n" to stdout
```

```
### === test_greatThan.SLAW === ###

if 2 > 1
  put "> works" to stdout
else
  put "> fails" to stdout
end if
```

```
if 3 > 5
  put "> fails" to stdout
else
  put "> works" to stdout
end if


if 3 > 3
  put "> fails" to stdout
else
  put "> works" to stdout
end if
```

```
### === test_greatThanOrEqualTo.SLAW === ###

if 2 >= 1
  put ">= works" to stdout
else
  put ">= fails" to stdout
end if


if -3 >= -3
  put ">= works" to stdout
else
  put ">= fails" to stdout
end if


if 4 >= -5
  put ">= works" to stdout
else
  put ">= fails" to stdout
end if
```

```
### === test_if.SLAW === ###

if 0
  put "ERROR: if 0 runs!\n" to stderr
else
  put "OK: if 0 does not run.\n" to stdout
end if

if 1
  put "OK: if 1 runs.\n" to stdout
else
  put "ERROR: if 1 does not run!\n" to stderr
end if

if false
  put "ERROR: if false runs!\n" to stderr
else
  put "OK: if false does not run.\n" to stdout
end if

if true
  put "OK: if true runs.\n" to stdout
else
  put "ERROR: if true does not run!\n" to stderr
end if
```

```
### === test_if_and_formal_parameters_locality_and_localize_in_a_procedure.SLAW === ###


set x to "Unmodified."
set y to "Unmodified."

do say_hello_if_positive[-10]
do say_hello_if_positive[0]
do say_hello_if_positive[10]

if x<>"Unmodified."
  put "Trouble at the mill: 'x' is now '"+x+"'.\n" to stderr
else
  put "Seems OK: 'x' is still '"+x+"'.\n" to stdout
end if

if y<>"Unmodified."
  put "Trouble at the mill: 'y' is now '"+y+"'.\n" to stderr
else
  put "Seems OK: 'y' is still '"+y+"'.\n" to stdout
end if


define procedure say_hello_if_positive[x]
  if x>0
    put "The parameter "+x+" is positive.\n" to stdout
    localize x # this should not have any effect because 'x' is already local, but it is
syntactically valid
    put "The parameter "+x+" is positive.\n" to stdout
    localize y
    set y to "Modified."
  end if
end procedure
```

```
### === test_instring.SLAW === ###

# Test multiple character substring position
set a to "Hello":"el"
if a == 2
  put ": WORKS" to stdout
else
  put ": FAILS" to stdout
end if

# Test single character substring position
set a to "Hello":"o"
if a == 5
  put ": WORKS" to stdout
else
  put ": FAILS" to stdout
end if

set b to "Hello":"x"
if b == 0
  put ": WORKS" to stdout
else
  put ": FAILS" to stdout
end if
```

```
# Test the implicitly contained empty string
set c to "x":""
if c == -1
  put ": WORKS" to stdout
else
  put ": FAILS" to stdout
end if
```

```
### === test_lessthan.SLAW === ###

if 1 < 2
  put "< works" to stdout
else
  put "< fails" to stdout
end if


if -3 < 4
  put "< works" to stdout
else
  put "< fails" to stdout
end if


if -6 < -5
  put "< works" to stdout
else
  put "< fails" to stdout
end if
```

```
### === test_lessThanOrEqualTo.SLAW === ###

if 1 <= 2
  put "<= works" to stdout
else
  put "<= fails" to stdout
end if


if -3 >= -3
  put "<= works" to stdout
else
  put "<= fails" to stdout
end if


if -4 <= 5
  put "<= works" to stdout
else
  put "<= fails" to stdout
end if
```

```
### === test_multiplication.SLAW === ###

## expect six results to be printed: 2, 0, 0, -2, 2, 2

set result1 to 1*2
set result2 to 2*0
set result3 to -2*0

set result4 to 2*-1
set result5 to (-2)*(-1)
set result6 to -2*-1

put "\n1*2=" to stdout
put 1*2 to stdout

put "\nhi 42 times =" to stdout
put "hi"*42 to stdout

put "\nhi 4.0 times=" to stdout
put "hi"*"4.0" to stdout

put "\nhi 0 times=" to stdout
put "hi"*0 to stdout

put "\nhi -0.5 times=" to stdout
put "hi"*-0.5+"\n" to stdout

put result1+"\n" to stdout
put result2+"\n" to stdout
put result3+"\n" to stdout
put result4+"\n" to stdout
put result5+"\n" to stdout
put result6+"\n" to stdout
```

```
### === test_multiplication_cases.SLAW === ###

# A=numeric string - "3.0"
# B=number - 10
# C=non-numeric string = "HI"

# put "10.0" to stdout
#AA,AB,AC with variants
#CASE 0 numeric-string on negative numeric-string
put "\nCASE 0: '3.0' * '-4.0'=" to stdout
put "3.0" * "-4.0" to stdout

#CASE 1 numeric-string on positive numeric-string
put "\nCASE 1: '3.0' * '2.0'=" to stdout
put "3.0" * "2.0" to stdout

#CASE 2 numeric-string on number
put "\nCASE 2: '3.0' * 10=" to stdout
put "3.0" * 10 to stdout

#CASE 3 numeric-string on non-numeric string
put "\nCASE 3: '3.0' * 'HI'=" to stdout
put "3.0" * "HI" to stdout

#CASE 4 negative numeric-string on non-numeric string (non zero after rounding)
#put "\nCASE 4: '-3.0' * 'HI'=" to stdout
#put "-3.0" * "HI" to stdout

#CASE 5 negative numeric-string on non-numeric string (**zero after rounding**)
put "\nCASE 5: '-0.3' * 'HI'=" to stdout
put "-0.3" * "HI" to stdout
```

```
#BA,BB,BC with variants

#CASE 6 - number on numeric string
put "\nCASE 6: 10 * '3.0'=" to stdout
put 10 * "3.0" to stdout

#CASE 7 - number on number
put "\nCASE 7: 10 * 10=" to stdout
put 10 * 10 to stdout

#CASE 8 - positive number on non-numeric string
put "\nCASE 8: 10 * 'HI'=" to stdout
put 10 * "HI" to stdout

#CASE 9 - negative number (after rounding) on non-numeric string
#put "\nCASE 9: -10 * 'HI'=" to stdout
#put -10 * "HI" to stdout

#CASE 10 - zero (after) rounding negative number on non-numeric string
put "\nCASE 10: -0.3 * 'HI'=" to stdout
put -0.3 * "HI" to stdout


#CA, CB, CC with variants

#CASE 11 - non-numeric string on positive numeric string
put "\nCASE 11: 'HI' * '3.0' =" to stdout
put "HI" * "3.0" to stdout

#CASE 12 - non-numeric string on negative numeric string
#put "\nCASE 12: 'HI' * '-3.0' =" to stdout
#put "HI" * "-3.0" to stdout

#CASE 13 - non-numeric string on zero after rounding number
put "\nCASE 13: 'HI' * -0.3 =" to stdout
put "HI" * -0.3 to stdout


#CASE 14 - non-numeric string on positive number
put "\nCASE 14: 'HI' * 10.0 =" to stdout
put "HI" * 10.0 to stdout

#CASE 15 - non-numeric string on negative number
put "\nCASE 15: 'HI' * -10.0 =" to stdout
put "HI" * -10.0 to stdout

#CASE 16 - non-numeric string on zero after rounding number
put "\nCASE 16: 'HI' * -0.3 =" to stdout
put "HI" * -0.3 to stdout

#CASE 17 - non-numeric string on non-numeric string
put "\nCASE 17: 'HI' * 'HI' =" to stdout
put "HI" * "HI" to stdout
```

```
### === test_negative.SLAW === ###

# expect printed values: 2, -3, 0

set a to -2
set b to -a
put b to stdout

set a to 3
set b to -a
put b to stdout
```

```
set a to 0
set b to -a
put b to stdout
```

```
### === test_not.SLAW === ###

# Test with true and with false...

if not false
  put "'not' works.\n" to stdout
else
  put "'not' fails.\n" to stdout
end if

if not true
  put "'not' fails.\n" to stdout
else
  put "'not' works.\n" to stdout
end if

# Test with (numerics and numeric strings) representing numbers other than zero and one...

if not 5
  put "'not' fails.\n" to stdout    # since non-zero is considered true
else
  put "'not' works.\n" to stdout
end if

if not "5"
  put "'not' fails.\n" to stdout    # since non-zero is considered true
else
  put "'not' works.\n" to stdout
end if
```

```
### === test_or.SLAW === ###

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
```

```
### === test_postfix.SLAW === ###

# expect 2, -50, 0.01, 0.202, and 0.036 to be printed to stdout

put 200% to stdout
put -5000% to stdout
put 1% to stdout
put 20! to stdout
```

```
### === test_precedence.SLAW === ###

set a to (3+2)*3
put a+"\n" to stdout    # expect 15 to print to stdout

set b to 4*(5-2)
put b+"\n" to stdout    # expect 12 to print to stdout
```

```
### === test_prefix.SLAW === ###

# expect 10, 20, 2, 4, and -4 to be printed to stdout
put ~10 to stdout
put ~"20" to stdout
put ~2.4 to stdout
put ~"3.5" to stdout
put ~(-3.5) to stdout
```

```
### === test_procedure_not_enough_params.SLAW === ###

do say_hello

define procedure say_hello[x]
  repeat x times
    put "hello\n" to stdout
  end repeat

  put "parameter x is " to stdout
  put x to stdout
  put "\n" to stdout

end procedure
```

```
### === test_procedure_one_param.SLAW === ###

do say_hello[3]

define procedure say_hello[x]
  repeat x times
    put "hello\n" to stdout
  end repeat

  put "parameter x is " to stdout
  put x to stdout
  put "\n" to stdout

end procedure
```

```
### === test_procedure_too_many_params.SLAW === ###

do say_hello[3,4]

define procedure say_hello[x]
  repeat x times
    put "hello\n" to stdout
  end repeat

  put "parameter x is " to stdout
  put x to stdout
  put "\n" to stdout

end procedure
```

```
### === test_procedure_zero_params.SLAW === ###

do say_hello

define procedure say_hello
  put "hello\n" to stdout
end procedure
```

```
### === test_recursion.SLAW === ###

# Testing of recursion using a factorial function.

# The implementation of this function in SLAWscript is pointless since
# the '!' exists; it's only for testing recursion.

define function factorial_func[n]
  if n <= 1
    return 1
  else
    return n*factorial_func[n-1]
  end if
end function

set a to factorial_func[4]
if a==4!
  put "Recursion WORKS\n" to stdout
else
  put "Recursion FAILS\n" to stdout
end if
```

```
### === test_relaxed_equality.SLAW === ###

if "9" = 9
   put "= works" to stdout
else
   put "= fails" to stdout
end if

if 9 = 9
   put "= works" to stdout
else
   put "= fails" to stdout
end if
```

```
### === test_relaxed_inequality.SLAW === ###

if "9" <> 9
   put "<> fails" to stdout
else
   put "<> works" to stdout
end if

if "9" <> 8
   put "<> works" to stdout
else
   put "<> fails" to stdout
end if

if 9 <> 8
   put "<> works" to stdout
else
   put "<> fails" to stdout
end if

if "9" <> 8
   put "<> works" to stdout
else
   put "<> fails" to stdout
end if
```

```
### === test_repeat_negstring_times.SLAW === ###

# this test should trigger an abort

put "Hi \"-1\" times:\n" to stdout
repeat "-1" times
  put "Hi\n" to stdout
end repeat
```

```
### === test_repeat_times.SLAW === ###

put "Test 1: 'Hi' 3 times...\n" to stdout
repeat 3 times
  put "Hi\n" to stdout
end repeat

put "Test 2: 'Hi' 3.3 times...\n" to stdout
repeat 3.3 times
  put "Hi\n" to stdout
end repeat

put "Test 3: 'Hi' 0 times...\n" to stdout
repeat 0 times
  put "Hi\n" to stdout
end repeat

put "Test 4: 'Hi' \"3\" times...\n" to stdout
repeat "3" times
  put "Hi\n" to stdout
end repeat

put "Test 5: 'Hi' \"0\" times...\n" to stdout
repeat "0" times
  put "Hi\n" to stdout
end repeat
```

```
# this test should NOT trigger an abort, despite using a negative number, because the
number rounds to zero
put "Test 6: 'Hi' -0.1 times...\n" to stdout
repeat -0.1 times
  put "Hi\n" to stdout
end repeat

# this test should trigger an abort
put "Test 7: 'Hi' -1.1 times...\n" to stdout
repeat -1.1 times
  put "Hi\n" to stdout
end repeat
```

```
### === test_repeat_with.SLAW === ###

put "Test 1: 'Hi' 3 times (from 1 to 3), with counter output...\n" to stdout
repeat with x from 1 to 3 # default step
  put "Hi: "+x+"\n" to stdout
end repeat

put "Test 2: 'Hi' 3 times (from 3 to 1), with counter output...\n" to stdout
repeat with x from 3 to 1 # default step
  put "Hi: "+x+"\n" to stdout
end repeat

put "Test 3: 'Hi' 3 times (from 0.5 to 1.5 step 0.5), with counter output...\n" to stdout
repeat with x from 0.5 to 1.5 step 0.5 # note: from 0.1 to 0.3 step 0.1 only executes
twice due to trouble with binary-based floating-point representation of decimal fractions
  put "Hi: "+x+"\n" to stdout
end repeat

put "Test 4: 'Hi' 3 times (from -0.5 to -1.5 step -0.5), with counter output...\n" to
stdout
repeat with x from -0.5 to -1.5 step -0.5
  put "Hi: "+x+"\n" to stdout
end repeat

put "Test 5: 'Hi' 3 times (from \"-0.5\" to \"-1.5\" step \"-0.5\"), with counter
output...\n" to stdout
repeat with x from "-0.5" to "-1.5" step "-0.5"
  put "Hi: "+x+"\n" to stdout
end repeat

put "Test 6: 'Hi' 3 times (from 0.1 to 0.3 step 0.1), with counter output...\n" to stdout
repeat with x from 0.1 to 0.3 step 0.1
  put "Hi: "+x+"\n" to stdout
end repeat

put "Test 7: 'Hi' 3 times (from 0.01 to 0.03 step 0.01), with counter output...\n" to
stdout
repeat with x from 0.01 to 0.03 step 0.01
  put "Hi: "+x+"\n" to stdout
end repeat

put "Test 8: 'Hi' 3 times (from 0.001 to 0.003 step 0.001), with counter output...\n" to
stdout
repeat with x from 0.001 to 0.003 step 0.001
  put "Hi: "+x+"\n" to stdout
end repeat

put "Test 9: 'Hi' 3 times (from 0.0001 to 0.0003 step 0.0001), with counter output...\n"
to stdout
repeat with x from 0.0001 to 0.0003 step 0.0001
  put "Hi: "+x+"\n" to stdout
end repeat
```

```
### === test_stop.SLAW === ###

stop

put "stop fails" to stdout
```

```
### === test_strict_equality.SLAW === ###

if "9" == 9
   put "== fails" to stdout
else
   put "== works" to stdout
end if

if 9 == 9
   put "== works" to stdout
else
   put "== fails" to stdout
end if

if 2 == 3
   put "== fails" to stdout
else
   put "== works" to stdout
end if
```

```
### === test_strict_inequality.SLAW === ###

if "9" <<>> 9
   put "<<>> fails" to stdout
else
   put "<<>> works" to stdout   # fails b/c it's comparing a number and a string
end if

if 9 <<>> 8
   put "<<>> works" to stdout
else
   put "<<>> fails" to stdout
end if
```

```
### === test_string_length.SLAW === ###

put "expecting 0: "+|""|+"\n" to stdout
put "expecting 1: "+|"1"|+"\n" to stdout
put "expecting 5: "+|"hello"|+"\n" to stdout



### === test_substring_postfix.SLAW === ###

set a to "123456789"

put a@3+"\n" to stdout  # expect "3456789"
put a@3;2+"\n" to stdout  # expect "34"
put a@3;400+"\n" to stdout  # expect "3456789"
```

```
# Test for errors
set a to "a"@1 # this should be OK
put "a"@2 to stdout # this should cause a warning to be sent to stderr, w/o program abort
put "still here\n" to stdout
put "a"@1;0 to stdout # this should _not_ cause a warning to be sent (zero is OK; it
means "I want an empty string")
put "still here\n" to stdout
put "a"@1;-1 to stdout # this should cause a warning to be sent to stderr, w/o program
abort
put "still here\n" to stdout
```

```
### === test_subtraction.SLAW === ###

# expect 45 to be printed out twice

set a to 9 * 5
set b to 9*5
put a+"\n" to stdout
put b+"\n" to stdout
```

```
### === test_undefined.SLAW === ###

put a to stdout
```

```
### === test_Unicode.SLAW === ###

# This file's text is encoded as UTF-8.

# This commment is a test, Señor/Señora/Señorita.

put "10÷2=5\n" to stdout
```

```
### === test_variableContentType.SLAW === ###

# expect 0, 1, and 2 to be printed to stdout

set a to "abc"
set b to "123"
set c to 456

put "expecting 0: "+a?+"\n" to stdout
put "expecting 1: "+b?+"\n" to stdout
put "expecting 2: "+c?+"\n" to stdout
put "expecting an error... " to stdout
put d? to stdout
```

```
### === test_variableValidity.SLAW === ###

# expect 0, 1, and 2 to be printed to stdout

set b to "123abc"
set c to 456

put a??+"\n" to stdout
put b??+"\n" to stdout
put c??+"\n" to stdout
```

```
### === test_while.SLAW === ###

set a to 10
while a > 0
  put "Counting down (using a while loop): "+a+"\n" to stdout
  set a to a-1
end while
put "Liftoff!\n" to stdout
```

```
### === variables.SLAW === ###

set num to 42
set str to "Hello"
put num to stdout
put "\n" to stdout
put str to stdout
put "\n" to stdout
```

8.4 SLAWscript sample code

```
### === colors.SLAW === ###

put "If your terminal supports ANSI color, then...\n" to stdout

define procedure show[code,bkgr]
  put escape+"[0;"+code+"m" to stdout

  put escape+"[31mThis should be red on a "+bkgr+" background.\n" to stdout
  put escape+"[32mThis should be green on a "+bkgr+" background.\n" to stdout
  put escape+"[34mThis should be blue on a "+bkgr+" background.\n" to stdout
  put escape+"[1;37mThis should be white on a "+bkgr+" background.\n" to stdout
  put escape+"[0;"+code+"m" to stdout # doing it again to cancel "bright on"
  put escape+"[36mThis should be cyan on a "+bkgr+" background.\n" to stdout
  put escape+"[35mThis should be magenta on a "+bkgr+" background.\n" to stdout
  put escape+"[1;33mThis should be yellow on a "+bkgr+" background.\n" to stdout
  put escape+"[0;"+code+"m" to stdout # doing it again to cancel "bright on"
  put escape+"[30mThis should be black on a "+bkgr+" background.\n" to stdout
end procedure

do show[41,"red"]
do show[42,"green"]
do show[44,"blue"]
do show[46,"cyan"]
do show[45,"magenta"]
do show[40,"black"]

put escape+"[0m" to stdout # restore defaults
```

```
### === colors_simple.SLAW === ###

put "If your terminal supports ANSI color, then...\n" to stdout

put escape+"[31mThis should be red.\n" to stdout
put escape+"[32mThis should be green.\n" to stdout
put escape+"[34mThis should be blue.\n" to stdout

put escape+"[36mThis should be cyan.\n" to stdout
put escape+"[35mThis should be magenta.\n" to stdout
put escape+"[01;33mThis should be yellow.\n" to stdout

put escape+"[0m" to stdout # restore defaults
```

```
### === HelloWorld.SLAW === ###

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

```
### === InteractiveColors_infinite.SLAW === ###

put "Please enter something you would like to see in several different colors: " to stdout

get input

while true
  put escape+"[0;31m"+input+"\n" to stdout
  put escape+"[32m"+input+"\n" to stdout
  put escape+"[34m"+input+"\n" to stdout
  put escape+"[36m"+input+"\n" to stdout
  put escape+"[35m"+input+"\n" to stdout
  put escape+"[1;33m"+input+"\n" to stdout

  # a little delay loop, so the effect doesn't blur to a flashing mess...
  repeat 10000 times
  end repeat
end while
```

```
### === InteractiveColors_once.SLAW === ###

put "Please enter something you would like to see in several different colors: " to stdout

get input

put escape+"[2;31m"+input+"\n" to stdout
put escape+"[32m"+input+"\n" to stdout
put escape+"[34m"+input+"\n" to stdout
put escape+"[36m"+input+"\n" to stdout
put escape+"[35m"+input+"\n" to stdout
put escape+"[1;33m"+input+"\n" to stdout

put escape+"[0m" to stdout # restore defaults
```

```
### === Interactive_GCD.SLAW === ###

# Written by Wei and Abe
# interactive_GCD.SLAW

# GCD function: recursive version
define function GCD[a,b]
  # note - assuming 'a' and 'b' are both numbers, for efficiency

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

end function


put "Welcome to the interactive Greatest Common Divisor ('GCD') program.\n" to stdout


set playing to true

while playing

    # ask for the first number in GCD
    set good to false
    while not good
      put "Please enter the first integer {range: [1,99]} to be used in the GCD: \n" to
stdout
      get num_first
      if not num_first?
        put "This is not a number: "+num_first+"\n" to stderr
      # else, we do have a numeric string
      else if not (~num_first = num_first)
        put "This is not an integer: "+num_first+"\n" to stderr
      else if num_first<=0
        put "This is a non-positive integer: "+num_first+"\n" to stderr
      else if num_first>99
        put "This is too big: "+num_first+"\n" to stderr
      else
        set good to true
      end if
    end while

    # ask for the second number in GCD
    set good to false
    while not good
      put "Please enter the first second {range: [1,99]} to be used in the GCD: \n" to
stdout
      get num_second
      if not num_second?
        put "This is not a number: "+num_second+"\n" to stderr
      # else, we do have a numeric string
      else if not (~num_second = num_second)
        put "This is not an integer: "+num_second+"\n" to stderr
      else if num_second<=0
        put "This is a non-positive integer: "+num_second+"\n" to stderr
      else if num_second>99
        put "This is too big: "+num_second+"\n" to stderr
      else
        set good to true
      end if
    end while
```

```
    # give result
    set GCD_result to GCD[num_first,num_second]
    put "The GCD of "+num_first+" and "+num_second+" is: " + GCD_result + "\n" to stdout

    # ask for whether or not to do it again
    set input to ""
    while input<>"y" and input<>"Y" and input<>"n" and input<>"N" and input<>"q" and
input<>"Q"
      put "If you want to do it again, please enter the letter 'Y'; please enter the
letter 'N' or 'Q' for no/quit.\n" to stdout
      get input
      set input to input+"   " # so input is definitely not an empty string
      set input to input@1;1
    end while

    set playing to (input="y" or input="Y")

end while

put "Thank you for playing!\n" to stdout
```

```
### === Interactive_LCM.SLAW === ###

# Written by Wei and Abe


# GCD function: recursive version
define function GCD[a,b]
  # note - assuming 'a' and 'b' are both numbers, for efficiency

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

end function


put "Welcome to the interactive Least Common Multiple ('LCM') program.\n" to stdout


set playing to true

while playing

    set good to false
    while not good
      put "Please enter the first integer {range: [1,99]}: \n" to stdout
      get num_first
      if not num_first?
        put "This is not a number: "+num_first+"\n" to stderr
      # else, we do have a numeric string
      else if not (~num_first = num_first)
        put "This is not an integer: "+num_first+"\n" to stderr
      else if num_first<=0
        put "This is a non-positive integer: "+num_first+"\n" to stderr
      else if num_first>99
        put "This is too big: "+num_first+"\n" to stderr
      else
        set good to true
      end if
    end while
```

```
    set good to false
    while not good
      put "Please enter the second integer {range: [1,99]}: \n" to stdout
      get num_second
      if not num_second?
        put "This is not a number: "+num_second+"\n" to stderr
      # else, we do have a numeric string
      else if ~num_second <> num_second
        put "This is not an integer: "+num_second+"\n" to stderr
      else if num_second<=0
        put "This is a non-positive integer: "+num_second+"\n" to stderr
      else if num_second>99
        put "This is too big: "+num_second+"\n" to stderr
      else
        set good to true
      end if
    end while

    # convert the input strings to numbers, to make sure the math in LCM calculation works
    # (otherwise '*' will do a string multiplication)
    set num_first to 0+num_first
    set num_second to 0+num_second

    set GCD_result to GCD[num_first,num_second]
    put "The Greatest Common Divisor ('GCD') of "+num_first+" and "+num_second+" is: " +
GCD_result + "\n" to stdout
    put "The LCM of "+num_first+" and "+num_second+" is: " +
(num_first*num_second)/GCD_result + "\n" to stdout

    # ask for whether or not to do it again
    set input to ""
    while input<>"y" and input<>"Y" and input<>"n" and input<>"N" and input<>"q" and
input<>"Q"
      put "If you want to do it again, please enter the letter 'Y'; please enter the
letter 'N' or 'Q' for no/quit.\n" to stdout
      get input
      set input to input+"   " # so input is definitely not an empty string
      set input to input@1;1
    end while

    set playing to (input="y" or input="Y")

end while

put "Thank you for playing!\n" to stdout
```

```
### === rounder.SLAW === ###

put "\nHello, and welcome to the 'rounder' program.\n" to stdout

set input to "" # this is here so 'input' won't be empty for the first check of the
'while'.

while input<>"stop"
  put "Please enter a number, or 'stop' if you want to stop.\n" to stdout
  get input

  if input?>0 # this means, "if 'input' contains numeric data"
    put "The number you entered was "+input+" and the nearest integer to that is "
+~input+".\n" to stdout
  end if
end while

put "\nThank you for running this program.\n" to stdout
```

```
### === spinner.SLAW === ###

put "\n\n " to stdout
while true
  put escape+"[D-" to stdout
  do delay
  put escape+"[D\\" to stdout # double-'\' because '\' is normally an escape character
  do delay
  put escape+"[D|" to stdout
  do delay
  put escape+"[D/" to stdout
  do delay
end while

define procedure delay
  repeat 1000000 times
    # empty body on purpose
  end repeat
end procedure
```

8.5 Shell Scripts

```
### === build === ###

#!/bin/sh

echo ===== Compiling... =====
echo

./compile

echo
echo ===== Jarifying... =====
echo

./jarify

echo
echo ===== Done building. =====
```

```
### === compile === ###

#!/bin/sh
for a in *va; do echo === $a ===; javac -cp antlr.jar:. $a; echo; done
```

```
### === jarify === ###

#!/bin/sh

# The following had to get a little messy in order to not include the ".svn" directories,
# which otherwise roughly doubled the size of the jar file!

jar -cfm SLAWscript.jar JarManifest.txt *.class org/antlr/runtime/*.class (line cont.`s)
                    org/antlr/runtime/*.class org/antlr/runtime/*/*.class
```

```
### === run === ###

#!/bin/sh
# This file was written by Abe.
java -cp antlr.jar:. SLAWscript $@
```

```
### === slaw === ###

#!/bin/sh

# This file was written by Abe.

# The following technique is in case this file is sourced from within an interactive
# shell, wherein $0 and $_ will both be e.g. "sh" or "-bash".  I was previously assuming
# in those cases that the directory from which to load the jar file should be '.',
# but that doesn't always work, so rather than allow it to work some of the time, I (Abe)
# decided to make it always not work.

if [ "$0" = "sh" -o "$0" = "bash" -o "$0" = "-bash" -o "$0" = "/bin/bash" -o "$0" = "/bin/sh" ];
then
  echo Please do not source this file.  Please execute it instead.
else

# Now, we deal with the fact that SLAWscript requires at least Java 1.5, and inform the
# user of this if needed.

  if echo "`java -version 2>&1 | grep version`" | grep '1.[56789]' > /dev/null ; then
    java -jar `dirname $0`/SLAWscript.jar $@
  else
    echo 'Sorry, SLAWscript requires a Java runtime version of 1.5 or higher.'
    echo 'Your system currently has this Java runtime version: '`java -version 2>&1 | grep version`
    echo 'Please upgrade your Java runtime or run this program on a different system.'
  fi

fi
```

```
### === tarify === ###

#!/bin/sh

tar cvf SLAWscript.tar SLAWscript.jar slaw
```

8.6 Miscellaneous Files

JarManifest.txt

```
Main-Class: SLAWscript
```