Columbia University
Programming Languages and Translators
Spring 2007
Professor Edwards

SLAWscript Language Reference Manual

Steve Henderson
Levi Lister
Abe Skolnik
Wei Teng

Table of Contents

Section Page Number

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.

1.1 Hello World

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

2. Lexical Conventions

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. This language 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
```

2.2 Constants

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

`e`	2.718281828459045…
`pi`	3.141592653589793… (i.e. π)
`false`	0
`true`	1

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

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.

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

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"
```

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 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: “`\"`”.

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. 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 parameters) 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".

3.1 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).

3.2 Procedures

Procedures do not return any values, and cannot be used 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 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 <u>not</u> appear inside a loop, including inside an "if" paragraph inside a loop.
Each execution of a function <u>must</u> end in a "return" sentence; arriving at the "end function" line without returning any value is an error.

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.

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:

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

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 a string
set a to b    #  This is not illegal, but it is confusing; please use "copy" instead.
copy a to b   #  Please note the opposite direction of data flow relative to "set".
```

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, and is ignored. 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 <u>not</u> 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.
```

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`".

5. Operators

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.
`??`	variable validity (postfix): returns 0 if the variable is undefined, 1 if it holds any 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.

5.2 Binary and Tertiary Operators

`^`	exponentiation.
`/`	division.
`*`	multiplication (both numeric and string).
`-`	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 Auto-conversion").
`==`	strict equality (see "Binary 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 <u>not</u> always evaluated).
`or`	boolean OR (short-circuited: left operand is always evaluated, right operand is <u>not</u> always evaluated).
`@`	substring (postfix): must be followed by either one operand or two operands separated by a semicolon; "`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. Returns the empty string if the third operand is present and is zero after rounding.
`:`	substring position: `"Hello":"el"` returns 2; `"Hello":"x"` returns 0; `"x":"Hello"` returns 0; `"":a` returns 0 for any defined 'a'; `a:""` returns -1 for any defined value of 'a' (including numbers), since the empty string is implicitly contained within every string, yet its position within the enclosing string cannot be defined. 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.

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.	`+`	binary –						
8.	`<`	`>`	`<=`	`>=`	`=`	`==`	`<>`	`<<>>`
9.	`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!))`". 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.

5.4 Operator Chaining

The binary and tertiary operators allow chaining; for example, "`set a to b+c+d`" is perfectly valid. However, please be advised that chaining operators will not always produce what you might expect. For example: in math classes, one is typically taught that "a<b<c" means that 'a' is less than b <u>and</u> 'b' is less than 'c'. However, in SLAWscript, as in many other programming languages, although "`a<b<c`" is a valid expression, it does <u>not</u> mean what it would mean in a math class. To get the math-class meaning of "a<b<c" in SLAWscript, you must use something equivalent to "`a<b and b<c`" instead.

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 <u>not</u> 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 <u>only</u> 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.

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: "( )", "| |", '%', '?', "??".

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. gets to keep its current type. Therefore, `"9"*9` yields the string "999999999", whereas `9*"9"` yields the number 81.

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' is now "9" because the left-hand empty string "won" the data-type conversion "contest"
copy i to j
set j to 0+j
# 'j' is now 9 (numeric) because the left-hand zero "won"
copy i to k
set k to 1*k # this is another way to force a numeric without 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+…`". Other techniques are left as an exercise for the reader.

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 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 '?' and “??” operators, while they do not produce only strictly boolean values, have been designed in such a way as to make them usable by themselves, as if they 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), and the “??” operator can be used by itself to verify that the variable is currently defined.

Note: 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” or “if x??>=2”, both of which are more future-proof than “x?=2”, since a future version of SLAWscript may allow '?' and/or “??” to return (for example) 3 for an integer and 4 for a boolean.
(Design concept: higher return values indicate a more-specific kind of number which is also enclosed within the lower positive-number return value number classes.)

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 “@” operator must take either one or two integers as its second and optional third operands. The second operand must be positive after rounding, and the third operand must be non-negative after rounding.

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

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
```

8. Loops

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

8.1 “repeat … times” Loops

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.

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 999999999 times
  put "Number 9... " to stdout
end repeat
```

8.2 “repeat with” Loops

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 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, 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.

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).
There is no particular restriction on this number; it can be any real number, so long as the loop makes sense.

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).
There is no particular restriction on this number; it can be any real number, so long as the loop makes sense.

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.

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
```

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
```

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.

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.

An example follows.

```
get foo
```

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

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
```

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.

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.

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” to a literal or constant value which follows the word “is” 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.

11. 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.