**LOLCODE Specification 1.2**

FINAL DRAFT — 12 July 2007

*The goal of this specification is to act as a baseline for all following LOLCODE specifications. As such, some traditionally expected language features may appear "incomplete." This is most likely deliberate, as it will be easier to add to the language than to change and introduce further incompatibilities.*

**Formatting**

**Whitespace**

* Spaces are used to demarcate tokens in the language, although some keyword constructs may include spaces.
* Multiple spaces and tabs are treated as single spaces and are otherwise irrelevant.
* Indentation is irrelevant.
* A command starts at the beginning of a line and a newline indicates the end of a command, except in special cases.
* A newline will be Carriage Return (/13), a Line Feed (/10) or both (/13/10) depending on the implementing system. This is only in regards to LOLCODE code itself, and does not indicate how these should be treated in strings or files during execution.
* Multiple commands can be put on a single line if they are separated by a comma (,). In this case, the comma acts as a virtual newline or a soft-command-break.
* Multiple lines can be combined into a single command by including three periods (...) or the unicode ellipsis character (u2026) at the end of the line. This causes the contents of the next line to be evaluated as if it were on the same line.
* Lines with line continuation can be strung together, many in a row, to allow a single command to stretch over more than one or two lines. As long as each line is ended with three periods, the next line is included, until a line without three periods is reached, at which point, the entire command may be processed.
* A line with line continuation may not be followed by an empty line. Three periods may be by themselves on a single line, in which case, the empty line is "included" in the command (doing nothing), and the next line is included as well.
* A single-line comment is always terminated by a newline. Line continuation (...) and soft-command-breaks (,) after the comment (BTW) are ignored.
* Line continuation and soft-command-breaks are ignored inside quoted strings. An unterminated string literal (no closing quote) will cause an error.

**Comments**

*(from 1.1)*

Single line comments are begun by BTW, and may occur either after a line of code, on a separate line, or following a line of code following a line separator (,).

All of these are valid single line comments:

I HAS A VAR ITZ 12 BTW VAR = 12

I HAS A VAR ITZ 12, BTW VAR = 12

I HAS A VAR ITZ 12

BTW VAR = 12

Multi-line comments are begun by OBTW and ended with TLDR, and should be started on their own lines, or following a line of code after a line separator.

These are valid multi-line comments:

I HAS A VAR ITZ 12

OBTW this is a long comment block

see, i have more comments here

and here

TLDR

I HAS A FISH ITZ BOB

I HAS A VAR ITZ 12, OBTW this is a long comment block

see, i have more comments here

and here

TLDR, I HAS A FISH ITZ BOB

**File Creation**

*(modified from 1.1)*

All LOLCODE programs must be opened with the command HAI. HAI should then be followed with the current LOLCODE language version number (1.2, in this case). There is no current standard behavior for implementations to treat the version number, though.

A LOLCODE file is closed by the keyword KTHXBYE which closes the HAI code-block.

**Variables**

**Scope**

*(to be revisited and refined)*

All variable scope, as of this version, is local to the enclosing function or to the main program block. Variables are only accessible after declaration, and there is no global scope.

**Naming**

*(from 1.1)*

Variable identifiers may be in all small or lowercase letters (or a mixture of the two). They must begin with a letter and may be followed only by other letters, numbers, and underscores. No spaces, dashes, or other symbols are allowed. Variable identifiers are CASE SENSITIVE – "cheezburger", "CheezBurger" and "CHEEZBURGER" would all be different variables.

**Declaration and Assignment**

*(modified from 1.1)*

To declare a variable, the keyword is I HAS A followed by the variable name. To assign the variable a value within the same statement, you can then follow the variable name with ITZ <value>.

Assignment of a variable is accomplished with an assignment statement, <variable> R <expression>

I HAS A VAR BTW VAR is null and untyped

VAR R "THREE" BTW VAR is now a YARN and equals "THREE"

VAR R 3 BTW VAR is now a NUMBR and equals 3

**Types**

*(updated from 1.1)*

The variable types that LOLCODE currently recognizes are: strings (YARN), integers (NUMBR), floats (NUMBAR), and booleans (TROOF) (Arrays (BUKKIT) are reserved for future expansion.) Typing is handled dynamically. Until a variable is given an initial value, it is untyped (NOOB).

**Untyped**

The untyped type (NOOB) cannot be implicitly cast into any type except a TROOF. A cast into TROOF makes the variable FAIL. Any operations on a NOOB that assume another type (e.g., math) results in an error.

Explicit casts of a NOOB (untyped, uninitialized) variable are to empty/zero values for all other types.

**Booleans**

The two boolean (TROOF) values are WIN (true) and FAIL (false). The empty string (""), an empty array, and numerical zero are all cast to FAIL. All other values evaluate to WIN.

**Numerical Types**

A NUMBR is an integer as specified in the host implementation/architecture. Any contiguous sequence of digits outside of a quoted YARN and not containing a decimal point (.) is considered a NUMBR. A NUMBR may have a leading hyphen (-) to signify a negative number.

A NUMBAR is a float as specified in the host implementation/architecture. It is represented as a contiguous string of digits containing exactly one decimal point. Casting a NUMBAR to a NUMBR truncates the decimal portion of the floating point number. Casting a NUMBAR to a YARN (by printing it, for example), truncates the output to a default of two decimal places. A NUMBAR may have a leading hyphen (-) to signify a negative number.

Casting of a string to a numerical type parses the string as if it were not in quotes. If there are any non-numerical, non-hyphen, non-period characters, then it results in an error. Casting WIN to a numerical type results in "1" or "1.0"; casting FAIL results in a numerical zero.

**Strings**

String literals (YARN) are demarked with double quotation marks ("). Line continuation and soft-command-breaks are ignored inside quoted strings. An unterminated string literal (no closing quote) will cause an error.

Within a string, all characters represent their literal value except the colon (:), which is the escape character. Characters immediately following the colon also take on a special meaning.

* :) represents a newline (\n)
* :> represents a tab (\t)
* :o represents a bell (beep) (\g)
* :" represents a literal double quote (")
* :: represents a single literal colon (:)

The colon may also introduce more verbose escapes enclosed within some form of bracket.

* :(<hex>) resolves the hex number into the corresponding Unicode code point.
* :{<var>} interpolates the current value of the enclosed variable, cast as a string.
* :[<char name>] resolves the <char name> in capital letters to the corresponding Unicode [normative name](http://www.unicode.org/Public/4.1.0/ucd/NamesList.txt).

**Arrays**

*Array and dictionary types are currently under-specified. There is general will to unify them, but indexing and definition is still under discussion.*

**Types**

The TYPE type only has the values of TROOF, NOOB, NUMBR, NUMBAR, YARN, and TYPE, as bare words. They may be legally cast to TROOF (all true except for NOOB) or YARN.

*TYPEs are under current review. Current sentiment is to delay defining them until user-defined types are relevant, but that would mean that type comparisons are left unresolved in the meantime.*

**Operators**

**Calling Syntax and Precedence**

Mathematical operators and functions in general rely on prefix notation. By doing this, it is possible to call and compose operations with a minimum of explicit grouping. When all operators and functions have known arity, no grouping markers are necessary. In cases where operators have variable arity, the operation is closed with MKAY. An MKAY may be omitted if it coincides with the end of the line/statement, in which case the EOL stands in for as many MKAYs as there are open variadic functions.

Calling unary operators then has the following syntax:

<operator> <expression1>

The AN keyword can optionally be used to separate arguments, so a binary operator expression has the following syntax:

<operator> <expression1> [AN] <expression2>

An expression containing an operator with infinite arity can then be expressed with the following syntax:

<operator> <expr1> [[[AN] <expr2>] [AN] <expr3> ...] MKAY

**Math**

The basic math operators are binary prefix operators.

SUM OF <x> AN <y> BTW +

DIFF OF <x> AN <y> BTW -

PRODUKT OF <x> AN <y> BTW \*

QUOSHUNT OF <x> AN <y> BTW /

MOD OF <x> AN <y> BTW modulo

BIGGR OF <x> AN <y> BTW max

SMALLR OF <x> AN <y> BTW min

<x> and <y> may each be expressions in the above, so mathematical operators can be nested and grouped indefinitely.

Math is performed as integer math in the presence of two NUMBRs, but if either of the expressions are NUMBARs, then floating point math takes over.

If one or both arguments are a YARN, they get interpreted as NUMBARs if the YARN has a decimal point, and NUMBRs otherwise, then execution proceeds as above.

If one or another of the arguments cannot be safely cast to a numerical type, then it fails with an error.

**Boolean**

Boolean operators working on TROOFs are as follows:

BOTH OF <x> [AN] <y> BTW and: WIN iff x=WIN, y=WIN

EITHER OF <x> [AN] <y> BTW or: FAIL iff x=FAIL, y=FAIL

WON OF <x> [AN] <y> BTW xor: FAIL if x=y

NOT <x> BTW unary negation: WIN if x=FAIL

ALL OF <x> [AN] <y> ... MKAY BTW infinite arity AND

ANY OF <x> [AN] <y> ... MKAY BTW infinite arity OR

<x> and <y> in the expression syntaxes above are automatically cast as TROOF values if they are not already so.

**Comparison**

Comparison is (currently) done with two binary equality operators:

BOTH SAEM <x> [AN] <y> BTW WIN iff x == y

DIFFRINT <x> [AN] <y> BTW WIN iff x != y

Comparisons are performed as integer math in the presence of two NUMBRs, but if either of the expressions are NUMBARs, then floating point math takes over. Otherwise, there is no automatic casting in the equality, so BOTH SAEM "3" AN 3 is FAIL.

There are (currently) no special numerical comparison operators. Greater-than and similar comparisons are done idiomatically using the minimum and maximum operators.

BOTH SAEM <x> AN BIGGR OF <x> AN <y> BTW x >= y

BOTH SAEM <x> AN SMALLR OF <x> AN <y> BTW x <= y

DIFFRINT <x> AN SMALLR OF <x> AN <y> BTW x > y

DIFFRINT <x> AN BIGGR OF <x> AN <y> BTW x < y

If <x> in the above formulations is too verbose or difficult to compute, don't forget the automatically created IT temporary variable. A further idiom could then be:

<expression>, DIFFRINT IT AN SMALLR OF IT AN <y>

*Suggestions are being accepted for coherently and convincingly english-like prefix operator names for greater-than and similar operators.*

**Concatenation**

An indefinite number of YARNs may be explicitly concatenated with the SMOOSH...MKAY operator. Arguments may optionally be separated with AN. As the SMOOSH expects strings as its input arguments, it will implicitly cast all input values of other types to YARNs. The line ending may safely implicitly close the SMOOSH operator without needing an MKAY.

**Casting**

Operators that work on specific types implicitly cast parameter values of other types. If the value cannot be safely cast, then it results in an error.

An expression's value may be explicitly cast with the binary MAEK operator.

MAEK <expression> [A] <type>

Where <type> is one of TROOF, YARN, NUMBR, NUMBAR, or NOOB. This is only for local casting: only the resultant value is cast, not the underlying variable(s), if any.

To explicitly re-cast a variable, you may create a normal assignment statement with the MAEK operator, or use a casting assignment statement as follows:

<variable> IS NOW A <type> BTW equivalent to

<variable> R MAEK <variable> [A] <type>

**Input/Output**

**Terminal-Based**

The print (to STDOUT or the terminal) operator is VISIBLE. It has infinite arity and implicitly concatenates all of its arguments after casting them to YARNs. It is terminated by the statement delimiter (line end or comma). The output is automatically terminated with a carriage return (:)), unless the final token is terminated with an exclamation point (!), in which case the carriage return is suppressed.

VISIBLE <expression> [<expression> ...][!]

There is currently no defined standard for printing to a file.

To accept input from the user, the keyword is

GIMMEH <variable>

which takes YARN for input and stores the value in the given variable.

*GIMMEH is defined minimally here as a holdover from 1.0 and because there has not been any detailed discussion of this feature. We count on the liberal casting capabilities of the language and programmer inventiveness to handle input restriction. GIMMEH may change in a future version.*

**Statements**

**Expression Statements**

A bare expression (e.g. a function call or math operation), without any assignment, is a legal statement in LOLCODE. Aside from any side-effects from the expression when evaluated, the final value is placed in the temporary variable IT. IT's value remains in local scope and exists until the next time it is replaced with a bare expression.

**Assignment Statements**

Assignment statements have no side effects with IT. They are generally of the form:

<variable> <assignment operator> <expression>

The variable being assigned may be used in the expression.

**Flow Control Statements**

Flow control statements cover multiple lines and are described in the following section.

**Flow Control**

**Conditionals**

**If-Then**

The traditional if/then construct is a very simple construct operating on the implicit IT variable. In the base form, there are four keywords: O RLY?, YA RLY, NO WAI, and OIC.

O RLY? branches to the block begun with YA RLY if IT can be cast to WIN, and branches to the NO WAI block if IT is FAIL. The code block introduced with YA RLY is implicitly closed when NO WAI is reached. The NO WAI block is closed with OIC. The general form is then as follows:

<expression>

O RLY?

YA RLY

<code block>

NO WAI

<code block>

OIC

while an example showing the ability to put multiple statements on a line separated by a comma would be:

BOTH SAEM ANIMAL AN "CAT", O RLY?

YA RLY, VISIBLE "J00 HAV A CAT"

NO WAI, VISIBLE "J00 SUX"

OIC

The elseif construction adds a little bit of complexity. Optional MEBBE <expression> blocks may appear between the YA RLY and NO WAI blocks. If the <expression> following MEBBE is WIN, then that block is performed; if not, the block is skipped until the following MEBBE, NO WAI, or OIC. The full expression syntax is then as follows:

<expression>

O RLY?

YA RLY

<code block>

[MEBBE <expression>

<code block>

[MEBBE <expression>

<code block>

...]]

[NO WAI

<code block>]

OIC

An example of this conditional is then:

BOTH SAEM ANIMAL AN "CAT"

O RLY?

YA RLY, VISIBLE "J00 HAV A CAT"

MEBBE BOTH SAEM ANIMAL AN "MAUS"

VISIBLE "NOM NOM NOM. I EATED IT."

OIC

**Case**

*(modified from 1.1)*

The LOLCODE keyword for switches is WTF?. The WTF? operates on IT as being the expression value for comparison. A comparison block is opened by OMG and must be a literal, not an expression. (A literal, in this case, excludes any YARN containing variable interpolation (:{var}).) Each literal must be unique. The OMG block can be followed by any number of statements and may be terminated by a GTFO, which breaks to the end of the the WTF statement. If an OMG block is not terminated by a GTFO, then the next OMG block is executed as is the next until a GTFO or the end of the WTF block is reached. The optional default case, if none of the literals evaluate as true, is signified by OMGWTF.

WTF?

OMG <value literal>

<code block>

[OMG <value literal>

<code block> ...]

[OMGWTF

<code block>]

OIC

COLOR, WTF?

OMG "R"

VISIBLE "RED FISH"

GTFO

OMG "Y"

VISIBLE "YELLOW FISH"

OMG "G"

OMG "B"

VISIBLE "FISH HAS A FLAVOR"

GTFO

OMGWTF

VISIBLE "FISH IS TRANSPARENT"

OIC

In this example, the output results of evaluating the variable COLOR would be:

"R":

RED FISH

"Y":

YELLOW FISH

FISH HAS A FLAVOR

"G":

FISH HAS A FLAVOR

"B":

FISH HAS A FLAVOR

none of the above:

FISH IS TRANSPARENT

**Loops**

*Loops are currently defined more or less as they were in the original examples. Further looping constructs will be added to the language soon.*

Simple loops are demarcated with IM IN YR <label> and IM OUTTA YR <label>. Loops defined this way are infinite loops that must be explicitly exited with a GTFO break. Currently, the <label> is required, but is unused, except for marking the start and end of the loop.

*Immature spec – \**subject to change\**:*

Iteration loops have the form:

IM IN YR <label> <operation> YR <variable> [TIL|WILE <expression>]

<code block>

IM OUTTA YR <label>

Where may be UPPIN (increment by one), NERFIN (decrement by one), or any unary function. That operation/function is applied to the , which is temporary, and local to the loop. The TIL evaluates the expression as a TROOF: if it evaluates as FAIL, the loop continues once more, if not, then loop execution stops, and continues after the matching IM OUTTA YR . The WILE is the converse: if the expression is WIN, execution continues, otherwise the loop exits.

**Functions**

**Definition**

A function is demarked with the opening keyword HOW IZ I and the closing keyword IF U SAY SO. The syntax is as follows:

HOW IZ I <function name> [YR <argument1> [AN YR <argument2> …]]

<code block>

IF U SAY SO

Currently, the number of arguments in a function can only be defined as a fixed number. The <argument>s are single-word identifiers that act as variables within the scope of the function's code. The calling parameters' values are then the initial values for the variables within the function's code block when the function is called.

*Currently, functions do not have access to the outer/calling code block's variables.*

**Returning**

Return from the function is accomplished in one of the following ways:

* FOUND YR <expression> returns the value of the expression.
* GTFO returns with no value (NOOB).
* in the absence of any explicit break, when the end of the code block is reached (IF U SAY SO), the value in IT is returned.

**Calling**

A function of given arity is called with:

I IZ <function name> [YR <expression1> [AN YR <expression2> [AN YR <expression3> ...]]] MKAY

That is, an expression is formed by the function name followed by any arguments. Those arguments may themselves be expressions. The expressions' values are obtained before the function is called. The arity of the functions is determined in the definition.