The Annotated LOLCODE Specification 1.2

Justin J. Meza

July 31, 2010

Introduction

The LOLCODE Specification 1.2 (hereafter simply called "the specification" or "the spec") was released on July 12, 2007. It reflected a community effort led by Adam T. Lindsay to build upon and refine the previous specifications of the language. Of course, 1.2 still lacked quite a few faculties present in most programming languages (arrays being the most glaring omission), but it seemed these were to be added in future specifications. Momentum has slowed in the LOLCODE community, however, and it is doubtful a new version of the language will be issued any time soon.

With this in mind, this document aims to accomplish three goals:

1. To mirror a copy of the 1.2 specification given the uncertain nature of the future of the language. For this, the original 1.2 specification has been copied verbatim from the one currently located at `http://www.lolcode.com/specs/1.2`.

2. To clarify points in the specification where information is lacking via annotations. In most cases, this advice comes from posts in the LOLCODE forums discussing the topic in question.

3. To highlight potential problems with the language in the event that future specifications are produced. These are geared toward anyone wishing to implement an interpreter or compiler for the language, or, if the language is ever revised.

All annotations are made as footnotes to preserve the original content of the specification.

LOLCODE 1.2 Specification

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[1]

whitespace

1. Spaces are used to demarcate tokens in the language, although some keyword constructs may include spaces.[2]

2. Multiple spaces and tabs are treated as single spaces and are otherwise irrelevant.

3. Indentation is irrelevant.

4. A command starts at the beginning of a line and a newline indicates the end of a command, except in special cases.

5. 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.[3]

6. 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.

7. 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.

8. 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.

[1] It is implied that files may be encoded in Unicode.

[2] What the specification defines is an LL(2) grammar.

[3] The spec is unclear as to whether line endings may be mixed within a code file. Doing this does not make sense but a robust parser should be able to handle such a case.

9. A line with line continuation may not be followed by an empty line.

10. 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.

11. A single-line comment is always terminated by a newline. Line continuation (...)[4] and soft-command-breaks (,) after the comment (`BTW`) are ignored.

12. Line continuation & 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 line, 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
```

[4] Also Unicode ellipsis character U+2026.

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)[5]. 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[6].[7]

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[8]. Variables are only accessible after declaration, and there is no global scope.

naming

from 1.1

Variable identifiers may be in all CAPITAL 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.[9]

declaration and assignment

modified from 1.1

[5]This is an interesting feature of the language: files can specify which version of the language their code uses. This was perhaps added as a result of the rapid early evolution of the lanugage to break old code as little as possible. From an implementation perspective, this could complicate matters if used to distinguish which version of the spec to use as it relies on the compiler or interpreter to "do the right thing" as opposed to the user choosing the right version of a program or setting the correct flag for compatibility.

[6]The notion of code blocks in LOLCODE is not well defined, however, and here the terminology is used just as an analogy for how the `HAI`/`KTHXBYE` pair work.

[7]The spec is not clear as to whether code following `KTHXBYE` is ever considered. Presumably it may contain comments or whitespace but nothing else.

[8]This definition of scope is problematic. If all variables within either the enclosing function or the main block are accessible after declaration, then any control path visited more than once within an instance of one of those blocks that defines a variable will result in an error because the variable will have already been defined. For example, the following would cause an error on the second iteration of the loop as the variable `a` would have already been defined in the first iteration of the loop: `IM IN YR loop, I HAS A a, IM OUTTA YR LOOP`.

[9]N.b.: Unicode characters are not allowed.

To declare a variable, the keyword is `I HAS A`[10] 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
```

Type conversion is handled automatically.[11]

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[12]. Until a variable is given an initial value, it is untyped (NOOB). ~~Casting operations operate on TYPE types, as well~~.

untyped

The untyped type (NOOB) cannot be implicitly cast into any type except a TROOF. A cast into TROOF makes the variable FAIL[13]. 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.[14]

booleans

The two boolean (TROOF) values are WIN (true) and FAIL (false). The empty string (“”), an empty array[15], and numerical zero are all cast to FAIL[16]. All other values evaluate to WIN.

[10] `I HAS A` is one of several multi-word keywords that introduces a problem with identifiers: in the statement `I HAS A I`, is the second “`I`” a valid identifier or an incomplete keyword? It depends on how the language is parsed: if the parser looks ahead far enough to identify every multi-word keyword as its own token, then the previous example is parseable at the expense of more lookahead (an LL(4) grammer as opposed to the minimal LL(2) grammer).

[11] This is a bit misleading as some type conversions are undefined or explicitly not allowed as we'll see in the next section: the point being the programmer must use caution when casting from certain types to certain other types.

[12] That is, type checks are performed at runtime.

[13] This is presumably a convenience for checking to see if a variable has been assigned any value.

[14] To wit: NOOB to NOOB gives NOOB; NOOB to TROOF gives FAIL; NOOB to NUMBR gives 0; NOOB to NUMBAR gives 0.0; NOOB to YARN gives “”.

[15] This refers to BUKKITs which are not covered in the spec.

[16] Also NOOB, from above.

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 NUMBR may have a leading hyphen (-) to signify a negative number.[17]

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[18]

String literals (YARN) are demarked with double quotation marks ("). Line continuation & 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.[19]

[17] There is currently no way to change the output precision of a decimal number.

[18] Notice that there is no mention either in this section or the booleans section as to how to cast booleans to strings! This seems like a rather useful operation but raises the question of how this would be done. Would boolean WIN be cast to the literal string "WIN" or be sort of indirectly cast to an integer (1) and then to a string ("1")? For now, the only way to "cast" a boolean to a string is to first cast it to some other type which can then be cast to a string.

[19] This is an obscure feature that introduces an undesirable dependency on the current Unicode normative name mapping. It would be better to just resolve a code point into the corresponding Unicode encoding in a deterministic fashion (the `:(<hex>)` approach, above.

arrays

Array and dictionary types are currently under-specified. There is general will to unify them, but indexing and definition is still under discussion.[20]

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.[21]

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[22]. 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 `MKAY`s 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[23]:

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

[20] There are actually several proposals on the forums regarding how to implement this much-needed feature (see `http://lolcode.com/proposals/1.3/bukkit` and `http://lolcode.com/proposals/1.3/bukkit2` for some good examples of how the ideas crystallized). In fact, the language is not Turing-complete without arrays.

[21] I'm not convinced it is useful to compare only the type of two variables. Perhaps comparing both the type *and* value of two variables (much like PHP's `===` comparison) but this is done anyway by the `BOTH SAEM` operator. With classes, this may be useful as it adds `instanceof` comparisons (but is this really the right way to use OOP in the first place?`</metacommentary>`).

[22] Okay, but let's be honest—it's because it just looks better!

[23] According the the first paragraph of this section, the following should actually include an EOL as an end marker: `<operator> <expr1> [[[AN] <expr2>] [AN] <expr3> ...] (MKAY|EOL)`.

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[24]

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

[24] The spec does not mention if short-circuit logic should be used or if all arguments must be checked (and all side-effects are triggered).

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[25].

```
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`[26] 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>
```

[25] I must harp on this point: greater-than and less-than operators needs to be addressed. The following examples are incredibly cumbersome.

[26] Note that `IT` is defined later in the section covering "expression statements".

Input/Ouput

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.[27]
To accept input from the user, the keyword is

```
GIMMEH <variable>
```

which takes YARN for input and stores the value in the given variable[28].
`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[29], 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.[30]

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.

[27] Although there are some interesting propositions including `http://forum.lolcode.com/viewtopic.php?id=312`.

[28] Does it keep the EOL marker or discard it?

[29] Do expressions have side-effects by most definitions?

[30] `IT` needs to be more rigorously defined: what is its value before an expression statement has been executed? NOOB? The same as the value of `IT` in the enclosing control block? Is accessing `IT` in this way undefined?

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 "JOO HAV A CAT"
  NO WAI, VISIBLE "JOO 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 true, 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 "JOO 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[31]. (A literal, in this case, excludes any YARN containing variable interpolation (`:{var}`).[32]) Each literal must be unique[33]. 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 `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`.

```
<expression>
WTF?
  OMG <value literal>
    <code block>
 [OMG <value literal>
    <code block> ...]
 [OMGWTF
    <code block>]
OIC

COLOR, WTF?
  OMG "R"
    VISIBLE "RED FISH"
    GTFO
```

[31] The spec is not clear as to if only the values are compared or both the values and types are compared (similar to `BOTH SAEM`). If indeed only the values are compared, are values cast to the type of the expression for comparison or the literal being compared?

[32] I'm not convinced this constraint is useful.

[33] Is 3 the same as 3.0? It is unclear.

```
  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[34].

*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>
```

[34]Some discussion has surrounded whether `<label>` would be useful to break out of nested loops.

Where `<operation>` may be `UPPIN` (increment by one), `NERFIN` (decrement by one), or any unary function[35]. That operation/function is applied to the `<variable>`, which is temporary, and local to the loop[36]. The `TIL <expression>` 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 <label>`. The `WILE <expression>` is the converse: if the expression is WIN, execution continues, otherwise the loop exits[37].

Functions

definition

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

```
HOW DUZ 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:

1. `FOUND YR <expression>` returns the value of the expression.
2. `GTFO` returns with no value (NOOB)[38].
3. 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.

[35] There has also been discussion as to whether `UPPIN` and `NERFIN` should be standard unary functions and not just a special keyword for use with loops.

[36] Presumably it starts with the value 0? There should be some way to modify its start value.

[37] It would be useful to have `TIL` and `WILE` used without `<operation> YR <variable>`—in this case, the loop manages its own iteration variable but ends on a condition—currently, it is an all-or-nothing choice.

[38] Note that because `GTFO` is overloaded from use within loops, returning from a function is not possible within a loop.

calling

A function of given arity is called with:

```
<function name> [<expression1> [<expression2> [<expression3> ...]]]
```

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[39].

[39]Note that in order to parse functions, a lookup table with the number of arguments each function takes is required. It might make sense to employ a tactic similar to `MKAY` to make this operation simpler.