LOLCODE Project Specification

CMSC 124 Final Project
1st Semester 2009-2010
Deadline: October 12, 2009, Monday, 5:30 pm

"Rather fail with honour than succeed by fraud." - Sophocles

I. OBJECTIVES

The goal of your project is to create an interpreter for the esoteric programming language, LOLCODE. Your program should be able to perform the following:

- Accept expressions and function calls from the user.
- Check for syntax and code errors.
- Properly execute LOLCODE expressions and function calls entered by the user.

II. IMPORTANT REMINDERS

A. Groupings

- Project shall be done by groups of 2 or 3, depending on your lab instructor's prerogative.

B. Grading Scheme

- Your project will be graded using the following criteria.

  80% - Actual Project
  20% - Progress Report
- Be reminded, also, part of this 80% is peer evaluation.
- **Non-submission and non-presentation of project will automatically mean 5.0.**

C. Weekly Progress Report

- You are required to submit a weekly progress report to your lab instructor.
- The weekly progress report must include the group's progress with minimum accomplishments for that week.

Week/Date	Minimum Accomplishment Expected
1 / Sept 14-18	Lexical analyzer
2 / Sept 21-25	Parser
3 / Sept 28-Oct 2	Variables
4 / Oct 5-9	Basic arithmetic operations

D. Submission

- **Deadline:** October 12, 2009, Monday, 5:30 pm @ C-114.
- **Requirements:** Place the following in a short brown envelope.
  - A diskette containing your fully documented program code.
  - A hard copy of your program code (stapled) with references.
- Failure to submit after the deadline implies no project.

E. Presentation

- Presentation will be on October 14-16 and 19, 2009.
- A sign-up sheet shall be posted on the ICS Corkboard.
- Failure to appear on your "big day" implies no project.

III. PROGRAMMING LANGUAGE SPECIFICATION

A. Formatting

Whitespace

1. Spaces are used to demarcate tokens in the language, although some keyword constructs may include spaces.
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.
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. A single-line comment is always terminated by a newline. Line continuation (…) and soft-command-breaks (,) after the comment (BTW) are ignored.
8. Line continuation & soft-command-breaks are ignored inside quoted strings. An unterminated string literal (no closing quote) will cause an error.

Comments

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
```

File Creation

All LOLCODE programs must be opened with the command HAI. A LOLCODE file is closed by the keyword KTHXBYE which closes the HAI code-block.

```
HAI
CAN HAS STDIO?
VISIBLE "HAI WORLD!"
KTHXBYE
```

B. Variables

Scope

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
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.

Declaration and Assignment
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
```

Type conversion is handled automatically.

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

Booleans
The two boolean (TROOF) values are WIN (true) and FAIL (false). The empty string (""),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 NUMBR 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 & soft-command-breaks are ignored inside quoted strings. An unterminated string literal (no closing quote) will cause an error.

C. <u>Operators</u>

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 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>
```

D. 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> …][!]
```

To accept input from the user, the keyword is
```
GIMMEH <variable>
```
which takes YARN for input and stores the value in the given variable.

E. 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.

F. <u>Flow Control</u>

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 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 "J00 HAV A CAT"
  MEBBE BOTH SAEM ANIMAL AN "MAUS"
    VISIBLE "NOM NOM NOM. I EATED IT."
OIC
```

Loops

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.

Iteration loops have the form:

```
IM IN YR <label> <operation> YR <variable> [TIL|WILE <expression>]
  <code block>
IM OUTTA YR <label>
```

Where <operation> may be UPPIN (increment by one), NERFIN (decrement by one), or any unary function. That operation/function is applied to the <variable>, which is temporary, and local to the loop. 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.

G. 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).
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.

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.